Towards Accurate and Efficient Document Analytics with Large Language Models

Yiming Lin[1], Madelon Hulsebos[1], Ruiying Ma[2], Shreya Shankar[1], Sepanta Zeighami[1], Aditya G. Parameswaran[1], Eugene Wu[3]
[1]UC Berkeley, [2]Tsinghua University, [3]Columbia University
{yiminglin,madelon,shreyashankar,zeighami,adityagp} @berkeley.edu
mry21@mails.tsinghua.edu.cn, ewu@cs.columbia.edu

ABSTRACT

Unstructured data formats account for over 80% of the data currently stored, and extracting value from such formats remains a considerable challenge. In particular, current approaches for managing unstructured documents do not support ad-hoc analytical queries on document collections. Moreover, Large Language Models (LLMs) directly applied to the documents themselves, or on portions of documents through a process of Retrieval-Augmented Generation (RAG), fail to provide high-accuracy query results, and in the LLM-only case, additionally incur high costs. Since many unstructured documents in a collection often follow similar templates that impart a common semantic structure, we introduce ZenDB, a document analytics system that leverages this semantic structure, coupled with LLMs, to answer ad-hoc SQL queries on document collections. ZenDB efficiently extracts semantic hierarchical structures from such templatized documents and introduces a novel query engine that leverages these structures for accurate and cost-effective query execution. Users can impose a schema on their documents, and query it, all via SQL. Extensive experiments on three real-world document collections demonstrate ZenDB 's benefits, achieving up to 30× cost savings compared to LLM-based baselines, while maintaining or improving accuracy, and surpassing RAG-based baselines by up to 61% in precision and 80% in recall, at a marginally higher cost.

1 INTRODUCTION

The vast majority—over 80%—of data today exists in unstructured formats such as text, PDF, video, and audio, and is continuing to grow at the rate of over 50% annually [2, 8]. In fact, an overwhelming 95% of businesses have recognized management of this unstructured data as a significant problem [1]. Consider *unstructured text documents*, such as Word or PDF documents, with a rich treasure trove of untapped information. Due to the inherently free-form nature of natural language, coupled with visual formatting, real-world unstructured documents pose a particularly difficult challenge for data management. *Is there any hope for successfully querying or extracting value from unstructured documents?*

Example 1.1 (Civic Agenda Report: Vanilla LLMs and RAG). Our journalism collaborators at Big Local News at Stanford have collected large tranches of civic meeting agenda PDF reports for various US counties as part of their agenda watch project, as in Figure 1-a, and want to analyze these reports. One such query could be to count the number of construction projects of a certain type, across meetings. To do so, one could use Large Language Models (LLMs). However, even advanced LLMs, such as GPT-4, struggle with queries issued on such reports (e.g., $Q1$ in Figure 1-d),

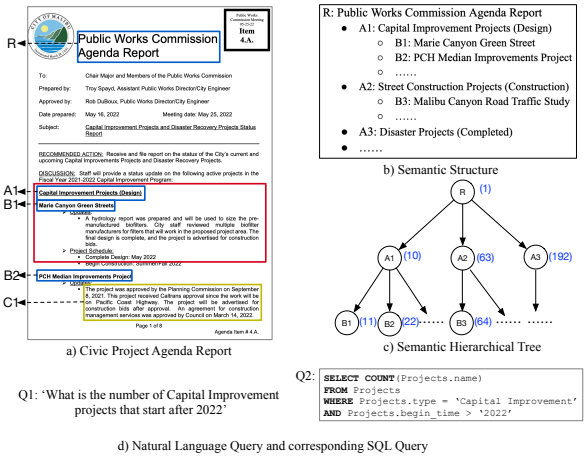


Figure 1: Civic Agenda Document and Semantic Structures.

especially when these queries involve aggregations and/or multiple filters on long documents. The error-prone nature of LLMs is not surprising given that LLMs can't effectively handle large contexts [19, 45], or complex data processing tasks [48, 49]. The costs of processing all documents in a collection via LLMs (e.g., through OpenAI APIs) are also high. Another strategy, Retrieval-Augmented Generation (RAG) [39, 41], identifies one or more text segments within each document that are most relevant (e.g., via embedding distance) to the given query, incorporating these segments into prompts, reducing the cost. However, RAG struggles to identify the appropriate text segments, even for simple queries. Suppose we want to identify the capital improvement projects. RAG retrieves the segments that most closely matches "capital improvement projects" within the document, such as the red box in Figure 1-a. However, it fails to capture over 20 additional projects in subsequent pages, such as the "PCH Median Improvement Project" (B2 in Figure 1-b) belonging to "Capital Improvement Projects" (A1). Overall, both the vanilla LLM approach and RAG are unsuitable: both have low accuracy, while the LLM approach additionally has high cost.

Leveraging Semantic Structure Helps. The reason RAG didn't perform well above was because the text segment provided to the LLM did not leverage the semantic structure underlying the document. Instead, if we are aware of this semantic structure, we can identify the capital improvement projects (A1 in Figure 1-b) by checking all of the subportions (e.g., B1, B2) under it, where each one corresponds to the description of such a project, and provide this

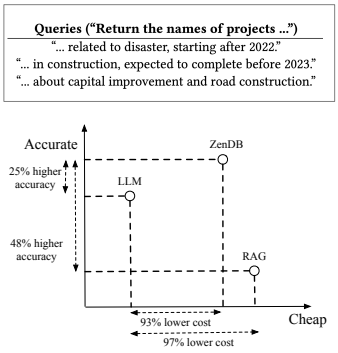
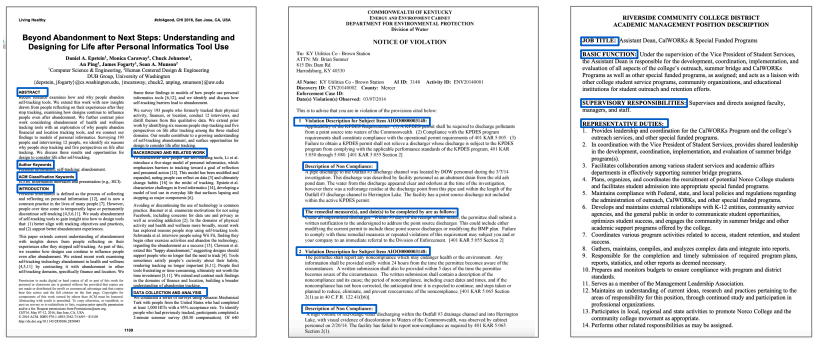


Figure 2: Understanding the differences between ZenDB, LLMs and RAG.



a) Scientific Papers b) Notice of Violations c) Employee Job Descriptions

Figure 3: Templatized Documents: Scientific Papers, Notice of Violations, Job Descriptions.

to an LLM to interpret. By doing so, we *provide all of the pertinent information to an LLM, unlike RAG, while also not overwhelming* it with too much information. Indeed, when we leverage semantic structure for a group of sample queries on GPT-4-32k, as in our system ZenDB, described next, we surpass the vanilla LLM and RAG approaches ***by 25% and 48% in accuracy, while only having 7% of the cost*** of LLMs, as detailed in Figure 2.

Templatized Documents Provide Semantic Structure. Given that semantic structure is helpful, *how do we extract this semantic structure within unstructured documents?* Turns out, while unstructured documents vary considerably in format, many documents that are part of collections are created using templates, which we call *templatized documents*. Templatized documents are observed across domains, including civic agenda reports, scientific papers, employee job descriptions, and notices of violations, as listed in Figure 1 and Figure 3. For instance, two scientific papers from the same venue use similar templates, just as civic documents for the same purpose from the same local county often adhere to a uniform template. Templatized documents often exhibit consistent visual patterns in headers (e.g., font size and type), when describing content corresponding to the same semantic "level" (e.g., section headers in a paper often follow the same visual pattern.) We highlight the "templates" using blue boxes in Figure 3. Thus, templatized documents are often have a discernible hierarchical structure that reflects different semantic levels within the document. For example, a 9-page complex civic agenda report (such as Figure 1-a) can be broken down into portions (e.g., A1, A2, A3 in Figure 1-b) and further into subportions (e.g., B2), indicating a possible semantic hierarchy, such as Figure 1-c, across the documents following the same template.

Leveraging Semantic Structure: Challenges. Unfortunately, the semantic structure of the templates isn't known—and neither do we expect these templates to be rigidly adhered to, nor do we expect there to just be one template across the collection of documents from a specific domain. Uncovering possible common semantic structures across documents is a challenge. In addition, to support queries over unstructured data where there isn't a predefined schema, it's not entirely clear what the data model or query interface should look like. Furthermore, using LLMs for query evaluation incurs high monetary costs and latencies; it's not obvious how we can leverage the semantic structures across documents to enable accurate query execution with low cost and latency.

Addressing Challenges in ZenDB. We introduce ZenDB, a document analytics system that supports ad-hoc advanced SQL queries on templatized document collections, and address the aforementioned challenges. First, we introduce the notion of *Semantic Hierarchical Trees (SHTs)* that represent the semantic structure for a given document, and effectively act as an index to retrieve only portions of the document that are pertinent to a given query. We build SHTs across documents by leveraging the uniform visual patterns in the document templates. We cluster the visual patterns found across documents to extract and detect various template instantiations, coupled with minimal LLM calls for this purpose. We show that if documents obey a property we term *well-formattedness*, then our procedure correctly recovers their semantic structure. Second, we introduce an extension to SQL to query unstructured documents (e.g., $Q1$ in Figure 1 could be expressed as a SQL query $Q2$.) Users can easily impose a schema on a collection of documents by simply listing a table name as well as a description for the entities in the table, without listing the attributes, which can then be lazily defined and populated in response to queries. Finally, we introduce a novel tree search algorithm that leverages SHTs to minimize cost and latency while answering queries without compromising on quality. Specifically, we propose a summarization technique to create summary sketches for each node within the tree. ZenDB can navigate through the tree, identifying the appropriate node to answer a given query by examining these sketches, akin to how a person might use a table of contents to find the right chapter for a specific task.

Other Related Work. Supporting queries on non-relational data isn't new. For unstructured data, the field of Information Retrieval (IR) [37, 53] investigates the retrieval of documents via keyword search queries, but doesn't consider advanced analytical queries. For semi-structured data [15, 16, 47], query languages like XQuery or XPath, as well as extensions to relational databases for querying XML and JSON, help query hierarchically organized data, as in our SHTs, but there, the hierarchy is explicit rather than implicit as in our setting. Recent efforts have sought to bridge the gap between structured queries, like SQL, and unstructured documents. One line of work [58, 61] has explored the upfront transformation of text documents into tables. Doing this ETL process with Large Language Models (LLMs) like GPT-4 on entire documents is expensive and

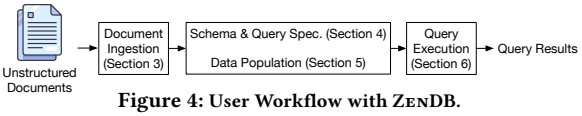


Figure 4: User Workflow with ZenDB.

error-prone relative to approaches that focus the LLM's attention on specific semantic portions, as we saw above. Others [24, 55, 56] have explored writing SQL queries directly on text data, as part of multi-modal databases. Most work there boils down to applying LLMs to the entire document, and only works well on simple, small documents. However, using these methods on complex, large documents we saw above leads to high costs and reduced accuracy. None of the approaches above have explored the use of semantic structure to reduce cost and improve accuracy when querying documents. We cover this and other related work in Section 8.

We make the following contributions in this paper, as part of building ZenDB, our document analytics system.

