Unstructured Data Analysis Using LLMs: A Comprehensive Benchmark [Experiments & Analysis]

Qiyan Deng, Jianhui Li, Chengliang Chai, Ye Yuan, Jinqi Liu, Junzhi She, Kaisen Jin, Zhaoze Sun, Yuhao Deng, Jia Yuan[†], Yuping Wang, Guoren Wang, Lei Cao[*†]
University of Arizona[†], MIT[*], Beijing Institute of Technology

Abstract

The explosion of unstructured data has immense analytical value. By leveraging large language models (LLMs) to extract table-like attributes from unstructured data, researchers are building LLM-powered systems that analyze documents as if querying a database. These unstructured data analysis (UDA) systems differ widely in query interfaces, optimization, and operators, making it unclear which works best in which scenario. However, no benchmark currently offers high-quality, large-scale, diverse datasets and rich query workloads to rigorously evaluate them. We present UDA-Bench, a comprehensive UDA benchmark that addresses this need. We curate 6 datasets from different domains and manually construct a relational database view for each using 30 graduate students. These relational databases serve as ground truth to evaluate any UDA system, regardless of its interface. We further design diverse queries over the database schema that evaluate various analytical operators with different selectivities and complexities. Using this benchmark, we conduct an in-depth analysis of key UDA components—query interface, optimization, operator design, and data processing—and run exhaustive experiments to evaluate systems and techniques along these dimensions. Our main contributions are: (1) a comprehensive benchmark for rigorous UDA evaluation, and (2) a deeper understanding of the strengths and limitations of current systems, paving the way for future work in UDA.

1 Introduction

Modern organizations store a large quantity of unstructured data such as clinical notes, legal contracts, financial reports, etc., which account for 80%-90% of global data based on IDC research [13]. These vast repositories of unstructured data, if analyzed appropriately, have immense value in various domains.

As an example, healthcare providers often manage a corpus of hundreds of thousands of heterogeneous medical documents, including disease documents (detailing etiology, symptomatology, and progression patterns), drug documents (detailing indications, mechanisms of action, and contraindications) and documents of medical institutions. If there were a powerful unstructured data analysis (UDA) system, it could enable a provider to easily assist a patient who, for example, has symptoms of frothy urine and dull flank pain, by running queries to identify possible diseases and recommend public hospitals within certain distances of the patient's home that specialize in these diseases.

LLM-powered Unstructured Data Analysis. To support such needs, the database community is actively developing LLM-based systems for UDA [9, 18, 19, 25, 30]. These systems harness LLMs to generate information from multi-modal data lakes (encompassing text, images, etc.) and perform analytical operations such as filtering, aggregation, and join, thanks to the ever growing semantic comprehension and reasoning capabilities of LLMs.

Although these systems use different query interfaces, e.g., Python or SQL queries, they are all declarative systems, which, similar to relational databases, offer a set of logical operators for users to write a simple program to analyze data. These systems provide optimized implementation of these operators to ensure accuracy and reduce LLM cost. Typically, these systems feature a query optimizer that automatically transforms a user program to an optimized query execution plan, with optimizations such as ordering the filters, converting joins to filters, selecting an appropriate LLM for the task, etc. In this way, these systems abstract away time-consuming engineering details in UDA, while transparently optimizing accuracy and addressing the performance and scaling bottleneck of LLMs.

The Need For a Comprehensive Benchmark. Different systems have distinct implementations of operators and query optimization strategies. It is unclear which system works best in which scenario. Furthermore, these systems all use small datasets or query workloads in their experimental evaluation, making the results less convincing. For example, Palimpzest [19] provides a text document dataset with 1,000 short emails, and only one query is available on this dataset. DocETL [30] evaluates extraction tasks on five datasets with hundreds of documents, but each of them defines only one or two extractable attributes, which limits the diversity and complexity of data analysis. Although a recent work, SemBench [16], constructs a benchmark for semantic queries, covering multiple domains and data modalities, on average, it only offers 11 queries per dataset involving about 5 attributes. This is not sufficient to thoroughly evaluate fine-grained cost/accuracy optimization strategies on queries with different complexities.

Therefore, a comprehensive benchmark is still in desperate need to standardize the evaluation of LLM-powered UDA systems and guide future research in this field.

Design Goals. To fill this gap, in this work we propose to construct a benchmark guided by the following design goals:

(1) Dataset Volume. To thoroughly evaluate the efficiency and scalability of different systems, the benchmark has to involve unstructured datasets of various volumes, especially large-scale ones, which are measured from two aspects: the number of unstructured documents and the length of each document. The rationale is that, when dealing with large-scale datasets, there are significant challenges related to both the latency and cost of LLM services.

(2) Dataset & Query Workload Variety. Evaluating the performance of UDA systems must take into account the properties of datasets and the query workloads. Important properties w.r.t. datasets include domains, data modality, whether the documents have a clear

structure, etc. The query workloads should cover different types of analytical operators (e.g., filter, join, aggregation), different number of operators with varying selectivities, and combinations of attributes w.r.t. different data modalities, etc.

(3) Precise labels. A high-quality benchmark must provide precise labels as ground truth. With these labels, the benchmark would enable an accurate evaluation w.r.t. the performance of different systems, regardless of their interfaces or execution strategies.

(4) Easy to evaluate existing systems. To comprehend the advantages/disadvantages of existing UDA systems and expose research opportunities, we have to implement them reliably, perform a thorough evaluation, and conduct a detailed analysis of the results.

Our Proposal. We construct `UDA-Bench` for unstructured data analysis that meets all the above goals.

Key Insight: A Relational Database w.r.t. Multi-modal Data Lake. To develop such a benchmark, our key insight is that constructing a relational database from the corresponding multi-modal data lake is sufficient to cover the evaluation needs of all existing systems, including accuracy, latency, and cost.

Example 1. *Still using the medical application as an example, given a category of files, e.g., disease documents, we can define a relational schema (i.e., a number of meaningful attributes such as disease name, etiology, symptoms, etc.) in advance. Then we extract the values of all attributes from the data lake as the ground truth. In this way, we obtain a relational database consisting of multiple tables, each corresponding to a specific category of files (i.e., disease, drug, medical institution, etc.). Consequently, any analytical queries concerning the aforementioned attributes can be evaluated based on corresponding data records in the constructed database, agnostic to their query semantics and execution strategies. For example, although the semantic operators in Lotus prefer natural language input, it can still compose queries about the attributes covered by the database to evaluate their implementation.*

This insight guides us to construct `UDA-Bench` that meets all the above goals.

To achieve the dataset volume goal, we collect **six** sets of unstructured documents, as shown in Table 2. In terms of the number of documents, all the datasets have at least hundreds of documents and, in particular, `Healthcare` has 100,000 long and complex documents, which is **100**× more than the existing benchmarks. In terms of the lengths of the documents, three of the datasets have an average length of more than 10,000 tokens. In particular, on `Finance` dataset, the length of the documents can be up to 838,418 tokens (≈100 pages). Effectively processing a large number of long documents that potentially incorporate much noisy information is prohibitively expensive and difficult. This poses challenges for document chunking, chunk retrieval and filtering, etc, which are critical to the performance of UDA systems, thus desiring a benchmark to evaluate thoroughly.

For the second design goal of ensuring the diversity of the datasets and query workloads, we construct datasets from **six** representative domains, including healthcare, law, art, sports, science, and finance. Among them, art and science include images and text, while science additionally includes tables. We construct a total of **608** queries, **10**× more than the current benchmarks. These queries are grouped into **five** main categories: `Select`, `Filter`, `Join`, `Agg`, and `Mixed` (other complex queries that are combinations of at least 3 operator types). These queries differ in multiple aspects, including different number of filters, various selectivities, binary or multiple joins, w/ or w/o aggregation operations, etc., capable of comprehensively assessing the capabilities of existing systems.

For the third goal of precise labels, we define **167** meaningful attributes derived from the six datasets, including categorical, numerical, and string types, exceeding the existing benchmarks by **5**×. These attributes cover a wide range of document types (from short passages to very long documents), modalities (text, tables, and images), and difficulty levels (from directly extractable attributes to those requiring multi-step reasoning), thereby providing a comprehensive and realistic evaluation environment. A team of 30 graduate students performs extensive manual annotation, spending in total 4,110 hours, approximately 0.5 hours per document, with the assistance of LLMs in cross-validation for quality assurance.

To achieve the last goal of easy evaluation,we have conducted extensive experiments to evaluate these systems on all datasets, measuring accuracy, cost, and latency across fine-grained query categories to capture differences in system performance. Furthermore, we analyze the experiment results from four perspectives: query interface, query optimization, operator design, and data processing, which are the key building blocks of such systems. This in depth, multi-faceted analysis offer actionable insights for future research.

Contributions. We make the following contributions:

(1) We present `UDA-Bench`, a benchmark for unstructured data analysis that includes large-scale and diverse datasets, as well as a rich set of queries; to the best of our knowledge, it is the first work that constructs a comprehensive benchmark and thoroughly evaluates existing LLM-powered UDA systems.

(2) We collect six sets of unstructured data from diverse domains, including more than 100,000 documents. We construct 608 queries that involve various analytical operators over the diverse attributes defined in the database.

(3) We provide comprehensive and cross-validated relational tables as ground truth, annotated by 30 graduate students over 4,110 hours, enabling objective and reproducible evaluation on UDA systems.

(4) We thoroughly evaluate seven representative UDA systems in accuracy, cost, and latency. Our in-depth analysis offers insights to guide future research in this field.

2 Unstructured Data Analysis Systems

We present the architecture of a typical LLM-powered UDA system, which typically consists of 4 key modules: *the query interface*, *logical optimization layer*, *physical optimization layer*, and *data processing layer*. As shown in Fig. 1, a user submits a query via the interface, which describes the analytical task. The system parses the query into some analytical operators, such as `Extract`, `Filter`, `Join`, and `Aggregation`. Then, the logical optimization layer determines a logical query plan, such as pushing down predicates or ordering the operators. Then the physical optimization layer selects the appropriate physical implement w.r.t. each operator in the logical plan, producing an optimized physical execution plan. The data processing layer optimizes the data layout, supporting the aforementioned optimizations. In addition to storing the raw data in distributed storage systems (e.g., Amazon S3), it often segments documents into chunks which are subsequently transformed

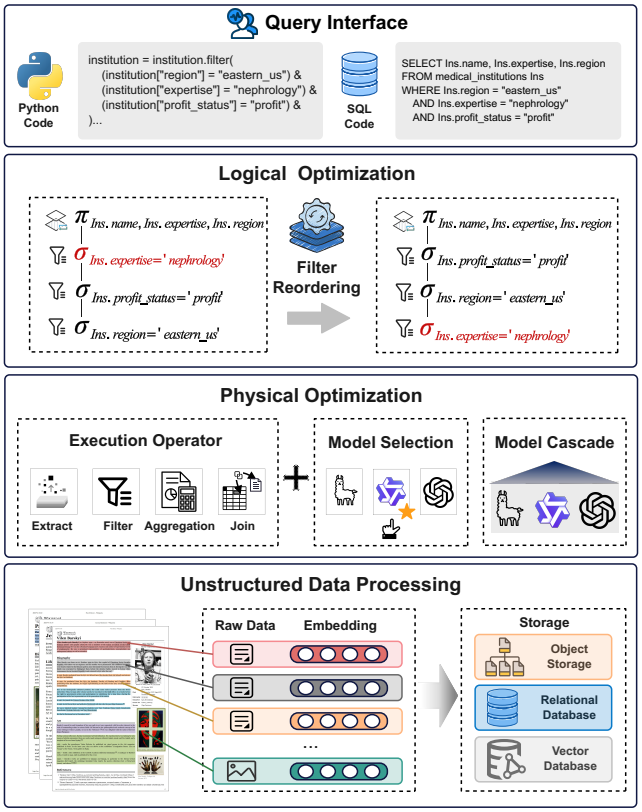

Figure 1: Architecture of Unstructured Data Analysis System.

into embeddings and loaded into a vector database. This enables accurate and efficient retrieval to identify information relevant to a query, speeding up query execution and reducing cost.

Optimization Goals. Unlike traditional databases that focus mainly on optimizing latency, LLM-powered UDA systems have multiple optimization goals, i.e., *accuracy*, *cost*, and *latency*. Accuracy is critical in such systems for two reasons. (1) LLMs are prone to hallucinations, leading to potential inference errors; and (2) to reduce costs, a common practice is to only feed the LLM document chunks highly relevant to the query, rather than entire documents. However, if the retrieval misses relevant chunks, it will degrade the analysis accuracy. On the other hand, the LLM cost and inference latency largely rely on the number of input and output tokens. Reducing them is typically achieved by minimizing the number of input tokens. However, the system still has to ensure that the LLM gets sufficient information to ensure the quality of the analysis.

We detail how existing systems use four modules to achieve the stated goals. Table 1 shows the modules of each system.

System	Data Processing			Operator				Query Optimization	
	Chunking	Embedding	Multi-modal	Extract	Filter	Join	Agg	Logical	Physical
Evaporate	✗	✗	✗	✓	✗	✗	✗	✗	✗
Palimpzest	✗	✗	✓	✓	✓	✓	✓	✓	✓
LOTUS	✗	✓	✓	✓	✓	✓	✓	✗	✓
DocETL	✓	✓	✗	✓	✓	✓	✓	✓	✓
ZenDB	✓	✗	✗	✓	✓	✓	✗	✓	✗
QUEST	✓	✓	✗	✓	✓	✓	✗	✓	✗
UQE	✗	✓	✓	✓	✓	✗	✓	✓	✗

Table 1: Overview of Existing UDA Systems.

2.1 Data Processing

The data processing component handles a wide variety of data types, including plain text, images, etc., which are extracted from complex documents such as PDF by OCR tools. Then, it organizes these contents into different formats, which are further processed by different strategies to support downstream analytical tasks.

LOTUS [25], UQE [9] and DocETL [30] typically organize plain text and images into semi-structured files (CSV for LOTUS and UQE, JSON for DocETL), where each entry corresponds to a document. For CSV, plain text is stored in the text column and the paths of the image files are recorded in the image column. For JSON, each document is represented as an object with text and image fields. Subsequently, these systems transform text into embeddings for later semantic analysis, adding the storage path of these embeddings as an additional column in the CSV file. UQE, which supports images, stores the embedding of images as another additional column to support the aggregation operator.

ZenDB [18] and QUEST [31] target analyzing plain text in documents, which is stored in a relational database. Both builds index to speed up retrieval. ZenDB leverages the visual features of PDF such as font size, boldness, and positioning to identify hierarchical section titles and divides the document into semantic units, forming a Semantic Hierarchical Tree (SHT) that reflects the structure of the document. Then, it summarizes the content under each node using the NLTK toolkit [20] and stores the summarization in the node. It calculates the embedding of each sentence to reduce the cost of attribute extraction.

QUEST, on the other hand, constructs two levels of indexes to support accurate and cost-effective attribute extraction. It first generates a summary for each document and encodes it into an embedding to build a vector-based, document-level index. This index allows the system to quickly exclude documents irrelevant to the query. Then, it splits each document into semantically coherent chunks, encodes these chunks into embeddings, and constructs a segment-level vector index, which is used to identify chunks relevant to a to-be-extracted attribute. This avoids feeding the entire document to LLMs to save cost.

Evaporate [3] targets plain text stored in a folder for subsequent analysis. Palimpzest [19] organizes the plain text and images in each document into a directory. Both do not support indexing.

2.2 Query Interface & Operators

Query Interface. Each system provides a query interface for users to define analytical tasks. UQE, ZenDB, and QUEST use SQL-like language to support analysis over unstructured data. For example, in the Healthcare dataset, to find private institutions specializing in nephrology and located in the eastern United States, a user can write a query shown in Figure 1. DocETL, Evaporate, LOTUS and Palimpzest offer declarative Python APIs, corresponding to logical operators in relational databases, for users to compose a query as a Python program, as shown in Fig. 1.

