Abacus: A Cost-Based Optimizer for Semantic Operator Systems

Matthew Russo
MIT
mdrusso@csail.mit.edu

Chunwei Liu
MIT
chunwei@csail.mit.edu

Sivaprasad Sudhir
MIT
siva@csail.mit.edu

Gerardo Vitagliano
MIT
gerarvit@mit.edu

Michael Cafarella
MIT
michjc@csail.mit.edu

Tim Kraska
MIT
kraska@mit.edu

Samuel Madden
MIT
madden@csail.mit.edu

ABSTRACT

LLMs enable an exciting new class of data processing applications over large collections of unstructured documents. Several new programming frameworks have enabled developers to build these applications by composing them out of semantic operators: a declarative set of AI-powered data transformations with natural language specifications. These include LLM-powered maps, filters, joins, etc. used for document processing tasks such as information extraction, summarization, and more. While systems of semantic operators have achieved strong performance on benchmarks, they can be difficult to optimize. An optimizer for this setting must determine how to physically implement each semantic operator in a way that optimizes the system globally. Existing optimizers are limited in the number of optimizations they can apply, and most (if not all) cannot optimize system quality, cost, or latency subject to constraint(s) on the other dimensions. In this paper we present Abacus, an extensible, cost-based optimizer which searches for the best implementation of a semantic operator system given a (possibly constrained) optimization objective. Abacus estimates operator performance by leveraging a minimal set of validation examples, prior beliefs about operator performance, and/or an LLM judge. We evaluate Abacus on document processing workloads in the biomedical and legal domains (BioDEX; CUAD) and multi-modal question answering (MMQA). We demonstrate that, on-average, systems optimized by Abacus achieve 6.7%-39.4% better quality and are 10.8x cheaper and 3.4x faster than the next best system.

PVLDB Reference Format:
Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael Cafarella, Tim Kraska, and Samuel Madden. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. PVLDB, 19(5): 1060 - 1073, 2026.
doi:10.14778/3796195.3796215

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mitdbg/palimpzest.

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796215

1 INTRODUCTION

Industry and academia are increasingly using large language models (LLMs) to solve problems which require semantic understanding. These problems range from unstructured document processing [6, 12], to multi-modal question answering [4, 35, 42], to semantic search and ranking [36]. In order to achieve state-of-the-art performance on these tasks, practitioners often decompose the problem into modular subtasks within an AI program.

Recently, programming frameworks including Palimpzest [23], LOTUS [28], DocETL [33], and others [2, 19, 24, 30, 31] have proposed building these LLM-based applications out of **semantic operators**. Inspired by relational operators [11], semantic operators are AI-powered data transformations with natural language specifications. These include LLM-powered maps, filters, joins, aggregations, etc. and are useful for unstructured data processing tasks such as information extraction, summarization, ranking, and classification.

Developers can define a **semantic operator system** by writing a declarative AI program (e.g., in Palimpzest or a similar framework). The program defines a logical plan, which an optimizer can compile into a physical plan. For example, Figure 1 illustrates a use case where a researcher wishes to search for papers relevant to their interests. First, the program loads the papers and filters for ones related to data systems. Then, the program computes a summary of each paper's main contributions. Finally, the papers are classified as having high or low relevance to the author's research interests.

In order to execute this program, the optimizer must decide how to implement each semantic filter and map in terms of underlying physical operations (e.g., calls to LLMs). For example, given many LLMs of different sizes, the optimizer may simply need to choose which LLM to use for each operator. However, the optimizer may also be allowed to choose from more complex techniques such as using an LLM ensemble [38], reducing the input context before feeding it to an LLM [21], and more. With access to a handful of models and hyperparameters, a few techniques can provide an optimizer with thousands of physical implementation alternatives that trade-off operator quality, dollar cost, and latency (Section 4.1).

Goal. The optimizer's goal is to compile a semantic operator program to a physical plan which is (near-)optimal for the developer's objective with respect to system quality, cost, and latency. For example, in the center and right-hand side of Figure 1, we show two physical plans for two different optimization objectives. The

