ThalamusDB: Approximate Query Processing on Multi-Modal Data

SAEHAN JO, Cornell University, USA
IMMANUEL TRUMMER, Cornell University, USA

We introduce ThalamusDB, a novel approximate query processing system that processes complex SQL queries on multi-modal data. ThalamusDB supports SQL queries integrating natural language predicates on visual, audio, and text data. To answer such queries, ThalamusDB exploits a collection of zero-shot models in combination with relational processing. ThalamusDB utilizes deterministic approximate query processing, harnessing the relative efficiency of relational processing to mitigate the computational demands of machine learning inference. For evaluating a natural language predicate, ThalamusDB requests a small number of labels from users. User can specify their preferences on the performance objective regarding the three relevant metrics: approximation error, computation time, and labeling overheads. The ThalamusDB query optimizer chooses optimized plans according to user preferences, prioritizing data processing and requested labels to maximize impact. Experiments with several real-world data sets, taken from Craigslist, YouTube, and Netflix, show that ThalamusDB achieves an average speedup of 35.0× over MindsDB, an exact processing baseline, and outperforms ABAE, a sampling-based method, in 78.9% of cases.

CCS Concepts: • **Information systems** → **Online analytical processing engines**; **Data model extensions**; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: query processing, multi-modal data, neural models

ACM Reference Format:
Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 186 (June 2024), 26 pages. https://doi.org/10.1145/3654989

1 INTRODUCTION

Prior work has introduced systems that integrate machine learning with databases, such as MindsDB [30] and EvaDB [17]. Using large neural models, these systems enable users to handle multi-modal data within a unified framework. In this context, the primary challenge in processing queries on multi-modal data is the computational cost associated with model inference, as shown in Figure 1. Processing large amounts of multi-modal data is very expensive and becomes the dominant factor in query processing. This processing overhead underscores the need for a more efficient solution. Hence, we introduce ThalamusDB, an approximate query processing (AQP) system designed for complex queries on multi-modal data. The code of ThalamusDB is available at https://github.com/saehanjo/thalamusdb.

ThalamusDB supports retrieval and aggregation queries (with joins) that integrate natural language predicates on multi-modal data. It supports an extended, relational data model, introducing specialized column types to integrate picture and audio data. Queries are formulated using an SQL dialect that supports predicates, formulated in natural language on text, picture, or audio data

Authors' Contact Information: Saehan Jo, Cornell University, Ithaca, New York, USA, sj683@cornell.edu; Immanuel Trummer, Cornell University, Ithaca, New York, USA, itrummer@cornell.edu.

ACM 2836-6573/2024/6-ART186
https://doi.org/10.1145/3654989

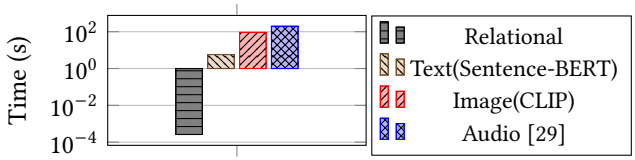

Fig. 1. Runtime comparison of predicate evaluation on 10,000 items between an equality predicate on relational data and natural language predicates on unstructured data.

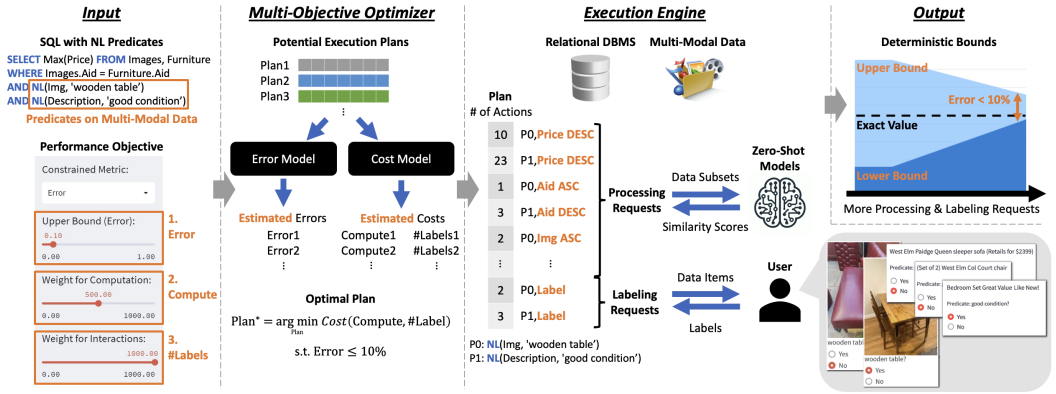

Fig. 2. Overview of ThalamusDB.

columns. For predicate evaluation, ThalamusDB utilizes zero-shot classifiers, like OpenAI CLIP [36] and Sentence-BERT [37], which enables classification of various data types merely from a natural language description of relevant classes.

ThalamusDB employs deterministic AQP techniques for processing multi-modal data. ThalamusDB approximates by applying models to a subset of data and asking users to label a small number of items. Instead of randomly sampling data, ThalamusDB carefully selects which subsets of data to process and label. ThalamusDB relies on a scenario-specific optimizer to determine the optimal data subsets for maximal result quality. Users submit SQL queries with natural language predicates on multi-modal data. They specify their preferred tradeoffs among three relevant metrics: result approximation error, labeling overheads, and computational overheads. Once query processing begins, users are asked to label a small number of multi-modal data items, determining whether they satisfy the predicate or not. Using the provided labels and processed data subsets, ThalamusDB generates bounds for query aggregates, guaranteed to contain the precise value.

Sampling-based approximation is the most classical approach to trade result precision for processing overheads [12]. However, sampling has several limitations in our problem setting. Generating offline data samples through methods like stratified sampling is challenging due to the unrestricted nature of natural language predicates. Furthermore, sampling is not a good match for reducing the number of required labels when identifying the similarity threshold. Instead, a more efficient approach is to systematically select items with similarity scores that help refine the threshold range the most. At the same time, sampling poses drawbacks for aggregates like maxima and minima [34]. In contrast, ThalamusDB relies on deterministic AQP for approximation. The term "deterministic" indicates that the result bounds always guarantee to contain the true value, as opposed to confidence bounds. Prior work on deterministic AQP [15, 34] demonstrates that processing a data subset can yield deterministic bounds on query results. ThalamusDB capitalizes on the observation that processing different data subsets may result in varying degrees of improvement in result

precision. ThalamusDB is unique in its focus on multi-modal data. Hence, ThalamusDB leverages the relatively cheap relational processing (compared to machine learning inference) to carefully select row subsets expected to minimize approximation error.

ThalamusDB relies on a multi-objective optimizer to explore alternative processing plans in the plan space (i.e., choosing which data subsets to process and to label for natural language predicates). Hence, the optimizer is equipped with scenario-specific error and cost models, estimating sizes of intermediate results as well as processing overheads, the required number of labels, as well as the approximation error after executing a series of processing steps. Classifiers associated with different natural language predicates may vary significantly in per-invocation costs (e.g., due to the type of data and the model size as shown in Figure 1). At the same time, predicate selectivity differs, making certain evaluation orders preferable over others. Finally, prioritizing predicate evaluations and item labeling (both can help to increase approximation precision) requires considering data and query properties. The scenario-specific query optimizer is able to make appropriate choices for all of the aforementioned tuning decisions. Given an optimal plan, ThalamusDB applies classifiers on a carefully selected data subset, processes the remaining part of the input query (not involving natural language predicates), requests manual labels from the user, and finally presents the final query result to the user.

It is crucial to distinguish different types of approximation errors that arise in the context of ThalamusDB, as well as associated guarantees. First, ThalamusDB uses neural models to process multimodal data. Such models are inherently imperfect and may make mistakes. However, improving the accuracy of neutral models is an orthogonal problem to the one addressed in this work and outside of our scope. Instead, ThalamusDB focuses on minimizing errors caused by processing a subset of the data, instead of the full data set. Assuming error-free neural models (with the understanding that this assumption is, of course, simplifying), ThalamusDB produces deterministic bounds that are guaranteed to contain values obtained by processing the full data set. Lastly, we need to distinguish the *approximation error in execution* from the *approximation error in optimization*. ThalamusDB uses models to estimate error (and cost) before processing specific data subsets for optimization. As usual in query optimization, these models *cannot guarantee* deterministic bounds or the selection of optimal processing plans. However, as the optimizer error model is *only used to select data subsets*, not to produce query results, the lack of guarantees of the optimizer model does not affect guarantees on the query result.

In our experiments, we evaluate ThalamusDB on various types of queries on three real-world data sets from Craigslist, YouTube, and Netflix. These benchmarks consists of queries on different data types, including pictures, audios, and texts, as well as queries on combinations of different data types. The performance of ThalamusDB depends on the data distribution, and pathological cases are, in principle, possible (see Example 3 in Section 4.3). However, our experiments demonstrate that it performs well on real-world datasets. We compare ThalamusDB against multiple baselines, MindsDB and ABAE, showing significant improvements. In summary, the original scientific contributions in this paper are the following:

- We present ThalamusDB, a deterministic approximate query processing system for answering complex queries with natural language predicates on multi-modal data.
- We outline processing methods that reduce the processing overheads of running queries involving machine learning inference on large data sets.
- We introduce a multi-objective optimizer that enables users to trade between different cost and error metrics.
- We evaluate ThalamusDB experimentally on several real-world data sets and compare to various baselines, including MindsDB and ABAE.

```
CREATE TABLE Ads (AdID int PRIMARY KEY,
  Created date, AdText text, Price int);
CREATE TABLE Pics (FilePath PICTURE PRIMARY KEY,
  AdID int, FOREIGN KEY (AdID) REFERENCES Ads(AdID));
```

Fig. 3. Schema of running example database.

```
SELECT Min(A.Price)
FROM Ads A, Pics P
WHERE Created > DATE'2022-01-01'
  AND P.AdID = A.AdID
  AND NL(P.FilePath,
    'Shows wooden chair with blue cushions')
```

Fig. 4. Query on example database, calculating minimal price of ads offering chairs after January 1st 2022.

2 SYSTEM OVERVIEW

Figure 2 shows an overview of the ThalamusDB system. ThalamusDB processes SQL queries with natural language predicates on data sets, integrating structured data, pictures, audio data, and text. Database schemata are defined using a slightly extended version of SQL DDL commands, integrating dedicated data types to reference pictures and audio files.

