Towards AI-Native Data Systems with the Semantic Operator Model and LOTUS

Liana Patel, Carlos Guestrin, Matei Zaharia

Abstract

The semantic capabilities of large language models (LLMs) hold transformative potential for data systems, enabling new language-based analytics over vast knowledge corpora. While LLM-based operations are promising, their integration within traditional data systems is inherently challenging both due to the ambiguity of language-based processing, which makes reasoning about correctness difficult, and the computational expense of scaling LLMs over large datasets. This paper describes semantic operators, a model we have been developing to provide the first formalism for general-purpose AI-based operations with natural language parameters (e.g., filtering, sorting, joining or aggregating records using natural language criteria). Each operator can be implemented by multiple algorithms, which compose individual LLM invocations within a pattern that specifies how to orchestrate the model over the dataset (e.g., a semantic join might be implemented by a nested-loop join algorithm). The semantic operator model defines the expected behavior of each operator with a high-quality reference algorithm, providing a correctness definition for arbitrary LLM-based transformations that go beyond simple batched inference primitives and apply even in the absence of user-labeled data. Our model further provides an optimization framework that reduces execution cost of each operator while providing statistical accuracy guarantees to ensure correct executions. This formalism opens up a rich research and design space towards accurate and efficient LLM-based data processing. In this work we provide an overview of the semantic operator model and extensions to it. Based on our experience deploying semantic operators in the open-source LOTUS system, we also discuss diverse user applications, including agent trace analysis, semantic clustering and deep research systems. Overall, we have been excited to see adoption of the semantic operator model in industry as well as an emerging sub-field of research and believe these early milestones represent steps towards AI-native data systems which will seamlessly integrate semantic reasoning capabilities for rich processing over structured and unstructured data.

1 Introduction

The powerful semantic capabilities of modern large language models (LLMs) have created exciting opportunities for building AI-based analytics systems that reason over vast knowledge corpora. Many applications require complex reasoning over vast knowledge corpora. For example, a researcher reviewing recent ArXiv [1] preprints may want to quickly obtain a summary of relevant papers from the past week, or find the papers that report the best performance for a particular task and dataset. Similarly, a medical professional may automatically extract biomedical characteristics and candidate diagnoses from many patient reports [2]. Likewise, organizations wish to automatically digest lengthy transcripts from internal meetings and chat histories to validate hypotheses about their business [3].

Each of these tasks requires a form of *bulk semantic processing*, where the analytics system must process large amounts of data and orchestrate models in complex patterns across a whole dataset. Supporting the full generality of these applications with efficient, easy-to-use analytics systems would have a transformative impact, similar to what RDBMSes had for tabular data. This prospect, however,

raises two challenging questions: first, *how should developers express semantic queries*, and secondly, *how should we design the underlying data system to achieve high efficiency and accuracy.*

Unfortunately, existing systems are insufficient for bulk semantic processing, either limiting their expressiveness to simple batched inference primitives or providing no accuracy guarantees. First, several systems only support simple batched inference primitives [4–13]. These systems do not support richer LLM-based operations, such as ranking, grouping or joining records, which require more complex *AI algorithms* that compose multiple model invocations to orchestrate the model over the data. Alternatively, more recent LLM-based analytics systems [14, 15] study more complex AI operations, but empirically optimize these operations with *no accuracy guarantees.* These systems lack a formalism to define correct behavior, which hinders their robustness and usability. These obstacles highlight a core challenge of integrating semantic-based processing within reliable query systems due to the inherent ambiguity of natural language instructions and LLM outputs.

Our work proposes *semantic operators*, a model we have been developing, which extends the relational model with AI-based operations. Semantic operators provide the first formalism *with statistical accuracy guarantees* for general-purpose AI-based operations with natural language parameters, including semantic filters, joins, top-k rankings, aggregations, and projections. Each operator takes a concise natural language signature, given by the programmer, and its behavior is fully specified by a tractable, high-quality reference algorithm. Our optimization approach then exploits the rich design space of semantic operator execution plans to reduce cost, while providing *statistical accuracy guarantees* for individual operators with respect to the reference algorithm (i.e., ensuring that the output of the optimized operator will be similar to that of the reference algorithm). We have implemented semantic operators in LOTUS (**L**LMs **O**ver **T**ext, **U**nstructured and **S**tructured data), an open source system that exposes these operators in a simple DataFrame-based API.

In this work we provide an overview of the semantic operator model, extensions of the model, and recent applications based on our experience deploying semantic operators in LOTUS. Specifically, we discuss how the semantic operator model, which originally defines the correctness of *individual* semantic operators in the absence of user-labeled data extends to two additional settings: multi-operator queries, and settings with user-labeled data. We discuss optimizations for both of these as well as our recent extensions in LOTUS, including the addition of a lazy API to optimize semantic operator queries for these settings. LOTUS' updated implementation includes logical plan optimizations, such as operator re-ordering, prompt optimization, for accuracy improvements with user-labeled data, and approximation algorithms, for cost-reduction using statistical techniques used in prior works [16, 17] with novel and efficient proxy scores. Lastly, we discuss emerging user applications of semantic operators among LOTUS users, highlighting 3 case studies involving agent trace analysis, semantic clustering [18] and deep research systems for generative research synthesis [19]. We demonstrate how queries for each of these applications are being implemented with LOTUS programs consisting of a few semantic operators. Overall, since open-sourcing LOTUS, we have been excited to see a growing user-base, adoption of semantic operators among large database vendors [20, 21], and a growing body of exciting research work. Ultimately, we believe semantic operators serve as a foundation for the future of AI-native data systems that will serve rich, semantic-based processing over vast knowledge corpora with unprecedented scale, accuracy and efficiency.