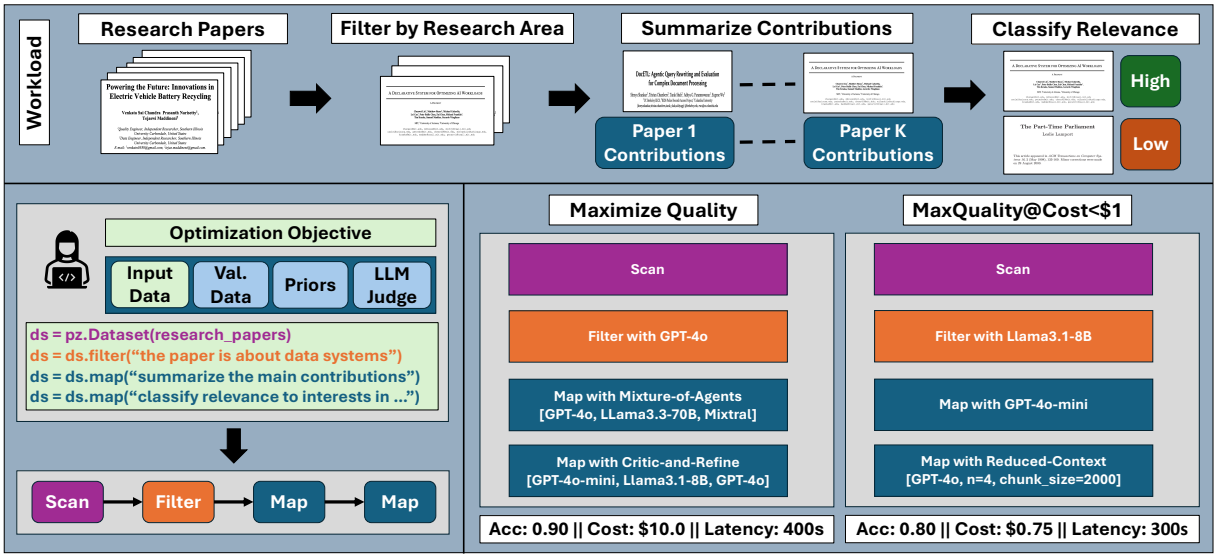

Figure 1: An illustration of Abacus compiling a program for a literature search workload (Top) into two different physical plans for two different optimization objectives. (Left) the user implements the workload in a Palimpzest program and provides the input data they wish to process and (optionally) validation data, priors, and/or an LLM judge which Abacus may use to guide its optimization. The program is compiled to a logical plan, which Abacus seeks to implement with a (near-)optimal physical plan. (Center) given the unconstrained objective of maximizing quality, Abacus produces a physical plan which achieves high quality for this task. (Right) given the objective of maximizing quality subject to a constraint of $1 in execution cost, Abacus produces a plan which satisfies the constraint while only trading-off a modest decrease in quality.

first plan is compiled with the goal of maximizing system output quality, while the second is compiled to maximize quality subject to spending less than $1 on processing the entire workload. Here, the cost-constrained plan uses lighter weight models and simpler strategies. Ideally, a cost-based optimizer can weigh the trade-offs of different physical operators to implement each plan optimally.

Our Approach. In this paper we describe Abacus, a new cost-based optimizer implemented in Palimpzest [23]. While some existing semantic programming frameworks have implemented optimizers, they typically only optimize for system quality, do not consider constraints on system cost or latency, and use a limited set of rewrite-style or proxy-based optimizations [28, 33]. By contrast, Abacus is a general-purpose, cost-based optimizer that optimizes system output quality, dollar cost, or latency with respect to zero or more constraints on the other dimensions. Abacus uses implementation and transformation rules to define a valid set of physical plans, similar to a Cascades query optimizer [8]. However, the uncertain nature of semantic operator quality—combined with the lack of principled models (e.g. cardinality estimators, histograms, etc.) for estimating operator quality, cost, and latency—makes cost-based optimization challenging.

To overcome this, Abacus applies three key ideas. First, Abacus models the search for useful operators as an infinite-armed bandit problem [1, 39] and uses sampling to estimate physical operator performance. Given the cost of invoking LLMs, Abacus must be judicious in choosing which physical operators to sample and how many samples to spend on each operator. This is especially difficult in constrained optimization settings, where Abacus must discover the Pareto frontier of physical operators as opposed to a single objective maximizing operator. To this end, Abacus modifies an upper-confidence bound (UCB) bandit algorithm to enable it to search for the Pareto frontier of physical operators. The multi-armed bandit (MAB) algorithm can also leverage prior beliefs about operator performance to significantly accelerate its search.

Second, similar to relational query optimization, the space of physical plans grows combinatorially with the number of operators in the system. However, while relational query optimizers can use precomputed statistics to estimate the performance of plans at scale, Abacus's sample-based approach quickly becomes too expensive. To mitigate this issue, Abacus approximates plan performance as a function of its individual operators' performance. This decomposition allows Abacus to estimate the performance of a combinatorially large space of plans given a much smaller set of operator estimates. Third, the traditional dynamic programming algorithm used in Cascades [8] is not designed to support constrained optimization problems. To overcome this, Abacus implements a new Pareto-Cascades algorithm which keeps track of the Pareto frontier of subplans throughout the optimization procedure.