ThalamusDB relies on two types of external components for data processing: a relational database management system (DBMS) and a repository with neural models for multi-modal data analysis. The relational DBMS is used for managing the database schema and for structured data processing. Internally, ThalamusDB represents picture and audio data columns as text columns in the DBMS, containing paths to files in a directory storing its non-relational data. Neural models are used to evaluate natural language predicates that appear in the input queries. Currently, ThalamusDB uses CLIP [36] for processing images, the model by Mei et al. [29] and CLAP [9] for processing audio data, and Sentence-BERT [37] as well as BART [26] for processing natural language predicates on text. Models are easily exchangeable for each data type with little effort. Internally, ThalamusDB decomposes incoming DDL instructions and queries into parts referencing multi-modal data (which are processed separately) and standard SQL (which are processed by the relational DBMS). The following example illustrates how users can use the system to query on multi-modal data.

Example 1. *Alice is furnishing her apartment and looking for a wooden chair with blue cushions on Craigslist. To get a sense of the price range, she aims to find the lowest price for offers over recent months. However, many listings lack color or material details in their text descriptions, requiring her to analyze associated pictures. Lacking the time to go through a large number of pictures, Alice decides to try out ThalamusDB, a novel system for multi-modal data analysis.*

Alice scrapes data from Craigslist and inserts it into ThalamusDB in relational format, using the schema in Figure 3. This schema uses the non-standard "PICTURE" data type, indicating that models for image analysis can be used on associated files. Now, Alice can formulate her request using the query from Figure 4. This query uses the "NL" keyword to enable natural language predicates on unstructured (i.e., picture, audio, or text) columns. ThalamusDB evaluates such predicates using large neural networks to calculate similarity scores between text and data. Predicates are considered satisfied once the similarity reaches a certain threshold. The best threshold varies across predicates and data sets, hence ThalamusDB asks users to label a limited number of samples for calibration. For instance, Figure 2 shows labeling requests for pictures as well as text. Furthermore, users can configure the system for different tradeoffs between the three metrics: approximation error, labeling overheads, and computational overheads. The user interface for performance objective (shown in Figure 2) allows

specifying constraints on one metric and weights on the remaining two[1]*. To save time, Alice limits the number of labeling requests to five. ThalamusDB chooses an optimal processing plan, analyzes a subset of pictures, requests labels for up to five pictures, and finally produces an approximate query result.*

For each input query, ThalamusDB executes the following stages.

Initialization. ThalamusDB prepares the relational database for query processing by creating temporary tables, one for each natural language predicate that appears in the query. Each table contains two columns, representing the item ID as well as a similarity score, capturing the similarity between predicate text and item (i.e., a picture, audio file, or text). After initialization, similarity scores are unknown, represented by SQL NULL values.

Pre-Processing. Next, ThalamusDB performs a pre-processing stage in which, for each predicate, it calculates similarity values between predicate text and items using multimodal models, for small samples of items. At the same time, for each predicate, it requests labels from users for a small number of items (three). This allows ThalamusDB to narrow down the threshold on similarity starting from which the predicate is considered satisfied. The purpose of pre-processing is to collect query-specific statistics, including natural language predicate selectivity (which cannot be estimated, e.g., based on histograms), that are helpful for query optimization.

Query Optimization. Then, ThalamusDB invokes a scenario-specific query optimizer to generate an optimized execution plan. Users may specify constraints (upper bounds) or weights on any of the three cost metrics, namely computation cost, number of requested labels, and approximation error. These performance objectives are taken into account by the optimizer. The resulting execution plan minimizes weighted cost among plans respecting the constraints.

Query Execution. The execution plan is forwarded to the execution engine which executes a sequence of actions. ThalamusDB supports two types of execution actions: prioritized data processing and labeling requests. First, prioritized data processing involves applying multi-modal models to calculate similarity scores for a specific predicate and a subset of items. Those items are not selected randomly but can be prioritized according to values in other columns of the same table (e.g., referring to Example 1, we could select pictures of Craigslist items in ascending order of price). The resulting similarity scores are stored in the temporary tables, created during the initialization phase. Second, labeling requests ask users to determine whether specific natural language predicates apply to specific items (e.g., asking Alice to verify whether a specific picture shows a wooden chair). ThalamusDB does not perform (costly) training on large models. Instead, it uses labels to narrow down the threshold on similarity scores, starting from which items with a higher score are considered satisfied. After calculating similarity values for a subset of predicates and items, using multi-modal models, as well as narrowing down thresholds on similarity values via user labels, the engine executes SQL queries to calculate lower and upper bounds on query result aggregates respectively.

Post-Processing and Early Termination. The optimizer selects execution plans based on preprocessing-based statistics, using (as it is typical in the domain of query optimization) simplifying assumptions to estimate execution overheads and the approximation error, i.e., the distance between lower and upper bounds on query aggregates. Hence, actual execution overheads and the resulting distance between bounds may deviate from estimates. ThalamusDB uses additional mechanisms to ensure that user-specified constraints are respected. More precisely, the execution engine interrupts execution, whenever aggregate computation overheads exceed the user constraint. If the approximation error does not satisfy constraints specified by the user, after plan execution,

[1]Allowing users to specify constraints on more than one metric may lead to situations where the system is unable to satisfy all constraints. This cannot happen if constraints are placed on at most one metric. E.g., any approximation error constraint can be met when processing enough data and requesting enough labels.

ThalamusDB performs post-processing to reduce the error further. In post-processing, ThalamusDB greedily selects actions that maximize the expected error reduction. Each action either requests labels from the user for a specific predicate and item or calculates similarity scores for a specific predicate and a subset of items. Post-processing stops once the distance between bounds becomes small enough such that the user-specified error constraints are met.

3 FORMAL MODEL

ThalamusDB supports the following types of queries and schemata.

Definition 1 (Multi-Modal Schema)**.** ThalamusDB supports star schemata using standard SQL DDL (data definition language) commands with extensions for multi-modal data: ThalamusDB supports two additional data types, `PICTURE` and `AUDIO`. Corresponding columns contain paths to images or audio files.

Definition 2 (Multi-Modal Query)**.** ThalamusDB supports SPJA (select-project-join-aggregate) queries on multi-modal schemata. Compared to standard SQL, ThalamusDB supports one additional keyword: `NL`. `NL` can be used in the `WHERE` clause and introduces a natural language predicate on a image, audio, or text column. The function takes two parameters, the column on which to apply the predicate and a string describing the predicate condition.

The current implementation is restricted to SQL queries referencing non-negative columns in aggregates and assumes joins in a star schema database. ThalamusDB processes natural language predicates in queries with the help of models, defined next.

Definition 3 (Zero-Shot Model)**.** Given a predicate text and a data item (picture, audio file, or text), zero-shot models calculate text-data similarity scores. If similarity is above a similarity threshold (discussed next), the associated predicate is considered satisfied.

Models may produce inconsistent similarity scores, leading to an imperfect separation between data items satisfying the predicate and others. However, improving the accuracy of classifiers is an orthogonal problem and outside the scope of this paper. ThalamusDB aims at exploiting available models as efficiently as possible. For each predicate, a suitable similarity threshold is determined via labeling requests (similar to the use of thresholds in other recent work on classification [13]).

Definition 4 (Labeling Request)**.** A labeling request is described by a pair $\langle d, t, s \rangle$ where d is a picture, audio file, or text, t text describing a predicate, and s a similarity score between text and predicate. The pair $\langle d, t \rangle$ is presented to the user (using a suitable representation, e.g. a playback console for audio files), asking for input whether d satisfies t.

ThalamusDB enables users to trade labeling or processing overheads for query result precision. It produces approximate results defined as follows.

Definition 5 (Approximate Result)**.** Having calculated similarity scores for a subset of rows and narrowed down similarity thresholds by labeling requests, ThalamusDB calculates (deterministic) lower and upper bounds for query aggregates. Given upper and lower bounds u and l, using the definition by Krenovich [24], the approximation error is $(u - l)/(u + l) \in [0, 1]$. If queries contain multiple aggregates, ThalamusDB uses the arithmetic mean error over all aggregates. For queries without aggregates, ThalamusDB uses $(u - l)/(u + l)$ where u and l are upper and lower bounds on the number of rows in the accurate query result.

ThalamusDB enables users to express preferences between all relevant cost and quality metrics.

Input q: Query with NL predicates; e: Execution plan; R: Relational DBMS; M: Multi-modal processor
Output Approximate result (i.e., deterministic bounds)

1: **function** EXECUTE(q, e, R, M)
    // Compute similarity scores based on prioritized data processing.
2: **for all** $\langle z, s, k\rangle \in e.C$ **do**
3: $ids \leftarrow R$.RUN("SELECT {$z.col$} {$q.from$} JOIN S{$z.id$} USING {$z.col$} WHERE Score IS NULL ORDER BY {s} LIMIT {k}")
        // Update scores to the similarity table.
4: **for all** $id \in ids$ **do**
5: $v \leftarrow M$.COMPUTESIM(id, z)
6: R.RUN("UPDATE S{$z.id$} SET Score={v} WHERE id={id}")
    // Update score thresholds based on labeling requests.
7: **for all** $\langle z, k\rangle \in e.L$ **do**
8: $z.\theta_l, z.\theta_u \leftarrow$ REQUESTLABELS(z, k)
    // Rewrite query to use similarity tables.
9: $Q \leftarrow$ REWRITE(q)
    // Execute rewritten queries to get deterministic bounds.
10: **return** {R.RUN(q) | $q \in Q$}

Algorithm 1. Execution process of ThalamusDB.

Definition 6 (User Preferences)**.** ThalamusDB considers approximation error, computational overheads, and the number of labeling requests as cost metrics. Users can constrain any of those metrics (by placing an upper bound) and choose weights on the two unconstrained metrics. ThalamusDB respects constraints during query processing while it aims to minimize weighted cost.

ThalamusDB solves the following query problem.

Definition 7 (Multi-Modal, Interactive Querying)**.** ThalamusDB processes requests of the form $\langle q, o\rangle$ where q is a query containing natural language predicates on a multi-modal schema and o are user preference specifications on performance objectives. ThalamusDB calculates similarity scores for a data subset and returns a set L of labeling requests. Users submit answers $A = \{\langle r, b\rangle | r \in L, b \in \{Yes, No\}\}$, assigning each labeling request to a Boolean answer. Finally, the system returns an approximate query result.