- We identify that we can leverage templates within document collections to support ad-hoc analytical queries.
- We introduce the notion of Semantic Hierarchical Trees (SHTs) that represents a concrete instantiation of a template for a specific document, as well as novel methods to efficiently extract SHTs from an array of templatized documents.
- We develop a simple extension to SQL to declare a schema, specify attributes on-demand, and perform analytical queries.
- We design a query engine that leverages SHTs, facilitating query execution in a cost-effective, efficient, and accurate manner.
- We implement all of these techniques within ZenDB and evaluate its performance on three real-world datasets, demonstrating substantial benefits over other techniques.

2 USER WORKFLOW WITH ZENDB

In this section, we present an overview of user workflows with ZenDB, as illustrated in Figure 4. First, ① document collections are ingested into the system by understanding common semantic structure (Section 3). Then, ② users (typically database administrators) can specify a schema for these documents, including tables and lazily-specified attributes, followed by queries that reference this schema, either specified by end-users who know SQL, or generated by applications (Section 4). ZenDB also populates upfront a set of system-defined tables/attributes to help capture the mapping between tuples and the documents (Section 5). Finally, ③ given queries on these documents, either generated by applications or by end-users directly, ZenDB will execute them efficiently, leveraging the semantic structure (Section 6).

① Semantic Structure Extraction. Given a collection of templatized documents that adhere to one or more predefined semantic structures, the first step within ZenDB involves extracting this structure in the form of Semantic Hierarchical Trees (SHTs), per document, so that they can be used downstream for query execution. This is broken down into two sub-problems: First, how do we extract an SHT from a single document? Second, how do we leverage common semantic structure across documents to scale up SHT extraction? Since templatized documents typically display consistent visual patterns in headers for similar semantic content, we cluster based on such visual patterns, coupled with minimal LLM invocations, to construct a single SHT (Section 3.2). Then, we use a visual pattern detection approach to determine whether we can reuse a previously identified semantic structure in the form of a template, synthesized from a concrete SHT, or extract a new one (when there are multiple templates in a collection), all without using LLMs (Section 3.3).

```
CREATE TABLE Projects WITH DESCRIPTION "The projects table contains
the description for a set of civic agenda projects."

ALTER TABLE Projects
      ADD name TEXT WITH DESCRIPTION "Name of Project",
      ADD type TEXT WITH DESCRIPTION "Type of Project",
      ADD begin_time DATE WITH DESCRIPTION "Begin time of Project";
```

Figure 5: Creating the Projects Table and Adding Attributes.

② Schema/Query Specification and Table Population. Given one SHT per document, ZenDB then enables users to specify a schema across documents in a selection, followed by issuing queries on that schema. Schema definition happens via an extension of standard SQL DDL: users (typically database administrators) provide a name and description for each table—that we call *document tables*, along with names, types, and descriptions for any attributes; the attributes can be lazily added at any point after the table is created (Section 4.1). For example, Figure 5 shows the query used to create a "Projects" table along with attributes (e.g., name). Subsequently, other users can write queries that reference such tables and attributes (e.g., *Q2* in Figure 1), as in standard SQL (Section 4.2); these queries could also be generated by applications (including form-based or GUI-based applications), or by translating natural language queries into SQL. We still concretize the query in SQL to provide well-defined semantics.

While attributes are added lazily and attribute values are computed or materialized in response to queries, we proactively identify mappings between tuples and documents during schema specification (Section 5). Specifically, we identify the SHT node that represents the portion of the document that captures all of the relevant tuples in a given user-specified table, as well as the mapping between tuples to individual SHT nodes, if they exist, using a combination of minimal LLM invocations and automated rules. These are then stored in our data model as hidden system-defined attributes, such as the span of the text that corresponds to the given tuple, leveraging nodes in the SHTs built earlier. These system-defined attributes allow for LLMs to extract the user-defined attribute values per tuple as needed, while reducing costs, while also leveraging the shared semantic structure across documents.

③ Query Execution. Finally, ZenDB executes the user-specified SQL queries using the pre-constructed SHTs per document, while minimizing cost and latency, and maximizing accuracy. Unlike traditional relational databases, where I/O and sometimes computation are often the bottleneck, here, the LLM calls invoked by ZenDB becomes both a cost and latency bottleneck. Therefore, ZenDB aims to minimize such calls, while still trying to extract attribute values as needed to answer queries, by using a combination of predicate pushdown and projection pull-up. We additionally develop a cost model for ZenDB, focusing on monetary cost (Section 6.1). Our cost model design is flexible and can be adapted to optimize for latency instead, e.g., if we instead use an open-source LLM on-prem. Furthermore, we design novel physical implementations that leverage SHTs (Section 6.2). In particular, we maintain a sketch for each node in each SHT, and leverage this sketch as part of a tree search to identify the appropriate text span to evaluate a given query, akin to how a person would use a table of contents to find the right chapter. Finally, we maintain provenance (i.e., the specific document text

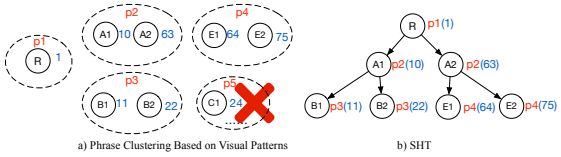


Figure 6: SHT Construction in Civic Agenda Report.

span) for query answers, ensuring that users can verify the source of the information and ensuring trust in the system outputs.

3 SEMANTIC HIERARCHICAL TREE

In this section, we describe our process for recovering structure from documents in the form of Semantic Hierarchical Trees (SHTs), which then acts as an index for subsequent querying. We start by formalizing the notion of SHTs and templates, and then describe how to extract an SHT for a single document, followed by extracting them across collections by leveraging shared templates.

3.1 Preliminaries

We focus on rich text documents, such as PDF and Word documents, that include visual formatting information (e.g., multiple font types and sizes), as shown in Figure 3.

Documents, Words, and Phrases. Consider a set of documents $\mathcal{D} = \{D_1, D_2, ..., D_l\}$. For each document $D \in \mathcal{D}$, which may be a PDF or Word document, we often instead operate on a plain text serialized representation, extracted as a preprocessing step. To generate this representation for a document D, we use an extraction tool such as pdfplumber [13], which generates a sequence of words $W_D = [w_1, ..., w_m]$, each with formatting and location features (e.g., font name/size/bounding boxes). For simplicity, we ignore images, but they can be treated as a special word. For any two consecutive words w_i and w_{i+1}, if they have the same formatting features: font size, name (e.g., Times New Roman), and type (e.g., bold or underline), we group them into a phrase s. We let $S_D = [s_1, ..., s_n]$ be the sequence of phrases corresponding to D—we often operate on S_D instead of the document directly.

Visual Patterns. For each phrase $s \in S_D$, we further define a *visual pattern*, $p(s)$, as a vector of visual formatting features; we currently use: $p(s) = [size, name, type, all\_cap, num\_st, alpha\_st, center]$ but other features may be included. Here, the first three features correspond to the font, as in the word-level features we had previously, and the remaining three features are phrase-level features: *all_cap* is a Boolean value that denotes whether the phrase s is capitalized, *num_st* and *alpha_st* indicate whether the phrase starts with a number (e.g., 1) or a letter (e.g., A), while *center* indicates if a phrase is in the center of a line.

Candidate SHTs. We are now in a position to define SHTs. We define a *candidate SHT* for a document D to be a single-rooted, ordered, fully connected, directed tree $T = (V, E)$, where each $v \in V$ corresponds to a single distinct phrase $s_i \in S_D$, denoted $ind(v) = i$, the *phrase index* for v, satisfying (1) $ind(v) < ind(v')$ for any children v' of v, and (2) $ind(v) < ind(v')$ for any right siblings v' of v. These two properties together imply that a pre-order traversal of T visits nodes in increasing phrase index order. A candidate SHT for Figure 1a is shown in Figure 6b. Node A1 represents the phrase (and section header) "Capital Improvement and Disaster Recovery Projects (Design)", while B2 represents the phrase (and subsection header) "PCH Median Improvement Project". The phrase index for each node is shown in parenthesis, e.g., $ind(\text{A1}) = 10$; i.e., A1 corresponds to s_{10}; ignore the p_i (in red) for now. The SHT obeys the two conditions listed, e.g., A1 (with phrase index 10) has children (11 and 22) and a sibling (63) with larger phrase indexes.

Note, however, that not all phrases in S_D are found in the SHT; this is by design: the SHT simply represents the phrases corresponding to the *headers* of the document, while those that correspond to the *content* are omitted. For example, Figure 6b omits phrases $s_2, .., s_9$. However, in certain cases, it may be convenient to refer to headers and content together. For this, we define *text span* or ts, to be a sequence of phrases $s_i, ..., s_{i+k} \in S_D$, or equivalently $[i, i + k]$. We define $next(v)$ for a given node v to be the phrase index corresponding to its sibling to the immediate right, if available, or, if not, the sibling to the immediate right of the closest ancestor that has one. If none of the ancestors of v have a right sibling, $next(v) = n$, where n is the total number of phrases in S_D. To illustrate, $next(\text{A1}) = next(\text{B2}) = 63$ (i.e., A2), while $next(\text{A2}) = next(\text{R}) = 100$, assuming s_{100} is the final phrase in our document. A given node $v \in V$ has a text span: $ts(v) = [ind(v), next(v) - 1]$, i.e., v "covers" all of the phrases until the next node with phrase index $next(v)$. Thus, $ts(\text{R})$ is $[1, 100]$, while $ts(\text{B2})$ is $[22, 62]$. That is, B2 "covers" both the header, s_{22}, as well as the content $s_{23}, \ldots, s_{62}$, until the next header, A2. In the following, we equivalently refer to a node v, its header phrase $s_{ind(v)}$ (i.e., the header corresponding to v), or text span $ts(v)$ (i.e., the header and content contained within v). We finally introduce the notion of a *granularity* or *height* of a node v, which is simply the depth of v in the SHT; in our example, the depth of R is 1, and A1 is 2.

3.2 SHT Construction on a Single Document

Given a document D with phrases S_D, there are exponentially many candidate SHTs; our goal is to identify the *true SHT* that correctly reflects the semantic structure of the document. To do so, our procedure, `oracle_gen`(D), first identifies which phrases are header phrases (and therefore correspond to SHT nodes). We then assemble these phrases into a tree, ensuring that it is a candidate SHT.

Header Phrase Identification. To identify if a phrase $s \in S_D$ is a header phrase, we make use of visual patterns $p(s)$. We cluster the phrases in S_D based on their visual patterns. For our running example, the clusters that emerge are shown in Figure 6a, each labeled with its visual pattern (in red). Here, the majority of the phrases end up in the cluster with pattern p5—this corresponds to the content phrases in the document (e.g., C1 in Figure 1-a is a paragraph). To remove clusters whose phrases do not correspond to header phrases, we use LLMs as an oracle. We randomly sample $min(|C|, k)$ (k is a predefined threshold) phrases in each cluster $C \in \mathcal{C}$. For each sampled phrase $s \in C$, we construct the LLM prompt "Is the phrase [s] a header in the document?". If over half of the sampled phrases in C are non-headers, then C is pruned (e.g., the cluster containing C1 is dropped since C1 is a paragraph). To verify if GPT-4 is effective at disambiguating headers from non-headers, we carefully examined over 200 documents from 16 datasets, covering six diverse domains. In our testing, when $k = 10$, GPT-4 effectively removes non-header clusters on 97% of the documents with total

cost as $0.37. Still, since this cost is non-zero, we would want to minimize it when working on a large collection of documents; as we illustrate in our next section, we only invoke LLMs for a small subset of documents, each corresponding to a different template.

