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 control that forces users to register machine learning UDFs manually within SQL or offload execution entirely to black-box LLMs, sacrificing usability and 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.

ACM Reference Format:
Guorui Xiao, Enhao Zhang, Nicole Sullivan, Will Hansen, and Magdalena Balazinska. 2025. KathDB: Explainable Multimodal Database Management System with Human-AI Collaboration: Vision Paper. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [11, 17, 18]. Consider the following natural language (NL) query in Figure 1: *"Sort the given films in the table by how exciting they are, but the poster should be 'boring'."* The database contains a relational table with movie metadata (title, release year), a column with the text plot, and a column with the pixel values of movies' posters. 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 in terms of the exact user goal, complicating query evaluation.

Recent work in machine learning (ML), most notably Large Language Models (LLMs) [3] and Vision Language Models (VLMs) [4], has motivated a new class of *multimodal* database management systems [6, 10–12, 15, 17, 18, 20, 21, 24]. Some of those systems [20, 24] 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 [6, 10–12, 15, 17, 18, 21] 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 [6, 7, 10, 17, 18, 20, 21] 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 [6, 11, 17], 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

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

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

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

verification of user intent. Unlike other systems, KathDB defines each step as a function, complete with a signature and implementation version history. This novel FAO design has several advantages: First, separating signature declaration from body synthesis lets KathDB explore different interpretations for the same sub-task (e.g., interpreting "exciting movies" as action vs. original titles). Second, splitting a query into small functions and generating each function separately reduces common problems in long generation such as hallucination [22] and error propagation [13]. Third, each function is assigned an identifier and a version tag, so the result tuples can be represented as a chain of states, allowing recursive lineage. Finally, the FAO design enables the cost-based optimizer to explore different physical plans.

(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, audio, 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 [19], 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 example, given the query in Figure 1, and a query sketch step: *"Analyze poster visual features using either extracted objects or 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.

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 query at once, it prevents autoregressive error propagation [13]. Additionally, it can generate these functions efficiently, in parallel. We describe the details of function generation in Section 4. We omit cost-based optimization due to space constraints as that component uses known techniques.

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

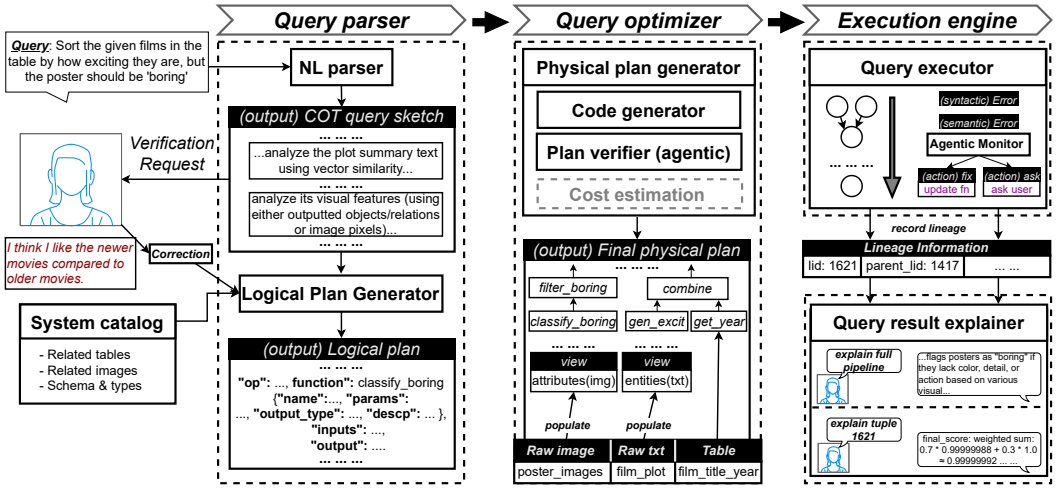

Figure 1: Overview of KathDB. KathDB consists of three main components: a query parser, an optimizer, and an execution engine. KathDB accepts an NL query from the user as input and generates a logical plan then a physical plan, interacting with the user to seek clarifications as needed. KathDB executes 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.

