Indexing Long Documents for LLM-Based Analysis

Donna Pham
University of Michigan
Ann Arbor, Michigan, USA
phdonn@umich.edu
Supervised by Lin Ma, University of Michigan

ABSTRACT

Long documents such as clinical records, legal contracts, and scientific papers are increasingly analyzed with large language models (LLMs). Naturally, feeding the full document to the model for every question can eventually become slow, expensive, prone to hallucination, and it reuses no work across questions. We explore an indexing-based solution for document analysis and propose a hierarchical plain-text index that is built once per document and consulted by subsequent queries. Inspired by the classic B+ tree, the index organizes a document into pages arranged from general summaries at the root to specific ones at the leaves, but it departs from the B+ tree in three ways suited to LLM access: content lives at every level, the index is plain text rather than attribute values, and navigation follows relevance between page summaries rather than comparing a search key. Its structure is discovered per document by the LLM rather than being hand-designed. In a preliminary evaluation on the NarrativeQA dataset, the index reaches accuracy comparable to DocETL, the strongest baseline, while being 40% cheaper to answer questions.

VLDB Workshop Reference Format:
Donna Pham. Indexing Long Documents for LLM-Based Analysis. VLDB 2026 Workshop: VLDB Ph.D. Workshop.

1 INTRODUCTION

Long-document analysis is an important but expensive task. Clinical records, legal contracts, and scientific papers are routinely examined in depth, and analyzing them by hand is slow and labor-intensive. The expense grows further when the same document must be analyzed many times: a patient record is consulted by clinicians over time, a contract is revisited across many matters, and a scientific paper is queried for many different research questions. A recent and promising line of work uses large language models (LLMs) to do this automatically, answering the free-form questions that previous keyword and extraction tools could not [13, 16].

When the LLM is prompted with the full document on each question, the cost of answering grows linearly with the size of the document. Long contexts add a problem of their own: as the input grows, the model fabricates answers more often at a rate that climbs steeply with context length, even when the answer is present in the document [12]. DocETL [16], together with related declarative pipelines such as Lotus [13] and Palimpzest [11], makes the analysis more capable through agentic query rewriting and chunked execution. These systems are stateless: they maintain no state between queries, processing each question from scratch over the original document that result in two consequences follow. First, optimizing a single pass offers only limited acceleration. Second, the full cost recurs on every question, since nothing is amortized across them.

A natural optimization is to precompute some intermediate state once and reuse it across questions. Two families of approaches do this. **The first builds a vector index over the document.** Retrieval-augmented generation [9] splits the document into chunks and embeds each as a vector; GraphRAG [5] and LightRAG [7] add graph structure and community summaries on top for questions that span the whole document. The shared limitation is the vector index itself: splitting fragments connections between distant passages and top-k retrieval surfaces only a few, so whole-document questions return incomplete context; the fixed-dimensional embedding is lossy; and the index is tied to one model, so it must be rebuilt whenever that model changes. **The second family extracts chosen attributes into a structured table.** Information-extraction systems such as Evaporate [1], Doctopus [3], and ZenDB [10] do this; the result is precise for filter-style lookups, but cannot answer open-ended natural-language questions, and the attributes must be fixed before anyone knows what will be asked.

In this paper, we explore a different path. *We return to the successful principles from the database literature, where answering many queries over a large body of data without scanning it is precisely the problem that the index, and in particular the B+ tree [2, 4], has solved for decades.* We read each document once and compile it into a **hierarchical, plain-text index that any LLM can read directly**: a few high-level summaries at the root, topic-level summaries beneath them, and detailed page summaries at the leaves. The index is built once per document and consulted by every later analysis, exactly as a database index is built once and reused by every query.

