KathDB: Explainable Multimodal Database Management System with Human-AI Collaboration

Vision Paper

Guorui Xiao, Enhao Zhang, Nicole Sullivan, Will Hansen, and Magdalena Balazinska
University of Washington
Seattle, WA, USA
{grxiao,enhaoz,nsulliv,willnh,magda}@cs.washington.edu

ABSTRACT

Traditional DBMSs execute user- or application-provided SQL queries over relational data with strong semantic guarantees and advanced query optimization, but writing complex SQL is hard and focuses only on structured tables. Contemporary multimodal systems (which operate over relations but also text, images, and even videos) either expose low-level controls that force users to use (and possibly create) machine learning UDFs manually within SQL or offload execution entirely to black-box LLMs, sacrificing usability or explainability. We propose KathDB, a new system that combines relational semantics with the reasoning power of foundation models over multimodal data. Furthermore, KathDB includes human-AI interaction channels during query parsing, execution, and result explanation, such that users can iteratively obtain explainable answers across data modalities.

1 INTRODUCTION

Traditional relational database management systems (DBMSs), such as PostgreSQL[1] and MySQL[2], were designed to store tabular data and answer queries expressed in SQL. Their query optimizers rely on relational algebra and a cost–based model, providing clear query semantics and high efficiency [1]. Yet these systems offer no native support for other data modalities (text, images, audio, ...), which modern data-intensive applications in science, healthcare, industry, and media now regard as first-class citizens [12, 14, 21, 22]. Consider the following natural language (NL) query: *"Sort the films in the table by how exciting they are, but the poster should be 'boring'."* The database consists of a relational table that stores movie metadata such as title and release year, along with a column containing the movie's plot text and another column holding its poster image, represented either by pixel values or, more commonly, by a file path to the image stored on disk. A classical DBMS cannot evaluate such a query, because computing "excitement" scores over plots requires interpreting unstructured text while labeling posters as "boring" requires understanding poster images. Additionally, the NL query is ambiguous about the user's intent: does the user want to filter posters based on whether they are boring or should boring be part of the ranking? Such ambiguities makes it hard for a system to reason about the query and to perform query evaluation.

Recent work in machine learning (ML), most notably Large Language Models (LLMs) [4] and Vision Language Models (VLMs) [5], has motivated a new class of *multimodal* database management systems [8, 12, 14, 15, 19, 21, 22, 24, 25, 28]. Some of those systems [24, 28] require users to manually compose SQL queries with optional ML user-defined functions (UDFs), providing flexibility yet remaining difficult for non-expert users and tedious for experts. Other systems [8, 12, 14, 15, 19, 21, 22, 25] delegate semantic interpretation entirely to foundation model operators. At query time, they invoke models on each record by prompting it to generate the target attribute (e.g., an `is_exciting` flag) and treat the model outputs as the final query result. While straightforward, this paradigm has one major drawback: The user receives only a final answer without information on how each tuple was derived because the generation process bypasses the relational layer.

Users thus face a trade-off: AI-assisted SQL engines that demand user effort, or powerful but opaque multimodal systems. To reconcile these worlds, we introduce KathDB[3], an explainable *multimodal DBMS* powered by LLM-driven human-AI collaboration. Specifically, KathDB makes the following three key contributions:
(1) Unified semantic layer based on the relational model. Unlike previous multimodal systems [8, 9, 12, 21, 22, 24, 25] that expose raw data to users, KathDB introduces a unified *relational* semantic layer of *views* over data, giving the data a systematic, relational representation. This design has several advantages: First, it unifies heterogeneous modalities under a single relational abstraction, enabling systematic, cost-based evaluation of cross-modal user queries. Second, the layer combines modality-specific powerful ML operators with the semantic guarantees of a traditional DBMS. Finally, the relational representation gives KathDB the ability to track fine-grained lineage, allowing every result tuple to be traced back to its exact source records, and enabling better explainability.
(2) Function-as-operator (FAO) query planning and execution. Similar to other multimodal systems [8, 14, 21], KathDB transforms an NL request into a workflow of smaller steps. Each step corresponds to a transformation (e.g., similarity search) over data, or the combination of intermediate results from previous steps (e.g., join over views). This decomposition improves the understanding and verification of user intent. Importantly, KathDB does this decomposition in three steps: it first converts the user NL query into a *query sketch*, which is a step-by-step decomposition of the query, with a natural language description of the intent of each step (e.g.,

[1] https://www.postgresql.org
[2] https://www.mysql.com/

[3] **kath**arós means clear and clean in Greek.

"Check the `Objects` table associated with each poster image to determine if a movie poster contains objects associated with excitement (e.g., weapons, motorcycles, etc.)"). Note that the keyword list is also generated by the LLM. During execution, however, if users are not satisfied with specific logic (e.g., the generated keyword list), they may provide feedback to KathDB, which automatically updates the logic accordingly. It then converts the query sketch into a logical plan where each node is a function, with a signature and description (e.g., "`gen_excitement_score()`"). Third, it generates the body of each function, associating a version identifier with each implementation (e.g., call a specific embedding model to embed the extracted objects in a poster image, embed the concepts from the generated keyword list, compute their similarity, and finally aggregate these values into an 'excitement' score per movie). This design has several advantages. First, it enables the system to explore different mappings between query sketch, logical plan, and physical plan (e.g., one step in a query sketch can correspond to multiple logical functions and logical function can further be decomposed in multiple physical functions or vice versa). Second, separating signature declaration from body synthesis lets KathDB explore different interpretations for the same sub-task (e.g., interpreting "exciting movies" as action movies, recent releases, or award-winning movies). Third, splitting a query into small functions and generating each function separately reduces common problems in long generation such as hallucination [26] and error propagation [16]. Fourth, each function is assigned an identifier and a version tag, enabling the system to record how each output tuple is derived through a sequence of data transformations (e.g., when a new column is produced by an aggregation such as a count). Each transformation is implemented as a function, and these functions are persisted locally on disk. This gives KathDB support for fine-grained lineage tracking, allowing it to trace any tuple in the final query result back through the intermediate materialized tables and the specific transformations (or functions) that produced it.