2 The Semantic Operator Model

The data independence model of relational systems [22] has had transformative impact on database systems by decoupling application logic from how data is stored and structured. The semantic operator model extends this concept of relational systems and introduces *model-data independence* for AI-based

```python
def paper_digest(research_interest: str, baseline: str):
  return papers_df\
    .sem_filter(f"the paper {{abstract}} claims to outperform {baseline}")\
    .sem_agg(f"Write a digest summary based on each {{abstract}} describing how these
    papers relate to {research_interest}")
```

Figure 1: Example LOTUS program using semantic operators to return a summary of relevant papers from the dataset papers_df. The user function, takes two strings, describing a research interest and a research baseline. The program filters papers in the dataset based on whether each paper claims to outperform the baseline. Using the filtered papers the program then constructs a summary.

data processing. Model-data independence decouples application logic from the underlying AI-based algorithm which specifies individual model invocations for processing each data record. The semantic operator model provides this form of independence by defining correct executions and optimizations for *arbitrary* LLM-based transformations over datasets, such as language-based joins, filters, top-k ranking and aggregations. This model provides a declarative data programming abstraction, where programmers write application logic with high-level operators, e.g., `sem_join`, `sem_filter`, `sem_topk`, `sem_agg`—as opposed to low-level LLM() calls—and systems can transparently optimize query execution. In this section, we begin by providing an example of a semantic operator program and will then provide an overview of the semantic operator model, including definitions of semantic operators and correct optimizations as well as common semantic operators.

2.1 Example Semantic Operator Program

To begin to understand the capabilities of semantic operators, we examine a simple program written with the LOTUS API, shown in Figure 1. The function `paper_digest` takes two string parameters, a user's research interest and a research baseline, and returns a digest of relevant research papers that claim to outperform the baseline. The dataset of research papers, `papers_df`, contains an *abstract* attribute. The `paper_digest` function processes this dataset using 2 semantic operators, a semantic filter and a semantic aggregation. The program first performs a semantic filter to find papers with abstracts claiming to outperform the baseline. The program then performs a semantic aggregation to summarize the filtered papers, returning a digest.

Semantic operators perform familiar transformations, such as filters and aggregations, reminiscent of relational operators. The crucial difference between semantic operators and their relational counterparts is that semantic operators are parameterized by *natural-language specifications*. For instance, the `sem_filter` operator takes the natural language predicate "the paper {abstracts} claim to outperform the baseline". The expected output of this operator is all records from the dataset with abstracts that pass the natural language predicate.

Due to each operator's language-based parameters, the behavior of each operator is inherently ambiguous without further specification and will depend on the specific algorithm used. For instance, one naive algorithm to implement a semantic filter might pass the entire dataset to a single LLM call, prompting the model to output all rows that pass the user's predicate. A simple but effective alternative might pass *each row* to a single LLM invocation, prompting the model to output a boolean assessing whether the user's predicate holds for that row. The latter option represents a high-accuracy implementation that will avoid issues like long-context degradation [23]. We can use this high-quality algorithm as a reference to define the semantic filter's behavior. More generally, each semantic operator has an expected behavior defined by a high-quality *reference algorithm*.

The actual execution of each semantic operator may deviate from the reference algorithm. For example, for the semantic filter, the execution engine might leverage various proxies, such as smaller

Table 1: Summary of Key Semantic Operators. T denotes a relation, X and Y denote arbitrary tuple types. l denotes a parameterized natural language expression ("langex" for short). Each operator may permit additional optional parameters, including accuracy targets.

Operator	**Description**	**Definition**	**Reference Algorithm**
$sem\_filter(l\colon X \to Bool)$	Returns the tuples that pass the langex predicate.	$\{t_i \mid t_i \in T \wedge l_M(t_i) = 1\}$	Compute $M(t_i, l), t_i \in T$
$sem\_join(t\colon T,\quad l\colon (X,Y) \to Bool)$	Joins a table against a second table t by keeping all tuple pairs that pass the langex predicate.	$\{(t_i, t_j) \mid l_M(t_i, t_j) = 1, t_i \in T_1, t_j \in T_2\}$	Compute $M(\{t_i, t_j\}, l), t_i \in T_1, t_j \in T_2$
$sem\_agg(l\colon T[X] \to X)$	Aggregates input tuples according to the langex reducer function.	$l_M(t_1, ..., t_n) \forall t_1, .., t_n \in T$	Perform a hierarchical reduce, recursively computing $acc_{i,r} \leftarrow M(\{acc_{n_i,r-1}, ..., acc_{n_i+n,r-1}\}, l)$
$sem\_topk(l\colon T[X] \to Seq[X],\quad k\colon int)$	Returns an ordered list of the k best tuples according to the langex ranking criteria.	$\langle t_1, ..., t_k \rangle$ st $\forall (t_i, t_j), i < j \implies l_M(t_i, t_j) = \langle t_i, t_j \rangle$	Perform quick-select top-k using pairwise comparisons, $M(\{t_i, t_j\}, l)$
$sem\_group\_by(l\colon X \to Y,\quad C\colon int)$	Groups the tuples into C categories based on the langex grouping criteria.	$\underset{\{\mu_1,...\mu_C\}, \mu_i \in V^{\mathbb{N}}}{\text{argmax}} \sum_{t_i \in T} \max_{j \in 1...C} l_M(t_i, \mu_j)$	Obtain centers $\mu_1, ...\mu_C$ with a clustering algorithm, and perform pointwise assignments $M(t_i, (l, \mu_1, ..., \mu_C)), t_i \in T$,
$sem\_map(l\colon X \to Y)$	Performs the projection specified by the langex.	$\{l_M(t_i) \mid t_i \in T\}$	Compute $M(t_i, l), t_i \in T$

models, embedding based search, keyword search or code synthesis to speed up execution and reserve LLM-based predicate evaluation only when needed. In doing so, the execution engine must still ensure that the resulting filter execution is faithful to the expected filter results, defined by the operator's reference algorithm. Using the semantic operator model, the data management system achieves model-data independence, allowing the programmer to compose their application logic with high-level dataset transformations while guaranteeing that the resulting execution will be close to the expected behavior defined by the model.

