DocDB: A Database for Unstructured Document Analysis

Zequn Li*
zqli@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Yuanhao Zhong*
zyh04@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Chengliang Chai
ccl@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Zhaoze Sun
Yuhao Deng
sunzhaoze@bit.edu.cn
dyh18@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Ye Yuan
Guoren Wang
yuan-ye@bit.edu.cn
wanggr@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Lei Cao
caolei@arizona.edu
University of Arizona
Tucson, United States

ABSTRACT

Recent studies have developed LLM-powered data systems that enable database-like analysis of unstructured text documents. While LLMs excel at attribute extraction from documents, their high computational costs and latency make extraction operations the primary performance bottleneck. Existing systems typically adopt traditional relational database query optimization strategies, which prove ineffective in minimizing LLM-related expenses. To fill this gap, we propose DocDB, a prototype system that features a bunch of novel optimization strategies designated to unstructured document analysis. First, we employ a two-level index to reduce LLM extraction costs by selectively retrieving and processing only text segments relevant to target attributes. Second, DocDB employs adaptive execution, generating document-specific plans to minimize LLM extraction frequency based on varying per-document attribute extraction costs. With a real-life scenario, we demonstrate that DocDB allows users to analyze unstructured documents accurately and affordably using SQL-like queries. The corresponding video is available at https://youtu.be/8yDIKOBHIOg.

PVLDB Reference Format:
Zequn Li, Yuanhao Zhong, Chengliang Chai, Zhaoze Sun, Yuhao Deng, Ye Yuan, Guoren Wang, and Lei Cao. DocDB: A Database for Unstructured Document Analysis . PVLDB, 18(12): 5387 - 5390, 2025.
doi:10.14778/3750601.3750678

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wisdomzq/DocDB.

1 INTRODUCTION

Modern corporations host a large amount of unstructured data. Recent Large Language Model (LLM)-powered systems, such as ZenDB [4], Lotus [3], and Palimpzest(PZ) [2] , enable structured analysis by employing LLMs to extract user-specified attributes and applying database operations (e.g., filter, join, etc.). If successful, these efforts could transform unstructured data into actionable insights, such as enabling lawyers to quickly identify murder cases with at least three charges and 15-year sentences.

As opposed to query optimization in relational database that aim to minimize query execution time, query optimization in such LLM-centric data systems faces unique challenges, and hence raising research opportunities. LLM inferences are much more expensive in both execution time and monetary cost than traditional database operations, making attribute extraction the critical bottleneck. Therefore, the key optimization objective in this scenario is to *minimize the LLM cost incurred during extraction*, equivalent to: (1) minimizing the LLM cost of each extraction operation, which depends on the number of input tokens to an LLM; and (2) minimizing the frequency of invoking the data extraction operations. However, existing systems [2–4], focusing on improving the prompting strategies and selecting LLMs appropriate for a query, or directly applying existing optimizations in databases to reduce query latency, largely overlook these unique optimization opportunities.

Our Proposal. To achieve the above optimization objective, we propose a novel system, DocDB, which reduces LLM costs via index-driven attribute retrieval and adaptive document-specific optimization, enabling scalable analysis of unstructured data with diverse attributes. Overall, DocDB first executes an index-based attribute extraction strategy to minimize the number of input tokens per extraction and then applies an instance-optimized query execution strategy to minimize the frequency of attribute extraction. Specifically, DocDB consists of the following key modules.

Index-based Attribute Extraction. DocDB improves LLM efficiency and accuracy for attribute extraction using a two-level indexing strategy inspired by RAG. It first discards irrelevant documents entirely before locating specific text segments, a more accurate approach than segment-only systems. Furthermore, it overcomes weak queries by automatically learning the features of relevant segments from a document sample, which eliminates the need for manual prompt engineering and boosts retrieval performance.