Tree Construction. Given the header phrases across the remaining clusters in C, we assemble the corresponding nodes into a tree. We proceed top-down, operating on one cluster at a time, adding the entire cluster to the partially constructed SHT. At each step, we pick the cluster C that contains the phrase with the lowest index. For each phrase s_i in this cluster C, we create a corresponding node v_i and add it to the partially constructed SHT, in increasing phrase index order, simultaneously. For each such node v_i, we examine the ts of all existing nodes in the partially constructed SHT, and pick its parent to be the ancestor v_j such that $ind(v_i) \in ts(v_j)$, and there is no other $v_k > v_j$ such that $ind(v_i) \in ts(v_k)$. This condition basically ensures that v_i is added under the most specific node v_j that can accommodate it. Once we've identified the appropriate parents for each node in the cluster, we then add all of these nodes together. The root (usually corresponding to s_1) merits special treatment: if there is no cluster that contains s_1, we create a node corresponding to s_1, else we start with the cluster that contains s_1. Usually this cluster just has s_1; if it contains other phrases, we create an artificial root node corresponding to an empty phrase s_0, and deem it to be the root. We then process the cluster that contains s_1 along with other phrases. Returning to our example, the cluster corresponding to visual pattern p_1 with phrase s_1 is processed first, allowing R to be added to the tree. Then, the cluster corresponding to p_2 is processed next as it has the lowest phrase index number 10, with A1 and A2 added to the tree together, both with R as parent. Then, the cluster corresponding to p_3 is processed, with B1 and B2 being added as children of A1, and so on.

Correctness for Well-Formatted SHTs. Next, we show that if the true SHT for a document has a property that we call *well-formattedness*, then `oracle_gen`(D) correctly outputs the true SHT.

Given an SHT T, the visual prefix $vispre(v)$ for a node v is defined to be the sequence of visual patterns from the root to v. In our example, $vispre(\text{B1}) = p_1p_2$. We extend the definition to a set in the natural way, e.g., $vispre(\{\text{B2, A1}\}) = \{p_1, p_1p_2\}$. Let $pset(p)$ be a function that accepts a visual pattern and returns all the nodes that obey that pattern. For example, $pset(p_2)$ ={A1, A2}.

Then, an SHT $T = (V, E)$ is said to be *well-formatted* if (1) for any two siblings v_i, v_j, $p(v_i) = p(v_j)$; (2) for all visual patterns p, $vispre(pset(p))$ is unique. The first condition mandates that sibling nodes, such as $B1$ and $B2$, must share the same visual pattern. However, it does not require that all nodes at the same depth, like $B2$ and $E1$, must have identical visual patterns. In our agenda watch dataset, subsection headers within a section often have similar formatting, but this need not hold across sections, i.e., different sections may use different formatting. The second condition states that nodes sharing the same visual pattern must have identical visual prefixes. For example, $B1$ and $B2$ have the visual prefix p_1p_2. Thus, the visual pattern signifies a certain "semantic level" within the SHT, following a consistent path to the root.

Theorem 3.1. *If the true SHT for a document D is* well-formatted, *and if an LLM can correctly identify non-headers, then* `oracle_gen`*(D) outputs the true SHT.*

Proof. Let T and GT be the SHT returned by `oracle_gen` and in the ground truth, respectively. We prove $T = GT$ when T is a well-formatted SHT by induction. Let v_i be the i-th node added in the `oracle_gen` approach, and $N_{i-1} = \{v_1, v_2, ...v_{i-1}\}$ be the first $(i-1)$-th nodes added in `oracle_gen`, respectively. Let T_{i-1} and GT_{i-1} be the induced subgraph of T and GT based on the set of nodes N_{i-1}. By induction, we assume that `oracle_gen` returns the correct SHT, i.e., $T_{i-1} = GT_{i-1}$, when adding the first $(i-1)$-th nodes, and we further prove that, by adding v_i, $T_i = GT_i$.

Let v_j and $v_j^{'}$ be the parent node of v_i in T_i and GT_i, respectively. We prove that $v_j = v_j^{'}$ by considering two cases: one where there exists a node $v_k \in T_{i-1}$ (and $v_k \in GT_{i-1}$, since $T_{i-1} = GT_{i-1}$) shares the same visual pattern as v_i, i.e., $p(v_k) = p(v_i)$, and one where it does not. Let $g(v)$ be the granularity (i.e., height of node) of v in T_i. Let $path(v_i)$ be the sequence of nodes from root to v_i in T_i.

Assume $\exists v_k \in T_{i-1}$ and $v_k \in GT_{i-1}$, s.t., $p(v_k) = p(v_i)$. $g(v_j) = g(v_i) + 1$ since v_j is the parent node of v_i in T_i. By definition, $\forall v \in path(v_i), v \neq v_i$, we have $ind(v) < ind(v_i)$ and $ind(v_i) \in ts(v)$. We call each $v \in path(v_i), v \neq v_i$ as a candidate parent node of v_i since adding an edge from v to v_i will make T_i a valid candidate SHT. Thus $v_j^{'} \in path(v_i)$ since GT should be at least a valid SHT and there is no other $v_m > v_j$ such that $ind(v_i) \in ts(v_m)$. If $g(v_j^{'}) \neq g(v_j)$, there at least exists one node $v_l \in path(v_i)$ and v_l is a child node of $v_j^{'}$, s.t., $p(v_l) = p(v_i)$, since GT_i is a well-formatted SHT and the sibling nodes v_l and v_i belonging to the same parent $v_j^{'}$ should have the same visual pattern. By $p(v_k) = p(v_i)$, we have $p(v_l) = p(v_k)$, and thus $g(v_l) = g(v_k)$, since $vispre(v_l) = vispre(v_k)$. $g(v_l) = g(v_k)$ implies $g(v_j^{'}) = g(v_j)$, which contradicts with $g(v_j^{'}) \neq g(v_j)$. By contradiction, we have $g(v_j^{'}) = g(v_j)$ and further $v_j = v_j^{j}$ since both v_j and $v_j^{'}$ are in $path(v_i)$.

Assume $\nexists v_k \in T_{i-1}$ and $v_k \in GT_{i-1}$, s.t., $p(v_k) = p(v_i)$. Similarly we show $v_j = v_j^{'}$ by contradiction in this case. Assuming $v_j \neq v_j^{'}$, there at least exist a node $v_l \in path(v_i), v_l \neq v_i$ and v_l is a child of $v_j^{'}$, s.t., $p(v_l) = p(v_i)$, since $v_j^{'} \in path(v_i)$ and GT_i is a well-formatted SHT. However, this contradicts to the assumption that $p(v_k) = p(v_i)$. By contradiction, we have $v_j = v_j^{'}$, which concludes the proof. □

3.3 SHT Construction across Documents

Given a set of documents $\mathcal{D} = \{D_1, ..., D_l\}$, applying `oracle_gen`($D_i$) to each D_i can be costly when l is large. Here, we leverage the fact that, in addition to being *well-formatted*, the documents share common *templates*. We define the notion of a template below. We process each document D_i in turn, attempting it to match against one of the existing templates $tp \in \mathcal{TP}$ via a function `template_gen`(tp, D_i); if a match is successful, a SHT for D_i is returned–without any LLM calls. Otherwise, we call `oracle_gen`(D_i)—here, the corresponding template tp for the returned SHT is added to $\mathcal{TP}$. If there are multiple successful matches in $\mathcal{TP}$, we return the largest SHT of them all; the rationale here is that we want to capture as much of the header information as possible as part of the SHT.

Template. We now define the notion of a template associated with an SHT. The *template* for an SHT T: $tp = \{g : \{p\}\}$ is a sorted dictionary that captures the mapping between the granularities g

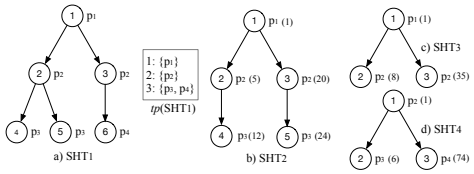


Figure 7: SHT construction by Pattern Matching; the documents represented by b and c are matches to tp(SHT1)) but not d.

of nodes and the set $\{p\}$ of visual patterns found at that granularity. This dictionary is additionally sorted by granularity in increasing order. tp(SHT1) is the template of SHT1 shown in Figure 7-a. We let $tp.g$ and $tp.p$ be the granularities and visual patterns in tp. For SHT1 in Figure 7-a, $tp.g = \{1, 2, 3\}$ and $tp.p = \{p_1, p_2, p_3, p_4\}$. Let $tp.g(p)$ be the granularity of a visual pattern p in tp, e.g., $tp.g(p_1) = 1$ for SHT1. (This value is unique by construction from Section 3.2.) If $p \notin tp.p$, $tp.g(p) = -1$.

Template Matching and Generation. We say a document D *matches* a template tp if the visual patterns contained amongst the phrases S_D cover each granularity $1 \ldots i$, for some i which is a prefix of tp. For instance, document D_1 with true SHT, SHT1, has a corresponding template tp(SHT1) in Figure 7-a, and document D_2, has a true SHT, SHT2, Figure 7-b. Since D_2 includes patterns $\{p_1, p_2, p_3\}$, it covers every granularity of the template of D_1, and therefore *matches* the template. Additionally, document D_3 with true SHT, SHT3, in Figure 7-c, which includes patterns $\{p_1, p_2\}$, also matches the template, since it covers a prefix of the granularities in the template (namely 1 and 2), even though it lacks patterns $\{p3, p4\}$. On the other hand, document D_4 with true SHT, SHT4, in Figure 7-d, does not contain a match for p_1, thereby not meeting the prefix constraint, and not being a match for the template. Our rationale for admitting prefix matches is the observation that as the granularity of a header becomes more fine-grained, its visual pattern tends to be more varied. For example, for two scientific papers obeying the same template, the visual patterns of sections remain consistent, but within each section the visual patterns used may vary depending on individual preferences. Note here that in our implementation, we allow for any non-zero prefix for a match; for more constrained document collections, a user may set a prefix threshold, e.g., at least three levels of the template must be covered.

Armed with templates and matches to a template, we can now describe our `template_gen`(tp, D) procedure, listed in Algorithm 1. We proceed in two phases, where we first identify all of the phrases $s \in S_D$ that match those in $tp.p$, we add these phrases as nodes to V for our yet-to-be-constructed SHT (Line 3-5). Given these phrases, we check if there is a match for the template tp, where a match is defined as above to be a prefix of the template. If no match is found, an empty result is returned (Line 6-7), else we assemble the nodes in V into an SHT; we use a similar tree construction procedure as in the previous section, operating on the phrases found in the first step, clustered based on visual pattern (Line 8-10).

4 DATA MODEL AND QUERY LANGUAGE

In the previous section, we described how we can extract SHTs for each document in a collection as part of document ingestion. Here,

Algorithm 1: `template_gen`(tp, D)

```
1  SHT_D = (V, E), V = ∅, E = ∅
2  G = {}
3  for s_i ∈ S_D do
4      if p(s_i) ∈ tp.p then
5          V = V ∪ s_i, G = G ∪ tp.g(p(s_i))
6  if G = ∅ or ∃i ∈ G, i > 1, (i − 1) ∉ G then
7      Return {}
8  for v_i ∈ V, v_j ∈ V do
9      if ind(v_j) ∈ ts(v_i) and
         ∄v_k ∈ V, ind(v_k) > ind(v_i), s.t., ind(v_j) ∈ ts(v_k) then
10         E = E ∪ (v_i, v_j)
11 Return SHT_D
```

we define the data model used by ZenDB to represent the SHTs as well as other system-specific information, along with user-defined tables that we call *DTables*, short for *Document Tables*.

4.1 Data Model Definition

In addition to traditional relational tables that we call *base tables*, ZenDB supports three new types of tables that respectively (i) represent the SHTs per document collection, (ii) let users specify one or more structured relations over the documents, called *DTables*, to be used within queries; (iii) maintain system metadata associated with the user-defined tables. We describe each one in turn.