4 EXECUTION ENGINE

Algorithm 1 executes a given query plan in ThalamusDB. The algorithm can be decomposed into three parts. First, in Lines 1 to 6, the algorithm applies multi-modal models to calculate scores quantifying similarity between predicate text and data items. The algorithm focuses on a subset of predicates and data items, specified as part of the input plan ("prioritized data processing"). Second, in Lines 7 to 8, Algorithm 1 asks users to label a carefully selected subset of predicate-item combinations. This helps to establish thresholds on similarity scores for specific predicates, starting from which predicates are considered as satisfied. Third, in Lines 9 to 10, Algorithm 1 uses all information gathered in previous steps to calculate lower and upper bounds on query aggregate values. To do so, the algorithm issues rewritten queries, derived from the original query.

The following three subsections each focus on one of the three parts of Algorithm 1. Table 1 describes query plan properties, used in Algorithm 1 and the following algorithms. Similarly, Algorithm 1 and the following algorithms refer to properties of query predicates. Table 2 summarizes the associated fields.

Table 1. Query q with NL predicates and its attributes.

Attribute	Semantics
$q.Z$	Natural language predicates in q
$q.aggs$	Aggregates in q
$q.from$	FROM clause in q
$q.where$	WHERE clause in q
$q.limit$	Number of rows in LIMIT clause in q

Table 2. NL predicate z and its attributes.

Attribute	Semantics
$z.id$	Unique id among NL predicates
$z.text$	Text describing predicate
$z.type$	Type of the unstructured data, e.g., picture, audio, text
$z.col$	Name of the column on which z is applied
$z.table$	Name of the table that contains the column $z.col$
$z.\theta_l$	Current lower bound on the score threshold
$z.\theta_u$	Current upper bound on the score threshold

4.1 Prioritized Data Processing

First, the engine executes processing steps that involve evaluating classifiers on multi-modal data (Lines 2 to 6 in Algorithm 1). Computation steps are described as triples $\langle z, s, k\rangle$ where z is a predicate, k the number of rows and items to evaluate, and s the row priority order. For each such triple, ThalamusDB first collects the IDs of the first k items in order s for which no similarity score for predicate z is known yet (Line 3). ThalamusDB collects IDs by a query on the underlying relational database used for processing (we use curly braces to include variables into our SQL query strings, similar to Python syntax). Next, ThalamusDB iterates over selected items, applies a suitable model (depending on data modality) to calculate a similarity score (Line 5), and stores the resulting score in the temporary table associated with the predicate (Line 6). As illustrated in the following example, the choice of a processing order can have significant impact on the resulting approximation error (we discuss how to select processing orders in Section 5).

Example 2. *The running example query, illustrated in Figure 4, requests the minimal price of offers satisfying the specified predicates. When ThalamusDB evaluates the natural language predicate to compute this aggregate (assuming that the inequality predicate on dates has already been processed), different approaches to processing rows yield varying results. If we process rows in a random order, the minimal price remains unknown even when we process more items (unless we process all items). This is because potentially rows with lower prices may yet be unseen. In contrast, if we process rows in an ascending order based on price values, we can be assured that the first row satisfying the predicate does indeed have the minimum price among all qualifying rows, eliminating the need to process subsequent rows. Similar reasoning can be applied to maxima as well as summations and counts with joins, especially with skewed distributions.*

The parameter k specifies the number of items to be processed in a single computation step. Employing a smaller k enables more fine-grained decision-making but also leads to increased optimization overheads. Therefore, it is important to select a k value that balances the overheads

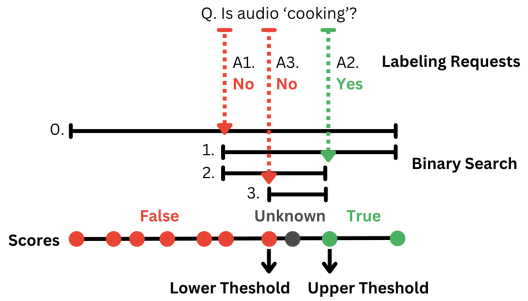

Fig. 5. Lower and upper score thresholds resulting from labeling requests.

with plan quality. ThalamusDB configures k to be smaller for higher-cost models (in our experiments, we use 0.1% of data items for audio and 1% for images and texts).

4.2 Labeling Requests

Second, in Lines 7 to 8 of Algorithm 1, ThalamusDB requests labels from users. Labeling requests are described as pairs of the form $\langle z, k \rangle$. Here, z denotes a zero-shot predicate and k the number of labels to retrieve. Labeling consists of presenting to users a corresponding data item (by showing a picture or text or playing an audio file), while asking to decide whether predicate z holds for the item or not. As shown in Figure 5, RequestLabels in Line 8 performs an algorithm that is similar to binary search. The goal of labeling is to narrow down the bounds on the similarity threshold as much as possible, using the given number of labels. Hence, RequestLabels starts with items whose similarity scores are as close to the similarity median as possible. Depending on the label (positive or negative), it either continues with an item whose similarity is close to the median in the upper half of similarity values (if positive) or in the lower half. To adopt a more robust approach, we could present several items with consecutive similarity scores, instead of just one. This method decreases the probability that an item inadequately represents its current score position and helps in finding accurate score thresholds. The result of labeling are updated lower and upper bounds on the threshold on similarity, $z.\theta_l$ and $z.\theta_u$, starting from which the predicate is considered satisfied.

Finally, ThalamusDB uses similarity values for a subset of predicates and data items, together with the results of labeling, to calculate an approximate query result. The approximate query result is calculated using the relational DBMS. Similarity scores, the results of model evaluations, are already stored in temporary tables inside of the database. Similarity thresholds, resulting from labeling, are integrated as constants inside of an SQL query.

4.3 Query Rewriting

Finally, in Lines 9 and 10 of Algorithm 1, ThalamusDB calculates upper and lower bounds on the query results which it returns as output to users. To calculate bounds, ThalamusDB processes SQL queries on the database containing the results of the previous execution stages. Those queries exploit similarity scores stored in that database (such scores are only stored for a subset of predicates and items, selected via prioritized data processing, whereas the scores of all other predicate-item combinations are set to the SQL NULL value). Also, it exploits thresholds on similarity values, inferred from user labels ($z.\theta_l$ and $z.\theta_u$ represent lower and upper bounds on the corresponding similarity threshold for predicate z).

Algorithm 2 describes how the original query is rewritten to obtain a set of queries generating the approximate result. This result consists of lower and upper bounds for query aggregates (or, in case of queries without aggregates, in a set of result rows, some of which are marked as tentative since

Input q: Query with NL predicates
Output Rewritten queries (on similarity tables) that compute deterministic bounds for each aggregate.

1: **function** REWRITE(q)
// Join a similarity table per NL predicate to FROM clause.
2: $f \leftarrow q.from$
3: **for all** $z \in q.Z$ **do**
4: $f \leftarrow f$+" JOIN S{$z.id$} ON S{$z.id$}.id = {$z.col$}"
// Initialize WHERE clauses for lower and upper bounds.
5: $w_l \leftarrow q.where$
6: $w_u \leftarrow q.where$
// Replace NL predicate laterals with predicates on similarity scores.
7: **for all** $z \in q.Z$ **do**
// Refer to Table 3 for DEFINITELYTRUE(·) and POSSIBLYTRUE(·).
8: w_l.REPLACE($\neg z$, DEFITRUE($\neg z$)).REPLACE(z, DEFITRUE(z))
9: w_u.REPLACE($\neg z$, POSSTRUE($\neg z$)).REPLACE(z, POSSTRUE(z))
// Case of query without aggregates.
10: **if** $q.aggs = \emptyset$ **:**
11: $sql_l \leftarrow$ "SELECT * {f} {w_l} LIMIT {$q.limit$}"
12: $sql_u \leftarrow$ "SELECT * {f} {w_u} LIMIT {$q.limit$}"
13: **return** $\{sql_l, sql_u\}$
// For each aggregate, add two queries for lower and upper bounds.
14: $Q \leftarrow \emptyset$
15: **for all** $a(col) \in q.aggs$ **do**
16: **if** a = Avg **:**
// For average, replace with sum divided by count.
17: $sql_l \leftarrow$ "SELECT (SELECT Sum(col) {f} {w_l})
/(SELECT Count(col) {f} {w_u})"
18: $sql_u \leftarrow$ "SELECT (SELECT Sum(col) {f} {w_u})
/(SELECT Count(col) {f} {w_l})"
19: **else if** a = Min **then**
// For minimum, w_l and w_u are swapped.
20: $sql_l \leftarrow$ "SELECT $a(col)$ {f} {w_u}"
21: $sql_u \leftarrow$ "SELECT $a(col)$ {f} {w_l}"
22: **else**
23: $sql_l \leftarrow$ "SELECT $a(col)$ {f} {w_l}"
24: $sql_u \leftarrow$ "SELECT $a(col)$ {f} {w_u}"
25: $Q \leftarrow Q \cup \{sql_l, sql_u\}$
26: **return** Q

Algorithm 2. Query rewriting of ThalamusDB.

it is unclear whether they satisfy all associated predicates). First, the rewriting procedure augments the original FROM clause by adding all relevant similarity tables, storing text-item similarity values for natural language predicates, together with suitable join conditions (Lines 2 to 4). Next, two different variants of the SQL WHERE clause are generated (Lines 5 to 9). The first one, w_l, selects rows which may or may not satisfy all natural language predicates. The second one, w_u, selects rows that definitely satisfy all natural language predicates. Due to limited computation and labeling, we cannot verify for all rows whether they satisfy all predicates (e.g., 'Unknown' in Figure 5 where the score thresholds are too broad to determine its state). The two different WHERE clauses are used to calculate lower and upper result bounds, expressing that uncertainty. To obtain these two variants, occurrences of natural language predicates are substituted according to Table 3. Negated predicate

Table 3. Rewrite rules for NL predicate occurrences.

Original SQL	`DefinitelyTrue(·)`	`PossiblyTrue(·)`
`NL(col, str)`	score$\geq \theta_u$	score$> \theta_l$ `OR score IS NULL`
`NOT NL(col, str)`	score$\leq \theta_l$	score$< \theta_u$ `OR score IS NULL`

```
SELECT Min(A.Price) FROM Ads A
  JOIN Pics P USING (AdID)
  JOIN S0 USING (FilePath)
WHERE Created > DATE'2022-01-01'
  AND (S0.Score > θl OR S0.Score IS NULL)
```

Fig. 6. Rewritten query that computes lower bound on the minimal price of ads offering chairs after January 1st 2022.

occurrences are treated separately and replaced before the others. Note the reference to NULL values in Table 3. If a row in the similarity table has a NULL value, the corresponding predicate has not been evaluated on the associated data item.

