Bulletin of the Technical Community on

Data Engineering

June 2026 Vol. 50 No. 2

IEEE Computer Society

Letters

Letter from the Editor-in-Chief . . . . *Haixun Wang* 1
Letter from the Special Issue Editors . . . . *Carsten Binnig, Paolo Papotti, Gerardo Vitagliano* 3

Special Issue on LLM-Native Data Systems

Semantic Query Plan Optimization for AI-Powered Analytics . . . .
. . . . *Gerardo Vitagliano, Matthew Russo, Michael Cafarella, Sam Madden, Tim Kraska* 5
What Do Users Actually Do with LLM-Powered Data Systems? . . *Shreya Shankar, Aditya G. Parameswaran* 18
Towards AI-Native Data Systems with the Semantic Operator Model and LOTUS . . . .
. . . . *Liana Patel, Carlos Guestrin, Matei Zaharia* 32
Expanding the Physical Design Space of LLM Data Systems . . . .
. . . . *Gabriele Sanmartino, Matthias Urban, Carsten Binnig, Paolo Papotti* 47
Compositional Online Learning for Semantic Data Processing Systems . . . .
. . . . *Paweł Liskowski, Fuheng Zhao, Benjamin Han, Anupam Datta, Dimitris Tsirogiannis* 61
ThalamusDB: Semantic Approximate Query Processing . . . . *Immanuel Trummer* 76

Conference and Journal Notices

TCDE Membership Form 90

Editorial Board

Editor-in-Chief
Haixun Wang
EvenUp
haixun.wang@evenup.ai

Associate Editors
Carsten Binnig
TU Darmstadt, Germany

Paolo Papotti
EURECOM, France

Gerardo Vitagliano
MIT CSAIL, USA

Fatma Özcan
Systems Research, Google, USA

Xi He
University of Waterloo, Canada

Nan Tang
Hong Kong Univ of Science and Technology (Guangzhou), China

Production Editor
Jieming Shi
The Hong Kong Polytechnic University
Hong Kong SAR, China

Distribution
Brookes Little
IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
eblittle@computer.org

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Community on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modeling, theory and application of database systems and technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of TC on Data Engineering, IEEE Computer Society, or authors' organizations.

The Data Engineering Bulletin web site is at
http://tab.computer.org/tcde/bull_about.html.

TCDE Executive Committee

Chair
Murat Kantarcioglu
University of Texas at Dallas

Executive Vice-Chair
Karl Aberer
EPFL

Executive Vice-Chair
Thomas Risse
Goethe University Frankfurt

Vice Chair
Erich J. Neuhold
University of Vienna, Austria

Vice Chair
Malu Castellanos
Teradata Aster

Vice Chair
Xiaofang Zhou
The University of Queensland

Editor-in-Chief of Data Engineering Bulletin
Haixun Wang
Instacart

Diversity & Inclusion and Awards Program Coordinator
Amr El Abbadi
University of California, Santa Barbara

Chair Awards Committee
S Sudarshan
IIT Bombay, India

Membership Promotion
Guoliang Li
Tsinghua University

TCDE Archives
Wookey Lee
INHA University

Advisor
Masaru Kitsuregawa
The University of Tokyo

SIGMOD Liaison
Fatma Ozcan
Google, USA

Letter from the Editor-in-Chief

For decades, data management has been guided by a remarkably durable abstraction: users declare what they want, and the system determines how to obtain it. This idea gave us the relational model, SQL, cost-based optimization, physical data independence, and the large-scale data platforms that now underlie science, commerce, government, and everyday digital life.

The rise of large language models and generative AI does not replace this tradition. It asks us to extend it. In the previous issue, we examined the emergence of agentic data management: systems that plan, reason, invoke tools, and act in pursuit of high-level goals. This issue turns to the data-management foundation required for that agentic future. If agents are to operate reliably over enterprise knowledge, scientific literature, legal records, financial filings, images, tables, and multimodal archives, they cannot rely on prompting alone. They need data systems built for semantic access, evidence, uncertainty, cost control, and trust.

The central challenge is that the data we now want to manage is no longer confined to well-formed tables. It is scattered across documents, spreadsheets, figures, PDFs, web pages, logs, emails, code, audio, video, and heterogeneous data lakes. Much of its meaning is latent. A database may store a document, but not the facts implied by the document. A warehouse may contain extracted fields, but not the long tail of concepts users will ask about tomorrow. A search index may retrieve a passage, but not know whether the passage supports, contradicts, or merely resembles the user's claim.

This changes the nature of query processing. In traditional systems, a predicate is usually symbolic and exact. In the LLM era, a predicate may be semantic: "the filing indicates liquidity risk," "the report recommends future surgery," "the table supports a year-over-year comparison," or "the document contains evidence of noncompliance." Such predicates are not simple filters over stored values. They may require retrieval, extraction, normalization, reasoning, comparison, computation, and evidence selection. They may also be uncertain. The result of a query may be an answer, but it may also need to include confidence, provenance, bounds, examples, and an estimate of what it would cost to improve the answer.

The current state of the field is promising but incomplete. Retrieval-augmented generation has shown that language models can be grounded in external context, but retrieval alone is not enough. Long-context models can read more, but more context is not the same as better evidence. Structure-aware retrieval can respect sections, tables, and reading order, but many questions require reusable structured facts, not just better passages. Agents can search and reason, but without disciplined execution environments they can become slow, expensive, and hard to audit. Graphs, summaries, embeddings, caches, and generated questions all help, but each introduces new tradeoffs in quality, cost, latency, and maintainability.

The next generation of data systems must therefore treat semantic work as a first-class systems problem. We need logical abstractions for semantic operators, but also physical plans for executing them. We need optimizers that reason not only about CPU, memory, and I/O, but also about model choice, token cost, latency, retrieval coverage, extraction quality, approximation error, and provenance. We need indexes not only over strings and vectors, but over document structure, evidence, entities, questions, dependencies, summaries, cached model states, and reusable semantic fields. We need materialized views not only of relational joins, but of extracted facts, validated evidence records, and frequently used concepts.

A key future direction is the separation of semantic intent from model execution. Users and applications should be able to express what they mean at a high level, while the system decides whether to answer from structured data, retrieve evidence, run targeted extraction, invoke a stronger model, use a cheaper proxy, sample the corpus, or return an approximate result with bounds. This is the natural extension of data independence into the generative-AI era. The goal is not to hide uncertainty, but to manage it explicitly.

Another direction is the movement of repeated semantic computation from query time to indexing

time. If many users will ask related questions over the same corpus, then expensive work should not be repeated for every query. Document structure can be parsed once. Tables can be normalized once. Candidate facts can be extracted once. Dependency links can be scored once. Frequently used semantic fields can be materialized and validated. The online system can then become faster, cheaper, and more predictable. This is familiar database thinking, but applied to a new substrate: meaning.

At the same time, not everything can or should be precomputed. Users will continue to ask new, ambiguous, and evolving questions. Real workflows are iterative: users explore, inspect samples, revise definitions, validate outputs, and gradually decide what they truly mean. Data systems must therefore support the full lifecycle of semantic analysis: drafting a query, estimating feasibility, sampling results, refining a rubric, validating evidence, executing at scale, and promoting useful concepts into durable data assets.

Trust will be the defining constraint. In conventional data systems, a result can often be traced to a query over stored facts. In generative systems, a result may pass through retrieval, prompting, extraction, reranking, model judgment, and synthesis. Without provenance, evaluation, and auditability, such systems will be impressive but unsafe. The future database must be not only a system of record, but a system of evidence: it must know where an answer came from, what was read, what was ignored, what remains uncertain, and what would be required to make the answer stronger.

This is the opportunity before our community. The foundations of data management—declarativity, optimization, physical design, approximation, provenance, transactions, evaluation, and data independence—are not obsolete. They are urgently needed. But each must be reimagined for a world in which data is heterogeneous, queries are semantic, execution is model-driven, and correctness is inseparable from evidence and cost.

The future will not be built by language models alone, nor by databases unchanged from the past. It will be built by bringing the discipline of data management to the power and ambiguity of generative AI. I invite you to read this issue as a call to that work: to build systems that do not merely store information, retrieve passages, or generate fluent answers, but that transform raw, messy, multimodal data into trustworthy, cost-aware, and actionable knowledge.

Haixun Wang
EvenUp

Letter from the Special Issue Editors

The rapid progress of large language models has created a new class of data-management workloads. Users increasingly expect data systems to operate not only over relational tables, but also over documents, spreadsheets, images, logs, scientific articles, contracts, transcripts, and other heterogeneous collections. They also expect to express intent in natural language, and to obtain answers that are not only fluent, but grounded, auditable, and cost-effective. This special issue examines how the data engineering community is responding to this shift.

The articles in this issue focus on an emerging research area that we may broadly call semantic data systems: systems that treat language-based interpretation, extraction, ranking, joining, aggregation, and reasoning as first-class data-processing tasks. This perspective brings familiar database questions into a new setting. How should users express semantic intent? What are the logical operators and programming abstractions? How should systems choose among alternative model calls, prompts, indexes, caches, approximations, and execution strategies? How should quality, cost, latency, and provenance be exposed to users? And how should we evaluate systems whose behavior depends on models, data distributions, and user-defined semantics?

The contributions collected here approach these questions from complementary angles. Vitagliano, Russo, Cafarella, Madden, and Kraska discuss semantic query plan optimization for AI-powered analytics. Their article describes how semantic operators can support declarative programs over unstructured and multimodal data, and how systems such as Palimpzest, Abacus, and Carnot move from fixed semantic plans toward optimized agentic analytics. A central theme is that optimization must account not only for traditional resource costs, but also for model quality, uncertainty, latency, and monetary cost.

Shankar and Parameswaran take a user-centered view of LLM-powered data systems. Drawing on their experience with DocETL and DocWrangler, they show that users rarely arrive with a final, well-specified query. Instead, they explore data, revise prompts, validate intermediate results, and construct bespoke operators that are specific to their documents and domains. Their article is an important reminder that semantic data systems must be designed for iterative authoring and validation, not only for one-shot execution.

Patel, Guestrin, and Zaharia present the semantic operator model and the LOTUS system. Their work formalizes natural-language operators such as semantic filters, joins, top-k, grouping, and aggregation, and studies how such operators can be optimized while preserving accuracy guarantees with respect to reference implementations. This line of work highlights the need for model-data independence: users should be able to specify high-level semantic transformations while systems decide how to execute them accurately and efficiently.

Sanmartino, Urban, Binnig, and Papotti explore the physical design space of LLM data systems. Their article argues that the performance of semantic workloads depends heavily on execution-time and storage-time design choices, including caching, compression, batching, reuse, and materialized semantic representations. As with classical database systems, logical expressiveness alone is not enough: the physical layer determines whether workloads can scale in practice.

Liskowski, Zhao, Han, Datta, and Tsirogiannis study compositional online learning for semantic data processing systems. Their article considers how systems can adapt during execution, learning from feedback, intermediate results, and workload structure. This perspective is particularly relevant when semantic predicates and transformations are uncertain, task-specific, and expensive to evaluate.

Finally, Trummer introduces ThalamusDB and the idea of semantic approximate query processing. The article connects semantic query execution with approximation, sampling, and uncertainty-aware answers. As semantic workloads grow in scale and cost, approximate execution will be essential: users often need useful answers quickly, together with an understanding of confidence, error, and the cost of further refinement.

Taken together, these articles show that LLM-native data systems are not simply existing data systems with model calls added as external functions. Rather, they treat large language models as first-class citizens in the design of declarative query interfaces, logical operators, physical execution strategies, and interactive analysis workflows. Across the papers, LLMs appear in different roles: as implementations of semantic operators, as components of cost- and quality-aware query plans, as agents that rewrite and evaluate document pipelines, as building blocks for physical design and reuse, and as adaptive components in online and approximate processing.

Several common abstractions emerge from these contributions. Semantic operators provide a declarative way to express language-based filtering, mapping, joining, ranking, grouping, and aggregation over mixed structured and unstructured data. Model-in-the-loop planning makes optimization depend not only on CPU, memory, and I/O, but also on model selection, prompt design, token cost, latency, quality, and uncertainty. Agentic pipelines extend query processing beyond fixed plans, allowing systems to search, compute, revise, validate, and reuse intermediate results. Physical design techniques such as caching, compression, batching, materialization, indexing, and approximate execution then determine whether these abstractions can scale in practice.

The issue also surfaces a set of open challenges for the next generation of LLM-native data systems. Quality must be measured together with cost, latency, reproducibility, and robustness to model drift. Provenance and auditability must become part of the execution model, so that users can inspect not only the final answer, but also the evidence, prompts, model choices, intermediate decisions, and uncertainty behind it. Continual learning and feedback-driven adaptation are needed because semantic predicates and user intent evolve over time. Multi-tenant execution raises questions about isolation, fairness, resource allocation, and predictable cost. Finally, the community needs benchmarks and evaluation standards that reflect real workloads: multimodal data, bespoke user-defined operators, iterative query development, and deployment constraints.

We hope this special issue helps define LLM-native data systems as an emerging area at the intersection of data management, machine learning, and human-centered analytics. The challenge ahead is to build systems that make LLMs usable as reliable data-processing components: declarative enough for users to express intent, optimized enough to run at scale, transparent enough to support inspection and trust, and flexible enough to handle the heterogeneous and evolving data found in real applications.

We thank all authors for their contributions. We also thank Haixun Wang for the opportunity to organize this special issue and for his guidance throughout the process.

Carsten Binnig, Paolo Papotti, Gerardo Vitagliano
TU Darmstadt, EURECOM, MIT CSAIL

Semantic Query Plan Optimization for AI-Powered Analytics

Gerardo Vitagliano, Matthew Russo
Michael Cafarella, Sam Madden, Tim Kraska
MIT CSAIL, USA
{gerarvit, mdrusso, michjc, kraska, madden}@csail.mit.edu

Abstract

Semantic operators make it possible to express analytics tasks as declarative programs over unstructured and multimodal data. This abstraction is powerful, but it also moves a familiar database problem into a less predictable setting: executing a semantic query plan with generative AI operators yields much more uncertainty about output quality and cost. This paper provides an outlook on semantic query plan optimization, organized around a progression from semantic operator programming to optimized agentic analytics.

We first describe how users can leverage semantic operators in Palimpzest to turn natural-language tasks over unstructured data into logical query plans. We then present the cost-based optimizer Abacus, which chooses physical implementations using sampled estimates of quality, cost, and latency, together with a multi-armed bandit strategy and Pareto-Cascades to respect user-specified objectives and constraints. We next describe Carnot, which extends this model toward agentic analytics by adding contexts, `search`, and `compute` operators that let agents invoke optimized semantic programs during open-ended analysis. Finally, we summarize experimental insights: across the BioDEX and CUAD workloads, Abacus-optimized executions achieve 18.7%–39.2% better quality, up to 23.6× lower cost, and 4.2× lower latency than the next best system; on KramaBench workloads, optimized agent-invoked semantic programs achieve up to 1.95× better F1 than a standard open Deep Research agent and save up to 76.8% cost and 72.7% runtime compared with an agent using semantic operators as tools.

We argue that the next generation of systems for semantic query optimization must move from optimized execution of fixed semantic plans toward cost-aware execution of agentic analytics workflows.

1 Introduction

Consider an analyst who asks a natural-language question over a data lake of consumer-report files: *What is the ratio of identity theft reports in 2024 versus 2001?* [9]. The relevant evidence may sit across multiple files inside a mixed collection of reports, spreadsheets, tables, and text. A useful analytics system must find the right records, extract the relevant values, check that the numbers refer to the requested years and report type, compute the ratio, and return an answer that the analyst can verify. Similar patterns arise in many domains: in legal compliance, scientific processing, and digital humanities, users want factual answers over raw data for which there is no relational model or structured database.

Semantic operator systems [1, 13, 15, 18, 22, 24, 26, 29] were introduced to let users answer such queries. Rather than forcing developers to write a bespoke script for each unstructured analytics task, these systems expose AI-powered operators that mirror and extend relational operators: semantic filters, maps, joins, aggregations, often offering natural language interfaces [14] or binding into SQL-like syntax [4, 12]. The semantic operators abstraction is valuable because it gives users primitives to build their programs, and it gives systems concrete units of computation to optimize.

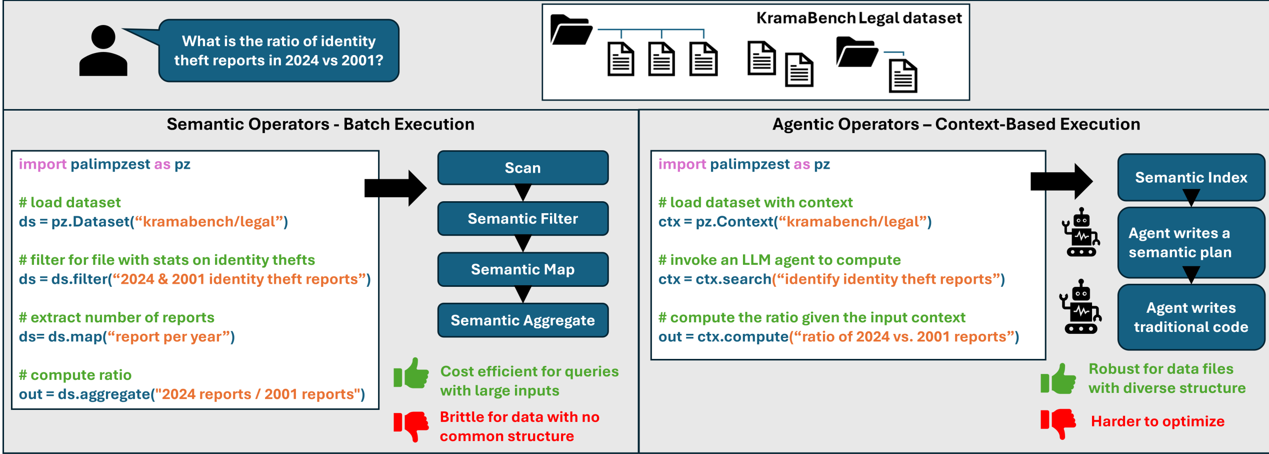

Figure 1: Overview of two different Palimpzest programs for the identity-theft query. The image compares a batch semantic operator pipeline (left) with one that leverages agentic `search` and `compute` (right).

This observation connects semantic analytics to a familiar database principle: declarative program optimization. Classical query optimizers search over equivalent physical plans and use cost estimates to choose an efficient plan, *semantic query optimization* inherits that goal but requires a different approach to quality and cost estimation. Optimizing semantic operators is challenging due to the uncertain trade-offs offered by a huge variety of their physical implementation choices. The implementation of an operator may use different LLM models, prompts, retrieval methods, operator orderings, and agentic loop tools. Each implementation differs significantly in its output quality, dollar cost, latency, and memory behavior, often in data-dependent ways. A large model with full context may recover evidence that a smaller model misses, but it may also be very expensive for a workload with thousands of records. Conversely, a cheaper implementation of a semantic filter may remove irrelevant records early, but can affect output quality by silently discarding data needed downstream.

In this paper, we describe a research progression across three systems: Palimpzest, Abacus, and Carnot. Palimpzest is the programming and runtime substrate: it lets users express semantic operator programs over unstructured data and compiles those programs into logical plans [13]. Abacus is the cost-based optimizer inside this substrate: it enumerates physical implementations, samples candidate operators on a small validation workload, estimates quality, cost, and latency, and selects a plan that satisfies the user's objective under those estimates [22].

The key algorithmic ideas are the use of a multi-armed bandit approach for estimating the performance of physical operators and a Pareto-aware extension of the Cascades-style plan search [6]. However, although the semantic operators in Palimpzest provide a structured framework for optimizing semantic query plans, they are not a complete solution to the challenges posed by the increasing complexity and variability of modern analytics tasks. Many emerging analytics tasks look less like fixed operator pipelines and more like open-ended investigation. Deep Research-style systems can plan, call tools, write code, inspect partial results, and revise their strategy while answering a natural-language question [5, 10, 16, 20, 28]. That flexibility helps on higher-level tasks such as the identity-theft ratio query, where a system is required to retrieve the correct files, search within records, compute intermediate values, and report a final answer. Full agentic systems have their own failure modes [17]: they may take shortcuts, terminate before processing enough evidence, repeat expensive work, or invoke tools in an order that is costly but not more accurate. Their plans are dynamic, but not necessarily optimized.

Carnot is the next step in this progression: it extends Palimpzest toward Deep Research-style

analytics, where agents can search, compute, write code, and invoke optimized semantic programs as part of a dynamic execution trace [21]. We frame this as a runtime problem: agents should keep their flexibility to plan and use tools, while optimized semantic operators should provide reliable execution for expensive or high-recall subtasks. The central abstractions are contexts, which expose data access methods and descriptions to agents; search and compute operators, which let agents produce and consume intermediate contexts; and materialized context management, which allows results from earlier queries to be indexed and reused. In Figure 1 we illustrate a query plan implemented both with semantic operators and with the new agentic capabilities in the Carnot system. Both approaches are effective in their own right, with semantic operators providing a more structured approach for batch processing, and agentic operators providing a more flexible and adaptive approach to handling complex analytics tasks.

Together, these systems move from semantic operators as programmable units, to estimated cost-based optimization of those units, to agentic analytics systems that can use optimized semantic programs inside a larger reasoning loop. In the remainder of this paper, we first describe semantic query plans and the abstractions that make unstructured analytics optimizable through the Palimpzest and Carnot systems (Section 2). We then examine Abacus as a cost-based optimizer for semantic operator systems, including its sampling strategy and Pareto-Cascades plan selection (Section 3). We next discuss the experimental evidence from Abacus and the broader benchmarking role of SemBench, emphasizing that semantic query systems must be evaluated on quality, cost, latency, memory, coverage, and scalability (Section 4). Finally, we outline open research challenges in the field: full-pipeline semantic query optimization for multimodal, agentic, and reusable analytics runtimes.

2 From Semantic Operators to Query Plans

Consider the identity-theft query from the introduction, inspired by the legal workload defined in the KramaBench benchmark [9]. The dataset consists of 132 files with statistics on fraud, identity theft, and other consumer reports, and the query asks the analytics system to compute the ratio of identity theft reports over a two-year span. Recent work has defined *semantic operators* as a set of AI-powered data transformations that mirror and extend relational operators [13, 19]. Using Palimpzest, users can define a program that combines semantic operators to answer the query by filtering for files with statistics on identity theft reports before using a map to compute the ratio.

The key difference between semantic operators and their relational counterparts is that their semantics are specified in natural language as opposed to a SQL expression or user-defined function. As a result, these operators' physical implementations typically require the use of one or more foundation models with semantic understanding. In this paper, we use *semantic query* to mean a query whose logical plan contains one or more semantic operators, and we use *semantic operator plan* to mean the pipeline or DAG formed by composing these operators. Optimizing the execution of semantic operator plans becomes possible once systems can identify a logical operator graph whose execution is possible with multiple physical implementation strategies.

Table 1 introduces the programming primitives to define semantic query plans available to users in the Palimpzest/Abacus systems. A scan introduces an object from a dataset. A map transforms an object into another object or a list of objects, which covers extraction, summarization, schema conversion, and classification. A filter applies a natural-language predicate and either keeps or removes an object. A retrieve operator augments an object with semantically similar objects from a vector database. Project, aggregate, and limit preserve familiar relational roles, although in semantic systems their inputs may have been produced by LLM-powered operators.