Finally, the FAO design enables both rich logical rewrites and cost-based physical optimization. Recent systems for unstructured and multimodal data [13, 15, 18, 19] treat each semantic operator as a black-box prompt and search over models, prompts, or cascades to trade off cost and accuracy. In KathDB, each FAO *signature* serves as a logical operator, while each concrete implementation (e.g., prompting technique, coding variants) serves as a physical operator. This separation allows KathDB to attach cost and accuracy statistics to individual FAO implementations and compare alternatives for the same sub-task under a unified cost model, optimizing query accuracy and token cost subject to constraints (e.g., minimal user effort when rating sampled query results during plan profiling). Because FAOs are composed into an explicit plan, the optimizer can jointly explore logical rewrites (e.g., predicate pushdown, fusing operators that share VLM calls) and physical choices (e.g., model cascades), much like traditional DBMS optimizers, but now over multimodal, model-driven operators.

(3) Rich user interactions for query clarification, debugging, and explanation. In KathDB, we explore novel human-AI interactions for multimodal data management. Unlike traditional approaches to data management, user-system interaction does not have to be limited to a query-result pair: it can be iterative. Toward this goal, we build several dedicated channels for multi-turn clarifications between the user and KathDB during the query interpretation, execution, and result explanation stages. This design brings several advantages: First, in the query parsing phase, vague user intent may cause KathDB's interpretation to differ from the user's expectations. An interactive channel helps KathDB better understand the query and draft more accurate plans. Second, in the query execution phase, unlike traditional DBMSs that abort on runtime errors, KathDB fixes errors on-the-fly by exploring alternative function implementations and optionally involving users in debugging. Finally, after query execution, KathDB enhances transparency by explaining how a tuple or any intermediate result was derived using the lineage information described above. Thus, we envision that KathDB will enhance query accuracy and strengthen user trust, despite using black-box LLMs as modules.

In summary, this vision paper makes the following contributions:

(1) We develop KathDB, the first multimodal DBMS that combines traditional relational guarantees with modern AI operators and rich human-AI interactions to deliver trustworthy and explainable multimodal data management.
(2) We unify text, images, and video under a relational layer of views with version information, taking advantage of relational semantics while enabling fine-grained lineage across modalities.
(3) We introduce function-as-operator (FAO) query planning and execution, compiling each operator into a reusable and explainable function for modular and semantically flexible, multimodal pipelines.
(4) We provide conversational channels that let users interactively clarify NL queries, debug FAO implementations, and explain query results despite KathDB containing LLM-powered modules.

KathDB takes an important step toward integrating AI models with DBMSs while ensuring explainability. This paper presents our vision, design, and preliminary results for KathDB, which we are developing at the University of Washington.

2 KATHDB ARCHITECTURE

We briefly describe the main components of KathDB (Figure 1).

2.1 Query Parser with Human-AI Verification

A query parser in KathDB converts a user's NL request into an executable logical plan. There are two sub-modules within the query parser: **(1) NL parser.** Inspired by recent work showing that chain-of-thought (CoT) reasoning improves LLM performance [23], the *NL parser* first generates a *query sketch*, a step-by-step description of the intended execution logic expressed entirely in NL. The *query sketch* deliberately avoids exposing operator-level details (e.g., function signatures or intermediate schemas) and thus remains one abstraction level above the final logical plan. This higher-level representation is easier for users to inspect and edit as we discuss further in Section 5, yet still provides sufficient structure for the downstream compiler. **(2) Logical Plan Generator.** Given a query sketch as input, the *logical plan generator* uses the system catalog as additional context and expands each step in the query sketch into a logical plan node equipped with a function signature. For