Instance-optimized Query Execution. DocDB utilizes a dynamic, instance-optimized strategy that interleaves data extraction with

*Both authors contributed equally to this research. Chengliang Chai and Yuhao Deng are corresponding authors.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750678

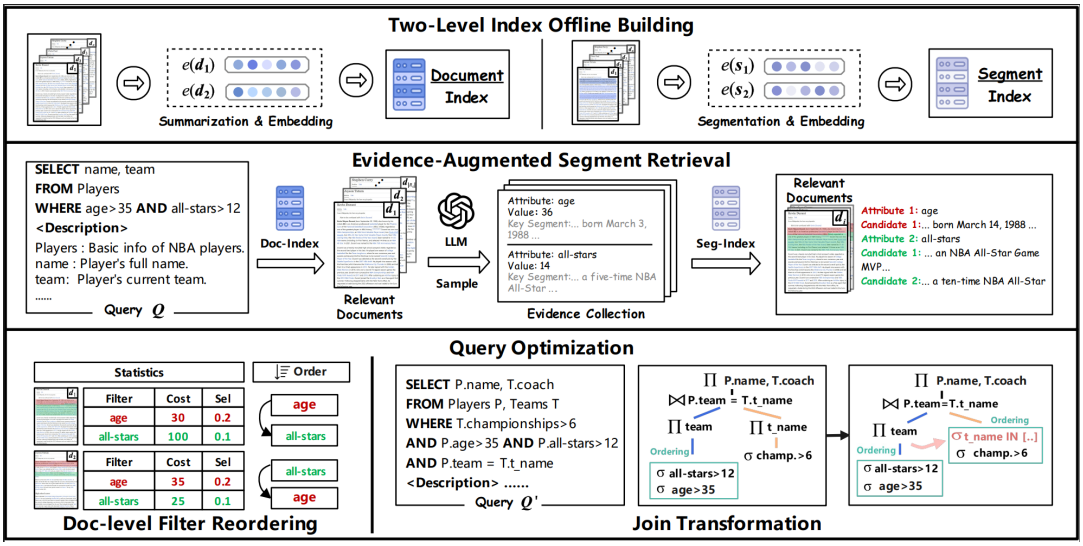

Figure 1: DocDB Framework

query operations to minimize costly extractions. Instead of creating a static plan, it generates an optimal, per-document execution plan at runtime, enabling adaptive techniques like instance-specific filter and join ordering. This approach is particularly effective for joins, where DocDB transforms them into specialized filters and dynamically reorders multi-way joins during execution.

Demonstration Scenarios. We propose to showcase DocDB's ability to effectively conduct unstructured data analysis with minimized LLM costs. Users can upload unstructured documents and write a SQL-like query that describes the analysis requirement. Then DocDB processes these documents to extract structured data, with the goal of minimizing the LLM costs and ensuring analysis quality.

2 SYSTEM ARCHITECTURE

2.1 User Queries

DocDB enables SQL-like queries, allowing users to select documents, extract attributes and perform analytical operations like filtering and joining. Specifically, DocDB targets optimizing Selection-Projection-Join (SPJ) queries over unstructured document set $\mathcal{D} = \{d_1, d_2, \cdots, d_n\}$. In this paper, A is the set of attributes in query Q, with each attribute denoted as a_i. The WHERE clause expression is ϑ, comprising multiple filters, each represented as θ_j. DocDB supports a broad range of filters. (1) DocDB supports queries that are conjunctions or disjunctions of any number of filters. (2) Any single filter for an attribute can be an equality filter or an open/close range filter. Furthermore, DocDB also supports join operations.

2.2 Overview

As shown in Figure 1, DocDB first constructs a two-level index. In response to a user's query and attribute selection, it samples documents to gather statistics for cost-based optimization. Unlike traditional databases that generate a static plan beforehand, DocDB dynamically optimizes the query plan for each document during execution, adapting to real-time statistics and varying LLM extraction costs. To achieve this, it employs lightweight yet effective strategies and an execution engine that integrates index-based extraction with analytical operations like joins for efficient query processing.

2.3 Stage 1: Index-based Attribute Extraction

Two-level Index Construction. Given a document set $\mathcal{D}$, DocDB first generates document summaries using NLTK, which are then embedded with a pre-trained model (e.g., E5Model). DocDB then dynamically divides documents into semantically coherent segments using LangChain's `SemanticChunker` function, ensuring each attribute is extracted from a single segment. These segments are then embedded and both document and segment embeddings are stored in high-dimensional vector indexes for efficient retrieval.

Searching the Index. For a given query Q, DocDB employs a two-level retrieval process, first returning a relevant document set D_Q and then identifying relevant segments. However, recognizing that simple embeddings for attributes in Q's attribute set A are often too uninformative to find key content, DocDB introduces an evidence-augmented strategy. It samples from D_Q and uses an LLM to extract actual attribute values and their corresponding text segments for the attributes in A. These segments, which capture common contextual patterns, are then used as "evidence" to enrich the original query. Subsequently, DocDB leverages this evidence to conduct a more accurate retrieval, combines the results, and uses an LLM to perform the final attribute extraction from the refined segments.