Semantic queries are declarative in the same sense as relational queries: they describe what should be computed without fixing the physical implementation of the operators.

Table 1: Semantic operators supported by Abacus. In the Abacus implementation, d is a valid JSON dictionary, but in principle d can be any serializable object. The $\cup$ symbol represents the union of output types. i is an integer index, P is a filter predicate, V is a vector database, and L is an integer limit. Aggregate includes group-by operations.

Operator Name	Symbol	Definition
Scan	ϕ	$\phi(i) \rightarrow d$
Map	μ	$\mu(d) \rightarrow d' \cup [d', d'', \ldots]$
Filter	σ	$\sigma(d, P) \rightarrow d \cup \emptyset$
Retrieve	ρ	$\rho(d, V) \rightarrow d'$
Project	π	$\pi(d) \rightarrow d' \subseteq d$
Aggregate	α	$\alpha([d', d'', \ldots]) \rightarrow \mathbf{R} \cup d$
Limit	λ	$\lambda([d', d'', \ldots], L) \rightarrow [d', \ldots, d^L]$

2.1 Logical Plans and Their Implementation

Each semantic operator corresponds to a **logical operator** that may be implemented by a variety of **physical operators**. For example, a semantic map can be implemented either with a Mixture-of-Agents architecture [31] or with a Reduced-Context generation [2, 32]. The former is a layered computation graph of LLM ensembles, while the latter samples only the most relevant input chunks before feeding them to an LLM. Each of these physical operators can be parameterized in numerous ways, e.g., the choice of models and temperature settings.

A logical semantic plan is therefore the operator graph directly induced by the user program. The performance of these physical operators can vary significantly based on the specific implementation and parameters used. Notably, users cannot easily predict which combination of physical operators will work well for a given logical plan. To address this challenge, Palimpzest leverages Abacus, an optimizer that given a user constraint (specified in terms of quality, latency, or costs), uses sampled estimates to select a near-optimal combination of physical operators under its cost model for a given logical plan. Abacus takes four inputs: a semantic program, an optimization objective, an input dataset, and optionally a small validation dataset. The semantic program is a pipeline of semantic operators defined by the user using the primitives of Table 1. The optimization objective is a constrained or unconstrained objective with respect to system output quality, dollar cost, and/or latency. The input dataset is an unstructured dataset of documents, images, songs, or other objects which the physical implementation will process. The validation dataset is a small set of sample labeled input-output pairs which Abacus can use to evaluate physical operators' quality using ground truth data.

Abacus compiles the program into a logical plan, applies rules to enumerate a search space of physical plans, builds a cost model by processing validation inputs with sampled physical operators, and returns an estimated Pareto-optimal plan based on its estimates and the user's objective. Section 3 will discuss details on the optimization algorithm.

2.2 Extending Semantic Plans with Analytical Reasoning

Palimpzest makes fixed semantic operator plans programmable and optimizable. However, high-level analytics tasks often start from a natural-language question rather than an explicit semantic operator program. Agentic systems are able to create code on the fly to query structured and unstructured data alike [7, 9, 17]. This class of system is often called "Deep Research," following commercial systems [5, 16, 27] that use extensive reasoning loops [33, 34] and external tool usage [23, 25] to obtain

answers to detailed queries. However, while these agentic systems often answer analytical queries by designing and executing query plans, their execution of these plans is often suboptimal because the exploratory nature of their reasoning involves planning and code execution without tight feedback loops [9]. For example, an agent may generate a plan to read every file until it finds the file with identity thefts in 2024, but reach context or timeout limitations when reading a large dataset, or stop after finding false positive files [17].

To support more efficient query execution patterns, we extended the semantic operators in Palimpzest to include three abstractions that allow for high-level query specification and agentic exploration.

First, the `Context` abstraction extends input datasets by adding flexible indexing methods for key-based point lookups and/or vector-based search. Then, we add the `Tool` abstraction, which supports custom tools beyond existing semantic operators. To tie these together, we introduce two logical operators that let agents perform reasoning and data retrieval: `compute` and `search`. Each operator takes a `Context` as input, but they have slightly different semantics: the `compute` operator seeks to generate a specific output, whereas the `search` operator tries to find information that can be used to enrich a `Context`'s description. These operators are implemented with coding agents that plan, write code, and use tools to execute the instruction.

With these operators, Palimpzest and Abacus still compile and optimize logical plans defined in terms of semantic operators, but that plan may also include agentic retrieval and computation methods. As an example, the program shown on the right of Figure 1 creates an initial `Context` object, which includes a natural-language description of the data along with support for indexing and tool use. The `search` operator takes the `Context` as input, and uses its description, tools, and data access methods to search for information on identity thefts.

Through these primitives, we extend the expressiveness of semantic programs, allowing users to define query intent using higher-level abstractions. At the same time, since these operators are defined in logical query plans provided to Abacus, they can be optimized to improve their performance based on user preferences.

3 Semantic Query Optimization using Abacus

Semantic query optimization begins once a declarative program has been compiled into a logical plan but still has many possible physical implementations. Consider again the identity-theft query from Section 1: an analyst asks for the ratio of identity theft reports in 2024 compared to 2001 over a mixed collection of consumer-report files. A possible semantic plan for this workload scans the files, applies a first filter to keep files containing statistics about identity theft reports, applies a second filter to keep evidence for the two requested years, joins the surviving evidence with year-specific report records or table entries, and finally uses a map to compute the requested ratio from the joined values. The logical plan contains two filters, a join, and a map, as shown in Figure 2.

The optimizer's goal is to compile this semantic operator program to a physical plan that is near-optimal under the cost model for the developer's objective with respect to system quality, dollar cost, and latency [22]. If the objective is to maximize quality, the optimizer may select heavier models and more complex inference-time strategies. If the objective is to maximize quality while spending less than a fixed budget on the whole workload, the optimizer may select lighter-weight models, cheaper join strategies, or reduced-context implementations while accepting only a modest decrease in quality.

Formally, given a logical plan with M semantic operators and N physical implementations per operator, the space of possible physical plans is $O(N^M)$ before considering operator reorderings. Even modest values of M and N make exhaustive plan-level measurement infeasible. Abacus makes a simplifying assumption: operators are independent, and each plan can be modeled as a function of

```python
import palimpzest as pz

# write PZ program for query
ds = pz.Dataset("kramabench/legal/")
ds1 = ds.filter("2024 identity thefts")
ds2 = ds.filter("2001 identity thefts")
ds3 = ds1.join(ds2).map("ratio")

# run optimized program
out = ds3.optimize_and_run()
print(out.to_df())
```

Figure 2: A semantic query plan for the KramaBench identity theft query implemented within Palimpzest. The final call to `optimize_and_run` invokes the Abacus optimizer.

its operators. For a plan with operator quality, cost, and latency estimates $\hat{o}_{qi}$, $\hat{o}_{ci}$, and $\hat{o}_{li}$, Abacus estimates plan quality, cost, and latency as follows:

$$\hat{p}_q = \prod_{i=1}^{M} \hat{o}_{qi} \qquad \hat{p}_c = \sum_{i=1}^{M} \hat{o}_{ci} \qquad \hat{p}_l = \max_{\text{path} \in p} \sum_{i \in \text{path}} \hat{o}_{li}. \tag{1}$$

In this model, we treat plan cost as the sum of operator costs, latency as the maximum-latency path through the semantic operator graph and quality as the product of operator qualities, assuming each $\hat{o}_{qi}$ lies in $[0, 1]$. Although this product form may not be a perfect semantic model of accuracy, it is useful for the purposes of optimization. Its useful property is monotonicity: replacing an individual operator with a higher-quality operator improves estimated plan quality, all else equal. That property lets the optimizer compare different physical operator choices locally while still scoring complete plans.

The estimates produced by Abacus are useful for plan search, but they should not be considered guarantees. Several assumptions and failure modes matter for interpreting the selected plan:

Operator independence: The independence of per-operator estimates can fail when operator errors are correlated. For example, an upstream filter may change the distribution of examples seen downstream in a non-uniform fashion, or the output quality of a map operator can make later extraction or join easier or harder.

Validation labels: The estimation of operator quality is strongest when users provide validation labels that match the deployment workload. When labels are unavailable, LLM-as-judge estimates can reduce annotation cost but may introduce judge bias, prompt sensitivity, and variance on partially correct outputs.

Sampling priors: The use of priors can bootstrap the bandit sampler to avoid clearly dominated operators. However, under small sample budgets, the prior distribution must closely match the deployment inputs. Otherwise, having weak or wrong priors can push sampling toward the wrong part of the frontier.

Semantic joins: Semantic joins require two estimates: the optimizer must estimate both the quality of the pairwise predicate and the size and quality of the candidate-pair set. An exhaustive LLM-based join can preserve recall by checking many pairs, but its cost and latency grow with the product of the input sizes; cheaper blocking or embedding-based approximations reduce this candidate set, but any pruned true match becomes an unrecoverable recall error for the downstream plan.

To obtain estimates for each physical operator, Abacus samples K physical implementations for each of the M logical operators, obtaining $K \cdot M$ operator estimates. Those estimates let it model $O(K^M)$ physical plans, including plans that have never been executed end to end.

3.1 Multi-armed Bandit Sampling

To maximize information gain from sampling, Abacus frames physical-operator sampling as a multi-armed bandit problem. Each physical implementation is an arm. Sampling an arm means running that operator on a subset of data inputs. We either use validation data with ground truth labels provided by users, or sample random inputs and observe quality, cost, and latency by using an LLM-as-a-judge estimation for the quality.

First, the optimizer starts with a small frontier of physical operators for each logical operator. If prior beliefs are available, it samples operators believed to lie near the Pareto frontier of the optimization objective. Otherwise, it samples operators at random. After each batch of observations, it updates the cost model, removes operators that are unlikely to remain useful, and samples replacements from the unsampled reservoir.

Since the optimizer is searching for an estimated Pareto frontier, it must consider the trade-offs between different objectives. The optimizer must therefore preserve plausible operator implementations rather than prematurely discarding everything except the current highest-estimated-quality implementation.

Abacus modifies the usual upper-confidence-bound bandit logic for this frontier search. For each operator and each metric relevant to the objective, it computes a sample mean, an upper confidence bound, and a lower confidence bound. The confidence interval is wider when the operator has been sampled fewer times. The optimizer then computes the current estimated Pareto-optimal operators according to mean performance. For every operator in the frontier, it checks whether the operator's upper confidence bound overlaps with the lower confidence bound of at least one operator on the current Pareto frontier. If the intervals overlap, the operator may still be estimated Pareto-optimal, so the optimizer keeps sampling it. If no such overlap exists, the operator is removed and replaced.

3.2 Pareto-Cascades for Plan Selection

Once sampling ends, Abacus must construct a final physical plan. At this point it has estimates for the average quality, cost, and latency of the sampled physical operators. The remaining problem is to choose a single implementation for each operator, and possibly among valid plan transformations, so that the final plan satisfies the objective most closely under the optimizer's estimates.

We extend the traditional Cascades optimization to handle multiple objectives. Traditional Cascades optimization organizes equivalent expressions into groups and uses dynamic programming to choose the lowest-estimated-cost physical expression for each group. This works cleanly for a single objective such as minimum estimated cost. It is insufficient for constrained optimization because a subplan that is preferred on one estimated dimension may not belong to the estimated feasible full plan selected by the optimizer. For example, the cheapest subplan may destroy too much quality while the most expensive subplan may not be able to meet the required cost.

We design an extension to the traditional Cascades [30] algorithm to address this issue within Abacus. Under the independence assumptions of the cost model in Equation 1, every subplan of an estimated Pareto-optimal physical plan is itself estimated Pareto-optimal. Rather than a single locally preferred expression, each transformation group maintains an estimated Pareto frontier of physical expressions. When optimizing a physical expression, the algorithm composes it with each estimated Pareto-optimal expression from its input groups and keeps the non-dominated alternatives. After the search finishes, it recursively constructs the estimated Pareto-optimal full plans and selects the one that satisfies the user's objective. For unconstrained optimization, Pareto-Cascades reduces to the traditional single-objective Cascades behavior. For constrained optimization, it avoids the greedy mistake of committing too early to a subplan that looks locally preferred under the current estimate.

Table 2: Abacus results on BioDEX and CUAD when optimizing for maximum quality. Quality is measured using RP@K for BioDEX and F1 score for CUAD. Mean values and standard deviations are adapted from the Abacus evaluation.

Workload	System	Quality	Opt. Cost ($)	Total Cost ($)	Latency (s)
BioDEX	DocETL	0.193 ± 0.032	3.50 ± 3.04	6.54 ± 5.53	$1,435 \pm 238$
	LOTUS	0.216 ± 0.042	–	18.9 ± 12.8	$2,348 \pm 1,489$
	Abacus	$\mathbf{0.260 \pm 0.015}$	0.146 ± 0.018	0.80 ± 0.256	557 ± 34
CUAD	DocETL	0.475 ± 0.106	6.04 ± 2.52	7.05 ± 2.63	$1,820 \pm 594$
	LOTUS	0.234 ± 0.005	–	0.196 ± 0.015	125 ± 19
	Abacus	$\mathbf{0.564 \pm 0.028}$	0.144 ± 0.024	0.506 ± 0.097	462 ± 64

4 Experimental Insights

In this section, we report experimental findings from the evaluation of our semantic data system. Readers interested in full experimental protocols, implementation details, and additional ablations should consult the full research papers [9, 11, 21, 22].

4.1 Workloads

When evaluating the performance of our semantic data system, we consider four distinct workloads that cover different aspects of the design space.

- **BioDEX** is an extreme multi-label classification task from the biomedical domain. Each input document describes adverse reactions experienced by a patient in response to taking a drug, and the system must produce a ranked list of adverse reaction labels from a set of 24,312 possible labels [3].
- **CUAD** is a legal contract understanding benchmark. Each input is a contract, and the task is to predict spans for 41 contract clauses; if a clause does not appear in the contract, the system should return null [8].
- **KramaBench** evaluates data-to-insight pipelines over data lakes; the legal workload used by Carnot contains 132 CSV and HTML files with statistics on fraud, identity theft, and other consumer reports [9].
- **SemBench** targets semantic query processing engines over multimodal data, with five scenarios and 55 queries spanning text, image, and audio modalities, and operators such as semantic filters, joins, maps, rankings, and classifications [11].

4.2 Optimizing Semantic Queries for Maximum Quality

The BioDEX and CUAD results in Table 2 show that the Abacus optimizer exploits the quality-cost-latency frontier, obtaining higher quality often at lower costs and latency than the baseline systems [19, 26]. On BioDEX, Abacus achieved higher mean quality than DocETL and LOTUS while also reducing total cost and latency relative to the next best quality-focused system. On CUAD, the LOTUS system provided a cheaper and faster implementation at a lower quality point, due to a simpler implementation of the semantic map operator. These results illustrate the trade-off between different user constraints:

Table 3: Carnot-style `compute` execution on the KramaBench identity-theft query. The query asks for the ratio of identity theft reports in 2024 versus 2001.

System	Percentage Error	Cost ($)	Time (s)
Palimpzest semantic operators	17.00%	1.66	215.2
Carnot `search` + `compute`	**0.02**%	1.17	583.0
Coding agent	27.56%	0.03	77.0

the highest-quality plan may not always be the cheapest plan, while a cheaper plan may lose too much semantic accuracy to be useful.

Insights from datasets BioDEX benefits from reduced-context generation because large portions of the medical report may be irrelevant to the adverse-reaction labels. The reduced context implementation of the map operator first chunks and embeds the input, selects the most relevant chunks, and then sends a smaller context to the LLM. This can reduce token processing while also focusing the model on the relevant evidence. For CUAD, the optimizer selects a mixture-of-agents map that aggregates answers from multiple model calls. These results highlight how physical operator choice is often workload-dependent, and the optimizer uses sampling measurements from the actual task to make a decision.

In Abacus, the use of prior beliefs improves optimization when the sample budget is small, especially in constrained settings where the optimizer must identify a frontier rather than one estimated winning operator. Pareto-Cascades improves constraint satisfaction because a locally preferred subplan may not belong to the estimated feasible global plan selected by the optimizer. These results support our central claim that, to be effective, semantic query optimization needs both effective sampling of operator implementations and a Pareto-aware plan selection strategy.

4.3 Optimizing Analytical and Reasoning Queries

The experiments on the KramaBench identity-theft query expose the importance of the `compute` and `Context` primitives for semantic optimization of analytic workloads. In this experiment, we compare a program using semantic operators with one leveraging the agentic primitives offered by Carnot. In our experiments, the semantic operator program computed the correct ratio in all three trials, but in two trials it also computed a second ratio because an errant file passed through one of the semantic filters. The result is a low average percent error only when the system finds the right file and does not mix in misleading evidence.

However, in the program leveraging agentic `search`, agents first searched for the right evidence, then used the `compute` operator to write optimized Palimpzest programs for the relevant data, and finally used Python to compute the ratio. This hybrid execution supports optimization over a more structured pipeline, where the runtime must decide when to search, when to invoke an optimized semantic plan, when to use code, and when to reuse a materialized context.

Finally, in our experiments with baseline coding agents, the planning/execution flexibility proved useful for interactive analytics, but with brittle execution. While capable of listing files, writing Python code, and dynamically adapting their plans, the agent often struggled to find the correct file and returned spurious ratios computed from non-ground-truth files. Its execution proved cheaper and faster, but with less accuracy.

These experiments show that implementing primitives to serve agentic operations is necessary to bridge the gap from semantic query optimization to deep research-style analytics.

Table 4: Palimpzest results on SemBench across all domains, grouped by semantic operator.

Operator group	Quality	Latency (s)	Cost ($)
Filter	0.777	72.2	0.40
Join	**0.703**	454.3	3.29
Map	0.762	89.3	0.09
Score	0.600	22.4	0.21
Classify	0.733	186.6	1.30
All operators	0.715	165.0	1.06
Filter and join only	0.740	263.2	1.84

4.4 Optimizing for Large-scale Processing

In running the SemBench workloads, we evaluated Palimpzest with a `gemini-2.5-flash` model, using a maximum-quality policy. Since query workloads in SemBench are mostly composed of single or homogeneous semantic operators, Table 4 shows the resulting behavior by operator type. Palimpzest is often able to reach a high output quality. Across all operator groups, it obtains an average quality of 0.715 at $1.06 and 165 seconds per query. Semantic joins expose the clearest quality-cost trade-off. On join queries, Palimpzest obtains the highest aggregate join quality among the evaluated systems, 0.703, but it also pays the highest aggregate join cost, $3.29. This happens because its join implementation evaluates candidate pairs in a nested-loop style with LLM calls. This result illustrates why joins stress the optimizer more than maps or filters: the system must choose both how to generate candidate pairs and how carefully to evaluate each pair. Exhaustive comparison spends more LLM calls to preserve recall, while blocking or embedding-based joins can reduce cost only by pruning pairs, which turns candidate-generation mistakes into missed matches. Compared to other systems using vector embedding-based approximation algorithms for joins [19], our implementation preserves recall, but it inherits the quadratic growth of candidate pairs. The result is exactly the kind of frontier Abacus exposes to users: choosing a minimum-cost join will trade off accuracy, while a maximum quality optimization will likely incur high costs as data sizes grow.

Overall, our results on SemBench show how often sample-based optimization becomes fragile when the physical search space is huge and the priors are weak. In future research, our goal is to guide sample-based optimization with better priors and estimates to address the challenges of large-scale datasets.

5 Future Outlook

In this paper, we first described how, through semantic operators within Palimpzest, natural-language tasks over documents, tables, images, and other objects can be expressed as logical query plans. We then described the Abacus optimizer which provides estimated near-optimal physical implementations under its cost model for these plans by sampling the estimated Pareto frontier of physical operators based on their quality, cost, and latency, leading to an estimated Pareto-optimal plan that satisfies user constraints. We also discussed how to extend this view toward Deep Research-style analytics, where agents can search, compute, write code, and invoke optimized semantic programs. Finally, we provided experimental evidence from different workloads to evaluate our approach.

The field of semantic query optimization still has open research challenges. Exploiting the separation between logical query planning and physical execution, future systems need to address the challenges of

designing logical query plans for large-scale multimodal workloads with user intent specified not through programming primitives but through natural language inputs. A central goal is to keep the usability of natural-language interfaces while preserving the optimization opportunities of declarative plans.

Acknowledgements

We are grateful for the support from the DARPA ASKEM Award HR00112220042, the ARPA-H Biomedical Data Fabric project, NSF DBI 2327954, a grant from Liberty Mutual, and the Amazon Research Award. Additionally, our work has been supported by contributions from Amazon, Google, and Intel as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, along with NSF IIS 1900933. This research was sponsored by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] E. Anderson, J. Fritz, A. Lee, B. Li, M. Lindblad, H. Lindeman, A. Meyer, P. Parmar, T. Ranade, M. A. Shah, B. Sowell, D. Tecuci, V. Thapliyal, and M. Welsh. The design of an llm-powered unstructured analytics system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2025.

[2] G. Corallo, E. Faure-Rolland, M. Lamari, and P. Papotti. TableKV: KV Cache Compression for In-Context Table Processing. In *Proceedings of the 4th Table Representation Learning Workshop*, pages 166–171, 2025.

[3] 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, 2023.

[4] S. Fernandes and J. Bernardino. What is bigquery? In B. C. Desai and M. Toyama, editors, *Proceedings of the 19th International Database Engineering & Applications Symposium, Yokohama, Japan, July 13-15, 2015*, pages 202–203. ACM, 2015.

[5] Google. Gemini deep research, 2025.

[6] G. Graefe. The cascades framework for query optimization. *IEEE Data(base) Engineering Bulletin*, 18:19–29, 1995.

[7] S. Guo, C. Deng, Y. Wen, H. Chen, Y. Chang, and J. Wang. Ds-agent: automated data science by empowering large language models with case-based reasoning. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[8] D. Hendrycks, C. Burns, A. Chen, and S. Ball. Cuad: An expert-annotated nlp dataset for legal contract review. *NeurIPS*, 2021.

[9] E. Lai, G. Vitagliano, Z. Zhang, O. Chabra, S. Sudhir, A. Zeng, A. A. Zabreyko, C. Li, F. Kossmann, J. Ding, J. Chen, M. Markakis, M. Russo, W. Wang, Z. Wu, M. Cafarella, L. Cao, S. Madden, and T. Kraska. KRAMABENCH: A benchmark for AI systems on data-to-insight pipelines over data lakes. In *The Fourteenth International Conference on Learning Representations*, 2026.

[10] LangChain. Open deep research, 2025.

[11] J. Lao, A. Zimmerer, O. Ovcharenko, T. Cong, M. Russo, G. Vitagliano, M. Cochez, F. Özcan, G. Gupta, T. Hottelier, H. V. Jagadish, K. Kissel, S. Schelter, A. Kipf, and I. Trummer. Sembench: A benchmark for semantic query processing engines. *PVLDB*, 19(8):1754–1767, 2026.

[12] 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, 2025.

[13] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, R. Shahout, et al. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2025.

[14] 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. ACM, 2025.

[15] D. Lu, S. Feng, J. Zhou, F. Solleza, M. Schwarzkopf, and U. Çetintemel. Vectraflow: Integrating vectors into stream processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2025.

[16] OpenAI. Deep research system card, 2025.

[17] M. Z. Pan, M. Cemri, L. A. Agrawal, S. Yang, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, K. Ramchandran, D. Klein, J. E. Gonzalez, M. Zaharia, and I. Stoica. Why Do Multiagent Systems Fail? In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*, 2025.