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)

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 leads to high complexity, makes view population expensive, and introduces attributes that rarely matter during actual query execution. Conversely, an over-simplified schema fails to capture essential semantics and thus cannot reliably answer queries correctly. Requiring a user-defined schema adds unwanted burden, as our goal is for users to reason at the level of NL. Our design must therefore achieve a balance: expressive enough to model modality-specific nuances, yet compact, tractable, and extensible to future modalities.

Images and Videos as Scene Graphs. Inspired by EQUI-VOCAL [23], KathDB adopts a scene graph [8] data model that represents visual content as objects interacting in space and time. Table 1 summarizes the relational schema that supports this model. Video frames are uniquely identified by the (`vid`, `fid`) pair, while the **Objects** table records each object's label ID (`cid`) and bounding boxes (`x1`, `y1`, `x2`, `y2`). Objects can have relationships with each other, and possess attributes, captured as key-value pairs. Finally, the **Frames** table provides a view over the pixels within frames or images. The 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"*. KathDB uses the scene graph to provide a grounded explanation to why the latter does not make a movie "exciting". Note: individual images are treated as single-frame videos.

Text content as text semantic graph. Unlike standalone images, where each object is unique, a textual corpus presents additional challenges, such as entity resolution [5, 15]. The relational schema in Table 2 summarizes KathDB 's text data model to address these challenges. An entity from the **Entities** table includes a document id (`did`), an entity id (`eid`) shared by all mentions within the document, and a class type (`cid`). KathDB ensures an `eid` is unique and consistent within a given text corpus, but makes no guarantees across documents, which we leave for future work. A mention from the **Mentions** table consists of a unique mention identifier (`mid`) in a sentence (`sid`) of a given document (`did`), and the starting and ending char positions (`span1`,`span2`) of the mention. For example, in the same document, "Taylor" and "Mrs. Swift" will have two different `mid`'s, but refer to the same entity, "Taylor Swift". Failure to consider these subtle distinctions leads to incorrect query results.

lid	parent_lid	src_uri	func_id	ver_id	data_type	ts
21	1	file://data/...	load_data	1	row	1.7...
...	...	...	...	...	...	...
1274	941	None	join_text_scene...	1	row	1.7...
1274	940	None	join_text_scene...	1	row	1.7...
1417	1274	None	gen_excitement...	1	row	1.7...

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

```
[{ "op": "Extended Projection",
     "function": { "name": "classify_boring", "params": [ {"name":
"films_with_image_scene", "type": "DataFrame"}, ...,
     "description": "Analyze visual features of each film's poster..." },
     "inputs": [ {"k": "films_with_image_scene", "v": "films_posters_imgid"}, ... ],
     "output": "films_with_boring_flag" },  ... ... ... ]
```

Figure 3: Function signature generated by the logical plan generator. Each input argument under the key 'name' inside the `function.params` has to match the k field of the `inputs`.

Similar to the image scene graph representation, KathDB defines relationships and attributes, in key-value format. Finally, the **Texts** table provides a view over the raw texts.

Provenance model. 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 path (`src_uri`; e.g., an image object from an s3 bucket), 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 registers itself with one lineage row with `data_type=table`. During function transformation, each output row is linked with the corresponding input row with `data_type=row`. An operator that yields a scalar (e.g., a table column name) records `data_type=scalar` and links its `parent_lid` to the input table. With such lineage tracking, recursive joins on `parent_lid` trace any output or intermediate result, in any granularity such as dataframe, row, or even scalar, back to exact sources and function that generates it.

Figure 2 shows some sample rows from the lineage table for the query in Figure 1. One particular tuple (`lid=1417`) contains an excitement score for a movie generated by `gen_excitement_score`, which takes the views over the movie plot as input and thus this tuple is linked to multiple parents (`lid=940,941,...`). In the beginning, a tuple (`lid=21`) containing the film release year and poster path information is loaded and used for this chain of derivations.