4.1.1 SHT Table. The *SHT table*, shown in Figure 8-c, is a system-defined and maintained table that represents the SHTs in a document collection. Each row captures information about an SHT Node, and is populated as described subsequently in Section 5. Its main attributes are:

- `doc_id`, `node_id` identify the node in a given document.
- `name` represents the header phrase s corresponding to the node.
- `granularity` represents the depth of the node in the tree.
- `context`, `summary`, `size` correspond to the entire sequence of phrases in the text span, a short summary of the text span, and the number of tokens in the text span.
- `st_page` and `ed_page`, listing the start and end pages for the text span.
- `child_ids` and `ancestor_ids`, the IDs for the children and entire sequence of ancestors.

We note that `summary`, `size`, `st/ed_page`, and `ancestor_ids` can be derived from the other attributes, but we store them explicitly for convenience. These attributes are all used during query processing.

4.1.2 User-defined DTables. Users can use SQL to define DTables, with those tables being used in subsequent queries (Figure 5). We use a special keyword `DESCRIPTION` to both designate the fact that this is not an ordinary table, and also allowing natural language to be provided that may be used in LLM prompts. To define such a table, the user can say:

```
CREATE TABLE [name] (...) WITH DESCRIPTION [description]
```

Here, the user provides a natural language description for the table. Attributes may be provided during table creation in parentheses (or omitted), and/or could be added afterwards, via the standard approach to alter schemas:

User-Defined Tables (* is system-defined attribute)

doc_id*	text_span*	node*	name	type	begin_time
1	TS1	B1	NULL	NULL	NULL
1	TS2	B2	NULL	NULL	NULL

a) Projects (Partial)

doc_id*	text_span*	node*	meeting_date	subject
1	TS1	R	NULL	NULL

b) Agenda Meeting (Partial)

System-Defined Tables

doc_id	node_id	name	granularity	st_page	ed_page	context	summary	size	child_ids	ancestor_ids
1	R	Public Works Commission Agenda Report	1	1	12	context_1	S(context_1)	TC(context_1)	[A0, A1, …]	[]
1	A1	Capital Improvement Projects (Design)	2	1	1	context_2	S(context_2)	TC(context_2)	[B1, B2, …]	[R]
1	B1	Marie Canyon Green Street	3	1	1	context_3	S(context_3)	TC(context_3)	[]	[A1, R]
1	B2	PCH Median Improvements Project	3	1	2	context_4	S(context_4)	TC(context_4)	[]	[A1,R]

c) SHT Table (Partial)

table_name	table_description	doc_id	table_text_span	table_node	t_range	multi_tuple
Projects	Projects table contains a set of projects in public agenda report…	1	TS3	R	[3,3]	False
Agenda Meeting	Agenda Meeting table describes the agenda meeting…	1	TS4	R	[1,1]	False
Projects	Projects table contains a set of projects in public agenda report…	2	?	?	?	?
Agenda Meeting	Agenda Meeting table describes the agenda meeting…	2	?	?	?	?

d) Table Catalog (Partial)

table_name	attr_name	attr_description	type
Projects	name	name of project	TEXT
Projects	type	type of project	TEXT
Agenda Meeting	subject	subject of meeting	TEXT

e) Attribute Catalog (Partial)

Figure 8: Data Model: User-Defined Tables and System-Defined Tables.

```
ALTER TABLE [name]
ADD [name] [type] WITH DESCRIPTION [description], ... ;
```

Again, a natural language description for the attributes are provided when they are added. As we will discuss in Section 5, when the user creates a DTable, ZenDB populates them offline with rows that correspond to tuples. Each tuple represents one entity that can be found in a document. User defined attributes for these tuples are populated with `NULL`, and are filled in on-demand during query time, as shown in Figure 8a. Here, the `Project` DTable contains user-defined attributes `name`, `type`, and `begin-time`. ZenDB also maintains three hidden system-defined attributes per DTable—the document id, text span used to extract the tuple, and SHT nodes used in the derivation. These attributes track how each tuple was derived, to provide context when extracting tuple attributes later on, and for debugging and provenance purposes. For instance, $B1$ corresponds to the "Marie Canyon Green Street" project tuple, and the tuple's text span may be the same as $B1$ or a subset (Figure 8c).

The user-defined attributes represent the result of a *read* operation over each attribute. In addition, every expression implicitly defines additional attributes in this table. For instance, if a query evaluates `Projects.name = "Capital Improvement"` directly using an LLM call, then the attribute `[Projects.name|eq|Capital Improvement]` is instantiated and populated with the LLM response.

Note that we chose to represent these user-specified DTables as regular tables as opposed to views or materialized views; but they could also be represented as such.

4.1.3 System-Defined Tables. In addition to the SHT table, ZenDB maintains two system-defined tables: `Table Catalog` and `Attribute Catalog` store metadata related to tables and attributes respectively (Figure 8d,e). In addition to names and descriptions, `Table Catalog` tracks the text span and SHT node(s) used to identify the contents of the table (since a table may be a small portion of the document), used to localize search when extracting tuples—thereby reducing cost during query processing. The attribute `t_range` refers to the min/max granularities of the nodes used to extract tuples in the table. For example, all `Project` tuples extracted so far have granularity 3, thus `t_range = [3,3]`; this is the setting where tuples correspond to nodes (of some granularity) within the SHT. Finally, to handle the special case where the table is extracted from a leaf node in the SHT, i.e., there are multiple tuples corresponding to a single node that has no finer granularity node below it, we mark this by setting `multi_tuple` to `True`. For instance, consider the scenario when users want to create a table called "References" and each tuple corresponds to a reference in a published paper.

4.2 Query Language

ZenDB currently supports a subset of SQL, corresponding to simple non-nested queries on one or more DTables with optional aggregation, as represented by the following template:

```
SELECT [attr] | agg(attr) FROM [ST]+
 WHERE [predicate]    GROUP BY [attr]
```

where `[..]` denotes a list of elements, `attr` refers to an expression over an attribute, `ST` refers to one or more DTables, and `agg()` includes `SUM, COUNT, AVG, MAX, MIN`[1]. A predicate has the form: `attr op operand`, where the operators include `>|≥|<|≤|=|LIKE|IN`, and `operand` is one or more constants. `LIKE` is used for fuzzy matching where either string similarity or semantic similarity could be used[2]. We add a restriction that if multiple DTables are listed in the `FROM` clause, then the `WHERE` clause includes a predicate specifying that the tuples are equi-joined on `doc_id`. We add this restriction for now to only allow for within-document joins, but we plan to relax this in future work.

Figure 9 shows a query where, for each document whose meeting time is before "2023 October", we count the "Capital Improvement" projects starting after "2022-06-01"; here, we make use of the within-document join across two tables.

The query semantics are defined as fully populating the user-defined DTables with the LLM results of all attribute reads and expressions, and then executing the SQL query as normal. We follow these semantics because it allows for minor consistencies during query evaluation. Specifically, under an oracle LLM that always returns complete and correct responses, the contents of the attribute reads and expressions will always be consistent (e.g., `type` is `['A', 'B']`, and `type = 'A'` is true). However, modern LLMs are imperfect and sensitive to the input prompt and context formulation, so the extracted attribute values and expressions

[1]Text attributes only support `COUNT`, date attributes only support `COUNT, MAX, MIN`.
[2]In ZenDB we use Jaccard similarity with a 0.9 threshold by default.

over the attributes may be inconsistent (e.g., extracted `type` is `'B'`, but `type='A'` is true). Better understanding and reconciling these potential inconsistencies is outside the scope of this paper, and is important future work.

5 TABLE POPULATION

We next describe how we can populate the system-defined tables and attributes described above. Populating the SHT table is straightforward and therefore omitted; we will describe how the `summary` field is populated in Section 6.

Populating Tables Overview. When a user defines a new DTable `T`, updating `Attribute Catalog` (Figure 8e) and `table_name`, `table_descr` in `Table Catalog` (Figure 8d) is easy. However, ZenDB must process the document collection $\mathcal{D}$ to fill in the system-defined attributes (SDAs) in `Table Catalog` and `T`, and populate `T` with tuples. While ZenDB proactively identifies tuples for `T`, it doesn't populate any user-defined attributes until query time.

Consider a partitioning of $\mathcal{D} = \bigcup_{\mathcal{D}_i \subseteq \mathcal{D}} \mathcal{D}_i$, where $\mathcal{D}_i$ is a set of documents sharing the same template, as identified during SHT construction. For each $\mathcal{D}_i$, ZenDB picks a document $D \in \mathcal{D}_i$ and uses an LLM to populate `T` with tuples, and fill in the SDAs. ZenDB then uses a rule-based approach to extract tuples from the remaining documents $D' \in \mathcal{D}_i - \{D\}$ without invoking LLMs. We describe the single document and multi-document extraction next.

Single Document Extraction. To populate SDAs for D for a given DTable `T`, we first identify the node in the SHT for D that captures all of the entities for the `T`; we call this the *table node*. We then identify nodes that correspond to tuples that lie underneath this node. We use two prompts, `table_oracle` and `tuple_oracle` to identify if a given node corresponds to a table or tuple respectively.

```
table_oracle: If the following text describes [table_name], [table_descr],
return true. Otherwise, return false. [node_context].

tuple_oracle: If the following text describes one [tuple_descr] in [
table_name], [table_descr], return true. Otherwise, return false. [
node_context].
```

In these prompts, `[]` is a placeholder. `[table_name]`, `[table_descr]`, and `[tuple_descr]` correspond to the table name and description, and the tuple description in `Table Catalog` (e.g., Figure 8d). `[node_context]` provides the entire text span corresponding to the node from `SHT table` (e.g., in Figure 8c).

To identify the table node, ZenDB walks the SHT top-down and submits `table_oracle` to LLMs for each node. If the response for all of a node v's children are true, then we add v as a candidate table node and stop descending into v's children. Finally, ZenDB fills in the Least Common Ancestor (LCA) of the candidate table nodes as `table_node` in `Table Catalog`.

Once the `table_node` is found, ZenDB attempts to populate `T` with tuples. Once again, ZenDB performs a top-down traversal starting from `table_node` and evaluates `tuple_oracle` on each node. If a node v evaluates to true, it means the node corresponds to an entity. We insert a new tuple into `T`, assign its node and text span to that of v's, and stop traversing v's descendants. If no nodes evaluate to true, it implies a leaf node contains multiple tuples and so we flag `multi_tuple` as true in `Table Catalog` without populating `T`. We handle this case separately in Section 6.

Multi-document Extraction. Repeated LLM calls for extracting tuple boundaries for every document is too expensive, so we use a rule-based approach to populate tuples (and other SDAs) from the rest of the documents that share the same template.

Consider populating `table_node` for document $D' \in \mathcal{D}_i, D' \neq D$, where tuples from D were populated as described previously. Let the `table_node` (i.e., the finest granularity node below which all the tuples are found) and `t_range` (i.e., tuple granularity range) of the table `T` in document D (that has already been populated) be v_{tn} and $[l, r]$, respectively. For D', if there exists a node v in its SHT such that v's granularity matches that of v_{tn} and the textual similarity between v's phrase and that of v_{tn} is greater than a threshold, then we set v to be the table node for D'; else if no such v exists, the root is set to be the table node.

Now, to populate tuples, suppose for the tuple range $[l, r]$ in $D, l = r = x$. In this easy case, there is a well-defined granularity in the SHT where tuples are found. Then, we add all nodes at granularity x from D' as candidate tuples to `T` (assuming there is a non-zero number of them). If $l \neq r$ or if the SHT for D' has a maximum height $< x$, then we simply set `multi_tuple` to true; in this case, the granularity for tuples is ambiguous, and so we treat it similar to the case where there may be multiple tuples at a given node.