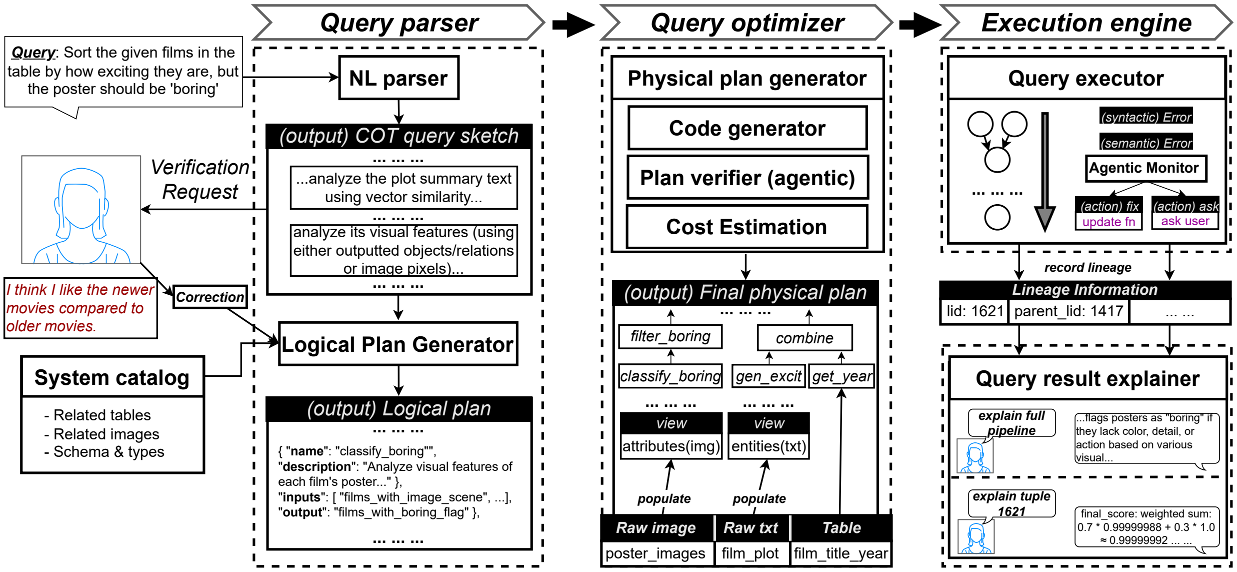


Figure 1: Overview of KathDB. KathDB consists of three main components: a query parser, an optimizer, and an execution engine. KathDB accepts an NL query from the user as input. It generates a query sketch, then a logical plan, and then a physical plan, interacting with the user to seek clarifications as needed. KathDB executes the physical query plan while fixing errors on-the-fly during execution and recording lineage information. Finally, KathDB can explain query results at different granularities.

example, given the query in Figure 1, and a query sketch step: *"Analyze poster visual features using both extracted objects and image pixels to determine if the poster appears 'boring' (e.g., lacks vivid colors, few objects, little action, plain background)"*, the logical plan generator produces a node in the logical plan shown in Figure 3 named `classify_boring`. We further describe the node's schema and its generation in Section 4.

In the current prototype implementation, we assume a simple database schema containing the relevant tables and columns to use in the query. We are exploring extensions to more complex schemas, where KathDB will need to automatically combine table lookups with similarity-based search to determine the relevant tables and columns to use in a query. We do not discuss this extension in this paper.

2.2 Query Optimizer

KathDB's query optimizer translates a logical plan, in which each node contains only function signatures and schema-related information, into a low-cost physical plan, where the body of each function has been generated. A function can contain a SQL query over a table, a view population using machine learning models, a vector-based similarity search for semantic keyword matching, and more. Since the KathDB optimizer generates each function independently instead of producing the entire physical query plan at once, it reduces autoregressive error propagation [16]. Additionally, it can generate these functions efficiently, in parallel. We describe the details of function generation in Section 4 and briefly discuss cost-based optimization with our proposed FAO paradigm.

2.3 Execution Engine And Explainer

As shown on the right of Figure 1, KathDB 's execution engine instantiates the physical plan, produces the result set, and exposes a channel for result explanations. Runtime errors are either *syntactic*, which raises exceptions, or *semantic*, where the LLM doubts that the output matches the user intent; KathDB self-repairs the former and seeks user clarification for the latter (Section 5). After execution, users can ask NL questions (e.g., how a particular tuple was derived or why an operator behaved as it did) about any intermediate tuple or the entire pipeline, a key capability enabled by our provenance model (Section 3) and the interactive debugger (Section 5).

3 KATHDB DATA MODEL

In this section, we describe the data model KathDB uses to align disparate data modalities under a unified relational schema. It has three main advantages: (1) it supports a variety of queries; (2) it provides a well-structured semantic foundation for queries; (3) it facilitates explainability and verification. Designing such a schema at the right level of granularity is non-trivial, as each modality has distinct characteristics (e.g., video combines spatial and temporal signals, whereas text may include multiple co-referring mentions to the same entity). An overly fine-grained schema may lead to high complexity, make view population expensive, and introduce attributes that may rarely matter during actual query execution. Conversely, an over-simplified schema may fail to capture essential semantics and thus might not reliably answer queries correctly. Requiring a user-defined schema adds unwanted burden, as our goal is for users to reason at the level of natural language. Our design must therefore achieve a balance: expressive enough to model

Table 1: Relational representation of image/video content.

Objects(vid, fid, oid, **lid**, cid, x_1, y_1, x_2, y_2)
Relationships(vid, fid, rid, **lid**, oid$_i$, pid, oid$_j$)
Attributes(vid, fid, oid, **lid**, k, v)
Frames(vid, fid, **lid**, pixels)

Table 2: Relational representation of text content.

Entities(did, eid, **lid**, cid)
Mentions(did, sid, mid, **lid**, eid, span$_1$, span$_2$)
Relationships(did, sid, rid, **lid**, eid$_i$, pid, eid$_j$)
Attributes(did, sid, eid, **lid**, k, v)
Texts(did, **lid**, chars)

Table 3: Unified provenance schema.

Lineage(lid, parent_lid, src_uri, func_id, ver_id, data_type, ts)

modality-specific nuances, yet compact, tractable, and extensible to future modalities.

Images and Videos as Scene Graphs. Inspired by EQUI-VOCAL [27], KathDB adopts a scene graph [10] data model that represents visual content as objects interacting in space and time. Images are treated as videos with a single frame. Table 1 summarizes the relational schema that supports this model. Video frames are uniquely identified by the pair (vid, fid), which identify the video and the frame. An **Object** is defined as

(vid, fid, oid, lid, cid, x_1, y_1, x_2, y_2),

where oid is the unique id for the object, cid the class label (e.g., person), and (x_1, y_1, x_2, y_2) the upper-left and bottom-right bounding box coordinates. lid refers to the lineage, which we explain later in this section. An object can have **Relationships** with another object, defined as

(vid, fid, rid, lid, oid_i, pid, oid_j).

