ThalamusDB: Semantic Approximate Query Processing

Immanuel Trummer
Cornell University
itrummer@cornell.edu

Abstract

ThalamusDB performs semantic query processing on multimodal data. It supports SQL queries integrating semantic operators that are evaluated via large language models (LLMs). Such operators may, for instance, filter collections of images in the database, based on filter conditions described in natural language. Query evaluation costs are primarily due to LLM invocations. ThalamusDB is optimized to keep LLM costs low, minimizing the number of tokens when processing semantic operators. In particular, it enables users to lower processing overheads in scenarios where approximate results are acceptable. Users can set bounds on processing time and costs, or bounds on the approximation error of the result. ThalamusDB processes data subsets via LLMs to comply with user-defined bounds.

This paper describes the newest version of ThalamusDB in detail, publicly available on GitHub at `https://github.com/itrummer/thalamusdb`, and gives an outlook on research challenges and ongoing work.

1 Introduction

Traditional database management systems are limited to processing tabular data using operations that require only shallow, i.e., syntactic understanding of data. Efforts to expand that scope date back decades, leveraging, for instance, human crowd workers to enable SQL queries on multimodal data that perform operations requiring a deep understanding of data semantics [7, 18, 20]. However, until quite recently, such approaches were inherently limited in scalability due to the reliance on human workers. With the dramatic advances in large language models (LLMs) over the past couple of years, processing novel tasks, merely based on a natural language instructions ("zero-shot learning" [3]), on various types of data has finally become possible in a scalable manner. Several recent data processing systems [5, 6, 6, 8, 9, 14, 15, 17, 21–23, 27], in industry as well as in academia, now support semantic query processing, interleaving traditional relational operators with calls to LLMs during query processing. This combination gives semantic query processing engines the ability to handle a much larger set of queries compared to traditional SQL engines.

Example 1: The following example query is supported by Google's BigQuery system, as well as by several other systems in industry and in academia. Imagine a table `Cars` containing images in the `pic` column (for instance, this table could have been extracted from car ads on platforms such as Craigslist). Using SQL extensions available in BigQuery and other systems, we can formulate arbitrary filter conditions in natural language on various types of data. In this case, we can count the number of red cars in our database using the `AI.IF` operator with the following query:

```
SELECT COUNT(*)
FROM Cars
WHERE AI.IF(pic, 'this is a red car');
```

This paper describes the current state as well as future research directions of ThalamusDB, an ongoing project at Cornell University. An open-source version of the system is publicly available[1].

ThalamusDB is a semantic query processing engine that supports SQL with semantic operators. Similar to Example 1, semantic operators can be used to filter data, based on natural language instructions, and perform semantic joins, meaning to find pairs of rows that satisfy join conditions (described in natural language as well). Semantic operators are generally evaluated by invoking state-of-the-art LLMs from providers such as OpenAI or Anthropic. ThalamusDB supports an extended relational model. Table columns may either be associated with one of the traditional SQL data types or refer to files on disk containing unstructured data. Currently, ThalamusDB supports references to images, text, and audio data.

Semantic operators are evaluated on columns containing references to files. ThalamusDB automatically selects the best-suited model based on the data types to process. For instance, ThalamusDB may use highly efficient text-only models (e.g., the GPT-OSS 20B model) for processing text documents, while resorting to more generic and powerful models when the data to analyze includes images (e.g., OpenAI's GPT-5 model). Typically, processing overheads for semantic queries are dominated by overheads due to LLM invocations. Compared to processing traditional relational operators, processing semantic operators using powerful LLMs is typically more expensive by orders of magnitude (for input of a fixed byte size). This motivates an approach that focuses on minimizing LLM invocation overheads.

ThalamusDB is designed from the ground up for approximate, semantic query processing. Acknowledging that processing large data sets via LLMs is often prohibitively expensive, ThalamusDB aims at processing only data subsets to derive bounds on the shape of the query result. For aggregation queries (such as the one in Example 1), ThalamusDB derives lower and upper bounds on the values of each aggregate. Different from sampling-based approximation, ThalamusDB does not derive confidence bounds but deterministic bounds on the query result. This means that the result obtained after processing all remaining data using LLMs must fall within the bounds returned by ThalamusDB. Similarly, for non-aggregate queries, ThalamusDB identifies result rows that must appear in the query result (independently of the results of outstanding evaluations) as well as row that may be part of the final query result (but are not guaranteed to be included).