Performance. We have implemented Abacus as an optimizer in Palimpzest and evaluate its ability to optimize systems for document processing workloads in the biomedical and legal domains (BioDEX; CUAD) and multi-modal question answering (MMQA). Our results show that Abacus identifies plans with 20.8%, 39.4%, and 6.7% better quality, respectively, than similar plans optimized by DocETL and LOTUS. Furthermore, plans optimized by Abacus are on-average

10.8x cheaper and 3.4x faster than plans optimized by the next best system (in terms of quality).

We also show that, at a fixed sample budget, Abacus can use prior beliefs (i.e., a relative ranking of operators' quality, cost, and latency) to optimize plans to have up to 3.04x better quality than without priors. We further demonstrate that Abacus satisfies constraints in a non-trivial manner and improves system performance as constraints are relaxed. Finally, we perform an ablation study to isolate the benefits of prior beliefs, Pareto-Cascades, and the MAB algorithms and show that each helps improve Abacus's performance on two constrained optimization queries.

In summary, we present Abacus — a cost-based optimizer for semantic operator systems. Our main contributions are:

- An extensible, cost-based optimizer which allows for new semantic operators and optimization rules without changes to its host programming framework (Section 2).
- The implementation of algorithms which enable (1) efficient search over the space of semantic operator systems and (2) constrained optimization of these systems (Section 3).
- Quality improvements of up to 39.4% over competing state-of-the-art systems with cost and runtime savings of 10.8x and 3.4x relative to the next best system (Section 4).
- An investigation of Abacus's algorithmic contributions which shows that prior beliefs, Pareto-Cascades, and MAB sampling can improve optimization outcomes (Section 4).

2 SYSTEM OVERVIEW

In this section we present an overview of Abacus. First, we provide a brief background on semantic operators and the programming frameworks which optimize them. Then, we describe the end-to-end process by which Abacus optimizes semantic operator systems. Finally, we motivate the need for two key algorithms to make Abacus's optimization tractable, which we discuss in Section 3.

2.1 Background: Semantic Operator Systems

Background and terminology. Recent work has explored the use of *semantic operators* to implement data processing pipelines over unstructured data. Semantic operators are a set of AI-powered data transformations which mirror and extend relational operators [11]. Unlike their relational counterparts, semantic operators are specified in natural language as opposed to a SQL expression or a user-defined function. As a result, these operators' physical implementations typically require the use of one or more foundation models with semantic understanding.

Semantic programming frameworks like Palimpzest [23], LOTUS [28], DocETL [33], and Aryn [2] enable users to compose semantic operators into pipelines or directed acyclic graphs (DAGs). We refer to these computation graphs of semantic operators as *semantic operator systems*. Each framework implements an evolving and growing set of semantic operators, thus we highlight the operators currently supported by Abacus in Table 1.

Each semantic operator corresponds to a **logical operator** which may be implemented by a variety of **physical operators**. For example, two of the semantic map operators in Figure 1 are implemented with a Mixture-of-Agents [38] architecture and a Reduced-Context generation (Section 4.1). The former is a layered computation graph of LLM ensembles, while the latter reduces the context to contain only the most relevant portions of the input before feeding it into an LLM. Each of these physical operators has multiple hyperparameters (e.g. the models and temperature settings for Mixture-of-Agents; the model, chunk size, and number of chunks for Reduced-Context) leading to a large space of physical operators.

Table 1: Semantic operators supported by Abacus. In our 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	$\mu(d) \rightarrow d' \cup [d', d'', \dots]$
Filter	σ	$\sigma(d, P) \rightarrow d \cup \emptyset$
Join	$\bowtie$	$\bowtie (d, d') \rightarrow d'' \cup \emptyset$
Top-K	ρ	$\rho(d, V) \rightarrow d'$
Project	π	$\pi(d) \rightarrow d' \subseteq d$
Aggregate	α	$\alpha([d', d'', \dots]) \rightarrow \mathbb{R} \cup d$
Limit	λ	$\lambda([d', d'', \dots], L) \rightarrow [d', \dots, d^L]$

2.2 Abacus Optimizer

We illustrate Abacus's end-to-end process for optimizing semantic operator systems, beginning with a high-level overview of its key steps which are shown in Figure 2.

Inputs and Compilation. Abacus requires three inputs: an AI program, an optimization objective, and an input dataset. The AI program must be a pipeline or DAG of semantic operators supported by Abacus. 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, etc. which the physical implementation of the AI program will process. Optionally, users may guide the optimization process by providing any combination of a small validation dataset, prior beliefs about operator performance, and/or an LLM to use as a judge. A typical validation dataset contains 5-10 inputs with (possibly partial) labels. Prior beliefs are a simple dictionary mapping operators to a three-tuple of their perceived quality, cost, and latency on a [0,1] scale. Finally, Abacus can use an LLM as a judge to evaluate the quality of physical operators' outputs when labels are not present.