Here, rid is a unique identifier for a relationship in the frame, and pid is the relationship class id between two objects.

Each object can also have **Attributes**

(vid, fid, oid, lid, k, v),

where k is the attribute key (e.g., "color") and v is the attribute value (e.g., "black"). Finally, the **Frames** table provides a view over the pixels within frames or images. This simple yet powerful scene graph representation enables complex visual reasoning, even when the NL queries are indirect. For example, to find exciting movies, KathDB may label two scenes as dangerous: one showing *"a man jumped off a plane"* and the other *"a dog fell into a pool"*. The scene graph allows KathDB to explain why the latter does not make a movie "exciting."

Text content as text semantic graph. Unlike standalone images, where each object is unique (or videos where each unique object can be tracked across frames), a textual corpus presents additional challenges, such as entity resolution [6, 19]. To address this challenge, we are experimenting with a schema that tries to capture the entity discussed in a document and the mentions that relate to those entities. Table 2 summarizes this text data model. An entity from the **Entities** table corresponds to the system's best identification of individual entities in each document. Each entity can be mentioned multiple times and in different ways: for example through a full name ("Taylor Swift"), a pronoun ("she"), or an indirect reference ("the artist behind the Eras tour"). The schema for an entity is:

(did, eid, lid, cid),

where did is the document id, eid is the entity id shared by all mentions of that entity within the document, and cid is the entity class type (e.g., person). KathDB ensures eid uniqueness within the text corpus, but does not guarantee consistency across documents. A mention from the **Mentions** table consists of a unique mention identifier (mid) occurring in a sentence (sid) of a given document (did), along with its character span (span1, span2). The schema is:

(did, sid, mid, lid, eid, span1, span2),

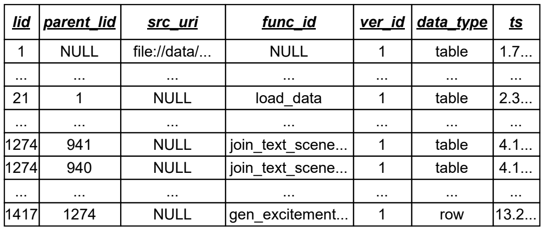
where span1 and span2 mark the start and end character positions of the mention. For example, "Taylor" and "Mrs. Swift" will have two different mid's but refer to the same entity "Taylor Swift." (thus same eid). If a query treats mentions as independent objects without resolving them to their entities, it may yield incorrect results; grouping mentions by entity avoids such errors. Similar to the image scene-graph representation, KathDB defines text **Relationships** (e.g., an entity "Irwin Winkler" can have relationship "director_of" with another movie entity "Guilty by Suspicion") and **Attributes** in key (e.g., movie_budget) value (e.g., 13M) format. Their schemas follow the same pattern:

(did, rid, lid, eid_i, pid, eid_j) for relationships,

(did, eid, lid, k, v) for attributes.

Finally, the **Texts** table provides access to the raw textual content. The lid in every row is a unique lineage id which we discuss later. It is worth noting that a text semantic graph is not the only way to represent unstructured text. In scientific papers, entire sentences expressing claims (e.g., "our model outperforms prior work") may serve as meaningful units, while legal documents often require finer granularity to track specific parties and events. KathDB's relational schema is flexible to these variations, and an important research questions is to explore alternative designs and their impact on query accuracy.

Provenance model. KathDB uses provenance to track how final and intermediate tuples are derived through a sequence of data transformations. Provenance is important in multimodal settings where derived information may come from heterogeneous sources (e.g., a bounding box from an image, a named entity from text, or a computed score). Users often need to understand *why* a tuple appears in the result, and provenance enables KathDB to provide grounded explanations of the derivation process. As shown in Table 3, each row records one edge in the provenance graph: a derived tuple (i.e., child) identifier (lid), an optional input tuple's identifier (parent_lid; NULL for external input data), an optional data



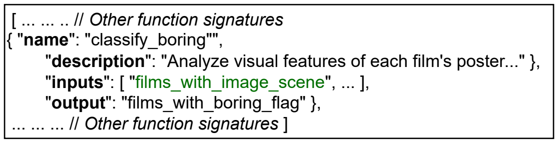
lid	parent_lid	src_uri	func_id	ver_id	data_type	ts
1	NULL	file://data/...	NULL	1	table	1.7...
...	...	...	...	...	...	...
21	1	NULL	load_data	1	table	2.3...
...	...	...	...	...	...	...
1274	941	NULL	join_text_scene...	1	table	4.1...
1274	940	NULL	join_text_scene...	1	table	4.1...
...	...	...	...	...	...	...
1417	1274	NULL	gen_excitement...	1	row	13.2...

Figure 2: Example rows of a lineage table for an output tuple.

path (src_uri; e.g., an image object from an s3 bucket; NULL for intermediate tuples), a function identifier (func_id) with a version number (ver_id; as described in Section 4) that produced the child at a certain timestamp (created_ts), and a lineage data type (data_type). Ingesting a raw table creates a single lineage entry with data_type=table. For each function (including non-relational algebra ones), KathDB also asks the same LLM that generates the function (Section 4) to classify its *dependency_pattern* as one of four types: one_to_one, one_to_many, many_to_one, or many_to_many. The first two indicate single-tuple dependency, allowing KathDB to record row-level lineage. In this case, the executor processes one input tuple at a time: when it reads an input tuple with lid and the function produces one or more output tuples, KathDB sets each output tuple's parent_lid to that input tuple's lid, assigns the output tuple a fresh lid, writes these fields into the tuple, and inserts a provenance entry reflecting this relationship with data_type=row. The latter two indicate wide dependency (e.g., aggregation, sorting), for which KathDB records only table-level lineage with data_type=table, and assume that all input tuples have contributed to all the output tuples. At present, KathDB does not attempt to recover finer-grained multi-tuple provenance for wide-dependency operators.