Internally, ThalamusDB processes queries using an iterative approach. In each iteration, semantic operators are applied to small data batches (i.e., LLMs are invoked on the corresponding data). After each iteration, ThalamusDB reasons about possible query results that are consistent with the rows processed using semantic operators so far. This requires reasoning about different possible outcomes for each semantic operator for the remaining rows. By merging all possible results into a concise representation (i.e., bounds for aggregation queries, and guaranteed as well as possible result rows for non-aggregation queries), ThalamusDB calculates error metrics, determining how far the current result may deviate from the final result. Users can set error bounds, prompting ThalamusDB to terminate iterations once the error falls below the user-defined threshold. Alternatively, users may specify bounds on computation cost metrics such as execution time, the number of tokens consumed, or the number of LLM invocations. If present, ThalamusDB exploits such bounds to terminate processing once any of the cost metrics exceeds the threshold.

The remainder of this paper is organized as follows. Section 2 introduces the problem model as well as related terminology. Section 3 gives an overview of the ThalamusDB system and its primary components. Section 4 describes the execution engine, the core of ThalamusDB, in detail. Section 5 describes the implementation of semantic operators, used during query execution. Section 6 discusses future and ongoing work on the ThalamusDB project. Finally, Section 7 distinguishes ThalamusDB from prior work.

[1] `https://github.com/itrummer/thalamusdb`

2 Problem Model and Terminology

ThalamusDB supports an extended, relational data model, defined next.

Definition 1 (Multimodal Database) *A multimodal database is described as a tuple $\langle R, F \rangle$ where F is a collection of files, including images, sound files, or text documents, and R is a relational database. In addition to the standard SQL column types, R may contain tables with columns of file type, each cell referencing one file in F.*

Note that this definition explicitly leaves open the option of referencing files of different types (e.g., images and audio files) in different cells in the same column. This is supported by ThalamusDB. The system processes semantic operators, evaluated via calls to LLMs, on unstructured data columns. The two semantic operators supported by ThalamusDB are defined next.

Definition 2 (Semantic Filter) *A semantic filter operation is characterized by a tuple $\langle I, U \rangle$ where I are filter instructions, formulated in natural language, and referring to an unstructured column U in a multimodal database (the specification includes both column and table name). The semantic filter is evaluated via calls to an LLM and returns all rows satisfying the filter condition.*

Definition 3 (Semantic Join) *A semantic join operation is characterized by a tuple $\langle I, U_1, U_2 \rangle$ where I describes the join condition in natural language, referencing unstructured columns U_1 and U_2, appearing in two different tables of a multimodal database. The semantic join is evaluated via calls to an LLM and returns pairs of rows from both input tables that together satisfy the join condition.*

ThalamusDB supports semantic queries, defined next.

Definition 4 (Semantic Query) *A semantic query refers to a multimodal database. It contains at least one semantic operator (i.e., semantic filter or semantic join) that is evaluated via invocations to an LLM.*

ThalamusDB is an approximate processing engine for semantic queries. During query evaluation, it regularly reasons about possible results, defined next.

Definition 5 (Possible Result) *Given a semantic query Q containing semantic operators O, denote by $R(Q, O, S)$ the result obtained for the query if $S(o, d)$ denotes the result obtained when applying semantic operator $o \in O$ to data item d. As long as query evaluation is ongoing, results have been obtained for a subset E of data items and operators. Denote by $S_P(o, d)$ the partially defined function mapping evaluated combinations in E to the associated result. Query result R_P is possible, iff S_P can be expanded into a complete function $S_F(o, d)$ such that $\forall \langle o, d \rangle \in E : S_F(o, d) = S_P(o, d)$ and $R_P = R(Q, O, S_F)$.*

ThalamusDB merges possible results into a concise representation that depends on the query type. ThalamusDB regularly shows merged results to users.

Definition 6 (Merged Result) *Denote by R a set of possible results for a query Q (after evaluating operators on a subset of the data). If Q is an aggregation query (without grouping), all possible results are merged into lower and upper bounds $\langle l_a, u_a \rangle$ for each query aggregate a with $l_a \leq u_a$ such that all possible results contain values for a between those bounds. If Q is not an aggregation query, the merged result is the intersection of result rows between all possible results.*

Note that ThalamusDB does not currently support approximation for queries with group-by clauses and queries with order-by clauses. Based on merged results, ThalamusDB calculates an approximation error, quantifying how far the current result is from the final result.

Definition 7 (Approximation Error) *For aggregation queries, denote by Δ the average relative distance between lower and upper bounds, averaging over all query aggregates. The approximation error is given as $\Delta - 1$ (reaching its minimum of zero if lower and upper bounds collapse). For non-aggregate queries, denote by Max the number of result rows in the largest possible result, and by Min the number of result rows in the intersection. The approximation error is given as $Max/Min - 1$, reaching its minimum of zero once all possible results are equal.*

Note that the approximation error does not account for mistakes made by the LLM, such as missclassification or halluzination. Based on the approximation error and several cost metrics, users can define one or multiple termination conditions for query processing via constraints.