2.4 Stage 2: Query Optimization

As discussed in Section 1, LLM-based attribute extraction is DocDB's primary and most variable cost bottleneck, requiring a novel optimization strategy. Since the high costs of LLM inferences, traditional assumptions that joins are always more expensive than filters no longer hold. Therefore, we propose a unified approach to minimize

unnecessary extractions through both instance-specific filter ordering and re-optimizing joins to function as efficient filters. More details are discussed in the full version paper [1].

Filter Ordering. In Figure 1, taking query Q as an example, although `all-stars` has lower selectivity than `age`, LLM costs vary across documents. For d_1, DocDB prioritizes `age` due to the high extraction cost of `all-stars`. Conversely, for d_2, where `all-stars` is cheaper to extract, it takes priority over `age`. Therefore, DocDB optimizes filter ordering by dynamically adapting to per-document LLM token costs and filter selectivities(estimated during sampling the subset), prioritizing efficient extraction for conjunctive, disjunctive, and mixed queries. Formally, let $C_Q(o)$ represent the total cost of executing query Q on a document with a given filter order o. The objective is to find the optimal order o^* that minimizes this cost:

$$o^* = \arg\min_{o \in O} C_Q(o) \tag{1}$$

Next, we present our filter ordering method and analyze its optimality across conjunctive, disjunctive, and mixed filter combinations.
Conjunctions. Consider a query Q that includes a `WHERE` clause containing a conjunction of filters. Let $A_s \subseteq A, A_w \subseteq A$ denote the set of attributes that appear in the `SELECT` and `WHERE` clause respectively. For the i-th filter $o[i]$ in an order, the cost associated with extracting its relevant attribute is represented as $c^F[i]$, and its selectivity is given by $p[i]$. For each attribute $a_j \in A_s$, the generation cost is denoted by c_j^E. Therefore, the expected query cost for a given order o can be represented as follows:

$$C_Q(o) = \sum_{i=1}^{|o|} c^F[i] \prod_{j=1}^{i-1} p[j] + \Big(\sum_{j=1}^{|A_s|} c_j^E\Big) \prod_{i=1}^{|o|} p[i] \tag{2}$$

To find the optimal order with respect to each document, we prove that sorting filters in descending order based on the following priority score minimizes the expected query cost.

$$priority(\theta_k) = \frac{1 - p_k}{c_k}, \theta_k \in \vartheta \tag{3}$$

Therefore, for each filter $\theta_k \in \vartheta$, DocDB retrieves attribute-specific segments via indexing, estimates its extraction cost c_k (proportional to token count) and selectivity p_k (estimated on the sampled table), computes a priority score (Eq. 3), and sorts filters by descending score to determine the optimal execution sequence.
Disjunctions. For conjunctive queries, optimal filter ordering prioritizes conditions that are more likely to return `False`, whereas, for disjunctive queries, the strategy reverses to favor filters more likely to return `True`. Accordingly, we modify the cost model as below:

$$C_Q(o) = \sum_{i=1}^{|o|} c^F[i] \prod_{j=1}^{i-1} (1 - p[j]) + \Big(\sum_{j=1}^{|A_s|} c_j^E\Big)\Big[1 - \prod_{i=1}^{|o|} (1 - p[i])\Big] \tag{4}$$

Similarly, DocDB achieves optimal ordering by sorting filters in descending order using a slightly adjusted priority score from Eq. 3.

$$priority(\theta_k) = \frac{p_k}{c_k}, \theta_k \in \vartheta. \tag{5}$$

Conjunctions and Disjunctions. To handle mixed `AND/OR` queries, DocDB models the `WHERE` clause as an expression tree which is executed via a post-order traversal. The optimization strategy first decomposes this tree into order-invariant sub-expressions. It then uses dynamic programming to find the globally optimal execution order for these independent sub-expressions.

Query Optimization For Join. Next, we introduce the optimization techniques with respect to join queries.
Single Join: Joining Two Tables. Consider a query that joins two tables T_1 and T_2 on attribute a while applying filters θ_1 on T_1 and θ_2 on T_2. Taking a concrete query as an example, θ_1 filters `Teams` (T_1) with *championships* > 8, and θ_2 filters `Players` (T_2) with *age* > 35. A traditional optimizer applies these filters separately before joining the results. DocDB, however, transforms the join into an `IN` filter on T_2, restricting `team_name` in `[Lakers, Celtics]`. It then updates θ_2 to $\hat{\theta}_2$, which includes both conditions. This early filtering prunes most documents before applying *age* > 35, reducing LLM costs.