This keeps the reuse benefit shared by vector indices and structured extraction while avoiding the weakness of each. Like a vector index, it is built once and reused, but because it is plain text it is model-independent and preserves the cross-document connections that chunking destroys. Like structured extraction, it imposes organization on the document, but because that organization is discovered by the LLM rather than fixed in a schema, it answers deep, open-ended questions rather than only filter-style lookups. Each node is a short summary sized to fit comfortably within the model's context window.

Although the index borrows the B+ tree's shape, hierarchical access over pages, it departs from it in three ways suited to LLM access. Content lives at every level, not only at the leaves, so many

Proceedings of the VLDB Endowment. ISSN 2150-8097.

questions are answered from an internal summary without descending further, much as a covering index answers a query from the index alone without touching the base table [6]. The index is plain text rather than embedded vectors, so any model can read it and context is preserved by construction. Navigation is semantic: the model judges which summaries are relevant rather than comparing a search key, so the same index serves open-ended analysis rather than only exact-match or range queries.

In this work we explore the design of this new kind of index: how its structure is discovered from a corpus rather than hand-designed, how each document is compiled into it, and how questions are answered against it and the index refined over time.

Section 2 examines the limits of current practice in more detail. Section 3 presents the index and its three design components. Section 4 reports a preliminary evaluation on NarrativeQA, and Section 5 concludes.

2 RELATED WORKS

We group existing work on LLM-based long-document analysis into three families and point out where each is limited.

The first family answers each question by reading the document directly, keeping nothing from one question to the next. The simplest version feeds the whole text to the model along with the question. More developed pipelines decompose the work: DocETL [16] breaks the analysis into a sequence of operators over chunks and uses agentic rewriting to improve accuracy, with follow-ups extending this line to interactive steering [17] and multi-objective rewrites [18]; related declarative systems such as Lotus [13] and Palimpzest [11] optimize similar pipelines. These are flexible and need no setup, but reuse nothing across questions, so cost grows with each question and the long context buries relevant details, raising hallucination on long inputs [14].

A second family avoids reprocessing by building reusable intermediate state. Retrieval-augmented generation [9] splits the document into chunks, embeds each as a vector, and at query time retrieves the chunks nearest the question. GraphRAG [5] and LightRAG [7] add structure over the chunks—an entity graph with community summaries in the former, a lighter dual-level graph in the latter—for broad questions that draw on many parts of the document at once. Closest to our design, RAPTOR [15] recursively clusters and summarizes chunks into a tree of summaries much like ours. All share two limitations: the index is tied to one embedding model, so changing either forces a rebuild; and chunking and top-k retrieval are lossy, breaking links between distant passages and returning only a few chunks, so whole-document questions are easily missed. RAPTOR is no exception, since its tree is built bottom-up by clustering and queried by similarity, making its nodes retrieval targets rather than a plain-text schema that can be read and navigated directly.

A third family extracts chosen attributes into a structured table ahead of time, as in Evaporate [1], Doctopus [3], and ZenDB [10]. Querying is then a structured lookup, which is fast and precise for questions that can be written as filters or aggregations over known fields. However, there are two main limitations: the schema is usually fixed before the workload is known, so attributes that no one anticipated are simply absent; and a structured table can be a poor fit for free-form questions, such as "why a character betrays a partner", which call for synthesis across the document rather than simple value lookups.

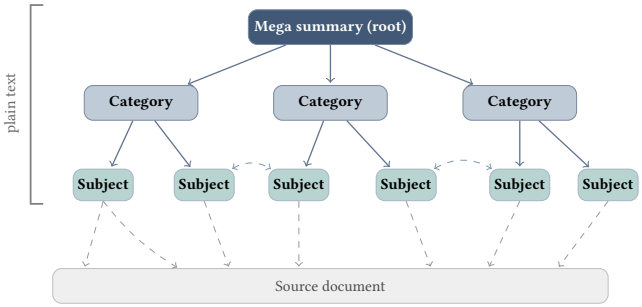

Figure 1: Proposed hierarchical plain-text index. A query enters at the root and follows the relevant pages, moving down toward specifics or hopping across to related subjects. Each subject links back to one or more spans in the source document.