Definition 8 (Constraints) *Users may specify constraints defining termination criteria for query evaluation. Constraints are defined as a vector C where different components represent upper bounds on cost metrics, namely the number of LLM invocations, the number of tokens consumed, and evaluation time (in seconds). Evaluation terminates once any of the associated metrics reaches the upper bound. Also, constraints include a lower bound on approximation error. Evaluation stops whenever the approximation error falls below that threshold.*

By default, all cost-related metrics are initialized to a large number, whereas the error bound defaults to zero (meaning that an exact result has been calculated). To ensure that ThalamusDB can satisfy the user's constraints, users may only specify constraints on error or on cost metrics (but not on both).

Definition 9 (Approximate Semantic Query Processing) *The input to approximate semantic query processing is a tuple $\langle R, F, Q, C\rangle$ where $\langle R, F\rangle$ describes a multimodal database, Q is a semantic query referencing that database, and C describes a termination condition. The result is a merged approximate result for Q satisfying error constraints, if any, while keeping processing overheads below the user-defined constraints.*

ThalamusDB is a system for approximate semantic query processing.

3 System Overview

Figure 1 shows an overview of the ThalamusDB system. ThalamusDB operates on multimodal databases, composed of files (including images, audio files, and text data) and relational tables that may reference files in their cells. Users submit semantic SQL queries that can contain calls to semantic operators. Along with queries, users submit constraints limiting computational overheads (such as run time and the number of tokens consumed by semantic operators). ThalamusDB generates a query result approximation under these constraints and returns it to the user.

ThalamusDB is based on an existing, relational database management system. The current implementation uses DuckDB. However, the queries issued by ThalamusDB during semantic query processing do not use any DuckDB-specific features. In principle, various relational database engines can be used. Incoming queries are first processed by the Query Parser. Pure relational queries, not containing any semantic operators, are directly forwarded and processed by the underlying database engine. Queries containing calls to semantic operators are processed by the ThalamusDB execution engine.

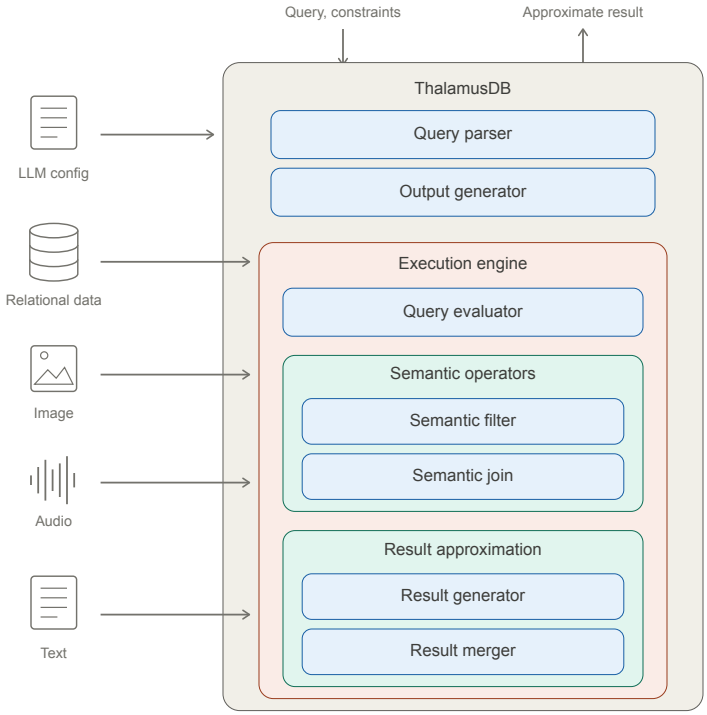

Figure 1: Overview of ThalamusDB system.

The Query Evaluator controls the iterative execution process. In each iteration, it processes semantic operators on data batches. Iterations stop once one of the user-defined constraints is violated. For instance, this happens once computational overheads for query execution reach a user-defined threshold, or when the quality of the result approximation satisfies user-defined constraints. ThalamusDB currently supports two semantic operators, namely unary semantic filters and a semantic join operator.

Semantic operators are evaluated using LLMs from providers such as OpenAI or Anthropic. ThalamusDB supports a variety of LLM providers, requiring users to provide provider-specific API access keys (which have to be defined in environment variables). When evaluating semantic operators, ThalamusDB dynamically selects the LLM best suited to the task based on a configuration file. The configuration file maps sets of data types (image, audio, and text) to specific LLMs with a priority value. A semantic operator may process multiple data types (e.g., a semantic join between rows containing image data and rows containing text). ThalamusDB identifies the set of relevant data types and searches the configuration files for LLMs supporting all of these data types. Among all matching LLMs, it selects one with maximal priority value.