2.2 Defining Semantic Operators

Definition 1: *A semantic operator is a declarative transformation over one or more datasets, parameterized by a natural language expression. Each semantic operator can be implemented by potentially many AI-based algorithms, and its correct behavior is defined with respect to a given reference algorithm.*

Table 1 lists a core set of semantic operators, which cover common semantic transformations in real-world applications and mirror key transformations in relational operators. We have implemented these transformations in LOTUS, and many of them have been recently adopted among existing open-source LLM-based data processing systems [14, 24, 25] and commercial ones [6, 20]. Specific systems may, of course, provide additional semantic operators or utilities beyond the ones we discuss here.

Each semantic operator takes a *parameterized natural language expression* (*langex* for short), which are natural language expressions that specify a function over one or more attributes. As Figure 1 demonstrates, the langex signature varies for different semantic transformations. For instance, while the `sem_filter` langex signature provides a natural language *predicate*, the `sem_agg` langex is a commutative, associative *aggregator* expression, which here indicates a summarization task over abstracts.

We provide a high-level definition of each semantic transformation with respect to the user langex and a world model[1], M, which captures a probability distribution over the vocabulary V. For example, semantic filter returns $\{t_i | t_i \in T \wedge l_M(t_i) = 1\}$, where T is the input relation and $l_M(t_i)$ represents a natural language predicate evaluated on tuple t_i with model M. Notably, the definition of each semantic operator can be implemented by multiple AI-based algorithms, and different decisions as to how to invoke the model over the relation have consequences on the algorithm's result quality. Thus, the correct behavior of each semantic operator is specified by a *reference algorithm*, a computable and tractable AI algorithm that produces results considered to be high-quality. Each reference algorithm specifies a model access pattern over the relation T via an algorithm composed of model invocations $M(x, l)$, describing the subset of the data $x \subseteq T$, each model call is invoked over and the task-specific language expressions, given by l.

2.3 Defining Correct Optimizations for Semantic Operators

Semantic operators create a rich design space of diverse execution plans. While reference algorithms provide high-quality implementations, they are often expensive, with the complexity of LLM calls scaling linearly or quadratically in the dataset cardinality. As such, we define correct optimizations for individual semantic operators below by considering alternate AI-based algorithms that can reduce cost while offering *close results*. This model allows an execution system to optimize individual operators even in the absence of user-labeled data. Moreover, this model can be naturally extended to multi-operator queries and settings with user-labeled data, and we describe opportunities for these optimization regimes in Section 3. In general, optimized semantic operator plans allow for both lossless optimizations and approximations of a reference algorithm. This formulation builds on approximate query processing and assumes some error is often tolerable, which our work finds is often reasonable for achieving high-quality results for semantic processing, which is inherently non-exact.

Definition 2: *A correct optimization for a given semantic operator, and a reference algorithm for that operator, reduces cost while providing statistical accuracy guarantees with respect to the reference algorithm. Specifically, the optimization should ensure an accuracy target, γ, is met with probability $1 - \delta$.*

2.4 Core Semantic Operators

We now overview several core semantic operators, providing the operator's definition and a reference algorithm for each. We discuss design decisions for our reference algorithms based on state-of-the-art algorithms studied in the AI literature, well-known failure cases, such as long-context challenges [23], or our experimental observations. Our prior work [19, 26, 27] confirms that the reference algorithms we present support state-of-the-art result quality in real ML applications; however, finding optimal reference algorithms for each operator remains an open research question.

Semantic Filter is a unary operator over the relation T and returns the relation $\{t_i | t_i \in T \wedge l_M(t_i) = 1\}$, where the langex provides a natural language predicate over one or more attributes.

Reference Algorithm. Our reference algorithm runs batched LLM calls over all tuples in relation T. Each model invocation, $M(t_i, l)$, prompts the LLM with a single tuple $t_i \in T$, the langex predicate, and an operator-specific instruction to generate a boolean value. This simple choice avoids well-studied long-context issues [23] by processing rows independently rather than in a single invocation.

[1] In practice, the world model, M may be the strongest LLM a practitioner has available.

```
join_res = papers_df.sem_join(dataset_df, "The paper {abstract:left} uses the {
    dataset_name:right}.")
topk_res = papers_df.sem_topk("The paper has the funniest {title}", K=5)
grouped_res = papers_df.sem_group_by("What is the main research topic of the paper {
    abstract}", C=10)
```

Figure 2: Example usage of sem_join, sem_topk, and sem_groupby. papers_df contains fields for the "title" "abstract", and dataset_df, contains a field called "dataset_name".

Semantic Join provides a binary operator over relations T_1 and T_2 to return the relation $\{(t_i, t_j) | l_M(t_i, t_j) = 1, t_i \in T_1, t_j \in T_2\}$. Here the langex is parameterized by the left and right join keys and describes a natural language predicate over both.

Reference Algorithm. The reference algorithm implements a *nested-loop join pattern*, performing a single predicate evaluation for each pair of tuples, with model invocations $M(\{t_i, t_j\}, l), t_i \in T_1, t_j \in T_2$. This yields an $O(|T_1| \cdot |T_2|)$ LLM call complexity.

Semantic Top-k imposes a ranking[2] over the relation T and returns the ordered sequence, $\langle t_1, ..., t_k \rangle$ *st* $\forall (t_i, t_j), i < j \implies l_M(t_i, t_j) = \langle t_i, t_j \rangle$. Here, the langex signature is a general ranking criteria.

Reference Algorithm. Two important algorithmic design decisions for the semantic top-k include how to implement LLM-based comparison and how to aggregate ranking information from these comparisons. Our reference algorithm uses pairwise LLM comparisons for the former, and a quick-select top-k algorithm [28] for the latter. We briefly describe the reason for these choices and alternatives considered. Our decisions build on prior works, which have studied LLM-based passage re-ranking [29–37] with the goal of achieving high quality results in a modest complexity of LLM calls or comparisons.