3 METHOD

3.1 Overview

Our proposed index is a hierarchical plain-text structure compiled once per document (Figure 1). At the top, a *root* (the mega summary) gives an overview of the document and points to the broad areas beneath it, called *categories*. A category may stand on its own or split into sub-categories, so the index has one or more layers depending on how the document divides. Under the lowest layer sit *subjects*, the specifics, such as a character or an event in a screenplay, each recording a small set of *fields* (the details about that particular character or event) filled in directly from the document. Every page, root, category, or subject, is plain text and small enough to fit within an LLM's context window.

The remaining sections work through the three design problems: how the structure is decided (Section 3.2), how a document is compiled into it (Section 3.3), and how questions are answered against it (Section 3.4).

3.2 Discovering the structure

For each source document, the LLM reads it and proposes a *schema* for the index: a hierarchical structure of categories and fields. At the top, a brief overview known as the *mega summary* sits at the root. Beneath it, the LLM organizes the document into categories, and under each category, into subjects with a small set of fields (the kinds of details to record about each entry). On a movie screenplay, for example, the schema might propose a movie-wide overview at the root, with categories *characters*, *events*, and *facts* beneath it; subjects under *characters* would record fields such as *role*, *actions*, and *relationships*. We do not dictate which categories or fields the model should produce; both are proposed by the model from what it sees in the document.

The subjects and the values that fill their fields are left to be determined during compilation (Section 3.3). In the current implementation, we keep each subject small with a fixed cap per field. A more principled mechanism would bound the number of links at

each layer, analogous to the fanout of a B+ tree, so lookup stays efficient as the index grows; designing such a mechanism is a direction we leave to future work.

3.3 Compiling a document

Compilation reads the document in full and fills in the schema from discovery. It runs in three steps: clean, extract, and write.

Cleaning. The raw text is stripped of download artifacts and markup, then split into chunks of about 30,000 characters that overlap by 2,000, so that a passage at the edge of one chunk also appears in full in the next.

Extraction. The schema from discovery is handed to the LLM as part of the extraction prompt: it tells the model which categories to look for and which fields to fill in under each. For every chunk, the model reads the text against this schema and returns the subjects it finds (the characters, events, and facts that fit each category) with their fields filled in. The categories themselves never change; only the list of subjects under each one grows. For example, in a screenplay, if chunk 1 mentions two events, the *events* category gains two subjects; if chunk 2 mentions one more, it gains another. We call the running record of all categories and the subjects under them the *merged record*. For populating fields, the first non-empty answer is kept; if a later chunk supplies a different one, both are retained so nothing the document says is quietly lost. Each extracted field is also tagged with the source offset of the passage that produced it, so any value on a subject can be traced back to the original text.

Writing. Writing produces one file per subject grouped under its category, and a single overview file at the root summarizing the schema, matching the hierarchy in Figure 1.

3.4 Answering queries

To answer a question, the system navigates the index. Each subject carries a short summary; navigation scores these summaries against the question to decide where to go next. It starts at the mega summary (root) and follows the most relevant link, descending into categories, jumping across to related subjects, or stepping back up when an area is exhausted.

Descending into the index is unbounded: the system may go as deep along a promising path as the index allows and visit neighboring pages. What is bounded is the number of times it may level up to revisit a higher node and then descend into a different branch, so traversal cannot wander indefinitely across the index. We currently set this branching budget to a fixed value, the same for every question; deciding it more intelligently, for instance from the complexity of the question or the size of the index, is left to future work. When the pages reached do not contain the answer, the system falls back to the most relevant part of the source document and answers from there. The index is not updated on a miss; folding fallback results back into the index is a direction we touch on in Section 5.

