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

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

ABSTRACT

Nowadays, the explosion of unstructured data presents immense analytical value. Leveraging the remarkable capability of large language models (LLMs) in extracting attributes of structured tables from unstructured data, researchers are developing LLM-powered data systems for users to analyze unstructured documents as working with a database. These unstructured data analysis (UDA) systems differ significantly in all aspects, including query interfaces, query optimization strategies, and operator implementations, making it unclear which performs best in which scenario. Unfortunately, there does not exist a comprehensive benchmark that offers high-quality, large-volume, and diverse datasets as well as rich query workload to thoroughly evaluate such systems. To fill this gap, we present UDA-Bench, the first benchmark for unstructured data analysis that meets all the above requirements. Specifically, we organize a team with 30 graduate students that spends over in total 10,000 hours on curating 5 datasets from various domains and constructing a *relational database view* from these datasets by manual annotation. These relational databases can be used as ground truth to evaluate any of these UDA systems despite their differences in programming interfaces. Moreover, we design diverse queries to analyze the attributes defined in the database schema, covering different types of analytical operators with varying selectivities and complexities. We conduct in-depth analysis of the key building blocks of existing UDA systems: query interface, query optimization, operator design, and data processing. We run exhaustive experiments over the benchmark to fully evaluate these systems and different techniques w.r.t. the above building blocks. The major outcomes of this project, including (1) a comprehensive benchmark that allows a rigorous evaluation of UDA systems and (2) a deep understanding of the strengths and limitations of existing systems, pave the way for future research of unstructured data analysis.

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 [12]. 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 unstructured data analysis 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, recommend public hospitals within certain distances of the patient's home that specialize in treating these diseases.

LLM-powered Unstructured Data Analysis. To support such needs, the database community is actively developing LLMs-based systems for UDA [7, 16, 17, 23, 29]. 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 also feature an optimizer that automatically transforms the 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 of a Comprehensive Benchmark. Clearly, different systems have different implementations of operators and query optimization strategies. It is unclear which system works best in which scenario. Furthermore, these systems all use small datasets and query workloads in their evaluation, making the results less convincing. For example, ZenDB [16] uses merely three datasets with 221 unstructured documents and 27 queries in total, and the ground truth is not publicly available. Palimpzest [17] provides a text document dataset with 1,000 short emails, and only one query is available on this dataset. DocETL [29] 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.

Therefore, a comprehensive benchmark is in desperate need to standardize the evaluation of LLM-powered UDA systems and guide the design of such systems.

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

(1) Dataset Volume. To thoroughly evaluate the efficiency and scalability of different methods, 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 LLMs.

(2) Dataset & Query Workload Variety. Evaluating the performance of UDA systems must take into account the properties of datasets and the query workloads. In general, important properties w.r.t. datasets include domains, data modality, whether the documents have a clear structure, etc. The query workloads should cover different combinations of analytical operators (e.g., filter, aggregation, join) as well as various selectivities of filters.
(3) Precise labels. A high-quality benchmark must provide precise labels as ground truth. With precise labels, the benchmark could provide an accurate evaluation with respect to the performance of different systems, regardless of their different system 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 the above approaches reliably, perform a thorough evaluation and conduct a detailed analysis of the results.
Our Proposal. We construct `UDA-Bench`–the first benchmark for UDA that meets all the above goals.
Key Insight: A Relational Database Review. To develop such a benchmark, the 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 view, 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.*

Guided by this insight, we construct the benchmark to achieve all the above goals. To achieve the dataset volume goal, we collect five 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 documents, which is **100**× more than the existing benchmarks. In terms of lengths of the documents, two of them have an average length of more than 10,000 tokens. In particular, on `Finance` dataset, the document length can be up to 838,418 tokens (≈100 pages).

For the second design goal of ensuring the diversity of the datasets and query workloads, we construct datasets from various domains, including healthcare, law, art, sports, and finance. Among them, art and finance include images and text. We construct a total of **240** queries, **10**× more than the current benchmarks. These queries are grouped into **5** categories: `Select`, `Select+Filter`, `Select+Aggregation`, `Select+Join`, and other complex queries that are combinations of at least 3 operator types.

For the third goal of precise labels, we define **147** meaningful attributes that can be extracted from the five datasets, including categorical, numerical, and string types, exceeding the existing benchmarks by **10**×. A team of 30 graduate students spends more than 10,000 hours on labeling, 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 behavior. Furthermore, we analyze the experimental 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 allows us to offer actionable insights.
Contributions. We make the following contributions:
(1) We present `UDA-Bench`, a comprehensive benchmark for unstructured data analysis that includes large-scale and diverse datasets, as well as a rich set of varied 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 five sets of unstructured data from diverse domains, including more than 100,000 documents. We construct 240 queries involving various analytical operators over the attributes defined in the database.
(3) We provide comprehensive, cross-validated relational tables as ground truth, annotated by 30 graduate students over 10,000 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

In this section, we present the architecture of a typical LLM-powered data analysis system, which in general consists of 4 key modules, i.e., *the query interface*, *logical optimization layer*, *physical optimization layer*, and *data processing layer*, respectively. As shown in Figure 1, the 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 an optimal execution plan, such as pushing down predicates or ordering the operators. Subsequently, there are typically multiple methods (e.g., using different types of LLMs) to implement each operator, resulting in various physical plans. The physical optimization layer alternatively selects the appropriate implements and produces an optimized physical execution plan. Finally, the data processing layer serves as the data foundation for the aforementioned optimizations. Beyond the raw data stored in distributed storage systems (e.g., Amazon S3), the systems often segment documents into chunks. These document chunks, along with images within documents, are subsequently transformed into embeddings, which are then loaded into a vector database. This infrastructure enables accurate and efficient retrieval to identify the information relevant to a query, ultimately speeding up the query execution and reducing the cost.
System Optimization Goals. To summarize, unlike traditional databases that focus mainly on optimizing latency, LLM-powered