ThalamusDB approximates query results during query evaluation by reasoning over possible query results, given the partial results already obtained for semantic operators. The result generator evaluates semantic queries under assumptions on the outcome of outstanding LLM calls. By systematically trying different value combinations for the outcomes of outstanding LLM calls, ThalamusDB obtains insights into the range of possible results. ThalamusDB merges possible results into a concise representation of result options. This merged representation is shown to users and can be used to calculate the approximation error during query evaluation. Once the approximation error falls below a user-defined threshold, query evaluation stops.

Algorithm 1 Evaluating semantic queries under user-defined cost and quality constraints.

```
1: // Evaluates semantic query Q while respecting cost and quality constraints C.
2: function EvaluateQuery(Q, C)
3:     // Identify semantic operators used in the query
4:     O ←SemanticOperators(Q)
5:     // Extract relevant rows for each operator
6:     for o ∈ O do
7:         o.prepare()
8:     end for
9:     // Initialize merged query result
10:    M ← null
11:    // Iterate until reaching cost or quality limits
12:    while ¬Terminate(O, M, C) do
13:        // Process row batch via LLMs for each operator
14:        for o ∈ O do
15:            o.processBatch()
16:        end for
17:        // Generate results under different assumptions on outcomes of outstanding LLM calls
18:        R ←GenerateResults(Q, O)
19:        // Merge possible results to infer common traits
20:        M ←MergeResults(Q, R)
21:    end while
22:    // Return approximate result
23:    return M
24: end function
```

4 Query Evaluation

Algorithm 1 describes the query evaluation process in ThalamusDB. The input is a semantic query, Q, together with user-defined constraints C. Users may set bounds on computation cost metrics such as computation time or computation cost (calculated as the cost for LLM calls, the dominant cost factor in semantic query processing), or, alternatively, specify constraints on result quality. Query processing terminates once the query result meets quality constraints or once evaluation overheads exceed at least one of the user-specified thresholds.

First, ThalamusDB extracts all semantic operators that appear in the input query, and prepares each operator for evaluation. This entails extracting rows on which the semantic operator has to be applied. When doing so, ThalamusDB automatically applies cheap, relational filter predicates to reduce the number of rows that have to be sent to the LLM during the following processing stages. Next, the system iterates until Function Terminate returns True. This function verifies whether any of the user-defined cost or quality thresholds have been reached. It takes as input the set of objects (O) representing semantic operators (each operator object stores internal counters capturing, for instance, the number of LLM calls and tokens used so far) and the current merged query result (M). The merged result is used to evaluate whether lower bounds on result quality have been satisfied. The operator counters are used to determine whether any of the user-defined cost limits for evaluating the current query have been reached.

In each iteration, ThalamusDB applies each semantic operator on a limited number of rows (a batch). To process rows, ThalamusDB instantiates operator-specific prompt templates and substitutes template

placeholders with data from the target rows. For each operator, we can classify rows into rows for which LLM results are available and rows for which the operator has not been processed yet. Currently, ThalamusDB supports unary semantic filter and semantic joins. Hence, the results of semantic operators are generally Boolean, assigning either single rows or row pairs to a Boolean value (representing whether or not conditions on rows or row pairs are satisfied).

ThalamusDB evaluates semantic queries under different assumptions on the results of outstanding LLM calls. I.e., it evaluates the query under different assumptions on whether or not rows or row pairs satisfy semantic filters and joins, as long as the actual result is unavailable. Different assumptions lead to different query results. ThalamusDB produces multiple alternative results, all consistent with the results of LLM evaluations received so far. By comparing those alternative results, ThalamusDB gains insights into result properties that hold independently of the results for outstanding LLM invocations.

ThalamusDB merges alternative results into an approximate result presentation, describing the range of possible query results precisely. For aggregation queries, ThalamusDB keeps track of lower and upper bounds for all query aggregates over possible results. For non-aggregate queries, ThalamusDB keeps track of result rows that appear in all possible results (meaning that they are certainly part of the query result), as well as rows that appear only in some possible results (meaning that they may or may not appear in the final query result). This merged result representation is used to evaluate whether constraints on output quality (limiting, for instance, the relative distance between lower and upper bounds of query aggregates) are met. Finally, after the iterations terminate, this merged result is returned and displayed to the user. The example presented next illustrates query evaluation in ThalamusDB.