Operators. UDA-Bench supports 4 common analytical operators. Extract generates attributes from a set of documents D, formally defined as $\texttt{Extract}(D, A)$, where A specifies the attributes to be extracted from D and the output is a relational table T_D. For example, to extract the rating and locations of institutions, this operation can

be expressed as `Extract (Healthcare, [rating, location])`. In an SQL-like interface, this corresponds to `SELECT rating, location FROM Healthcare`", while in the Python API, it could be written as `pz.addcolumns(Healthcare, [rating, location])`". In addition, users can provide attribute descriptions as prompts, helping LLMs produce accurate answers. `Evaporate` employs LLMs to generate code to extract each attribute. Other systems feed the attribute (with optional user descriptions) and relevant document chunks (possibly the entire document) to LLMs for extraction.

`Filter.` Given a condition C, `Filter` operation selects a subset of documents that satisfy C from D, denoted by $\texttt{Filter}(D, C)$. A filter on documents D evaluates whether the relevant attributes of each document satisfy condition C. For example, "`lotus.sem_filter('the {document} satisfy {profit_status} is profit')`" is a filter in `LOTUS`. Existing systems adopt two strategies to implement such filters: (1) `Palimpzest`, `QUEST`, and `ZenDB` extract `profit_status` from each document and then evaluate whether it satisfies the filter; and (2) `LOTUS`, `DocETL` and `UQE` take the filter as part of prompts and leverage LLMs to determine whether the filter condition is satisfied.

`Join` is a cross-document operation, i.e., combining information from two sets of documents based on a join condition a, formally defined as $\texttt{Join}(D_1, D_2, a)$. D_1 and D_2 are two subsets of documents, and a is the join attribute. If the system extracts two tables (both contain the attribute a), the tables can be joined on a. For example, from the document subset of `disease`, the system could extract a table describing the disease and another table with drug attributes from the `medication` subset; and the two tables can be joined by the disease name. Through a join operation, a user can easily identify possible diseases based on symptoms and then find medications that can treat those diseases. `ZenDB` and `QUEST` implement the `Join` by first extracting the disease table and the medication table, respectively, from two sets of documents and joining the two tables. On the other hand, `LOTUS` first extracts the disease table and embeds the values of the join key attribute. It then uses the embedding of each disease to retrieve relevant medication documents. From this subset of documents, it extracts the medication table. Finally, the two tables are joined to produce the join result.

`Aggregation` is defined as $\texttt{Agg}(D, a, F)$, where a specifies the grouping attribute and F defines the aggregation functions to apply, such as `Count`, `Sum`, `Avg`, `Min` or `Max`. This operator supports analytical tasks like "computing the number of institutions grouped by expertise", i.e., `Agg (Healthcare, 'expertise', Count)`, which can be represented as "`pz.GroupBySig(Healthcare, Count,'expertise')`" using the Python API of `PALIMPZEST`. In implementation, `Evaporate`, `ZenDB`, `QUEST` and `Palimpzest` extract the `'expertise'` from all documents, group the values in a table, and then perform aggregation on the grouped data. `LOTUS`, `UQE`, and `DocETL`, on the other hand, first preprocess the documents by clustering them based on their embeddings, and then perform aggregation or batch inference within each cluster as an approximation to save costs. Besides, `UQE` supports grouping images according to their embeddings.

2.3 Logical Optimization

Given a user-specified query involving multiple operators, UDA systems generate an optimized logical plan to reduce LLMs costs and query latency. The optimizations adopted by existing systems mainly include filter reordering, filter pushdown, and join ordering.

Filter Reordering. Consider a query that selects artists and their birthdates with two filters, i.e., F_1 =`filter(D, "lifespan is less than 35")` and F_2 =`filter(D, "tone is warm")`). Different systems employ different optimization strategies for reordering filters to reduce costs.

(1) Selectivity-only Strategy. `Palimpzest` and `UQE` adopt a selectivity-only filter reordering strategy that prioritizes a filter with low selectivity. This indicates that the attribute values that a document contains have a small probability to satisfy the predicates of this filter. This reduces the chance of extracting other attributes and evaluating the corresponding filters. These systems estimate the selectivity (denoted by $sel()$) by sampling. For example, if $sel(F_1) = 0.2$ and $sel(F_2) = 0.1$, applying F_2 first would leave ≈10% documents for F_1 to evaluate, potentially reducing the cost. However, only considering the selectivities tends to be suboptimal in minimizing the cost. For example, although $sel(F_2) < sel(F_1)$, if the document chunks that F_2 has to examine contain much more tokens than those of F_1, using this order might lead to higher costs than applying F_1 first.

(2) Selectivity-cost strategy. To address the above limitation, `ZenDB` ranks filters based on scores computed by $sel(F) \times cost(F)$ and prioritizes those with lower scores. Given the attribute a of a filter F and an unstructured document $d \in D$, it uses $d[a]$ to denote the chunk(s) that are highly relevant to a. The relevance can be computed by measuring the similarity between the embeddings of chunks and the attribute. Let $c(d[a])$ denote the number of tokens of $d[a]$. Then in `ZenDB`, $cost(F)$ corresponds to the average number of tokens of chunks relevant to a across all documents, i.e., $cost(F) = \frac{\sum_{d \in D} c(d[a])}{|D|}$. Therefore, it produces one single filter order w.r.t. all documents, similar to traditional databases. This is suboptimal, as different documents might contain different numbers of tokens w.r.t. an attribute.

`QUEST` instead produces different orders for different documents considering both their selectivities and costs, Therefore, `QUEST` does not compute $cost(F)$ w.r.t. the whole document set D. Instead, for each document $d \in D$, it uses $cost_d(F) = c(d[a])$ to denote the number of tokens w.r.t. the attribute a in d. Then, it assigns d a specific optimal order which prioritizes the filter with lower values of $sel(F) \times cost_d(F)$.

Filter Pushdown. Given a query that joins two sets of documents with filters applied on them, the most straightforward way (`ZenDB`) is to first pushdown the filters to the two document sets respectively, extract the join key attribute, and then join. Traditional databases adopt this strategy because filters typically have a lower time complexity than joins and thus should be prioritized. However, in this unstructured data analysis scenario where LLM cost is the primary optimization goal, a join may not be more costly than a filter and potentially could have a higher priority. Inspired by this insight, `QUEST` proposes a join transformation strategy that first extracts the join attribute of one table and then uses the extracted values as filters to filter the other table, i.e., transforming a join into a filter. Treating this automatically generated filter equally to other filters, `QUEST` uses the cost model discussed above to order these filters. Therefore, `QUEST` might prioritize joins over filters to minimize the LLM cost, contradictory to filter pushdown in relational databases.

Join Ordering. For multi-join, `ZenDB` and `QUEST` dynamically and progressively decide the join order during query execution. More

specifically, they first selects two tables to join based on their cost model, and it will determine the next join only after the first join finishes execution. This process iterates in a left-deep manner until all joins have been executed.

2.4 Physical Optimization

Next, the systems select an appropriate implementation w.r.t. each operator in the logical plan based on the properties of the data and user preferences, i.e., physical optimization. We introduce the typical physical optimizations in UDA below.

Model Selection. For each logical operator, the optimizer selects the most suitable model from a set of candidate based on users' preference, e.g., accuracy or cost. In `Palimpzest`, if a user wants to achieve target accuracy while at a relatively low cost, the system prefers a lightweight model for simple extraction tasks to reduce cost, e.g., using `GPT-4.1-mini` to extract "`birthdate`" from the `Art` dataset. Conversely, for complex semantic analysis where `Llama` fails to meet the target accuracy, it employs more advanced models, such as using `GPT-4.1` to infer whether a case is a first-instance trial in the `Legal` dataset.

Model Cascade. In addition to selecting one model that is the most suitable for the operator, the optimizer in `LOTUS` applies the model cascade technique to save more cost while meeting the users' target accuracy. More specifically, it uses a sequence of different models to execute an operator. These models have various characteristics, such as diverse qualities, varying costs, and different levels of latency. For example, the model cascade can be {`GPT-4.1-nano`, `GPT-4.1-mini`, `GPT-4.1`}, which begins with the cheapest `GPT-4.1-nano`. `LOTUS` immediately returns the result if the `GPT-4.1-nano` output meets the target accuracy. If not, the input proceeds to the next model in the cascade, i.e., `GPT-4.1-mini`, until obtaining a satisfactory output or reaching the last model.

Parallel Execution. Leveraging efficient batched inference with vLLM [15], `LOTUS` and `DocETL` process multiple documents concurrently, allowing efficient operator execution over large-scale document collections.

Dataset	#-Domain	#-Attributes	#-Files	Tokens (Max/Min/Avg.)	Multi-modal
Art	1	19	1,000	1,665 / 619 / 789	✓
CSPaper	1	20	200	107,710 / 5,325 / 29,951	✓
Player	4	28	225	51,378/73/8,047	✗
Legal	1	19	566	45,437 / 340 / 5,609	✗
Finance	1	30	100	838,418 / 7,162 / 130,633	✗
Healthcare	3	51	100,000	63,234 / 2,759 / 10,649	✗

Table 2: Statistics of datasets.

3 The Benchmark Construction

In this section, we first overview the construction process of `UDA-Bench` and then introduce each step in detail.

3.1 Overview

`UDA-Bench` consists of 6 datasets: `Art`, `CSPaper`, `Player`, `Legal`, `Healthcare` and `Finance`. We first collected and pre-processed the raw data. Next, we followed a semi-automatic, iterative process to identify the attributes of the relational tables. We then extract the values of these attributes from these datasets, i.e., labeling ground truth, combining iterative LLM prompting and human labeling. Finally, we produce a comprehensive set of queries, which can be divided into 5 major categories and 42 sub-categories.

For each dataset, we provide benchmark results in a JSON file, where all documents are decomposed into separate modalities (text, tables, and images). Each document maps to a field in the JSON file, and the different modalities are stored in distinct entries. In this way, if a user wants to test her system using `UDA-Bench`, she can directly download the processed data, load the data into the system, run her queries, and compare the results with the ground truth table stored in the relational databases.

3.2 Datasets

`UDA-Bench` consists of six datasets with their statistics summarized in Table 2. Next, we describe these datasets below. For more details, refer to our website.

`Art` is collected from WikiArt.org [4], which covers artists and their artworks spanning from the 19th to the 21st centuries. Each document corresponds to an artist including biographical information, artistic movement, a list of representative works, and an image of a representative work.

`CSPaper` dataset is crawled from Arxiv[1] containing 200 research papers annotated with key attributes, including authors, baselines and their performance, the modalities of experimental datasets etc. In particular, some papers describe the performance of all baselines in the main text, while other papers only describe the best-performing baselines and leave other results in tables or figures, resulting in an analysis scenario with mixed-modal.

`Player` dataset is crawled from Wikipedia[22] that contains information about basketball including players, teams, team managers, etc., from the 20th century to the present, covering their basic and statistic information.

`Legal` is sourced from AustLII [5] with 570 professional legal cases from Australia between 2006 and 2009, covering different types such as criminal and administrative. Each case document typically includes evidence, charges, legal fee, etc.

`Finance` are collected from the Enterprise RAG Challenge [11], containing annual and quarterly financial reports published in 2022 by 100 listed companies worldwide with an average token length of 130,633. Each record typically includes mixed types of content like company name, net profit, total assets, etc.

`Healthcare` is obtained from MMedC [26], with healthcare documents since 2020. This dataset contains a massive amount of files (100,000), each file having 10,649 tokens on average. It covers various types of healthcare information, like drugs, diseases, medical institutions, news, interviews crawled from large-scale web corpora and open-access healthcare websites.

Summarization. These datasets show various characteristics. Compared to other datasets, the `Art` dataset is less complex due to its short documents and well-defined structure, but each document contains a corresponding image. `CSPaper` is a moderately complex academic dataset composed of research papers that contain text, tables, and images. `Legal` is more complex because it is a domain-specific dataset containing multiple attributes that require semantic reasoning to extract. `Finance` is another complex domain-specific dataset. The key challenge it introduces is the length of the documents – up to 100 pages. Lastly, `Healthcare` has the largest number of documents, containing rich information. `Healthcare` and `Player` contain multiple categories of files, e.g., disease, drug, medical institution, suitable for evaluating join queries.

Note that previous works [3, 7] also use the sources of `Player`, `Art`, and `Legal` in their experiments. We largely augment them

by adding new documents, attributes and comprehensive query workloads, addressing the issue that the original datasets were curated primarily for extraction and offered limited query coverage.

Data processing. We design a unified data preprocessing pipeline to handle datasets with various features. For each dataset, we first collected the raw data and utilized the MinerU [35] toolkit to parse the data when dealing with complex formats such as PDF (e.g., the `Finance` dataset). Then, we organized the dataset into a JSON file, where each object corresponds to an unstructured document with multiple fields including text, image URL, and metadata. In particular, for the `Healthcare` and `Player` datasets, we divided the documents into multiple categories with different topics, each of which corresponds to a relational table. For `Healthcare`, it originally has 680,000 documents on healthcare information, from which we sampled 100,000 ones. Then we used LLMs to read these samples and identified three major categories of files (6,100 documents in total), i.e., disease, drug, and medical institution. Note that we only define and extract attributes over the three major categories. Other documents are retained to create a realistic large-corpus setting, where only a subset of documents (the above three categories) is relevant to a given query. This allows our benchmark to evaluate the retrieval cability of UDA systems. For `Player`, we identify 4 subsets of documents belonging to different categories, i.e., players, teams, team managers, cities, amenable to evaluate join queries.

3.3 Ground Truth Labeling

To label the ground truth, we first identify a number of significant attributes from each dataset and then manually extract their values.

Attributes Identification. We hire 6 Ph.D. students from different majors (e.g., finance, law, medical) in our university to read these documents carefully and identify attributes that pose different levels of challenges to extract. For example, the attribute `Judge_name` is easily identifiable, since it is usually located at the beginning of each document in `Legal`. In `Finance`, `Business_cost` is the sum of several cost components, which vary across industries—for example, raw materials and wages for car manufacturers versus product sourcing and logistics for supermarkets. A labeler must identify the relevant components of `Business_cost` from the document context, extract their values, and aggregate them into the final `Business_cost`. In addition, image files also contain easy to extract attributes like `Tone`, whose answers often fall into "Neutral", "Bright" and "Dark" that can be easily categorized. Difficult attributes, such as "Style", require art expertise to reliably extract.

Labeling. To ensure high-quality ground truth for `UDA-Bench`, we hire a total of 30 graduate students to manually label these attributes, spending approximately 4,110 human hours To be specific, for each document, human labeler relies on LLMs to identify the content relevant to each attribute, and then manually extract the corresponding attribute values as ground truth from the retrieved content. Note that the labeler uses LLMs as an assistant. If the labeler cannot extract the attributes from the retrieved content, she will manually read the document and provide the answer. Importantly, during extraction, we also ask humans to record the chunks where each attribute is extracted so that users can verify the source of every labeled value. Moreover, to further ensure labeling quality, we use LLM-as-a-judge to verify the accuracy of the human-labeled

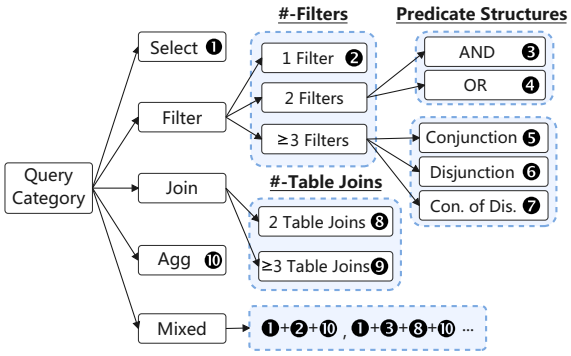

Figure 2: Query Categories.

results. Whenever the LLM flags an extracted attribute as incorrect, we ask humans to double check the corresponding ground truth.