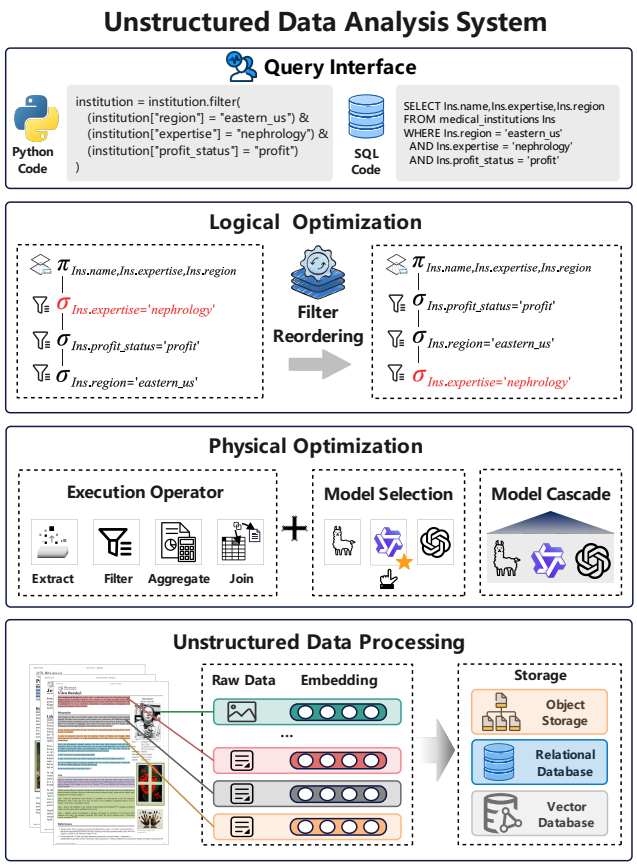

Figure 1: Architecture of Unstructured Data Analysis System.

UDA systems have multiple optimization goals, i.e., accuracy, cost, and latency. Firstly, accuracy is critical in such systems for two primary reasons. (1) LLMs are prone to hallucinations, leading to potential inference errors; and (2) to reduce costs, it is a common practice to only feed the LLM document chunks highly relevant to the analysis, rather than entire documents. However, if the retrieval misses relevant chunks or erroneously returns irrelevant chunks, it will degrade the analysis accuracy. Secondly, LLM inference consumes substantial computational costs, which is typically calculated based on the number of input and output tokens. In UDA tasks, outputs are generally succinct, making their token cost negligible. Consequently, reducing LLM cost 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. Finally, LLMs face high inference latency due to numerous parameters, making it crucial to minimize query latency in data analysis systems. Inference latency largely depends on the number of input tokens, so reducing these tokens can decrease query latency, with further improvements possible through strategies like parallel processing.

We detail how existing systems use four modules to achieve the stated goals. Table 1 shows each system's module composition.

2.1 Data Processing

In general, these systems are built to handle a wide variety of data types, including plain text, images, etc., which are extracted from complex documents such as PDF by OCR tools. Then, they organize these contents into different formats, which are further processed by different strategies to support downstream analytical tasks.

`LOTUS` [23], `UQE` [7] and `DocETL` [29] 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` [16] and `QUEST` [30] target analyzing plain text in documents, which is stored in a relational database. `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 title (i.e., a tree node) using the NLTK toolkit [18] and stores the summarization in the node. In addition, `ZenDB` calculates the embedding of each sentence for a subsequent cost-effective attribute extraction. The SHT is stored in a database table, where each row corresponds to a node, recording the document ID, node name, plain text, summarization, etc.

`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 using the NLTK toolkit and encodes it with the E5 model to build a document-level index. This index allows the system to quickly exclude documents that are irrelevant to the query. Then, it applies LangChain's *SemanticChunker* to split each document into semantically coherent chunks. That is, within each chunk, every two adjacent sentences have similar semantics, i.e., high similarity between their embeddings. `UDA-Bench` again uses the E5 model to embed these chunks and constructs a segment-level 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. The embeddings of both document summaries and chunks are stored in a vector database, while the corresponding text chunks and their metadata (e.g. plain text, embedding indices) are stored in a relational database.

`Evaporate` [2] only supports plain text as well, which is stored in a folder for subsequent analysis. `Palimpzest` [17] organizes the plain text, images each document into a specific directory, where distinct subfolders correspond to different types of data.

2.2 Query Interface & Operators

Query Interface. Each system provides a query interface for users to define analytical tasks on processed data. `UQE`, `ZenDB`, and `QUEST` use SQL-like language, using relational syntax to support analysis over unstructured data. For example, in the `Healthcare` dataset, to find private institutions specializing in nephrology and located in

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

Table 1: Overview of Existing Unstructured Data Analysis Systems.

the eastern United States, a user can write a query as shown in Figure 1. DocETL, Evaporate, LOTUS and Palimpzest offer declarative Python APIs, corresponding to the logical operators in relational databases, for users to compose a query as a Python program. For example, the user query described above could be represented using code as shown in Figure 1.

Query Operators. In UDA-Bench, we include 4 common analytical operators as below.