First, pairwise-prompting methods offer a simple and high-quality approach that feeds a single pair of tuples to each LLM invocation, $M(\{t_i, t_j\}, l)$, prompting the model to compare the two inputs and output a binary label. The two main classes of alternatives are point-wise ranking methods [29–31, 36], and list-wise ranking methods [32–34, 37], both of which have been shown to face quality issues [29, 35, 37]. Our prior work verifies these limitations [27]. In contrast, pairwise comparisons have been shown to be effective and relatively robust to input ordering [35].

In addition, we consider several possible rank-aggregation algorithms, including quadratic sorting algorithms, a heap-based top-k algorithm and a quick-select-based top-k ranking algorithm. Our prior work [27] demonstrates that each of these sorting algorithms yield high-quality results, comparable to one another. However, the quick-select-based algorithm offers an efficient implementation with at least an order of magnitude fewer LLM calls than the quadratic sorting algorithm and more opportunities for efficient batched inference, leading to lower execution time, compared to a heap-based implementation. The quick-select top-k algorithm proceeds in successive rounds, each time choosing a pivot, and comparing all other remaining tuples to the pivot tuple to determine the rank of the pivot. Because each round is fully parallelizable, we can efficiently batch these LLM-based comparisons before recursing.

Semantic Aggregation performs a many-to-one reduce over the input relation, returning $l_M(t_1, ..., t_n)$, $\forall t_1, .., t_n \in T$. Here, the langex signature is a commutative, associative aggregation function[3], which can be applied over any subset of rows to produce an intermediate result. We note that the langex itself is model-agnostic, assuming infinite context. Managing finite context limits of the underlying model M is an implementation detail of the system.

Reference Algorithm. Our reference algorithm uses a hierarchical reduce pattern. Our choice builds on the LLM-based summarization pattern studied by prior research works [38], which we briefly overview. Prior works primarily study two aggregation patterns. First, a fold pattern performs a linear pass

[2]This definition implies that l_M imposes a total and consistent ordering. However, this definition can also be softened to assume partial orderings and noisy comparisons with respect to model M.

[3]We note that the ordering of inputs within an LLM prompt invocation can in fact affect results quality for some tasks. To allow programmers to override the commutativity and associativity, LOTUS exposes a partitioner function.

over the data, iteratively updating the accumulated partial answer with the next tuple t_i, given by $acc_i \leftarrow M(\{acc_{i-1}, t_i\}, l)$. Alternatively, the hierarchical reduce pattern recursively aggregates n inputs in each round r and produce multiple partial answers, given by $acc_{i,r} \leftarrow M(\{acc_{n_i,r-1}, ..., acc_{n_i+n,r-1}\}, l)$ until a single answer remains. Both represent candidate reference algorithms, however, the hierarchical pattern has been shown to produce higher quality results for commutative, associative aggregation tasks, like summarization, in prior work [38] and allows for greater parallelism during query processing, making it our default choice.

Semantic Group-by takes a langex that specifies a projection from a tuple to an unknown group label, as well as a target number of groups, which specifies the desired granularity of group labels. As an example, a user might group-by the topics presented in a set of ArXiv papers, wishing to find 10 key groups. The group-by operator must *discover* representative group labels and assign a label to each tuple. In general, performing the unsupervised group discovery is a clustering task, which is NP-hard [39]. Clustering algorithms over points in a metric space typically optimize the potential function tractably using coordinate descent algorithms, such as k-means. For the semantic group-by, the clustering task is over unstructured fields with a natural language similarity function specified by $l_M(t_i, \mu_j)$, which imposes a real-valued score between a tuple t_i and a candidate label μ_j. This operator poses the following optimization problem:

$$\underset{\{\mu_1,...\mu_C\},\ \mu_i \in V^{\mathbb{N}}}{\operatorname{argmax}} \sum_{t_i \in T} \max_{j \in 1...C} l_M(t_i, \mu_j)$$

where μ_i is a group label, consisting of tokens in vocabulary, V.

Reference Algorithm. Since this operator, by definition, entails the NP-hard clustering problem [39], our reference algorithm uses a tractable clustering heuristic to discover group labels, then performs point-wise classification to assign each record to a discovered group label. Specifically, our LLM-based clustering algorithm discovers centers $\mu_1, ..., \mu_C$ by first performing a semantic projection, with model invocations $M(t_i, l), t_i \in T$, prompting the LLM to predict a candidate label for each input tuple. Then, we embed these candidate labels and perform an efficient vector clustering using k-means to construct C groups. For each group, we top-k sample by centroid-similarity scores, and perform a semantic aggregation to synthesize an appropriate label over each group. This provides a reasonable clustering heuristic similar to prior work [15], although alternative heuristics are possible. In the second stage, our reference algorithm uses the C generated labels, $\mu_1, ...\mu_C$, and performs point-wise assignments $M(t_i, (l, \mu_1, ..., \mu_C)), t_i \in T$. We choose point-wise classification to avoid the long-context scaling challenges studied in prior works [15, 23]. This algorithm yields $O(|T|)$ LLM call complexity.

3 Optimized Execution Plans for Semantic Operators

Semantic operators open up a rich design space for optimizations to reduce cost while providing statistical accuracy guarantees. This design space includes novel opportunities specific to the unique properties of LLM-based execution, as well as the application of traditional optimizations from relational data processing (e.g. operator re-ordering [11–13, 40]). In this section, we begin by describing our existing research studying optimizations for semantic operator queries. We overview three optimization regimes of the semantic operator model. We will begin with optimizations for individual operators (3.1), which represents the simplest setting and follows directly from the definitions provided in Section 5 involving cost reduction for individual operators with respect to their reference algorithm. We then discuss extensions of the semantic operator model to optimizations for multi-operator queries (Section 3.2), and to optimizations with user-labeled data (Section 3.3), which open up additional opportunities.