However, for the large-scale dataset `Healthcare`, it is impractical to manually label all the ground truth. Therefore, we adopted a semi-automated iterative labeling strategy. Recap that `Healthcare` contains a large number of documents that belong to categories other than the three major ones mentioned above. Therefore, these documents rarely include entities in the major categories. Consequently, we asked LLMs to analyze 100,000 documents to label whether each of them belongs to the three categories. If so, we ask humans to label attributes; otherwise, the ground truth is `NULL`.

3.4 Query Construction

As shown in Figure 2, we divide the queries into five main categories based on the primary query operators, as different operators introduce different technical challenges: (1) `Select`, involving only `Extract` operations; (2) `Filter`, combining `Extract` with `Filter` operations; (3) `Join`, involving `Extract` and `Join` operations; (4) `Agg`, incorporating `Extract` with `Aggregation` operations; and (5) `Mixed`, including several categories of operators listed above. Note that every query includes an `Extract` step to return the attributes to be analyzed. We then further decompose each main category into subcategories to construct a broad range of queries that differ in modality, filter complexity in terms of the number of filters, predicate structures (the combination of conjunction or disjunction), selectivities, single or multiple joins, w/ or w/o aggregation, etc. These queries are in the form of both SQL-like queries and Python code, thus able to test systems with different interfaces, like `ZenDB` and `Palimpzest`. An example is shown below.

```
SELECT {Attribute}(s),
    {agg_func}({Attribute}(s))
FROM diseases
JOIN drug ON disease.name = drug.disease
WHERE {Attribute}{operator}{literal}
GROUP BY {group_by};
```

A SQL subcategory Example.

(1) *For the* `Select` *category*, for example, a query such as "SELECT {Attribute}(s)" represents a `Select` operation used to extract attributes. Here, "{Text Attribute}(s)" serves as a placeholder that can be instantiated with any number of available attributes with different modalities, which are randomly selected to generate concrete

```
disease_doc = disease
drug_doc = drug
disease_doc = Filter(disease_doc,
    {Attribute}{operator}{literal})
disease_drug = Join(disease_doc, drug_doc,
    disease_doc.name = drug_doc.disease)
result = Extract(disease_drug,
    {Attribute}(s))
result = Aggregate(result, {group_by},
    {agg_func}({Attribute}(s))
```

A Code subcategory Example.

queries. These Select queries can clearly evaluate the system's capability of extracting attributes with different modalities.

(2) *For the* Filter *category*, we further classify it by the number of filters (one filter, two filters, and more than two filters) and the way of combining the filters (conjunction only, disjunction only, and conjunction of disjunctions), resulting in a total of 6 Filter sub-categories. With varying complexities, these sub-categories of queries offer a large space for cost optimization (e.g., more filters bring a large optimization opportunity for filter reordering). As shown in the example, the pattern "{Attribute}{Operator}{Literal}" serves as the placeholder format for filter predicates. During query generation, we randomly sample both the filter attribute and the comparison operator (such as $\leq$, $=$, or $\geq$) with equal probability, and we further sample literal values to produce varying selectivities.

(3) *For the* Join *category*, to thoroughly evaluate the capacities of different systems in optimizing join queries, we curate two sub-categories of join queries, i.e., binary joins and multi-table joins, where more tables incur higher computation cost. During query generation, we explicitly define a join graph w.r.t. each dataset. For example, in Healthcare, valid join paths include Disease $\bowtie$ Drug, along with their corresponding join keys (e.g., Disease.name = Drug.disease). Similarly, in Player, we have Players $\bowtie$ Teams $\bowtie$ Managers, together with join keys that link these tables.

(4) *For the* Agg *category*, during query generation, we randomly sample the group-by attribute, the aggregated attributes, and the aggregation functions (such as Count) to populate the corresponding placeholders (i.e., {agg_func}({Attribute}(s)).

(5) *For the* Mixed *category*, it combines multiple types of operators in one query, e.g., first joining two tables and then run aggregation on the join results. It includes 32 subcategories, each of which is combined with at least three of the aforementioned main categories. The above example illustrates one such combination, i.e., selecting attributes with one filter, an aggregation and a 2-table join.

We then apply these subcategories to each dataset to construct queries. After generating a large pool of candidate queries, we ask human experts to select those that are both meaningful and representative. Specifically, for the first four main categories, we retain 10 queries per subcategory in each dataset. Since the Mixed category aims to evaluate the performance of combining different operators, a single query per template per dataset is sufficient. Details of the queries w.r.t. each dataset can be found in the website.[1] Moreover, we have open-sourced the query generation code so that users can create queries on their own.

[1] https://db-121143.github.io/uda-bench-page/

4 Evaluation

We evaluate existing systems on UDA-Bench to answer the questions:
- **RQ 1**: What is the accuracy of different systems on the benchmark?
- **RQ 2**: What is the cost of different systems on the benchmark?
- **RQ 3**: How efficient are different systems on the benchmark?
- **RQ 4**: How do different logical optimization strategies perform?
- **RQ 5**: How do different physical optimization strategies perform?

4.1 Experimental Settings

Systems for Evaluation. Our benchmark evaluates 7 existing UDA systems. (1) Evaporate is a table extraction system, so we employ Evaporate to extract structured tables from documents and subsequently execute SQL queries on the resulting tables. (2) Palimpzest provides Python API-based operators for unstructured data processing. We convert each SQL query into the corresponding Palimpzest code, execute it and obtain the results. (3) LOTUS also provides an open-source Python library, which we use to execute queries through its interface. (4) DocETL is an open-source project using Python code. We rewrite our queries with DocETL library and execute the Python programs. (5) QUEST provides SQL-like query interface for processing unstructured documents, and we directly use their code to execute queries. (6) ZenDB does not provide their code; therefore, we implement their SHT chunking and filter reordering strategies and evaluate them on UDA-Bench. (7) UQE does not release its code, so we reimplement its Filter and Agg operators, along with its logical optimizations, and run them on UDA-Bench. For a comprehensive evaluation, we adapted and modified these systems to support our evaluation. We also list all other details (e.g., evaluation prompts) in Appendix A.

Evaluation Metrics. Following existing works [18, 31], we measure accuracy, cost, and latency w.r.t. all queries. For accuracy, we follow [31] and report the average precision, recall, and F1-score of queries. Given a query Q, the set of tuples returned by a method is $T(Q)$, and the ground truth is $GT(Q)$. For each element $t \in T(Q)$, we evaluate whether it can be matched to a corresponding tuple in $GT(Q)$. Specifically, many extracted attributes often appear in varying surface forms. Exact string matching would distort the accuracy. Therefore, we apply entity matching to align lexical variants referring to the same underlying entity. For LLM costs, we report the average number of tokens (in thousands) per document per query by using the LLM API, which counts the token consumption w.r.t. all data modalities, e.g., text and images. For latency, we report the mean execution time in seconds per document per query.

Environment. We implemented all experiments in Python 3.7 on an Ubuntu Server with four Intel(R) Xeon(R) Gold 6148 2.40GHz CPUs with 80 cores, 2 NVIDIA 4090 GPUs, 1TB main memory and 6TB SSD. The same environment ensures a fair comparison over systems. We adopt GPT-4.1-mini as the default LLMs for API calls, and Qwen3-Embedding-0.6B as the default embedding model.

4.2 Overall Accuracy Comparison (RQ1)

Figure 3 presents the effectiveness results for Select queries. Evaporate performs the worst because it uses LLM-generated code for extraction. This code corresponds to a small set of rules, which are less effective for complex documents.Almost all systems perform well on dataset Player, with an F1 score around 0.85 because

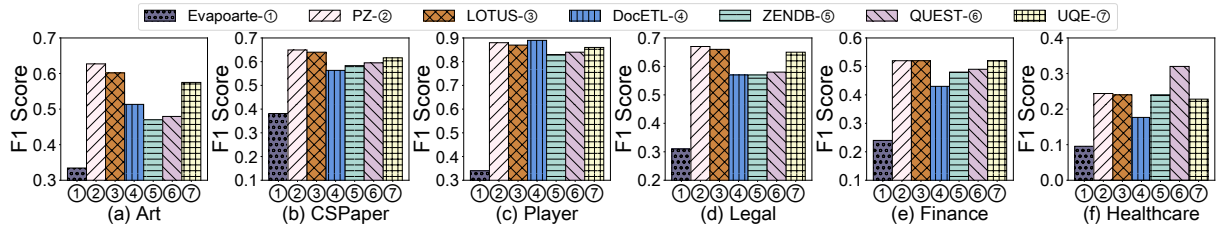

Figure 3: F1-score Comparison of `Select` Queries.

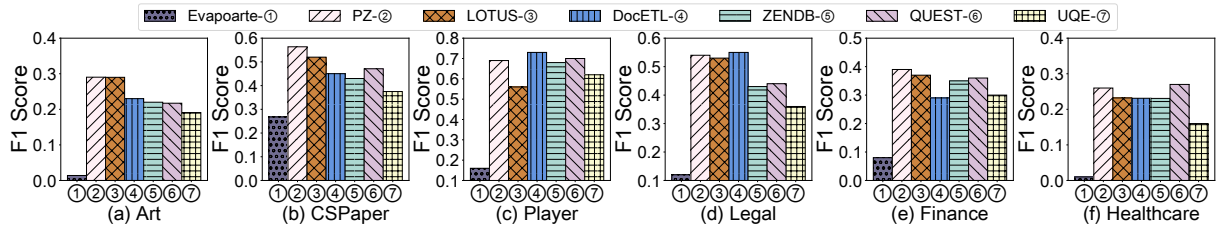

Figure 4: F1-score Comparison of `Filter` Queries.

the dataset contains easy-to-extract attributes. DocETL performs the best because it leverages multi-agent techniques for extraction.

On datasets Legal, Finance, and Healthcare, with long documents and multiple challenging attributes, the accuracy varies across different systems. Specifically, LOTUS, UQE, and Palimpzest achieve similar performance and are top-ranked on the first three datasets, with an accuracy around 0.66 on Legal, and 0.52 on Finance, as they feed each entire document to LLMs and fully leverage the models' in-context reasoning abilities. However, in Healthcare, a lot of irrelevant and noisy information can mislead LLMs. Besides, DocETL always selects the plan that splits documents into chunks, feeds each one to the LLM for extraction, produces multiple candidate answers per attribute and finally aggregates them. It is less effective on Healthcare and Finance because long documents introduce many noisy chunks, leading to incorrect extractions. For most datasets, QUEST and ZenDB perform worse because they only provide relevant chunks to LLMs to save cost. However, their performance is different on Healthcare, which lacks structured information. In this case, ZenDB feeds the entire document to LLMs, resulting in performance similar to that of LOTUS, UQE, and Palimpzest. However, since QUEST only retrieves and feeds LLMs the relevant chunks, it filters out much noisy information and achieves higher accuracy.

Summary I: For simple datasets with short content and easy-to-extract attributes, almost all systems perform well. For most datasets, the systems that feed entire documents to LLMs perform the best because retrieval-based strategies may miss in-context information. However, for datasets containing significant noise, systems supporting retrieval-then-extraction performance better due to filtering irrelevant and noisy context.

Overall, from the data perspective, the performance of all systems drop significantly on Finance and Healthcare, with F1-scores below 0.5 because : (1) noisy information: in Healthcare, individual documents frequently combine content from unrelated topics. For example, a single medical report might describe hypertension management in the first half and then switch to dermatitis treatment in the second. (2) Long context length: Finance documents are extremely long, making it difficult for an LLM to locate segments relevant to the target attribute within such massive texts. (3) Complex attributes: Certain attributes require multi-step reasoning rather than direct extraction. For example, the attribute Business_cost in Finance is not explicitly stated but must be calculated by extracting and aggregating various cost components (e.g., raw materials cost and wages for car manufacturers). Existing systems, which most rely on a single LLM pass, fail to extract such attributes effectively.

Opportunity I: Extraction is the fundamental operator in UDA systems, but existing systems fall short in extracting complex attributes from complex and noisy datasets. This underscores the need for finer-grained retrieval, decomposition-based or multi-round extraction in the future.

On the Art and CSPaper datasets (Fig. 3), which heavily rely on visual information, LOTUS, UQE, and Palimpzest demonstrate superior performance due to their native support for multi-modal inputs. However, the performance variance among them is marginal, as all three systems rely solely on multi-modal LLMs to handle this information. No additional optimizations are conducted, such as cross-modal validation to verify consistency and joint reasoning to extract attributes from diverse modalities. Furthermore, existing systems require users to manually specify the modality of source data (text vs. image) for each attribute, instead of automatically mapping target attributes to the appropriate source data.

Opportunity II: This highlights the opportunity to develop a unified mechanism capable of automatically inferring attribute modalities, guiding them to suitable models, and leveraging cross-modal retrieval for more accurate information extraction.

Figure 4 shows the effectiveness of Filter queries. Existing systems implement filters in two ways. Given a filter, LOTUS and DocETL feed the description of the filter together with the document into an LLM and ask it to directly output a boolean output. Other systems first extract the attribute w.r.t. the filter and then determine whether the extracted value satisfies the filter. We observe that LOTUS performs worse than Palimpzest, ZenDB, QUEST on Player, as extracting the relevant information before evaluating the filter

leads to more accurate answers. DocETL still achieves the best performance due to its multi-agent strategies. On complex datasets, LOTUS outperforms many systems because it feeds the entire document into LLMs, whereas the performance of DocETL declines due to the imperfect chunking strategy. UQE performs worse than other systems because it samples a subset of documents to train a regression model, which is then used to predict whether the remaining documents satisfy the filter.

Summary II: For the filter operation, extracting the attribute first and evaluating the filter thereafter lead to more accurate results than directly executing it with LLMs. This is because decomposing the filter into the above two steps provides LLMs with more explicit instructions.

4.3 Overall Cost Comparison (RQ2)

Table 3 shows the cost of Select queries. On the dataset Art with short documents, systems that use chunking strategies (i.e., QUEST, ZenDB, and DocETL) consume more tokens than simply processing entire documents (UQE). The reason is that for each attribute, QUEST and ZenDB select attribute-related chunks. Given multiple attributes, there tends to be a number of duplicated chunks especially when the documents are short. As a result, the total token consumption exceeds that of simply processing full documents. The same pattern shows on images, which are more expensive to process than text documents. DocETL incurs the highest cost because (1) it examines all the chunks. When precessing each chunk, it also feeds adjacent chunks into LLMs; and (2) require executing all possible plans on sampled documents to select the optimal one. Palimpzest and LOTUS also process entire documents, but the chain-of-thought mechanism in Palimpzest leads to more output tokens, while LOTUS processes each attribute separately, multiplying the cost by the number of attributes.

Summary III: For datasets with short documents, strategies that feed the entire document to LLMs and extract attributes all at once are the most cost-effective.

On datasets CSPaper, Player, Legal, Finance and Healthcare with long documents, we can observe significant differences in cost across different systems. DocETL incurs the highest cost because of feeding a number of repeated chunks into the LLM and evaluating possible execution plans. LOTUS follows, as it repeatedly feeds the entire document into the LLM for multiple attributes. Palimpzest and UQE are next, as both feed the entire document to the LLM. QUEST and ZenDB, which employ chunking strategies, cost less. In particular, QUEST is more cost-effective than ZenDB, as it feeds more fine-grained chunks into the LLM, further reducing the cost. Evaporate is the most cost-efficient strategy because it only uses LLMs to analyze several documents for code generation and then runs the code for extraction without additional LLM calls.

Summary IV: For datasets with long documents, strategies that retrieving attribute-related chunks instead of scanning entire documents are more cost-effective without sacrificing accuracy much.