Extract aims to extract relevant attributes from a set D of unstructured data. Formally, Extract is defined as 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 of the institutions and their location, 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. Specifically, 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 Filter(D, C). Specifically, C defines a filter on document attributes, and evaluates whether the relevant attributes satisfy the filter for each document. For example, "lotus.sem_filter('the {document} satisfy {profit_status} is profit')" is a filter in LOTUS expressed with Python APIs. In general, existing systems adopt two strategies to implement such filters: (1) Palimpzest, QUEST, and ZenDB first extract the profit _status value from each document and then evaluate whether it satisfies the filter; and (2) LOTUS, DocETL and UQE take the filter as a part of prompts and leverage LLMs to determine whether the filter condition is satisfied.

Join is a cross-document operation in UDA, i.e., combining information from two sets of documents based on a specified join condition a. Formally, Join is defined as Join(D_1, D_2, a), where D_1 and D_2 are two subsets of documents, and a is the join attribute. This indicates that if the system extracts two tables (both contain the attribute a) respectively, 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. In this way, a user can easily identify possible diseases based on symptoms and then find medications that can treat those diseases through the join operation. ZenDB, 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 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. The 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. To achieve this, 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 evalute, potentially significantly 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. To be specific, given the attribute a of a filter F and an unstructured document $d \in D$, we use $d[a]$ 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. We use $c(d[a])$ to 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, ZenDB produces one single filter order with respect to all documents, similar to traditional databases. This is a coarse-grained optimization because different documents might contain different numbers of tokens with respect to an attribute.

Leveraging this optimization opportunity, QUEST produces different orders for different documents considering both the selectivity and the cost of every document, abandoning the above "one single order for one query" strategy. Therefore, QUEST does not have the $cost(F)$ for the overall document set, but instead, for each $d \in D$,it uses $cost_d(F) = c(d[a])$ to denote the number of tokens w.r.t. the attribute a in d. Then, each document d should follow 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, the QUEST optimizer uses the cost model discussed above to order these filters. In this way, QUEST might prioritize joins over filters to minimize the LLM cost, contradictory to the filter pushdown principle in traditional 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

As discussed above, each logical plan consists of a sequence of operators. Subsequently, the systems have to select an appropriate implementation w.r.t. each operator based on the properties of the data and user preferences, i.e., physical optimization. Next, we introduce the typical physical optimization strategies in UDA.

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 WikiArt 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 Llama 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.

Operator Decomposition. For each operator, DocETL pre-defines several possible decomposition strategies, each breaking down the operator into finer-grained steps to improve accuracy. For example, it uses two decomposition strategies to implement the operator $\texttt{Extract}(D, [a_1, a_2])$: (1) $\texttt{split}(D, k) \rightarrow \texttt{reduce}\{\texttt{Extract}(D_1, [a_1, a_2]), \cdots, \texttt{Extract}(D_k, [a_1, a_2])\}$, which splits D into k chunks and performs joint attribute extraction on each chunk. (2) $\texttt{Extract}\ (D, a_1)$, $\texttt{Extract}\ (D, a_2)$, which extracts a_1 and a_2 separately over the entire data set D without chunking. Then DocETL executes each strategy on a small validation set and uses a validation agent to evaluate and compare the quality of the output. The best-performing strategy is selected to replace the original operator in the pipeline.

Parallel Execution. Leveraging efficient batched inference with vLLM [14], LOTUS and DocETL process multiple documents concurrently, allowing efficient operator execution over large-scale document collections.

Dataset	#Attributes	#Files	Tokens (Max/Min/Avg.)	Multi-modal
WikiArt	19	1,000	1,665 / 619 / 789	✓
NBA	28	225	51,378 / 73 / 8,047	✗
LCR	19	566	45,437 / 340 / 5,609	✗
Finance	30	100	838,418 / 7,162 / 130,633	✓
Healthcare	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

Our benchmark consists of 5 datasets, which are NBA, WikiArt, Legal, Healthcare and Finance. We first collected and pre-processed the raw data. Then we followed a semi-automatic, iterative process

to define the attributes of the relational tables. After that, we spent a huge amount of human effort labeling the ground truth, i.e., the attribute values that could be extracted from the datasets, applying cross-validation and iterative prompt tuning methods. Finally, we manually designed 5 query templates w.r.t. each dataset and generated queries using Python scripts for benchmark evaluation.

We support the benchmark result that, each raw dataset, the benchmark associates it with a JSON file containing processed data, where different modalities within the same document are stored in a single field across various 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 five datasets with their statistics summarized in Table 2. Next, we describe these datasets below. For more details, please refer to Appendix.

NBA dataset is crawled from Wikipedia[20] that contains information about NBA including players, teams, team owners, etc., from the 20th century to the present, covering their basic and statistic information like player personal honors, team founding year, owner nationality etc.

WikiArt is collected from WikiArt.org [3], which covers artists and their artworks spanning from the 19th to the 21st centuries. For each artist document, it includes biographical information, artistic movement, a list of representative works, and high-resolution images of them as metadata.

Legal is sourced from AustLII [4] 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 [8], 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 [25], with numerous 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 three datasets, the NBA and WikiArt datasets are less complex due to their brevity and well-defined structure. Legal is more complicated because it is a domain-specific dataset containing multiple attributes that require semantic deduction to extract. Finance is another complex domain-specific dataset. The key challenge it introduces is the length of the documents, which can be up to 100 pages. Lastly, Healthcare has the largest number of documents, containing rich information, such as healthcare advertisements. Healthcare and NBA contain multiple categories of files, e.g., disease, drug, medical institution in Healthcare, which can support join queries. Finance and WikiArt are multi-modal datasets covering images. We define attributes over these images to verify the capability of systems in handling images.