Example 2: Consider the following query, counting ads for houses with a pool, as indicated by the picture or the associated text, in a certain area:

```
SELECT COUNT(*)
FROM Houses
WHERE Region = 5 AND
(NLFILTER(pic, 'the picture shows a pool') OR
NLFILTER(description, 'the text mentions a pool'))
```

ThalamusDB will immediately evaluate classical (non-semantic) predicates, in this case, the region predicate. Assume that, after a few iterations, ThalamusDB has obtained the following results for the semantic predicates, considering only houses in Region 5:

House ID	Picture Shows Pool	Text Mentions Pool
1	✓	✗
2	✓	✓
3	✗	✗
4	✗	✗
5	✓	?
6	✗	?
7	?	?
8	?	?

Here, ✓ indicates a predicate that evaluated to True for a specific house, ✗ a predicate that evaluated to False, and ? a predicate that has not been evaluated for a specific house yet. Houses 1 and 2 definitely have a pool, whereas Houses 3 and 4 definitely do not have a pool. House 5 has a pool, whereas Houses

6 to 8 may have a pool, depending on the outcome of the outstanding predicate evaluations. Hence, ThalamusDB calculates a range between 3 and 6 for the count aggregate. Assuming that the user has specified a relative error of factor two or larger, referring to the relative distance between upper and lower bounds, query evaluation ends with that approximate result.

5 Semantic Operators

Each semantic operator in ThalamusDB is associated with a prompt template. The prompt template contains placeholders for the instructions, configuring operator behavior, and submitted by users as part of their semantic queries. In addition, the prompt template contains placeholders for data on which the operator is applied. The execution engine instantiates prompt templates by substituting placeholders with user instructions, as well as with the target data. User instructions remain constant over the execution of a query (for a specific operator instance), whereas data changes over different invocations.

```
Find indexes x,y where x is the number of an entry
in collection 1 and y the number of an entry in
collection 2 such that [j] (make sure to catch
all pairs!)!
Separate index pairs by semicolons.
Write "Finished" after the last pair!
Text Collection 1:
1. [B1[1]]
2. [B1[2]]
...
Text Collection 2:
1. [B2[1]]
2. [B2[2]]
...
Index pairs:
```

Figure 2: Prompt template used for block nested loops join in ThalamusDB.

Besides user-provided instructions and data, each prompt template contains a precise description of the desired output format. This is required to automatically parse the output generated by the LLM. In rare cases, the output generated by the LLM does not comply with the instructions on output format, making it impossible to parse. If so, the execution engine assigns default values as the corresponding result (for filter conditions, it assigns a value of False by default).

For instance, Figure 2 shows the prompt template used by the join operator. Placeholder `[j]` represents the join condition. Users specify the join condition in natural language in the call to the semantic join operator, as part of their queries. For each prompt, ThalamusDB selects a batch of (unprocessed) input rows from both input tables. The prompt template contains placeholders `B1[i]` and `B2[i]` representing the i-th row from the first (`B1`) or second (`B2`) batch.

The prompt instructs the LLM to find pairs of matching rows, according to the user-defined join condition, among the input batches. Each matching row pair is represented as a pair of indexes (referring to index numbers assigned to rows in the prompt), separated by a comma. Index pairs are separated by semicolons. Note that each LLM invocation may produce multiple matching row pairs (as opposed

to a simpler algorithm, invoking the LLM to evaluate the join condition on specific row pairs). This approach comes with significant efficiency gains, compared to naive variants, as analyzed in more detail in a recent paper [26].

The unary (semantic) filter operator is handled similarly. The associated prompt template features placeholders for the unary filter condition, as well as for the target data. ThalamusDB uses multi-threading to evaluate operators on multiple rows in parallel. The number of rows evaluated in parallel is a tuning parameter. Evaluating fewer rows in parallel may sometimes reduce evaluation costs (since few rows may suffice to achieve an acceptable approximation error). On the other hand, evaluating more rows in parallel tends to speed up processing.

6 Future and Ongoing Work

The ongoing work on ThalamusDB focuses on two directions: expanding the set of semantic operators supported by the system, and optimizing data representation for efficient access and processing. The following two subsections discuss those research directions in detail.

6.1 Semantic Operators

Currently, ThalamusDB supports only two semantic operators (unary filters and joins) and only one single implementation for each. In the future, we plan to expand the set of supported operators, as well as add new implementations for joins and unary filters. Different implementations realize different tradeoffs between execution costs and output quality. Having a range of implementations to choose from enables the system to better cater to user preferences.

As in classical, relational query processing, the join operator tends to be the most expensive one (its complexity is quadratic in the input size). This makes it particularly worthwhile to create more implementations of the join operator. Whereas the current implementation is generic and works for arbitrary join conditions and input data formats, we plan to explore more specialized join operators that improve performance, compared to the generic operator implementation, for the scenarios they are applicable to.