For example, in Figure 1, the AI program consists of a semantic filter followed by two semantic maps. The figure shows two objectives: maximizing quality and maximizing quality subject to a constraint on cost. The input dataset is a set of research papers, and the validation dataset (not shown) could be a handful of additional research papers whose relevance has been labeled. Finally, given these inputs, Abacus compiles the program into a logical plan where each semantic operator corresponds to a logical operator.

Creation of Search Space. Once the user's program has been compiled to a logical plan, Abacus uses its rules to enumerate a space of valid physical operators for each logical operator. This

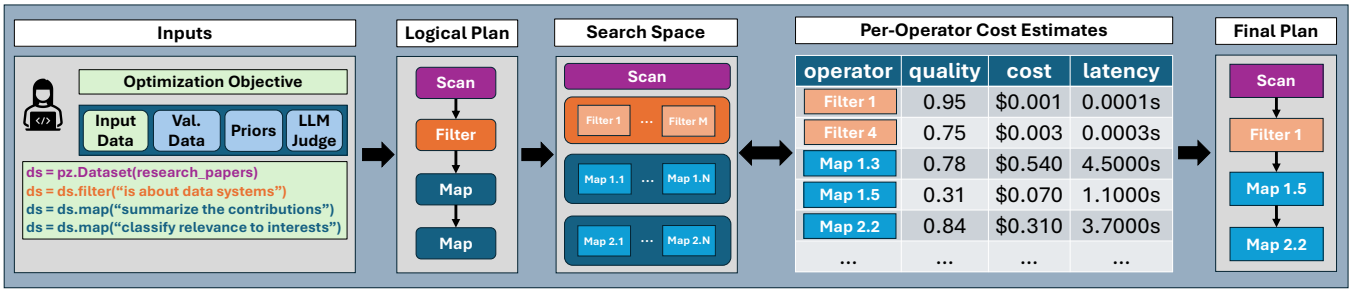

Figure 2: Overview of Abacus. The developer provides an AI program, optimization objective, input data, and (optionally) validation data, priors, and/or an LLM judge. Abacus (1) compiles the program to an initial logical plan, (2) applies rules to enumerate a search space of physical plans, (3) builds a cost model by processing validation inputs with sampled physical operators to measure their performance, and (4) returns the Pareto-optimal plan based on its estimates and the user objective.

corresponds to the Search Space in Figure 2. Each rule consists of two parts: (1) a pattern matching function which defines the logical subplan the rule can be applied to, and (2) a substitution function which applies the rule.

Transformation rules produce new functionally equivalent logical subplans. For example, a transformation rule may swap a filter and a map operation such that the filter is executed before the map. *Implementation rules* define ways to implement semantic operator(s) in a logical plan. For example, an implementation rule may implement a map operator with a Mixture-of-Agents or a Reduced-Context generation as depicted in Figure 1. Currently, Abacus only applies implementation rules when generating the search space, but in the future it may apply transformation rules as well. Abacus will apply transformation rules during the Final Plan Selection (see below) to ensure the final physical plan makes use of optimizations like filter pushdown and join re-ordering.

Operator Sampling. Given the search space of physical operators, Abacus seeks to identify ones which can be composed into physical plans that optimize the user's objective. For unconstrained optimization (e.g. maximizing plan quality), this implies finding high-quality physical operator(s). For optimization with constraints (e.g. maximizing plan quality subject to a cost constraint), this suggests finding physical operators which lie on the Pareto frontier of the cost vs. quality trade-off.

Abacus initially samples a small batch of physical operators for each logical operator. If Abacus has access to prior beliefs about operator performance, it samples operators which it believes lie closest to the Pareto frontier of the optimization objective first. Otherwise, it samples operators at random. Given these operators, Abacus executes them on inputs sampled from the validation dataset (or the input dataset if no validation data is present). Abacus measures the quality, cost, and latency of each operator on each input. To measure quality, Abacus uses output label(s) from the validation dataset when they are available. However, if no label exists—either because validation data is not provided or it does not contain some intermediate label(s)—then Abacus evaluates each operator's output with an LLM judge. By default, Abacus will use `o4-mini` as the LLM judge, but the user may specify another model.

Once it has estimated each operator's quality, cost, and latency, Abacus computes the Pareto frontier of physical operators (with respect to the optimization objective) for each logical operator. Physical operators which fall too far from the frontier are removed, and new operators are sampled to replace them. The next batch of inputs is then processed with the new operator frontiers, and the process repeats until the sample budget (measured in dollars or operator invocations) has been reached. The cost overhead of the sampling algorithm is bounded above by the sample budget, and the latency overhead only scales with the depth of the logical plan.

Final Plan Selection. Once the sample budget is exhausted, Abacus needs to construct a final plan to process the input dataset. First, it computes each physical operator's average quality, cost, and latency on sampled inputs. Abacus then passes these estimates and the user's optimization objective to its Pareto-Cascades algorithm (Section 3.2) to compute the optimal plan. The algorithm applies both transformation and implementation rules to search the full space of logical and physical plans, using Abacus's cost model (subsection 2.3) to assess their quality, cost, and latency.