Queries without aggregates are a special case and handled in Lines 10 to 13. Here, the approximate result are two row sets, one definitely satisfying all predicates, the other one possibly satisfying all predicates. These result sets can be generated directly, based on the two `WHERE` clause variants obtained in the previous step.

Finally, in Lines 14 to 29, Algorithm 2 iterates over different aggregates in the select clause. For each aggregate, it generates two queries (collected in query set Q), calculating the lower bound for that predicate and the upper bound respectively. For instance, for maxima, sum, and count aggregates, the upper bound is obtained by considering all rows that possibly satisfy the predicates (assuming non-negative values in aggregation columns). Considering only rows definitely satisfying all predicates yields lower bounds on the aggregate result value. For the minimum, considering all rows possibly satisfying predicates leads to a lower bound, restricting the scope to rows definitely satisfying predicates yields an upper bound. For the average, lower and upper bounds can be obtained by reduction to lower and upper bounds of the sum and the count. Algorithm 2 ultimately returns the set of rewritten queries which are executed in Algorithm 1, generating the approximate result.

Example 3. *To calculate bounds on the price of offers for wooden chairs with blue cushions, ThalamusDB would run the rewritten query in Figure 6 to compute the lower bound for the minimal price. In its FROM clause, it refers to the original tables, Ads and Pics, as well as to table S0. The latter table contains similarity scores for the items that were selected during prioritized processing (see Section 4.1).*

The rate at which these result bounds narrow depends on the specific predicate and the data distribution. For this minimum query, ThalamusDB would ideally employ prioritized processing, sorting ads in ascending order by price. However, consider a challenging scenario where there are 1,000 ads selling pencils (priced lower than the cheapest chair) and a user constraint on computational overheads, limiting the maximum number of items that can be evaluated to 500. In such a case, some pencil ads remain unevaluated, leading to a less accurate lower bound. This is because, as shown in Table 3, the lower bound is calculated based on rows that may satisfy the predicate, i.e., `score` $> \theta_l$ `OR score IS NULL`*. To obtain a lower bound with zero error, it is necessary to evaluate all pencil ads*

Input q: Query with NL predicates; o: Performance objectives
Output Optimal execution plan

function OPTIMIZE(q, o)
  // Generate all possible actions for the query.
  $A \leftarrow \emptyset$
  for all $z \in q.Z$ **do**
    // Add computation actions.
    $A \leftarrow A \cup \{\langle$'compute', z, "NULL"$\rangle\}$
    for all $col \in q.cols$ **do**
      $A \leftarrow A \cup \{\langle$'compute', z, "{col} ASC"$\rangle\}$
      $A \leftarrow A \cup \{\langle$'compute', z, "{col} DESC"$\rangle\}$
    // Add labeling request actions.
    $A \leftarrow A \cup \{\langle$'labeling', $z\rangle\}$
  // Find a plan that satisfies the user constraint.
  $e \leftarrow 0^A$
  $\delta \leftarrow 1$
  while $\neg$ COST$(e, q) \preceq o.b$ **:**
    // δ: exponentially increasing increment value ($\beta \geq 1$).
    $\delta \leftarrow \beta\delta$
    // Add an action that improves the constrained metric the most.
    // u_a: unit vector where the dimension of a has a value of one.
    $a^* \leftarrow \arg\min_{a \in A} \max_m(\text{COST}(e + \lfloor\delta\rfloor u_a, q)_m - o.b_m)$
    $e_{a^*} \leftarrow e_{a^*} + \lfloor\delta\rfloor$
  // Local search until no cost improvement.
  $e^* \leftarrow$ Null
  $\delta \leftarrow 1$
  while $e^* \neq e$ **:**
    $e^* \leftarrow e$
    $c \leftarrow \text{COST}(e, q)$
    $\delta \leftarrow \beta\delta$
    // Iterate over neighboring plans with distance $\lfloor\delta\rfloor$.
    for all $e' \in \{e + s\lfloor\delta\rfloor u_a | s \in \{-1, +1\}; a \in A\}$ **do**
      // Update if e' satisfies constraint and has better cost.
      $c' \leftarrow \text{COST}(e', q)$
      if $c' \preceq o.b$ **and** $c' \cdot o.w < c \cdot o.w$ **:**
        $e \leftarrow e'$
        break
  return e^*

Algorithm 3. Optimization process of ThalamusDB.

before ThalamusDB can identify the first chair with the lowest price. Nevertheless, our experiments on real-world datasets demonstrate that minimum queries are, in fact, scenarios where significant performance gains are observed compared to baselines. For example, in our Craigslist dataset, there are ads of people offering chairs for free, avoiding the aforementioned case.

5 QUERY OPTIMIZATION

Algorithm 3 is executed by the query optimizer component. As input, the optimizer considers the query to optimize as well as user-specified preferences and constraints with regards to performance tradeoffs. The optimizer performs multi-objective optimization, considering the three metrics approximation error, execution time, and number of labels requested from users. As discussed in

more detail in the last section, approximation errors are due to missing labels or missing similarity scores for natural language predicates. At the same time, processing overheads are dominated by inference time of large neural networks, making SQL query processing time negligible. Hence, in summary, plan processing costs according to all three metrics depend mostly on the treatment of natural language predicates. Therefore, optimization focuses on carefully selecting data subsets for labeling and predicate evaluations.

Plan Space. Execution plans are represented as vectors. Vector components represent possible actions of the execution engines. Actions are divided into two categories: requesting labels from users and calculating similarity scores for predicates. Each type of action is parameterized by the target predicate as well as, optionally, by a row sort order. For each action, the execution plan vector contains an integer number, representing the number of rows (in sort order) to which the operation is applied. Algorithm 3 collects relevant actions in Lines 2 to 8. It considers all natural language predicates in the query ($q.Z$). For each predicate, it adds actions representing model invocations in Lines 4 to 7. We consider the default row order (Line 4) as well as sorting by any column that appears in the query ($q.cols$). As discussed in Section 6, approximation error may vary depending on the subset of rows to which operations are applied.

Satisfying User Constraint. Given the set of relevant actions, A, the optimizer initializes an empty plan (setting each vector component to zero in Line 9). Next, the optimizer greedily adds actions to obtain a plan that satisfies all cost constraints, imposed by the user (Lines 11 to 14). Given an execution plan and a vector, Function Cost returns a vector containing cost estimates for all three cost metrics. The only cost metric that can improve by adding more actions is approximation error. Hence, if user-defined error constraints on error are not satisfied, the optimizer greedily adds actions to minimize approximation error. This process is guaranteed to terminate: if all similarity scores are calculated and all labels obtained, the approximation error must reduce to zero.

Local Search for Optimal Plan. Next, the optimizer improves the plan via local search until a local optimum is found (Lines 15 to 25). The loop terminates once the best plan in the previous iteration equals the best plan after the current iteration. Each iteration considers neighbors of the currently selected plan in the search space, obtained by changing the number of actions for one action type (i.e., one vector component). Plans are evaluated based on their cost vectors. First, each plan must satisfy the user-defined cost constraints, expressed via cost vector $o.b$. The expression $c' \preceq o.b$ denotes a multi-dimensional comparison, evaluating to true only if the cost vector c' of the current plan is below or at the bound $o.b$ for every cost metric. Second, a plan satisfying the cost constraint is evaluated according to its weighted cost, $c'.o.w$ (i.e., the dot product between plan cost vector and cost weights). An iteration ends once a new plan is found which improves weighted cost.

When constructing initial plans as well as during local search, we vary the "step size", i.e., the number of actions added or subtracted in one iteration. This is motivated by our initial experiences with a fixed step size, leading to slow optimization for plans that require many actions to satisfy user constraints. Using a large step size, however, may lead to sub-optimal results for plans that require few actions. Hence, we generally start with a small step size that is increased over time, thereby optimizing small plans within a more fine-grained search space than large ones. Here, β is a tuning parameter, determining how quickly step size increases. Note that we floor raw float values, representing the number of actions, as the final number of actions must be an integer. We performed experiments, summarized in Section 7.3, showing that a variable step size leads to performance improvements.

6 ERROR ESTIMATION MODEL

The optimizer described in Section 5 relies on models, estimating processing costs, the number of labels, and the approximation error after plan execution. ThalamusDB predicts cost according to the first two metrics via standard methods (using the number of labeling requests as well as the number of model invocations, multiplied by the average time per invocation that ThalamusDB measures during pre-processing). This section describes how ThalamusDB estimates approximation error after executing a specific plan.

6.1 Estimating Selectivity

To estimate approximation error, we first must estimate the selectivity of natural language predicates. After plan execution, given one specific predicate, table rows can be classified into rows that definitely satisfy that predicate, rows that definitely do not satisfy the predicate, and rows for which it is unclear whether or not they satisfy the predicate. ThalamusDB estimates the ratio of rows in each category. First, ThalamusDB estimates the threshold θ_u on the similarity value (similarity between predicate text and analyzed item), starting from which the predicate is considered definitely satisfied. Also, it estimates the threshold θ_l below which the predicate is considered definitely not satisfied (the predicate status is undetermined for similarity values in $[\theta_l, \theta_u]$). Both thresholds are determined by user answers to labeling requests. As user answers are hard to predict, ThalamusDB assumes a uniform distribution over both possible answers for each (binary) labeling request and calculates estimates under that assumption. Second, ThalamusDB estimates the ratio of rows whose similarity values are known and above θ_u, below θ_l, or in the range $[\theta_l, \theta_u]$. The number of rows whose similarity values are known is directly given by the plan (specifying for each predicate the number of rows for which the associated multi-modal model is applied). Based on similarity values obtained during pre-processing for the same predicate and a data sample, ThalamusDB estimates the ratios of rows with similarity values above and below the two thresholds.

Estimating selectivity for composite predicates is more challenging. In particular, we need to take into account that different predicates may have been processed on different row subsets. Consider two natural language predicates involved in a composite predicate (i.e., a conjunction or disjunction). As it is common in the domain of selectivity estimation [39], we assume statistical independence between both predicates. Denote by S the set of sort orders considered by the optimizer (based on columns that appear in query aggregates). Further, denote by $p_{i,s}$ for $i \in \{0, 1\}$ and $s \in S$, the ratio of rows, selected according to sort order s, for which predicate i was evaluated. Then, we can write $p = \sum_{s \in S} \min_i(p_{i,s})$ to capture the total ratio of rows on which both predicates were evaluated. Considering only those rows, the ratio of rows known to satisfy a conjunction of predicates is $p\hat{t_1}\hat{t_2}$ where $\hat{t_i}$ for $i \in \{0, 1\}$ refers *only* to rows on which the i-th predicate was evaluated. It denotes the ratio of rows, known to satisfy that predicate after execution, within that row subset (and is estimated from the pre-processed sample). We denote by $r_i = \sum_{s \in S} p_{i,s} - p$ the ratio of remaining rows processed for the i-th predicate. For those rows, it is not guaranteed that the other predicate was evaluated as well. However, there is a chance that rows in different orders overlap. More precisely, we expect ratio $r_1 \cdot r_2$ of rows on which both predicates are evaluated, even though the corresponding evaluation actions used different row orders. The expected ratio of rows known to satisfy all predicates among them is $r_1 r_2 \hat{t_1}\hat{t_2}$. Hence, in total, the expected ratio of rows known to satisfy the conjunction, is $\hat{t_1}\hat{t_2}(p + r_1 r_2)$.