Multi-document Extraction Rules. In more detail, we define the following two rules. For each node v in an SHT, we use $v.attr$ to denote any attribute $attr$ belonging to v in the SHT table (e.g., $v.granularity$). For the document $D' \in \mathcal{D}_i$, let $V_{D'}$ be the set of nodes corresponding to D' in the SHT table, and D'.`table_node` be the `table_node` of `T` in document D' in Table Catalog.

Rule 1: $\forall v_i \in V_{D'}$, if $v_i.granularity = v_{tn}.granularity$ as well as $Sim(v_i.name, v_{tn}.name > \theta$, then D'.`table_node`$= v_i$.
Else, D'.`table_node`$= root$.

If the rule is unsatisfied, we set the `table_node` to be the root node of SHT corresponding to D'. To populate the nodes corresponding to tuples, we first populate the granularity range of tuples `t_range`.

Rule 2: If $\exists v_j \in D'$.`table_node`.`child_ids`, $l \leq v_j.granularity \leq r$, then D'.`t_range`$= [l, r]$. Else, `multi_tuple` $= true$.

If the granularities of tuples of `T` in document D' are consistent, i.e., $l = r$ in D'.`t_range`, then we create a set of nodes V, where for each $v \in V$, $v.granularity = l$ and D'.`table_node`$\in v.ancestor\_ids$. V is further converted to a set of tuples whose $text\_span = v.context$ and $nodes = \{v\}$. These tuples are inserted into the table `T`. If Rule 2 is violated, we set `multi_tuple` as true to denote that we do not have a one-to-one mapping between the set of nodes and tuples when populating the table for D'. Note that doing so might introduce false positives instead of false negatives. False positives are permissive since they will not lose the context of where the answers may be present, and in Section 6 we will discuss how to reduce false positives during query execution. When `multi_tuple` in D is true, we don't populate `t_range` but set `multi_tuple` as true for D'. Overall, when the number of distinct templates (i.e., $|\mathcal{D}_i|$) in documents $\mathcal{D}$ is small, the cost incurred by LLMs to populate the SDAs is minimal, since we only invoke LLMs on a single document for each cluster.

6 QUERY ENGINE

We discuss how ZenDB generates a query plan for a given query Q in Section 6.1, and then describe our physical operator implementations that leverage SHTs in Section 6.2.

```
SELECT Agenda_Meeting.doc_id, COUNT(Projects.name)
FROM Projects, Agenda_Meeting
WHERE Projects.type = 'Capital Improvement'
AND Projects.begin_time > '2022-06-01'
AND Agenda_Meeting.meeting_time < '2023 October'
AND Projects.doc_id = Agenda_Meeting.doc_id
GROUP BY Agenda_Meeting.doc_id
```

Figure 9: A Query on Civic Agenda Documents.

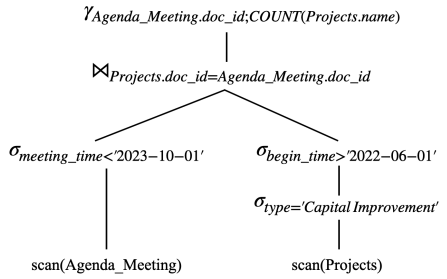


Figure 10: A Query Plan for the Query in Figure 9.

6.1 Logical Query Plan

Unlike traditional settings where I/O and computation costs dominate, here, LLM invocations add to monetary cost[3] and/or latency, and thus must be minimized if possible. Keeping this guideline in mind, when generating a logical query plan for a given query Q, ZenDB first parses the SQL query into a parse tree of relational operators. Subsequently, predicates are pushed down to reduce intermediate sizes and thereby downstream LLM invocations—but also taking into account the fact that predicate evaluations that rely on LLMs can be expensive. ZenDB relies on the standard approach from prior work [32] for expensive predicate reordering that takes into account both the selectivity and cost. Specifically, we define a metric $f(o)$ for each selection operator o. Let s_o be the selectivity of o, computed as $s_o = \frac{|T_s|}{|T_c|}$, where T_c (T_s) are tuples that are processed (satisfy) the predicate associated with o. Let e_o be the average cost for evaluating a tuple using operator o, which is estimated adaptively during query execution as more tuples are processed by o. The goodness of a selection operator o is then defined as $f_o = e_o \times s_o$. Intuitively, if an operator o has lower cost e_o and selectivity c_o, o is preferred to be executed early. ZenDB will sort the set of selection operators on the same table in the increasing order of $f(o)$. Projections on the other hand, are pulled up, to avoid having to populate attributes through LLM calls for tuples that may get discarded. Until a selection or projection is encountered that requires a specific attribute for a tuple, that attribute stays uninterpreted, and therefore NULL.

From a join order standpoint, ZenDB adopts a greedy algorithm to generate a left-deep tree, in an approach akin to standard relational query optimization techniques. Here, instead of optimizing for reducing the sizes of intermediate results, we focus on reducing the LLM invocation cost. Let $E(T)$ be the cost (in terms of dollars or latency) for evaluating all of the predicates in Q corresponding only to table T on all of the tuples of T. ZenDB ranks the tables in Q as $T_1, T_2, \ldots$ based on their $E(T_i)$ in increasing order, forming a left deep tree with T_1 as the driving table, followed by T_2 to form $T_1 \bowtie T2$, with the remaining tables being selected based on $E(.)$. When `multi_tuple` is false, implying that in table T, we have pre-populated potential tuples, and therefore have a more precise estimate, $E(T) = |T| \times e$ is estimated at query time, where $|T|$ is the number of tuples in T, e denotes the average cost of evaluating a single tuple. Initially, $E(T)$ is set to be $|T|$ to prioritize evaluating the table with the smaller number of tuples, and e will be estimated adaptively as more tuples are processed during query execution. One logical plan for the query in Figure 9 is shown in Figure 10, where `agenda_meeting` only has one tuple compared to the `Projects` table with more than 40 tuples, and thus is evaluated first. The estimation of $E(T)$ when `multi_tuple` is true will be described in Section 6.2.

[3]This is common for several commercial LLMs like OpenAI, Claude-3 [7], Google Gemini [3].

Algorithm 2: $tree\_evaluate(SHT, tuple, e)$

```
1  CurrentNodes = {tuple.node}
2  Ans = ∅
3  T = getTree(SHT, node)
4  /*Refine candidate nodes*/
5  while stop_condition(T) = False do
6      CNs = ∅
7      for n ∈ CurrentNodes do
8          if search_oracle(n, e) = True then
9              CNs = CNs ∪ n
10     CurrentNodes = CNs.childs_id
11 if e.type = predicate then
12     /*Evaluating A Predicate*/
13     for node ∈ CNs do
14         if evaluate_oracle(node.summary, e) = True then
15             Ans = Ans ∪ node
16             Return Ans
17 if e.type = attribute then
18     /*Extracting Attribute Values*/
19     for node ∈ CNs do
20         Ans = Ans ∪ extract_oracle(node.summary, e)
21 Return Ans
```

6.2 Physical Query Plan

During query execution, each tuple in the user-defined DTables has attribute values that begin as `NULL` as in Figure 8a, but some attributes will get populated through selections or projections. When `multi_tuple` is true, ZenDB leverages LLMs to create a set of tuples satisfying the corresponding predicates with their attributes listed in the projections to be computed, as will discussed shortly. We now discuss our implementations of various operators.

Scan. As part of our scan operator, ZenDB executes the query document by document (which explains the restriction of join on `doc_id` in Section 4.2). This operator first retrieves the tuples in the first document as a batch, followed by tuples in the second document; thus only one SHT is processed at a time.

Selections and Projections. Consider a predicate *pred* or a projection *proj* on table T; a similar procedure is followed in either case. Say `multi_tuple` is false, so each row in T corresponds to a single potential tuple. ZenDB then calls a function $evaluate(SHT, tuple, e)$,

listed in Algorithm 2, with e set to $pred$ (respectively, $proj$) to evaluate whether $tuple$ satisfies $pred$, returning it if so (respectively, the value of the attribute in $proj$). This function implements a tree search on the SHTs, leveraging summaries for each node, as defined in Section 4.1. We next describe how we populate this *summary* per node in the SHT table (Figure 8c).

Summary Creation. Given the SHT for a document D and the expression e, $S(v)$, the summary for a node v, comprises the following: (1) The phrase(s) corresponding to both v and its ancestors. (2) An extractive summary of the text span of v, which is a set of important sentences determined using standard (non-LLM) NLP tools like NLTK [10]. (3) The top-1 sentence the text span of v with the highest semantic similarity (e.g., cosine similarity) with e.

Parts (1) and (2) are prepared offline when the SHT is built. Part (3) is added during query processing. Including phrases (i.e., headers) of ancestors in (1) often helps enhance accuracy by including additional background for interpreting v's text span. For example, in Figure 1, the summary of node $B2$ contains the header phrase of its parent, "Capital Improvement Projects (Design)", helping us identify v as a candidate node when evaluating a predicate such as `type = Capital Improvement`.

Tree Search Algorithm. Given a document D with its SHT, a tuple node $node$, an expression e (either a predicate or a projection), our Algorithm 2, first identifies a sub-tree T in SHT with $node$ as the root (Line 4), searches T top-down. For each node n in one layer, it calls $search\_oracle(n, e)$ to check whether n's summary contains the right information to evaluate expression e. It then adds all the nodes that pass $search\_oracle$ into a candidate set CNs (Line 6-12), and recursively searches their children until a stopping condition is met (Line 6). This condition is (1) the leaf node is reached, (2) the number of tokens in the summary of the node is larger than that of its context (i.e., text span).

```
search_oracle(node, e): If the following text contains the information
that describes [e.descr], return True; otherwise, return False. The
context is [node.summary].
Example: [e.descr] = 'the type of project is Capital Improvement'
```

For each candidate node $n \in CNs$, if the expression e is a predicate, then a call to an LLM with prompt $evaluate\_oracle(node.summary, e)$ is issued to evaluate if the summary of node satisfies the predicate. This step stops early when there exists one node that passes $evaluate\_oracle(rc, e)$ (Line 11-17). When e is a projected attribute, $extract\_oracle(node.summary, e)$ is instead used to extract the value of the projected attribute (Line 18-22).

```
evaluate_oracle(context, e):  Return True if  [e.descr] based on the
following context [context]. Otherwise, return False.
Example: [e.descr] = 'type of project is Capital Improvement'
```

```
extract_oracle(context, e): Return [e.descr]  based on the following
context [context].
Example: [e.descr] = 'name of project'
```

Each selection operator o returns the set of tuples in table T satisfying the predicate associated with o to downstream operators. We handle the case where `multi_tuple` is true for table T in Section 6.3.

Even though executing a tree search procedure by exposing node summaries to LLMs incurs additional cost, it is minimal in practice since the height of the tree is often small (thus, the number of iterations is small), and the size of the summary is small and controllable. In Section 7 we show that the benefit introduced by summaries, which achieves better accuracy and lower cost, dominates the additional cost.

Other Operators. We use nested loop as our join algorithm. As mentioned earlier, even if we consider latency to be the primary optimization criterion, the evaluation of predicates and projections through LLM invocations would dominate overall latency, and the number of intermediate tuples to be processed during query execution is often not a large number. If we further treat monetary cost as the primary criterion, then joins are effectively free. Thus, a simple nested loop join suffices. Similarly, other operators like aggregation and group-by use simple relational variants.

Provenance of Query Answers. ZenDB maintains the provenance in the form of the corresponding text span(s) for the returned query answers in a manner analogous to classical relational provenance [30]. During query processing, we keep track of the sequence of text spans consulted to populate attributes or verify predicates, as an additional metadata attribute, per tuple. These text spans are combined into an array during joins. While we could apply the same idea to aggregations and capture the provenance of contributing tuples into an array, this representation is unwieldy. Determining how best to show all of this provenance to end-users to ensure trust in query answers is an important topic for future work.