Full Algorithm. The full algorithm for Abacus is shown in Algorithm 1. The user program is compiled into an initial logical plan on line 1. On line 2, Abacus applies rules to create a search space of physical operators. Line 3 initializes a cost model which keeps track of each operator's average quality, cost, and latency. We describe the cost model in more detail in Section 2.3. On line 4, Abacus samples an initial "frontier" of k physical operators for each logical operator. On line 7, each frontier processes a sample of j inputs, updating the number of samples drawn. This also yields a set of observations of operator quality, cost, and latency, which are used to update the cost model on line 8. On line 9, operators which perform poorly are replaced in each frontier. We discuss the algorithm for updating the operator frontiers in detail in Section 3.3. Once the number of samples drawn (or the dollar cost of sampling) exceeds the sample budget on line 6, the operator sampling stops. Finally, on line 10 Abacus's Pareto-Cascades algorithm returns the optimal physical plan with respect to the the operator estimates and the optimization objective. We discuss the Pareto-Cascades algorithm in detail in Section 3.2.

2.3 Key Challenges in Optimization

We now motivate the design of Abacus's cost model, operator sampling algorithm, and final plan selection algorithm.

Algorithm 1 Abacus algorithm

Require: program P, objective O, val. data D
Parameters: budget B; k, j

```
logical_plan = compile(P)
search_space = applyRules(logical_plan)
M = initCostModel()
F = sampleOpFrontiers(search_space, k)
samples_drawn = 0
while samples_drawn < B do
    outputs, samples_drawn = processSamples(F, D, j)
    M = updateCostModel(M, outputs)
    F = updateFrontiers(F, M, O)
return ParetoCascades(logical_plan, M, O)
```

Cost Model. Given a logical plan with M semantic operators and a choice of N physical implementations per operator, the space of possible physical plans is of size $O(N^M)$ before considering operator re-orderings. Even for relatively modest values of M and N, the space of plans quickly grows too large to sample each plan and measure its output quality, cost, and latency.

To address this, Abacus makes the simplifying assumption that operators are independent, and that each plan can be modeled as a function of its operators. Abacus's model for the plan quality (p_q), cost (p_c), and latency (p_l) as a function of its operators' quality (o_{qi}), cost (o_{ci}), and latency (o_{li}) is shown below:

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

One limitation of this cost model is that it fails to model interactions between operators. For example, if a semantic filter uses the summary produced by an upstream map operator as input, then this cost model will fail to capture that the filter's performance is correlated with the quality of the map operator's summary. We evaluate Abacus on some queries with this property in Section 4.3 and discuss the limitations of this cost model in Section 5.

Operator Sampling Challenges. For large enough N, it can be computationally infeasible to sample every physical operator for even a single semantic operator. For example, in our implementation of Abacus (Section 4.1), a semantic map can be implemented with approximately 2,800 different operators. While the task of finding and choosing physical operator(s) may seem daunting, in most settings Abacus simply needs to produce a plan which is "good enough" for the user's application goals. This relaxes the operator search problem from finding the single best "needle in a haystack" to finding at least one operator from a handful of good options.

In the *sampleOpFrontiers*() function in Algorithm 1, we sample an initial set (i.e., frontier) of physical operators for each logical operator in the plan. Then, in the *updateFrontiers*() function we update each frontier of physical operators based on their observed quality, cost, and latency on sampled inputs. We model the sampling of physical operators as a multi-armed bandit (MAB) problem. Intuitively, given a fixed sampling budget, we seek to navigate the exploration-exploitation trade-off between sampling new (potentially better) operators and sampling the previously best observed operator(s) to refine our confidence in their performance.

Algorithm 2 Full Cascades algorithm

Require: logical plan P, cost model M, rules R

```
procedure Cascades(P, M, R)
    G = createInitialGroups(P)
    G = searchPlanSpace(G, M, R)
    final_group_id = getFinalGroupId(G)
    return getMinCostPlan(final_group_id, G)
procedure getMinCostPlan(group_id, G)
    best_expr = G[group_id].best_expr
    input_group_id = best_expr.input_group_id
    if input_group_id is None then
        return Plan(best_expr.operator)
    best_subplan = getMinCostPlan(input_group_id, G)
    return Plan(best_subplan, best_expr.operator)
```

Algorithm 3 Cascades Subroutine: Plan Search

Require: groups G, cost model M, rules R

```
procedure searchPlanSpace(G, M, R)
    task_stack = [OptimizeGroup(getFinalGroupId(G))]
    while len(task_stack) > 0 do
        task = task_stack.pop()
        new_tasks = task.perform(G, M, R)
        for new_task in new_tasks do
            task_stack.push(new_task)
    return G
```

Unfortunately, the traditional MAB formulation is focused on finding the single most-optimal arm for an unconstrained objective. However, constrained optimization requires that we account for the trade-off between the optimization objective and the constraint(s). To this end, we modify the traditional MAB formulation to encourage the exploration-exploitation trade-off of *the entire Pareto frontier of operators*. We formalize this algorithm in Section 3.3.