On Healthcare, QUEST offers better cost-effectiveness than most baselines (excluding Evaporate). This advantage is due to its hierarchical two-stage retrieval strategy: (1) Document-level retrieval, which utilizes lightweight summaries to filter out massive amounts of irrelevant documents, and (2) Chunk-level retrieval, which further filters the irrelevant chunks within the remaining documents. This dual mechanism significantly reduces the tokens sent to LLMs. Moreover, QUEST attains an accuracy comparable to systems that exhaustively scan the entire corpus. Systems that scan the full corpus can indeed find the largest number of documents related to the target attributes. However, they typically do not filter input data at the chunk level, thus introducing substantial noise and harming accuracy. In contrast, although the document-level retrieval of QUEST may risk excluding some relevant documents, it prevents feeding much noise into the LLM. As a result, these effects offset each other, making QUEST reach an overall F1-score similar to exhaustive scanning approaches, but at a significantly lower cost. This indicates that the two-stage retrieval strategy is crucial: when retrieval is sufficiently accurate, this strategy can deliver considerable cost reductions without compromising accuracy.

Summary V: On large-scale datasets, employing a multi-stage retrieval strategy (filtering both documents and irrelevant chunks) is beneficial for cost reduction. However, this strategy relies heavily on the quality of retrieval, making the design of robust, highly effective retrieval mechanisms a critical direction for future research.

Table 4 shows the cost of Filter queries. UQE incurs a lower cost on most datasets because it trains a regression model to determine whether each filter condition is satisfied. Chunking-based systems like ZenDB and QUEST reduce token usage compared to Palimpzest, LOTUS, and DocETL by feeding only relevant chunks and using logical optimizations like prioritizing low selectivity and computational cost filters. Datasets Legal and Healthcare lack clear structures, leading to larger chunks in ZenDB and potential duplication when processing multiple attributes. Palimpzest is more cost-effective than LOTUS and DocETL because it employs a logical plan that prioritizes filters with low selectivity.

Summary VI: For queries with filters, QUEST and ZenDB apply logical optimization and chunking-based strategies to reduce costs while maintaining accuracy.

Overall, based on the above two sets of experiments, we observe that on complex datasets, all systems have a relatively low accuracy (e.g., 0.5-0.6 F1-score) and incur high cost. The reasons are two-fold. (1) The documents are long, including much noisy information that misleads LLMs, while the chunk strategies are not perfect. (2) Some of the attributes are difficult to extract (e.g., first_judge, which indicates whether a judgement was the first judgement), as it might involve analyzing multiple chunks and employing inferential reasoning. This reveals research opportunity as below.

Opportunity III: A promising direction is to design a sophisticated chunking approach to produce chunks containing precise information w.r.t. the to-be-extracted attribute, leading to a high-quality and cost-effective strategy for UDA analysis over long documents.

4.4 Overall Latency Comparison (RQ3)

Table 3 and 4 compare the query latency of different systems. We observe that DocETL is the most time-consuming, as it applies the multi-agent technique that calls LLMs multiple times, resulting in the highest overall token usage. LOTUS is the next because it calls LLMs multiple times and each time it feeds the entire document

Method	Art		CSPaper		Player		Legal		Finance		Healthcare	
	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency
Evaporate	-	0.16	-	2.40	-	0.30	-	0.25	-	1.50	-	0.52
Palimpzest	1.64	1.34	20.08	5.82	6.67	1.43	6.96	1.39	138.90	8.38	10.80	2.36
LOTUS	2.33	1.47	32.37	6.31	12.41	1.75	9.93	1.43	297.85	13.24	25.10	3.28
DocETL	7.04	15.86	16.04	48.63	55.53	79.05	154.26	270.88	818.10	1509.08	184.30	304.96
ZenDB	1.94	1.18	4.98	3.61	3.72	0.98	3.92	0.94	32.33	3.55	10.31	1.92
QUEST	1.20	0.89	6.52	1.73	2.06	0.92	3.09	0.89	29.30	2.65	4.70	1.72
UQE	0.92	0.83	42.54	3.67	5.96	1.03	6.12	1.04	124.02	5.83	10.09	1.91

Table 3: Cost and Latency Comparison for `Select` Queries.

Method	Art		CSPaper		Player		Legal		Finance		Healthcare	
	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency
Evaporate	-	0.16	-	2.40	-	0.30	-	0.25	-	1.50	-	0.52
Palimpzest	2.35	1.46	34.86	9.57	11.77	6.23	10.70	4.75	213.80	12.71	18.30	7.16
LOTUS	4.70	6.78	36.61	10.12	11.91	6.61	14.89	4.25	216.10	18.49	18.40	7.81
DocETL	6.15	13.71	26.77	77.66	52.92	74.86	32.70	31.75	184.06	14.67	113.35	47.06
ZenDB	3.51	2.13	71.76	61.59	9.59	4.56	26.69	27.72	49.18	4.73	19.50	2.28
QUEST	2.90	1.86	14.73	3.05	5.60	2.12	9.33	2.10	36.00	4.33	9.46	2.10
UQE	0.97	0.74	15.96	2.15	4.81	0.99	8.00	1.32	33.27	3.98	8.50	1.34

Table 4: Cost and Latency Comparison for `Filter` Queries.

into LLMs. `Palimpzest` follows because it incorporates the chain-of-thought mechanism, leading to more outputs. `UQE` is relatively efficient, but still processes entire documents for each call. However, when filters are applied, its regression model significantly improves the efficiency. `QUEST` and `ZenDB` are efficient because they consume fewer input tokens and trigger fewer LLM calls. `Evaporate` is the most efficient because it generates code for extraction.

Summary VII: Overall, latency is closely related to the input & output tokens as well as the number of LLM calls. Hence, chunking-based strategies with logical optimization (i.e., `QUEST` and `ZenDB`) are the most efficient because they reduce both the number of input tokens and LLMs calls.

4.5 Evaluation of Logical Optimization (RQ4)

Filter Reordering. We compare with four filter reordering strategies as follows. (1) `Random`: the filters are executed in random order; (2) `Sel`: the filters are ordered based on the selectivity; (3) `Sel+Cost` (`ZenDB`): the filters are ordered based on both the selectivity and the estimated average cost of extracting each attribute from the sampled documents; (4) `(Sel+Cost)/doc` (`QUEST`): each document has its own plan considering the selectivity and the estimated extraction cost w.r.t. this document. We compare the above strategies based on `QUEST`. To evaluate sufficiently, we conduct experiments on `Player` over different query subcategories w.r.t. different numbers of filters: S1 with one filter, S2 with 2 filters, and S3 with 3 or more. In Figure 5, for queries in S1, all baselines have nearly the same cost since each query uses a single filter and thus only one order. For queries with more filters, these methods are ranked as follows by the LLMs cost: `(Sel+Cost)/doc` < `Sel+Cost` < `Sel` < `Random`. The first two strategies save more cost because they optimize the order considering the cost. `(Sel+Cost)/doc` is the most cost-effective because it provides fine-grained optimization for each individual document.

Filter Pushdown. We compare with two strategies as follows. (1) `DB-Pushdown`: like in traditional databases, it pushes down filters respectively to the relevant tables before join. (2) `Trans-Join` (`QUEST`): it transforms a join to a filter operation and orders the filters using the above (Sel+Cost)/doc strategy to reduce the cost. We compare the above strategies using `Mixed` queries with filters on `Player` and `Healthcare`. We observe from Figure 6 that `Trans-Join` is more cost-effective than `DB-Pushdown` because given two tables to join, `Trans-Join` builds a cost model to judiciously determine which table will be extracted first and transformed to a filter on the other table, which provides the opportunity to run a join first if it incurs a smaller data extraction cost.

Join order. We evaluate three strategies as follows. (1) `Pushdown`: all filters are pushed down first, and then join is performed; (2) `Random`: tables (document subsets) are joined in random order, with each pair of tables joined using `Trans-Join`; (3) `Dynamic` (`ZenDB`, `QUEST`): it uses a cost model to dynamically identify two tables to join in a left-deep manner, with each pair of tables joined using `Trans-Join`. We compare the above strategies using `Join` queries involving three tables on and `Healthcare`. We observe from Figure 7 that `Dynamic` saves much cost because it dynamically selects the join operation that leads to the lowest cost.

4.6 Evaluation of Physical Optimization (RQ5)

Model Selection. We evaluate two strategies in `Palimpzest` as follows. (1) `Model-Sel`: We define a set of candidate models (GPT-4.1-nano (`Nano`), GPT-4.1-mini (`Mini`), and GPT-4.1 (`Normal`)) and set the selection objective as "minimize cost while achieving the target accuracy." (2) `no-Model-Sel`: The system always uses a specific model (`Mini`) for all queries. The user has to specify a desired accuracy. In our experiments, we set this value as the average accuracy obtained by running `Filter` queries with `Mini` on each dataset (`Art` and `Legal`). We evaluate both strategies using `Filter` queries on `Art` and `Legal`. The cost is calculated by multiplying the number of tokens by the model's price per token. As shown in Figure 8, on `Art`, `Model-Sel` exceeds the target accuracy, while also reducing cost by using a cheaper model. This is achieved by assigning `Normal` to harder attributes and `Nano` to easier ones, while `Art` has more easy attribute than hard attributes. On the challenging `Legal` dataset, `Model-Sel` outperforms `No-Model-Sel` in accuracy, but with higher cost due to more frequently using `Normal`.

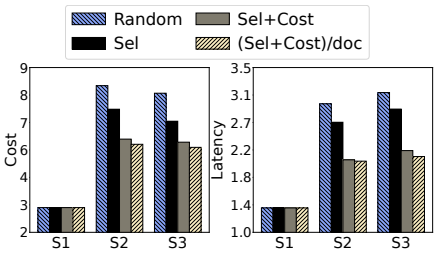

Figure 5: Evaluation of Filter Reordering.

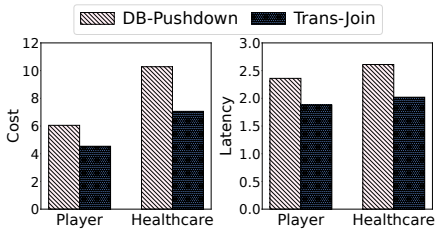

Figure 6: Evaluation of Filter Pushdown.

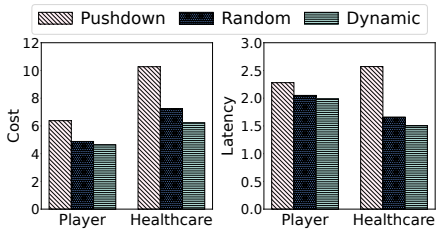

Figure 7: Evaluation of Join Order.

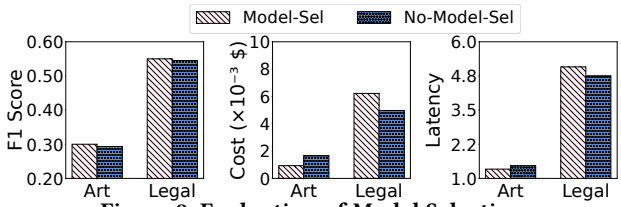

Figure 8: Evaluation of Model Selection.

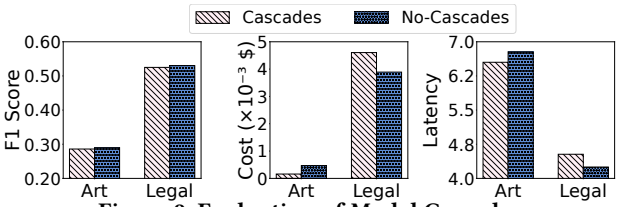

Figure 9: Evaluation of Model Cascades.

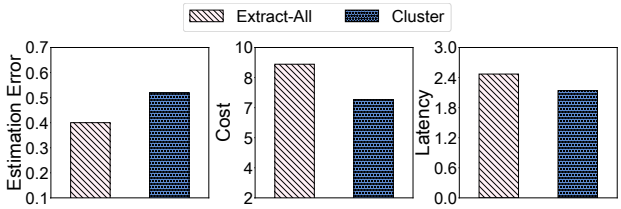

Figure 10: Aggregation Optimization.

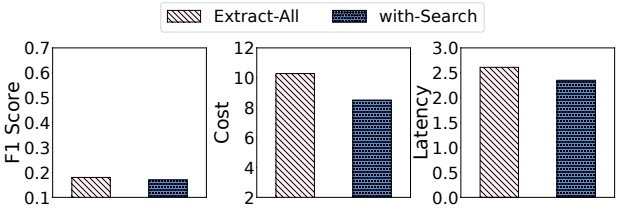

Figure 11: Join Optimization.

Model Cascades. We evaluate two strategies in LOTUS as follows. (1) `Cascades`: we construct a model cascade using two models, GPT-4.1-nano (`Nano`) and GPT-4.1-mini (`Mini`). For each query, the system first uses the lower-cost `Nano` to execute the queries. If the accuracy does not meet the desired accuracy, the query is then forwarded to the larger `Mini` for further processing. (2) `No-Cascades`: The system uses only a single model, `Mini` to process all queries. We evaluate both strategies on the same set of queries for each dataset and use the same cost and target accuracy settings as the above model selection experiment. As shown in Figure 9, on `Art`, `Cascades` achieves the target accuracy while reducing cost by primarily using the less expensive `Nano`. This is because, for most attributes in `Art`, the cheaper model is sufficient. On the challenging `Legal` dataset, `Cascades` perform similar as `No-Cascades`, but incurs higher costs. This is because almost every attribute eventually requires `Mini`, and all documents have to be first processed by `Nano` before being fed to `Mini`.

> Opportunity IV: Currently, there is no system supporting end-to-end query optimization, including all above logical and physical optimization strategies, which would be a promising direction to achieve high-efficacy query execution.

Optimization for Join. We evaluate two join strategies on `Healthcare`. (1) `Extract-All`: it extracts the join key attribute from the two tables and join them. (2) `with-Search`: it extracts each value of the join key attribute from one table, leverages it as a semantic search key to prune many documents in the other table and then join. We can observe from Figure 11 that `with-Search` is more cost-effective than `Extract-All` since many documents in the other table are pruned without extraction. However, the accuracy is low because it incorrectly prunes documents that can be joined.

Optimization for Aggregation. We use the estimation error as the metric following [9], defined as the absolute difference between the estimated and true values divided by the true value. We evaluate two strategies on `Art` using `Agg` queries as follows. (1) `Extract-All`: it extracts the attributes involved in Groupby and Aggregation and then executes the query. (2) `Cluster`: it clusters the attribute-related chunks based on the attribute in Groupby, extracts the attribute in Aggregation within each cluster and executes the query. In Figure 10, `Cluster` is more cost-effective than `Extract-All` because it assumes all documents in a cluster share the same Groupby attribute value and thus avoids extracting that attribute. However, its relative error is high because high-quality clustering based solely on chunk embeddings is difficult.

> Opportunity V: Another promising direction is to develop more effective strategies to align the embeddings of attributes with the embeddings of their chunks. In this way, the clustering in aggregation and pruning in join can be more accurate, and thereby the costs can be reduced more.

4.7 Evaluation of Mixed Queries

We discuss the performance of different systems on three representative `Mixed` queries: `Select` +`Filter` +`Join`, `Select` +`Filter` +`Agg`, and `Select` +`Filter` +`Agg` +`Join`. The figures are put in technical report [10] due to space limitation.

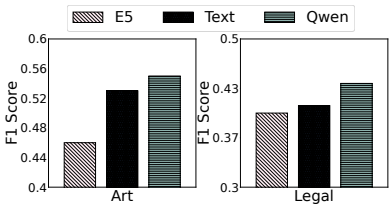

Figure 12: Ablation Study of Embedding Models.

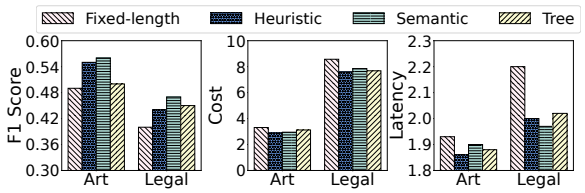

Figure 13: Ablation Study of Chunking Strategies.

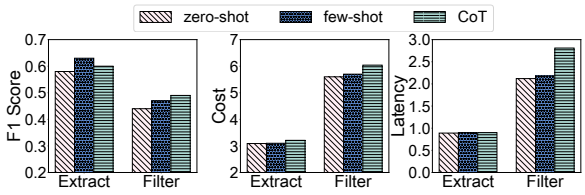

Figure 14: Ablation Study of Prompts.