6.3 Operators for the Multiple Tuple Case

When `multi_tuple` is true for table `T`, there are no tuples in `T` after population in Section 5, and the context of `table_node` may contain multiple tuples. Let $pred(T)$ and $proj(T)$ be a set of predicates and projected attributes associated with table `T` in a given query Q. In this case, ZenDB searches the text span corresponding to the `table_node` of `T`, and creates a set of tuples satisfying $pred(T)$ with $proj(T)$ being populated by LLMs.

When `table_node` is a leaf node in its SHT, ZenDB submits the prompt `multi_tuple_oracle (table_node,` $pred(T), proj(T)$) to LLMs to extract the projected values for the tuples that satisfies the given predicate $pred(T)$.

```
multi_tuple_oracle(node,pred(T),proj(T)): The following text describes
one or more [tuple_descr]. For each [tuple_descr], if pred(T), then
return [proj(T)] based on the following context [node.context].
Example:
[tuple_descr] = 'paper'
[predT] = 'publication year is greater than 2009 and conference is VLDB'
[proj(T)] = 'name of paper, authors of paper'
```

As an example, consider a publication document D, where users want to create a table called *Reference* with the schema as {name, year}, whose text span corresponds to the references section in a paper. Assume that in the SHT of D, the references section is a leaf node. In this case, ZenDB will not further parse the reference section into individual references, but will call `multi_tuple_oracle()` to extract the paper name and authors per reference from VLDB whose publication year is later than 2009, directly over the references section.

When `table_node` is not a leaf node in its SHT of document D, let D' be a document sharing the same template with D and populating its system-defined attributes via D in Section 5. Let $stop\_granularity$ be the granularity for stopping searching in Algorithm 3, and $stop\_granularity = D.tuple\_range.l$, i.e., the smallest granularity of tuples in D. Note that this may introduce false

Algorithm 3: *tree_evaluate_multi_tuple*

Input: $SHT, table\_node, pred(T), proj(T), stop\_granularity$

```
1  CurrentNodes = {table_node}
2  Tuples = ∅
3  granularity = table_node.granularity
4  T = getTree(SHT, table_node)
5  /*Refine candidate nodes*/
6  while granularity ≤ stop_granularity do
7      CNs = ∅
8      for n ∈ CurrentNodes do
9          if search_oracle(n, e) = True then
10             CNs = CNs ∪ n
11     granularity = granularity + 1
12     CurrentNodes = CNs.childs_id
13 Ans = ∅
14 for n ∈ CNs do
15     Ans = Ans ∪multi_tuple_oracle(t, pred(T), proj(T))
16 Return Ans
```

Datasets	# of Documents	Avg # Pages	Avg # Tokens
Publication	100	11.5	13230
Civic Agenda	41	8.7	3185
Notice	80	7.1	3719

Table 1: Characteristics of Datasets.

positives (one node might correspond to multiple tuples) but would avoid false negatives (there will not exist nodes that correspond to portions of a tuple). ZenDB executes `tree_evaluate_multi_tuple` in Algorithm 3. ZenDB starts searching the subtree of SHT with `table_node` as the root (Line 4). We use the same summary-based search as in `tree_evaluate` in Algorithm 2 to refine the nodes that are related to the given query top-down layer by layer, and stop the search when the granularity of current layer exceeds *stop_granularity* (Line 6-12). For each node $n \in CNs$ that are related to the query and might contain multiple tuples, we call `multi_tuple_oracle` to extract the corresponding tuples (Line 13-15).

7 EVALUATION

In this section, we evaluate ZenDB over three real document collections on accuracy, latency, and cost.

7.1 Methodology

7.1.1 Data & Query Sets. We collected three real-world datasets (i.e., document collections): scientific publications, civic agenda reports, and notice of violations; details are displayed in Table 1.

Scientific Publications. This dataset was collected from a systematic review study that examined research questions in the field of personal data management at UC Irvine [11]. The study analyzed over 500 publications; we randomly selected 100 papers for our dataset. The study explored 20 research questions with human-labeled answers for all of the publications.

Civic Agenda Reports. This dataset, from our collaborators at Big Local News, comprises 41 civic agenda reports from 2022 to 2024 in the City of Malibu [14]. Each report details a series of government projects, including their status, updates, decisions, and timelines for beginning, ending, and expected construction.

Notice of Violations. This dataset, also from Big Local News, of 80 documents describe notices of violations issued by the US Dept. of Transportation from 2023 to 2024 [12]. Each document concerns potential violations detailed by the Hazardous Materials Safety Administration, including detailed violation orders and descriptions, penalty decisions, and proposed compliance orders.

Query Workload. For each dataset, we devise a query workload comprising 9 SQL queries, informed by the needs of our collaborators. These 9 queries are divided into groups of three, QG1, QG2, and QG3, varying in the number of predicates, from one to three respectively. To generate these queries, we first define tables along with a set of attributes per dataset. Then we randomly select i attributes to create i predicates for the queries in group QGi, and in `SELECT`, we additionally include one attribute that is not used in the predicates, as well as `doc_id`. When we end up sampling attributes across multiple relations, we list both in the `FROM` clause, and additionally add an equijoin condition on `doc_id`. So, overall, our queries include selections, projections, and joins. We omit aggregations in our workload since we use relational versions for those operators evaluated after the corresponding attribute values are extracted; and thus the performance on such queries would be similar to that on the queries without them.

7.1.2 Strategies Compared and Evaluation Metrics. We compare ZenDB with four baselines, GPT_single, GPT_merge, RAG_seq, and RAG_tree. The first two operate on an entire document at a time. GPT_single uses a separate LLM call per predicate and projection by constructing a corresponding prompt, appending the entire document as context. GPT_merge combines all of the predicates and projections into a single LLM call alongside the entire document. RAG_seq and RAG_tree refer to RAG-based techniques in two variants implemented by LlamaIndex [9], a state-of-the-art open-source RAG framework: sequential chunking and tree-style chunking, respectively. In RAG_seq, we set the chunk size to 128 tokens and selected top-k chunks, where $k = \max(1, 5\% \times \texttt{doc\_size}/128)$. That is, we retrieve at least one chunk, but no more than 5% of the of the document. RAG_tree constructs a hierarchical tree from the document without leveraging semantic structure. This tree is constructed by first chunking the leaves at a fixed granularity. Nodes higher up in the hierarchy are formed by recursively summarizing the nodes below. Subsequently, a path from the root to leaf is retrieved, instead of just one leaf. GPT-4-32k is used to evaluate the queries for all strategies.

We use precision and recall to measure the quality of query answers. Given a query Q, let $T\_truth(Q)$ and $T\_pre(Q)$ be the set of tuples in the ground truth vs. predicted by an approach, respectively. Precision is measured as $\frac{|T\_truth(Q) \cap T\_pre(Q)|}{|T\_pre(Q)|}$, and recall is $\frac{|T\_truth(Q) \cap T\_pre(Q)|}{|T\_truth(Q)|}$. We count the number of input and output tokens to measure the cost of LLM invocations [6]. Finally, we measure the latency of query execution by taking three runs and reporting the average.

7.2 Experimental Results

Experiment 1: ZenDB vs. GPT-only Strategies. We first compare ZenDB with GPT_single and GPT_merge, both operating on

Strategies	Precision			Recall			Cost ($) / Tokens (× 1000)			Latency (Seconds)		
	PUB	CIVIC	NOTICE	PUB	CIVIC	NOTICE	PUB	CIVIC	NOTICE	PUB	CIVIC	NOTICE
GPT_single	**0.74**	0.45	**0.71**	0.38	0.45	**0.77**	0.98 / 16.2	0.33 / 5.4	0.3 / 5.3	14.6	15.3	6.1
GPT_merge	0.63	0.34	0.66	0.4	0.45	**0.72**	0.8 / 13.2	0.2 / 3.2	0.2 / 3.7	12.9	7.4	5
RAG_seq	0.51	0.12	0.36	0.38	0.13	0.38	**0.02 / 0.4**	**0.02 / 0.29**	**0.01 / 0.18**	**3.76**	**5.1**	**1.3**
RAG_tree	0.51	0.2	0.2	0.38	0.04	0.17	0.07 / 1.2	0.04 / 0.66	0.02 / 0.35	10	8.9	**1.3**
ZenDB	**0.72**	**0.73**	**0.73**	**0.53**	**0.84**	**0.74**	**0.03 / 0.56**	0.03 / 0.53	**0.02 / 0.25**	**4.8**	7	**1.7**

Table 2: Average Precision, Recall, Cost / # of Tokens and Latency of Strategies Per Query, Per Document, in Publication (PUB), Civic Agenda (CIVIC), Notice of Violation (NOTICE) Datasets. (GPT-4-32k is Used.)

Strategies	Precision				Recall				Cost ($) / Tokens (× 1000)				Latency (Seconds)			
	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg
GPT_single	**0.94**	**0.66**	**0.62**	**0.74**	0.65	0.16	0.32	0.38	0.8 / 13.2	1 / 16.6	1.1 / 18.9	0.98 / 16.2	12.8	14.1	16.9	14.6
GPT_merge	**0.94**	0.41	0.63	0.63	0.65	0.13	0.41	0.4	0.8 / 13.2	0.8 / 13.2	0.8 / 13.2	0.8 / 13.2	12.8	12.9	13.1	12.9
RAG_seq	0.73	0.4	0.39	0.51	0.6	0.23	0.31	0.38	**0.01 / 0.23**	**0.02 / 0.38**	**0.03 / 0.59**	**0.02 / 0.4**	**2.7**	**3.9**	**4.5**	**3.76**
RAG_tree	0.79	0.33	0.42	0.51	**0.68**	0.19	0.27	0.38	0.05 / 0.82	0.08 / 1.3	0.1 / 1.6	0.07 / 1.2	8.4	10.4	11.2	10
ZenDB	**0.93**	**0.64**	**0.6**	**0.72**	**0.7**	**0.54**	**0.34**	**0.53**	**0.02 / 0.41**	**0.03 / 0.56**	**0.04 / 0.71**	**0.03 / 0.56**	**3.9**	**5.2**	**5.4**	**4.8**

Table 3: Average Precision, Recall, Cost / # of Tokens and Latency of Strategies Per Query, Per Document, in Publication Dataset. (GPT-4-32k is Used.)

Strategies	Precision				Recall				Cost ($) / Tokens (× 1000)				Latency (Seconds)			
	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg
GPT_single	0.64	0.36	0.36	0.45	0.73	0.37	0.24	0.45	0.2 / 3.2	0.33 / 5.4	0.47 / 7.6	0.33 / 5.4	7.5	15.3	23.1	15.3
GPT_merge	0.64	0.22	0.16	0.34	0.73	0.32	0.29	0.45	0.2 / 3.2	0.2 / 3.2	0.2 / 3.2	0.2 / 3.2	7.3	6.9	7.5	7.4
RAG_seq	0.25	0.11	0	0.12	0.36	0.04	0	0.13	**0.01 / 0.14**	**0.02 / 0.3**	**0.03 / 0.43**	**0.02 / 0.29**	**3.3**	**5.2**	**6.9**	**5.1**
RAG_tree	0.36	0.23	0	0.2	0.12	0.01	0	0.04	0.03 / 0.49	0.04 / 0.6	0.05 / 0.88	0.04 / 0.66	5.9	8.9	12.3	8.9
ZenDB	**0.89**	**0.72**	**0.61**	**0.73**	**0.86**	**0.79**	**0.83**	**0.84**	0.02 / 0.43	0.04 / 0.59	0.04 / 0.68	0.03 / 0.53	5.1	7.2	8.8	7

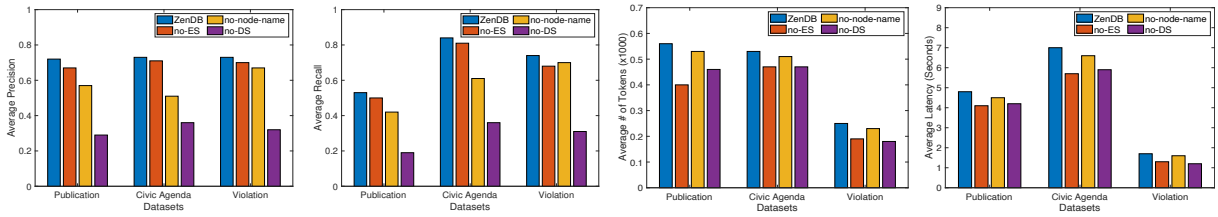
Table 4: Average Precision, Recall, Cost / # of Tokens and Latency of Strategies Per Query, Per Document, in Civic Dataset. (GPT-4-32k is Used.)