3.1 Optimizations for Individual Operators

To begin, we consider optimizations for the relatively simple setting that involves a single semantic operator and no user-labeled data. Here, we directly apply the definition of a correct optimization from Section 2.3, and our goal is to reduce the execution cost of the semantic operator while providing a statistical accuracy guarantee for the execution with respect to the operator's reference algorithm. Excitingly, even this seemingly simple setting of single-operator optimizations already opens up a tremendous design space that can be transparently exploited by system optimizers, creating new research opportunities. In fact, in our prior work [27] we demonstrate up to 1000× speedups with accuracy guarantees with respect to an operator's reference algorithm.

Two key mechanisms for reducing cost of each operator are the use of proxies and cost-based planning. First, proxies refer to any means of approximating the LLM-based invocations of the reference algorithm. This could be a small language model, code-based execution, an embedding-similarity score, or other tool executions. The use of proxies in conjunction with statistical techniques from approximate query processing often allows for significant cost reduction with statistical accuracy guarantees. Secondly, cost-based planning allows the optimizer to enumerate multiple possible AI algorithms—each of which may use one or more proxies—before choosing to execute the lowest cost one to exploit the accuracy-cost tradeoff space. This is often necessary since the effectiveness of a candidate plan with particular models and proxies may vary widely depending on the specific task and dataset given by the user's query.

3.1.1 Example: Semantic Joins

To provide an example, we consider the semantic join operator, where the operator's accuracy metrics are precision and recall. The reference algorithm for the semantic join invokes an LLM over every pair of tuples from the right and left join table, prompting the model to output a boolean value indicating whether the predicate holds for the input tuple pair. This reference algorithm has a quadratic LLM call complexity with respect to the cardinality of the input tables, making it expensive at the size of datasets scale. One way we can approximate this algorithm is using a proxy model to perform predicate evaluations in place of the LLM-based predicate evaluation when possible—this represents a *proxy-join* pattern. In our prior work [27], we experiment with embedding-based proxies that provide similarity scores between the right and left join key of each tuple pair. Our method uses model cascades[41–43] to decide when to use the proxy model or resort to the LLM based on the accuracy guarantees requested by the user query. When the user predicate is easy to approximate with embedding similarity, we expect this approximation to reduce cost significantly. For instance, if the user predicate is "the {article:left} is relevant to the {topic:right}", we might expect many LLM calls to be well-approximated by similarity scores taken between each article embedding and each topic embedding. Since the proxy is far cheaper than an LLM call, using this approximation can significantly reduce cost while maintaining high accuracy.

However, in other cases, embedding similarities may be too weak of a proxy. For instance, suppose the user predicate is "the {article:left} claims to outperform the {method_name:right}", where the right table is a dataset about research methods with an attribute for method_name, and the left table contains research papers and an attribute for the article. Here, the predicate requires more nuanced reasoning that embedding similarity scores between the article text and method name cannot sufficiently capture. As a result, an alternative algorithm, a *project-proxy-join* pattern, might be more appropriate, where the system first projects the right or left join keys to make their domains more similar, allowing the proxy, in this case embedding similarities, to perform better. A simple form of projections which we have studied [27] provides the LLM with the left join key and prompts it to predict the right join key based on the predicate relationship. In this example, the algorithm would invoke an LLM over each article, prompting the model to extract the method names which the article claims to outperform.

Similar to the proxy-join pattern, this algorithm then uses model cascades to leverage the proxy when possible while ensuring statistical accuracy guarantees. These two join patterns represent just two candidate plans, but many others could be generated by a query optimizer, and finding new join plans remains an open research question with exciting recent work emerging in this area [44]. Crucially, once multiple plans are generated, the optimizer must pick the lowest-cost plan to optimize for the given user query, configured models, and dataset. Efficient search over many possible plans, with possibly many choices of proxies and alternative algorithms remains an interesting, open area of research.

3.2 Extension to Multi-Operator Queries

We now consider an extension of the semantic operator model to multi-operator queries, containing one or more semantic operators possibly composed with relational operators. In this setting, the user specifies accuracy targets at the *query-level*, and the query's reference algorithm is constructed by composing together the reference algorithm of individual semantic operators. We have been actively exploring optimizations in this setting, and recent research likewise studies this space, demonstrating strong promise for achieving global-level accuracy guarantees by directly extending the semantic operator model [45]. In the future, we envision a rich space of logical optimizations, many of which may apply techniques from relational query processing, such as operator re-ordering or fusion. The optimizer is then responsible for assigning the appropriate error budget to each logical semantic operator to reduce cost and meet the accuracy guarantees of the composite query. For instance, if a user composes a query with two semantic filters, the optimizer might fuse these filters into a single logical filter, assigning the query-level error budget entirely to the fused filter. Alternatively, the optimizer may retain the two logical filters as separate operators, and assign a higher error budget to the filter with the more difficult predicate evaluation, requiring a search and cost-based plan selection to jointly optimize the error-budgets assigned to both individual operators.

3.3 Optimizations with Labeled Data

So far we have considered the general setting, where the user provides only a semantic operator query, and the optimizer must explore, select and execute plans in the absence of any ground truth data. In this case, the optimizer's goal is cost reduction with accuracy guarantees with respect to the reference algorithm of the provided query. However, if the user provides a labeling function to score the correctness of query results, a wider scope of optimizations are possible, including changes to the reference algorithm itself. Each reference algorithm, as given by a user-written semantic operator query, is specified by user-programmed natural language expression, a configured model and the operators' transformations. These parameters can be optimized against the user-provided labeling function creating a large space of optimization over prompts [46, 47], model selection, and logical plans. Since our initial work implementing semantic operators in LOTUS [26], we have extended the system to support this setting with prompt optimizations and logical plan updates, such as operator re-ordering, for multi-operator queries. Tractably searching the accuracy-cost tradeoff space and ensuring reliability of selected plans remains an open research question that holds promise for future work.

