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.

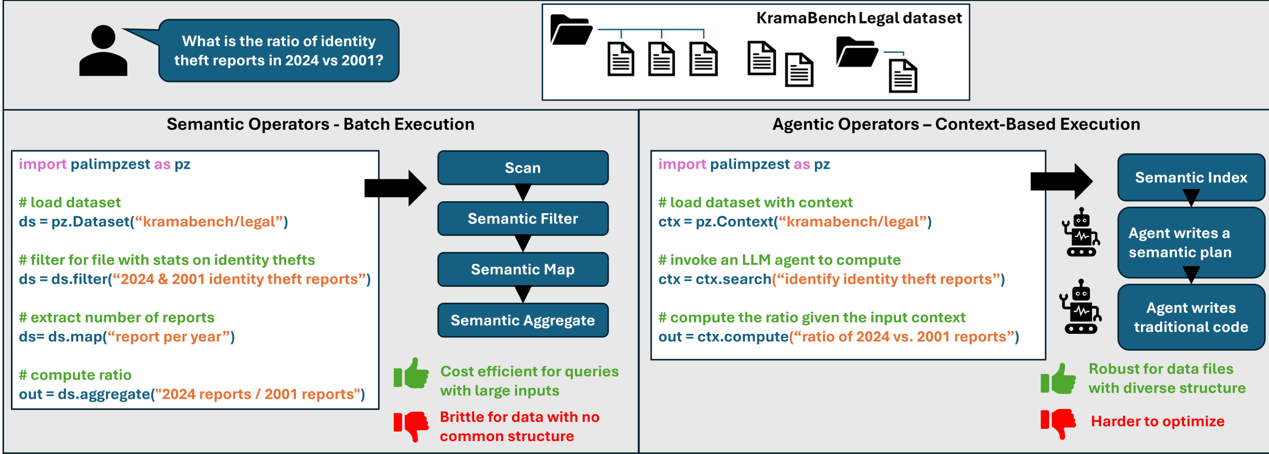

Figure 1: Overview of two different Palimpzest programs for the identity-theft query. The image compares a batch semantic operator pipeline (left) with one that leverages agentic `search` and `compute` (right).

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.