Strategies	Precision				Recall				Cost ($) / Tokens (× 1000)				Latency (Seconds)			
	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg	QG1	QG2	QG3	Avg
GPT_single	**0.71**	**0.65**	**0.76**	**0.71**	**0.9**	**0.67**	**0.75**	**0.77**	0.2 / 3.7	0.31 / 5.2	0.43 / 7.1	0.3 / 5.3	4.9	6.2	7.3	6.1
GPT_merge	**0.7**	0.56	0.62	0.66	0.8	0.6	**0.77**	**0.72**	0.2 / 3.7	0.2 / 3.7	0.2 / 3.7	0.2 / 3.7	4.8	5	5.1	5
RAG_seq	0.61	0.31	0.17	0.36	0.67	0.22	0.26	0.38	**0.01 / 0.12**	**0.01 / 0.19**	**0.01 / 0.23**	**0.01 / 0.18**	**0.9**	**1.3**	**1.7**	**1.3**
RAG_tree	0.58	0.36	0.24	0.2	0.39	0.5	0.17	0.17	0.02 / 0.25	0.02 / 0.38	0.03 / 0.41	0.02 / 0.35	2.1	2.7	3.1	2.6
ZenDB	**0.79**	**0.67**	**0.72**	**0.73**	**0.87**	0.62	**0.73**	**0.74**	**0.01 / 0.19**	**0.02 / 0.26**	**0.02 / 0.3**	**0.02 / 0.25**	1.4	1.7	2.1	1.7

Table 5: Average Precision, Recall, Cost / # of Tokens and Latency of Strategies Per Query, Per Document, in Notice Violation Dataset. (GPT-4-32k is Used.)

an entire document at a time. Table 2 reports our metrics of interest on the three datasets, while Table 3, Table 4, and Table 5 provide a breakdown per dataset. We first note that ZenDB achieves comparable precision and recall to GPT_single on the publication and notice datasets. Notably, ZenDB surpasses GPT_single in the civic dataset, ***improving precision by 28% and recall by 39%***, due to this dataset's complex semantic structure, which poses challenges for GPT_single in generating high-quality responses. ZenDB's approach of querying based on SHTs, focuses LLM attention on portions of documents at a time, thereby enhancing performance. We also observe that combining multiple predicates into a single prompt makes it more difficult for the LLM to provide the correct answer, resulting in performance degradation. On the cost and latency front, ZenDB significantly reduces both relative to GPT_single and GPT_merge. Specifically, ZenDB achieves ***cost savings of approximately 29×, 10×, and 4×*** for the publication, civic, and notice datasets respectively. It's noteworthy that ZenDB's cost savings increase with document size, as the number of tokens it uses is somewhat independent of document size. Instead, it relies on the size of the summary and the number of levels of the SHTs explored during execution, which are controllable factors. Accordingly, we observe varying levels of latency savings with ZenDB, ***up to a 4× reduction*** across datasets.

Experiment 2: ZenDB vs. RAG-only Strategies. When compared with RAG_seq and RAG_tree, we observe that RAG_seq achieves significant cost and latency savings compared to GPT-only strategies. However, relying solely on retrieving physical chunks based on embedding similarity as in RAG, fails to accurately identify the appropriate text spans related to the queries, leading to a substantial degradation in precision and recall. While ZenDB incurs a slightly higher cost, it offers substantial advantages over RAG-based approaches thanks to the use of semantic structure, with ***increases in precision by up to 61% and recall by up to 80%***. RAG_tree generally shows slight improvements in precision and recall over RAG_seq, but it similarly falls short of ZenDB for a similar reason. Its use of tree-style physical chunking often fails to accurately identify the appropriate text spans. Moreover, the



(a) Average Precision (b) Average Recall (c) Average # of Tokens (× 1000) (d) Average Latency (Seconds)

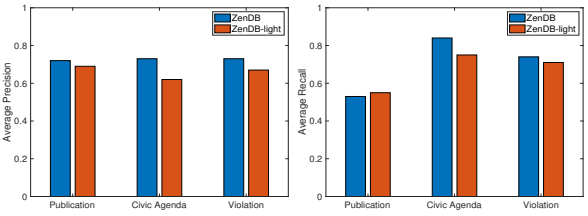
Figure 11: The Effect of Summary Construction to Performance of ZenDB in Real Datasets.

Datasets (# of Docs)	# of Nodes	# of Layers	Cost ($) /Tokens	Latency
Publication (100)	13.4	2.8	0.05 / 1.8k	6min
Civic (41)	32.1	2.9	0.01 / 0.36k	1min
Violation (80)	8.9	2.2	0.01 / 0.32k	1min

Table 6: SHT Construction. (GPT-4 is Used.)

Datasets	Cost ($) / Tokens	FP	FN	Latency
Publication (100)	0.048 / 95.4k	0	0	7 min
Civic (41)	0.005 / 10.1k	0.08	0	3 min
Violation (80)	0.005 / 8.9k	0.04	0	2 min

Table 7: Table Population. (GPT-3.5-Turbo is Used.)



(a) Precision (b) Recall

Figure 12: ZenDB vs. ZenDB-light: Precision and Recall.

exhaustive summary construction and usage in RAG_tree results in higher cost and latency compared to ZenDB.

Experiment 3: Data Preparation. Next, we examine two phases within ZenDB happening prior to queries, SHT construction and table population, and compare it to the costs of online queries.

Experiment 3.1: SHT Construction. We present the average number of nodes and layers per SHT, and the *total* cost, number of tokens, and latency on three datasets, in Table 6. SHT construction is an offline process, making latency at the level of minutes not problematic. The cost is affected by the number of distinct templates in the datasets. ZenDB uses LLMs to verify headers for SHT generation for one document per template, with the remaining SHTs created through visual pattern matching. The cost is further reduced by sampling the phrase clusters. In the publication dataset, the publications originate from 6 conferences, whereas the other two datasets follow a consistent template. Therefore, the publication dataset has a higher cost than the others, although all costs are minimal.

Experiment 3.2: Table Population. When users define a DTable, ZenDB populates the system-defined attributes using LLM-based and rule-based approaches. Table 7 presents the *total* cost and number of tokens (we use GPT-3.5-Turbo)[4], with additional latency and

[4]When the context size of a node exceeds the token limit (e.g., the root node in publication dataset), we use NLTK [10] to summarize the context and adjust the summary size to approximately match the token limit of a prompt.

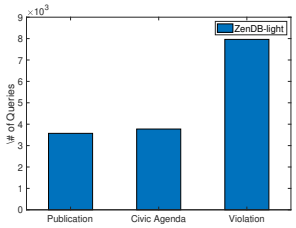


Figure 13: # of Queries on 1 Document by 1 $.

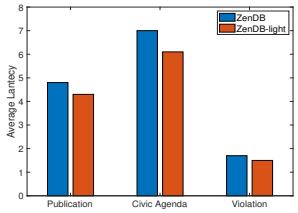


Figure 14: ZenDB VS ZenDB-light: Latency.

quality results. In particular, to show the quality of table population, let $ts_g(e)$ and $ts_p(e)$ be the text span of an entity e (a table or a tuple) in the ground truth and predicted by ZenDB, respectively. We label $ts_g(e) \subset ts_p(e)$ as a false positive (FP), indicating that the predicted text span contains the true text span but is larger, which is acceptable since it doesn't miss the correct answers will be refined by the tree-search algorithm. In constrast, $(ts_p(e) \subset ts_g(e)) \vee (ts_p(e) \cap ts_g(e)) = \emptyset)$ is considered a false negative (FN) because the predicted text span does not encompass all the true text spans, potentially resulting in missed answers. Notably, ZenDB demonstrates a low FP rate in the violation and civic agenda datasets, showcasing the effectiveness of the approach. The cost incurred in this step is minimal, thanks to the use of the affordable LLM GPT-3.5-Turbo (around 100x cheaper than GPT-4).

End-to-end cost comparison: ZenDB vs. Others. Although ZenDB incurs costs to construct SHTs and populate tables before a query arrives, these costs are minimal, totaling 0.1, 0.015, and 0.015 dollars for the publication, civic agenda, and notice of violations datasets, respectively. ***Even if we ran just a single query subsequently, we would have lower end-to-end costs*** for ZenDB compared to GPT-single and GPT-merge, with the loading costs getting amortized across queries.

Experiment 4: The Effect of Summary Construction in ZenDB. We examine the effect of summary construction on ZenDB performance in Figure 11. Recall that in Section 6.2, the summary of each node v in a SHT consists of three components: an extractive summary (ES), the phrases of v and its ancestors (node-name), and the top-1 sentence related to a given query predicate or projection within the text span of v (DS, i.e., Dynamic Summary). We explored three variations of ZenDB by removing one component at a time from the summary: no-ES, no-node-name, and no-DS (e.g., no-ES refers to the strategy that excludes the extractive summary from the summary of the node). We observe that the extractive summary impacts the quality of query answers (i.e., precision and recall) the least, while both dynamic summaries and node names (i.e.,

the header phrases) affect performance more significantly. Node names provide useful metadata that adds more context for the LLM, helping refine the search space. The dynamic summary plays a critical role in summary construction by not only identifying the relevant nodes but also retrieving the text span most related to the given query. We also note that storing node names has a minimal impact on cost and latency due to their compact size. In contrast, both extractive and dynamic summaries have a greater size, though they still represent a relatively small portion of the overall cost and latency.

Experiment 5: ZenDB Driven by A Cheaper LLM: GPT-3.5-Turbo. We next study the impact of replacing the more expensive LLM used in ZenDB, GPT-4-32k, with an almost 100× cheaper LLM, GPT-3.5-turbo, when evaluating queries. We denote this version as ZenDB-light. In Figure 12, ***ZenDB-light exhibits approximately a 7% decrease in precision and a 3% decrease in recall compared to ZenDB, at 100× lower cost***. This demonstrates that by refining the text span that ZenDB uses for evaluating queries, as opposed to the entire complex document, ZenDB is able to provide a much simpler and more precise context for LLMs to evaluate. This makes it easier for less-advanced but cheaper models like GPT-3.5-turbo to not just process the entire text span, but also answer the query accurately. We report the average number of SQL queries that can be executed on a single document by spending 1 dollar using ZenDB-light, in Figure 13. ZenDB-light can run approximately 3.5k, 3.7k, and 8k SQL queries with 2 predicates and one projection on average in one document within budget for the publication, civic agenda reports, and notices of violations, respectively, demonstrating the practicality of ZenDB-light.

8 RELATED WORK

We now survey related work on querying unstructured data.

Text-to-Table Extraction. One approach to querying unstructured data is by simply extracting unstructured data into tables, following which they are queried as usual. This approach is followed by Google DocumentAI [4] and Azure Document Intelligence [5], as well as approaches such as text-to-table [61]. Using an LLM to populate entire tables upfront can be expensive and error-prone on large and complex document collections as in our case. Evaporate [18] uses an LLM to infer schema, and then populate tables, using synthesized rules if possible. Simple extraction rules, such as ones generated by Evaporate, are not applicable in our setting.