4 Function-as-operators (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 generates 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 generate the code while not being excessive to confuse the model [9]. Thus logical plan generator generates 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, `op` labels the conceptual operator for query plan visualization; `inputs` is a dictionary whose keys match those parameter names and whose values point to concrete data sources; `output` declares the output produced by the node; `function.name` becomes the identifier of the function; `function.params` enumerates the formal arguments. Here, a data source may refer to (i) a base relation already materialized in KathDB, whose schema is given by the catalog, or (ii) an intermediate table produced by a preceding node. Finally, `description` field offers semantic hints for downstream code synthesis. Inspired by the principles in [14], 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 user's 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* that it needs additional information from, 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 to judge again. Once the verifier is satisfied that the plan faithfully realizes 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*. To begin with, 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 the function bodies. 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.

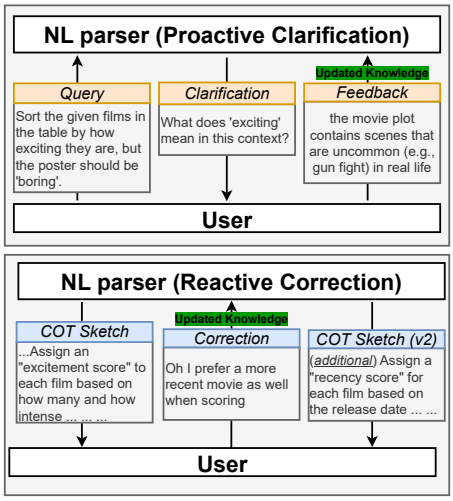

Figure 4: Example of NL parser interactions in two modes.

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 inputs, produced outputs, 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.

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. Rather than showing the full query sketch and asking users to edit their original NL query, 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 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

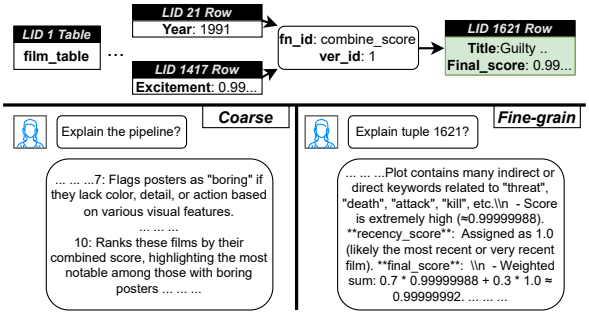

Figure 5: Example result of query explanations in two modes.

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`.

Interactive Query Execution Debugger. During execution, a function that cleared optimizer checks may still fail on unseen inputs. When the *monitor* detects a syntactic fault (e.g., shape mismatch), 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 relies on `cv2` [4] for image and 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. The monitor does a similar job to the semantic checking at the function generation stage but with full input this time. For example, `classify_boring` can label *all* movies as `False` and thus the `filter_boring` will filter out all rows and output an empty table. The monitor flags the anomaly, explains a likely cause (e.g., an overly strict threshold), and prompts the user to accept, adjust, or rewrite.

Interactive Query Result Explanation. After the query execution stage, if a user receives undesired results, simply re-running the query or looking through 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 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

[4] 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.

offers a low-level explanation: it takes a specific `lid` as input, inspects the function signature and implementation, traces parent tuples, and details how every output tuple field is 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.

6 Initial Result

We execute the example query shown in Figure 1 with KathDB over MMQA [16], a dataset that contains tables, texts, and images crawled from Wikipedia. The top two results are shown in Figure 6. 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 8 steps. After the user adds the additional requirement, *"I prefer a more recent movie as well 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) select all related columns from the tabular input data (i.e., release year, plot, and image path); (2) join all views of the text data; (3) join views of images; (4) compute excitement scores by vector similarity between keywords (e.g., *gun*, *murder*, ...) and all entities and relations in plot views; (5) assign recency scores based on how recent a movie is; (6) combine excitement and recency scores as requested by the user; (7) use raw poster image and corresponding scene graph view to classify each row as boring or not; (8) filter out boring posters when the label is `True`; (9) and (10) join the intermediate results and output the ranked list of movies by their combined score (generated by (6)). 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 [7, 10, 12, 18, 20, 24] 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 [6, 11, 17] 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 within a single semantic layer, improving query semantics and 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.

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

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

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

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

[7] 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.

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

[9] 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

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

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

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

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

[14] 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

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

[16] 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

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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).

[23] 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.

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