We establish a cost model to compute the expected cost under this optimization. C_1^i denotes the expected cost of executing operation θ_1 on the i-th document in table $T1$, p_1 represents the conditional probability of extracting attribute a after θ_1, and c_a^i corresponds to the attribute extraction cost for the i-th document. The expected cost of $\theta_2(T_2)$ is calculated in the same manner. The traditional relational method pushes θ_1 to T_1, θ_2 to T_2 and joins, so the expected cost under the optimal order can be calculated as below:

$$Cost(\theta_1(T_1) \bowtie \theta_2(T_2)) = \sum_{i=1}^{|T_1|} C_1^i + p_1 \sum_{i=1}^{|T_1|} c_a^i + \sum_{i=1}^{|T_2|} C_2^i + p_2 \sum_{i=1}^{|T_2|} c_{a'}^i \tag{6}$$

As discussed above, DocDB's optimal plan is to sort join and filters together. Since the join can be converted into an `IN` filter on either T_1 or T_2, DocDB establishes two cost models to determine the optimal choice. Specifically, if we push θ_1 to T_1 and transform the join to filter on T_2(Plan ①), the cost is estimated as below:

$$Cost(\theta_1(T_1) \rightarrow \hat{\theta}_2(T_2)) = \sum_{i=1}^{|T_1|} C_1^i + p_1 \sum_{i=1}^{|T_1|} c_a^i + \sum_{i=1}^{|T_2|} \hat{C}_2^i \tag{7}$$

where $\hat{C}_2^i$ represents the optimal expected cost obtained by sorting θ_2 with join on the i-th document in T_2. Similarly, we compute the cost that pushes θ_2 to T_2 and transforms the join to filter on T_1(Plan ②). As the selectivity of θ(`IN`) is low, the third term remains small. Therefore, if a plan minimizes the first two terms, its overall cost is likely lower, i.e., if $\sum_{i=1}^{|T_1|} C_1^i + p_1 \sum_{i=1}^{|T_1|} c_a^i < \sum_{i=1}^{|T_2|} C_2^i + p_2 \sum_{i=1}^{|T_2|} c_{a'}^i$, we choose Plan ①, otherwise we choose Plan ②.
Adaptive Join Ordering. DocDB dynamically orders multiple joins by iteratively building a left-deep plan. At each stage, it executes only the single, lowest-cost join, which it identifies by efficiently estimating costs through `IN`-filter conversion.

3 DEMONSTRATION SCENARIOS

The demonstration scenarios target showcasing the key functionality of our DocDB system: *extracting structured tables under different SQL statements with minimal LLM cost.*

(1) Single Table Query. DocDB supports SQL statements with `SELECT`, `FROM` and `WHERE` clauses for a single table. The user can first upload document set and write an SQL-like query that specifies some attributes and the combinations of filters. Then, DocDB constructs the two-level index, samples some documents, extracts attribute values and collects query optimizer statistics. Subsequently,

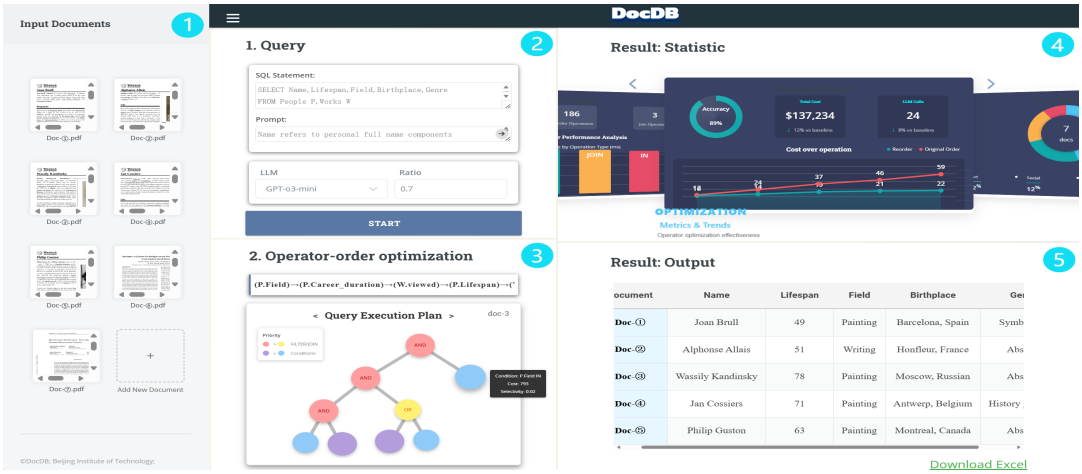

Figure 2: A Running Example of DocDB

DocDB produces query plans to order the filters at a *document by document* base. Figure 2 shows the details of our demonstration.