Retrieval-Augmented Generation (RAG). RAG techniques [20, 34, 41, 60], help identify smaller text portions that are most relevant to a given query in order to fit into finite context windows, reduce cost, and in some cases improve accuracy. Most techniques use fixed granularity chunking policies and don't account for semantic structure, while recent extensions rely on potentially expensive recursive summarization to build a hierarchy [9, 52]. We showed that this RAG_tree approach suffers from the same issues as vanilla RAG. The leaf nodes still use fixed size chunks that are divorced from semantics, and thus fail to find relevant text segments. In comparison, ZenDB leverages semantic structure to boost precision and recall by up to 61% and 80%.

Multi-Modal Databases. Recent work creates of multi-modal databases [24, 35, 55, 56, 58] that support SQL-like interfaces over text, images, and/or video. However, they all apply LLMs or other pre-trained models to entire documents at a time, and are thus limited to simple, small documents. This is equivalent to our vanilla LLM approach, which is expensive and not very accurate. Other work [31] has used interactive query processing to improve query results through user feedback. None of these approaches have explored the use of semantic structure to reduce cost and improve accuracy.

Natural Language Interfaces to Data. Supporting natural language querying over structured data is a long-standing question in the database community; a recent survey is one by Quamar et al. [50]. While the database community has been working on this problem for over a decade, e.g., [40], LLMs have dominated recent benchmarks [21, 42]. In our work, we instead focus on the inverse problem of structured (SQL) queries over unstructured data—but this line of work could aid the first step of SQL query construction.

LLMs meet Data Management. LLMs potentially disrupts the field of data management [29], but the first step is to actually understand tables. Recent work [25, 28, 62] explores how well LLMs understand tabular data, and representing knowledge learned by the LLM as structured data [51, 59]. Many data management problems have been revisited, including query rewriting [44], database tuning [57], data preprocessing [63], data and join discovery [26, 27, 36], data profiling [33], and data wrangling [23, 43, 48]. Some recent work has also explored how well LLMs can generate tables [54]. ZenDB also uses LLMs, but to a new setting: document analytics.

Structured Extraction. Structured extraction from web pages, pdfs, and images has a long history of work. For instance, Snowball [17] proposed structured extraction over the open web, and leverage common techniques such as wrapper induction [38, 46] which also leverage the hierarchical structure of HTML documents and headings. In contrast, ZenDB takes as input PDFs, which are often not hierarchically encoded. Other works, such as Shreddr [22] extract from images of forms where the templates are identical, and focus on efficient use of crowd workers. These are also relevant due to the similarities between LLMs and crowdsourcing [49].

9 CONCLUSION

We presented ZenDB, a document analytics system that leverages templatized structure present in documents in a collection to support cost-efficient and accurate query processing. During ingest, ZenDB extracts structure from documents in the form of SHTs, guaranteeing that the results are correct for well-formatted documents. Then, during table creation, ZenDB maps tuples to nodes in the SHT, with attribute values to be populated during querying. ZenDB supports SQL queries on user-defined document tables, applying predicate reordering and pushdown, and projection pull-up techniques, coupled with a summary-based tree-search approach to optimize query processing. Across multiple domains, ZenDB provides a compelling trade-off point relative to LLM-only or RAG based approaches. In future work, we plan to study the setting where there are no templates or when the templates are very noisy, as well as expand the space of SQL queries supported. In addition, we envision a rich design space for user interfaces to allow users to explore the results of ZenDB queries alongside their provenance.

REFERENCES

[1] 2019. *https://www.forbes.com/sites/rkulkarni/2019/02/07/big-data-goes-big/?sh=45b1c73420d7*.

[2] 2021. *https://mitsloan.mit.edu/ideas-made-to-matter/tapping-power-unstructured-data*.

[3] 2023. *gemini.google.com*.

[4] 2023. *https://cloud.google.com/document-ai?hl=en*.

[5] 2023. *https://cloud.google.com/document-ai?hl=en*.

[6] 2023. *https://openai.com/pricing*.

[7] 2023. *https://www.anthropic.com/news/claude-3-family*.

[8] 2023. *https://www.forbes.com/sites/stevemcdowell/2023/03/09/komprise-unleashes-fresh-insights-about-your-unstructured-data/?sh=5f444c474aa9*.

[9] 2023. *https://www.llamaindex.ai/*.

[10] 2023. *https://www.nltk.org/*.

[11] 2024. *http://personal-informatics.depstein.net*.

[12] 2024. *https://primis.phmsa.dot.gov/enforcement-data/cases/NOPV*.

[13] 2024. *https://pypi.org/project/pdfplumber/0.1.2/*.

[14] 2024. *https://www.malibucity.org/AgendaCenter*.

[15] Serge Abiteboul. 1997. Querying semi-structured data. In *Database Theory—ICDT'97: 6th International Conference Delphi, Greece, January 8–10, 1997 Proceedings 6*. Springer, 1–18.

[16] Serge Abiteboul, Peter Buneman, and Dan Suciu. 2000. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann.

[17] Eugene Agichtein and Luis Gravano. 2000. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the fifth ACM conference on Digital libraries*. 85–94.

[18] 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.

[19] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508* (2023).

[20] Deng Cai, Yan Wang, Lemao Liu, and Shuming Shi. 2022. Recent advances in retrieval-augmented text generation. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 3417–3419.

[21] Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, et al. 2023. Dr. spider: A diagnostic evaluation benchmark towards text-to-sql robustness. *arXiv preprint arXiv:2301.08881* (2023).

[22] Kuang Chen, Akshay Kannan, Yoriyasu Yano, Joseph M Hellerstein, and Tapan S Parikh. 2012. Shreddr: pipelined paper digitization for low-resource organizations. In *Proceedings of the 2nd ACM Symposium on Computing for Development*. 1–10.

[23] Zui CHen, Lei Cao, Sam Madden, Ju Fan, Nan Tang, Zihui Gu, Zeyuan Shang, Chunwei Liu, Michael Cafarella, and Tim Kraska. 2023. Seed: Simple, efficient, and effective data management via large language models. *arXiv preprint arXiv:2310.00749* (2023).

[24] 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. In *Conference on Innovative Data Systems Research, CIDR*. 8–151.

[25] Tianji Cong, Madelon Hulsebos, Zhenjie Sun, Paul Groth, and HV Jagadish. 2023. Observatory: Characterizing Embeddings of Relational Tables. *arXiv preprint arXiv:2310.07736* (2023).

[26] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2022. Turl: Table understanding through representation learning. *ACM SIGMOD Record* 51, 1 (2022), 33–40.

[27] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2022. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. *arXiv preprint arXiv:2212.07588* (2022).

[28] Xi Fang, Weijie Xu, Fiona Anting Tan, Jiani Zhang, Ziqing Hu, Yanjun Qi, Scott Nickleach, Diego Socolinsky, Srinivasan Sengamedu, and Christos Faloutsos. 2024. Large Language Models on Tabular Data–A Survey. *arXiv preprint arXiv:2402.17944* (2024).

[29] Raul Castro Fernandez, Aaron J Elmore, Michael J Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How large language models will disrupt data management. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3302–3309.

[30] Boris Glavic et al. 2021. Data provenance. *Foundations and Trends® in Databases* 9, 3-4 (2021), 209–441.

[31] Benjamin Hättasch, Jan-Micha Bodensohn, Liane Vogel, Matthias Urban, and Carsten Binnig. 2023. WannaDB: Ad-hoc SQL Queries over Text Collections. In *BTW 2023*. Gesellschaft für Informatik eV, 157–181.

[32] Joseph M Hellerstein and Michael Stonebraker. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 267–276.

[33] Zezhou Huang and Eugene Wu. 2024. Cocoon: Semantic Table Profiling Using Large Language Models. *arXiv preprint arXiv:2404.12552* (2024).

[34] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299* (2022).

[35] Saehan Jo and Immanuel Trummer. 2023. Demonstration of ThalamusDB: Answering Complex SQL Queries with Natural Language Predicates on Multi-Modal Data. In *Companion of the 2023 International Conference on Management of Data*. 179–182.

[36] Moe Kayali, Anton Lykov, Ilias Fountalis, Nikolaos Vasiloglou, Dan Olteanu, and Dan Suciu. 2023. CHORUS: foundation models for unified data discovery and exploration. *arXiv preprint arXiv:2306.09610* (2023).

[37] Mei Kobayashi and Koichi Takeda. 2000. Information retrieval on the web. *ACM computing surveys (CSUR)* 32, 2 (2000), 144–173.

[38] Nicholas Kushmerick. 2000. Wrapper induction: Efficiency and expressiveness. *Artificial intelligence* 118, 1-2 (2000), 15–68.

[39] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[40] Fei Li and Hosagrahar V Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 709–712.

[41] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. 2022. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110* (2022).

[42] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2024).

[43] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *arXiv preprint arXiv:2004.00584* (2020).

[44] Jie Liu and Barzan Mozafari. 2024. Query Rewriting via Large Language Models. *arXiv preprint arXiv:2403.09060* (2024).

[45] 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.

[46] Tomohiro Manabe and Keishi Tajima. 2015. Extracting logical hierarchical structure of HTML documents based on headings. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1606–1617.

[47] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. 1997. Lore: A database management system for semistructured data. *ACM Sigmod Record* 26, 3 (1997), 54–66.

[48] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).

[49] Aditya G Parameswaran, Shreya Shankar, Parth Asawa, Naman Jain, and Yujie Wang. 2023. Revisiting prompt engineering via declarative crowdsourcing. *arXiv preprint arXiv:2308.03854* (2023).

[50] Abdul Quamar, Vasilis Efthymiou, Chuan Lei, Fatma Özcan, et al. 2022. Natural language interfaces to data. *Foundations and Trends® in Databases* 11, 4 (2022), 319–414.

[51] Mohammed Saeed, Nicola De Cao, and Paolo Papotti. 2023. Querying large language models with SQL. *arXiv preprint arXiv:2304.00472* (2023).

[52] 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*.

[53] Amit Singhal et al. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[54] Xiangru Tang, Yiming Zong, Yilun Zhao, Arman Cohan, and Mark Gerstein. 2023. Struc-Bench: Are Large Language Models Really Good at Generating Complex Structured Data? *arXiv preprint arXiv:2309.08963* (2023).

[55] James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Halevy. 2021. Database reasoning over text. *arXiv preprint arXiv:2106.01074* (2021).

[56] James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Halevy. 2021. From natural language processing to neural databases. In *Proceedings of the VLDB Endowment*, Vol. 14. VLDB Endowment, 1033–1039.

[57] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.

[58] Matthias Urban and Carsten Binnig. 2023. Towards Multi-Modal DBMSs for Seamless Querying of Texts and Tables. *arXiv preprint arXiv:2304.13559* (2023).

[59] Matthias Urban, Duc Dat Nguyen, and Carsten Binnig. 2023. OmniscientDB: a large language model-augmented DBMS that knows what other DBMSs do not know. In *Proceedings of the Sixth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–7.

[60] Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023. Learning to Filter Context for Retrieval-Augmented Generation. *arXiv preprint arXiv:2311.08377* (2023).

[61] Xueqing Wu, Jiacheng Zhang, and Hang Li. 2021. Text-to-table: A new way of information extraction. *arXiv preprint arXiv:2109.02707* (2021).

[62] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314* (2020).

[63] Haochen Zhang, Yuyang Dong, Chuan Xiao, and Masafumi Oyamada. 2023. Large language models as data preprocessors. *arXiv preprint arXiv:2308.16361* (2023).