Final Plan Selection Challenges. Once Abacus finishes sampling operators, it still needs to identify and return the optimal physical plan. For an unconstrained objective such as minimizing plan cost, Abacus can invoke a traditional Cascades [8] algorithm to recover the minimum cost plan.

However, for constrained optimization the traditional Cascades algorithm is insufficient. The key issue is that Cascades will only keep track of the "best" implementation of every subplan. However, in the constrained setting—where we care about multiple dimensions of plan performance—finding the optimal plan requires considering the Pareto frontier of optimization trade-offs at each subplan. We implement the Pareto-Cascades algorithm to overcome this challenge, and discuss its implementation in Section 3.2.

3 ALGORITHMS

In this section, we first present a high-level overview of the traditional Cascades algorithm from relational query optimization. We then discuss the Pareto-Cascades and multi-armed bandit (MAB) operator sampling algorithms we developed for Abacus.

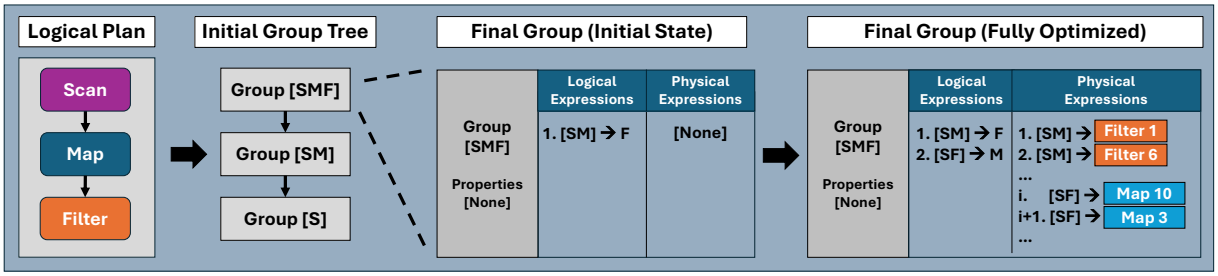

Figure 3: Example of the Cascades algorithm applied to a simple logical plan; it first constructs an initial group tree with one logical expression per group and then applies a task to optimize the final group (SMF), which uses dynamic programming to search plans via repeated application of tasks. After all possible tasks have been applied (or a limit on the total number of tasks has been reached), it recursively constructs the optimal physical plan by selecting the best physical expression at each group.

3.1 Traditional Cascades Optimization

Cascades [8, 41] takes a logical plan, a cost model, and a set of rules as input. Given these inputs, Cascades seeks to find an implementation of each operator that globally optimizes the plan to meet some objective (e.g. minimizing execution cost).

Consider the toy example in Figure 3. First, Cascades converts the logical plan into an **initial group tree** (Algorithm 2, Line 2). Each **group** represents the execution of a unique set of operators. In Figure 3, we expand the **final group** which represents the execution of all operators in the plan. Each group has a set of **logical** and **physical expressions**, which represent unique logical and physical subplans which implement that group.

Initially, each group has a single logical expression which is translated directly from the logical plan (in this case, executing filter F after map M and scan S). Given the initial group tree, the Cascades algorithm searches the space of possible physical plans (Algorithm 2, Line 3) by applying a series of **tasks** in a dynamic programming algorithm shown in Algorithm 3. There are four main tasks: Optimize Group, Optimize Logical Expression, Apply Rule, and Optimize Physical Expression. The call to optimize a group triggers tasks for optimizing each of its logical and physical expressions. Logical expressions are optimized by applying transformation rules (to generate new equivalent logical expressions) and implementation rules (to generate physical expressions). Finally, the task to optimize a physical expression computes the minimum cost plan for executing the given expression.

We present the full Cascades algorithm in Algorithm 2. The algorithm takes a logical plan, a cost model, and a set of rules as input. It constructs the initial groups (i.e, the group tree) on Line 2 and then invokes the plan search procedure in Algorithm 3 on Line 3. Once the search finishes, the group tree is traversed to construct the final physical plan by using the physical operator in the optimal (i.e. min. cost) physical expression for each group (Line 5). With this understanding in place, we will now discuss Abacus's Pareto-Cascades algorithm.