4 Application Case Studies

Since our initial work introducing semantic operators and our open-source implementation in LOTUS [26], we have been excited to see a diverse set of user applications and emerging workloads that underscore the need for these new primitives for semantic bulk processing. In this section, we share our experience

```
# find examples of agent failures
traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")

# explain each failure
traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")\
              .sem_map("summarize the failure cause of the agent {trace}")

# summarize common failure patterns
traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")\
              .sem_map("summarize the key failure and cause from the agent {trace}",
suffix="agent_failure_summary")\
              .sem_agg("given each {agent_failure_summary}, summarize recurring
failure patterns and ways to improve my agent")
```

Figure 3: Example LOTUS program using semantic operators for agent trace analysis. The dataset, traces_dataset, is a large dataset of agent traces with the attribute, trace, containing the long-form text of each agent trace.

deploying semantic operators in the LOTUS open-source system and highlight several exciting case studies, including agent trace analysis, semantic clustering and generative research synthesis.

4.1 Agent Trace Analysis

As agentic systems become increasingly more pervasive, agent traces serve as rich artifacts capturing common agent patterns, failure modes and unexpected behaviors. As a result, effective tools for processing these large unstructured datasets hold potential to unlock new insights critical for improving deployed agents. LOTUS users have used semantic operator queries for this purpose, automating insights which would otherwise require tedious manual inspection. Typically, agent traces can be long, containing thousands of tokens per trace, making manual inspection over even dozens to hundreds of execution traces difficult and time-consuming. Figure 3 provides several example user queries. The first one demonstrates a simple semantic filter, where the user processes the dataset, `traces_dataset` with the text attribute `trace`, to find traces that demonstrate a failure. The next query chains on a semantic projection, taking the traces with agent failures and summarizing the key cause of the failure in each one. In the final query, the user program chains a final semantic aggregation, which takes all failure summaries of individual traces from the `agent_failure_summary` attribute and creates a single summary of recurring patterns and ways the user could improve the agent.

4.2 Semantic Clustering and Classification

Another common use case among LOTUS users is semantic clustering to taxonomize and classify records in an unstructured dataset. In this application, users have large unstructured datasets, and they must first *discover* key groups before classifying each record in the dataset. One example of this comes from recent research with collaborators studying agents in production [18]. The study involved processing large datasets of human-written responses provided as free-form text and extracting insights. For example, in one question, users listed applications and use cases they were using agents for, and the survey analysis aimed to taxonomize these responses according to 5-10 key domains and classify each response using these labels. As shown in Figure 4, a semantic aggregation is used to first surface patterns across the human-written survey answers to identify candidate labels to produce a taxonomy. In the study, researchers then validated these labels and chose a subset as the taxonomy labels (e.g., Technology, Finance & Banking, Corporate Services, Legal & Compliance, etc). These labels were then used to classify each individual human response, which as the figure shows, can be performed with a semantic projection. We have also seen similar patterns in other settings, including research analysis

```
# taxonomize survey response data and list candidate labels
responses.sem_agg("list the key domains for agent use based on the survey responses given by {Q4_answers}")

# label each response using the produced taxonomy
responses.sem_map(f"classify the {{Q4_answers}} according to one of the following labels: {taxonomy_labels}")
```

Figure 4: Example LOTUS program using semantic aggregation to surface candidate taxonomy labels, followed by classification with a semantic map using chosen the labels, taxonomy_labels.

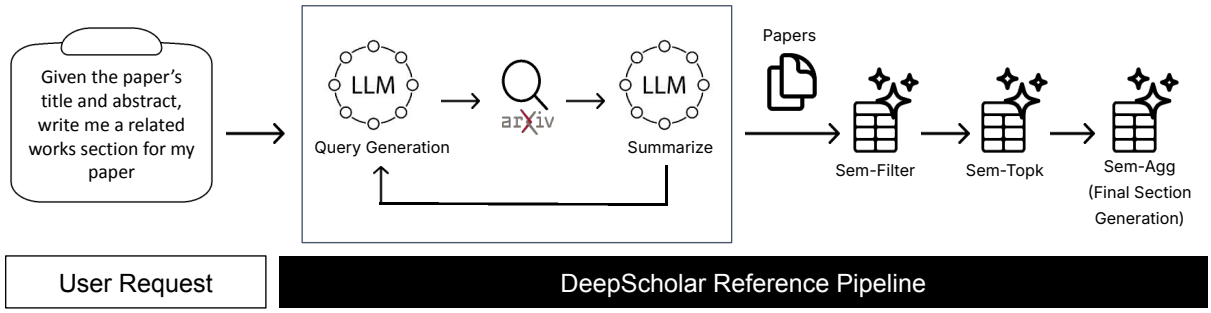

Figure 5: Overview of DeepScholar-ref. The system iteratively writes queries and performs web search, before passing the search results through series of semantic operators using the LOTUS system for LLM-based data-processing, including filtering step to discard irrelevant sources, a top-k ranking step to find most relevant sources, and an aggregation step to generate the final report from all remaining sources.

that processes LLM responses [48]. In these settings of exploratory data analysis, we observe iterative development with a human in the loop is often a key component, for which we find LOTUS' API with an option for eager execution useful.

4.3 Deep Research and Generative Research Synthesis

Recently, systems for *generative research synthesis* have emerged, promising to automate synthesis tasks that require processing large numbers of sources to produce long-form reports. These synthesis tasks are complex and traditionally demand hours of literature searching, reading and writing by human experts. Over the past few months, we have deployed DeepScholar, a deep research system for generative research synthesis which has over 12.8 thousand research queries and thousands of users. DeepScholar is a system implemented on top of LOTUS using our open-source pipeline, DeepScholar-ref, for semantic bulk processing. As Figure 5 shows, DeepScholar-ref decomposes the difficult research synthesis task into a semantic operator query. After iteratively searching for documents, the retrieved source set is processed with a semantic filtering step, which assesses whether each document is relevant to the user query. Our pipeline optionally performs a semantic ranking to retain the most relevant documents and then performs a semantic aggregation to produce the final report which answers the user query.