[18] L. Patel, S. Jha, M. Pan, H. Gupta, P. Asawa, C. Guestrin, and M. Zaharia. Semantic operators: A declarative model for rich, ai-based data processing, 2025.

[19] 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. *PVLDB*, 18(11):4171–4184, 2025.

[20] Perplexity. Introducing perplexity deep research, 2025.

[21] M. Russo and T. Kraska. Deep Research is the New Analytics System:Towards Building the Runtime for AI-Driven Analytics. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, 2026.

[22] M. Russo, S. Sudhir, G. Vitagliano, C. Liu, T. Kraska, S. Madden, and M. Cafarella. Abacus: A cost-based optimizer for semantic operator systems, 2025.

[23] J. Saad-Falcon, A. G. Lafuente, S. Natarajan, N. Maru, H. Todorov, E. Guha, E. K. Buchanan, M. Chen, N. Guha, C. Ré, and A. Mirhoseini. Archon: An architecture search framework for inference-time techniques, 2024.

[24] 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(3):1–28, 2025.

[25] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools, 2023.

[26] S. Shankar, T. Chambers, T. Shah, A. G. Parameswaran, and E. Wu. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *PVLDB*, 18(9):3035–3048, 2025.

[27] SmolAgents. Introducing smolagents, a simple library to build agents, 2024.

[28] SmolAgents. Open-source deepresearch â€“ freeing our search agents, 2025.

[29] M. Urban and C. Binnig. CAESURA: Language Models as Multi-Modal Query Planners. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024.

[30] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I, 2001.

[31] J. Wang, J. Wang, B. Athiwaratkun, C. Zhang, and J. Zou. Mixture-of-Agents Enhances Large Language Model Capabilities. *The Thirteenth International Conference on Learning Representations*, 2025.

[32] X. Wang, P. B. Chen, G. Vitagliano, M. Russo, J. Chen, M. Cafarella, S. Madden, and C. Liu. SAGE: Selective Attention-Guided Extraction for Token-Efficient Document Indexing, 2026.

[33] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pages 24824–24837, Red Hook, NY, USA, Nov. 2022. Curran Associates Inc.

[34] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

What Do Users Actually Do with LLM-Powered Data Systems?

Shreya Shankar and Aditya G. Parameswaran
UC Berkeley
{shreyashankar, adityagp}@berkeley.edu

Abstract

Large language models (LLMs) are now widely integrated into many database management systems. Users can now define relational operators in natural language and query unstructured data at scale, with the database using LLMs under the hood. Our community has devised many techniques to make such LLM-powered queries cheaper and more accurate, but all of our work assumes the user arrives with a well-specified query and ground-truth labels (or a labeling function). Through building and deploying DocETL, an open-source LLM-powered data analysis system, and DocWrangler, an IDE on top of it, we found that this assumption rarely holds. Drawing on 1,173 queries from an 8-month deployment and a long-term collaboration with public defenders on the California Racial Justice Act, we examine how users actually use LLM-powered data systems: how they struggle to specify and validate queries, how they iterate extensively to converge on bespoke operators specific to their documents and domains, and what these patterns imply for the systems and benchmarks our community should be building.

1 Introduction

Large language models (LLMs) have upended data management. For the first time, users can query unstructured data like long-form documents and images right inside their databases, at scale and in genuinely complex ways. Many major data systems now let users define relational operators (e.g., filter, map, join) in natural language and have the database invoke LLMs to execute them. For example, Snowflake Cortex exposes operators like `AI_FILTER`, `AI_CLASSIFY`, `AI_COMPLETE`, `AI_JOIN`, and `AI_AGG` [1, 2]; BigQuery offers `AI.GENERATE_TABLE` and `ML.GENERATE_TEXT` [3]; Google AlloyDB provides a similar suite [4]. We will refer to the interface as "AI-X"—e.g., AI-SQL, AI-MapReduce, AI-Pandas—from hereon out.

The broad goal of AI-X is to let users *simply* specify what they want in natural language, declaratively, and leave all choices about how to execute the query to the database. For example, suppose a user has a table of customer reviews, with schema `reviews(id, review_txt)`, and wants to find the reviews from upset customers. An AI-SQL query for this might look like `SELECT * FROM reviews WHERE AI_FILTER('the customer is upset', review_txt)`. One (very naive) query plan might, for each row in the table, issue a call to an LLM with a prompt containing the review text and the question "is this customer upset?", and drop any row where the answer comes back negative. The database could choose to execute this plan with a top-of-the-line LLM that scores well on AI benchmarks, e.g., GPT-5.5, but even at modest scale the cost is prohibitively expensive. A frontier model like GPT-5.5 charges roughly $1.25 per million input tokens, so a two-million-row `AI_FILTER` over ∼300-token reviews costs on the order of $750 per query (before accounting for any output tokens or retries). Moreover, cost is only part of the problem. The user may not have expressed their query precisely enough for the LLM to understand what "upset" means in the context of their reviews. Or the LLM might do well on some rows and poorly on others, and the user has no easy way to tell which.

Since there is no guarantee that a naive plan produces acceptable costs or accuracies, our community has devised a number of techniques to optimize AI-X queries. Some high-level ideas include ensembling multiple models to push accuracy up [5, 6]; routing "easy" rows to cheaper models [7, 8]; batching LLM calls or reusing prefix context across rows [9–11]; using LLMs themselves to rewrite queries into cheaper or more accurate plans [12, 13]; applying cost-based optimization to choose among these alternatives [14–16]; and many more. However, *nearly all of these techniques assume the user shows up with both: a well-specified query, and either ground-truth labels or a trusted "oracle" plan that they believe is accurate enough.* In practice, these assumptions are really hard to satisfy, and the upfront cost of getting started makes our systems inaccessible even to expert users.

In this article, we take a step back from query optimization and focus on the people actually using these systems. Over the last few years, while we have written a number of papers on optimizing LLM-powered data systems [12, 17–22], we have also deployed our ideas both "wide" (open-source, across a broad user base) and "deep" (in sustained, application-specific collaborations, including a multi-year partnership with public defenders on motions filed under the California Racial Justice Act [23] and work analyzing police misconduct records with the Berkeley CLEAN initiative [24]). Through this work, we have come to believe that *our community lacks an understanding of what people actually want to use LLM-powered data systems for, and how they actually use them.* We offer what we believe is the first user-centered perspective on LLM-powered data systems, on our deployments to examine where users struggle and working back from those observations to the systems and benchmarks we should be building. Concretely:

- In Section 2, we describe the top user-facing challenges in detail, drawing on our early engagements with users of DocETL [12], an open-source LLM-powered unstructured data analysis system that we built.
- In Section 3, we report our experience deploying these systems to a much larger user base via DocWrangler [25], an IDE we built on top of DocETL. Drawing on 1,173 queries from an 8-month deployment, we report what users actually build: queries grow more specific through revision, the operators inside them are bespoke to a user's documents rather than off-the-shelf NLP tasks, and users frequently add LLM-powered operators whose only job is to help author or validate other operators.
- In Section 4, given what we now know about how users actually write AI-X queries, we reflect on the benchmarks our community evaluates on (including the ones we ourselves used in DocETL) and argue that they are not a good proxy for queries written in deployment.
- In Section 5, we offer ideas for future work: designing for revision rather than single execution (reusing compute, outputs, and optimizer estimates across revisions), pushing validation-oriented parts of workloads into the data system itself (rather than an IDE), and building execution strategies and benchmarks that match the bespoke operators users actually write.

2 Challenges users hit when authoring AI-X queries

We built DocETL [12], an open-source AI-X system for querying unstructured data (3.7k+ GitHub stars). Users write queries as pipelines of LLM-powered operators (map, filter, reduce, resolve, etc.), each parameterized by a natural-language prompt and a user-defined output *schema* of typed attributes. DocETL is written in Python, and the query DSL is YAML, designed to be accessible to non-technical users and, increasingly, AI agents.

Through deploying DocETL to a broad open-source user base, we observed several recurring challenges that users hit when authoring AI-X queries. Traditional query processing assumes the user knows what they want and the system executes it correctly; LLM-powered operators break both assumptions. Even when users understand the DSL, they struggle to specify their intent in a prompt, and even when they do, the system may not execute it correctly. We describe these challenges below, grounding the discussion in Example 1, our multi-year collaboration with public defender offices on the California Racial Justice Act.

Example 1 (Racial Justice Act evidence search) *The California Racial Justice Act (RJA), codified at Cal. Penal Code § 745 [23], makes it unlawful for the State to seek or obtain a conviction or sentence on the basis of race, ethnicity, or national origin. The law grants relief if a judge, attorney, juror, expert witness, or law-enforcement officer involved in the case "exhibited bias or animus" or "used racially discriminatory language" during trial. AB 256 (2022) extends § 745 retroactively, so it even applies to judgments entered before the law was passed.*

To file a § 745 motion, defense counsel has to point a judge at specific passages in the record that meet the statutory bar, and reading the record by hand is the slow step. A single client's file can run 3,000 to 5,000 pages spanning trial transcripts, police reports, RAP sheets, news articles, and more, and an attorney drafting one motion reads all of it to flag a few dozen qualifying passages. The analysis is so hard that only about a dozen motions have succeeded statewide in the law's first four years [26]. We have worked on the task with post-conviction units at three California county public defender offices.

What does a public defender drafting a motion want from an LLM-powered data system? They want to issue many related queries against all the data associated with their clients, not a single one-shot extraction. For example:

- ***Extract.** Over a single client's record, return every passage that matches a specific category of bias (e.g., an officer's racial epithet on body-worn-camera audio, "urban" or "predator" framing in a prosecutor's closing, dehumanizing descriptions in a probation report), each tagged with the category and a one-sentence rationale quoting the offending language.*
- ***Filter.** Over an existing pool of extracted passages, keep only those that fit the legal theory of a particular motion (e.g., § 745(a)(1) bias by a state actor vs. § 745(a)(2) discriminatory language during trial).*
- ***Aggregate.** Over many clients' records, surface patterns (e.g., a single officer or prosecutor's office that recurs across cases, or a category of coded language that appears disproportionately for defendants of one race).*

We now describe two key user challenges in turn, drawing on the RJA task above.

Operator prompts are hard to revise. Writing the first draft of an AI-X operator is easy, since all the user has to write is natural language. However, iterating on the prompt is where users get stuck. Users often do not know what they want until they see outputs. For example, suppose the user wants an LLM-powered map to extract "themes" from product reviews. "Themes" could mean product attributes, customer emotions, reasons for returning a product, or feature requests, and the user often does not know the right granularity a priori. It may be easier to try a few definitions of "themes," see the outputs each one produces, and pick.

The RJA task from Example 1 has the same "know it when you see it" trait. Given a specific scenario,

a public defender can say whether it counts as racial bias, but writing a prompt that enumerates the kinds of bias to extract up front is much harder. For example, a first draft might ask for passages with explicit racial mentions (e.g., references to “black” or “urban”). After reading the outputs, the public defender realizes most of what the model surfaced are benign mentions of race that do not carry animus (e.g., a witness describing a suspect's appearance as “Black male, 5'10"'). The outputs also do not tell the public defender what the prompt *failed* to catch. The public defender has to page through the underlying documents themselves to notice language that is bias-laden only in context (e.g., an officer writing “these people” when referring to residents of a predominantly Black neighborhood. The phrase is not a racial slur, but it groups residents by an implicit “them,” and in context signals that the officer views the community as an outgroup). Overall, the user is simultaneously discovering what they want to ask for and learning what is actually in the data.

Once the public defender knows, implicitly, what they want to ask, they still have to express it in a prompt. A public defender talking to another public defender can lean on common grounding [27]: both sides already know the kinds of records, the statutory standard, and which framings tend to fly with which judges or district attorneys. An LLM has none of this shared context, so the public defender has to encode their domain knowledge, the legal standards (down to the precise sub-clauses of § 745), successful prior motions [26], and relevant examples (from the documents they want to query) directly into the prompt. In our experience, settling on the extraction categories took the public defenders more than ten rounds (more than two months) of running the query, reading the outputs, going back to the underlying source data, and revising the prompt.

The burden of validation falls on the user. Even if the user feels confident they have expressed their query correctly, they still cannot take the outputs for granted. Unlike SQL, which always returns the correct answer for a given query, AI-X queries do not. Prompts may be ambiguous, since natural language is imprecise (e.g., “passages that show racial bias” from Example 1 admits dozens of plausible definitions). And even when the prompt is unambiguous, the LLM carrying out the operator may still be error-prone and produce incorrect outputs [28, 29]. So the user has to read the outputs themselves to decide whether to trust them, which usually means opening a spreadsheet and painstakingly reading rows one by one against the source documents.

One solution is to have users externalize their correctness criteria, either as labels on a sample or as an LLM-judge prompt, and automatically score outputs. Each approach has drawbacks. Hand-labeling is slow because source documents are long, and in heavy domains like the RJA task (Example 1), it is emotionally taxing for readers to wade through slurs, dehumanizing descriptions, and graphic accounts of violence. LLM judges may seemingly sidestep the human labor cost but introduce their own problem: aligning a judge with the user is itself a hard prompt-engineering task, and off-the-shelf judges disagree with humans in ways that take deliberate effort to close [30, 31].

While the aforementioned challenges cause friction, they can be partially absorbed by the right tooling and interface around the data system, and partially mitigated by users once they find a workflow that fits their task. In the next section, we describe what we observed once we gave users a purpose-built integrated development environment (IDE) for writing DocETL queries.

3 How users tackle query authoring and validation challenges

Motivated to lower barriers for our users, we built DocWrangler [25], an IDE for DocETL that targets three “gulfs” of challenges users face: understanding the data, specifying the query, and validating that the LLM generalized correctly across the full corpus (Figure 1). Users author a query in a notebook-like pipeline builder where every cell is an LLM-powered operation (A). To help users understand the data,

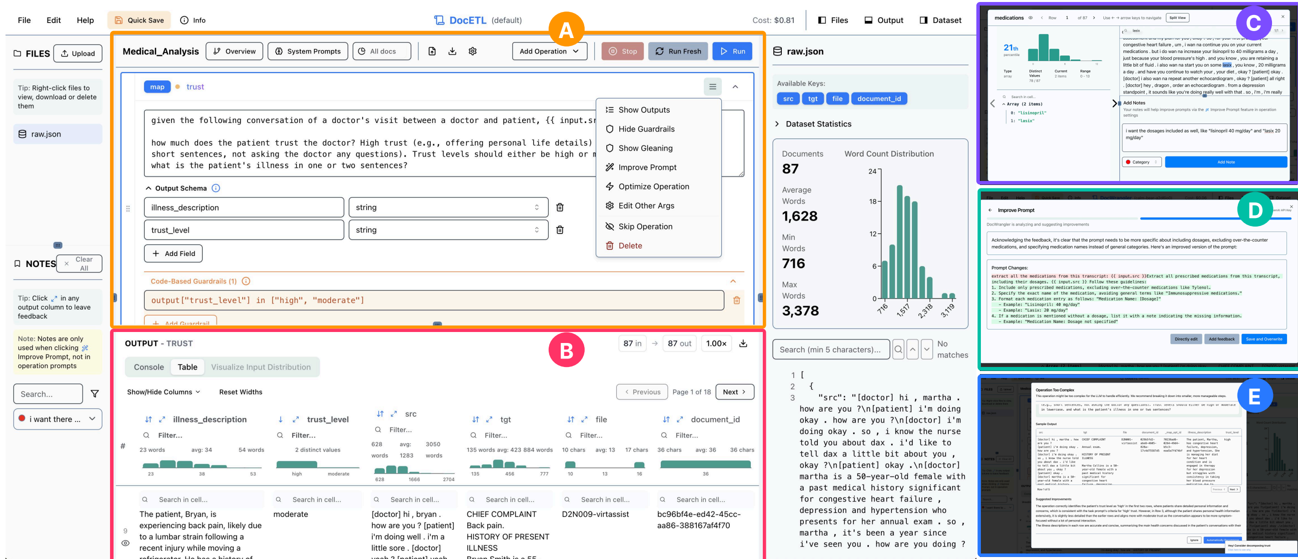

Figure 1: The DocWrangler IDE. A notebook-like pipeline builder where each cell is an LLM-powered operation; B spreadsheet-like output viewer showing each operator's outputs across documents; C per-cell view of the inputs and outputs that produced a value, with open-ended notes the user can pin to specific outputs; D built-in assistant that rewrites a draft prompt grounded in those notes and suggests decomposing a complex operator into smaller ones; E more expensive LLM judge that scores sample outputs and flags when the query could benefit from rewriting.

a spreadsheet-like output viewer (B) shows each operator's outputs alongside its inputs, and users can write open-ended notes pinned to specific output cells (C). To help users specify queries, a built-in assistant rewrites a draft prompt grounded in those notes and suggests decomposing a complex operator into smaller ones (D). To help users validate query outputs, a more expensive LLM judge runs on sample outputs and proactively flags when the query could benefit from rewriting (E).

We studied DocWrangler through an IRB-approved lab study with 10 participants (software and ML engineers, data scientists, and domain experts) and an 8-month public deployment open to anyone; the patterns below come from the deployment. We define a *query* as a saved DocETL pipeline and a *revision* as any edit that changed an operator's prompt, type, or schema. After cleaning and deduplicating the logged queries and their edit histories, we were left with 1,173 queries, and report only aggregate statistics. Every prompt we show is anonymized, with dataset names, organizations, languages, and identifying phrases redacted.

Users write LLM-powered operators to help author and validate other operators. Users preferred to build a query one operator at a time, running each operator and reading its outputs before adding the next, rather than writing the whole query up front and drilling into intermediates afterward. But reading every output row by hand is slow, so some users found a shortcut: writing additional LLM-powered operators to offload work they would otherwise do manually.

When users felt stuck on a prompt because they did not know how to express what they wanted, we might expect them to read through example documents for inspiration. Instead, some users wrote a simpler LLM-powered operator to "survey" the data for them. For example, an attorney who needs to extract categories of racial bias from police reports but does not know which categories actually show up in the corpus might first write an operator that asks the model to "summarize the kinds of biased language present in each report." After reading through the summaries, the attorney can see concrete

examples of the kinds of bias that appear in the corpus, copy the most relevant ones into the original operator's prompt as the categories to extract, and delete the summary operator.

Similarly, we might expect users to read every output row to check correctness. Instead, some wrote additional LLM-powered operators that acted as judges, where a separate model would score each upstream output or check whether the answer was supported by the source document. The user could then look at the distribution of scores in the output viewer's histograms (B) to decide which outputs to spot-check. For cheaper, deterministic checks, users also attached code-based guardrails to individual operators, small Python predicates DocETL runs on each output (e.g., asserting an extracted date parses, a quote actually appears in the source document). Roughly 2% of queries contained a validation-oriented operator, though this is a lower bound, since we detect these from the final query revision and users often delete such operators once they have served their purpose. Overall, roughly 20% of all operators created were eventually deleted, suggesting many were written for exploration or validation rather than to stay in the final query.

Queries are bespoke and grow more specific through revision. No user in our deployment "one-shot" a working LLM-powered query. The bottleneck is not syntax; users do not need to know DocETL syntax to use DocWrangler, since the IDE exposes each operator with an intuitive form-like UI. Users iterate or revise their queries because they discover what they truly want by running the query and reading the outputs. Queries went through a median of 5 revisions (mean 8.7, 90th percentile 21, max 82) before the user was satisfied. Total prompt lengths also grow over the course of iteration (median 704 characters, 90th percentile 3,187, 95th percentile 4,408) as users add more concrete definitions and clarify edge cases. The number of LLM-powered operators per query spans a wide range: the median query has just 1 (mean 1.79), but the 90th percentile has 3, the 95th has 4, and the largest query has 36 LLM-powered operators.

What are users' operators actually doing? Most do not perform tasks that an existing model on a platform like HuggingFace could already handle (e.g., sentiment analysis). They are specific to the user's documents and questions, to the point that an LLM prompted with a custom instruction is the only practical way to execute the task. To measure how prevalent such "bespoke" operations are, we used GPT-5.4 to informally classify every LLM-powered operator in our deployment as either "off-the-shelf" or "bespoke." An off-the-shelf operator is one that could reasonably be performed by an existing pretrained model (e.g., sentiment analysis, topic classification, summarization, named entity recognition, generic question-answering). A bespoke operator is one whose categories, output schema, or domain context make it specific to the user's workflow. The GPT-5.4 classifier was given each operator's prompt, name, type, output fields, and the surrounding operators as context. We manually verified a sample of 20 classifications and agreed with 19 of them. Roughly 62% of operators were labeled bespoke, accounting for about two-thirds of all output attributes, and 47% of queries contained at least one bespoke operator. Table 1 shows a sample of these bespoke operators across many domains in our deployment.

All in all, our biggest lesson from the DocWrangler project was that users do not write LLM-powered queries the way they write SQL. Users explore the data, decompose tasks across multiple LLM-powered operators (including operators whose only job is to scope or validate other operators), and converge on bespoke operators (i.e., bespoke schemas) through many revisions.

4 What our benchmarks miss

Given what we learned from our deployment, we reflect on how LLM-powered data systems are evaluated in our community, including in our own work. Are the benchmarks we run representative of the queries and workloads real users author? And if not, what are the potential harms of optimizing systems and

Table 1: A sample of bespoke LLM-powered operators from our DocWrangler deployment, spanning multiple domains; each operator is one step in a larger user-authored query. At the surface, every one is a familiar NLP task—e.g., extraction, classification, or summarization—but the categories and output schemas are defined inside the prompt itself, so no off-the-shelf model could execute these tasks without the user's instructions.