`Select` +`Filter` +`Join`. Systems like `QUEST` and `ZenDB`, which decompose the task into retrieval, attribute extraction, and then relational joins, consistently outperform methods that ask LLMs to perform semantic joins directly. This step-by-step process yields more reliable answers. However, `QUEST` 's accuracy on these queries is on average 0.32 F1-score, 10% lower than on `Filter` queries because of error propagation: incorrect filters may drop relevant documents, and imperfect extraction of join keys further harms join performance. The cost difference between these two types of systems is modest, since both process multiple attributes and apply filters and joins. Still, `QUEST` and `ZenDB` significantly cut total cost by sending only retrieved chunks, rather than full documents, to the LLM.

`Select` + `Filter` + `Agg`. Similarly, approaches that first extract structured tables and then run exact aggregations obtain the best accuracy with 0.28 F1-score. By decoupling extraction from aggregation, they avoid requiring the LLM to reason over large, noisy contexts. In contrast, semantic aggregation methods (`LOTUS`, `UQE`) perform worse because the LLM must interpret all relevant text and compute aggregations (e.g., count, max) directly in the prompt. `UQE` has the lowest accuracy because it approximates aggregation via sampling instead of exact group-by, but it is also the cheapest since it aggregates only over samples and skips full extraction. `QUEST` and `ZenDB` remain cost-effective by using chunk-level extraction and performing relational group-by after extraction, avoiding multi-step LLM reasoning.

`Select` + `Join` + `Agg` + `Filter`. All systems experience error propagation, causing the accuracy to fall to the level of the weakest mixed query. The cost remains high due to the execution many operators. Approaches that pass entire documents to LLMs (`LOTUS`, `Palimpzest`) incur the greatest expense, whereas `QUEST` and `ZenDB` save cost by using retrieval-based mechanism.

> Opportunity VI: A promising research direction is to design an end-to-end optimization framework that simultaneously consider retrieval, extraction, filtering, joining, and aggregation. This would significantly enhance accuracy and cost-efficiency in complex analytical settings.

4.8 Ablation Studies

Ablation Study of Embedding Models. We evaluate different embedding models to analyze whether using a better embedding model can improve the system performance. We utilize multilingual-e5-large (`E5`) [36], text-embedding-3 (`Text`) [23] and Qwen3-Embdding (`Qwen`) [38]. On MTEB benchmark [21], their performance is ranked as follows: `Qwen` > `Text` > `E5`. We evaluate the performance of different embedding models by replacing the embedding model in `QUEST` and testing on the `Art`, and `Legal` datasets, each with `Filter` queries. As shown in Figure 12, system performance improves as the quality of the embedding model increases. This is because better embedding models provide more accurate semantic representations, resulting in more precise chunk retrieval, and thus improve overall performance.

Ablation Study of Chunking Strategies. We evaluate how different chunking strategies affect system performance: (1) `Fixed-length` [6]: fixed 512-token chunks. (2) `Heuristic` [29]: grammar-based chunks, up to 512 tokens. (3) `Semantic` (`QUEST`): semantic chunks, 128–512 tokens. (4) `Tree` (`ZenDB`): SHT tree–based splitting. We run this experiment by changing the chunking strategy in `QUEST` while running `Filter` queries on `Art` and `Legal`. `Semantic` performs best by capturing rich semantic representations, yielding superior embedding similarity. `Tree` surpasses `Heuristic` on `Art` and `Legal` because it preserves more tokens per chunk, providing richer context at the cost of higher latency and expense. `Fixed-length` performs worst, as it can split sentences across chunks, leading incomplete information.

Ablation Study of Prompts. We evaluate the impact of different prompt design strategies on system performance to determine whether prompt selection will lead to improved results. The prompt design strategies we consider are as follows: (1) `Zero-shot`: directly provides the task instruction to the model without any examples. (2) `Few-shot`: provides the task instruction along with a few annotated examples. (3) `CoT` (Chain-of-Thought): incorporates explicit reasoning steps or prompts the model to generate intermediate reasoning, enabling multi-step problem solving. We evaluate the performance of different prompt design strategies by replacing the prompts in `QUEST` and testing with `Select` and `Filter` queries on the `Legal` dataset. As shown in Figure 16, for extraction tasks, all three approaches perform similarly, with `Few-shot` slightly outperforming others due to its detailed examples, and `CoT` outperforming `Zero-shot` due to its guided reasoning. For queries with filters, `CoT` achieves the best performance by guiding the model through step-by-step reasoning. However, `CoT` produces longer output sequences, resulting in higher latency and greater token costs compared to the other two approaches. Additionally, `Few-shot` requires more input tokens than `Zero-shot`. `Few-shot` ranks next, as concrete examples help the model learn the task pattern. `Zero-shot` performs worst, as it lacks examples and reasoning guidance.

Ablation Study of Different LLMs. models.pdf We evaluate the

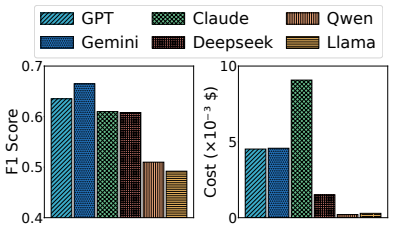

Figure 15: Ablation Study of Different LLMs.

impact of different types of LLMs, including close-source models (GPT-4.1 (`GPT`), Gemini-2.5-pro (`Gemini`) and Claude-sonnet-4 (`Claude`)), and open-source models (Deepseek-V3 (`Deepseek`), Qwen3-30B-A3B (`Qwen`) and Llama-4-scout-17b (`Llama`)) on system performance to assess whether stronger LLMs lead to improved results. To ensure fair comparison, we use the same third-party LLM service provider (`OpenRouter`[2]) and invoke all models via API. We then evaluate the performance of various LLMs by replacing the LLM in `QUEST` and testing each model on `Select` queries from the `Legal` dataset. Figure 15 presents the results, with performance ranked as: `Gemini` > `GPT` > `Claude` > `Deepseek` > `Qwen` > `Llama`, while the inference cost per document is ranked as `Qwen` < `Llama` < `Deepseek` < `GPT` < `Gemini` < `Claude`. We find that higher inference costs do not necessarily guarantee superior performance, and the gap between open-source and close-source models is narrowing rapidly.

5 Related Work

UDA Benchmark. Concurrent with our effort, SemBench [16] builds a benchmark for semantic query processing over multimodal data, critical for pushing this research direction forward. However, it incurs several limitations. First, the numbers of attributes and queries are relatively small (55 queries and 21 unstructured attributes in total), which is not sufficient to evaluate the complex scenarios of unstructured data. Besides, on average, each document has only 173 tokens in SemBench, while documents typically are much more complicated in real-world. `UDA-Bench` instead offers 167 querable attributes with varying characteristics and diverse queries (608 in total); and on average the document length reaches 10,674. Second, the design of SemBench does not consider the real-world large-corpus setting where only a subset of documents is relevant to a given query. This makes it hard to evaluate the document retrieval ability, which is critical to system performance, as confirmed in our experiments. Third, SemBench does not evaluate the key components of UDA systems, e.g., chunking strategies, filter reordering, join order selection, which are essential for fine-grained query optimization.

In Lotus [25], its experimental evaluation is limited to small datasets with few queries – five queries corresponding to five tasks. In Palimpzest [19], the dataset used in the evaluation is relatively small – 1,149 documents in total. In DocETL [30], its evaluation is performed on five datasets, with only one representative query per dataset. Consequently, there remains a lack of benchmarks that simultaneously evaluate complex query logic, large-scale document processing, and cost-accuracy trade-offs.

LLM-powered Unstructured Data Analysis Systems. Lotus [25] introduces semantic operators for unstructured data processing, including indexing, extraction, filtering and joining capabilities that enable the construction of complex analytical pipelines. It provides optimized physical implementation for each operator but lacks logical optimizations. Palimpzest [19] offers libraries for users to write declarative Python code to analyze unstructured data. It optimizes the logical plan of each program via filter reordering based on selectivities. DocETL[30] focuses on improving the accuracy of UDA using a multi-agent strategy, but it does not conduct logical optimization to save cost. ZenDB[18] uses semantic hierarchical trees to identify relevant document sections and applies filter ordering and predicate pushdown for optimization. However, it requires well-structured documents and is evaluated on just 221 documents and 27 queries. UQE [9] provides SQL-like analysis with sampling-based aggregation, while CAESURA [34] decomposes queries into operators to handle data in different modalities.

In addition, extracting data from unstructured sources is a key component in analyzing unstructured information, the strategies have evolved from rule-based methods [17, 24, 27, 28] to modern deep learning approaches [14, 37]. Nowadays, LLM-based systems seek to resolve these issues. `Evaporate` [3] generates extraction code through LLMs, balancing cost and quality through weak supervision. Similarly, early systems [8, 12, 32, 33] employ LLMs for document analysis but overlook cost optimization, which is a critical concern given the computational expense of LLM inference, and lack a thorough benchmarking.

6 Conclusion

In this paper, we build a comprehensive LLM-powered benchmark for unstructured data analysis. We collect six diverse unstructured datasets, define key attributes, and label them manually with the assistance of LLMs in cross-validation. We then construct hundreds of queries with various analytical operators, capable of comprehensively assessing the capability of existing systems, run the queries on the labeled datasets, and analyze their performance in depth.

References

[1] [n.d.]. arXiv. https://arxiv.org. Accessed: 2025-02-17.

[2] [n.d.]. OpenRouter: Unified API access to multiple large language models. https://openrouter.ai/. Accessed: 2025-11-17.

[3] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2025. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. arXiv:2304.09433 [cs.CL] https://arxiv.org/abs/2304.09433

[4] Art-.org. 2025. *Art-.org – Artists and Artworks (19th–21stC.).* https://www.art-.org/

[5] Australasian Legal Information Institute (AustLII). [n.d.]. AustLII – Australasian Legal Information Institute. https://www.austlii.edu.au/.

[6] Sinchana Ramakanth Bhat, Max Rudat, Jannis Spiekermann, and Nicolas Flores-Herr. 2025. Rethinking Chunk Size For Long-Document Retrieval: A Multi-Dataset Analysis. arXiv:2505.21700 [cs.IR] https://arxiv.org/abs/2505.21700

[7] Chengliang Chai, Jiajun Li, Yuhao Deng, Yuanhao Zhong, Ye Yuan, Guoren Wang, and Lei Cao. 2025. Doctopus: Budget-Aware Structural Table Extraction from Unstructured Documents. *Proc. VLDB Endow.* 18, 11 (July 2025), 3695–3707. https://doi.org/10.14778/3749646.3749647

[8] Zui Chen, Zihui Gu, Lei Cao, Ju Fan, Sam Madden, and Nan Tang. 2023. Symphony: Towards Natural Language Query Answering over Multi-modal Data Lakes. https://www.cidrdb.org/cidr2023/papers/p51-chen.pdf

[9] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. 2024. UQE: A Query Engine for Unstructured Databases. arXiv:2407.09522 [cs.DB] https://arxiv.org/abs/2407.09522

[10] DB-121143. 2025. *UDA-Bench: A Benchmark for Unstructured Data Analysis.* Technical Report. DB-121143 Group. https://github.com/DB-121143/UDA-Bench/blob/main/technical_report.pdf Technical Report, accessed 2025-12-01.

[11] Enterprise RAG Challenge. [n.d.]. Enterprise RAG Challenge. https://rag.abdullin.com/. Accessed 17 July 2025.

[12] Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. , 26 pages. https://dl.acm.org/doi/10.1145/3654989

[13] K. V. Kanimozhi and M. Venkatesan. 2015. Unstructured Data Analysis–A Survey. *International Journal of Advanced Research in Computer and Communication Engineering* 4, 3 (2015), 223–225.

[14] Keshav Kolluru, Vaibhav Adlakha, Samarth Aggarwal, Mausam, and Soumen Chakrabarti. 2020. OpenIE6: Iterative Grid Labeling and Coordination Analysis for Open Information Extraction. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 3748–3761. https://doi.org/10.18653/v1/2020.emnlp-main.306

[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody-Hao Yu, JosephE Gonzalez, Hao Zhang, and Ion Stoica. [n.d.]. Efficient Memory Management for Large Language Model Serving with PagedAttention. ([n. d.]).

[16] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. 2025. SemBench: A Benchmark for Semantic Query Processing Engines. arXiv:2511.01716 [cs.DB] https://arxiv.org/abs/2511.01716

[17] Taesung Lee, Zhongyuan Wang, Haixun Wang, and Seung-won Hwang. 2013. Attribute extraction and scoring: A probabilistic approach. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 194–205. https://doi.org/10.1109/ICDE.2013.6544825

[18] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G. Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. arXiv:2405.04674 [cs.DB] https://arxiv.org/abs/2405.04674

[19] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A Declarative System for Optimizing AI Workloads. arXiv:2405.14696 [cs.CL] https://arxiv.org/abs/2405.14696

[20] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. arXiv:cs/0205028 [cs.CL] https://arxiv.org/abs/cs/0205028

[21] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. arXiv:2210.07316 [cs.CL] https://arxiv.org/abs/2210.07316

[22] National Basketball Association – Wikipedia. [n.d.]. NBA – Wikipedia. https://en.wikipedia.org/wiki/National_Basketball_Association. Accessed 17 July 2025.

[23] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. 2022. Text and Code Embeddings by Contrastive Pre-Training. arXiv:2201.10005 [cs.CL] https://arxiv.org/abs/2201.10005

[24] Christina Niklaus, Matthias Cetto, André Freitas, and Siegfried Handschuh. 2018. A Survey on Open Information Extraction. arXiv:1806.05599 [cs.CL] https://arxiv.org/abs/1806.05599

[25] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators: A Declarative Model for Rich, AI-based Data Processing. arXiv:2407.11418 [cs.DB] https://arxiv.org/abs/2407.11418

[26] Pengcheng Qiu, Chaoyi Wu, Xiaoman Zhang, Weixiong Lin, Haicheng Wang, Ya Zhang, Yanfeng Wang, and Weidi Xie. 2024. Towards Building Multilingual Language Model for Medicine. arXiv:2402.13963 [cs.CL] https://arxiv.org/abs/2402.13963

[27] Swarnadeep Saha and Mausam. 2018. Open Information Extraction from Conjunctive Sentences. , 2288–2299 pages. https://aclanthology.org/C18-1194/

[28] Swarnadeep Saha, Harinder Pal, and Mausam. 2017. Bootstrapping for Numerical Open IE. , 317–323 pages. https://doi.org/10.18653/v1/P17-2050

[29] Roie Schwaber-Cohen and Arjun Patel. 2025. Chunking Strategies for LLM Applications. Pinecone Blog. https://www.pinecone.io/learn/chunking-strategies-for-llm-applications/ Retrieved 17 July 2025 from Pinecone website.

[30] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. arXiv:2410.12189 [cs.DB] https://arxiv.org/abs/2410.12189

[31] Zhaoze Sun, Qiyan Deng, Chengliang Chai, Kaisen Jin, Xinyu Guo, Han Han, Ye Yuan, Guoren Wang, and Lei Cao. 2025. QUEST: Query Optimization in Unstructured Document Analysis. arXiv:2507.06515 [cs.DB] https://arxiv.org/abs/2507.06515

[32] James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Halevy. 2021. From Natural Language Processing to Neural Databases. , 1033–1039 pages. https://doi.org/10.14778/3447689.3447706

[33] Matthias Urban and Carsten Binnig. 2023. Towards Multi-Modal DBMSs for Seamless Querying of Texts and Tables. arXiv:2304.13559 [cs.DB] https://arxiv.org/abs/2304.13559

[34] Matthias Urban and Carsten Binnig. 2024. CAESURA: Language Models as Multi-Modal Query Planners. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, CA, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p14-urban.pdf

[35] Bin Wang, Chao Xu, Xiaomeng Zhao, Linke Ouyang, Fan Wu, Zhiyuan Zhao, Rui Xu, Kaiwen Liu, Yuan Qu, Fukai Shang, Bo Zhang, Liqun Wei, Zhihao Sui, Wei Li, Botian Shi, Yu Qiao, Dahua Lin, and Conghui He. 2024. MinerU: An Open-Source Solution for Precise Document Content Extraction. arXiv:2409.18839 [cs.CV] https://arxiv.org/abs/2409.18839

[36] Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2024. Multilingual E5 Text Embeddings: A Technical Report. arXiv:2402.05672 [cs.CL] https://arxiv.org/abs/2402.05672

[37] Bowen Yu, Yucheng Wang, Tingwen Liu, Hongsong Zhu, Limin Sun, and Bin Wang. 2021. Maximal Clique Based Non-Autoregressive Open Information Extraction. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 9696–9706. https://doi.org/10.18653/v1/2021.emnlp-main.764

[38] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. *arXiv preprint arXiv:2506.05176* (2025).