For instance, equality joins are common in traditional data processing. In the context of semantic queries, equality joins aim at finding elements (e.g., pictures or text documents) that relate to each other. One of the most efficient classical join operators for equality joins is the hash join operator. It reduces the number of required pairwise comparisons by partitioning data from both input tables into buckets, based on the hash values of the join column. We plan to explore similar approaches for semantic join operators, partitioning data into buckets such that pairs satisfying the join condition are located within the same bucket with a high probability. This partitioning pass could be based on a combination of embedding vectors (assigning rows to the same partition if their embedding vectors are close) or other data properties (e.g., if the join condition correlates with values in other columns). Within each partition, the block-based join operator available in the current system can be used. Note that the number of partitions could become a tuning parameter, allowing the system to trade computational overheads for result completeness (choosing fewer partitions to increase the probability of uncovering all join pairs).

Beyond join operators, we plan to add support for other semantic versions of relational operators, including sort operators, distinct operators (retrieving unique items from a table containing duplicates), and an exists operator (efficiently checking for the existence of rows with certain properties).

6.2 Physical Design

The current implementation of ThalamusDB stores unstructured data files, such as images, test, or audio data, as-is, without changing their representation. Going forward, we will explore physical design optimization with the goal of making access more efficient or reducing storage overheads.

For instance, files can be associated with embedding vectors representing their semantics. Such embedding vectors can be used, for instance, to efficiently retrieve items that are likely to satisfy a certain filtering condition. Instead of relying on embedding vectors alone, they can also be used to prioritize data processing via LLMs. As ThalamusDB generally only processes a subset of data items, bounded by user-defined constraints on computation time and costs, the subset to process could be selected based on embedding vectors. For instance, for filter predicates, the similarity between the embedding of the filter condition and the embedding of row data could be used to prioritize rows for processing. Similarly, for joins, embedding vectors can be used to group items based on similarity, reducing the number of required comparisons.

Beyond embedding vectors, other types of meta-data could be interesting. For instance, associating images in the database with a text description would enable substituting some operators that refer to images with operators that refer to the text description. If the text description consumes fewer tokens compared to the associated image, this approach can reduce processing costs. Similarly, a text representation can be exploited to pre-filter images, reserving image processing to the images most likely to be relevant (based on the text description). Similar approaches can be used to associate audio data with a more token-efficient alternative representation (transcribing speech to text) or to associate long text documents with a short summary.

Adding embedding vectors or more alternative data representations consumes storage space and creates computational overhead. Both, embedding vectors and token-efficient, alternative data representations, must be generated by LLM invocations. If not used carefully, this approach risks increasing computational costs by creating data representations that are ultimately not useful for answering queries. We are exploring options to formalize physical design tuning for semantic query processing as a combinatorial optimization problem, estimating the benefit of additional data structures, based on representative workloads, and weighing them against the overheads of generating them.

7 Related Work

ThalamusDB relates to other research proposing systems that support queries mixing relational operators with calls to LLMs [5, 6, 6, 8, 9, 14, 15, 17, 21–23, 27], often referred to as semantic query processing. Prior systems can be categorized based on the query interface they support, including natural language interfaces [4, 16, 27], SQL variants [5, 6, 8, 9, 14, 23], or Python libraries introducing functions for semantic operators [15, 21]. Here, ThalamusDB belongs in the category of SQL-centric systems. Alternatively, semantic query processing engines can be categorized based on the design of the underlying execution engine. For instance, some prior work uses LLMs to generate code for implementing operators dynamically [27]. Instead, ThalamusDB uses a fixed set of relational operator implementations that exploit LLMs for implementing sub-functions (e.g., evaluating filter conditions).

Some prior systems for semantic query processing feature parameters, allowing users to trade computation overheads for result quality [17, 21]. However, the primary mechanism by which prior work trades computation overheads for result precision is selecting smaller models to implement semantic operators (resulting in cheaper query evaluation but, possibly, less accurate results). Instead, ThalamusDB trades costs for quality by reducing the number of rows processed via AI operators. This results in a stronger type of guarantee on the accuracy of the result (deterministic bounds as opposed to confidence bounds). Finally, ThalamusDB is based on an extended relational data model, extending column data types by

Table 1: Extract of experimental results on SemBench by Lao et al. [12].

System	Quality	Latency	Costs
LOTUS	0.755	367.7s	$1.54
Palimpzest	0.740	263.2s	$1.84
ThalamusDB	0.592	244.2s	$0.17
BigQuery	0.587	48.1s	$0.69

adding support for unstructured data such as images or audio files. In that, it connects to several of the recent semantic query processing engines [17, 21], whereas it differs from prior work supporting multimodal data processing over unstructured data lakes [4].