4 PRELIMINARY EVALUATION

4.1 Experimental Setup

We evaluate on the NarrativeQA dataset [8], using ten movie screenplays and the 295 questions paired with them; the screenplays run to hundreds of pages and the questions are free-form, which is close

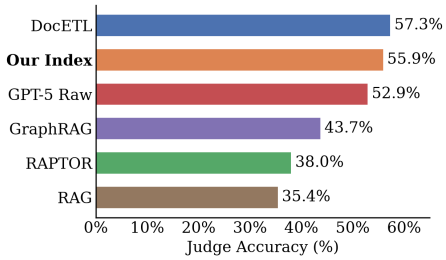

Figure 2: Judge accuracy on NarrativeQA. Our index is close to the DocETL pipeline and ahead of the retrieval baselines.

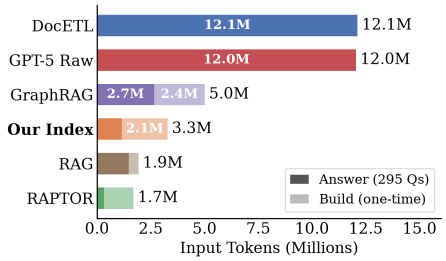

Figure 3: Total input-token cost of each system, split into one-time build (faded) and recurring per-query-batch answer (solid), for 295 questions over ten movies. Our index pays a one-time build cost and a low marginal answer cost.

to the setting we target. GPT-5 serves as both the compiler and the question-answering model, and an LLM judge scores answer equivalence against the reference answers.We compare against a full-document baseline (the entire script fed to the model on each question), the DocETL pipeline [16], and three retrieval baselines (RAG [9], GraphRAG [5], and RAPTOR [15]).

We report two measures. **Accuracy** is the fraction of answers the LLM judges equivalent to the reference. **Cost** is the total number of input tokens a system consumes, which we separate into a one-time index build cost and the recurring cost of answering each question. All numbers are measured from a complete, automated run of each system, with no manual intervention or hand-tuned estimates.

4.2 Results

Figure 2 gives the accuracy ranking. Our index reaches 55.9% judge accuracy, essentially matching DocETL, the strongest baseline, at 57.3%; the gap is about one and a half points, roughly four questions out of 295. It also outperforms feeding the full script to the model (52.9%). The three retrieval baselines trail: GraphRAG reaches 43.7%, RAPTOR 38.0%, and RAG 35.4%. They struggle on long screenplays because they keep only a small slice of the document. RAG's chunked retrieval returns just a few passages per question, and GraphRAG's extractor produces a sparse graph (about 63 nodes per film with 88% of them disconnected), so the parts of the document a free-form question depends on are usually missing. For RAPTOR we apply our system's strict rule: it must return *not found* when the retrieved nodes or top-k chunks lack the answer, rather than guessing from context, so all systems are judged on equal footing.

Figure 3 breaks each system's cost into the recurring per-query answer (solid) and the one-time build (faded). Our index spends

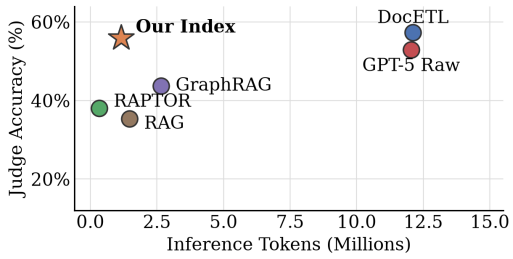

Figure 4: Answer cost versus judge accuracy on NarrativeQA (ten screenplays, 295 questions). Our index (star) reaches accuracy close to the most accurate baseline at a lower cost.

2.1M tokens to build the index once and 1.2M to answer all 295 questions, a marginal cost of roughly 3.9K tokens per question. DocETL and GPT baselines carry no build cost, but their entire 12M tokens is recurring, about 41K tokens per question for DocETL. Because the build cost is paid once and amortized across every later question, our total of 3.3M is about 73% below DocETL's 12.1M and 73% below the full-document baseline's 12.0M. Figure 4 places each system on the answer cost vs. accuracy plane; we count only inference tokens here and exclude the one-time build cost so per-query economics are directly comparable. Our index sits near the top of the accuracy range while spending fewer inference tokens than any other system: at 1.2M it answers all 295 questions for less than even the retrieval baselines, including RAPTOR's 1.7M, while reaching nearly the same accuracy as DocETL at a fraction of its per-question cost.