Figure 2 shows some sample rows from the lineage table for the query in Figure 1. One example lineage entry is the tuple (lid=1417), whose data_type is row. This tuple stores the excitement score for a movie, computed by the one-to-one function gen_excitement_score, which produces exactly one output row for each input row. Its parent is an intermediate result with (lid=1274), whose data_type is table and it is produced by the many-to-many operator join_text_scene_graph. Here, the function join_text_scene_graph joins two tables to associate each movie with the entities extracted from its plot description, and therefore its output is treated as a table-level artifact in the lineage graph. Accordingly, the tuple (lid=1274) has two parent tables, (lid=940) and (lid=941), both of which were previously loaded into the system by other functions (omitted here for brevity).

Some important research questions related to KathDB's provenance model are that lineage tracking adds a significant overhead, so how should KathDB perform tracking without sacrificing much query execution speed? How fine-grained do we need the provenance to be in a multimodal DBMS? Can KathDB's multimodal schema together with lineage provide grounded explanations for complex query evaluation results based on user studies?



```
[ ... ... .. // Other function signatures
{ "name": "classify_boring"",
      "description": "Analyze visual features of each film's poster..." },
      "inputs": [ "films_with_image_scene", ... ],
      "output": "films_with_boring_flag" },
... ... ... // Other function signatures ]
```

Figure 3: Function signature generated by the logical plan generator.

4 FUNCTION-AS-OPERATOR (FAO)

After receiving a *query sketch*, the *logical plan generator* produces a *logical plan* whose nodes are function signatures. The *query optimizer* subsequently instantiates each signature with an executable function body (e.g., an SQL sub-query, a model-based inference routine that populates a view in the relational schema of Section 3, and so on). Each function is stamped with a monotonically increasing ver_id. Whenever the optimizer generates a new implementation for a function, KathDB increments the version ID, leaving earlier versions intact. During execution, every output tuple carries the ver_id of the function that produced it, enabling precise lineage queries, safe roll-backs to a prior version, and the iterative refinement workflows described in Section 5. We refer to this design principle in KathDB as **function-as-operator (FAO)**. There are a few challenges when adopting FAO in KathDB.

Generating function definitions with the *logical plan generator*. To generate functions to evaluate the query, the logical plan generator first produces a tree of function signatures as the "logical plan", an example of which is shown in Figure 3. Each generated signature must contain the necessary information for the query optimizer to later generate the code while not being excessive to confuse the model [11]. Thus the logical plan generator produces each logical plan node strictly following our schema: every generated plan node is emitted in the *exact JSON layout* we defined so the downstream parser can ingest it without any post-processing. Here, name is the identifier of the function; inputs is the list of datasource names that the function will consume (e.g., classify_boring reads from a dataframe named "films_with_image_scene"). A datasource may refer to (i) a base relation already materialized in KathDB, whose schema is recorded in the catalog, or (ii) an intermediate table produced by a preceding node. output declares the table produced by the function. Finally, the description field provides semantic hints to support downstream code synthesis.

Inspired by the principles in [17], we adopt a three-stage agentic workflow comprising a *plan writer*, a *tool user*, and a *plan verifier*. The plan writer combines catalog metadata with the query sketch to draft a tree of logical-plan nodes. A verifier then reads the draft plan with the initial sample data (e.g., sample rows, column attributes, data types) from all related relations; if this snapshot is enough to judge correctness, it approves, otherwise it identifies *specific relations* for which it needs additional information, invokes the tool user, which owns a small set of database utilities (e.g., rows sampler, joinability tester between two tables) to retrieve such information and judge again. Once the verifier is satisfied that the plan has realized the sketch, it forwards the logical plan to the query optimizer discussed next, otherwise it sends hints and the draft plan back to the writer to improve and review it again.

Ensuring function executability with the *query optimizer*. The optimizer implements each logical plan node. Nodes whose input does not depend on other nodes can be compiled in parallel, and for any given signature (e.g., Figure 3) the optimizer may initialize multiple model instances to explore alternative implementations; our current prototype, however, implements functions sequentially. Three specialized agents collaborate on every node: a *coder*, a *profiler*, and a *critic*. The optimizer first extracts column names, types, and sample rows from relations whose names are mentioned in a node's `inputs`. Reading both the sampled rows and node specification, the coder writes a function body. Then, the profiler uses the same set of sampled rows and executes the freshly generated function to ensure it can be executed and records its runtime for optimization purposes. Samples of intermediate results are provided to subsequent agents. If execution raises an exception, instead of aborting the entire query execution, KathDB captures the stack trace, sampled data, parameters, and node metadata, and forwards them to the critic, which proposes a patch instead of aborting the query.

Unlike traditional relational systems, KathDB must profile function implementations *on-the-fly* during query execution, which can slow down the query. Although in practice this overhead is typically dominated by the LLM invocation time, an important research question is how KathDB can reduce online profiling effort (e.g., through offline profiling) to speed up query plan generation.