i). Dataset. The user can provide different types of documents. These documents are typically unstructured or semi-structured. Specifically, the user can click the "plus" button to upload new documents and previews the documents before uploading.

ii). Write SQL-like query. As shown in Figure 2 - ② , the user can first write a query with multiple filters. Then, she can describe each attribute, which will be used as prompts for subsequent extraction. The user interface allows for the selection of an LLM and a sampling rate. Upon clicking START, DocDB first samples a subset of documents at the specified rate. This sample is crucial for two purposes: collecting statistics for the query optimizer and gathering evidence for retrieval. The system then proceeds to extract and analyze attributes based on its instance-specific query plans.

iii). Filter ordering for optimization. The user can clearly view the optimal query plan in Figure 2 - ③ . For each document, DocDB generates an expression tree from the WHERE clause filters. It then uses collected statistics to recursively compute a priority score for each node, which dictates the execution sequence during a post-order traversal. This dynamic ordering, visualized by a color code (e.g., purple before blue), ensures higher-priority operations run first, with the final optimal execution order displayed to the user.

(2) Join Query. DocDB also allows the user to write a join query. After reconfiguring settings and clicking on START, DocDB will automatically produce query plans and transform joins into filters to further minimize the execution cost. The user can click on join operation nodes (i.e., the corresponding non-leaf node) in Figure 2 - ③ to explore the detailed join optimization process.

Result. In Figure 2 - ④ , the interface provides a comprehensive view of execution statistics. Key metrics such as extraction accuracy, total cost, and LLM calls are displayed, alongside a line chart that contrasts the cumulative cost of the optimized plan with the original order. Other visualizations break down the execution by tracking the frequency and time of filter, join, and reorder operations, and showing the distribution of extracted document types. The extracted attributes are presented in a structured table (Figure 2 - ⑤). Users can click on any value to trace it back to its source, which highlights the relevant text segment in the original document. The entire table of extracted values can also be downloaded.

	PZ	ZenDB	Lotus	DocDB
Accuracy	0.43	0.45	0.45	**0.63**
Latency(s)	2.85	2.73	3.36	**2.68**
#-Token Cost	2610	2530	12480	**2030**

Table 1: Performance Comparison of Different Systems

We evaluate DocDB against three LLM-driven systems (PZ, ZenDB and Lotus) using the LCR dataset, which contains 3,000 Australian court cases. The comparison focuses on three key metrics: accuracy (average precision), cost (tokens consumed per document), and latency (execution time per document). As shown in Table 1, DocDB outperforms all baseline systems across all three metrics.

ACKNOWLEDGMENTS

Chengliang Chai is supported by the NSF of China (62472031), the National Key Research and Development Program of China (2024YFC3308200), Beijing Nova Program, CCF-Baidu Open Fund (CCF-Baidu202402), and Huawei. Yuhao Deng is supported by the NSFC (624B2023) and the BIT Research and Innovation Promoting Project (2024YCXZ004). Ye Yuan is supported by the Beijing Natural Science Foundation (L241010), the National Key Research and Development Program of China (2022YFB2702100), and the NSFC (61932004, 62225203, U21A20516). Guoren Wang is supported by the NSFC (62427808, U2001211), and the Liaoning Revitalization Talents Program (XLYC2204005). Lei Cao is supported by the NSF (DBI-2327954) and Amazon Research Awards.

REFERENCES

[1] [n.d.]. https://anonymous.4open.science/r/QUEST/Full_version.pdf
[2] Chunwei Liu, Matthew Russo, and Michael J. et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. *CIDR* (2025).
[3] Liana Patel and Siddharth Jha et al. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. *CoRR* (2024).
[4] Deng Zheye and Chan Chunkit et al. 2024. Text-tuple-table: Towards information integration in text-to-table generation via global tuple extraction. *EMNLP* (2024).