This paper relates most to prior papers introducing a now-outdated version of ThalamusDB [8, 9]. The ThalamusDB version described in this paper shares no code with the prior version and features a new execution model, query interface, and query scope. First, the new ThalamusDB version is designed from the ground up for the latest generation of LLMs. These LLMs are able to solve a variety of tasks via zero-shot learning [3] (i.e., based on a natural language description of a task alone, but without requiring task-specific training samples). Whereas the prior version of ThalamusDB [9] requests data labels from users, enabling the system to obtain more precise answers from LLMs, the new version relies entirely on operator configurations specified by users in natural language as part of the input query. Beyond the query interface, this change in focus impacts query processing and query optimization (which previously considered the number of labeling requests as a criterion). Second, the new ThalamusDB version features a new operator model. Whereas the prior model was specific to unary semantic filter operations, framing operator evaluation as a comparison between embedding vectors in general, the new version features a more general model, associating operators with operator-specific prompt templates that are instantiated and sent to LLMs for evaluation. Third, whereas the prior ThalamusDB version only supports a small set of SQL operators, implemented by a custom execution engine, the new version expands the scope of supported SQL features by moving more responsibilities to the underlying SQL execution engine.

A recent paper [12] provides detailed experimental results for ThalamusDB on the newly introduced SemBench benchmark. Table 1 shows average per-query values for monetary execution fees (due to LLM invocations), latency, and result quality (scaled to the interval $[0, 1]$ where 1 represents an accurate result) for ThalamusDB and other systems, restricting the scope to queries supported by ThalamusDB. Compared to other systems, ThalamusDB supports only a restrictive set of semantic operators (semantic join and filter). Its latency and result quality are significantly below the optimum. However, for the supported queries, ThalamusDB achieves optimal processing costs.

Beyond work introducing novel systems for semantic query processing, this work relates to recent research proposing efficient implementations for specific semantic operators, such as joins [26], sort operators [29], or filters [5]. New operator implementations can be integrated with the existing system to make processing specific types of queries more efficient. Recent work explores the idea of extracting structured from unstructured data [2], providing an alternative to semantic query processing. However, this approach is limited to text data and does not work for ad-hoc queries in which user queries focus on data properties not considered during the extraction step.

More broadly, this research connects to all prior work in the database community, aimed at exploiting LLMs for tasks related to data management. This includes, for instance, prior work aimed at using LLMs for database tuning [11, 25, 28], generating code for data wrangling and processing [19, 24], or different variants of text-to-SQL translation [1, 10, 13].

Beyond prior research exploiting LLMs, this work relates to early work implementing semantic query processing based on human crowd workers. Corresponding systems, including CrowdDB [7], Deco [20], and Qurk [18], enable users to write queries including natural language snippets that are then evaluated by crowd workers. While the underlying technology is different (and not scalable), the query model implemented by these systems is close to the ones supported by current semantic query processing engines.

8 Conclusion

LLMs are dramatically expanding the scope of relational query processing. A new generation of database systems, semantic query processing engines, support operators that exploit a deep understanding of data semantics. They expand the scope from purely relational to various other types of data, including images, audio data, and text documents. While significantly broadening the scope in terms of queries, semantic query processing engines create new challenges due to the significant costs of LLM invocations.

ThalamusDB addresses this challenge by adopting an approximate processing framework, centered on reducing cost overheads for LLM invocations. It gives users fine-grained controls, allowing them to trade computational overheads for result quality. This paper described the design and implementation of the newest version of ThalamusDB, evaluated experimentally in a recent paper [12], as well as future research.

Acknowledgment

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

References

[1] A. Abouzied, F. Alam, R. Ali, and P. Papotti. Combating misinformation in the arab world: Challenges and opportunities. *Communications of the ACM*, 68:48–53, 10 2025.

[2] S. Arora, B. Yang, S. Eyuboglu, A. Narayan, A. Hojel, I. Trummer, and C. Re. Language models enable simple systems for generating structured views of heterogeneous data lakes. *PVLDB*, 17:92 – 105, 2023.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Z. Chen, Z. Gu, L. Cao, J. Fan, S. Madden, and N. Tang. Symphony: Towards natural language query answering over multi-modal data lakes. In *CIDR*, pages 1–7, 2023.

[5] Y. Chung, R. Desai, J. He, Y. Xiao, T. Hottelier, Y.-L. K. Samo, P. Kadilkar, X. Chen, S. Idicula, F. Özcan, et al. 100x cost & latency reduction: Performance analysis of ai query approximation using lightweight proxy models. *arXiv preprint arXiv:2603.15970*, 2026.