Denoting by $\sigma_T(x)$, $\sigma_F(x)$, and $\sigma_U(x)$ the expected ratio of rows known to be true, false, or not known to be either for a predicate x, after execution, we generally have $\sigma_F(x) = \sigma_T(\neg x)$, $\sigma_U(x) = 1 - \sigma_T(x) - \sigma_F(x)$, and $\sigma_T(x_1 \vee x_2) = \sigma_T(\neg(\neg x_1 \wedge \neg x_2))$. Hence, we can apply the same

reasoning as before to obtain estimates for all of the aforementioned selectivity values and predicate types.

6.2 Estimating Error

We exploit selectivity estimates to estimate error bounds for different types of aggregates, after plan execution. We denote by n_T the expected number of join result rows, known to satisfy all predicates. By n_Ux, we denote the number of rows with unknown status. For aggregates of type `count(*)`, lower and upper bounds are obtained immediately from selectivity estimates as $[n_T, n_T + n_U]$. For aggregates of type `sum(a)`, we simply multiply the previous bounds with the average value v_a in the aggregation column `a`: $[v_a n_T, v_a(n_T + n_U)]$. We reduce average aggregates to bounds for sum and count aggregates. For an aggregate `avg(a)`, we estimate bounds $[sum_L(a)/count_U(*), sum_U(a)/count_L(*)]$ where $sum_L(a)$, $sum_U(a)$, $count_L(*)$, and $count_U(*)$ denote upper and lower bounds for count and sum aggregates.

We estimate lower and upper bounds for minima, `min(a)`, using an auxiliary function. Consider values $V = \{v_i\}$ such that $v_i \leq v_{i+1}$ (i.e., values are sorted in ascending order). Function $F(V, d, b)$ calculates the expected outcome of the following experiment: considering values in ascending order, we flip a coin with bias b after each value and return that value if the coin flip succeeds. If no coin flip succeeds, default value d is returned. We use this function in our scenario as follows. Denote by V_a the values that appear in the aggregation column `a` and by $d = m_a$ the maximal value for this column in the database. We set $b = \hat{t}$ where $\hat{t}$ is the probability (calculated as described in the previous section) that all predicates evaluate to (certainly) true after evaluation. Considering values in column a in ascending order, the first row satisfying all predicates provides us with an upper bound on the minimum (the largest value serves as a default bound). Function $F(V_a, m_a, \hat{t})$ calculates that upper bound. Similarly, considering rows in ascending order of their value in column `a`, starting with the minimal value in the entire database, the first row whose predicates evaluate to true or to unknown status provides us with a lower bound on the minimum. Setting $b = \hat{t} + \hat{u}$ (i.e., the probability of true or unknown status), V_a to the minimal values in column `a`, and $d = \max(V_a)$, the expression $F(V_a, d, \hat{t} + \hat{u})$ yields a lower bound as well.

We calculate $F(V, d, b)$ as follows. The probability of a successful coin flip is b. Hence, the probability to return the first value in V is b. The probability to return the second value is $b \cdot (1 - b)$ (the probability to succeed in the second try but not in the first). Generalizing this reasoning, we obtain $\sum_i b \cdot (1 - b)^{i-1} \cdot v_i + (1 - b)^{|V|} \cdot d$ for the expected value. Evaluating this expression would require retrieving and iterating over database content. This may be costly for large databases (as the cost function is invoked repeatedly during optimization). Instead, we approximate F by assuming uniform distribution of values between the minimal and maximal value in V. This simplifies the previous formula to $\sum_i b \cdot (1-b)^{i-1} \cdot (\min(V) + \delta \cdot i) + (1-b)^{|V|} \cdot d$ where $\delta = (\max(V) - \min(V))/|V|$ is the average distance between elements in V. For estimating bounds, the relevant value ranges derive from the row orders that appear in the execution plan. We use column value statistics to estimate minima and maxima for specific row orders and specified number of rows evaluated. We omit the discussion of maxima as well as summations and counts with certain sort orders, which are handled in a similar manner.

7 EXPERIMENTAL EVALUATION

Section 7.1 introduces the experimental setup. Section 7.2 presents an end-to-end performance comparison of ThalamusDB to baselines, including MindsDB [30] and ABAE [20]. Section 7.3 analyzes ThalamusDB in detail and Section 7.4 provides error analysis.

Table 4. Overview of benchmark data sets.

Dataset	Table	#Rows	#Columns	Multi-Modal Column
YouTube	Videos	46,774	11	AudioClip `AUDIO` Description `TEXT`
Craigslist	Pics	14,674	2	FilePath `PICTURE`
	Ads	3,000	7	Title `TEXT`
Netflix	Movies	17,770	4	FeaturedReview `TEXT`
	Ratings	100M	4	-

7.1 Experimental Setup

Benchmarks. We created three benchmarks for multi-modal data processing, based on real-world data from YouTube, Craigslist, Netflix, and IMDb. Table 4 shows an overview of the properties of the data sets. The YouTube data set contains information about videos on YouTube. The `Videos` table consists of columns on multi-modal data, including audio and text. The `AudioClip` column contains paths to audio files from the AudioCaps data set [10, 23]. The AudioCaps data set consists of short audio clips extracted from YouTube. We augment the AudioCaps data set by extracting information (e.g., title, description, viewcount, and likes) from the corresponding video posts in relational format. The Craigslist data set contains information (e.g., pictures, price, title) about ads on the Craigslist website. We crawled 3,000 ad posts on furniture for sale, with at least one picture per ad. The data set consists of the `Ads` table as well as the `Pics` table since there can be more than one picture for an ad. The Netflix data set [16] consists of movies and their ratings. We introduce a new column in the `Movies` table by augmenting each movie with a review from the IMDb reviews data set [28]. The YouTube, Craigslist, and Netflix benchmarks consist of 24, 25, and 6 queries, respectively (the Netflix benchmark has fewer queries due to having one multi-modal column). The benchmark consists of Min, Max, Sum, Avg, and Count aggregation queries as well as select queries with Limit clauses. We introduce NL predicates on audio, image, and text columns described in Table 4 in our benchmark queries. Every query includes at least one NL predicate and may also include a combination of two predicates linked by conjunction or disjunction. For the Craigslist and Netflix benchmarks, we introduce queries that join the `Pics` table with the `Ads` table and the `Movies` table with the `Ratings` table. To experiment with a large number of configurations efficiently, we simulate user answers to labeling requests, based on manually determined thresholds. Thresholds range from -28 to 27 and vary significantly even for the same model and column, depending on the predicate (e.g., between 16 and 27 for the `Pic.FilePath` column with predicates `'wooden'` and `'leather sofa'`). This wide range of thresholds justifies calibrating models with efficient labeling requests.

Baselines. We compare our system against two recent baselines that support a subset of our benchmark queries. MindsDB [30] is a data processing system that integrates machine learning (ML) into the database. At present (v23.1.3.2), it lacks support for image and audio classification. ABAE [20] is a query processing system for accelerating aggregation queries with expensive predicates involving costly ML methods. The system utilizes stratified and pilot sampling techniques and thus supports Sum, Count, and Avg aggregates. Note the the current version (v0.3.7) of EvaDB [17] does not support zero-shot classification, making it difficult for comparison.

Implementation. ThalamusDB is implemented in Python 3. It uses DuckDB [35] as the underlying relational DBMS. We use CLIP [36] for image processing, the model by Mei et al. for audio processing [29], and Sentence-BERT [37] as well as BART [26] for text processing. Taking into

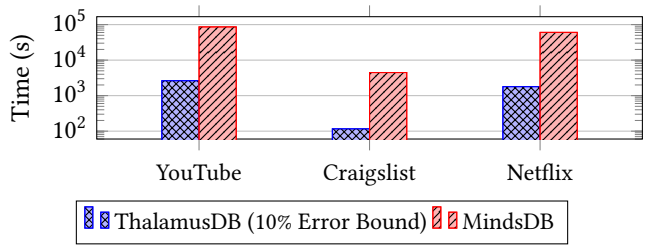

Fig. 7. Runtime comparison against MindsDB on all benchmarks.

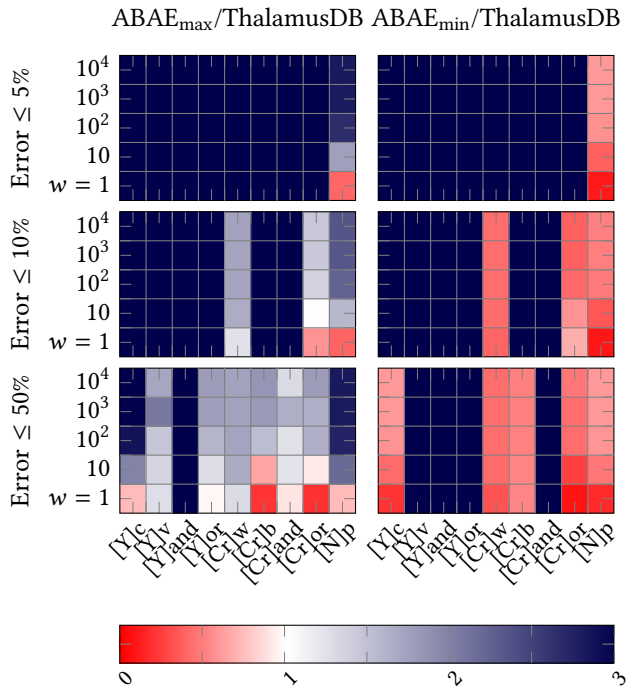

Fig. 8. Scaled cost of ABAE compared to ThalamusDB on all benchmarks (Y, Cr, N) with different predicates ('and' for conjunction, 'or' for disjunction, and the rest are single predicates). A higher degree of blue shade indicates a lower cost for ThalamusDB compared to ABAE (while red indicates the opposite).

account the inference speed of each model, ThalamusDB processes 1% of data items for images and texts and 0.1% for audio per computation step. During preprocessing, we request three labels and process one computation step per NL predicate in the query. All experiments were executed on a server with two Intel Xeon Gold 5218 CPUs (2.3GHz with 32 physical cores), 384 GB of RAM, and two GeForce RTX 2080Ti GPUs.