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 [34] 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 NBA datasets, we divide the documents into multiple related categories but with different topics, each of which corresponds to a relational table. For Healthcare, originally the dataset contains 680,000 documents about heathcare information, from which we sampled 100,000 documents. Then, we leveraged LLMs to read these sampled documents and identified the three major categories of files (6,100 documents in total), i.e., disease, drug and medical institution. The remaining files are mostly categories like medical devices, health policy, etc. For NBA, we identify 4 related document sets with different categories (i.e., NBA players, NBA teams, team managers, cities).

3.3 Ground Truth Labeling

To label the ground truth, we first identify a number of significant attributes from each dataset or file category 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 significant attributes that pose different levels of challenges to extract. For example, the attribute Judge_name can be easily identified since it can usually be found at the beginning of each document in Legal. In Finance, the attribute values of Business_cost vary among different industries such as raw materials and wages for car manufacturers, versus product sourcing and logistics for supermarkets. Hence, a labeler has to determine the relevant costs according to the context, extract their values, and aggregate them. 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 correctly 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 10,000 human hours. Moreover, to ensure labeling quality, we also utilize multiple LLMs (e.g., Deepseek-V3, GPT-4.1, Claude-sonnet-4) to extract attributes and ask humans to double-check the manually labeled ground truth based on the results provided by LLMs. 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 in Section 3.2, Healthcare contains a large number of documents belonging to categories other than the three major ones mentioned above. Therefore, these documents rarely include the entities in the major categories. Consequently, we ask LLMs to analyze the 10,000 documents to label whether each of them belongs to the major categories (finally we obtain 6,100 documents).

If so, we ask humans to label the attributes; otherwise, the ground truth is NULL.

3.4 Query Construction

In general, we first ask human experts to design meaningful query templates, and then automatically instantiate these templates with different predicate values and join conditions to construct diverse queries. In total, we created 240 queries, including 220 single-table queries and 20 multi-table queries based on the templates over the 5 datasets.

To be specific, we ask the PH.D. students to design 5 query templates per dataset based on real-world scenarios. These templates are in the form of SQL-like queries and Python code, which can be utilized to test systems with different interfaces, like ZenDB and Palimpzest. We list all the query templates in Appendix. An example is shown as below.