[6] A. Dorbani, S. Yasser, J. Lin, and A. Mhedhbi. Beyond quacking: Deep integration of language models and rag into duckdb. *Proceedings of the VLDB Endowment*, 18:5415–5418, 8 2025.

[7] M. Franklin and D. Kossmann. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[8] S. Jo and I. Trummer. Demonstration of thalamusdb: Answering complex sql queries with natural language predicates on multi-modal data. In *SIGMOD*, pages 179–182, 2023.

[9] S. Jo and I. Trummer. Thalamusdb: Approximate query processing on multi-modal data. *SIGMOD*, 2:1–26, 2024.

[10] G. Karagiannis, M. Saeed, P. Papotti, and I. Trummer. Scrutinizer: A mixed-initiative approach to large-scale, data-driven claim verification. *PVLDB*, 13:2508–2521, 2020.

[11] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang. Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization. *arXiv preprint arXiv:2311.03157*, 2023.

[12] J. Lao, A. Zimmerer, O. Ovcharenko, T. Cong, M. Russo, G. Vitagliano, M. Cochez, F. Özcan, G. Gupta, T. Hottelier, et al. Sembench: A benchmark for semantic query processing engines. *arXiv preprint arXiv:2511.01716*, 2025.

[13] F. Li and H. Jagadish. Nalir: an interactive natural language interface for querying relational databases. In *SIGMOD*, pages 709–712, 2014.

[14] P. Liskowski, B. Han, P. Aggarwal, B. Chen, B. Jiang, N. Jindal, Z. Li, A. Lin, K. Schmaus, J. Tayade, et al. Cortex aisql: A production sql engine for unstructured data. *arXiv preprint arXiv:2511.07663*, 2025.

[15] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, R. Shahout, and G. Vitagliano. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*, 2025.

[16] C. Liu, G. Vitagliano, B. Rose, M. Printz, D. A. Samson, and M. Cafarella. Palimpchat: Declarative and interactive ai analytics. In *Companion of the 2025 International Conference on Management of Data*, pages 183–186, 2025.

[17] S. Madden, M. Cafarella, M. Franklin, and T. Kraska. Databases Unbound: Querying All of the World's Bytes with AI. *PVLDB*, 17(12):4564–4554, 2024.

[18] A. Marcus, E. Wu, D. R. Karger, S. Madden, R. C. Miller, S. Acm, and N. York. Demonstration of qurk : A query processor for human operators. In *SIGMOD*, pages 1315–1318, 2011.

[19] A. Narayan, I. Chami, L. Orr, and C. Ré. Can foundation models wrangle your data? *PVLDB*, 16:738–746, 2022.

[20] A. G. Parameswaran, H. Park, H. Garcia-Molina, J. Widom, and N. Polyzotis. Deco: Declarative crowdsourcing. In *Information and Knowledge Management*, pages 1203–1212, 2012.

[21] L. Patel, S. Jha, M. Pan, H. Gupta, P. Asawa, C. Guestrin, and M. Zaharia. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. In *Proceedings of the VLDB Endowment*, volume 18, pages 4171–4184, 2025.

[22] G. Sanmartino, M. Urban, P. Papotti, and C. Binnig. The stretto execution engine for llm-augmented data systems, 2026.

[23] D. Satriani, E. Veltri, D. Santoro, S. Rosato, S. Varriale, and P. Papotti. Logical and physical optimizations for sql query execution over large language models. *Proceedings of the ACM on Management of Data*, 3:1–28, 6 2025.

[24] I. Trummer. Codexdb: Synthesizing code for query processing from natural language instructions using gpt-3 codex. *PVLDB*, 15:2921 – 2928, 2022.

[25] I. Trummer. Db-bert: a database tuning tool that "reads the manual". In *SIGMOD*, pages 190–203, 2022.

[26] I. Trummer. Optimal block nested loops implementations for semantic joins. *Data Engineering Bulletin*, 2026.

[27] M. Urban and C. Binnig. CAESURA: Language Models as Multi-Modal Query Planners. In *CIDR*, 2024.

[28] W. Zhang, W. S. Lim, and A. Pavlo. This is going to sound crazy, but what if we used large language models to boost automatic database tuning algorithms by leveraging prior history? we will find better configurations more quickly than retraining from scratch! *Proceedings of the ACM on Management of Data*, 4:1–29, 4 2026.

[29] F. Zhao, J. Chen, Y. Pan, T. Rabbani, D. Agrawal, A. E. Abbadi, P. Aggarwal, A. Datta, D. Tsirogiannis, et al. Access paths for efficient ordering with large language models. *arXiv preprint arXiv:2509.00303*, 2025.