7.2 Comparison against Baselines

MindsDB. In Figure 7, we present the computation overheads (i.e., runtime) of ThalamusDB and MindsDB on all queries supported by MindsDB. MindsDB is limited to queries that involve natural language predicates on text data types. Hence, it is able to execute 12 out of our 49 benchmark queries. Furthermore, to address the lack of support for collecting labels in MindsDB, we provide

the system with ground truth score thresholds for NL predicates (without counting additional cost). For ThalamusDB, we apply an error constraint of 10%. ThalamusDB achieves an average speedup of 35.0× over MindsDB, due to its approximate query processing and prioritized data evaluation. For single queries, ThalamusDB achieves up to 1,648.4× speedup. In this experiment, we employ BART in both systems because of technical issues related to running BERT on MindsDB. We use BERT in the rest of our experiments.

ABAE. In Figure 8, we present the scaled costs of ABAE, compared to ThalamusDB. A greater intensity of blue color indicates lower cost for ThalamusDB, compared to ABAE (red color represents the opposite). ABAE available on [18] is able to run 8 out of 49 benchmark queries. Similar to MindsDB, we provide ABAE with labels for NL predicates with no computational overheads. We evaluate two versions of ABAE: one collects three labels per NL predicate (named "ABAE_{min}"), the minimal number collected by ThalamusDB, and the other collects the maximal number of labels to determine precise thresholds (named "ABAE_{max}"). ABAE requires an efficient proxy model per predicate to compute proxy scores for all data items. We provide ABAE with ground truth similarity scores evaluated on the oracle models as proxy scores at no cost (i.e., without counting any computational overheads). We opt for a confidence value of $1 - 10^9$ for ABAE to ensure a high level of confidence, approaching the 100% confidence guarantee of ThalamusDB as closely as possible. MindsDB employs exact query processing, whereas ThalamusDB and ABAE exploit approximation. Thus, for both systems, we vary the error constraint between 5%, 10%, and 50% and the weight values between 1, 10, 10^2, 10^3, and 10^4. The darkest shade of blue indicates the highest cost differences or cases where ABAE fails to meet the error constraint, despite processing the same number of rows as the entire dataset. ThalamusDB has lower costs than ABAE_{max} and ABAE_{min} for the majority of cases (91.1% and 67.7%, respectively). On average, ABAE_{max} incurs 2.4× cost overhead and ABAE_{min} incurs 2.1× cost overhead, compared to ThalamusDB. In general, ThalamusDB outperforms ABAE by a larger margin when the error constraint is stricter (e.g., 5%). ABAE struggles to meet the error constraint for highly selective predicates (e.g., conjunction) since it is based on sampling. In the next subsection, we further present how predicate selectivity affects the performance of our system.

7.3 Further Analysis

We demonstrate that ThalamusDB has the capability to adapt to user preferences and achieve optimal performance. We evaluate ThalamusDB with six different user preference configurations, denoted as Cl, cL, Ce, cE, Le, and lE in the following plots. Here, we use letters c, l, and e to denote computation overheads (in seconds), the number of labels, and the error. The two letters associated with a configuration represent the two metrics that appear in the cost formula, lowercase letters represent small weights (here: 1) while uppercase letters represent large weights (10^4). In each case, the third metrics (not referenced by any letter) is constrained using thresholds: 10 seconds, five labels per predicate, and 10% for computation overheads, the number of labels, and the error, respectively.

Simplified Versions of ThalamusDB. For an ablation study, we introduce three simplified versions of ThalamusDB. First, we consider an exact processing baseline (named "Exact") that computes similarity scores for all data items constrained by NL predicates. It then finds the precise similarity threshold for each predicate via labeling requests using binary search. The second baseline (named "+Ordered") uses the optimal evaluation order for NL predicates (considering selectivity and evaluation costs) to reduce computational overheads. The third baseline (named "+Approx.") exploits deterministic approximation, thereby reducing the amount of data that needs to be processed to answer a query. ThalamusDB additionally uses a cost-based query optimizer that optimizes execution plans according to user preferences and constraints.

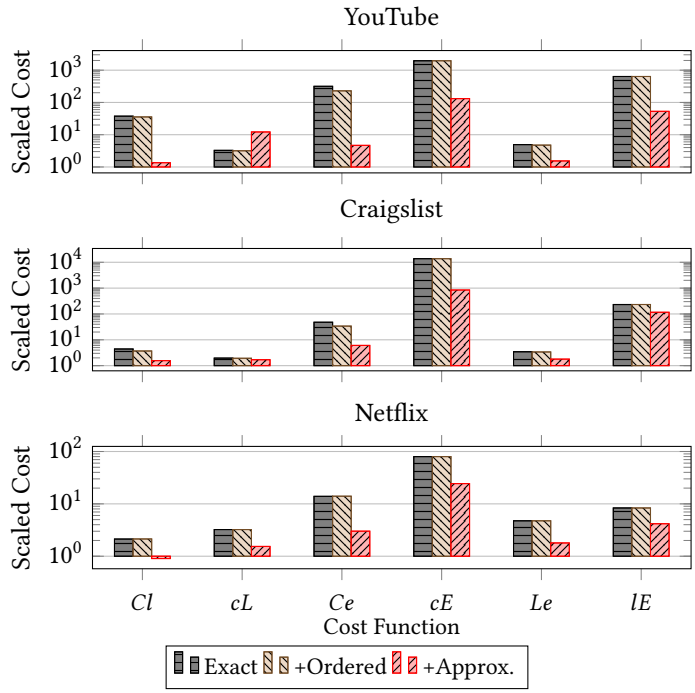

Fig. 9. Scaled costs of simplified versions of ThalamusDB compared to the final version (i.e., the final version, not plotted, has a scaled cost of one).

Figure 9 shows the scaled costs of different ablations compared to ThalamusDB (i.e., ThalamusDB has a scaled cost of one). We present average values for all benchmarks and different performance objective specifications. Compared to ThalamusDB, on average, the Exact baseline has a 951.5× average cost overhead across all benchmarks. The +Ordered baseline demonstrates better performance, compared to Exact, but, compared to ThalamusDB, has a 945.5× average cost overhead. The +Approx. baseline is able to reduce computational overheads, compared to the other two baselines, but suffers from the lack of an optimizer to select most effective actions. As a result, it suffers 34.0×, 162.1×, and 6.0× average cost overheads for YouTube, Craigslist, and Netflix, compared to ThalamusDB.

As part of another ablation study, we deactivate the coarsening scheme integrated in the optimization process. In the YouTube benchmark, out of the 12 cases that are particularly time-consuming in terms of optimization time, using a variable step size ($\beta = 1.01$) improves the optimization speed, compared to a fixed step size ($\beta = 1$), by an average factor of 2.4× with comparable result quality.

Runtime Breakdown. Figure 10 depicts the runtime breakdown of different processing parts of ThalamusDB into percentages. Across all cases, the computation of similarity scores through model inference constitutes the majority of the runtime. On average, model inference makes up 92.4% of the total runtime for YouTube, 89.6% for Craigslist, and 50.7% for Netflix. In contrast, relational processing, which includes sorting data for prioritized data processing, accounts for an average of 4.1%, 2.5%, and 45.5% for YouTube, Craigslist, and Netflix, respectively. Netflix has a higher percentage for relational processing compared to other benchmarks. This is because the natural language predicates in the Netflix benchmark apply only to text data and not to image or audio data. Note also that this breakdown is based on the runtime after our optimization to minimize

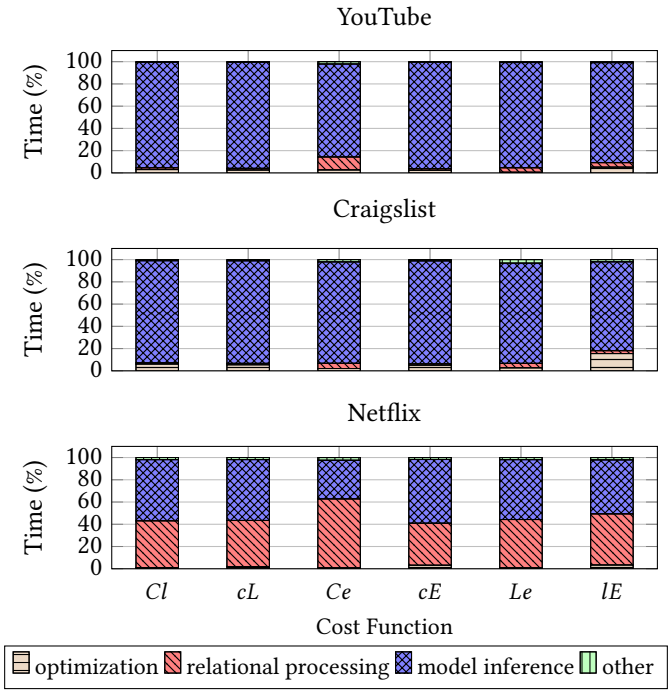

Fig. 10. Runtime breakdown of ThalamusDB.

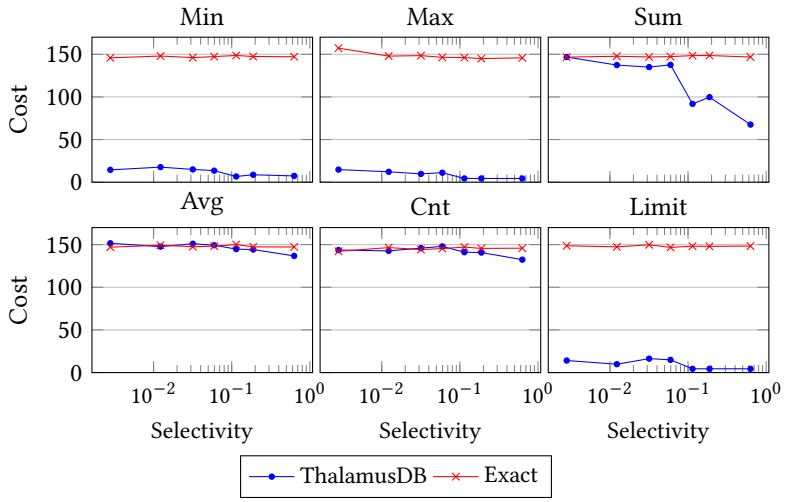

Fig. 11. Comparison of ThalamusDB and the Exact baseline on multiple NL predicates with varying selectivity.

the inference overheads. The expensive nature of model inference justifies the effort invested in optimizing data processing for reduced processing overheads.