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

A SQL Template Example.

```
disease_doc = disease
drug_doc = drug
disease_doc = Filter(disease_doc,
    disease.symptoms{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 Template Example.

The expert first identifies a real-life scenario (e.g., a user wants to identify disease based on his symptoms and find appropriate drugs for treatment), and then uses SQL-like queries and Python code to build a template and leaves some placeholders (i.e., {literal}, {group_by_attribute}). Next, we populate the placeholders according to their roles in the template to generate various queries. Taking the SQL-like query as an example: (1) For the SELECT clause, we randomly sample from all available attributes for the population. (2) The FROM clause does not need a population, as all relevant tables are specified in the template. For WHERE clause, we randomly select the filter attribute and operator (i.e., $\leq, =, \geq$) with equal probability when generating each query. Literal values are sampled to vary the selectivities. (4) For AGGREGATION clause, we use categorical attributes for grouping and numerical attributes for aggregation, and each aggregation operator (e.g., AVG) is randomly chosen with equal probability when constructing queries. (5) For JOIN clause, we explicitly define the join graph to guide the construction of queries. 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 NBA, we construct join paths such as Players $\bowtie$ Teams $\bowtie$ Managers, together with the specific join keys that link these tables.

Moreover, we varied the number of filters in WHERE clause to enhance the diversity of the templates. To be specific, we control the frequencies of queries with different numbers of filters ranging from 1 to 5, corresponding to 20%, 30%, 30%, 10%, and 10% percent of all queries, respectively. This allows us to thoroughly evaluate the systems with query in different levels of complexity. In addition, we define eight query types, ranging from simple Extract queries to more complex forms such as Extract + Filter + Agg, covering a broad range of real-world analytical scenarios.

4 EVALUATION

In this section, we evaluate existing systems on UDA-Bench and analyze the results, aiming to answer the following questions.
- **RQ 1**: What is the accuracy of different systems when evaluated 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 in such systems?
- **RQ 5**: How do different physical optimization strategies perform?

4.1 Experimental Settings

Systems for Evaluation. Our benchmark evaluates 7 existing unstructured data analysis systems as below. (1) Evaporate is a table extraction system. In our evaluation, 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 allowing users to execute queries by writing 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 also does not provide their code; therefore, we implement its Filter and Agg operators, as well as its logical optimizations, and then execute them on UDA-Bench. For a comprehensive evaluation, we adapted and modified these systems to support our evaluation (details are provided in Appendix). We also list all the evaluation prompts in Appendix.

Evaluation Metrics. Following existing works [16, 30], we measure accuracy, cost, and latency with respect to all queries. For accuracy, we follow [30] and report the average precision, recall, and F1-score across all queries. Given a query Q, the set of tuples returned by a method is denoted as $T(Q)$, and the ground truth is denoted as $GT(Q)$. For each element $t \in T(Q)$, we evaluate whether it can be matched to a corresponding tuple in $GT(Q)$. Therefore, we have $P = \frac{|T(Q) \cap GT(Q)|}{|T(Q)|}$, $R = \frac{|T(Q) \cap GT(Q)|}{|GT(Q)|}$, $F1 = \frac{2 \times P \times R}{P+R}$. For LLM costs, we report the average number of tokens (in thousands) per

document per query. For latency, we report the mean execution time in seconds per document per query.

Environment. We implemented all experiments in Python 3.7 and run experiments on an Ubuntu Server with four Intel(R) Xeon(R) Gold 6148 2.40GHz CPUs with 80 cores in total, two NVIDIA GeForce 4090 GPUs, 1TB DDR4 main memory, and 6TB SSD. The same environments ensure a fair comparison over different methods. 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 2 shows the effectiveness of extraction only queries. Almost all systems perform well on datasets `WikiArt` and `NBA`, with an F1-score around 0.85. This is because the two datasets are relatively short or contain easy-to-extract attributes. In particular, `DocETL` performs the best because it leverages multi-agent techniques to extract the attributes. `Evaporate` performs the worst because it uses LLM-generated code to extract data. However, the code essentially corresponds to a limited number of rules, which tend to be less effective when handling complex documents.

On datasets `Legal`, `Finance`, and `Healthcare`, which include long documents and multiple challenging attributes, the accuracy varies across different systems. Specifically, `LOTUS`, `UQE`, and `Palimpzest` achieve similar performance and perform the best, with an accuracy around 0.66 on `Legal`, 0.52 on `Finance` and 0.51 on `Healthcare`, as they all feed the entire document to LLMs and fully leverage the models' in-context reasoning abilities. This achieves more accurate answers. 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, chunking often misses relevant information, leading to lower accuracy. The exception is on `Healthcare`, which lacks structured information. Therefore, `ZenDB` feeds the entire document to LLMs, resulting in performance similar to that of `LOTUS`, `UQE`, and `Palimpzest`. All datasets do not perform well on `Healthcare` because many irrelevant information are extracted from the documents of other categories.