A Evaluation

A.1 System Adaptation for Evaluation

We have made several modifications to the open source codes for different systems to ensure that all systems can operate within a unified environmental configuration, thus enabling the convenient acquisition of the test results and reliable conclusions. The specific modifications are detailed as follows.

Evaporate. The original code applied an exact matching approach when identifying attribute-related texts; however, the specified text may not necessarily appear in actual documents. To more accurately capture the content potentially requiring extraction, we have introduced a synonym matching function, enabling the system to recognize and extract synonymous texts relevant to the queried attributes.

(1) Synonym Matching

Original:

```
for chunk in chunks:
    if attribute.lower() in chunk.lower():
        cleaned_chunks.append(chunk)
```

New:

```
def match_with_synonyms(attribute, chunk):
    synonyms = ATTRIBUTE_SYNONYMS.get(attribute.
     lower(), [attribute])
    for syn in synonyms:
        if syn.startswith(r"\b"):
            if re.search(syn, chunk):
                return True
            else:
            if syn.lower() in chunk.lower():
                return True
    return False

for chunk in chunks:
    if match_with_synonyms(attribute, chunk):
        cleaned_chunks.append(chunk)
```

Palimpzest. During the testing process, we encountered several errors related to numerical and null-value attributes that we have resolved through targeted modifications. In addition, we have adjusted the method for invoking the LLM to ensure compatibility with the API interface utilized in our testing procedures.

(1) Empty Value

Original:

```
field_lengths = [(field, len(value) if value is not
     None else 0) for field, value in context.items
     ()]
```

New:

```
def _get_value_length(value):
        if value is None:
                return 0
        if isinstance(value, str):
                return len(value)
        if isinstance(value, (list, dict)):
                return len(str(value))
        # change to str and get the length
        return len(str(value))
field_lengths = [(field, _get_value_length(value))
     for field, value in context.items()]
```

(2) Numeric Value

Original:

```
num_outputs = sum(record.passed_operator for record
     in record_set.data_records)
```

New:

```
num_outputs = sum(record.passed_operator for record
     in record_set.data_records if record.
     passed_operator is not None)
```

(3) LLM

Original:

```
OpenAI(api_key=api_key)
```

New:

```
OpenAI(api_key=api_key, base_url=api_base)
```

LOTUS. To enable the model cascade function, we made simple modifications to this component.

(1) Synonym Matching

Original:

```
def _get_top_choice_logprobs(self, response:
     ModelResponse) -> list[
     ChatCompletionTokenLogprob]:
      choice = response.choices[0]
      assert isinstance(choice, Choices)
      logprobs = choice.logprobs["content"]
      return [ChatCompletionTokenLogprob(**logprob)
     for logprob in logprobs]
```

New:

```
def _get_top_choice_logprobs(self, response:
     ModelResponse) -> list[
     ChatCompletionTokenLogprob]:
    choice = response.choices[0]
    assert isinstance(choice, Choices)
    logprobs = choice.logprobs["content"]
    return logprobs
```

DocETL. We did not make any special modifications to this system.

A.2 Dataset Details

This section will first provide an overview of the dataset, then introduce all the attributes in each dataset in our benchmark and all detailed descriptions about them.

- **`Art` Dataset.** `Art` include 20 textual attributes and 6 image attributes. These 26 attributes are combined into one table in the ground truth.
  - `Art` Text: Biographies and information about the artists.
  - `Art` Image: Visual features and characteristics of the images.
- **`Player` Dataset:** `Player` include statistics of professional basketball players, teams, owners, and cities
  - Player: Personal information, draft history, and career achievements
  - Owner: Ownership details and personal demographics
  - Team: Organizational information and championship records
  - City: Geographic and economic data of NBA team cities
- **`Legal` Dataset:** `Legal` include documentation of court cases and legal proceedings
- **`Healthcare` Dataset:**
  - Disease: Medical conditions and disease information
  - Drug: Pharmaceutical drugs and medication information
  - Medical Institutes: Medical research institutions and organizations
- **`Finance` Dataset:** `Finance` The data set consists of financial reports and company information.

(1) `Art` Text Dataset Attributes

This dataset contains 20 attributes capturing comprehensive biographical and career information about artists, including their personal details, artistic movements, educational background, and professional achievements.

—

—

—

(2) `Art` Image Dataset Attributes

This dataset contains 6 attributes focusing on the visual characteristics and artistic elements of artworks, including style, composition, and color properties.

Attribute	Description
style	The style of the artwork (e.g., realism, abstract)
image_genre	The genre of the artwork (e.g., landscape, portrait)
object	The primary objects or subjects in the artwork
color	The predominant color used in the artwork
tone	The tonal qualities (e.g., bright, dark, neutral)
composition	The composition style of the artwork

(3) NBA Dataset - Player Attributes

This dataset contains 13 attributes covering NBA players' personal information, draft history, career achievements, and championship records across different basketball competitions.

Attribute	Description
name	Full name of the NBA player (e.g., LeBron James)
birth_date	Birth date of the player; use format YYYY/%-m/%-d (e.g., 1984/2/30)
nationality	Nationality of the player (e.g., USA, France)
age	Age of the player (if death_date is given, compute as death_date - birth_date; otherwise, compute as 2025/1/1 - birth_date)
team	The current NBA team the player belongs to, or the last NBA team the player joined if not currently active (e.g., Los Angeles Lakers)
position	Player's primary position on the team; choose one from ['Frontcourt', 'Backcourt']
draft_pick	The pick number at which the player was selected in the NBA draft (e.g., 1). If the player was selected in the second round, add 30 to the pick number (e.g., 2nd round, 1st pick → 31)
draft_year	The year the player was selected in the NBA draft; use format YYYY (e.g., 2003)
college	The college the player attended, if applicable (e.g., Duke University)
nba_championships	Number of NBA championships the player has won (e.g., 4)
mvp_awards	Number of NBA MVP awards the player has won; use 0 if none (e.g., 2)
olympic_gold_medals	Number of Olympic gold medals the player has won (e.g., 3)
fiba_world_cup	Number of FIBA World Cup titles the player has won (e.g., 1)

(4) NBA Dataset - Team Attributes *This dataset contains 5 attributes covering NBA teams' organizational information, historical data, geographic location, and championship achievements.*

Attribute	Description
team_name	Official name of the NBA team (e.g., Los Angeles Lakers, Boston Celtics)
founded_year	Year when the team was founded or established; use format YYYY (e.g., 1947)
location	Current city where the team is based and plays home games (e.g., Los Angeles, Boston)
ownership	Full name of the current team owner or primary owner (e.g., Jeanie Buss)
championships	Total number of NBA championships won by the team throughout its history (e.g., 17)

(5) NBA Dataset - Owner Attributes *This dataset contains 5 attributes covering NBA team owners' personal information, ownership details, and tenure with their respective teams.*

Attribute	Description
name	Full name of the NBA team owner (e.g., Steve Ballmer, Jeanie Buss)
age	Current age of the team owner (e.g., 68)
nationality	Nationality of the team owner (e.g., USA, Canada)
NBA_team	Name of the NBA team owned by this individual (e.g., Los Angeles Clippers)
own_year	Year when the owner acquired or started owning the NBA team; use format YYYY-MM-DD (e.g., 2014-05-29)

(6) NBA Dataset - City Attributes *This dataset contains 5 attributes covering NBA team cities' geographic information, demographic data, and economic indicators.*

Attribute	Description
city_name	Name of the city where NBA teams are located (e.g., Los Angeles, Boston)
state_name	Name of the state where the city is located (e.g., California, Massachusetts)
population	Total population of the city (e.g., 3971883)
area	Total area of the city in square kilometers; use decimal format (e.g., 1302.15)
gdp	Gross domestic product of the city in millions of dollars; use decimal format (e.g., 789543.50)

(7) Legal Case Reports Dataset Attributes

This dataset contains 19 attributes documenting legal proceedings, including case participants, judicial decisions, financial penalties, and procedural details of court cases.

(8) Healthcare Dataset - Disease Attributes

This dataset contains 19 attributes providing comprehensive medical information about diseases, including their classification, clinical manifestations, diagnostic approaches, treatment options, and impact on patients' lives.

(9) Healthcare Dataset - Drug Attributes

This dataset contains 17 attributes detailing pharmaceutical drugs, including their composition, administration methods, therapeutic uses, safety profiles, and storage requirements.

(10) Healthcare Dataset - Medical Institutes Attributes

This dataset contains 15 attributes describing medical research institutions, including their organizational structure, research focus areas, technological capabilities, and collaborative partnerships.

(11) Finance Dataset Attributes

This dataset contains 25 attributes capturing comprehensive financial and corporate information, including company structure, financial performance metrics, shareholder information, and business risk assessments.

A.3 Query Templates and Examples

This section illustrates our template design in detail about the semantic meaningful queries in each dataset. It should be noted that the focus of this section is to illustrate our `Filter`, thus we show our templates in the form of `select|filter` and `select|filter|join`.

(1) Art Dataset. Art dataset includes 10 manually designed templates without `join` operation.

Template 1: nationality_focus

```
SELECT {Selected_Column}
FROM Art
WHERE Nationality = {Nationality};
```

Template 2: style_focus

```
SELECT {Selected_Column}
FROM Art
WHERE Style = {Style};
```

Template 3: nationality_or_style

```
SELECT {Selected_Column}
FROM Art
WHERE Nationality = {Nationality}
   OR Style = {Style};
```

Template 4: century_or_color_preference

```
SELECT {Selected_Column}
FROM Art
WHERE Century = {Century}
   OR Color = {Color};
```

Template 5: age_range_or_theme

```
SELECT {Selected_Column}
FROM Art
WHERE (Age >= {Min_Age}
   AND Age <= {Max_Age})
   OR Theme = {Theme};
```

Template 6: european_masters_or_award_winners

```
SELECT {Selected_Column}
FROM Art
WHERE Birth_continent = 'Europe'
   OR Awards IS NOT NULL;
```

Template 7: renaissance_italian_masters

```
SELECT {Selected_Column}
FROM Art
WHERE Nationality = 'Italian'
   AND Style = 'Renaissance';
```

Template 8: married_or_mature_artists

```
SELECT {Selected_Column}
FROM Art
WHERE Marriage = 'Married'
   OR Age >= 50;
```

Template 9: contemporary_painters_or_sculptors

```
SELECT {Selected_Column}
FROM Art
WHERE (Century = '20th'
   AND Field = 'Painting')
   OR Field = 'Sculpture';
```

Template 10: exceptional_artists

```
SELECT {Selected_Column}
FROM Art
WHERE Age > 80
   OR Awards IS NOT NULL
   OR Style = {Style1}
   OR Style = {Style2}
   OR Style = {Style3};
```

(2) Legal Dataset. Legal dataset includes 10 manually designed templates without `join` operation.

Template 1: judge_focus

```
SELECT {Selected_Column}
FROM Legal
WHERE judge_name = {Judge_Name};
```

Template 2: case_type_focus

```
SELECT {Selected_Column}
FROM Legal
WHERE case_type = {Case_Type};
```

Template 3: specific_judge_or_location

```
SELECT {Selected_Column}
FROM Legal
WHERE judge_name = {Judge_Name}
   OR hearing_location = {Hearing_Location};
```

Template 4: case_type_or_outcome

```
SELECT {Selected_Column}
FROM Legal
WHERE case_type = {Case_Type}
   OR verdict = {Verdict};
```

Template 5: recent_or_high_value_cases

```
SELECT {Selected_Column}
FROM Legal
WHERE hearing_year >= {Recent_Year}
   OR legal_fees > {High_Fee_Threshold};
```

Template 6: multi_year_analysis

```
SELECT {Selected_Column}
FROM Legal
WHERE hearing_year = {Year1}
   OR hearing_year = {Year2}
   OR hearing_year = {Year3}
   OR (hearing_year >= {Start_Year}
       AND hearing_year <= {End_Year});
```

Template 7: major_courts_or_complex_cases

```
SELECT {Selected_Column}
FROM Legal
WHERE hearing_location = {Major_Court1}
   OR hearing_location = {Major_Court2}
   OR hearing_location = {Major_Court3}
   OR (case_type = 'Commercial'
     AND legal_fees > {Complex_Fee_Threshold});
```

Template 8: successful_outcomes_or_settlements

```
SELECT {Selected_Column}
FROM Legal
WHERE verdict = 'Guilty'
   OR verdict = 'Liable'
   OR verdict = 'Settled';
```

Template 9: judicial_efficiency_analysis

```
SELECT {Selected_Column}
FROM Legal
WHERE hearing_year = judgment_year
   OR (judgment_year - hearing_year) >
   {Long_Duration_Threshold};
```

Template 10: high_profile_litigation

```
SELECT {Selected_Column}
FROM Legal
WHERE legal_fees > {High_Profile_Threshold}
   OR (case_type = 'Commercial'
       AND (hearing_location = {Major_City1}
            OR hearing_location = {Major_City2}
           OR hearing_location = {Major_City3}))
   OR (plaintiff = {Corporate_Plaintiff}
       AND defendant = {Corporate_Defendant});
```

(3) Finance Dataset. Finance dataset includes 10 manually designed templates without `join` operation.

Template 1: exchange_focus

```
SELECT {Selected_Column}
FROM Finance
WHERE exchange_code = {Exchange_Code};
```

Template 2: high_performers

```
SELECT {Selected_Column}
FROM Finance
WHERE revenue > {High_Revenue_Threshold}
  OR net_profit_or_loss > {High_Profit_Threshold};
```

Template 3: tech_or_finance_companies

```
SELECT {Selected_Column}
FROM Finance
WHERE principal_activities = 'Technology'
   OR principal_activities = 'Finance';
```

Template 4: major_exchanges_or_profitable

```
SELECT {Selected_Column}
FROM Finance
WHERE exchange_code = 'NYSE'
   OR exchange_code = 'NASDAQ'
   OR net_profit_or_loss > {Profit_Threshold};
```

Template 5: regional_leaders_or_dividend_payers

```
SELECT {Selected_Column}
FROM Finance
WHERE registered_office = {Target_Region}
   OR dividend_per_share > {Min_Dividend};
```

Template 6: value_or_growth_stocks

```
SELECT {Selected_Column}
FROM Finance
WHERE total_Debt < {Low_Debt_Threshold}
   OR revenue > {High_Revenue_Threshold};
```

Template 7: stable_cash_rich_companies

```
SELECT {Selected_Column}
FROM Finance
WHERE total_assets > {Min_Assets}
   AND cash_reserves > {Min_Cash};
```

Template 8: market_leaders_or_emerging

```
SELECT {Selected_Column}
FROM Finance
WHERE revenue > {Leader_Revenue_Threshold}
  OR earnings_per_share > {Growth_EPS_Threshold};
```

Template 9: defensive_or_income_stocks

```
SELECT {Selected_Column}
FROM Finance
WHERE dividend_per_share > 0
   OR (revenue > {Stable_Revenue}
       AND total_Debt < {Conservative_Debt});
```

Template 10: quality_companies_comprehensive

```
SELECT {Selected_Column}
FROM Finance
WHERE (revenue > {Min_Revenue}
       AND net_profit_or_loss > {Min_Profit})
   OR (earnings_per_share > {Min_EPS}
      AND dividend_per_share > {Min_Dividend});
```

(4) NBA Dataset. NBA dataset includes 10 manually designed templates with `join` operation.