Predicate Selectivity. Figure 11 illustrates the impact of predicate selectivity on the performance of ThalamusDB. We compare against the Exact baseline on the Craigslist data set, using benchmark queries with a single predicate on the `Pic.FilePath` column. Specifically, we apply seven NL

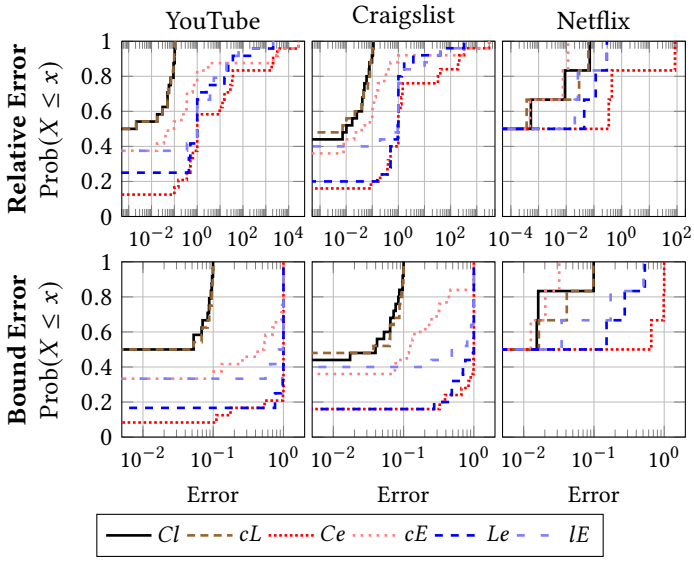

Fig. 12. Cumulative distribution function of relative errors (first row) and bound errors (second row) for all benchmark queries.

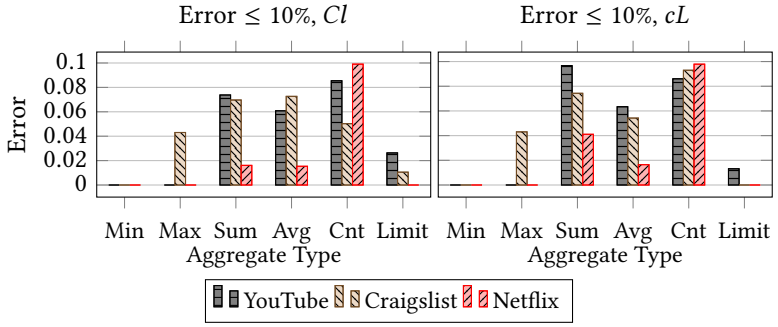

Fig. 13. Bound errors across various aggregate types.

predicates with varying selectivity: `leather sofa`, `blue chair`, `wooden chair`, `dining table`, `sofa`, `table`, and `wooden`. We use a performance objective with a constraint on the error metric since the Exact baseline fails to satisfy constraints on the other two metrics for most of the queries. We use a weight value of $w = 1$, presenting the cost value for each predicate and aggregate function. For Sum aggregates, ThalamusDB shows larger cost improvements over the Exact baseline for less selective predicates. By prioritizing data items with large values in the aggregated column, ThalamusDB can process fewer data items to satisfy the error constraint. However, when predicates are highly selective, ThalamusDB needs to process almost all data items to meet the error constraint. ThalamusDB demonstrates large cost improvements for Min, Max, and Limit queries. For Count and Avg aggregates, ThalamusDB show small improvements (note that the Netflix benchmark shows better improvements on these two aggregate types). Nonetheless, in all cases, there is a tendency of cost improvements for less selective predicates.

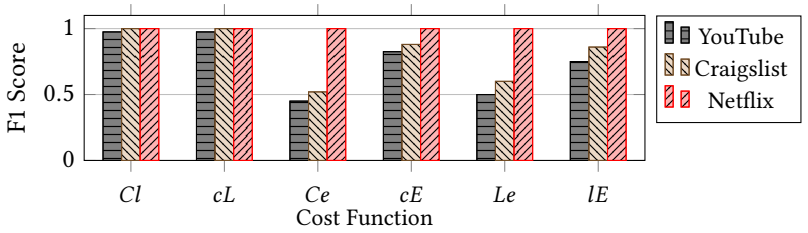

Fig. 14. F1 scores of select queries with limit clause.

7.4 Error Analysis

We provide detailed insights into the approximation errors of deterministic bounds generated by ThalamusDB. We consider two types of errors: 1) bound error and 2) relative error. First, the bound error is the error defined in Definition 5 as $(u - l)/(u + l)$, which measures the relative distance between the lower and upper bounds l and u [24]. This error is advantageous as it does not require us to know the true value v_t. Thus, it is the error metric utilized during query optimization in ThalamusDB. Second, the relative error is computed using the formula $|(v_t - v_e)/v_t|$, where v_t is the true value and v_e is the estimated value of v_t. We obtain v_t through exact processing, in which all items are evaluated and a sufficient number of labels are provided to determine the precise score threshold. This value of v_t is indeed the ground truth value if the model exhibits perfect precision. For the estimated value v_e, we calculate it as the midpoint between the deterministic bounds, using the formula $(u - l)/2$.

Relative Error and Bound Error. Figure 12 displays the cumulative distribution functions of relative errors and bound errors for all queries across each benchmark. It demonstrates that ThalamusDB effectively generates appropriate bounds in accordance with the specified performance objective. When the bound error is constrained to 10% (Cl and cL), the cumulative probability indeed approaches 100% as the error nears 10%. This observation holds true not only for the bound error but also for the relative error. As proposed in [24], this demonstrates that bound error is a good alternative to relative error when the true value is unknown. Performance configurations that assign large weights to the error metric (cE and lE) accumulate probability more rapidly, indicating lower errors, compared to their counterparts (Ce and Le). For instance, at the 50th percentile, cE displays relative errors of 5.2% and 3.9% for YouTube and Craigslist, respectively, whereas Ce exhibits an error of 100%. Similarly, in the Netflix benchmark, cE shows a relative error of 0.8% at the 80th percentile, while Ce has an error of 44.4%.

Error per Aggregate Type. Figure 13 illustrates the approximation error for each aggregate type. We employ a performance objective constrained by the error metric. ThalamusDB yields lower errors for Min, Max, and Limit aggregates compared to Sum, Avg, and Count aggregates. As minima and maxima are determined by single items, the potential of prioritized processing increases. In all cases, the bound error is less than or equal to 10%, meeting the error constraint. Figure 14 displays the F1 scores of Limit queries across all benchmarks. It shows the highest F1 scores when the error is constrained (Cl and cL), followed by the second-highest scores for large weights on error (cE and lE), and lower scores when the error is unconstrained and not given high weights (Ce and Le).

Impact of Pre-processing. Figure 15 displays the relative error in estimated selectivity across different numbers of items processed during the pre-processing phase. As more items are processed to collect statistics, the estimates become more accurate. However, there are diminishing returns with increasing numbers of pre-processed items. Furthermore, extensive pre-processing limits the ability of ThalamusDB to reduce computational overheads.

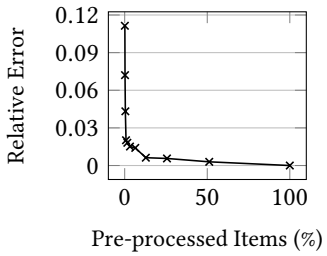

Fig. 15. Relative error between estimated and true selectivity across varying percentages of pre-processed items.

Table 5. Ratio and error improvement of post-processing actions for performance objectives with error ≤ 10%.

Benchmark	Cost Function	$\frac{\#\text{ Post-Processing}}{\#\text{ All Actions}}$	Error Improvement
YouTube	*Cl*	0.183	0.153
YouTube	*cL*	0.248	0.246
Craigslist	*Cl*	0.145	0.254
Craigslist	*cL*	0.170	0.247
Netflix	*Cl*	0.007	0.001
Netflix	*cL*	0.000	0.000

Impact of Post-processing. Table 5 presents the ratio and error improvement of post-processing actions across all benchmarks. The ratio of post-processing actions to the total number of actions is relatively small, being less than 25% for all benchmarks and cost functions. Similarly, the error improvement due to post-processing actions is also modest (under 26% in all cases), suggesting that the initial error was already reasonably close to the target error.

Impact of Model Accuracy. Figure 16 illustrates the lower and upper bounds produced by ThalamusDB on the Netflix benchmark, using two models with varying levels of accuracy: Sentence-BERT (all-MiniLM-L6-v2) and BART. This figure compares these bounds against the exact query results that employ the ground truth labels from the Netflix benchmark. BART, being a larger model than Sentence-BERT (with pytorch_model.bin sizes of 1.02 GB compared to 90.9 MB), allows ThalamusDB to establish bounds closer to the ground truth value (for queries q3 and q5). Note that these bounds might not contain the exact query result due to the imperfect accuracy of the models. However, ThalamusDB consistently delivers accurate result bounds using the larger model in all cases, except for query q3, where the results are still very close to the ground truth value.

8 RELATED WORK

Zero-shot Models. Traditional classifiers, like those trained with ImageNet [14], have evolved with the advent of zero-shot and few-shot models, which leverage vast pre-training on web-scale text data to comprehend and perform classification tasks from natural language descriptions alone [5]. These models have permeated various domains including computer vision [36] and audio processing [29], but often face limitations in handling large input sizes and require frameworks like ThalamusDB to deconstruct complex queries into more manageable processing steps for efficient processing.