> Summary I: For simple datasets with short content and easy-to-extract attributes, almost all systems perform well. For complex datasets, the systems that feed entire documents to LLMs perform the best because chunk-based strategies may miss relevant information.

Figure 3 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 `WikiArt` and `NBA`, 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 extraction only queries. On the dataset `WikiArt` 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 the full 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 `NBA`, `Legal`, and `Finance` 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 the LLMs to analyze a small number of 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.

Table 4 shows the cost of filter queries. `UQE` incurs lower cost than other systems 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 to LLMs and using logical optimizations, such as prioritizing low selectivity and computational cost filters. Datasets

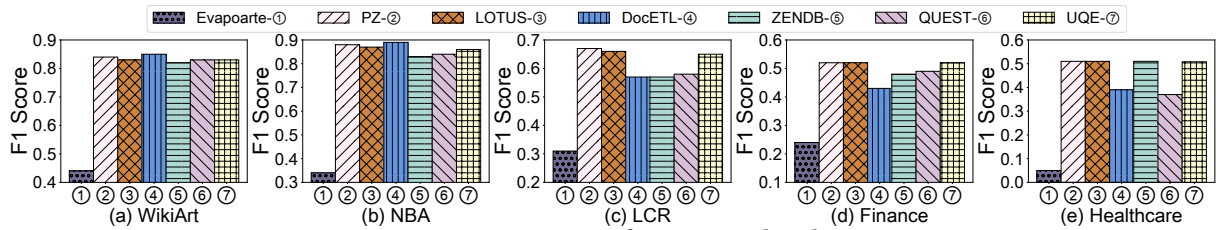

Figure 2: F1-score Comparison of Queries with only Extraction.

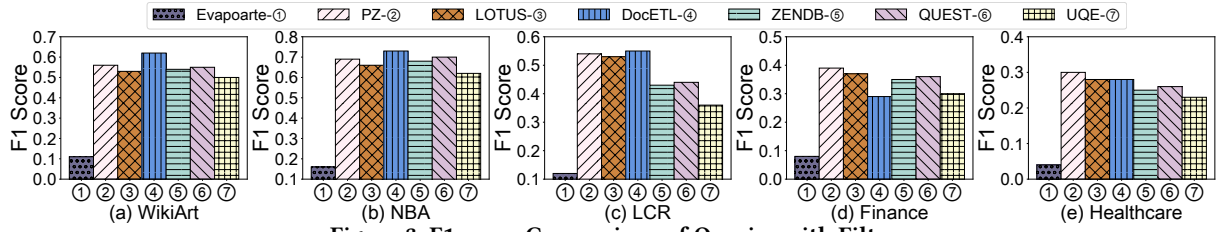

Figure 3: F1-score Comparison of Queries with Filter.

`WikiArt`, `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 V: 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 I: A promising direction is to investigate high-quality and cost-effective strategies for extracting difficult attributes in long documents. One key problem is to design a sophisticated chunking approach to produce chunks that contain precise information w.r.t. the to-be-extracted attribute.

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 time, 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 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.

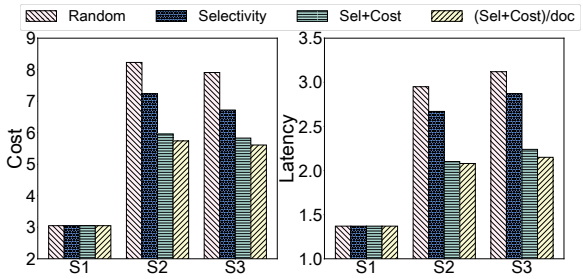

Figure 4: Evaluation of Filter Reordering.

> Summary VI: 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) `Selectivity`: 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 vary the number of filters: S1 with one filter, S2 with 2-3 filters, and S3 with 4 or more. We execute five queries in each of these three categories on `NBA`. In Figure 4, for queries in C1, the cost of all baselines is almost identical because there is only one filter and hence one order per query. For queries with more filters, these methods are ranked as follows by the LLMs cost: `(Sel+Cost)/doc` < `Sel+Cost` < `Selectivity` < `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.

Method	WikiArt		NBA		LCR		Finance		HealthCare	
	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency
Evaporate	-	0.16	-	0.30	-	0.25	-	3.50	-	0.52
Palimpzest	1.64	1.34	6.67	1.43	6.96	1.39	138.90	8.38	10.80	2.36
LOTUS	2.33	1.47	12.41	1.75	9.93	1.43	297.85	13.24	25.10	3.28
DocETL	7.04	15.86	55.53	79.05	154.26	270.88	818.10	1509.08	184.30	304.96
ZenDB	1.94	1.18	3.72	0.98	3.92	0.94	32.33	3.55	10.31	1.92
QUEST	1.20	0.89	2.06	0.92	3.09	0.89	29.30	2.65	4.70	1.72
UQE	0.92	0.83	5.96	1.03	6.12	1.04	124.02	5.83	10.09	1.91

Table 3: Cost and Latency Comparison for Extraction Queries.

Method	WikiArt		NBA		LCR		Finance		HealthCare	
	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency
Evaporate	-	0.16	-	0.30	-	0.25	-	1.50	-	0.52
Palimpzest	2.35	1.46	11.77	6.23	10.70	4.75	213.80	12.71	18.30	7.16
LOTUS	4.70	6.78	11.91	6.61	14.89	4.25	216.10	18.49	18.40	7.81
DocETL	6.15	13.71	52.92	74.86	32.70	31.75	184.06	14.67	113.35	47.06
ZenDB	3.51	2.13	9.59	4.56	26.69	27.72	49.18	4.73	19.50	2.28
QUEST	2.90	1.86	5.60	2.12	9.33	2.10	36.00	4.33	9.46	2.10
UQE	0.97	0.74	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.

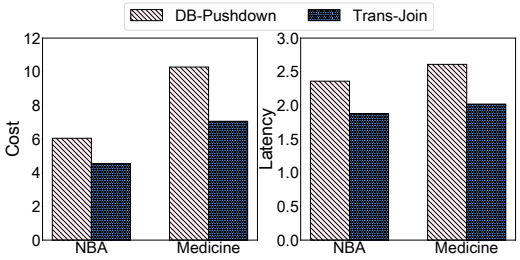

Figure 5: Evaluation of Filter Pushdown.

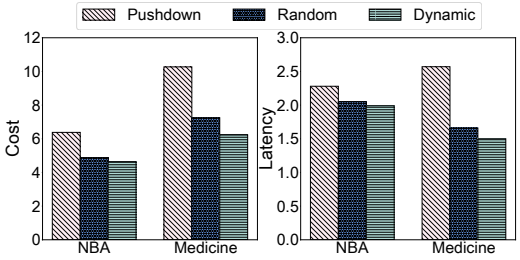

Figure 6: Evaluation of Join Order.

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 5 join queries with filters on `NBA`. We observe from Figure 5 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.

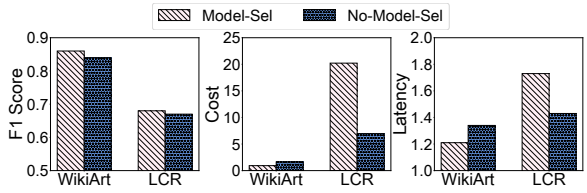

Figure 7: Evaluation of Model Selection.

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 five join queries involving more than three tables on `NBA`. We observe from Figure 6 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`, `GPT-4.1-mini`, and `GPT-4.1`) and set the selection objective as "minimize cost while achieving the target accuracy." (2) `no-Model-Sel`: The system always uses a specific model (`GPT-4.1-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 five queries with `GPT-4.1-mini` on each dataset (`WikiArt` and `Legal`). We evaluate both strategies using the same set of queries on each dataset, including both the simple (`WikiArt`) and challenging (`Legal`) datasets. The cost here is calculated by multiplying the number of tokens by the model's price per token. As shown in Figure 7,

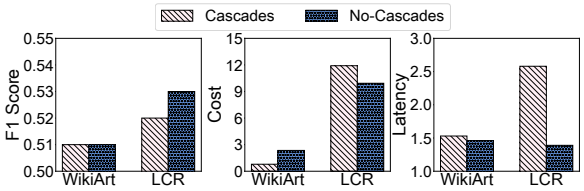

Figure 8: Evaluation of Model Cascades.

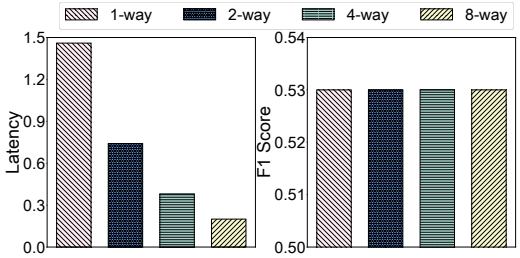

Figure 9: Evaluation of Parallel Execution.

on `WikiArt`, `Model-Sel` exceeds the target accuracy, while also reducing cost by using a cheaper model. This is achieved by assigning `GPT-4.1` to harder attributes and `GPT-4.1-nano` to easier ones, while `WikiArt` 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 `GPT-4.1`.