Domain	**Excerpt from user's operator prompt**
Local climate policy	*"Classify each climate policy mentioned in the document as a **Measure** (a regulation or mandate, e.g., a building-efficiency standard), an **Instrument** (a financing or pricing mechanism, e.g., a carbon tax), or **Both**, with a one-sentence justification quoted from the document."*
EPA chemical safety	*"Determine whether the document qualifies as a Data Evaluation Record. Look for detailed data analysis, evaluation of chemical safety, and references to EPA regulatory guidelines."*
Medical-education flashcards	*"You are a medical-education grader. For each flashcard, decide **Relevant (1)** or **Not Relevant (0)** to the user's query passage. Relevant means the card directly answers, explains, or extends a factual claim, mechanism, complication, or treatment explicitly mentioned in the query; mere topical overlap is not relevant."*
Automotive head-unit program mgmt.	*"Analyze an automotive Head Unit program. Produce: top execution risks (cite `phase_ids`), supply-chain flags (cite `part_ids` and suppliers), key cost drivers, and a timeline assessment of phase overlaps and SOP feasibility."*
Battery materials science	*"Extract three categories of tests from the paper: symmetric cell tests (Li$\|$Li), critical current density (CCD) tests, and full-cell tests. For each, return electrode material, electrolyte, total cycle time, and current density with explicit units; set fields to `null` if not stated."*
Procurement / tender extraction	*"From a tender page, extract the procuring entity, tendering agency, bidding stage, winning bidders and amounts, budget/control price, and tender identifier. Quote the verbatim passage (in the source language, [redacted]) supporting each field as provenance; do not hallucinate amounts."*
Public health surveillance	*"Classify whether a surveillance record pertains to a chronic disease (cancer, CVD, diabetes, COPD/asthma, stroke, kidney disease, hypertension, obesity, chronic activity limitation). Infer a concise disease category, extract a race/ethnicity hint from the stratification fields, and cite which input fields drove the decision."*
Vendor contract risk review	*"You are reviewing software-vendor contract text from the customer's perspective. Identify passages that pertain to Data Ownership & Control: vendor claims of perpetual or irrevocable rights to customer content, ambiguous IP/licensing terms, or unclear customer-control rights. Return the verbatim excerpt and a reason it qualifies."*
Clinical lab biomarkers	*"Analyze the lab report and extract each biomarker mentioned. For each, return name (e.g., AST, Sodium), numerical value, unit of measurement (e.g., U/L, mmol/L), and reference range if available. Leave fields blank if missing rather than guessing."*
Book metadata	*"Extract title, subtitle, original title, category, audience, tone, author bio, and a structured 'coreIdea' (promise, problem, thesis, target outcome) from a book PDF rendered to markdown. Provide quoted provenance snippets (max $\sim$15 words) supporting each field and a 0–1 confidence score."*
Legal discovery contradictions	*"Extract atomic, comparable factual claims from each discovery document (date sent, case number, subject, sender, content assertions). Normalize each (ISO dates, uppercase doc numbers) and attach the verbatim provenance snippet and source field. Later, flag contradictions across documents in the same case."*
FAA aviation publications	*"Extract structured entities from a National Flight Data Digest page: airport identifiers, runway changes, NAVAID status, obstruction updates. For each entity, return the effective date, the affected facility code, and the verbatim FAA language describing the change."*
Special-education research coding	*"Identify types of students explicitly described in the paper (not inferred). Prefer fewer, well-supported categories; for each, capture context (e.g., math task, group work), observable behaviors (off-task, refusal), explicitly stated cognitive traits, and intervention used. Mark each evidence dimension present/absent and a 0–1 eligibility rate."*
Banker commitment audit ([language redacted])	*"You are reviewing a client meeting transcript in [language redacted]. Identify only commitments of future action made by a recognized banker ([redacted trigger phrases, e.g. 'I'll send', 'I'll prepare']), ignoring client promises and vague intentions. Record the action, assignee, any explicit deadline, and the verbatim quote with timestamp."*
Vendor-policy risk categorization	*"Categorize each excerpt of a software-vendor policy into one of 12 risk categories (Policy Gap, Data Use & Purpose Limitation, Cross-Border Transfers, Sub-processors, Operational Burden, Liability & Legal, . . .). Quote the multi-sentence excerpt that drove the decision and the reasoning behind the chosen category."*
[Asian country, redacted] $\rightarrow$ English govt. circulars	*"Translate this government circular from [source language redacted] to English—do not summarize. Preserve register, headers, and numbering; translate acronyms with the original in parentheses on first mention ([redacted local-language acronym] $\rightarrow$ [redacted English expansion]). Extract the circular's year and ID, and rate confidence High/Medium/Low."*
Investigatory link analysis	*"For one document in an investigatory database, classify it as email / legal filing / article / social-post / other, then extract people, organizations, case numbers, court names, dates, phone numbers, and attachments. Build relationship edges (`email_exchange`, `legal_opposition`, `mention`) with quoted provenance."*

writing papers against them?

Existing benchmarks don't represent bespoke tasks, and they are saturated. We informally surveyed 34 cs.DB- and cs.CL-primary arXiv papers on LLM-powered operators published between January 2024 and May 2026. About 92% of the datasets used in their evaluations are publicly downloadable, and about 85% of those are standard NLP benchmarks that were published with fixed tasks, labels, and train/test splits long before the current generation of frontier models. Nine datasets (AGNews, IMDB, DBpedia, CUAD, FEVER, BioDEX, BIRD, Enron, MiDe22) were each reused by three or more of the surveyed papers; the most popular among them draw hundreds of thousands of HuggingFace downloads per month (e.g., 172K/mo for IMDB, 107K/mo for AGNews). Even benchmarks designed specifically for LLM-powered data processing rely on similar data: SemBench [32], for example, draws from a Rotten Tomatoes movie review dataset for all of its text tasks, and every LLM-oriented task is a classification task. These benchmarks also appear saturated. We also ran a simple experiment to see how much headroom is left: we asked GPT-5.4-nano, the cheapest tier model, to classify every example in each dataset with no special prompting or optimization. It already scores 92.3% on IMDb-50K, 92.4% on DBpedia-14, and 86.9% on AGNews. If the cheapest model nearly maxes out accuracy, there is little room for an optimizer to demonstrate improvement.

Memorization could distort Pareto-optimal query plans. LLMs are trained on large swaths of the internet, and many of these benchmark datasets (along with their labels, discussions, and leaderboard entries) are widely represented across the web. It is plausible that LLMs have memorized at least some of the data underlying our benchmarks. If so, an LLM may be able to produce the right answer not because it understood the task, but because it has "seen" the example before. The query plans that look Pareto-optimal on those benchmarks may simply be exploiting memorization rather than reflecting what works on bespoke, unseen workloads users actually care about.

To probe whether memorization could affect benchmark accuracy, we run a small, suggestive experiment on two popular movie-review datasets: the Rotten Tomatoes dataset used in SemBench, and IMDB, one of the most reused datasets across the papers we surveyed (Figure 2). The setup has three conditions. In the *full review* condition, we run a sentiment-classification query using the complete review text. In the *without dataset name* condition, we replace each review with a short random snippet (10% of the original tokens, so for a typical 24-token Rotten Tomatoes review, the model sees roughly 2 words). In the *with dataset name* condition, we use the same short snippet but add the dataset's name to the prompt. Two words is not enough information to determine the sentiment of a full review, so if accuracy goes up when we add the dataset name, the model must be recalling the review from its training data. Interestingly, as shown in Figure 3, naming "Rotten Tomatoes" boosts GPT-5.4's accuracy from 50.1% to 58.8% on spans averaging only 2.4 tokens. IMDB shows the same pattern, with accuracy rising from 74.5% to 79.2%. Both gaps are statistically significant (95% confidence intervals shown in Figure 3).

This result, together with the saturation analysis above, is suggestive, not conclusive, but may be an early indicator that we need benchmarks built around bespoke tasks on data the model has not seen during training. But designing such benchmarks is hard. Designing good benchmarks for data systems is already difficult; designing benchmarks that are also robust to model memorization and training-data contamination adds a whole new dimension. But getting this right is essential if we want our systems to actually be useful to the people who use them.

Figure 2: Two versions of a query with an LLM-powered filter, applied to the same random 10% token span of a movie review. The left query names the dataset and tells the model the review was in its training data; the right query asks only to classify the sentiment of the visible text. `{snippet}` is replaced with the 10% token span and `{dataset_name}` with either "Rotten Tomatoes" or "IMDb 50K" at query time.

5 Takeaways and Future Directions

AI-X interfaces promise the best of both worlds: the expressiveness of natural language and the structure of a data querying language. From watching users, it is clear they also get the worst of both worlds: the ambiguity and iteration cost of prompt engineering, combined with the debugging difficulty of a complex, multi-operator query. The patterns we observed suggest how data systems should be designed differently to support LLM-powered query processing. We describe a few directions below.

Designing for revision, not single execution. LLM-powered data systems today plan and execute each query as if the user were asking the system to run it exactly once, even though queries in our deployment went through a median of five revisions before the user was satisfied. Each revision re-plans, re-optimizes, and re-executes from scratch, even though the next revision is usually a small edit to the previous one, and the system has already paid to run the previous version and learn which models and plans worked. The system could reuse work across revisions instead; several ideas follow.

One idea is to spend less compute on early revisions. When a user is still exploring, the outputs do not need to be fully accurate. They just need to be good enough for the user to learn from and steer their query. The system could run cheaper, less accurate plans in early revisions and reserve expensive, high-accuracy execution for when the user is satisfied with the query and wants final results. The key challenge would be to determine a "correct enough" threshold for early revisions, which may likely require new ways of modeling both the data (different tasks and datasets affect what counts as informative) and the user (how people learn from query outputs and decide what to change).

Another idea is to reuse outputs from earlier revisions. Obviously, if an operator in the new revision is identical to one in a previous revision, the system can cache and skip re-executing it. A more interesting question is whether outputs can be reused even when operators change. For example, suppose a map operator produced a summary of each police report covering the officers' names and any recorded statements. If the user's next revision edits the prompt so that the summary also covers the suspect's demographics, the system does not have to re-summarize every report from scratch. Instead, the system can keep the previous summaries, extract the suspect's demographics from each report, and edit each existing summary to include them. More generally, given query versions A and B, the system could use an LLM to describe how B differs from A, compute only the differing piece, and "fold" the result into A's outputs, a form of "semantic" incremental computation. The key challenge would be to build a cost model over semantic diffs: when has a revision changed enough that folding in the difference costs more

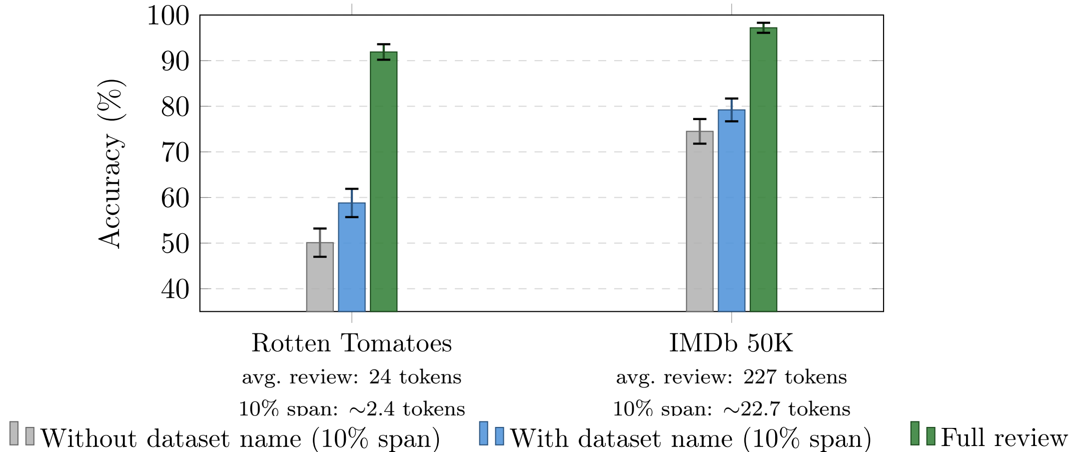

Figure 3: Results of the queries in Figure 2, where GPT-5.4 executes the LLM-powered filter row by row over 1,000 reviews per dataset. The gray and blue bars use only a random 10% token span of the review; the green bar uses the full review text. Error bars show 95% confidence intervals (20,000 bootstrap resamples). On Rotten Tomatoes, the 10% span averages just 2.4 tokens—roughly two words—which is far too little to determine the sentiment of a full review, yet naming the dataset boosts accuracy from 50.1% to 58.8%!

than re-executing from scratch?

A third idea is to reuse what the query optimizer learned from earlier revisions. Most query optimizers run candidate plans on samples to estimate each operator's accuracy and cost across different models [7, 12, 15, 16, 20], which can take upwards of an hour per query [16]. If a small, cheap model was tried for an operator and produced outputs that were not accurate enough, future revisions should not have to retry it. The key challenge would be to determine when a prompt revision is significant enough to invalidate earlier estimates, since a minor wording change might not affect which model is best, but a substantive revision to the task definition could.

Pushing validation into the data system. Users in our deployment wrote their own validation operators by hand, which suggests the query engine should support validation natively. Today, most systems only support type checks on operator outputs [1, 3, 12, 14] and generally do not return provenance from each output row back to the source span it came from. As a starting point, query plans could additionally run a more expensive LLM judge on a sample of each operator's outputs and surface the scores alongside the results. The system could also tailor its validation budget to the user's intent: an operator that asks the model to "find an interesting idea" needs less validation than one that asks it to "find all instances of" a specific category.

But even with scores and provenance, users still need to make sense of it all. Spreadsheets are painful for reading LLM outputs given the volume of text, and the interactive visualizations in DocWrangler (inspired by Wrangler and Trifacta [33]) are also difficult to scan. A promising direction we have been exploring is using coding agents (e.g., Claude Code Opus 4.6) to generate custom dashboards tailored to each query: if an operator extracts a list, the dashboard renders extractions as a list; if an operator produces a summary, it pins the summary alongside the source document; if there is provenance, it highlights the relevant spans in the source (Figure 4). In one deployment, a generated dashboard improved validation throughput from three documents per day with spreadsheets to ten per hour. However, dashboard generation is currently *expensive*, taking upwards of ten minutes with a coding

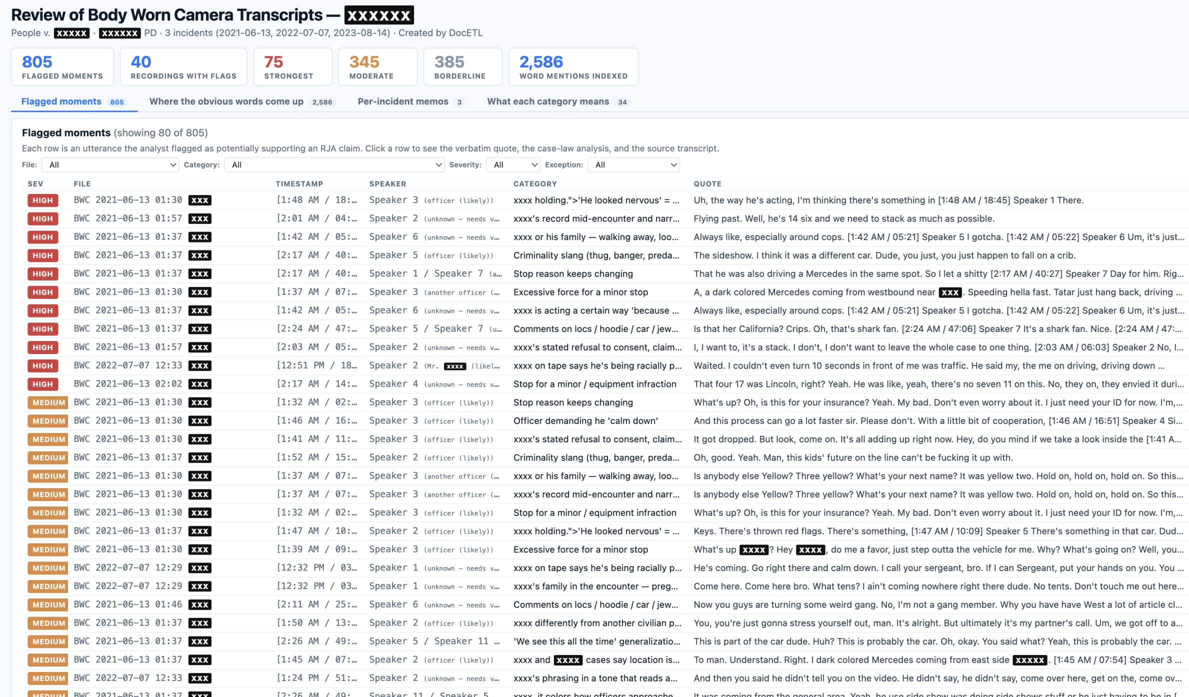

Figure 4: A custom dashboard generated for a deployment with public defenders reviewing body-worn camera transcripts for evidence of racial bias. Each row is an utterance the LLM extracted as potentially supporting a legal claim, shown with its severity, speaker, timestamp, claim category, and the quoted passage from the transcript. The layout is tailored to the query's operators and output schema.

agent and costing roughly $5 per dashboard. A more scalable direction would be to build reusable UI components for common patterns (e.g., lists for extractions, threaded views for conversations, collapsible layouts for large output schemas) and have the data system generate and compose them, with users able to provide feedback directly in the dashboard (e.g., correcting a label or flagging a bad output) that propagates back to the data system as revised prompts or constraints on future query plans.

Building systems for bespoke operators. The bespokeness of the operators users actually write has implications for how systems plan queries and are evaluated.

Even when bespoke operators resemble standard NLP tasks like extraction, classification, or summarization, they operate on custom data with domain-specific definitions, and different models turn out to be better or worse at different task types. Work from the ML community demonstrates that no single LLMs achieves the best accuracy on all possible tasks [34], and our prior work also corroborates that different models are Pareto-optimal for different types of data processing tasks [16]: for example, GPT-5 Nano is commonly selected for large-scale extraction, while more expensive models tend to be selected for summarization and report generation. Today, query optimizers discover which LLMs works best for which operator by trying multiple LLMs on a sample and comparing results, which is slow and expensive. Some prior work allows encoding priors about LLM acuracy [15], but how to develop such priors for LLM-powered data processing tasks remains unclear.

A deeper question is how pretraining and post-training impact LLM performance across different LLM-powered query processing strategies, such as how much of the document to include in the context window, whether to decompose an operator into sub-tasks, or how much prompting is needed. Prior work from the ML community has found that an LLM's accuracy on a question correlates with how

often relevant documents appeared in its pretraining data [35]. If a query optimizer knew what a model was trained on, it could make cheaper and better decisions: e.g., if the model encountered the documents during training, it may not need the full document in the context window, since the relevant content is already encoded in its parameters; if the model is already familiar with the task (e.g., sentiment classification), minimal prompting may suffice, and a smaller fine-tuned model may be equally accurate at a fraction of the cost. We do not yet have a principled way to detect what an LLM already knows about a given document or task.

Ultimately, all of the directions above share a common theme: the systems we build should be designed around how people actually use them. Users do not write a perfect query and run it once. They explore, revise, validate, and converge over many iterations, on tasks that are specific to their data and their domain. The closer our systems and benchmarks match that reality, the more useful they will be.

6 Conclusion

LLM-powered data systems have made it possible to query unstructured data at scale, but the people using them face challenges that our community has largely overlooked. Through building and deploying DocETL and DocWrangler, we found that users struggle to specify what they want, cannot easily tell whether to trust the outputs, and work around both problems through extensive iteration and creative use of LLM-powered operators. The queries they write are overwhelmingly bespoke, and our benchmarks do not reflect them. Designing systems around how people actually work, rather than how we assume they do, is the central challenge ahead. We hope the observations and directions in this article inspire our community build LLM-powered data systems that are not just powerful and scalable, but also accessible to the people who need them.

References

[1] Snowflake, “Snowflake cortex ai: Llm functions.” `https://docs.snowflake.com/en/user-guide/snowflake-cortex/llm-functions`, 2024. Accessed 2025.

[2] 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.” `https://arxiv.org/abs/2511.07663`, 2025.

[3] Google Cloud, “Bigquery ml: Generative ai functions.” `https://cloud.google.com/bigquery/docs/generative-ai-overview`, 2024. Accessed 2025.

[4] Google Cloud, “Alloydb ai: Work with generative ai.” `https://cloud.google.com/alloydb/docs/ai/work-with-generative-ai`, 2024. Accessed 2025.

[5] K. Huang, Y. Shi, D. Ding, Y. Li, Y. Fei, L. V. S. Lakshmanan, and X. Xiao, “Thriftllm: On cost-effective selection of large language models for classification queries,” *arXiv preprint arXiv:2501.04901*, 2025.

[6] S. Jo and I. Trummer, “Sparellm: Automatically selecting task-specific minimum-cost large language models under equivalence constraint,” *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, 2025.

[7] 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, no. 11, pp. 4171–4184, 2025.

[8] M. Urban and C. Binnig, “Efficient learned query execution over text and tables,” *arXiv preprint arXiv:2410.22522*, 2024.

[9] A. Mhedhbi *et al.*, "Beyond quacking: Deep integration of language models and rag into duckdb," *Proceedings of the VLDB Endowment*, vol. 18, pp. 5415–5428, 2025.

[10] G. Sanmartino, M. Urban, P. Papotti, and C. Binnig, "The stretto execution engine for llm-augmented data systems," *arXiv preprint arXiv:2602.04430*, 2026.

[11] S. Liu, A. Biswal, A. Kamsetty, A. Cheng, L. G. Schroeder, L. Patel, S. Cao, X. Mo, I. Stoica, J. E. Gonzalez, and M. Zaharia, "Optimizing llm queries in relational data analytics workloads," *arXiv preprint arXiv:2403.05821*, 2024.

[12] S. Shankar, T. Chambers, T. Shah, A. G. Parameswaran, and E. Wu, "Docetl: Agentic query rewriting and evaluation for complex document processing," *Proceedings of the VLDB Endowment*, vol. 18, no. 9, pp. 3035–3048, 2025.

[13] G. Xiao, E. Zhang, N. Sullivan, W. Hansen, and M. Balazinska, "Kathdb: Explainable multimodal database management system with human-ai collaboration," *arXiv preprint arXiv:2512.11067*, 2025.

[14] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, R. Shahout, *et al.*, "Palimpzest: Optimizing ai-powered analytics with declarative query processing," in *Proceedings of the Conference on Innovative Database Research (CIDR)*, p. 2, 2025.

[15] M. Russo, C. Liu, S. Sudhir, G. Vitagliano, M. Cafarella, T. Kraska, and S. Madden, "Abacus: A cost-based optimizer for semantic operator systems," *PVLDB*, vol. 19, no. 5, 2026.

[16] L. L. Wei, S. Shankar, S. Zeighami, Y. Chung, F. Ozcan, and A. G. Parameswaran, "Multi-objective agentic rewrites for unstructured data processing," *To Appear at VLDB*, 2026.

[17] A. G. Parameswaran, S. Shankar, P. Asawa, N. Jain, and Y. Wang, "Revisiting prompt engineering via declarative crowdsourcing," in *Conference on Innovative Data Systems Research (CIDR)*, 2024.

[18] S. Zeighami, Y. Lin, S. Shankar, and A. Parameswaran, "Llm-powered proactive data systems," *Data Engineering*, p. 90.

[19] S. Shankar, S. Zeighami, and A. Parameswaran, "Task cascades for efficient unstructured data processing," *Proc. ACM Manag. Data*, vol. 4, Apr. 2026.

[20] S. Zeighami, S. Shankar, and A. Parameswaran, "Cut costs, not accuracy: Llm-powered data processing with guarantees," *Proc. ACM Manag. Data*, vol. 3, Dec. 2025.

[21] S. Zeighami, S. Shankar, and A. G. Parameswaran, "Featurized-decomposition join: Low-cost semantic joins with guarantees," *To Appear at VLDB*, 2026.

[22] Y. Sun, S. Zeighami, B. Chopra, S. Shankar, and A. G. Parameswaran, "Semantic data processing with holistic data understanding," *arXiv preprint arXiv:2604.02655*, 2026.

[23] California State Legislature, "Assembly bill no. 2542: California racial justice act of 2020." Cal. Penal Code § 745, 2020. Approved by the Governor, September 30, 2020.

[24] Berkeley Institute for Data Science, UC Berkeley Investigative Reporting Program, and Stanford Big Local News, "California police records access project." `https://bids.berkeley.edu/california-police-records-access-project`, 2025. Part of the California Law Enforcement Accountability Network (CLEAN) initiative.

[25] S. Shankar, B. Chopra, M. Hasan, S. Lee, B. Hartmann, J. M. Hellerstein, A. G. Parameswaran, and E. Wu, "Steering semantic data processing with docwrangler," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, 2025. Best Paper Honorable Mention.

[26] CalMatters and The Garrison Project, "California's racial justice act boasts few successes, four years in." `https://calmatters.org/justice/2024/11/california-racial-justice-act/`, 2024. Reports that the state keeps no systematic count of § 745 motions; roughly a dozen successful motions statewide in the law's first four years.