Template 1: nationality_analysis

```
SELECT {Selected_Column}
FROM player
WHERE nationality = {Nationality};
```

Template 2: international_or_veteran_players

```
SELECT {Selected_Column}
FROM player
WHERE nationality != 'USA'
   OR age >= 35;
```

Template 3: champions_or_mvps

```
SELECT {Selected_Column}
FROM player
WHERE nba_championships > 0
   OR mvp_awards > 0;
```

Template 4: recent_draft_or_experienced

```
SELECT {Selected_Column}
FROM player
WHERE draft_year >= {Recent_Draft_Year}
   OR age >= {Veteran_Age};
```

Template 5: successful_teams

```
SELECT {Selected_Column}
FROM player p
JOIN team t ON p.team_id = t.team_id
WHERE t.championship > {Min_Team_Championships}
   OR p.mvp_awards > 0;
```

Template 6: young_teams

```
SELECT {Selected_Column}
FROM player p
JOIN team t ON p.team_id = t.team_id
WHERE t.founded_year > {Recent_Foundation_Year}
   OR p.nationality != 'USA';
```

Template 7: experienced_ownership

```
SELECT {Selected_Column}
FROM player p
JOIN team t ON p.team_id = t.team_id
JOIN owner o ON t.owner_id = o.owner_id
WHERE o.own_year < {Long_Ownership_Threshold}
   OR (p.age <= {Young_Age}
       AND p.draft_pick <= {High_Draft_Pick});
```

Template 8: championship_culture_analysis

```
SELECT {Selected_Column}
FROM player p
JOIN team t ON p.team_id = t.team_id
WHERE (t.championship > 0
       AND p.nba_championships > 0)
   OR p.mvp_awards > 1;
```

Template 9: age_performance

```
SELECT {Selected_Column}
FROM player
WHERE age >= {Min_Age}
   AND age <= {Max_Age}
   AND nba_championships > {Min_Championships};
```

Template 10: draft_analysis

```
SELECT {Selected_Column}
FROM player
WHERE draft_year = {Draft_Year}
   AND draft_pick <= {Max_Draft_Pick};
```

(5) Healthcare Dataset. Healthcare dataset includes 10 manually designed templates with `join` operation.

Template 1: disease_join_analysis

```
SELECT {Selected_Column}
FROM disease d
JOIN drug dr ON d.disease_id = dr.target_disease_id
WHERE d.disease_type = {Disease_Type};
```

Template 2: common_diseases_or_rare_conditions

```
SELECT {Selected_Column}
FROM disease d
JOIN drug dr ON d.disease_id = dr.target_disease_id
WHERE d.disease_type = 'Diabetes'
   OR d.disease_type = 'Hypertension'
   OR d.disease_type = 'Cancer'
   OR d.treatments = {Rare_Treatment_Type};
```

Template 3: oral_medications

```
SELECT {Selected_Column}
FROM drug dr
JOIN institutes i ON
dr.manufacturer_id = i.institute_id
WHERE dr.pharmaceutical_form = 'Tablet'
   OR dr.administration_route = 'Intravenous';
```

Template 4: prescription_drugs

```
SELECT {Selected_Column}
FROM drug dr
JOIN institutes i ON
dr.manufacturer_id = i.institute_id
WHERE dr.prescription_status = 'Prescription'
   OR i.institution_type = 'Hospital';
```

Template 5: western_pharma

```
SELECT {Selected_Column}
FROM drug dr
JOIN institutes i ON
dr.manufacturer_id = i.institute_id
WHERE i.institution_country = 'USA'
   OR i.institution_country = 'Germany'
   OR i.institution_country = 'Switzerland'
   OR i.institution_type = 'University';
```

Template 6: chronic_disease_management

```
SELECT {Selected_Column}
FROM disease d
JOIN drug dr ON d.disease_id = dr.target_disease_id
WHERE (d.disease_type = 'Diabetes'
```

```
      OR d.disease_type = 'Hypertension')
   AND d.treatments = {Long_Term_Treatment};
```

Template 7: innovative_treatments

```
SELECT {Selected_Column}
FROM disease d
JOIN drug dr ON
d.disease_id = dr.target_disease_id
JOIN institutes i ON
dr.manufacturer_id = i.institute_id
WHERE (dr.pharmaceutical_form = 'Injection'
       AND i.research_fields =
       {Biotech_Research_Field})
   OR (dr.pharmaceutical_form = 'Tablet'
     AND d.treatments = {Traditional_Treatment});
```

Template 8: cancer_research

```
SELECT {Selected_Column}
FROM disease d
JOIN drug dr ON
d.disease_id = dr.target_disease_id
JOIN institutes i ON
dr.manufacturer_id = i.institute_id
WHERE d.disease_type = 'Cancer'
   OR i.research_fields = 'Cardiology';
```

Template 9: global_health_solutions

```
SELECT {Selected_Column}
FROM disease d
JOIN drug dr ON
d.disease_id = dr.target_disease_id
JOIN institutes i ON
dr.manufacturer_id = i.institute_id
WHERE dr.prescription_status = 'OTC'
   OR (i.institution_type = 'University'
       AND i.institution_country != 'USA');
```

Template 10: drug_join_analysis

```
SELECT {Selected_Column}
FROM drug dr
JOIN institutes i ON
dr.manufacturer_id = i.institute_id
WHERE dr.pharmaceutical_form = {Pharma_Form};
```

A.4 Evaluation Results

Table 5 and 6 present the average precision, recall, and F1-score of all queries across all the datasets. Given a query Q, the set of tuples returned by a method is denoted as $T(Q)$, we compute the precision, recall, and F1-score for each column individually. The average of these column-level scores is then reported as the overall precision, recall, and F1-score for the corresponding SQL query. Table 5 shows the experimental results of different methods on queries with only extraction. And Table 6 shows the experimental results of different methods on queries with filter.

A.5 Ablation Study of Prompts Strategies

Ablation Study of Prompts. We evaluate the impact of different prompt design strategies on system performance to determine whether prompt selection will lead to improved results. The prompt design strategies we consider are as follows: (1) `Zero-shot`: directly provides the task instruction to the model without any examples. (2) `Few-shot`: provides the task instruction along with a few annotated examples. (3) `CoT` (Chain-of-Thought): incorporates explicit reasoning steps or prompts the model to generate intermediate reasoning, enabling multi-step problem solving. We evaluate the performance of different prompt design strategies by replacing the prompts in `QUEST` and testing with five queries on the `Legal` dataset. As shown in Figure 16, for extraction tasks, all three approaches perform similarly, with `Few-shot` slightly outperforming others due to its detailed examples, and `CoT` outperforming `Zero-shot` due to its guided reasoning. For queries with filters, `CoT` achieves the best performance by guiding the model through step-by-step reasoning. However, `CoT` produces longer output sequences, resulting in higher latency and greater token costs compared to the other two approaches. Additionally, `Few-shot` requires more input tokens than `Zero-shot`. `Few-shot` ranks next, as concrete examples help the model learn the task pattern. `Zero-shot` performs worst, as it lacks examples and reasoning guidance.

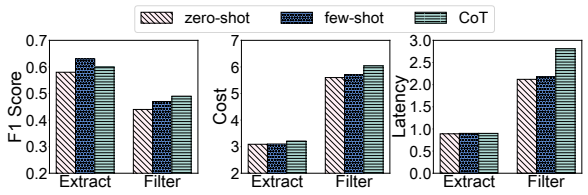

Figure 16: Ablation Study of Prompts.

A.6 Ablation Study of Chunking Strategies

In our ablation studies, several text chunking strategies are utilized to divide long documents for subsequent processing tasks. In this study, we consider four representative chunking methods: fixed-length chunking, heuristic chunking, semantic chunking, and tree-based chunking. The implementation details of each strategy are provided in the following paragraphs.

Fixed-length chunking splits the text into segments of a fixed length (e.g., 512 tokens), with an overlap of 128 tokens between adjacent chunks to reduce information loss in long-text scenarios. This method is simple and fast to process, but it often breaks complete sentences, resulting in information fragmentation and semantic incompleteness.

Heuristic chunking divides the text recursively based on the priority of delimiters (such as \n\n, ., !, ?), ensuring that each chunk contains no more than 512 tokens. Higher-level delimiters are preferred, and when the chunk size constraint cannot be satisfied, the method falls back to the next delimiter level. This strategy minimizes sentence splitting and preserves information integrity, but its effectiveness depends on the original text format and paragraph segmentation.

Semantic chunking (e.g., QUEST) first applies sentence splitters (such as spaCy or nltk) to segment the text into sentences, and then merges semantically similar adjacent sentences using the semantic chunking module in LangChain. A sliding window calculates

Method	Art			CSPaper			Player			Legal			Finance			Healthcare		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Evaporate	0.33	0.33	0.33	0.38	0.38	0.38	0.34	0.34	0.34	0.31	0.32	0.31	0.21	0.33	0.24	0.10	0.11	0.10
Palimpzest	0.61	0.65	0.63	0.65	0.65	0.65	0.88	0.88	0.88	0.68	0.67	0.67	0.53	0.51	0.52	0.24	0.24	0.24
LOTUS	0.58	0.63	0.60	0.64	0.64	0.64	0.86	0.87	0.87	0.66	0.66	0.66	0.51	0.54	0.52	0.24	0.25	0.24
DocETL	0.51	0.51	0.51	0.56	0.56	0.56	0.89	0.89	0.89	0.57	0.57	0.57	0.43	0.44	0.43	0.17	0.18	0.18
ZenDB	0.47	0.47	0.47	0.58	0.58	0.58	0.82	0.84	0.83	0.57	0.57	0.57	0.49	0.48	0.48	0.24	0.24	0.24
QUEST	0.45	0.55	0.48	0.60	0.60	0.60	0.84	0.84	0.84	0.58	0.58	0.58	0.50	0.49	0.49	0.56	0.23	0.32
UQE	0.57	0.57	0.57	0.62	0.62	0.62	0.86	0.87	0.86	0.65	0.66	0.65	0.51	0.53	0.52	0.23	0.22	0.22

Table 5: Performance Comparison of Queries with only Extraction.

Method	Art			CSPaper			Player			Legal			Finance			Healthcare		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Evaporate	0.07	0.01	0.01	0.27	0.27	0.27	0.31	0.12	0.16	0.13	0.11	0.12	0.06	0.18	0.08	0.03	0.01	0.01
Palimpzest	0.32	0.33	0.29	0.57	0.65	0.56	0.66	0.77	0.69	0.54	0.55	0.54	0.36	0.46	0.39	0.22	0.41	0.26
LOTUS	0.24	0.53	0.29	0.51	0.52	0.52	0.48	0.71	0.56	0.49	0.62	0.53	0.23	0.51	0.37	0.19	0.34	0.23
DocETL	0.23	0.23	0.23	0.45	0.45	0.45	0.70	0.79	0.73	0.55	0.57	0.55	0.31	0.32	0.29	0.21	0.40	0.23
ZenDB	0.21	0.23	0.22	0.38	0.67	0.43	0.72	0.64	0.68	0.49	0.44	0.43	0.34	0.40	0.35	0.18	0.47	0.23
QUEST	0.21	0.25	0.22	0.42	0.70	0.47	0.64	0.79	0.70	0.50	0.42	0.44	0.35	0.42	0.36	0.20	0.58	0.27
UQE	0.19	0.19	0.19	0.38	0.38	0.38	0.55	0.73	0.62	0.37	0.36	0.36	0.22	0.59	0.30	0.18	0.23	0.16

Table 6: Performance Comparison of Queries with Filter.

the average semantic similarity, and a new chunk is created whenever the similarity exceeds a predefined threshold (typically set at the 95th percentile of all pairwise sentence similarities). Chunk lengths are restricted to between 128 and 512 tokens to balance recall and embedding quality. This approach better preserves semantic continuity and information richness, thereby improving embedding quality, but it incurs higher computational cost and slower processing speed.

Tree-based chunking (e.g., ZenDB) utilizes the hierarchical structure of document headings by constructing an SHT tree. The content between every two headings is assigned to a node, and a summary for each section is generated via post-order traversal to facilitate retrieval. This method is particularly suitable for PDF and Markdown documents with clear hierarchical structures: Markdown headings are extracted using regular expressions, and PDF headings are identified via pattern matching (see [18] for details). Tree-based chunking effectively avoids sentence fragmentation and preserves more tokens and context, but introduces increased chunking and processing overhead, resulting in greater latency.

Method	Player			Healthcare		
	Precision	Recall	F1	Precision	Recall	F1
Extract-All	0.57	0.77	0.64	0.15	0.33	0.18
with-Search	0.50	0.83	0.62	0.12	0.41	0.17

Table 7: Evaluation of Join Optimization.

Query	Method	Art			Player			Legal		
		Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Extract	E5	0.46	0.46	0.46	0.83	0.83	0.83	0.53	0.53	0.53
	Text	0.53	0.53	0.53	0.84	0.84	0.84	0.55	0.55	0.55
	Qwen	0.55	0.53	0.55	0.89	0.89	0.89	0.58	0.58	0.58
Filter	E5	0.40	0.41	0.40	0.61	0.76	0.67	0.41	0.43	0.40
	Text	0.40	0.44	0.41	0.64	0.77	0.68	0.37	0.44	0.40
	Qwen	0.42	0.50	0.44	0.64	0.79	0.70	0.48	0.46	0.44

Table 8: Ablation Study of Embedding Models.

Method	Art			Player			Legal		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Fixed-length	0.50	0.56	0.49	0.60	0.80	0.67	0.41	0.44	0.40
Heuristic	0.58	0.57	0.55	0.64	0.79	0.69	0.50	0.44	0.44
Semantic	0.59	0.58	0.56	0.68	0.80	0.72	0.52	0.48	0.47
Tree	0.51	0.56	0.50	0.66	0.79	0.70	0.49	0.45	0.45

Table 9: Ablation Study of Chunking Strategies.

Method	Extract			Filter		
	Precision	Recall	F1	Precision	Recall	F1
gpt-4.1-mini	0.58	0.58	0.58	0.50	0.44	0.44
gpt-4.1-nano	0.52	0.52	0.52	0.42	0.42	0.40
gpt-4.1	0.63	0.62	0.63	0.53	0.48	0.48
DeepSeek-V3	0.61	0.61	0.61	0.53	0.45	0.45
claude-sonnet-4-20250514	0.61	0.61	0.61	0.51	0.49	0.46

Table 10: Ablation Study of LLMs.

A.7 Evaluation Prompts

To conduct our evaluation, we designed prompts for each baseline according to the implementation design of different operators in each baseline. Here we present part of the necessary prompt for each.