Data Processing Frameworks using Deep Neural Models. While related works such as SeeSaw [31], Symphony [7], VIVA [22], and TASTI [21] utilize deep neural models for various data processing and query-answering tasks, ThalamusDB distinctively supports queries involving natural language predicates on multi-modal data using zero-shot models. Evidenced by significant

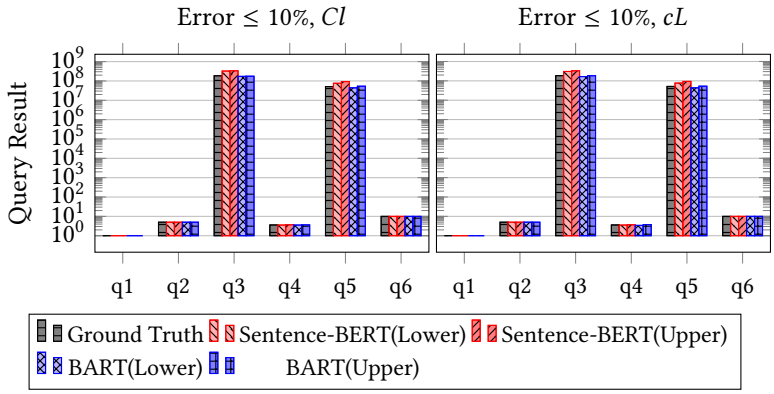

Fig. 16. Impact of model accuracy on ThalamusDB. Comparison between ThalamusDB bounds using models of varying accuracy against query results from ground truth labels.

speed-ups in experiments, ThalamusDB also demonstrates pronounced efficiency through its use of approximate query processing, compared to exact processing systems like MindsDB [30], EvaDB [17], and SQLFlow [40]. Other recently proposed frameworks on top of neural models often focus on video data processing [8, 11, 19, 22, 25].

Thresholds for Classification. Informed by studies on multi-label classification [13, 38], ThalamusDB uses thresholds to assign classes to instances. In multi-label classification, it is customary to compute relevance scores for label-instance pairs and then assign labels whose scores exceed a threshold value [13]. ThalamusDB is designed to serve the needs of a single user and thus assumes that the user has the best knowledge of what they wish to classify as True. Hence, ThalamusDB determines the score threshold per natural language predicate by soliciting a small number of labeling requests from the user.

Approximate Query Processing. ThalamusDB relates to a large body of prior work on approximate query processing [6]. Contrary to approaches employing data sampling [1–3, 12, 32, 33] including ABAE [20], ThalamusDB utilizes deterministic approximation methods [4, 15, 27, 34], ensuring 100% confidence in its result bounds. ABAE requires a proxy model per predicate (to do better than random sampling), which calculates approximate scores for the predicate and is much cheaper to evaluate compare to oracle models. In contrast, ThalamusDB does not rely on proxy models.

9 CONCLUSION

ThalamusDB is a deterministic approximate query processing system for running complex queries on multi-modal data. It extends a relational database system to support natural language predicates on image, audio, and text data. In our experiments on real-world data sets from YouTube, Craigslist, and Netflix, we demonstrate that ThalamusDB is significantly more efficient than several baselines, including MindsDB and ABAE.

10 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

REFERENCES

[1] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. 1999. Aqua: a fast decision support systems using approximate query answers. In *VLDB*. 754–757.

[2] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. 2000. Congressional Samples for Approximate Answering of Group-By Queries. *SIGMOD 2000 - Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (2000), 487–498. https://doi.org/10.1145/342009.335450

[3] Sameer Agarwal, Barzan Mozafari, and Aurojit Panda. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *European Conf. on Computer Systems*. 29–42. arXiv:arXiv:1203.5485v2 http://dl.acm.org/citation.cfm?id=2465355

[4] Etienne Boursier, E N S Paris-saclay, and Chunbin Lin. 2018. Plato: Approximate Analytics over Compressed Time Series with Tight Deterministic Error Guarantees, Vol. 1. https://doi.org/10.1145/nnnnnnn.nnnnnnn arXiv:arXiv:1709.01073v2

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*. 1877–1901. arXiv:2005.14165

[6] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate query processing : no silver bullet. In *SIGMOD*. 511–519. https://doi.org/10.1145/3035918.3056097

[7] 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 *CIDR*.

[8] Maureen Daum, Magdalena Balazinska, Brandon Haynes, Ranjay Krishna, Apryle Craig, and Aaron Wirsing. 2022. VOCAL : Video Organization and Interactive Compositional AnaLytics. In *CIDR*.

[9] Benjamin Elizalde, Soham Deshmukh, Mahmoud Al Ismail, and Huaming Wang. 2022. CLAP: Learning Audio Concepts From Natural Language Supervision. *CoRR* abs/2206.04769 (2022). https://doi.org/10.48550/arXiv.2206.04769

[10] Jort F Gemmeke, Daniel P W Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R Channing Moore, Manoj Plakal, and Marvin Ritter. 2017. Audio set: An ontology and human-labeled dataset for audio events. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 776–780.

[11] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2020. VisualWorldDB : A DBMS for the Visual World. In *CIDR*.

[12] Joseph M. JM Hellerstein, PJ Peter J. Haas, and HJ Helen J. Wang. 1997. Online aggregation. *SIGMOD Record* 26, 2 (1997), 171–182. https://doi.org/10.1145/253262.253291

[13] Yutai Hou, Yongkui Lai, Yushan Wu, Wanxiang Che, and Ting Liu. 2021. Few-shot Learning for Multi-label Intent Detection. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 13036–13044. https://ojs.aaai.org/index.php/AAAI/article/view/17541

[14] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. (2009), 248–255. https://doi.org/10.1109/cvprw.2009.5206848

[15] Saehan Jo and Immanuel Trummer. 2020. BitGourmet: deterministic approximation via optimized bit selections. In *CIDR*. 1–7.

[16] Kaggle. 2023. Netflix Prize data. https://www.kaggle.com/datasets/netflix-inc/netflix-prize-data.

[17] Gaurav Tarlok Kakkar, Jiashen Cao, Pramod Chunduri, Zhuangdi Xu, Suryatej Reddy Vyalla, Prashanth Dintyala, Anirudh Prabakaran, Jaeho Bang, Aubhro Sengupta, Kaushik Ravichandran, Ishwarya Sivakumar, Aryan Rajoria, Ashmita Raju, Tushar Aggarwal, Abdullah Shah, Sanjana Garg, Shashank Suman, Myna Prasanna Kalluraya, Subrata Mitra, Ali Payani, Yao Lu, Umakishore Ramachandran, and Joy Arulraj. 2023. EVA: An End-to-End Exploratory Video Analytics System. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning, DEEM 2023, Seattle, WA, USA, 18 June 2023*. ACM, 8:1–8:5. https://doi.org/10.1145/3595360.3595858

[18] Daniel Kang. 2023. Accelerating Approximate Aggregation Queries with Expensive Predicates. https://github.com/stanford-futuredata/abae.

[19] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. Blazelt: Optimizing declarative aggregation and limit queries for neural networkbased video analytics. *VLDB* 13, 4 (2019), 533–546. https://doi.org/10.14778/3372716.3372725 arXiv:1805.01046

[20] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating Approximate Aggregation Queries with Expensive Predicates. *Proc. VLDB Endow.* 14, 11 (2021), 2341–2354. https://doi.org/10.14778/3476249.3476285

[21] Daniel Kang, John Guibas, Peter D. Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. TASTI: Semantic Indexes for Machine Learning-based Queries over Unstructured Data. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1934–1947. https://doi.org/10.1145/3514221.3517897

[22] Daniel Kang, Francisco Romero, Peter Bailis, Christos Kozyrakis, and Matei Zaharia. 2022. VIVA: An End-to-End System for Interactive Video Analytics. In *CIDR*. Association for Computing Machinery.
[23] Chris Dongjoo Kim, Byeongchang Kim, Hyunmin Lee, and Gunhee Kim. [n. d.]. AudioCaps: Generating Captions for Audios in The Wild.
[24] Vladik Kreinovich. 2013. How to define relative approximation error of an interval estimate: A proposal. *Applied Mathematical Sciences* 7, 5-8 (2013), 211–216. https://doi.org/10.12988/ams.2013.13019
[25] Sanjay Krishnan, Adam Dziedzic, and Aaron J. Elmore. 2019. DeepLens: Towards a visual data management system. *CIDR 2019 - 9th Biennial Conference on Innovative Data Systems Research* (2019). arXiv:1812.07607
[26] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703
[27] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. 2021. Combining Aggregation and Sampling (Nearly) Optimally for Approximate Query Processing. In *SIGMOD*. 1129–1141. https://doi.org/10.1145/3448016.3457277 arXiv:2103.15994
[28] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, Dekang Lin, Yuji Matsumoto, and Rada Mihalcea (Eds.). The Association for Computer Linguistics, 142–150. https://aclanthology.org/P11-1015/
[29] Xinhao Mei, Xubo Liu, Jianyuan Sun, Mark D. Plumbley, and Wenwu Wang. 2022. On Metric Learning for Audio-Text Cross-Modal Retrieval. (2022). arXiv:2203.15537 http://arxiv.org/abs/2203.15537
[30] MindsDB. 2023. MindsDB. https://mindsdb.com.
[31] Oscar Moll, Manuel Favela, Samuel Madden, and Vijay Gadepally. 2022. SeeSaw: interactive ad-hoc search over image databases. In *ArXiV*. Association for Computing Machinery. arXiv:2208.06497 http://arxiv.org/abs/2208.06497
[32] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5 (1995), 25–42. https://doi.org/10.1007/BF00140664
[33] Gregory Piatetsky-Shapiro and Charles Connell. 1984. Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD*. 256–276. https://doi.org/10.1145/602259.602294
[34] Navneet Potti and Jignesh M. Patel. 2015. DAQ: A new paradigm for approximate query processing. *VLDB* 8, 9 (2015), 898–909. https://doi.org/10.14778/2777598.2777599
[35] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.
[36] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. (2021). arXiv:2103.00020 http://arxiv.org/abs/2103.00020
[37] Nils Reimers and Iryna Gurevych. 2020. Sentence-BERT: Sentence embeddings using siamese BERT-networks. *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference* (2020), 3982–3992. https://doi.org/10.18653/v1/d19-1410 arXiv:1908.10084
[38] Tal Ridnik, Emanuel Ben Baruch, Nadav Zamir, Asaf Noy, Itamar Friedman, Matan Protter, and Lihi Zelnik-Manor. 2021. Asymmetric Loss For Multi-Label Classification. In *2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021*. IEEE, 82–91. https://doi.org/10.1109/ICCV48922.2021.00015
[39] PG G Selinger, MM M Astrahan, D D Chamberlin, R A Lorie, and T G Price. 1979. Access path selection in a relational database management system. In *SIGMOD*. 23–34. http://dl.acm.org/citation.cfm?id=582095.582099
[40] Yi Wang, Yang Yang, Weiguo Zhu, Yi Wu, Xu Yan, Yongfeng Liu, Yu Wang, Liang Xie, Ziyao Gao, Wenjing Zhu, et al. 2020. SQLflow: a bridge between SQL and machine learning. *arXiv preprint arXiv:2001.06846* (2020).

Received October 2023; revised January 2024; accepted March 2024