Ensuring function semantic correctness with the *query optimizer*. The same set of agents also checks that each function *semantically* implements the logical node schema correctly. The critic first inspects the function source, sampled input records, produced output records, and node description to judge whether the results plausibly satisfy the intended semantics. For instance, in the workflow of Figure 1, a scoring function meant to generate a recency score based on user's request might be mistakenly implemented to do the reverse: giving higher score to the older movies. When a mismatch is detected, the critic returns a corrective hint to the coder, which iterates until the output is acceptable. Extending this loop with optional human review for more complex edge cases is one of the research questions that we are exploring.

Cost optimization with the *query optimizer*. The optimizer can perform cost-based optimization at two levels. The first level is logical plan optimization. Here the optimizer may push predicates closer to data sources and merge two function signatures into one to avoid unnecessary intermediate result materialization. These rewrites shape the structure of the logical plan and influence how much work the system must perform later. The second level is physical plan optimization. Here, each logical function signature can be implemented in multiple ways. For example, an image-to-text extraction operator may be instantiated using either a VLM-based implementation or an OCR-based implementation such as Tesseract[4], each represented in KathDB as a distinct function version (`ver_id`). This versioning lets the optimizer choose among multiple concrete implementations of the same logical operator. The optimizer profiles these implementations on sample input records and chooses the one that produces acceptable outputs at the lowest cost.

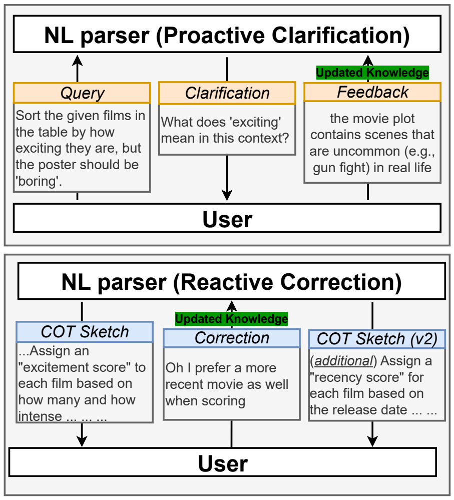

NL parser (Proactive Clarification)
Query
Sort the given films in the table by how exciting they are, but the poster should be 'boring'.
Clarification
What does 'exciting' mean in this context?
Updated Knowledge
Feedback
the movie plot contains scenes that are uncommon (e.g., gun fight) in real life
User
NL parser (Reactive Correction)
COT Sketch
...Assign an "excitement score" to each film based on how many and how intense ... ...
Updated Knowledge
Correction
Oh I prefer a more recent movie as well when scoring
COT Sketch (v2)
(additional) Assign a "recency score" for each film based on the release date ... ...
User

Figure 4: Example of NL parser interactions in two modes.

Here, a critical challenge is that the query optimizer must balance cost against two goals: query accuracy and explainability. A compact logical and physical plan with fewer larger functions may executes more quickly, but larger functions are more difficult to generate accurately, so such a plan may reduce accuracy. A smaller number of larger functions may also make explanations harder because little information about intermediate results is available. A more detailed plan can improve accuracy and provide clearer explanations, but it typically slows down execution. Exploring these trade-offs is an important research question. Additionally, as we discuss next, a user may provide comments and ask questions about query plans, which also opens the possibility of re-generating functions at different granularity if a user asks questions and needs to see additional intermediate results.

5 INTERACTIONS

Interactive NL Parser. When the NL parser translates the user's potentially ambiguous NL query into a *query sketch*, clarification or correction may be necessary. A straightforward but inefficient approach would be to show the user the entire query sketch and ask them to revise their original natural language request. This approach, however, introduces unnecessary human effort as the entire query needs to be rewritten by the user. Instead, in KathDB, we experiment with two finer grained interaction models, and implement two collaborative agents: a *reviewer* and a *sketch generator*.

Inspired by recent work that advocates user involvement in the AI disambiguation process [2], KathDB's NL parser proactively asks clarification questions when it cannot confidently map a NL query to a single interpretation. As shown on the left of Figure 1, the reviewer agent first inspects the NL query and decides between two actions: (i) ask a clarification question if it detects unresolved ambiguity, or (ii) forward the request directly to the *sketch generator*. Here, ambiguity depends on whether a term's meaning is context dependent or user dependent. For example, in Figure 4 the word *"exciting"* could be user-specific. When such ambiguity is detected by the reviewer agent (prompted by *"Look for ambiguous terms or*

[4] https://github.com/tesseract-ocr/tesseract

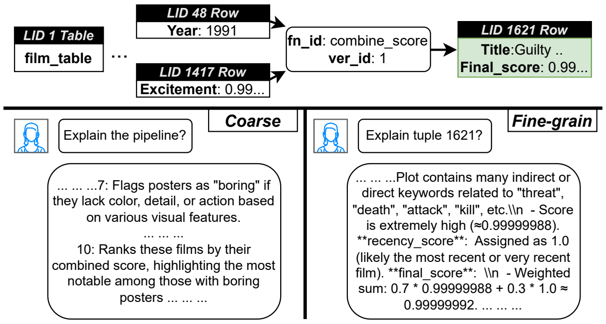
subjective words..."), it asks the user a focused question (e.g., *"What does 'exciting' mean in this context?"*). The user then provides additional context, and the agent reassesses the query based on this newly provided information. After generating the query sketch, the NL-parser performs reactive query correction based on user feedback. For example, the user may review the query sketch and realize that an important factor is missing (e.g., movie release year) (Figure 4, bottom, `Correction` box). The user can tell the query writer to refine the sketch accordingly. The query writer incorporates the feedback, produces a revised sketch, and submits it for another round of review. This refinement cycle repeats until the user explicitly responds `OK`.