3.2 Pareto-Cascades Optimization

As discussed in Section 2.3, Cascades is not designed to handle optimization problems with constraints. The key issue is the **Principle of Optimality**, which states that *every subplan of an optimal physical plan is itself optimal.* This principle enables Cascades to optimize each physical expression by composing it with the optimal expression for its input group. This is insufficient for problems such as minimizing cost with a lower bound on plan quality, because selecting the minimum cost expression for each group may result in constructing a plan that fails to meet the quality constraint.

In order to address this issue, each dimension of the optimization problem (e.g., cost and quality) must be accounted for. Fortunately, there is a natural way to extend the Principle of Optimality into the constrained optimization setting, which we present as a theorem:

Theorem 3.1. *(Under the operator independence assumptions of our cost model in Section 2.3) every subplan of a Pareto-optimal physical plan is itself Pareto-optimal.*

Proof. We prove this by contradiction. Assume a Pareto-optimal physical plan P has a subplan S which is not Pareto-optimal. By the definition of S not being Pareto-optimal, there exists a subplan S' which dominates S. Replacing S with S' strictly improves the quality, cost, and/or latency of the subplan. Given the operator independence assumptions of our cost model in Equation (1), strictly improving the subplan will also strictly improve the quality, cost, and/or latency of the entire physical plan. This new physical plan P' will be strictly better than our original plan P – but this contradicts our assumption that the original plan P is Pareto-optimal. □

This theorem enables us to extend the Cascades algorithm to the constrained optimization setting by modifying each group to maintain its Pareto frontier of physical expressions during the search procedure in Algorithm 3. For example, if a user's objective is to maximize plan quality with an upper bound on plan cost, then each group maintains its set of physical expressions which are Pareto-optimal with respect to quality and cost. The task to optimize physical expressions is modified to compute the Pareto frontier of executing the current physical expression with any of the Pareto-optimal expressions from its input group(s). Finally, once the search procedure is finished, the Pareto-optimal plan is recovered by recursively composing all Pareto-optimal subplans before selecting the final plan which is optimal for the given optimization objective. (While the Pareto frontier introduces a branching factor to the plan search space, it is bounded above by the number of physical operators sampled for a given semantic operator).

Algorithm 4 Pareto-Cascades algorithm

Require: logical plan P, cost model M, rules R, objective O

```
1: procedure ParetoCascades(P, M, R, O)
2:     G = createInitialGroups(P)
3:     G = searchPlanSpace(G, M, R)
4:     final_group_id = getFinalGroupId(G)
5:     pareto_plans = getParetoOptPlans(final_group_id, G)
6:     return selectOptimalPlan(pareto_plans, O)
```

We present the algorithm for our new Pareto-Cascades algorithm in Algorithm 4. We use the same function for searching the plan space with the modifications described in the previous paragraph. The *getParetoOptPlans*() function is similar in spirit to *getMinCostPlan*() in Algorithm 2, except it builds and returns a list of Pareto-optimal plans. The *selectOptimalPlan*() function picks the plan on the Pareto frontier which is optimal for the optimization objective O (e.g. selecting the max quality plan which is cheaper than a cost upper bound). If the Pareto-Cascades algorithm cannot find a plan which satisfies the given constraint, then the algorithm will return the plan which best optimizes the given objective. Finally, we note that in the case of unconstrained optimization, this algorithm naturally reduces to the traditional Cascades algorithm.

3.3 Multi-Armed Bandit Operator Sampling

The second key optimization challenge in Abacus is choosing which physical operators to sample in order to obtain estimates of operator quality, cost, and latency. As discussed in Section 2.3, we assume that the number of physical operators N is large enough that Abacus cannot realistically sample every physical operator. To overcome this issue, we draw inspiration from the infinite-armed bandit problem [1, 39], which can also serve as a model for settings with more arms than total samples.

In our setting, the physical operators comprise the "arms" of our search space and we are given an initial sample budget B. At each step of the search, we must choose a physical operator to sample (decreasing our budget by one or by the cost of invoking the operator if B is in dollars) and obtain a stochastic observation of that operator's performance. In contrast to the traditional multi-armed bandit (MAB) setting, where the objective is to identify the single best arm achieving the highest performance in expectation, Abacus's goal is to identify the potentially many physical operators which lie on the Pareto frontier of its optimization objective.

We present Abacus's MAB operator sampling algorithm in Algorithm 5. The inputs to the algorithm are an initial set of physical operator frontiers F (one for each logical operator, from line 4 of Algorithm 1), the cost model M, and the optimization objective O. The algorithm begins by computing the upper confidence bounds (UCBs), lower confidence bounds (LCBs), and means for each operator on each metric of interest for the objective O. The equations for computing the UCB and LCB of a given metric are shown below:

$$ucb_{m,i} = \mu_{m,i} + \alpha \cdot \sqrt{\frac{\log(N)}{n_i}} \quad lcb_{m,i} = \mu_{m,i} - \alpha \cdot \sqrt{\frac{\log(N)}{n_i}}$$

The $\mu_{m,i}$ term is the sample mean of the observed performance for the given metric m (e.g. operator latency) for the i^{th} physical operator. N is the total number of samples drawn and n_i is the number of samples drawn for the i^{th} physical operator. Finally, $\alpha \in [0, 1]$ is the exploration coefficient, which we dynamically scale to be 0.5 times the spread between the largest and smallest observed metric values across all physical operators.