We measure the performance of our open-source reference pipeline using Deepscholar-bench [19], which we developed as a systematic benchmark for generative research synthesis to provide an automated framework and holistically measure performance of systems across three key dimensions—knowledge synthesis, retrieval quality, and verifiability. We find that DeepScholar-ref offers competitive performance to OpenAI's DeepResearch while being 4.3× cheaper and 2.28× faster. Notably, even with the same or weaker models, DeepScholar-ref attains up to 6.3× higher Verifiability scores, which likely reflects the effectiveness of semantic operators as primitives for task decomposition and synthesis over large datasets.

5 Conclusion

In this paper, we provided an overview of our ongoing research developing the semantic operator model, the first formalism for general-purpose, AI-based operations using natural language parameters. Our work introduces a new set of expressive, language-based operators—including filters, joins, top-k ranking and aggregations—and provides a definition for correct operator executions and optimizations. This model naturally extends to multi-operator queries and settings with user-labeled data, creating rich design spaces for optimization that open up exciting areas for new research. The diverse applications we have seen among LOTUS users underscores the value semantic operators hold as key primitives for LLM-based data analysis and semantic processing over large datasets. Since open-sourcing LOTUS, we have seen a growing user-base, as well as adoption of semantic operators among large database vendors, and a growing body of exciting research work. Ultimately, we believe semantic operators serve as a foundational piece towards the future of AI-native data systems that will enable rich semantic-based processing over vast knowledge corpora.

References

[1] "arXiv.org ePrint archive."

[2] K. D'Oosterlinck, F. Remy, J. Deleu, T. Demeester, C. Develder, K. Zaporojets, A. Ghodsi, S. Ellershaw, J. Collins, and C. Potts, "BioDEX: Large-Scale Biomedical Adverse Drug Event Extraction for Real-World Pharmacovigilance," Oct. 2023. arXiv:2305.13395 [cs].

[3] "Discovery Insight Platform."

[4] MotherDuck, "Introducing the prompt() Function: Use the Power of LLMs with SQL! - MotherDuck Blog."

[5] WilliamDAssafMSFT, "Intelligent Applications - Azure SQL Database," Dec. 2024.

[6] "Large Language Model (LLM) Functions (Snowflake Cortex) | Snowflake Documentation."

[7] "LLM with Vertex AI only using SQL queries in BigQuery."

[8] "AI Functions on Databricks."

[9] "Large Language Models for sentiment analysis with Amazon Redshift ML (Preview) | AWS Big Data Blog," Nov. 2023. Section: Amazon Redshift.

[10] S. Liu, A. Biswal, A. Cheng, X. Mo, S. Cao, J. E. Gonzalez, I. Stoica, and M. Zaharia, "Optimizing LLM Queries in Relational Workloads," Mar. 2024. arXiv:2403.05821 [cs].

[11] S. Liu, J. Xu, W. Tjangnaka, S. J. Semnani, C. J. Yu, and M. S. Lam, "SUQL: Conversational Search over Structured and Unstructured Data with Large Language Models," Mar. 2024. arXiv:2311.09818 [cs].

[12] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, and G. Vitagliano, "A Declarative System for Optimizing AI Workloads," May 2024. arXiv:2405.14696 [cs].

[13] Y. Lin, M. Hulsebos, R. Ma, S. Shankar, S. Zeigham, A. G. Parameswaran, and E. Wu, "Towards Accurate and Efficient Document Analytics with Large Language Models," May 2024. arXiv:2405.04674 [cs].

[14] S. Shankar, T. Chambers, T. Shah, A. G. Parameswaran, and E. Wu, "DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing," Dec. 2024. arXiv:2410.12189 [cs].

[15] H. Dai, B. Y. Wang, X. Wan, B. Dai, S. Yang, A. Nova, P. Yin, P. M. Phothilimthana, C. Sutton, and D. Schuurmans, "UQE: A Query Engine for Unstructured Databases," Nov. 2024. arXiv:2407.09522 [cs].

[16] D. Kang, J. Guibas, P. Bailis, T. Hashimoto, Y. Sun, and M. Zaharia, "Accelerating approximate aggregation queries with expensive predicates," *Proceedings of the VLDB Endowment*, vol. 14, pp. 2341–2354, July 2021.

[17] D. Kang, E. Gan, P. Bailis, T. Hashimoto, and M. Zaharia, "Approximate selection with guarantees using proxies," *Proceedings of the VLDB Endowment*, vol. 13, pp. 1990–2003, Aug. 2020.

[18] M. Z. Pan, N. Arabzadeh, R. Cogo, Y. Zhu, A. Xiong, L. A. Agrawal, H. Mao, E. Shen, S. Pallerla, L. Patel, S. Liu, T. Shi, X. Liu, J. Q. Davis, E. Lacavalla, A. Basile, S. Yang, P. Castro, D. Kang, J. E. Gonzalez, K. Sen, D. Song, I. Stoica, M. Zaharia, and M. Ellis, "Measuring Agents in Production," Feb. 2026. arXiv:2512.04123 [cs].

[19] L. Patel, N. Arabzadeh, H. Gupta, A. Sundar, I. Stoica, M. Zaharia, and C. Guestrin, "DeepScholar-Bench: A Live Benchmark and Automated Evaluation for Generative Research Synthesis," Feb. 2026. arXiv:2508.20033 [cs].

[20] "python-bigquery-dataframes/notebooks/experimental/semantic_operators.ipynb at main · googleapis/python-bigquery-dataframes."