An important research question is to comparatively evaluate these various feedback mechanisms and explore other possible ways of seeking user feedback at this stage of query execution to make the *query sketch* as accurate as possible while minimizing user effort, as a *query sketch* that does not match the user's intent will inevitably lead to semantically incorrect functions being generated and erroneous final query results.

Interactive Query Execution Debugger. During execution, a function that cleared optimizer checks may still fail on unseen inputs. The monitor performs a role similar to the semantic checks at function-generation time, but now with the full input data.

When the monitor detects a **syntactic fault** (e.g., unsupported file format), it launches a two-agent loop: the *reviewer* diagnoses the exception, the *rewriter* patches the code, increments its `ver_id`, and resumes execution from the failed operator. Tuples unaffected by the error continue through the old function definition in a parallel process, preserving throughput. For example, the `classify_boring` operator in Figure 1 may rely on `cv2`[5] to load and analyze image pixels to assess whether the poster's colors are vibrant, a feature that contributes to whether the image is 'boring' as specified in its function signature. If it encounters an unsupported *HEIC* file, the pipeline proceeds on other images while the rewriter adds a conversion step to a `cv2` compatible format.

Semantic anomalies are subtler: the code runs but produces an outcome that the user does not expect. For example, a similarity-based vector join may mistakenly match the same poster image to several different movie titles, even though the code executes without error. In this case, the monitor inspects the resulting table, detects that a single poster image is linked to multiple movies, and flags this as unlikely to match the user's intent. It then asks the user for confirmation or correction and, based on the feedback, updates the function logic so that the join behaves as intended. The monitor also explains a likely cause (e.g., the LLM may have implicitly assumed a one-to-one correspondence between poster images and tuples in `movie_table`, an assumption that does not hold in practice [20] and produces spurious matches). To resolve the issue, the monitor prompts the user to either accept the operator as is, request an adjustment (e.g., enforce that each poster can be linked to only one tuple in `movie_table`), or request a complete rewrite of the operator.



Explain the pipeline?

... ...7: Flags posters as "boring" if they lack color, detail, or action based on various visual features.
... ...
10: Ranks these films by their combined score, highlighting the most notable among those with boring posters ... ...

Explain tuple 1621?

... ...Plot contains many indirect or direct keywords related to "threat", "death", "attack", "kill", etc.\\n - Score is extremely high (≈0.99999988). **recency_score**: Assigned as 1.0 (likely the most recent or very recent film). **final_score**: \\n - Weighted sum: 0.7 * 0.99999988 + 0.3 * 1.0 ≈ 0.99999992. ... ...

Figure 5: Example result of query explanations in two modes.

A research question at this stage of query execution is that an LLM-based *monitor* examining intermediate results will incur additional token costs, so some type of sampling is necessary. Additionally, if a semantic error is found for the current tuple, then all the previous tuples must be reprocessed, and a question becomes whether any of the past work may be somehow leveraged.

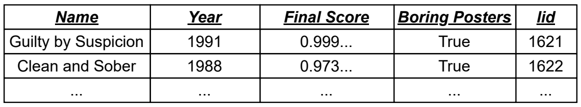
Interactive Query Result Explanation. After the query execution stage, if a user receives any unexpected results, looking through the input data offers little insight and is prohibitively time-consuming on a large database. KathDB overcomes this limitation by exposing the *full provenance* of query results and *makes it queryable in NL*: For every output tuple, KathDB can show the materialized view it came from (described in Section 3), how it was derived by the pipeline of FOAs, and how each function may have been updated during the agentic query-generation process. Additionally, the user can also ask NL queries over this lineage information as shown in Figure 5. KathDB supports two explanation modes. The coarse-grained mode shows a high-level overview of the transformations performed during query execution (e.g., showing that KathDB decides whether a poster is "boring" by analyzing its visual features and the objects depicted). The fine-grained mode offers a low-level explanation: it takes a specific `lid` as input, inspects the function signature and implementation, traces parent tuples, and shows the details of how every output tuple field was derived (e.g., showing that the final score combines a recency score of 1 with an excitement score from earlier functions, each traceable by the user). Overall, our intermediate relational view layer and fine-grained lineage enable KathDB to provide more detailed and consistent explanations than larger-scoped, black-box LLM invocations.

An important research question here is that query execution over very large databases produces large intermediate results and large lineage information. How should the system best summarize this information and provide answers to users' query explanation questions efficiently and at low cost? Can the system not save all the lineage in the first place but only sample lineage information?

6 INITIAL RESULT

We execute the example query shown in Figure 1 with KathDB over MMQA [20], a dataset that contains tables, texts, and images crawled from Wikipedia. The top two results are shown in Figure 6.

[5] https://opencv.org/



Name	Year	Final Score	Boring Posters	lid
Guilty by Suspicion	1991	0.999...	True	1621
Clean and Sober	1988	0.973...	True	1622
...	...	...	...	...

Figure 6: Example final output of KathDB.