[27] H. H. Clark and S. E. Brennan, "Grounding in communication," in *Perspectives on Socially Shared Cognition* (L. B. Resnick, J. M. Levine, and S. D. Teasley, eds.), pp. 127–149, American Psychological Association, 1991.

[28] A. T. Kalai and S. S. Vempala, "Calibrated language models must hallucinate," in *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pp. 160–171, 2024.

[29] P. Sui, E. Duede, S. Wu, and R. J. So, "Confabulation: The surprising value of large language model hallucinations," *arXiv preprint arXiv:2406.04175*, 2024.

[30] Y. Liu, H. Zhou, Z. Guo, E. Shareghi, I. Vulić, A. Korhonen, and N. Collier, "Aligning with human judgement: The role of pairwise preference in large language model evaluators," in *First Conference on Language Modeling*, 2024.

[31] S. Shankar, J. Zamfirescu-Pereira, B. Hartmann, A. Parameswaran, and I. Arawjo, "Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–14, 2024.

[32] J. Lao, A. Zimmerer, O. Ovcharenko, T. Cong, M. Russo, G. Vitagliano, M. Cochez, F. Özcan, G. Gupta, T. Hottelier, H. V. Jagadish, K. Kissel, S. Schelter, A. Kipf, and I. Trummer, "Sembench: A benchmark for semantic query processing engines," *arXiv preprint arXiv:2511.01716*, 2025.

[33] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Wrangler: Interactive visual specification of data transformation scripts," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3363–3372, 2011.

[34] T. Shnitzer, A. Ou, M. Silva, K. Soule, Y. Sun, J. Solomon, N. Thompson, and M. Yurochkin, "Large language model routing with benchmark datasets," *arXiv preprint arXiv:2309.15789*, 2023.

[35] N. Kandpal, H. Deng, A. Roberts, E. Wallace, and C. Raffel, "Large language models struggle to learn long-tail knowledge," *Proceedings of the 40th International Conference on Machine Learning*, 2023.

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: $X \to Bool$)	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: T, l: $(X, Y) \to Bool$)	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: $T[X] \to X$)	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: $T[X] \to Seq[X]$, k: 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: $X \to Y$, C: 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: $X \to Y$)	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

```
1     # find examples of agent failures
2     traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")
3
4     # explain each failure
5     traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")\
6                   .sem_map("summarize the failure cause of the agent {trace}")
7
8     # summarize common failure patterns
9     traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")\
10                  .sem_map("summarize the key failure and cause from the agent {trace}",
      suffix="agent_failure_summary")\
11                  .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

```
1    # taxonomize survey response data and list candidate labels
2    responses.sem_agg("list the key domains for agent use based on the survey responses
     given by {Q4_answers}")
3
4    # label each response using the produced taxonomy
5    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.

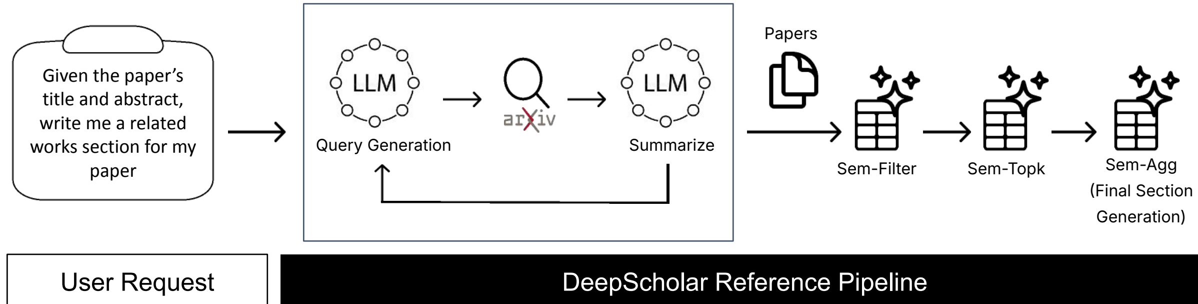

Figure 5: Overview of DeepScholar-ref. The system iteratively writes queries and performs web search, before passing the search results through series of semantic operators using the LOTUS system for LLM-based data-processing, including filtering step to discard irrelevant sources, a top-k ranking step 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].

Expanding the Physical Design Space of LLM Data Systems

Gabriele Sanmartino[1], Matthias Urban[2], Carsten Binnig[2], and Paolo Papotti[1]
[1]EURECOM, France
[2]TU Darmstadt, Germany

Abstract

Large language models are becoming first-class components of data systems through semantic operators that declaratively process text, images, audio, and other unstructured data. While recent systems have made progress in defining such operators and optimizing their execution, their physical design space remains comparatively narrow: an optimizer often chooses among models, prompts, cascades, or approximate filters, but treats inference itself largely as a black box. This article argues that LLM-native data systems should expose selected inference-time mechanisms to the optimizer as physical design choices. We focus on key–value (KV) caches in transformer decoders as a concrete example. Rather than viewing KV caches only as a serving optimization, we treat them as materializable physical representations of data items that can be precomputed, compressed, reused, and selected during query planning. Different cache profiles induce different trade-offs among latency, memory, cost, and output quality, thereby expanding the set of physical implementations available for semantic operators. To make this larger space manageable, we advocate constructing Pareto frontiers of candidate implementations and exposing only non-dominated choices to the optimizer. We discuss how this design pattern applies to LLM-native systems such as LOTUS and STRETTO, where selecting the right physical operators can preserve quality guarantees while reducing inference time. The broader message is that LLM-native query engines require an inference-aware physical layer, not merely better prompts or faster model serving.

1 Introduction

Large language models are increasingly becoming first-class components of data systems. Instead of treating text, images, audio, and other unstructured objects as inputs to an external preprocessing pipeline, recent *LLM data systems* expose *semantic operators* that allow users to query such data declaratively. A user may ask for documents that satisfy a natural language (NL) predicate, extract structured attributes from reports, join records according to semantic similarity, or rank multi-modal objects according to a task-specific criterion. Systems such as PALIMPZEST [9], LOTUS [12], DOCETL [18], THALAMUSDB [5], GALOIS [17], and CAESURA [19] illustrate this shift from LLMs as external tools to LLMs as components of the query execution stack.

This shift raises a familiar question for database systems: what is the physical design space behind a logical query? Classical systems separate the logical meaning of an operator from its physical realization. A join may be implemented as a hash join, sort-merge join, nested-loop join, or index-nested-loop join; the optimizer chooses among them using cost and cardinality estimates. LLM data systems need a similar separation. A semantic filter, map, or join specifies *what* the system should compute, but the system may implement it using different models, prompts, retrieval strategies, cascades, thresholds, batching policies, or approximate methods. The difference is that physical choices in LLM systems affect not only latency and resource consumption, but also output quality. The optimizer must therefore reason about a multi-objective space involving cost, latency, memory, and task-level quality.

The current physical space. Early LLM data systems have defined semantic operators and introduced ideas for reducing the cost of executing them. Lotus, for example, shows how semantic operators can be optimized using cascades [12]. Other systems explore cost-based search over semantic-operator implementations [15], SQL-style optimizations [17], logical and prompt-level rewrites [14, 21], or quality-preserving routing between cheaper and more expensive LLM calls [22, 23]. These approaches, together with inference-aware techniques such as reduced-context execution, model routing, and serving-level cache management [6, 9, 11], demonstrate that semantic queries should not be executed by naively invoking the largest available model for every tuple.

However, the design space exposed to semantic-query optimizers is still often coarse-grained. A system may choose between a small model and a large LLM, between one prompt and another, between a reduced and a full context, or between invoking an LLM and applying a cheaper embedding-based filter. These choices are useful, and prior work shows that they can reduce inference cost. Our point is complementary: they can still leave gaps along the cost–quality curve when the optimizer lacks intermediate physical implementations. A small LLM or aggressively reduced context may be too inaccurate for a given quality target, while the next available high-quality implementation may be more expensive than necessary. In such settings, the optimizer would benefit from a denser menu of physical operators.

Inference state as physical design. This article argues that LLM data systems should further expand their physical design space by exposing selected inference-time mechanisms to the query optimizer. In particular, we focus on the key–value (KV) cache used by transformer decoders. During autoregressive decoding, the KV cache stores attention keys and values for the already processed context, avoiding repeated recomputation during later decoding steps [13]. In standard serving systems, this cache is primarily managed as a runtime optimization for efficient inference and memory reuse. From a data-management perspective, however, the KV cache can be viewed differently: it is a materializable physical representation of a data item under a given model. Because a cache captures how the model has encoded an input object, it can support repeated semantic processing over that object and can also provide signals for estimating the behavior of semantic operators before they are executed.

This observation changes the role of the cache. If some queries repeatedly condition on the same set of objects—for example, the documents, images, or records—then their KV caches can be precomputed, stored, compressed, and reused across operators and queries. This reuse has two complementary benefits. First, caches can accelerate *processing*: an operator can avoid recomputing the model representation of an input object and can instead condition directly on a cached or compressed representation. Second, caches can improve *planning*: the same representations can be used to estimate which records are likely to satisfy a semantic predicate, how many pairs may survive a semantic join, or which inputs require a more expensive model call. In classical databases, physical design structures such as indexes and materialized views support both efficient access and better optimization. KV caches can play an analogous role for LLM-native systems, acting both as execution artifacts and as statistical objects for semantic cardinality estimation [20].

Moreover, different cache variants can be materialized for the same item. A full cache may preserve high quality but require more memory and lower batching throughput. A compressed cache may reduce memory footprint and accelerate execution, while introducing a controlled degradation in task quality or estimation accuracy. By creating a Pareto frontier of cache profiles, the system obtains multiple physical implementations of the same logical semantic operator, as well as multiple summaries that can support optimization-time estimates. Recent work on KV-cache compression and token pruning provides mechanisms for constructing such variants [3, 4, 8], while LLM serving and routing systems show the importance of inference-time decisions such as cache management, batching, and model selection for efficient inference [6, 10, 11].

KV caches should be exposed to the *optimizer*, not to the user: users should continue to write

declarative queries over semantic operators. The system decides whether an operator instance should use an embedding filter, a full KV cache, or one of the compressed cache profiles. Similarly, it decides when cached representations are useful for estimating selectivities and intermediate result sizes before committing to an execution plan.

From more operators to better choices. Expanding the physical design space is useful only if the optimizer can navigate it. Blindly materializing many cache variants can increase storage cost and make planning harder. A practical system therefore needs a compact representation of the useful choices. We advocate constructing *Pareto frontiers* of physical operator implementations. For each logical semantic operator, or for each class of operator instances, the system profiles candidate implementations along dimensions such as latency and estimated output quality. Dominated implementations are removed: if one implementation is both slower and lower-quality than another, it should not be presented to the optimizer. The remaining frontier gives the optimizer a compact menu of non-dominated physical alternatives.

This frontier-based view makes the system architecture modular. The mechanism that creates physical alternatives—for example, KV-cache compression—can be separated from the mechanism that chooses among them. A rule-based optimizer, a classical cost-based optimizer, or a gradient-based optimizer can all consume a frontier of candidate implementations, provided the frontier exposes the relevant cost and quality estimates.

Physical design for LLM-native data systems. In this paper, we do not argue for an optimizer, a single cache-compression method, or a new semantic-operator API. Instead, we identify a systems pattern that is likely to recur as LLMs become part of data-processing engines: logical semantic operators should be paired with a rich set of physical implementations; inference state, including KV caches, can be materialized and managed as part of that physical space; and Pareto frontiers can make the resulting space usable by query optimizers.

The remainder of the paper develops this argument. We start by giving an example of the role of KV cache in semantic processing. We then describe how physical operator choices arise in LLM data systems and why the cost–quality dimension makes them different from classical physical operators. Next, we introduce KV caches as reusable physical representations of data items and explain how compressed cache profiles expand the operator space. We then discuss how Pareto frontiers can be constructed and used to select among cache-aware and non-cache-aware implementations. Finally, we use systems such as Lotus and Stretto to illustrate the practical impact of this design: by selecting the right physical operators, an engine can preserve quality guarantees while reducing inference time.

2 A Running Example: Cache-Aware Semantic Processing

We use an example to illustrate how KV caches expand the physical design space of an LLM-native data system. Consider a table of building-inspection records, where each tuple contains a textual report and one image:

```
SELECT id, sem_extract(report, "recommended repair action") AS action
FROM Inspections
WHERE sem_filter(photo,  "shows visible water damage")
  AND sem_filter(report, "mentions mold or moisture")
WITH QUALITY precision >= 0.90 AND recall >= 0.90;
```

The query is declarative. It specifies two semantic filters, one over images and one over text, followed by a semantic extraction over the surviving reports. It also specifies an end-to-end quality target on the final output.

Naive physical execution. A straightforward implementation invokes a high-quality model for each semantic operator and each input item. For the image predicate, the model receives the image and the predicate, and produces a Boolean decision. For the text predicate, it receives the report and the predicate. For the extraction operator, it receives the report again and generates the action. Before producing any output, the model must process the input context and construct its internal attention state. For long documents and high-resolution visual inputs, this *prefill* step can dominate the latency of semantic operators.

Cache-aware physical execution. A cache-aware system separates object encoding from query execution. In an offline preprocessing phase, the system feeds each object into the model and materializes its KV cache. For the example above, it may construct caches for every inspection photo and every report. At query time, a semantic operator no longer needs to send the full object through the model from scratch. Instead, it retrieves the corresponding cache, appends an operator-specific suffix that encodes the predicate or extraction request, and performs only the remaining decoding work. The same cached report can be used by the textual filter, by the extraction operator, and by future queries over the same collection, i.e., across multiple semantic operations instead of recomputing it for every call.

Compressed cache profiles. A full KV cache can be large, especially for long contexts and multimodal inputs. Rather than using only the full cache, the system can materialize several cache *profiles* for the same object, each corresponding to a model and a compression ratio. For example, an image may be represented by a full cache, a lightly compressed one, and a more aggressively compressed one. These profiles become distinct physical implementations of the same operator. For the first predicate in the example, the optimizer may determine that a highly compressed image cache is sufficient to reject many obvious non-matches. For the extraction operator, which produces a more information-sensitive output, the optimizer may choose a less compressed cache.

Cascades over cache-aware operators. The physical choices need not be used in isolation. A semantic filter can be implemented as a cascade of increasingly expensive operators. For instance, the image predicate may first use a small model with an aggressively compressed cache. If the model is confident, the tuple is accepted or rejected immediately. If the decision is uncertain, the tuple is forwarded to a less compressed cache profile or to a larger model. This cascade view is useful because not all tuples require the same amount of computation. In the inspection example, many photos may clearly show no water damage, and many reports may clearly not mention moisture. Processing those tuples with the most expensive model and the least compressed cache would waste resources. Conversely, a small model or an aggressive compression profile may be insufficient for borderline cases. A cache-aware cascade gives the optimizer intermediate choices between these extremes.

Speed-up and Optimizer. Compressed KV caches create two complementary speed-up opportunities. First, as the cache has already been computed, execution bypasses the prefill phase and focuses on output generation. This is especially beneficial for semantic filters and short extractions, where the output is small w.r.t the input context. Second, compression reduces the memory footprint of each cached item. Smaller caches allow the executor to fit more items in a GPU batch, improving throughput when evaluating an operator over a collection.

These profiles create intermediate execution choices between the extremes of a cheap approximate operator and an expensive high-fidelity one. Section 3 describes how these choices are profiled and exposed to the optimizer for query execution. Although this example focuses on execution, the same cached representations may also provide useful signals for optimization-time tasks such as estimating

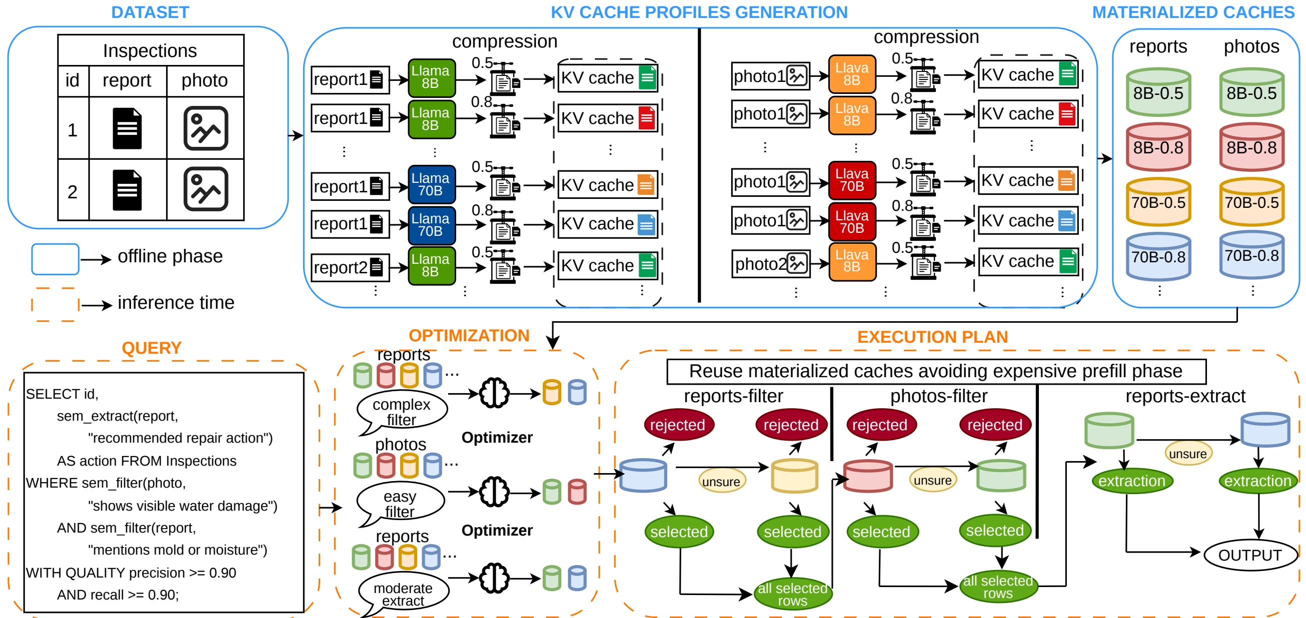

Figure 1: Cache-aware execution of the running example. The user specifies a logical query with semantic filters and an extraction operator. Offline, the system materializes KV-cache profiles for each base object under different models and compression ratios. Online, the optimizer selects a physical implementation for each semantic operator: aggressive compression may be sufficient for easy filtering decisions, while extraction or uncertain cases may use higher-fidelity profiles. The speed-up opportunity comes from bypassing repeated prefill computation and from using smaller compressed caches that permit larger batches and lower latency.

predicate selectivity or operator difficulty; we leave that direction as an opportunity rather than the focus of this article [20].

3 Cache-Aware Physical Design

We now describe how cache-aware physical design is implemented. The design has two phases. Offline, the system profiles candidate physical implementations, including KV-cache-aware choices, and retains a compact set of useful alternatives. Online, the optimizer selects implementations for a logical semantic plan while satisfying user-specified quality targets. We use Stretto as a reference example, but the pattern is not specific to a particular optimizer or cache-compression method.

3.1 From Semantic Operators to Physical Implementations

A semantic operator specifies a task over data, but not the mechanism used to execute that task, e.g., a filter predicate "shows visible water damage" or "mentions mold or moisture". These operators define the logical meaning of the query. The system can choose among different physical implementations that preserve this meaning up to a requested quality target, e.g., a filter implemented by invoking a large model directly, a smaller model, or a cached representation of the input object.

This separation mirrors the logical–physical distinction in classical database systems. The difference is that, in LLM systems, physical implementations affect not only runtime and resource consumption but also output quality. The optimizer therefore reasons over a multi-dimensional space in which each

implementation has an estimated latency, memory footprint, throughput, and quality.

KV caches expand this space with a new class of physical implementations. The full cache may offer higher quality but require more memory. A compressed cache may be faster and allow larger batches, but may introduce some quality degradation. More precisely, let o be a logical semantic operator, such as a semantic filter or extraction operator. The system associates o with a set of candidate physical implementations:

$$\mathcal{P}(o) = \{p_1, p_2, \ldots, p_k\}.$$

Each implementation p_i specifies how the operator may be executed, for example by choosing a model, a cache profile[1], a compression ratio, a decision threshold, or a cascade policy. We write this configuration as

$$p_i = \langle m_i, c_i, \tau_i \rangle,$$

where m_i is the model, c_i is the cache profile or compression setting, and τ_i denotes operator-specific parameters such as thresholds. The optimizer also needs estimates of how this implementation behaves on the data and query class at hand. These estimates are obtained by profiling p_i on a calibration sample or representative workload:

$$\widehat{\mu}(p_i, o, D) = \langle \widehat{q}_i, \widehat{r}_i \rangle,$$

where $\widehat{q}_i$ is the estimated output quality and $\widehat{r}_i$ is the measured or estimated execution time when applying p_i to operator o over a data sample D. In addition, the system can associate each implementation with a cache footprint s_i, which is determined by the model and compression ratio rather than by the operator output: $s_i = \text{size}(m_i, c_i)$. Thus, quality and runtime are empirical properties of a physical implementation under a given query, data distribution, and hardware, while the cache footprint follows from the materialized profile.

Different operator types may favor different profiles: filters often benefit from aggressive compression, while maps and extractions may require higher-fidelity caches because they generate longer outputs.

3.2 Offline Generation and Pruning of KV-Cache Profiles

The offline phase creates the physical design space exposed to the optimizer. Given a dataset, the system can theoretically materialize KV caches for every base object (e.g., text documents, images) across multiple models and compression ratios. Compression is applied to deliberately trade task fidelity for faster execution and a smaller memory footprint.

However, KV caches are large. Materializing and storing every possible combination of model and compression ratio would incur prohibitive storage costs and bloat the optimizer's search space. Therefore, we must carefully select which profiles to actually materialize.

Constructing the Pareto Frontier Offline. To decide which profiles to keep, we evaluate a grid of candidate model–compression pairs on a calibration sample to measure expected execution time and output quality (e.g., F1 score). The calibration workload need not contain all future queries; it is used to estimate the behavior of each candidate implementation on the class of semantic tasks the system expects to support.

The grid is a design choice: the system starts from a finite set of models and compression ratios chosen by the system designer or from prior profiling. Using these measurements, we construct a *Pareto frontier* offline. We plot the candidates in a runtime–quality space and discard any *dominated* implementations. An implementation is dominated if another profile provides at least the same quality

[1] A cache profile becomes a physical operator only when it is paired with an operator-specific prompt, threshold, or cascade policy.

with a lower runtime. For instance, empirical profiling often reveals that an aggressively compressed 70B model might degrade in quality so much that it matches the accuracy of an uncompressed 8B model, but still runs slower. In such cases, the compressed 70B profile is discarded. The system only physically materializes the KV caches that lie on the non-dominated frontier.

Manual Pruning for Diverse KV-cache Profiles. Moreover, manual pruning of the Pareto frontier can further decrease storage costs and make the optimizer's search space more compact. We favor diversity across the selected points on the Pareto frontier: the final selected profiles should include cheap aggressive-compression options, high-fidelity options, and intermediate choices. Configurations that are too close to already selected points are removed.

This offline pruning yields a compact, pre-computed repository of non-dominated cost–quality trade-offs, reducing storage and optimizer search cost.

3.3 Online Query Planning

At query time, the optimizer's job is greatly simplified. It does not need to reason about raw compression algorithms or search an infinite space of hyperparameters; it is simply handed the compact, pre-computed Pareto frontier of physical operators generated offline.