5 CONCLUSION AND FUTURE WORK

We explored an indexing-based solution for long-document analysis and proposed a compiled, hierarchical, plain-text index: built once per document, organized like a B+ tree but with content at every level, with its structure discovered from the corpus rather than hand-designed. On NarrativeQA, the index matches the accuracy of the strongest stateless baselines at 40% lower cost, while remaining model-independent and able to answer open-ended questions. These results are preliminary and confined to narrative text, but they suggest the design is worth pursuing.

Currently, we keep pages small with fixed caps on how much each page records, the same for every document, and we do not bound the number of pages or balance the hierarchy. A more principled scheme would derive these limits from a single constraint, how much context the question-answering model can use before its accuracy degrades on long inputs [12]. The index would then be built by a single recursive rule and fit a node's content on one page if it can. Otherwise the node should split into children and apply the same rule to each, so the depth grows only as far as the content forces rather than being fixed in advance. This raises further questions we want to study: how many layers a document of a given size should induce, how to choose the summary at each internal node so it guides navigation without duplicating the pages below it, and how to bound the children of a node so lookup stays efficient as the index grows. A separate direction is whether the index should change while it is being queried: today it is fixed once compiled and simply reports a miss when a question falls outside it, but a more capable system would re-read the source on a miss, fold what it finds back into the index, and eventually grow new structure toward the questions actually being asked, without bloating back into a copy of the full document. The work we are undertaking next is to solve these questions, while validating the approach across more domains and recalibrating the judge for each.

REFERENCES

[1] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *Proceedings of the VLDB Endowment* 17, 2 (2023), 92–105. https://doi.org/10.14778/3626292.3626294

[2] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1 (1972), 173–189.

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

[4] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[5] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. *arXiv preprint arXiv:2404.16130* (2024).

[6] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402. https://doi.org/10.1561/1900000028

[7] Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. 2025. LightRAG: Simple and Fast Retrieval-Augmented Generation. *arXiv preprint arXiv:2410.05779* (2025).

[8] Tomáš Kočiský, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. 2018. The NarrativeQA Reading Comprehension Challenge. *Transactions of the Association for Computational Linguistics* 6 (2018), 317–328.

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 9459–9474.

[10] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeighami, Aditya G. Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. *arXiv preprint arXiv:2405.04674* (2024).

[11] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *15th Annual Conference on Innovative Data Systems Research (CIDR)*.

[12] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[13] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4171–4184. https://doi.org/10.14778/3749646.3749685

[14] Jesus Vicente Roig. 2026. How Much Do LLMs Hallucinate in Document Q&A Scenarios? A 172-Billion-Token Study Across Temperatures, Context Lengths, and Hardware Platforms. *arXiv preprint arXiv:2603.08274* (2026).

[15] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. 2024. RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval. In *The Twelfth International Conference on Learning Representations (ICLR)*.

[16] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *arXiv preprint arXiv:2410.12189* (2025).

[17] Shreya Shankar, Bhavya Chopra, Mawil Hasan, Stephen Lee, Björn Hartmann, Joseph M. Hellerstein, Aditya G. Parameswaran, and Eugene Wu. 2025. Steering Semantic Data Processing with DocWrangler. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[18] Lindsey Linxi Wei, Shreya Shankar, Sepanta Zeighami, Yeounoh Chung, Fatma Ozcan, and Aditya G. Parameswaran. 2026. Multi-Objective Agentic Rewrites for Unstructured Data Processing. *Proceedings of the VLDB Endowment* (2026).