Model Cascades. We evaluate two strategies in `LOTUS` as follows. (1) `Cascades`: we construct a model cascade using two models, `GPT-4.1-nano` and `GPT-4.1-mini`. For each query, the system first uses the lower-cost `GPT-4.1-nano` to execute the queries. If the accuracy does not meet the desired accuracy, the query is then forwarded to the larger `GPT-4.1-mini` for further processing. (2) `No-Cascades`: The system uses only a single model, `GPT-4.1-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 8, on `WikiArt`, `Cascades` achieves the target accuracy while reducing cost by primarily using the less expensive `GPT-4.1-nano`. This is because, for most attributes in `WikiArt`, 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 `GPT-4.1-mini`, and all documents have to be first processed by `GPT-4.1-nano` before being fed to `GPT-4.1-mini`.

Parallel Execution. We evaluate two strategies on `Healthcare` using five queries as follows. (1) `Parallel`: employing parallel execution with different levels of parallelism, specifically with 2-way, 4-way, and 8-way thread parallelism. (2) `no-Parallel`: using a sequential approach without any parallelism. As shown in Figure 9, we observe that increasing the levels of parallelism leads to a proportional reduction in execution time. For example, with 2-way, 4-way, and 8-way thread parallelism, the execution time is reduced from 1.46 seconds (`no-Parallel`) to 0.74, 0.38, and 0.2 seconds, respectively, while the F1 score remains unchanged at 0.53.

Opportunity II: 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.

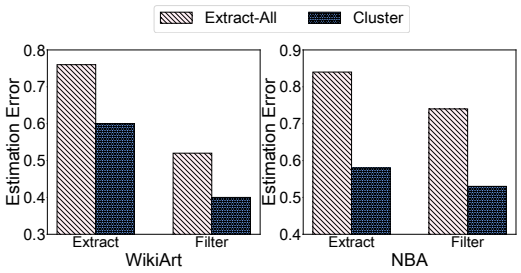

Figure 10: Evaluation of Aggregation.

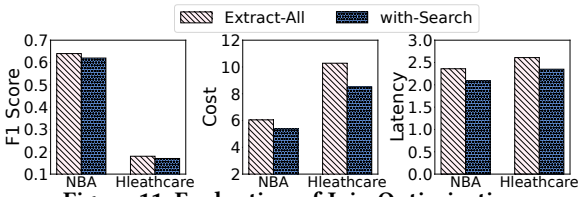

Figure 11: Evaluation of Join Optimization.

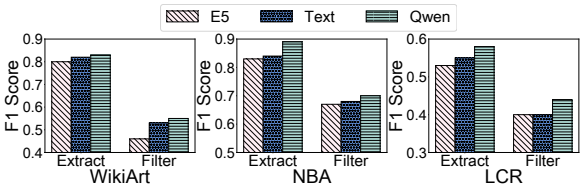

Figure 12: Ablation Study of Embedding Models.

Optimization for Aggregation. We use the estimation error as the metric following [7], defined as the absolute difference between the estimated and true values divided by the true value. We evaluate two strategies on `NBA` using five 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. We observe from Figure 10 that `Cluster` is more cost-effective than `Extract-All` because it does not need to extract the Groupby attribute, i.e., assuming the documents in each cluster share the same attribute value. However, the relative error is very high because it is hard to have a high-quality clustering simply based on the chunk embeddings.

Optimization for Join. We compare with two strategies on `NBA` using five queries as follows. (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 compare the above strategies using five join queries. We can observe from Figure 11 that `with-Search` is more cost-effective than `Extract-All` because many documents in the other table are pruned without extraction. However, the accuracy is low because it incorrectly prunes documents that can be joined.

Opportunity III: 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.

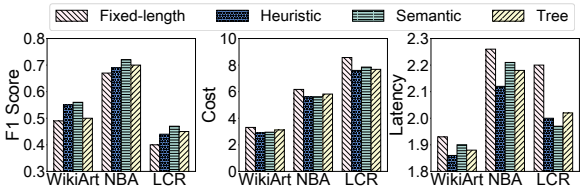

Figure 13: Ablation Study of Chunking Strategies.

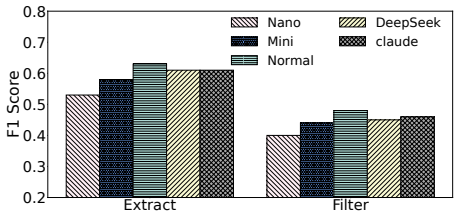

Figure 14: Ablation Study of Different LLMs.

4.7 Ablation Studies

Ablation Study of Embedding. We evaluate different embedding models to analyze whether using a better embedding model can improve the system performance. We utilize `multilingual-e5-large(E5)` [35], `text-embedding-3(Text)` [21] and `Qwen3-Embdding(Qwen)` [38]. On MTEB benchmark [19], 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 `WikiArt`, `NBA`, and `Legal` datasets, each with five queries each 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 various chunking strategies affect system performance. We use the following chunking strategies: (1) `Fixed-length` [5]: chunks are set to a fixed size of 512 tokens. (2) `Heuristic` [28]: grammar-based chunking with a maximum chunk size of 512 tokens. (3) `Semantic` (`QUEST`): semantic-based chunking with a maximum length of 512 tokens and a minimum length of 128 tokens. (4) `Tree` (`ZenDB`): an SHT tree to split the documents. Details for each chunking strategy are in Appendix. We test performance by replacing the chunking strategy in `QUEST` and conducting five queries on datasets `WikiArt`, `NBA`, and `Legal`. As shown in Figure 13, `Semantic` always performs the best since it captures the rich semantic representations and thus leads to a better embedding similarity than other strategies. `Tree` outperforms `Heuristic` on `NBA` and `Legal` because it preserves more tokens per chunk, providing richer in-context information, although this comes at a higher cost and latency. `Fixed-length` performs the worst, as it may split complete sentences into different chunks, resulting in incomplete information.

Ablation Study of Different LLMs. We evaluate the impact of different types of LLMs (`GPT-4.1-nano (Nano)`, `GPT-4.1-mini (Mini)`, `GPT-4.1` (Normal), `Deepseek-V3 (Deepseek)`, and `Claude-sonnet-4 (Claude)`) on system performance to assess whether stronger LLMs lead to improved results. We evaluate the performance of various LLMs by replacing the LLM API in `QUEST` and testing each model on five queries from the `Legal` dataset. Figure 14 presents the results, with performance ranked as: `GPT-4.1` > `Claude-sonnet-4` > `Deepseek-V3` > `GPT-4.1-mini` > `GPT-4.1-nano`, which closely aligns with the expected capability of these LLMs [1].

5 RELATED WORK

In Section 2, we have reviewed the key components of existing LLM-powered UDA systems. This section focuses on the pro and cons of these systems and analyzes their overall performance. In addition, we review techniques for data extraction, which is a key operation in such systems.