Given a logical plan containing multiple semantic operators, the optimizer selects one physical implementation from the corresponding frontier for each operator. Because users specify *global* quality targets and errors propagate through the execution pipeline, the optimizer uses the frontier to strategically allocate the query's error budget. It may assign highly compressed, cheap profiles to easy or low-impact operators, while reserving the uncompressed, full-cache implementations for operators that heavily influence final output quality. The offline frontier ensures that the optimizer reasons over implementations that were non-dominated on the calibration workload, rather than over redundant or clearly inferior cache profiles.

3.4 Executing Semantic Operators with Cached Representations

Once the optimizer finalizes the physical plan, the system executes it. For cache-aware operators, the executor loads the selected KV-cache profile from disk instead of re-encoding the raw input object from scratch. The operator simply appends its task-specific suffix (e.g., a natural language filter predicate or an extraction prompt) to the cached state and performs the remaining decoding work.

This approach yields two major runtime benefits:

1. **Bypassing Prefill:** By avoiding the expensive context-encoding phase, execution latency depends mainly on the chosen cache profile and the length of the generated output.

2. **Higher Throughput:** Because offline compression reduces the memory footprint of the surviving cached items, the executor can fit more objects into GPU memory simultaneously. This increases the maximum batch size, significantly improving throughput when evaluating semantic operators over large collections.

4 Illustration in LLM-Native Systems

This section illustrates the effect of cache-aware physical design in LLM-native workloads. The goal is not to introduce a benchmark, but to show two consequences of the design pattern described above. First, KV-cache profiles create a richer runtime–quality space for individual semantic operators. Second,

Table 1: Physical operator variants considered before pruning. Each candidate implementation is defined by an operator type, a model, and a KV-cache compression ratio. Compression ratio 0 denotes the uncompressed KV cache and is also the baseline used in the end-to-end experiment. The optimizer is only exposed to the subset of variants (in bold) retained after Pareto and diversity pruning.

Dataset	Modality	Operators	Candidate Variants 70B	Candidate Variants 8B
Movie	Text	filter, map	$\{0.00, \mathbf{0.30}, 0.50, \mathbf{0.60}, \mathbf{0.80}, 0.90\}$	$\{\mathbf{0.00}, 0.30, \mathbf{0.50}, \mathbf{0.80}, 0.90\}$
Rotowire	Text	filter, map	$\{0.00, \mathbf{0.30}, 0.50, \mathbf{0.60}, \mathbf{0.80}, 0.90\}$	$\{\mathbf{0.00}, 0.30, \mathbf{0.50}, \mathbf{0.80}, 0.90\}$
Artwork	Image	filter, map	$\{0.00, 0.30, \mathbf{0.50}, 0.80, \mathbf{0.90}, \mathbf{0.99}\}$	$\{\mathbf{0.00}, \mathbf{0.50}, 0.80, \mathbf{0.90}, 0.99\}$

when these profiles are exposed to a query optimizer, the same optimizer can find faster plans under the same quality targets.

We use Stretto as the reference optimizer in the end-to-end experiment [16]. The comparison is therefore not between two different optimizers, but between two physical design spaces: one where Stretto can choose among compressed KV-cache profiles, and one where the same optimizer does not have access to these profiles.

4.1 Evaluation Questions

The evaluation focuses on two questions.

Q1: Do KV-cache profiles create useful runtime–quality Pareto frontiers? We first study whether different model–compression configurations produce distinct, non-dominated choices for individual semantic operators.

Q2: Do KV-cache profiles improve end-to-end query execution under fixed quality targets? We then evaluate complete semantic queries with global precision and recall targets. We compare Stretto with access to the KV-cache frontiers against the same optimizer without compressed physical operators. Both configurations optimize for the same workload and quality targets. The relevant question is whether the richer physical design space allows the optimizer to reduce runtime while preserving the requested guarantees.

4.2 Experimental Setting

We consider semantic workloads over text and image data, following the practice of evaluating semantic query engines across multiple modalities and operator types [7]. For the operator-level frontier experiment, we use three datasets: Movie and Rotowire for text operators, and Artwork for image operators. These datasets cover different input modalities and different operator behavior, which is useful for showing that cache profiles do not induce a single universal trade-off. Instead, each workload has its own runtime–quality frontier.

For the Pareto frontier study (**Q1**), we evaluate cache-aware implementations obtained from a finite grid of models and compression ratios for each dataset (Table 1). The grid is chosen manually, based on the models available to the system and on a range of compression levels that produce visibly different runtime–quality behavior. The workload used for this study consists of single-operator tasks, i.e., each execution involves eithera semantic filter or map; they characterize the runtime–quality tradeoff of each physical operator. After profiling the grid, we compute the Pareto frontier to prune dominated configurations. To retain a compact set of configurations, we manually further prune, favoring diversity across selected points and removing configurations that are too close together in the runtime-quality trade-off.

Table 2: Number of queries and distribution of operators in the test workload for Q2 (F = Filter, M = Map).

Dataset	**F→M**	**F→F**	**M→M**	**F→M→M**	**F→F→M**	**F→M→M→M**	**F→F→M→M**	**Total**
Movie	10	10	10	10	–	10	–	50
Rotowire	10	10	–	10	10	10	10	60
Artwork	10	10	–	10	10	10	10	60

For the end-to-end experiment (**Q2**), we use STRETTO as the optimizer in both configurations [16]. STRETTO is an optimizer that selects physical implementations for semantic operators using profiled runtime and quality estimates, while enforcing global precision and recall targets. The first configuration, STRETTO w/o compression (Uncompressed KV), uses KV caches but only at compression ratio 0, i.e., uncompressed caches. The second configuration, STRETTO w/ compression (Compressed KV), optimizes over the expanded physical space that includes compressed KV-cache profiles. Both configurations benefit from cache reuse and avoid repeated prefill computation; the comparison isolates the additional value of compression as a physical design dimension. They also receive the same logical queries and the same global precision and recall targets. Runtime is the main metric and results are interpreted only among plans that satisfy the requested quality guarantees. In this study, the test workload includes tasks with multiple operators, as shown in Table 2. Starting from a base set of individual semantic filters and joins, two to four semantic operators are randomly combined to construct queries.

4.3 Operator-Level Trade-Offs from Cache Profiles (Q1)

Table 3 shows the runtime–quality behavior of cache-aware physical implementations, averaged over semantic filter and map operators for each dataset. Each result corresponds to one candidate implementation, obtained by selecting a model and a KV-cache compression ratio. The table covers the two text datasets and the image one (ARTWORK).

The table illustrates the main role of compressed KV caches. For a fixed model, increasing compression generally reduces runtime, but may also reduce quality. Across models, larger models tend to provide higher quality, but they are more expensive. The useful points are therefore not concentrated in a single model or a single compression ratio. Instead, different model–compression combinations occupy different regions of the runtime–quality space. For example, a compressed large model can provide an intermediate option between a cheap small model and a full high-quality large model. Conversely, when compression degrades quality too much, a large compressed model may become dominated by a smaller model.

The selected frontier contains diverse points along this curve, rather than every measured configuration. This gives the optimizer a compact set of choices between coarse alternatives such as “small” and “large” models.

4.4 End-to-End Planning with and without KV Caches (Q2)

Figure 2 evaluates the impact of compressed KV-cache profiles on semantic queries. In this experiment, the optimizer is STRETTO in both configurations. The baseline configuration uses uncompressed KV caches, corresponding to compression ratio 0. The cache-aware configuration exposes the same optimizer to the full KV-cache frontier, including compressed profiles. The comparison does not measure the benefit of caching itself (both configurations use cached representations), but it isolates the benefit of adding compressed cache profiles to the physical design space.

Table 3: Runtime–quality trade-off induced by KV-cache profiles. Each row represents a physical implementation defined by a model and compression ratio. Results are averaged over queries (single filter or map operator) for each dataset. Compressed cache profiles add intermediate points between cheap and expensive operators. The Gold Operator is the largest model without compression.

Gold Model : 70B model with no compression **Pareto Optimal** : On the Pareto frontier

		Movie		Rotowire		Artwork	
Model	**Ratio**	**Runtime (s)**	**F1 Score**	**Runtime (s)**	**F1 Score**	**Runtime (s)**	**F1 Score**
70B	0.00	47.59	1.0000	252.86	1.0000	499.51	1.0000
	0.30	33.98	0.9612	166.54	0.9287	360.96	0.9573
	0.50	32.71	0.9228	85.19	0.8833	268.28	0.9361
	0.60	29.83	0.8921	71.75	0.8468	232.38	0.9186
	0.80	24.35	0.7217	51.67	0.7334	123.52	0.8653
	0.90	22.72	0.4656	36.91	0.5858	38.02	0.8479
	0.99	22.93	0.3074	34.31	0.1745	8.89	0.6830
8B	0.00	16.19	0.7161	50.31	0.6285	20.18	0.7410
	0.30	14.70	0.6970	36.82	0.6195	16.64	0.7409
	0.50	12.41	0.6759	27.90	0.6008	12.41	0.7281
	0.80	8.89	0.5852	16.18	0.5038	9.51	0.6783
	0.90	7.12	0.4064	13.56	0.4063	8.29	0.6663
	0.99	–	–	–	–	8.37	0.4971

The comparison is made under the same global precision and recall targets. This point is important: both configurations satisfy the requested statistical quality guarantees. The difference is therefore not that the cache-aware version trades away correctness for speed. Rather, compressed KV-cache profiles give the optimizer additional non-dominated physical implementations, allowing it to find lower-runtime plans that still meet the same quality constraints.

The effect is visible across the datasets in Figure 2. For loose quality targets, the optimizer can often select aggressively compressed profiles or cheap cascade stages, leading to large runtime reductions. For stricter targets, the optimizer uses higher-fidelity profiles more often, but cache-aware planning can still reduce runtime by selecting smaller cache profiles that improve batching and reduce cache-loading costs. The overall message is that KV caches are useful not only as a serving optimization, but as physical operators that improve end-to-end query planning.

The gains are not uniform across datasets. They are largest on Movie, where many predicates are sentiment-style judgments over text. Such tasks remain accurate even under aggressive compression, because the relevant signal is coarse across the review. Stretto can therefore select highly compressed profiles while still satisfying the precision and recall targets. The gains are smaller on Rotowire, where factual details are more sensitive to information loss. This illustrates why exposing a frontier is useful: the optimizer can exploit compression when the task is robust to it and fall back to higher-fidelity profiles otherwise. Finally, for Artwork, the KV caches are very large, so avoiding the loading of multiple cache profiles can outweigh the benefit of using smaller compressed caches in this small-scale experiment. However, this should not be interpreted as evidence that uncompressed KV caches are preferable in general: for realistic corpora with thousands of images, the uncompressed cache repository may become too large to store, load and batch, making compression necessary to keep the physical design space practical.

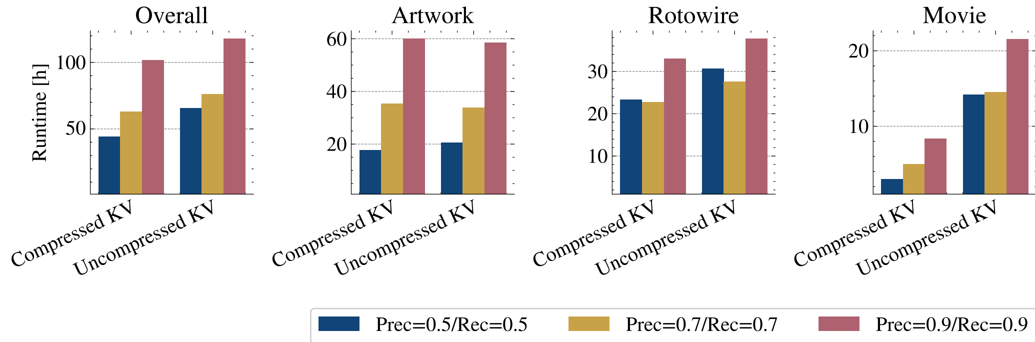

Figure 2: Runtime comparison of STRETTO with uncompressed KV caches and STRETTO with compressed KV-cache profiles. Both configurations use the same optimizer and workload, reuse KV caches, and satisfy the requested statistical guarantees on output quality. The cache-aware configuration exposes a richer physical design space by adding compressed profiles, enabling faster plans under the same precision and recall targets.

Table 4: KV-cache storage sizes (in GB) across textual and visual datasets for different combinations of model and compression ratio.

8B Model				70B Model			
Ratio	**Movie**	**Rotowire**	**Artwork**	**Ratio**	**Movie**	**Rotowire**	**Artwork**
0.0	6.8	38	298	0.0	17	93	5800
0.5	3.4	19	149	0.3	12	65	4060
0.8	1.4	7.4	60	0.8	3.3	19	1160

4.5 Memory Requirements and Compression Impact

To quantify the storage requirements of maintaining multiple KV-cache configurations, Table 4 reports the memory footprints across our benchmark workloads. The numbers show why cache profiles cannot be materialized indiscriminately: for example, the uncompressed 70B cache footprint reaches 93GB on ROTOWIRE and 5800GB on ARTWORK, while compression substantially reduces these requirements. In the end-to-end experiments, exposing compressed profiles to the optimizer yields average speedups for all precision/recall targets w.r.t. a baseline that uses uncompressed KV caches.

4.6 Takeaways

The experiments support two conclusions. First, compressed KV-cache profiles create a richer Pareto frontier of physical operator implementations, with useful intermediate choices between cheap low-quality operators and expensive high-quality ones. Second, exposing this frontier to STRETTO improves end-to-end execution: the optimizer can select faster physical plans while satisfying the same statistical quality guarantees.

5 Conclusion and Outlook

LLM-native data systems need a richer physical layer than model and prompt selection alone. As semantic operators become part of declarative query engines, the system must decide not only *which* model to invoke, but also *how* inference state should be materialized, compressed, reused, and exposed

to the optimizer. KV caches provide one concrete example of this broader principle. When treated as physical design structures, they create additional implementations of semantic operators and fill gaps in the runtime–quality space. Pareto frontiers then make this expanded space manageable: the optimizer can reason over a compact set of non-dominated alternatives instead of over a large collection of redundant model–compression choices.

Empirical results illustrate the potential of this design. At the operator level, compressed KV-cache profiles create diverse runtime–quality trade-offs across workloads. At the query level, exposing these profiles to an optimizer enables faster physical plans while preserving the same quality guarantees.

Several research directions remain open.

Cache-aware physical design. Current cache-profile selection relies on a finite grid of models and compression ratios. Future systems should make this process more adaptive. Given a workload, hardware budget, and quality target, the system should decide which profiles to materialize, which to discard, and when to refresh them. More generally, memory should become an explicit constraint in semantic query optimization, rather than an implicit property of the serving backend. An optimizer would then need to choose physical operators subject not only to runtime and quality constraints, but also to storage and GPU-memory budgets for cached representations. This raises questions similar to index and materialized-view selection, but with an additional quality dimension. Existing semantic query optimizers would need to expose memory footprints in their cost models, propagate memory requirements across multi-operator plans, and coordinate with the serving layer on batching, cache eviction, and recomputation policies.

Robust quality estimation. The optimizer depends on estimates of the quality and runtime of each physical implementation. These estimates are currently obtained from calibration workloads, but future queries may differ from the calibration sample. A key challenge is to make quality estimates robust under workload shift, model updates, and changing data distributions. This is especially important when the system promises end-to-end quality guarantees.

Cache-derived statistics. Although this article focuses on execution, KV caches may also support optimization-time statistics. Cached representations can help estimate predicate selectivity [20], join survival rates, or the difficulty of an operator. This direction would make caches play a role closer to classical database statistics: useful not only for faster access, but also for better planning. However, it requires careful validation, since cache-derived signals may be model-specific and workload-dependent.

Beyond KV caches. KV caches are one instance of a more general idea: inference artifacts can become physical design structures. Future LLM-native systems may expose other reusable internal representations, intermediate reasoning traces, retrieval states, or learned operator summaries to the optimizer. As the community shifts toward agentic workflows and AI-powered data clouds [1, 2], caching and reusing the intermediate tool-use and reasoning traces of these agents will become a pivotal dimension of physical design. The broader research question is how to define a principled physical algebra for LLM-native execution, where inference state, quality estimates, memory footprint, and runtime all become first-class optimization dimensions.

Acknowledgments

This work was funded by the DFG/ANR Project MAgiQ (ANR24-CE92-0077; DFG, German Research Foundation – Project No. 545611510), the LOEWE Spitzenprofessur programme (III 5-

519/05.00.003(0005)), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC3057/1 "Reasonable Artificial Intelligence", Project No. 533677015), and by the French government, through the 3IA Côte d'Azur Investments in the IA-cluster project managed by the National Research Agency (ANR-23-IACL-0001). This project was provided with resources by GENCI at IDRIS, thanks to grants 2025-AD010616649 and 2025-AD010616180. We also thank DFKI and hessian.AI.

References

[1] Elaine Ang, Chenxi Huang, Georgios Liargkovas, Jerry Liu, Jinhui Liu, Nikos Pagonas, Charlie Summers, Haonan Wang, Jiakai Xu, Tianle Zhou, Yusen Zhang, Zhou Yu, Zhuo Zhang, Tianyi Peng, Kostis Kaffes, and Eugene Wu. Agentic data environments. *IEEE Data Eng. Bull.*, 50(1):5–21, 2026.

[2] Yeounoh Chung, Thibaud Hottelier, Cosmin Arad, Brenton Milne, Per Jacobsson, Sam Idicula, Fatma Özcan, Alon Y. Halevy, and Yannis Papakonstantinou. Architecting the ai-powered agentic data cloud. *IEEE Data Eng. Bull.*, 50(1):22–39, 2026.

[3] Giulio Corallo and Paolo Papotti. FINCH: prompt-guided key-value cache compression for large language models. *Trans. Assoc. Comput. Linguistics*, 12:1517–1532, 2024.

[4] Alessio Devoto, Maximilian Jeblick, and Simon Jégou. Expected attention: Kv cache compression by estimating attention from future queries distribution, 2025.

[5] Saehan Jo and Immanuel Trummer. Thalamusdb: Approximate query processing on multi-modal data. *Proceedings of the ACM on Management of Data*, 2(3):1–26, 2024.

[6] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. Sembench: A benchmark for semantic query processing engines, 2025.

[8] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[9] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, 2025.

[10] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. Optimizing LLM queries in relational data analytics workloads. In *Eighth Conference on Machine Learning and Systems*, 2025.

[11] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Representation Learning*, volume 2025, pages 34433–34448, 2025.

[12] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators and their optimization: Enabling llm-based data processing with accuracy guarantees in lotus. *Proc. VLDB Endow.*, 18(11):4171–4184, September 2025.

[13] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Karthik Kefato, Tate Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems (MLSys)*, 5, 2023.

[14] Kangkang Qi, Dongyang Xie, Wenbo Li, Hao Zhang, Yuanyuan Zhu, Jeffrey Xu Yu, and Kangfei Zhao. Sema: A high-performance system for llm-based semantic query processing, 2026.

[15] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael J. Cafarella. Abacus: A cost-based optimizer for semantic operator systems. *CoRR*, abs/2505.14661, 2025.

[16] Gabriele Sanmartino, Matthias Urban, Paolo Papotti, and Carsten Binnig. The stretto execution engine for llm-augmented data systems, 2026.

[17] Dario Satriani, Enzo Veltri, Donatello Santoro, Sara Rosato, Simone Varriale, and Paolo Papotti. Logical and physical optimizations for sql query execution over large language models. *Proc. ACM Manag. Data*, 3(3), 6 2025.

[18] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. Docetl: Agentic query rewriting and evaluation for complex document processing. *Proc. VLDB Endow.*, 18(9):3035–3048, September 2025.

[19] Matthias Urban and Carsten Binnig. CAESURA: language models as multi-modal query planners. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024.

[20] Matthias Urban, Vu Huy Nguyen, Gabriele Sanmartino, Paolo Papotti, and Carsten Binnig. Selectivity estimation for semantic filters on image data, 2026.

[21] Lindsey Linxi Wei, Shreya Shankar, Sepanta Zeighami, Yeounoh Chung, Fatma Ozcan, and Aditya G. Parameswaran. Multi-objective agentic rewrites for unstructured data processing, 2026.

[22] Sunny Yasser, Anas Dorbani, and Amine Mhedhbi. Factorized and vectorized execution: Optimizing analytical and semantic queries over relations. *Proc. ACM Manag. Data*, 4(3), 2026.

[23] Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. Cut costs, not accuracy: Llm-powered data processing with guarantees, 2025.

Compositional Online Learning for Semantic Data Processing Systems

Paweł Liskowski Fuheng Zhao Benjamin Han Anupam Datta Dimitris Tsirogiannis
Snowflake Inc.
name.surname@snowflake.com

Abstract

An LLM call in a semantic data processing system is expensive enough to dominate query cost, yet slow enough to hide a CPU-side learner's update behind its round-trip. In production, LLM compute accounts for 80–90% of query cost, and each call costs 10^5–$10^7\times$ a relational predicate. The latency window inverts a design constraint of classical adaptive query processing, where online learners had to stay lightweight to avoid dominating the predicates they optimize. At LLM latency, per-call gradient steps and per-batch threshold solves fit inside the round-trip.

We develop *compositional online learning at the LLM call boundary*: a framework for combining online-learning components in semantic data processing systems. Each component makes execution-time decisions and refines its learned artifacts online. The design space spans two axes, decision granularity and learner update cadence, and the components share a single learning pattern that hides each trainer step inside the next LLM round-trip.

A production case study in Cortex AISQL composes three components: a memoization layer, an online per-call filter-ordering learner, and an online per-batch cascade-routing learner. A conditional cost decomposition assigns each learning component to a distinct factor of per-row LLM cost. Under independence, the two learning components compose multiplicatively to an 11.4× upper bound on a representative conjunction-filter workload. Self-selection at the cascade boundary, sample-budget shrinkage, and selectivity-estimation drift reduce it to a realistic figure near 8×.

1 Introduction

Semantic data processing systems extend SQL with operators that invoke a Large Language Model (LLM) on every qualifying row to analyze unstructured data at scale. Industrial systems (Cortex AISQL [1], BigQuery AI [2], Microsoft Fabric AI Functions [3], SEMA-SQL [4]) and academic systems (LOTUS [5], Palimpzest [6], Quest [7], ThalamusDB [8], SWAN [9]) all share the same cost profile, with per-row LLM calls dominating query cost. We propose a framework for optimizing semantic data processing systems through *compositional online learning at the LLM call boundary*. Classical adaptive query processing operates at relational latency, where online learners must be lightweight to avoid dominating the predicates they aim to optimize. At LLM latency, that constraint inverts. Each round-trip is wide enough to hide a CPU-side learner's update, opening a regime where multiple online-learning components can refine the system's per-call decisions during query execution. The resulting design point raises a new question: how do multiple online-learning components compose?

Cortex AISQL, a production deployment of native semantic operators [1], makes the cost case concrete. A single `AI_FILTER` over a million-row table can trigger up to a million LLM calls, each 10^5–$10^7\times$ more expensive than a relational predicate [10]. Production data shows that LLM compute accounts for 80–90% of total query cost, with end-to-end latencies measured in hours rather than seconds [1]. Each call takes hundreds of milliseconds, so a CPU-side learner's update fits inside the round-trip rather than on the critical path.