The query parser accepts the query and asks the following clarification question: *"What does 'exciting' mean in this context?"* We simulate the following user reply: *"The movie plot contains scenes that are uncommon in real life"* (Figure 4); the parser then generates a query sketch with eight steps. We further simulate the user adding an additional requirement, *"I prefer more recent movies when scoring,"*. The parser updates the plan and produces an 11-step query sketch. For the current prototype, we have pre-written the view-population function that invokes GPT-4o and supplies schema information to KathDB as the first step, leaving 10 remaining logical plan nodes. The query plan optimizer generates the following functions: (1) selects the relevant columns from `movie_table` (e.g., title, release year); (2) joins the relational view over text with `movie_table`; (3) joins the relational view over images with `movie_table`; (4) computes excitement scores by measuring vector similarity between keywords (e.g., *gun*, *murder*, ...) and all extracted text entities (Note: it is worth noting that a LLM generates the keyword list here); (5) assigns recency scores based on the release year; (6) combines excitement and recency scores according to user request; (7) classifies each poster as boring or not using the raw image and its scene-graph view; (8) filters out posters labeled as boring; and (9) and (10) joins all intermediate results to produce the final ranked list of movies by their combined score. Finally, a tuple (`lid=1621`) is generated, as shown in Figure 6. The user then requests result explanations for the entire pipeline (Figure 5, left) and for how the final tuple is produced (right). Only a snippet is shown due to space constraints.

7 RELATED WORK

Recent work has focused on building data systems that support queries over multimodal data. These systems can be broadly grouped into two categories: One set of systems [9, 12, 15, 22, 24, 28] consider ML models (e.g., an object classifier) as operators and requires users to write SQL or Python code explicitly. Instead, KathDB accepts NL queries, performs iterative and interactive query refinement, and leverages LLMs both as a query planner and as a function generator. A second line of work [8, 14, 21] also takes NL queries and uses LLMs *as black-boxes* to plan and execute them. However, KathDB differs from these systems in two key ways: (i) it supports richer communication with the user, enabling iterative query parsing and execution, and (ii) it unifies diverse modalities behind a single, relational semantic layer, improving query semantics and explainability. A third line of work [3, 7] proposes approaches that generate functions for populating relational views or performing data curation tasks over single-modality unstructured text. KathDB differs from them in that it supports multimodal data, generates functions not only for view population but also for query execution, and includes a lineage tracking system for explainability.

8 CONCLUSION

We introduced KathDB, a multimodal DBMS with explainable query execution. KathDB combines multimodal data under a unified relational view, implements a new FAO model, and keeps users in the loop through interactive channels for clarification, correction, execution guidance, and result explanation.

ACKNOWLEDGMENTS

This work was supported in part by the NSF through awards 2211133. We thank the reviewers for the helpful comments.

REFERENCES

[1] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.

[2] Amershi et al. 2019. Guidelines for Human-AI Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3290605.3300233

[3] Arora et al. 2023. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *Proceedings of the VLDB Endowment* 17, 2 (2023), 92–105.

[4] Brown et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Bordes et al. 2024. An introduction to vision-language modeling. *arXiv preprint arXiv:2405.17247* (2024).

[6] Christophides et al. 2020. An overview of end-to-end entity resolution for big data. *ACM Computing Surveys (CSUR)* 53, 6 (2020), 1–42.

[7] Chen et al. 2023. SEED: Domain-specific data curation with large language models. *arXiv preprint arXiv:2310.00749* (2023).

[8] Chen et al. 2023. Symphony: Towards Natural Language Query Answering over Multi-modal Data Lakes.. In *CIDR*. 1–7.

[9] Jo et al. 2024. Thalamusdb: Approximate query processing on multi-modal data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[10] Krishna et al. 2017. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International journal of computer vision* 123 (2017), 32–73.

[11] Liu et al. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[12] Liu et al. 2025. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2.

[13] Lindsey Linxi Wei et al. 2025. Multi-Objective Agentic Rewrites for Unstructured Data Processing. https://api.semanticscholar.org/CorpusID:283458157

[14] Nooralahzadeh et al. 2024. Explainable Multi-Modal Data Exploration in Natural Language via LLM Agent. *arXiv preprint arXiv:2412.18428* (2024).

[15] Patel et al. 2024. Semantic Operators: A Declarative Model for Rich, AI-based Data Processing. *arXiv preprint arXiv:2407.11418* (2024).

[16] Peng et al. 2024. Stepwise reasoning error disruption attack of llms. *arXiv preprint arXiv:2412.11934* (2024).

[17] Pan et al. 2025. Why Do Multiagent Systems Fail?. In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*. https://openreview.net/forum?id=wM521FqPvI

[18] Russo et al. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *arXiv preprint arXiv:2505.14661* (2025).

[19] Shankar et al. 2024. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *arXiv preprint arXiv:2410.12189* (2024).

[20] Talmor et al. 2021. MultiModal{QA}: complex question answering over text, tables and images. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ee6W5UgQLa

[21] Urban et al. 2024. Demonstrating CAESURA: Language Models as Multi-Modal Query Planners. In *Companion of the 2024 International Conference on Management of Data*. 472–475.

[22] Urban et al. 2024. Eleet: Efficient learned query execution over text and tables. *Proceedings of the VLDB Endowment* 17, 13 (2024), 4867–4880.

[23] Wei et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[24] Xu et al. 2022. EVA: A symbolic approach to accelerating exploratory video analytics with materialized views. In *Proceedings of the 2022 International Conference on Management of Data*. 602–616.

[25] Yuan et al. 2024. nsdb: Architecting the next generation database by integrating neural and symbolic systems. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3283–3289.

[26] Yang et al. 2025. Hallucinate at the Last in Long Response Generation: A Case Study on Long Document Summarization. *arXiv preprint arXiv:2505.15291* (2025).

[27] Zhang et al. 2023. Equi-vocal: Synthesizing queries for compositional video events from limited user interactions. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2714–2727.

[28] Google Cloud. 2025. BigQuery: Fully Managed Data Warehouse. https://cloud.google.com/bigquery. Accessed 23 Jul 2025.