LLM-powered Unstructured Data Analysis Systems. Lotus [23] introduces semantic operators for unstructured data processing, including indexing, extraction, filtering, joining capabilities that enable the construction of complex analytical pipelines. It provides optimized physical implementation for each operator but lacks logical optimizations. Its experimental evaluation is limited to small datasets with few queries –five queries corresponding to five tasks. Palimpzest [17] 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. It optimizes the physical plan mainly by selecting LLMs for a task based on user preference. However, the dataset used in the evaluation is relatively small –1,149 documents in total. DocETL[29] focuses on improving the accuracy of UDA using a multi-agent strategy, but it does not conduct logical optimization to save cost. Its evaluation is performed on five datasets, with only one representative query per dataset. ZenDB[16] uses semantic hierarchical trees for identifying 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, with no publicly available ground truth. UQE [7] provides SQL-like analysis with sampling-based aggregation, while CAESURA [33] decomposes queries into operators handling data in different modalities. Both systems, however, are evaluated on limited-size datasets. For example, UQE provides one query and the dataset contains only 1,000 shot emails. Similarly, early systems [6, 11, 31, 32] 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.

LLM-powered Data Extraction. Extracting information rom unstructured sources has evolved from rule-based methods [15, 22, 26, 27] to modern deep learning approaches. OpenIE6 [13] employs iterative grid labeling for triple extraction, while MacroIE [37] uses BERT-based encoders to identify entity relationships. However, these methods struggle with implicit relationships and complex document structures. LLM-based systems seek to resolve these issues. `Evaporate` [2] generates extraction code through LLMs, balancing cost and quality through weak supervision.

Recent works focus on Pre-trained Language Models (PLMs) for data extraction, including DebertaV3 [9], which uses QA task pre-training for text information extraction, Text-to-Table[36] using a sequence-to-sequence model for text-to-table conversion, STable[24] with a permutation-based decoder for flexibility, and

ODIE [10] applying LoRA to fine-tune a LLaMA-7B model for unstructured text extraction. These techniques can enhance the cost-effectiveness and quality of UDA systems.

6 CONCLUSION

In this paper, we build a comprehensive benchmark for unstructured data analysis powered by LLMs. We collect five unstructured datasets with diverse characteristics, based on which we define a number of significant attributes and accurately label their values with human efforts. We also construct hundreds of meaningful queries with various analytical operators. Finally, we implement existing systems, run queries over the labeled datasets to test their performance and conduct in-depth analysis.

REFERENCES

[1] 2024. Vellum AI LLM Leaderboard. https://www.vellum.ai/llm-leaderboard. https://www.vellum.ai/llm-leaderboard.

[2] 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

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

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

[5] 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

[6] 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

[7] 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

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

[9] Pengcheng He, Jianfeng Gao, and Weizhu Chen. 2023. DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing. arXiv:2111.09543 [cs.CL] https://arxiv.org/abs/2111.09543

[10] Yizhu Jiao, Ming Zhong, Sha Li, Ruining Zhao, Siru Ouyang, Heng Ji, and Jiawei Han. 2023. Instruct and Extract: Instruction Tuning for On-Demand Information Extraction. arXiv:2310.16040 [cs.CL] https://arxiv.org/abs/2310.16040

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

[12] 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.

[13] 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

[14] 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.]).

[15] 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

[16] 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

[17] 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

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

[19] 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

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

[21] 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

[22] 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

[23] 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

[24] Michał Pietruszka, Michał Turski, Łukasz Borchmann, Tomasz Dwojak, Gabriela Nowakowska, Karolina Szyndler, Dawid Jurkiewicz, and Łukasz Garncarek. 2024. STable: Table Generation Framework for Encoder-Decoder Models. arXiv:2206.04045 [cs.CL] https://arxiv.org/abs/2206.04045

[25] 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

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

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

[28] 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.

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] Xueqing Wu, Jiacheng Zhang, and Hang Li. 2022. Text-to-Table: A New Way of Information Extraction. arXiv:2109.02707 [cs.CL] https://arxiv.org/abs/2109.02707

[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).