The composition question is largely open. Recent semantic-query systems [5, 6, 11–14] optimize individual components offline or at compile time: per-query selectivity samples for filter ordering, per-deployment thresholds for cascade routing, per-pipeline static cost models. Classical adaptive query processing (LEO [15], Eddies [16], POLAR [17], mid-query re-optimization [18, 19]) runs its learners online but at relational latencies, so the techniques transfer in spirit but not in scale. Across both lines of work, components are studied in isolation. To our knowledge, no prior work has analyzed how multiple online-learning components in the LLM-bound regime interact, or whether their gains stack.

We instantiate the framework in Cortex AISQL with two concrete learning components and a non-learning baseline. Filter ordering (Larch [20]) refits a per-predicate selectivity model on every LLM outcome and chooses the next predicate per row at online per-call cadence. Cascade routing (GAMCAL [21]) refits a calibrated GAM on a doubling schedule and routes each row through a cheap proxy or to the oracle at online per-batch cadence. Response caching provides the non-learning baseline by memoizing exact prior calls. We use these three components to develop a composition theory analytically. The two learning components target distinct cost factors and compose multiplicatively under independence. Three cross-component interactions then lower the realistic figure on a representative conjunction-filter workload.

The contributions of this paper are:

1. A framework for **optimizing semantic data processing systems through compositional online learning at the LLM call boundary**, organized around a layered design space (decision granularity $\times$ learner update cadence) and a shared learning pattern that hides each CPU-side learner's update inside the next LLM round-trip, so online per-call gradient steps and online per-batch threshold fits become feasible at scale.

2. An initial step towards a **composition theory** for the framework: a sequential composition rule with cost semantics, a conditional cost decomposition that assigns each learning component to a distinct factor of per-row LLM cost, and three named cross-component interactions. The analytical case study yields an 11.4$\times$ upper bound under independence and a roughly 8$\times$ realistic figure on a representative conjunction-filter workload.

2 The Framework: Online-Learning Composition at the LLM Call Boundary

The LLM round-trip is wide enough to hide a CPU-side online learner's update. That observation drives the framework and has two consequences. First, it admits a class of in-query optimizations that classical adaptive query processing could not afford (§2.1). Second, it makes one structural pattern repeat across all layers that exploit it (§2.2). The remainder of this section develops the design space the pattern lives in (§2.3), where response caching is the simplest case.

2.1 Why Classical Adaptive Query Processing Falls Short

Adaptive query processing was designed for a regime where predicates evaluate in nanoseconds to microseconds. At that budget, online learning has room only for the lightest mechanisms: tuple-level routing as in Eddies [16], plan-shape switches as in mid-query and progressive re-optimization [17–19], or coarse cardinality re-estimation as in feedback-driven optimizers [15]. Anything more elaborate would dominate the very predicate cost it aims to reduce.

Semantic operators run at a different latency entirely. Per-call LLM latency is hundreds of milliseconds to seconds, which is 10^5–$10^7\times$ a relational predicate's per-call cost [10, 20]. Some operators issue many

such calls per query: hierarchical text aggregations invoke the LLM at multiple reduction stages, and semantic joins evaluate a predicate on every candidate row pair.

The latency budget defines a new design point. Online per-call gradient updates, online per-batch threshold fits, and combinatorial planning passes over an expression tree all become feasible because anything that fits in a few milliseconds of CPU computation can hide inside the LLM call's round-trip. AQP literature explicitly argues for *lightweight* online learners because relational predicates run in nanoseconds. The latency-window observation inverts that design constraint at LLM latency, where the predicate-cost-dominated optimization rule still holds but online learners can be substantively heavier without violating it. Classical AQP techniques carry over in spirit, but the affordable learner footprint is far larger.

2.2 Online Learning Behind the LLM Call

One structural pattern recurs across every layer that operates online in the LLM-bound regime. Each LLM call generates one or more supervision signals (the call's output, an oracle judgment on a sampled row, or a ground-truth label). The system buffers them as they arrive, and a learner runs concurrently with the next round's LLM call(s), consuming the buffer and writing back to a learned artifact that the optimizer consults on the next decision.

Figure 1 traces the three-phase cycle. In Phase 1, the optimizer queries the current artifact to choose the next decision, and a background thread is dispatched to run a training step on supervision signals buffered from prior rounds. In Phase 2, the chosen evaluation is sent to the LLM, the training step completes inside the round-trip, and the trainer writes back to the artifact. In Phase 3, the LLM's outcome is appended to the supervision buffer for the next trainer step to consume. The trainer thus stays one round behind the supervision but never blocks it.

The pattern is structurally LLM-native: the same updates that would dwarf a relational predicate fit inside the LLM call's round-trip, hidden behind it rather than added to it. In Cortex AISQL, training steps run 7–11 ms across both variants of Larch [20] while LLM evaluations take hundreds of milliseconds, so the trainer's compute fits inside one round-trip. An ablation on the one-round staleness shows that deferring the update keeps per-query token cost within ±0.6% on average across both Larch variants and three datasets. In Larch's deployment, the one-round delay is benign: the training cost is paid in idle CPU cycles, not in the latency budget.

The cadence dictates the affordable update method. At online per-call cadence, the update must fit in one LLM round-trip of typically hundreds of milliseconds, so it stays a lightweight CPU step such as a single gradient. At online per-batch cadence, the update hides behind an entire batch of round-trips (N calls $\times$ per-call latency), so it can be a heavier method such as a Bayesian update or a constrained optimization problem. The two cadences are not interchangeable. Each admits a class of update methods the other cannot afford in practice. Deployment semantics are the same in both cases. The learned artifact is consulted from the first call and refined as new supervision arrives.

Hiding, however, does different work at the two cadences. Online per-call refits lie on the critical path of every LLM call, so a synchronous variant would inflate every call's latency by the trainer's compute. Online per-batch refits fire infrequently (the typical schedule yields $O(\log n)$ events per query), so a synchronous variant would stall only those rare events. Hiding at the online per-batch cadence is a latency-smoothing optimization rather than an online per-call necessity. By contrast, response caching sits at the same per-call boundary but relies on exact memoization, not statistical estimation.

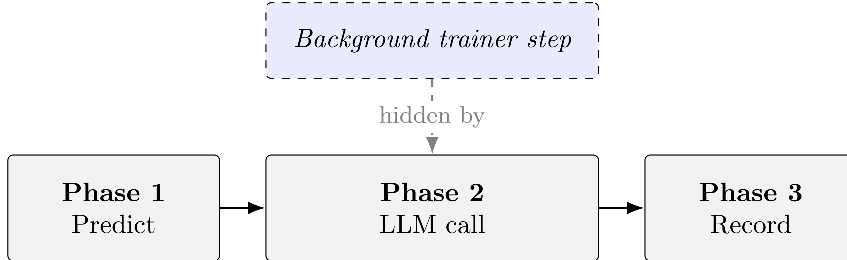

Figure 1: The pipelined online-learning pattern. Phase 1 predicts the next decision and dispatches a background trainer step. Phase 2 issues the LLM call, which hides the trainer's compute. Phase 3 records the outcome into the optimizer's state. §4 instantiates this at per-call cadence (one gradient step per LLM call). §5 instantiates it at per-batch cadence (calibration fit and threshold solve on a doubling schedule).

Table 1: The layered design space. Three rows are populated by the AISQL instantiation: response caching (§2.3), filter ordering (§4), and cascade routing (§5). The remaining four rows name LLM-bound research directions revisited in §8.1. AQP-style layers populate analogous decisions at relational latency and are discussed in §7 rather than in this LLM-bound design space.

Layer	Decision granularity	Learner update cadence	Artifact / state	Instance
Response caching	per-call	none (memoization)	exact-match response cache	AISQL response cache (§2.3)
Filter ordering	per-call	online per-call	selectivity model + DP solver	**Larch** (§4)
Cascade routing	per-row	online per-batch	calibrated GAM probability map	**GAMCAL** (§5)
Prompt caching	per-call	none (memoization)	prefix cache + affinity map	future direction (§8.1)
Model substitution	per-batch	one-shot at compile time	distilled student net	future direction (§8.1)
Prompt structure	per-call	online per-batch	prompt template / rewriting rules	future direction (§8.1)
Adaptive batching	per-batch	online per-batch	batch-size schedule	future direction (§8.1)

2.3 The Layered Design Space

Table 1 enumerates the framework's design space along two axes. Each row is a *layer*, a category of execution-time decision the system makes about LLM invocation, and each concrete implementation occupying a layer is a *component*. The framework composes components, and the design space is a typology of the layers components can occupy.

Decision granularity asks at what unit each layer decides whether or how the LLM is invoked: per-call, per-row, or per-batch in the LLM-bound regime considered here. Other granularities (per-tuple, per-pipeline, per-query) appear in classical AQP. *Learner update cadence* asks when the layer's learned artifact is refined relative to query execution: online (per-call or per-batch), one-shot at compile time, offline, or none (memoization). Response caching, filter ordering, and cascade routing are the three layers the AISQL instantiation fills.

The AISQL instantiation exercises both axes. Filter ordering (Larch, §4) runs at online per-call cadence and targets $\bar{e}$, the expected number of LLM-evaluated predicates per row, through short-circuit ordering. Cascade routing (GAMCAL, §5) runs at online per-batch cadence and targets p_{deleg}, the fraction of evaluated predicates sent past the proxy to the oracle, through calibrated thresholds. Both cost factors are formalized in §6.2.

The two components compose multiplicatively under independence. §6 formalizes this as a sequential rule with cost semantics, decomposes the per-row LLM cost across the two components, and catalogs the cross-component interactions that set the realistic figure. Response caching, the third such cell, has no learner or update cadence: a hit on an exact-match key returns the stored response and suppresses

both learning components, so savings track the duplicate fraction. Because hit rates can correlate with predicate position or row subpopulation, the cost model conditions on the non-cached path instead of parameterizing cache effects.

3 The Cortex AISQL Substrate

Cortex AISQL is a production system that embeds native semantic operators in relational SQL and the case study for this framework [1]. It provides six operators: `AI_FILTER` for per-row boolean predicates, `AI_CLASSIFY` for per-row labels from a fixed set, `AI_COMPLETE` for text generation, `AI_AGG` and `AI_SUMMARIZE_AGG` for text-column aggregations, and `AI_JOIN` for semantic joins over row pairs. Each invokes an LLM on every qualifying row or row pair and composes with relational operators in one declarative statement. The two learning components target `AI_FILTER`, so the cost decomposition in §6.2 is scoped to that operator and generalizes to any per-row semantic predicate.

To the optimizer, these operators are black boxes whose cost and selectivity are both unknown at compile time. Per-row cost depends on the model, prompt length, and inference infrastructure, while selectivity depends on a natural-language predicate with no histogram to read against. AISQL's existing optimizer addresses both unknowns in two stages. At compile time, cost-aware placement pulls expensive `AI_FILTER` predicates above joins and orders semantic predicates by relative cost. At runtime, coarse per-predicate cost and selectivity statistics trigger a reorder when a different order proves more effective on the observed batch. The framework's two learning components extend this runtime side, replacing passively collected population statistics with per-outcome online refinement.

4 Filter Ordering

Because AND/OR are commutative, the evaluation order of an `AI_FILTER` conjunction or disjunction changes only the cost, never the row's truth value. Order determines how often short-circuit reduction prunes the remaining predicates, and the optimal order at each row depends on per-row pass probabilities that classical optimizers cannot pre-compute. Larch instantiates the filter-ordering component on AISQL `AI_FILTER` conjunctions. It refits a per-predicate selectivity model after every LLM outcome and reorders the remaining predicates per row, both at online per-call cadence.

Larch-Sel decomposes the problem into two pieces, an online per-predicate selectivity estimator and an exact dynamic-programming solver over the AND/OR tree. The estimator is a two-layer MLP with roughly 144K trainable parameters, sigmoid output, and binary cross-entropy training. Its input is built from pre-computed document and predicate embeddings, produced at ingestion and exposed at query time at a cost roughly two orders of magnitude below an `AI_FILTER` evaluation [20]. Each embedding is first projected to 64 dimensions via learned linear maps $\mathbf{W}_{\text{doc}}$ and $\mathbf{W}_{\text{filter}}$, then combined into a feature vector that captures per-side identity, multiplicative interaction, and directional alignment:

$$\mathbf{x} = [\,\mathbf{d} \,\|\, \mathbf{f} \,\|\, \mathbf{d} \odot \mathbf{f} \,\|\, \cos(\mathbf{d}, \mathbf{f})\,].$$

All predicates share the MLP's weights, so the model transfers what it has learned about one predicate to the next from the first evaluation onward.

The DP solver evaluates the recurrence

$$\mathrm{OPT}(T') = \begin{cases} 0 & T' \text{ resolved,} \\ \min\limits_{f_i \in \mathrm{leaves}(T')} \big[\, c_i + \hat{s}_i \cdot \mathrm{OPT}(T'|_{f_i=\top}) + (1-\hat{s}_i) \cdot \mathrm{OPT}(T'|_{f_i=\bot}) \,\big] & \text{otherwise,} \end{cases} \quad (1)$$

Table 2: Token-overhead percentages relative to the per-row optimal lower bound on the Mix workload (50% AND, 50% OR) across three datasets [20]. Lower is better.

Dataset	PZ	Quest	**Larch-Sel**
GovReport	21.3%	33.6%	**5.1%**
PubMed	26.1%	29.5%	**6.5%**
BigPatent	28.3%	29.7%	**3.6%**

where $\mathrm{OPT}(T')$ is the minimum expected cost to resolve a partially evaluated tree T' given the current selectivity estimates. Solving the recurrence for one document runs in $O(n \cdot 3^n)$ time ($\approx$ 590K operations at n=10, $\approx$ 20 ms on a single CPU core). A sibling end-to-end RL variant, Larch-A2C, encodes the expression tree with a Gated Graph Neural Network and is the appropriate fit when predicate outcomes within a row are correlated.

The decomposition exploits a structural property: if per-predicate pass probabilities are accurate, the minimum-cost evaluation order can be solved exactly under the independence assumption. The DP recurrence (Eq. 1) generalizes Krishnamurthy, Boral, and Zaniolo's heterogeneous-cost ordering theorem [22] from flat conjunctions to arbitrary AND/OR trees. The original theorem itself builds on Ibaraki and Kameda's uniform-cost result [23]. Accurate per-row selectivities therefore suffice for an exactly-optimal ordering, leaving no residual signal for a learned policy to extract. Filter ordering is *estimation-dominated*, so a sample-efficient selectivity model matters more than a richer planner. Empirically, Larch-Sel reaches near-optimal performance from a few hundred rows on the smallest benchmark, while Larch-A2C closes the gap to the lower bound only as the horizon grows [20].

Larch-Sel instantiates the framework's learning pattern. Each `AI_FILTER` outcome is a binary label that supervises one BCE gradient step on the shared selectivity MLP. The step runs in a background thread during the next predicate's LLM call and completes in roughly 7 ms on CPU, inside the LLM's hundreds-of-millisecond round-trip. The pipeline is one step stale, with training during round t+1 using observations from round t. The asynchronous structure introduces one cross-component interaction with cascade routing.

Larch-Sel reduces total token-cost overhead by up to 19× relative to Palimpzest (PZ) [6] and Quest [7], with typical Mix-workload reductions in the 4–8× band [20]. Table 2 shows the per-dataset slice. The reduction comes from per-document selectivity carrying strictly more information than the population averages PZ and Quest learn from. The extra signal lets Larch-Sel match or exceed an oracle baseline with access to ground-truth global selectivities. PZ and Quest also pay an upfront 5% sampling cost in actual `AI_FILTER` calls to estimate those averages before execution begins. Larch-Sel pays no startup tax: it learns from the same `AI_FILTER` calls the query already performs.

Larch reduces $\bar{e}$, the expected predicate evaluations per row, through short-circuit ordering. Prior filter ordering instead fixes its estimates before execution, one-shot or offline (Palimpzest, Quest, SWAN, SEMA-SQL), as §7 details.

5 Cascade Routing

In cascade routing, we use two proxy score thresholds to partition the rows: rows below τ_{low} are rejected, rows at or above τ_{high} are accepted, and rows in the uncertain region $[\tau_{\text{low}}, \tau_{\text{high}})$ are delegated to the oracle LLM. AISQL's production cascade already pairs the two models. The proxy runs on every row at low cost, while the oracle accounts for the dominant per-call cost. Cost minimization reduces to choosing the threshold pair that balances classification quality against the oracle delegation rate. GAMCAL

instantiates the cascade-routing component on AISQL `AI_FILTER` rows, refitting a calibrated GAM and re-solving the thresholds as oracle labels accumulate across batches, at online per-batch cadence [21].

GAMCAL fits a Generalized Additive Model (GAM) [24, 25] that maps raw proxy scores to calibrated true-positive probabilities. Thresholds are then chosen to minimize a single cost-quality objective:

$$\min_{\tau_{\text{low}} \leq \tau_{\text{high}}} \alpha \cdot \frac{1 - \mathbb{E}[F_1(\tau_{\text{low}}, \tau_{\text{high}})]}{1 - \mathbb{E}[F_1(0.5, 0.5)]} + (1 - \alpha) \cdot \text{deleg}(\tau_{\text{low}}, \tau_{\text{high}}), \tag{2}$$

where $\alpha \in [0, 1]$ is a user-controlled trade-off knob, the first term is the normalized F_1 degradation against the no-delegation baseline, and deleg is the delegation rate. The calibrator f is a smoothing spline in log-odds space, fit on the accumulated oracle-label buffer S by penalized maximum likelihood with a smoothness penalty and a monotonicity constraint:

$$f^{\star} = \arg \max_{f \text{ monotone}} \sum_{(s,y) \in S} \left[y \log \sigma(f(s)) + (1-y) \log(1 - \sigma(f(s))) \right] - \lambda \int_0^1 \left(f''(s) \right)^2 ds. \tag{3}$$

GAMCAL routes each row through a stochastic calibrated score, computed as the GAM's mean prediction plus a Gaussian draw scaled by the calibrator's posterior standard error:

$$\tilde{g}_i = \sigma\big(f(s_i) + \Phi^{-1}(q_i) \cdot \text{se}(s_i) \big), \tag{4}$$

with s_i the row's proxy score, $q_i \sim U(0, 1)$ a per-row quantile drawn once and held fixed for the row's lifetime, σ the logistic, and Φ^{-1} the standard-normal quantile function. Rows in poorly calibrated regions of the score range receive more dispersed calibrated scores and tend to land in the uncertain region, so routing explores without an explicit importance-sampling scheme. Both calibrator and thresholds refit on a doubling schedule, holding retraining events to $O(\log n)$ per query. Each batch B_t adds $\rho|B_t|$ uniformly sampled rows from the uncertain region to a labeled buffer, and the first refit fires once both classes contain at least $n_{\min}$ labels.

A sibling target-based algorithm, SUPG-IT, handles workloads with explicit precision-recall contracts. We focus on GAMCAL because its smooth α knob composes directly with the filter-ordering component's cost lever. Two arguments motivate learned calibration over the SUPG family of statistical threshold estimators. First, modern proxy classifiers are often poorly calibrated [26]. Importance-weighted thresholding methods that assume calibrated proxies therefore produce conservative thresholds with high delegation rates. Second, the GAM generalizes oracle labels across the full proxy-score distribution. The spline interpolates each label at one score level into predictions at every other level, while running-statistics estimators in the SUPG family [27] extract information only from the score region around each observed label.

GAMCAL instantiates the framework's learning pattern at batch scale (Figure 1). Each refit runs concurrently with the spawning batch's oracle calls and is installed as the routing state at the start of the next batch. At a batch size of roughly 4,000 rows with oracle calls taking hundreds of milliseconds, the aggregate per-batch oracle window absorbs the CPU GAM fit and threshold solve. The pipeline lags by one batch: routing in batch $t{+}1$ uses calibration fit from labels through batch $t{-}1$. The lag mirrors at batch scale what filter ordering sees at call scale and introduces another cross-component interaction.

GAMCAL exceeds $F_1 = 0.95$ at its best operating point on every one of six benchmarks spanning classification, filtering, and join workloads [21]. To reach that threshold against LOTUS's single-pass SUPG cascade [5], GAMCAL needs up to 58% fewer oracle calls and leads on five of six benchmarks (Table 3). ArXiv is the exception: its proxy is already well-calibrated, so importance-weighted thresholding suffices and there is little for the GAM to recover. The advantage is largest on BoolQ and MMLU, where the proxy is reasonably accurate but poorly calibrated, so the GAM's smooth interpolation gains a clear sample-efficiency edge over running statistics.

Table 3: Minimum delegation rate d required to reach $F_1 \geq 0.95$ across all six benchmarks [21]. Lower is better. Bold marks the best algorithm per row. GAMCAL leads on five of six datasets, often by a wide margin.

Dataset	SUPG-SP (LOTUS)	SUPG-IT	GAMCAL
ArXiv	**55.6%**	61.9%	61.6%
BoolQ	69.7%	58.2%	**29.3%**
IMDB	69.9%	69.1%	**68.4%**
MMLU	67.4%	62.9%	**43.9%**
NYT	21.9%	22.4%	**17.0%**
SST-2	28.6%	31.6%	**21.2%**

GAMCAL's calibrated thresholds reduce p_{deleg}, the fraction of evaluated predicates delegated past the proxy to the oracle and the second factor of the cost decomposition. §6 composes this reduction with Larch's into the multiplicative bound, and §7 sets GAMCAL's per-batch calibration against the offline and one-shot cascades (FrugalGPT, SUPG, LOTUS).

6 Composition Across Learning Components

6.1 Sequential Composition with Cost Semantics

The two learning components, filter ordering (Larch) and cascade routing (GAMCAL), run in a fixed order: filter ordering first, then per-row cascade routing on each evaluated predicate. Total cost depends on each component's standalone behavior and on how the upstream reshapes the distribution the downstream sees, so we state composition as a Hoare-style rule with cost semantics. The rule makes the multiplicative-decomposition assumption explicit and classifies cross-component interactions into forward precondition mismatches and a backward supervision flow.

Following the standard form of Hoare's sequential composition rule and its probabilistic extensions with cost semantics [28, 29], we write each learning component as a triple

$$\langle D_{\text{in}},\, S,\, D_{\text{out}} \rangle_{c(D_{\text{in}})},$$

where the precondition D_{in} is the input distribution over rows or (row, predicate) pairs, the program S is the component's online learner plus decision rule, the postcondition D_{out} is the distribution of items the component forwards to the next component, and $c(D_{\text{in}})$ is the expected per-input LLM cost the component pays as a function of the input distribution. Parameterizing cost by D_{in} rather than a nominal value matters because Larch and GAMCAL train against the rows they actually see, so their realized cost shifts with the distribution the upstream produces.

Sequential composition combines two triples into one. The downstream's cost is weighted by the upstream's fan-out $\phi_1(D_0)$, the expected number of items S_1 forwards per input:

$$\frac{\langle D_0,\, S_1,\, D_1 \rangle_{c_1(D_0)} \quad \langle D_1,\, S_2,\, D_2 \rangle_{c_2(D_1)}}{\langle D_0,\, S_1; S_2,\, D_2 \rangle_{c_1(D_0)\,+\,\phi_1(D_0)\cdot c_2(D_1)}}. \tag{5}$$

Composition is sound only when S_2's precondition is the same distribution S_1 produces, mirroring Hoare logic's postcondition-precondition matching.