PZ Prompts:

(1) convert

```
COT_QA_BASE_SYSTEM_PROMPT = """You are a helpful
     assistant whose job is to {job_instruction}.
You will be presented with a context and a set of
     output fields to generate. Your task is to
     generate a JSON object that fills in the output
      fields with the correct values.
You will be provided with a description of each
     input field and each output field. All of the
     fields in the output JSON object can be derived
      using information from the context.
{output_format_instruction} Finish your response
     with a newline character followed by ---
An example is shown below:
---
INPUT FIELDS:
{example_input_fields}
OUTPUT FIELDS:
{example_output_fields}
CONTEXT:
{example_context}
{image_disclaimer}
Let's think step-by-step in order to answer the
     question.
REASONING: {example_reasoning}
ANSWER:
{example_answer}
---
"""

COT_QA_BASE_USER_PROMPT = """You are a helpful
     assistant whose job is to {job_instruction}.
You will be presented with a context and a set of
     output fields to generate. Your task is to
     generate a JSON object which fills in the
     output fields with the correct values.
You will be provided with a description of each
     input field and each output field. All of the
     fields in the output JSON object can be derived
      using information from the context.
{output_format_instruction} Finish your response
     with a newline character followed by ---
---
INPUT FIELDS:
{input_fields_desc}
OUTPUT FIELDS:
{output_fields_desc}
CONTEXT:
{context}
<<image-placeholder>>
Let's think step-by-step in order to answer the
     question.
REASONING: """
```

(2) filter

```
COT_BOOL_BASE_SYSTEM_PROMPT = """You are a helpful
     assistant whose job is to {job_instruction}.
You will be presented with a context and a filter
     condition. Output TRUE if the context satisfies
      the filter condition, and FALSE otherwise.
Remember, your answer must be TRUE or FALSE. Finish
     your response with a newline character followed
      by ---
An example is shown below:
---
INPUT FIELDS:
{example_input_fields}
CONTEXT:
{example_context}
{image_disclaimer}
FILTER CONDITION: {example_filter_condition}
Let's think step-by-step in order to answer the
     question.
REASONING: {example_reasoning}
ANSWER: TRUE
---
"""

COT_BOOL_BASE_USER_PROMPT = """You are a helpful
     assistant whose job is to {job_instruction}.
You will be presented with a context and a filter
     condition. Output TRUE if the context satisfies
      the filter condition, and FALSE otherwise.
Remember, your answer must be TRUE or FALSE. Finish
     your response with a newline character followed
      by ---
---
INPUT FIELDS:
{input_fields_desc}
CONTEXT:
{context}
<<image-placeholder>>
FILTER CONDITION: {filter_condition}
Let's think step-by-step in order to answer the
     question.
REASONING: """
```

LOTUS Prompts:

(1) extract

```
prompt = "What" + {attribute} + "in {context}?" + {
     attribute's description} + "If there are
     multiple values, separate them with '||' and
     leave empty if not applicable. Please keep each
      extracted value concise and avoid lengthy
     content."
```

(2) filter

```
prompt = "{context}." + {attribute1's description} +
      {attribute2's description} + {filters}
```

DocETL Prompts:

A pipeline in DocETL consists of five main components:

- **Default Model:** The language model to use for the pipeline.
- **System Prompts:** A description of your dataset and the "persona" you'd like the LLM to adopt when analyzing your data.
  - Dataset Description: A concise explanation of what kind of data the model will be processing.
  - Persona: The role or perspective the model should adopt when analyzing the data.
- **Datasets:** The input data sources for your pipeline.
- **Operators:** The processing steps that transform your data.
- **Pipeline Specification:** The sequence of steps and the output configuration.
  - Steps: The sequence of operations to be applied to the data.
  - Output: The configuration for the final output of the pipeline.

Example Operators:

(1) Map Operator

```
- name: analyze_news_article
  type: map
  prompt: |
    Analyze the following news article:
    "{{ input.article }}"

    Provide the following information:
    1. Main topic (1-3 words)
    2. Summary (2-3 sentences)
    3. Key entities mentioned (list up to 5, with
     brief descriptions)
    4. Sentiment towards the main topic (positive,
     negative, or neutral)
    5. Potential biases or slants in reporting (if
     any)
    6. Relevant categories (e.g., politics,
     technology, environment; list up to 3)
    7. Credibility score (1-10, where 10 is highly
     credible)
  output:
    schema:
      main_topic: string
      summary: string
      key_entities: list[object]
      sentiment: string
      biases: list[string]
      categories: list[string]
      credibility_score: integer

  model: gpt-4o-mini
```

(2) Filter Operator

```
- name: filter_high_impact_articles
  type: filter
  prompt: |
    Analyze the following news article:
    Title: "{{ input.title }}"
    Content: "{{ input.content }}"

    Determine if this article is high-impact based
     on the following criteria:
    1. Covers a significant global or national event
    2. Has potential long-term consequences
    3. Affects a large number of people
    4. Is from a reputable source

    Respond with 'true' if the article meets at
     least 3 of these criteria,
    otherwise respond with 'false'.

  output:
    schema:
      is_high_impact: boolean

  model: gpt-4-turbo
  validate:
    - isinstance(output["is_high_impact"], bool)
```

(3) Reduce Operator

```
- name: summarize_feedback
  type: reduce
  reduce_key: department
  prompt: |
    Summarize the customer feedback for the {{
     inputs[0].department }} department:

    {% for item in inputs %}
    Feedback {{ loop.index }}: {{ item.feedback }}
    {% endfor %}

    Provide a concise summary of the main points and
      overall sentiment.
  output:
    schema:
      summary: string
```

```
      sentiment: string
```

(4) Equijoin Operator

```
- name: match_candidates_to_jobs
  type: equijoin
  comparison_prompt: |
    Compare the following job candidate and job
     posting:

    Candidate Skills: {{ left.skills }}
    Candidate Experience: {{ left.years_experience
     }}

    Job Required Skills: {{ right.required_skills }}
    Job Desired Experience: {{ right.
     desired_experience }}

    Is this candidate a good match for the job?
     Consider both the overlap
    in skills and the candidate's experience level.
     Respond with "True" if
    it's a good match, or "False" if it's not a
     suitable match.
  output:
    schema:
      match_score: float
      match_rationale: string
```

ZENDB Prompts:

(1) Syntax Tree Constructing

```
prompt = (
        "You are analyzing a document. "
        "Section headers in a document typically
     indicate the start of a new section or
     subsection, such as chapter titles, main
     headings, or important topic divisions. "
        "Section headers are often short,
     descriptive, and may be formatted differently
     from the main text. "
        "Given the following phrase from the
     document:\n\n"
        f'"{phrase}"\n\n'
        "Is this phrase a section or subsection
     header in the document? "
        "Answer only 'yes' or 'no'."
```

(2) Node Retrieval

```
prompt = f"""You are given a list of sections from a
      document like a Art artist, biographical
     article and so on, each with a title and a
     summary.
Your task is to select the section most likely to
     contain data for the table "{table_name}".
Table description: "{description}"
Table fields: {fields_str}

Sections:
{chr(10).join(node_info)}

Return ONLY the node ID (e.g., "doc\_0\_node\_5")
     that best matches the table content, or "NONE"
     if no match.
"""

prompt = f"""Check if this academic paper section
     might contain data about:
{conditions_text}

Section Title: {node.name}
Section Summary: {node.summary[:200]}
Section Content: {node.full_context[:300]}

Answer ONLY 'yes' or 'no'"""
```

(3) Filter

```
if operator in ['=', '==']:
    prompt = f"""Check if the {field} in this text
     equals "{value}":\n\n{node.full_context[:500]}\
     n\nAnswer ONLY 'yes' or 'no'"""
elif operator == 'LIKE':
    prompt = f"""Check if the {field} in this text
     contains or is similar to "{value}":\n\n{node.
     full_context[:500]}\n\nAnswer ONLY 'yes' or 'no
     '"""
elif operator == '>':
    prompt = f"""Check if the {field} in this text
     is greater than {value}:\n\n{node.full_context
     [:500]}\n\nAnswer ONLY 'yes' or 'no'"""
elif operator == '<':
    prompt = f"""Check if the {field} in this text
     is less than {value}:\n\n{node.full_context
     [:500]}\n\nAnswer ONLY 'yes' or 'no'"""
elif operator == '>=':
    prompt = f"""Check if the {field} in this text
     is greater than or equal to {value}:\n\n{node.
     full_context[:500]}\n\nAnswer ONLY 'yes' or 'no
     '"""
elif operator == '<=':
    prompt = f"""Check if the {field} in this text
     is less than or equal to {value}:\n\n{node.
     full_context[:500]}\n\nAnswer ONLY 'yes' or 'no
     '"""
elif operator == '!=':
    prompt = f"""Check if the {field} in this text
     is NOT equal to "{value}":\n\n{node.
     full_context[:500]}\n\nAnswer ONLY 'yes' or 'no
     '"""
else:
```

```
            return True
```

(4) Attribute Value Extraction

```
prompt = f"""
    You are analyzing a document. Please extract the
      content of the "{field}" field from the text.
    - If the field is not found in the text, return
     "NULL".
    The original text is as follows:
    {node.full_context[:1000]}
    Only return the content of the field itself, and
      do not include any additional information.
    """
```

QUEST Prompts:

(1) Sampling

User Prompt:

```
Your Task is to extract key-value pairs from text
     chunks with following guides:

1. Input:
        - Schema: Attributes to be extracted and
     their corresponding descriptions
        - Chunks: A list of text chunks to be
     extracted, each marked with its ID at the
     beginning.

2. Output:
        - `key`: lowercase attribute_name from
     schema (e.g.,name)
        - `value`: attribute_value with exact casing
     /spacing (e.g., iPhone 14)
        - `confidence`: int, between 0 to 100.
        - `chunkid`: int, id of the chunk from which
      the key-value pair is extracted.
        - Output one tuple per line, formatted as  (
     attr_name, attr_value, confidence, chunkid).

        Schema: ```{attr_schema}```

        Document: ```{doc}```
```

System Prompt:

```
You are an attribute extraction assistant. Only
     respond with (key, value, confidence, chunkid)
     pairs. Do not include any explanations or extra
      text.
```

(2) Extract

User Prompt:

```
Extract the following fields from the given document
     : {attributes}.

        Instructions:
        - Format your response as lines, each in the
      format: `field: value`
        - Use the exact field names: {attributes}
        - Follow the descriptions of the fields:
        ``` {related_attr_descriptions_str} ```
        - If a field is missing or unknown, leave
     its value empty (e.g., `team: None`)
        - use the line break (`\\n`)to split the
     lines
        - Do not add any extra text, comments, or
     explanations

        Document: ```{doc}```
```

System Prompt:

```
You are an information extraction assistant. Only
     respond with key-value pairs using the exact
     field names provided. Do not include any
     explanations or extra text.
```

(3) Filter

User Prompt:

```
Check if the value satisfy the condition or
     semantically similar to the condition.

        Instructions:
        - Format your response as a single line, in
     the format: `fcondition: True/False`.
        - Check if the value satisfies the condition
      {filters}.
        - The filter condition `==` or `IN` can be
     considered emantically for strings. For example
     , \'Lakers\' and \'Los Angeles Lakers\' are
     equal, \'fashion\' and \'Fashion ||
     Illustration\' are also equal.
        - The filter conditioin `<`, `>`, `>=`,
     `<=` can be considered as numeric comparison or
      a date comparison, note that the eariler date
     is smaller.
        - Do not add any extra text, comments,
     quotes, or explanations.

        Value: ```{value}```

        Condition: ```{filters}```
```

System Prompt:

```
You are an condition check assistant. Respond in a
     single line, includes a pair format as `
     fcondition: True/False`, the boolean True or
     False represents whether the condition is met.
     Do not include any explanations or extra text.
```

(4) Extract + Filter

User Prompt:
```

```
Extract the following field from the given document: {attributes}. Then Check if the value satisfy the condition.
   Instructions:
   - Format your response as two lines, the first line in the format: `field: value`, and the second line in the format: `fcondition: True/False`.
   - Use the exact field name: {attributes}.
   - Check the condition {filters}.
   - If the field is missing or unknown, leave its value empty (e.g., `team: None`), and leave the condition as False.
   - use the line break (`\\n`) to split the lines.
   - You should first extract the field value and then check. For example, we first do extract, and get filed is \'name\', value is \' Lee\'. Then we do check, the condition is \' name==\'Frank\' \', the value does not satisfy the condition, so the fcondtion is False.
   - The filter condition `==` or `IN` can be considered emantically for strings. For example , \'Lakers\' and \'Los Angeles Lakers\' are equal, \'fashion\' and \'Fashion || Illustration\' are also equal.
   - The filter conditioin `<`, `>`, `>=`, `<=` can be considered as numeric comparison or a date comparison, note that the eariler date is smaller.
   - Follow the descriptions of the field:
   ``` {related_attr_descriptions_str} ```
   - Do not add any extra text, comments, quotes, or explanations.

   Document: ```{doc}```
```

System Prompt:

```
You are an information extraction and check assistant. Respond in two lines, the first line is a key-value pair using the exact field name provided; the second line is a key-value pair, and value is a boolean True or False whether the condition is met. Do not include any explanations or extra text.
```

UQE Prompts:

(1) Extract Prompt

```
system_prompt["description"]["extract"] =
"You are an art analyst. The following paragraph " +
"provides some basic information about the artist. *info_intro* " +
"Please Help me to extract the value about *attrs_in_prompt*. " +
"You should present the result in the form of 'attr: value' and " +
"connect different pairs with separator '&&'. " +
"For example, 'attr1: 5&&attr2: public&&...'. If the value of the " +
"attr is not available, please return 'None'."
```

(2) Filter Prompt

```
system_prompt["description"]["filter"] =
"You are an art analyst. The following paragraph " +
"provides some basic information about the company. *info_intro* " +
"Please Help me to extract the information about *attr_name* " +
"and determine whether the it satisfies '*attr_name* *cond_op* *cond_val*'. " +
"If it does, please only return True. Otherwise, please return False. " +
"Please only return 'True' or 'False'."
```

(3) GroupBy Labels Split Prompt

```
system_prompt["description"]["groupby_labels_split"] =
"I will provide an introdution of the artist. *info_intro* " +
"Please help me to extract the attribute '*attr_name*'. " +
"You should only present the result, without any other information."
```

(4) GroupBy Class Prompt

```
system_prompt["groupby_class"] =
"I will provide you several paragraphs with their column names in the " +
"following conversation. " +
"Also, I will provide a list of attributes, including the possible " +
"categories for each attribute " +
"in the form of '<column>.<attribute>:<list of categories>'. " +
"The attributes and their candidate catetories are: *attr_with_labels*. " +
"Please extract the attribute from the corresponding paragraph and " +
"classify them into the given categories. " +
"You should present the classification result of each attributes in the " +
"form of 'attr1:category1&&attr2:category2&&...' " +
"If none of the given category is matched for the given attribute, " +
"please return 'attr:None' as its classfication result." +
"You should only return the results, without any other information."
```
```

(5) Merge Filter Prompt

```
system_prompt["merge_filter"] =
"I will provide you several paragraphs in the
     following conversation. " +
"Each paragraph is preceded by a name indicator. " +
"*col_list* " +
"Please help me to filter the paragraphs according
     to the given conditions. " +
"The conditions on a paragraph are presented in the
     form of " +
"'<name1>:[<condition1>,<condition2>,<condition3
     >,...]'. " +
"Each condition is presented in the form of <
     attr_name> <cond_op> <cond_val>. " +
"You should first extract the information of <
     attr_name> from the paragraph " +
"and then determine whether the condition is
     satisfied. " +
"The conditions are: *condition_list*. " +
"The explaination on the attributes included in the
     conditions are: " +
"*info_intro*" +
"You only need to give an overall decision on all
     these paragraphs and " +
"conditions. " +
"Return 'True' if all the conditions are satisfied "
      +
"and return 'False' if some of the conditions are
     not satisfied. " +
"Please only return the result 'True' or 'False',
     without other information."
```

(6) Merge Filter Aggregation Split Prompt

```
system_prompt["merge_filter_aggr_split"] =
"I will provide you several paragraphs in the
     following conversation. " +
"Each paragraph is preceded by a name indicator. " +
"Please help me to filter the paragraphs according
     to the given condition. " +
"The conditions to be satisfied are: *
     filter_condition* " +
"Each condition is presented in the form of (<name1
     >.<attribute1> <op> <value1>). " +
"The explaination on the attributes included in the
     conditions are: " +
"*info_intro*" +
"Return 'True' if all the conditions are satisfied "
      +
"and return 'False' otherwise. Please Only return '
     True' or 'False', " +
"without any other information."
```

Additional Indicators:

```
system_prompt["description"]["
     merge_filter_column_indicators"] = "The
     paragraph 'description' is a description of the
      artist in text form."

system_prompt["description"]["
     merge_filter_cond_indicators"] =
"The conditions on paragraph 'description' include:
     [*col_conditions*]"
```