[21] P. Liskowski, B. Han, P. Aggarwal, B. Chen, B. Jiang, N. Jindal, Z. Li, A. Lin, K. Schmaus, J. Tayade, W. Zhao, A. Datta, N. Wiegand, and D. Tsirogiannis, "Cortex AISQL: A Production SQL Engine for Unstructured Data," Nov. 2025. arXiv:2511.07663 [cs].

[22] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," vol. 13, no. 6, 1970.

[23] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," Nov. 2023. arXiv:2307.03172 [cs].

[24] 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," 2025.

[25] S. Jo and I. Trummer, "ThalamusDB: Approximate Query Processing on Multi-Modal Data," *Proc. ACM Manag. Data*, vol. 2, pp. 186:1–186:26, May 2024.

[26] L. Patel, S. Jha, C. Guestrin, and M. Zaharia, "LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data," July 2024. arXiv:2407.11418 [cs] version: 1.

[27] 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," *Proceedings of the VLDB Endowment*, vol. 18, pp. 4171–4184, July 2025.

[28] C. A. R. Hoare, "Algorithm 65: find," *Commun. ACM*, vol. 4, pp. 321–322, July 1961.

[29] S. Desai and G. Durrett, "Calibration of Pre-trained Transformers," Mar. 2020.

[30] A. Drozdov, H. Zhuang, Z. Dai, Z. Qin, R. Rahimi, X. Wang, D. Alon, M. Iyyer, A. McCallum, D. Metzler, and K. Hui, "PaRaDe: Passage Ranking using Demonstrations with Large Language Models," Oct. 2023. arXiv:2310.14408 [cs].

[31] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, B. Newman, B. Yuan, B. Yan, C. Zhang, C. Cosgrove, C. D. Manning, C. Ré, D. Acosta-Navas, D. A. Hudson, E. Zelikman, E. Durmus, F. Ladhak, F. Rong, H. Ren, H. Yao, J. Wang, K. Santhanam, L. Orr, L. Zheng, M. Yuksekgonul, M. Suzgun, N. Kim, N. Guha, N. Chatterji, O. Khattab, P. Henderson, Q. Huang, R. Chi, S. M. Xie, S. Santurkar, S. Ganguli, T. Hashimoto, T. Icard, T. Zhang, V. Chaudhary, W. Wang, X. Li, Y. Mai, Y. Zhang, and Y. Koreeda, "Holistic Evaluation of Language Models," Nov. 2022.

[32] X. Ma, X. Zhang, R. Pradeep, and J. Lin, "Zero-Shot Listwise Document Reranking with a Large Language Model," May 2023.

[33] R. Pradeep, S. Sharifymoghaddam, and J. Lin, "RankVicuna: Zero-Shot Listwise Document Reranking with Open-Source Large Language Models," Sept. 2023. arXiv:2309.15088 [cs].

[34] R. Pradeep, S. Sharifymoghaddam, and J. Lin, "RankZephyr: Effective and Robust Zero-Shot Listwise Reranking is a Breeze!," Dec. 2023. arXiv:2312.02724 [cs].

[35] Z. Qin, R. Jagerman, K. Hui, H. Zhuang, J. Wu, L. Yan, J. Shen, T. Liu, J. Liu, D. Metzler, X. Wang, and M. Bendersky, "Large Language Models are Effective Text Rankers with Pairwise Ranking Prompting," Mar. 2024. arXiv:2306.17563 [cs].

[36] D. Sachan, M. Lewis, M. Joshi, A. Aghajanyan, W.-t. Yih, J. Pineau, and L. Zettlemoyer, "Improving Passage Retrieval with Zero-Shot Question Generation," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, (Abu Dhabi, United Arab Emirates), pp. 3781–3797, Association for Computational Linguistics, 2022.

[37] W. Sun, L. Yan, X. Ma, S. Wang, P. Ren, Z. Chen, D. Yin, and Z. Ren, "Is ChatGPT Good at Search? Investigating Large Language Models as Re-Ranking Agents," Apr. 2023.

[38] Y. Chang, K. Lo, T. Goyal, and M. Iyyer, "BooookScore: A systematic exploration of book-length summarization in the era of LLMs," Apr. 2024. arXiv:2310.00785 [cs].

[39] S. Dasgupta, "The hardness of k-means clustering,"

[40] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri, "Accelerating Machine Learning Inference with Probabilistic Predicates," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), pp. 1493–1508, Association for Computing Machinery, May 2018.

[41] M. Yue, J. Zhao, M. Zhang, L. Du, and Z. Yao, "Large Language Model Cascades with Mixture of Thoughts Representations for Cost-efficient Reasoning," Feb. 2024. arXiv:2310.03094 [cs].

[42] L. Chen, M. Zaharia, and J. Zou, "FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance," May 2023. arXiv:2305.05176 [cs].

[43] D. Kang, E. Gan, P. Bailis, T. Hashimoto, and M. Zaharia, "Approximate Selection with Guarantees using Proxies," Jan. 2022. arXiv:2004.00827 [cs].

[44] I. Trummer, "Implementing Semantic Join Operators Efficiently," Oct. 2025. arXiv:2510.08489 [cs].

[45] G. Sanmartino, M. Urban, P. Papotti, and C. Binnig, “The Stretto Execution Engine for LLM-Augmented Data Systems,” Feb. 2026. arXiv:2602.04430 [cs].

[46] M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, Z. Huang, C. Guestrin, and J. Zou, “TextGrad: Automatic "Differentiation" via Text,” June 2024. arXiv:2406.07496 [cs].

[47] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, “DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines,” Oct. 2023.

[48] L. Dunlap, K. Mandal, T. Darrell, J. Steinhardt, and J. E. Gonzalez, “VibeCheck: Discover and Quantify Qualitative Differences in Large Language Models,” Apr. 2025. arXiv:2410.12851 [cs].