For the Larch + GAMCAL composition, Larch (S_1) makes no LLM calls of its own ($c_1 \equiv 0$), forwards $\bar{e}$ evaluated predicates per row in expectation ($\phi_1(D_0) = \bar{e}$), and produces a postcondition

D_1 over (row, predicate) pairs sent to the cascade. GAMCAL (S_2) pays per-evaluated-predicate cost $c_2(D_1) = c_{\text{proxy}} + p_{\text{deleg}}(D_1) \cdot c_{\text{oracle}}$, with p_{deleg} the delegation rate evaluated on Larch's actual postcondition. Substituting into Eq. 5 yields the per-row form

$$\bar{e} \cdot \big(c_{\text{proxy}} + p_{\text{deleg}}(D_1) \cdot c_{\text{oracle}}\big). \tag{6}$$

When D_1 matches GAMCAL's calibration distribution $D_1^\star$, $p_{\text{deleg}}(D_1)$ takes its nominal calibrated value, and Eq. 6 is the matched-precondition instance of Eq. 5. §6.2 states the workload assumptions behind the form.

The per-row form depends only on the realized values of $\bar{e}$ and $p_{\text{deleg}}(D_1)$, not on the runtime sequence in which Larch chooses predicates and the cascade evaluates them. Cross-component interactions are the mechanisms that shift those values. When $D_1 \neq D_1^\star$, GAMCAL's per-item cost picks up a multiplicative penalty $c_2(D_1)/c_2(D_1^\star)$, and the composition rule classifies the three by direction. Self-selection at the cascade boundary (κ_1) and sample-budget shrinkage (κ_2) are forward precondition mismatches, while selectivity-estimation drift (κ_3) is a backward supervision flow from cascade outputs into Larch's selectivity learner.

The two-component rule extends by induction to longer chains of learning components. Each component whose cost is parameterized by its input distribution fits the triple notation, and the composed per-input cost decomposes by the same fan-out-weighted accounting. Two extensions lie outside the present formulation: branching topologies, where one component fans out to multiple downstreams, and backward supervision flows like κ_3, where downstream outputs return as labels into an upstream learner. Both are future work.

6.2 Conditional Cost Decomposition

For the conjunction-filter workload, the per-row form (Eq. 6) becomes an explicit cost equation once its four parameters and underlying assumptions are stated. The natural unit of analysis is *expected predicate evaluations* per row ($\bar{e}$), because an `AI_FILTER` conjunction with m predicates can incur up to m LLM calls per row in the unoptimized case.

For N rows flowing through a filter conjunction with m semantic predicates and per-predicate cascade routing, total cost factors as

$$\text{Cost} = N \cdot \bar{e} \cdot \big[\, c_{\text{proxy}} + p_{\text{deleg}} \cdot c_{\text{oracle}} \,\big], \tag{7}$$

with the four parameters defined as follows:

- $\bar{e} \in [1, m]$, the expected number of LLM-evaluated predicates per row on the non-trivial conjunction path, equal to m in the unoptimized baseline.
- c_{proxy}, the per-call proxy cost, equal to 0 in the no-cascade baseline.
- $p_{\text{deleg}} \in [0, 1]$, the per-predicate oracle delegation rate, equal to 1 in the no-cascade baseline.
- c_{oracle}, the per-call oracle cost, which dominates the unoptimized total.

Each learning component targets a distinct factor of Eq. 7. Larch reduces $\bar{e}$ via short-circuit ordering. The cascade reduces p_{deleg} via calibrated thresholds at the cost of paying c_{proxy} on every evaluated predicate. Under independence, the two reductions compose multiplicatively, the matched-precondition case of the sequential composition rule (§6.1).

The decomposition rests on three assumptions: per-call costs are uniform across rows and predicates, predicate selectivities are independent within a row, and the analysis is conditioned on a cache miss.

Table 4: Cost relative to the no-optimization baseline at $m = 5$ and $c_{\text{proxy}}/c_{\text{oracle}} = 0.05$. The two-component upper bound is 11.4× under §6.2's independence assumption and matched-precondition calibration ($D_1 = D_1^\star$ in the sense of §6.1). §6.4 catalogs cross-component interactions that lower the realistic figure to roughly 8×.

Configuration	$\bar{e}$	p_{deleg}	Cost (rel.)
Baseline (no optimization)	5.0	1.0	1.00
+ Filter ordering alone	≈ 1.25	1.0	≈ 0.25
+ Cascade routing alone	5.0	≈ 0.30	≈ 0.35
Both learning components (upper bound)	≈ 1.25	≈ 0.30	$\approx$ **0.0875** (11.4×)

The cache-miss conditioning means Eq. 7 describes the non-cached path, with cache hits held outside the model rather than parameterized within it. The decomposition extends to non-conjunctive expression trees by generalizing $\bar{e}$ through the DP recurrence (Eq. 1) over AND/OR nodes.

The independence and cache-miss assumptions each hide a workload interaction. Within-row independence fails because rows that survived earlier predicates have correlated pass probabilities downstream, and the self-selection interaction κ_1 captures the resulting shift in GAMCAL's input distribution. A joint cache-and-learning model would require an analogous workload-specific assumption, since cache hit rates vary with predicate position, row subpopulation, and routing decisions. Quantifying that variation requires evidence beyond what this paper presents, so cache effects remain outside the displayed decomposition.

6.3 An Analytical Case Study

A representative conjunction-filter workload has $N = 10^6$ rows, $m = 5$ semantic predicates in the WHERE clause, and proxy-to-oracle cost ratio $c_{\text{proxy}}/c_{\text{oracle}} \approx 0.05$. Each row of Table 4 substitutes one configuration's parameter values into Eq. 7 and reports the resulting cost relative to the no-optimization baseline.

$\bar{e} \approx 1.25$ at $m = 5$ corresponds to a 4× cut in expected predicate evaluations, at the lower end of Larch-Sel's typical 4–8× Mix-workload band and well below its 19× ceiling against Palimpzest and Quest [20]. Counting LLM calls instead of token cost keeps $\bar{e} \geq 1$, since every row evaluates at least one predicate to resolve the conjunction. $p_{\text{deleg}} \approx 0.30$ falls inside GAMCAL's 17–68% range of minimum delegation rates to reach $F_1 \geq 0.95$ across six benchmarks [21], near the easier-to-cascade end. The cost ratio $c_{\text{proxy}}/c_{\text{oracle}} \approx 0.05$ follows from inference-cost scaling: compute scales with model parameters at roughly 10× between an 8B proxy and a 70B oracle, and drops further when the smaller model runs on cheaper hardware [10].

The final row substitutes the two optimized values into Eq. 7:

$$\text{Cost}_{\text{both}} = N \cdot 1.25 \cdot \left[0.05 \cdot c_{\text{oracle}} + 0.30 \cdot c_{\text{oracle}}\right] = N \cdot 0.4375 \cdot c_{\text{oracle}} \approx 0.0875 \cdot \text{baseline}.$$

The naive product of the two standalone cost ratios, $0.25 \times 0.35 = 0.0875$, matches the formula exactly because c_{proxy} enters the per-call cost factor that both components multiply through. The 11.4× figure is therefore the upper bound under independence and matched-precondition calibration ($D_1 = D_1^\star$). Three cross-component interactions push the realistic figure to roughly 8×.

The 11.4× figure holds approximately constant for $m \geq 4$, where Larch's reduction does not hit the $\bar{e} \geq 1$ floor. Cost ratio and p_{deleg} matter more: a cost ratio of 0.10 compresses it to about 10×, and the harder end of GAMCAL's range ($p_{\text{deleg}} \approx 0.68$) to about 5.5×.

6.4 Cross-Component Interactions

Three interactions lower the 11.4× upper bound, each a multiplicative penalty κ_i on one term of Eq. 7. We treat the three as independent. Mechanistic coupling between κ_1 and κ_3 would push the realistic figure below 8×.

Self-selection at the cascade boundary (κ_1). Larch routes only conjunction-surviving rows to a given cascade-evaluated predicate, and survivors skew toward harder cases with proxy scores nearer the decision boundary. The per-batch refresh tracks shift between batches, but steady-state p_{deleg} on this self-selected stream stays above the global rate. We bound the penalty at $\kappa_1 \in [1.0, 1.5]$ on p_{deleg}. At the high end, delegation rises by roughly 50% over the global rate.

Sample-budget shrinkage (κ_2). The cascade draws its ρ-fraction sample budget per batch, so Larch's short-circuiting shrinks the effective batch at later predicates and yields fewer oracle labels per refinement there, slowing convergence. Because the shortfall peaks before thresholds stabilize, the penalty $\kappa_2 \in [1.0, 1.3]$ falls on early-query p_{deleg} alone, its upper end a cascade running at about 70% of steady-state effectiveness during cold start.

Selectivity-estimation drift across the cascade (κ_3). Unlike the forward mismatches κ_1 and κ_2, this one runs backward. Larch's selectivity MLP trains on predicate outcomes, but when a downstream filter is served by a cascade that returns proxy decisions in its accept and reject regions, those labels are no longer pure oracle labels and the gradient signal grows noisier in proportion to the accept-and-reject fraction. The drift puts $\kappa_3 \in [1.0, 1.4]$ on $\bar{e}$, reaching its maximum at an accept-and-reject fraction near 70%, where degraded selectivity precision yields less aggressive short-circuiting.

κ_2 contributes during the cascade's pre-steady-state phase only, so its contribution averages out over a long operator. On a 10^6-row query at batch size 4,096 and $\rho = 0.10$, the pre-steady-state phase covers roughly the first 1% of batches under the doubling schedule. The operator-level sweep holds $\kappa_2 = 1$ and combines κ_1 on p_{deleg} with κ_3 on $\bar{e}$, giving:

$$\text{Cost}_{\text{rel}} = 0.25 \cdot \kappa_3 \cdot (0.05 + 0.30 \cdot \kappa_1). \tag{8}$$

At midpoint values $\kappa_1 = 1.25$ and $\kappa_3 = 1.2$, this yields $\sim 7.8\times$, which we round to 8× as the realistic figure. The pessimistic corner ($\kappa_1 = 1.5$, $\kappa_3 = 1.4$) yields $\sim 5.7\times$, and the optimistic corner ($\kappa_1 = \kappa_3 = 1$) recovers the 11.4× upper bound exactly.

7 Related Work

On the framework's two axes (decision granularity, learner update cadence), the AISQL instantiation is the only system to fill the *online per-call* and *online per-batch* cells in the LLM-bound regime. Other LLM-bound systems operate at the same decision granularities but at one-shot or offline cadences, and classical AQP runs online only at relational latency, where the latency-window observation does not apply. At LLM latency, a constraint the AQP and cascade literatures treated as fixed no longer holds: one-shot or offline cadences become optional rather than necessary.

AQP at relational latency. Classical AQP spans per-tuple, per-pipeline, and per-query granularities at online cadences (Eddies [16]; POLAR and mid-query reopt [17–19]; LEO [15] and ThalamusDB [8]), but only at relational latency, where the predicate-cost constraint analyzed in §2.1 forces sub-microsecond learners. The latency window does not apply there, so these techniques apply conceptually but at a far smaller learner footprint.

The filter-ordering cadence argument. Prior LLM-aware filter ordering fixes its selectivity estimates once, before execution. Palimpzest [6], Quest [7], SWAN [9], and SEMA-SQL [4] sit in the *one-shot at compile time* cell, learning population-level averages from a sampling pass and freezing

them. Larch instead refines a per-document selectivity model on every LLM outcome, occupying the *online per-call* cell that prior work leaves empty. Model substitution shows the same pattern at coarser granularity: UQE [12] distills a student model at compile time rather than online.

The cascade cadence argument. Prior cascade work populates the cascade-routing layer at two cadences, offline and one-shot at compile time. SUPG [27], FrugalGPT [30], and BigQuery proxy work [11] fit calibration sets or proxy classifiers before deployment and freeze them afterward. LOTUS [5] deploys a variant called SUPG-SP that fits its threshold on a per-query calibration sample at compile time, placing it in the *one-shot at compile time* cell. GAMCAL [21] extends both to an *online per-batch* cadence in which calibration is refreshed during query execution on a doubling schedule. The framework's distinctive contribution to cascade theory is the online per-batch cadence, made feasible by the latency window.

The online cadences the latency window opens thus remain largely empty in practice: across LLM-aware filter ordering, cascade routing, and model substitution, only Larch and GAMCAL sit at online cadences, while the rest occupy compile-time or offline cells consistent with the AQP inversion argument.

8 Discussion

The framework and its composition analysis leave four open directions: filling the rest of the design surface, tightening composition across the components in place, extending the learning loop across query boundaries, and reconciling the components' heterogeneous quality semantics.

8.1 Future Layers in the Design Space

Response caching shows that a productive layer can rely on memoization rather than online estimation; the remaining design-space rows stay open. *Prompt caching* shares that mechanism, but its hit rates correlate with predicate position and routing in ways the displayed cost decomposition does not parameterize. *Model substitution* generalizes the cascade to per-batch granularity: UQE's distilled student [12] sits at the one-shot cell, and the latency window admits an online per-batch variant that refits both student and gate. *Prompt structure* shapes the call itself; AISQL's join-to-classify rewrite [1] is a compile-time instance, while an online variant could key rewrite rules to observed outcomes, dropping error-correlated exemplars and restoring hallucination-correlated hints. *Adaptive batching* trades per-call latency, throughput, and quality, and fits the online per-batch cadence directly. These layers couple through shared prompt, model, and batch choices, so their refit signals belong in a joint controller rather than in isolated components.

Joint optimization across components. The two learning components are tuned independently today, but the interactions suggest concrete couplings a controller could exploit: refit the cascade against the post-Larch self-selected stream (κ_1), size its sample budget to the post-short-circuit batch at each predicate position (κ_2), and down-weight Larch's gradient signal from labels collected in the cascade's accept and reject regions (κ_3). The natural interface generalizes GAMCAL's single-α knob across both components, exposing the composed system's cost-quality frontier rather than each component's in isolation. The same controller is the natural home for future model-substitution, prompt-structure, and adaptive-batching layers, whose decisions couple back to filter ordering and cascade routing.

Cross-query learning. Both components are intra-query: state builds from the first call and is discarded at query end. Production AISQL workloads reuse predicate templates across queries, so per-predicate selectivity histories, per-template calibration curves, and distilled cost models could carry over. Persistence faces prompt and data drift, template re-binding, and isolation boundaries; a per-template warm-start cache with bounded staleness preserves the intra-query loop while removing

the cold-start tax for recurring predicates. The closest classical analogue is feedback-driven optimization such as LEO [15], adapted to artifacts richer than a histogram.

Quality contracts in production. The components' deployment criteria sit at different guarantee levels. Larch's gradient-driven refinement is best-effort: a bad prediction costs at most a few extra predicate evaluations on a row. The cascade's two thresholds control a cost-quality objective with explicit probabilistic semantics, and a sibling target-based variant (SUPG-IT [21]) delivers hard precision-recall guarantees at higher delegation. These form three tiers (guarantee, calibrated trade-off, best-effort) that compose loosely, so the system's guarantee is the weakest of its components. A unified contract surfacing per-component guarantees to the optimizer would let workloads swap in guarantee-preserving variants (SUPG-IT for the cascade; a budget-bounded Larch loop capping worst-case calls per row for filter ordering) and opt into a stricter overall tier. The layered framing makes such substitutions local rather than global.

9 Conclusion

We have developed a framework for online-learning composition at the LLM call boundary, built on the latency-window observation: a CPU-side learner's update fits inside the LLM round-trip, opening a regime classical adaptive query processing could not reach. Cortex AISQL populates three cells of the framework's design space. The two cost factors of an `AI_FILTER` operator are decomposed across two learning components. Filter ordering reduces the expected number of LLM-evaluated predicates per row, and cascade routing reduces the fraction of evaluated predicates that escalate to the oracle. Both components instantiate the same learning pattern, with filter ordering at an online per-call cadence and cascade routing at an online per-batch cadence. Each hides its training compute behind the round-trip of the LLM call it precedes. Response caching covers the same boundary through exact memoization rather than online estimation.

A Hoare-style sequential composition rule with cost semantics shows that the two learning components compose multiplicatively under independence, yielding an analytical upper bound of $11.4\times$ total cost reduction on a representative conjunction-filter workload. Three concrete cross-component interactions, classified by the rule as two forward precondition mismatches and one backward supervision flow, lower the realistic figure to roughly $8\times$. Each is structural rather than fundamental and admits a remedy at the optimizer level.

LLM-native data systems will be defined less by the semantics of any individual operator and more by the layered, latency-budget-aware optimization architecture in which the operators are embedded. The learning pattern is one expression of that architecture. The natural next steps are empirical validation of the composition argument at scale, cross-query learning, joint controllers across components, and future layers in the design space.

References

[1] Paweł Liskowski, Benjamin Han, Paritosh Aggarwal, Bowei Chen, Boxin Jiang, Nitish Jindal, Zihan Li, Aaron Lin, Kyle Schmaus, Jay Tayade, Weicheng Zhao, Anupam Datta, Nathan Wiegand, and Dimitris Tsirogiannis. Cortex AISQL: A production SQL engine for unstructured data. *arXiv preprint arXiv:2511.07663*, 2025.

[2] Jian He and Vaibhav Sethi. SQL reimagined for the AI era with BigQuery AI functions. Google Cloud Blog, Data Analytics, nov 2025. `https://cloud.google.com/blog/products/data-analytics/sql-reimagined-for-the-ai-era-with-bigquery-ai-functions`.

[3] Microsoft. Use AI functions (preview). Microsoft Learn, Fabric Data Warehouse Documentation, mar 2026. `https://learn.microsoft.com/en-us/fabric/data-warehouse/ai-functions`.

[4] Yin Lin, Tianjing Zeng, Zhongjun Ding, Rong Zhu, Bolin Ding, H. V. Jagadish, and Jingren Zhou. SEMA-SQL: Beyond traditional relational querying with large language models. *arXiv preprint arXiv:2604.23477*, 2026.

[5] Liana Patel, Siddharth Jha, Parth Asawa, Melissa Pan, Carlos Guestrin, and Matei Zaharia. Semantic operators: A declarative model for rich, AI-based analytics over text data. *arXiv preprint arXiv:2407.11418*, 2024.

[6] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. Palimpzest: Optimizing AI-powered analytics with declarative query processing. In *Conference on Innovative Data Systems Research (CIDR)*, 2025.

[7] Zhaoze Sun, Qiyan Deng, Chengliang Chai, Kaisen Jin, Xinyu Guo, Han Han, Ye Yuan, Guoren Wang, and Lei Cao. Quest: Query optimization in unstructured document analysis. *arXiv preprint arXiv:2507.06515*, 2025.

[8] Saehan Jo and Immanuel Trummer. ThalamusDB: Approximate query processing on multi-modal data. *Proc. ACM Manag. Data*, 2(3), May 2024.

[9] Fuheng Zhao, Divyakant Agrawal, and Amr El Abbadi. Hybrid querying over relational databases and large language models. In *Conference on Innovative Data Systems Research (CIDR)*, 2025.

[10] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[11] Yeounoh Chung, Rushabh Desai, Jian He, Yu Xiao, Thibaud Hottelier, Yves-Laurent Kom Samo, Pushkar Kadilkar, Xianshun Chen, Sam Idicula, Fatma Özcan, Alon Halevy, and Yannis Papakonstantinou. 100x cost & latency reduction: Performance analysis of AI query approximation using lightweight proxy models. In *Proceedings of the 2026 ACM SIGMOD International Conference on Management of Data*, 2026.

[12] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. UQE: A query engine for unstructured databases. In *Advances in Neural Information Processing Systems*, 2024.

[13] Fuheng Zhao, Jiayue Chen, Yiming Pan, Tahseen Rabbani, Sohaib, Divyakant Agrawal, Amr El Abbadi, Paritosh Aggarwal, Anupam Datta, and Dimitris Tsirogiannis. Access paths for efficient ordering with large language models. *arXiv preprint arXiv:2509.00303*, 2025.

[14] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. DocETL: Agentic query rewriting and evaluation for complex document processing. *Proceedings of the VLDB Endowment*, 18(9):3035–3048, 2025.

[15] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO – DB2's Learning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 19–28, 2001.

[16] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272. ACM, 2000.

[17] David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, and Matthias Boehm. POLAR: Adaptive and non-invasive join order selection via plans of least resistance. *Proceedings of the VLDB Endowment*, 17(6):1350–1363, 2024.

[18] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 106–117. ACM, 1998.

[19] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 659–670. ACM, 2004.

[20] Fuheng Zhao, Paweł Liskowski, Zihan Li, Benjamin Han, Puxuan Yu, Varich Boonsanong, Dimitris Tsirogiannis, and Anupam Datta. Larch: Learned query optimization for semantic predicates. *arXiv preprint arXiv:2606.07923*, 2026.

[21] Paweł Liskowski and Kyle Schmaus. Streaming model cascades for semantic SQL. *arXiv preprint arXiv:2604.00660*, 2026.

[22] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *Proceedings*

of the 12th International Conference on Very Large Data Bases (VLDB), pages 128–137, 1986.

[23] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[24] Trevor Hastie and Robert Tibshirani. Generalized additive models. *Statistical Science*, 1(3):297–310, 1986.

[25] Simon N. Wood. *Generalized Additive Models: An Introduction with R.* Chapman and Hall/CRC, 2nd edition, 2017.

[26] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning (ICML)*, pages 1321–1330, 2017.

[27] Daniel Kang, Edward Gan, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. Approximate selection with guarantees using proxies. *Proceedings of the VLDB Endowment*, 13(11):1990–2003, 2020.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[29] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected runtimes of randomized algorithms. *Journal of the ACM*, 65(5):30:1–30:68, 2018.

[30] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

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.

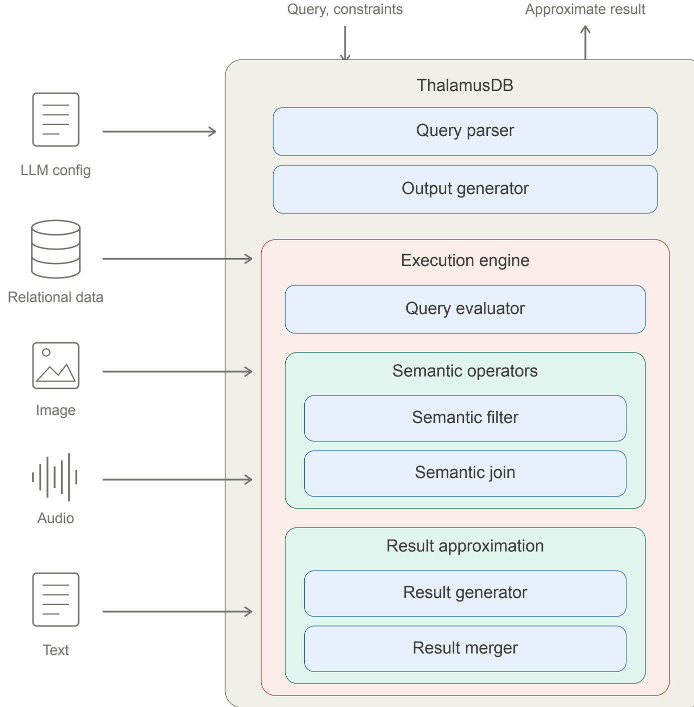

Figure 1: Overview of ThalamusDB system.

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.

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name ____________________

IEEE Member # ____________________

Mailing Address ____________________

____

Country ____________________

Email ____________________

Phone ____________________

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314

Non-profit Org.
U.S. Postage
PAID
Los Alamitos, CA
Permit 1398