Algorithm 5 MAB operator sampling algorithm

Require: initial operator frontiers F, cost model M, objective O

```
1: procedure updateFrontiers(F, M, O)
2:     for op_frontier in F do
3:         op_UCBs = computeUCBs(op_frontier, M, O)
4:         op_LCBs = computeLCBs(op_frontier, M, O)
5:         op_means = computeMeans(op_frontier, M, O)
6:         pareto_ops = computeParetoOps(op_means, O)
7:         num_new_ops = 0
8:         for op in op_frontier do
9:             ucbs = op_UCBs[op]
10:            pareto_lcbs = op_LCBs[pareto_ops]
11:            if no_overlap(ucbs, pareto_lcbs) then
12:                op_frontier.pop(op)
13:                num_new_ops += 1
14:        new_ops = sampleReservoir(num_new_ops)
15:        op_frontier.add(new_ops)
```

Once the UCBs, LCBs, and means are computed for every operator and metric, we compute the set of Pareto-optimal operators based on their mean performance. Then, for each operator in the frontier, we check whether its upper confidence bound overlaps with the lower confidence bound of at least one operator on the Pareto frontier. Such an overlap implies that there's enough uncertainty in our estimates of operator performance that it is possible for the operator to lie on the Pareto frontier. If no overlap exists, then we remove the operator from the frontier and sample a replacement from our reservoir of not yet sampled physical operators. This completes the update of the operator frontier.

The key difference between this algorithm and a traditional UCB algorithm for MABs is that we must consider overlap between each operator and the current Pareto frontier of sampled operators. Overlap on any dimension implies that the operator may still be Pareto-optimal, thus eliminating operators from consideration can be slightly more sample intensive and time consuming. In order to speed up the algorithm, we construct batches of samples rather than processing one sample at a time.

Finally, one benefit of this problem formulation is that it allows for a number of extensions which can accelerate Abacus's search for Pareto-optimal operators. For example, if there exist prior beliefs about operator performance, Abacus can use them to inform its initial operator frontier as well as the next operator(s) it draws from the reservoir during replacement. We explore the benefit of prior beliefs on operator performance in section Section 4.4.

4 EVALUATION

We evaluate Abacus on a diverse set of benchmarks to examine three experimental claims. First, we demonstrate that semantic operator systems optimized by Abacus outperform similar systems optimized by prior work. Second, we show Abacus leverages

prior beliefs to identify better plans with fewer samples. Third, we demonstrate Abacus improves system performance as constraints are relaxed. Finally, we perform an ablation study to isolate the effects of our key algorithmic contributions.

4.1 Implementation

We implement Abacus as an optimizer inside of the open-source Palimpzest [23] framework, which supports all of the semantic operators in Table 1. We wrote standard implementation rules for each semantic operator to provide Abacus with the ability to implement any Palimpzest program. We also implemented the following rules for optimizing map, filter, join, and top-k operators.

Map and Filter. We wrote four implementation rules which can be applied to map and filter operations. The **Model Selection** rule implements the operator with a single LLM call and is parameterized by the set of models supported by Palimpzest. The **Mixture-of-Agents** rule implements a Mixture-of-Agents architecture [38] consisting of an ensemble of proposer models followed by an aggregator model. The rule is parameterized by (1) the size of the ensemble (1-3 proposers), (2) the model used for each proposer, (3) the model used for the aggregator, and (4) the temperature of the proposers (0.0, 0.4, or 0.8). The **Reduced-Context Generation** rule chunks the input, computes an embedding for each chunk, and then concatenates the top-k embeddings (based on similarity with the map/filter instruction) and feeds them into the operator. The rule is parameterized by the chunk size (1000, 2000, or 4000 characters) and k (1, 2, or 4). Finally, the **Critique-and-Refine** rule uses an LLM to generate an initial output, which is then critiqued by a second model, before a third and final model generates a refined output. The rule is parameterized by the model used for each step.

Top-K and Join. We wrote a single rule to implement a top-k operator. The rule is parameterized by the value k which determines the number of objects returned by the operator. We wrote two rules to implement a semantic join. The **Nested Loops Join** rule implements the join by evaluating the join condition with an LLM for every join tuple. The **Embedding Join** rule implements the join by automatically dropping tuples with low embedding similarity and automatically joining tuples with high embedding similarity (tuples in-between a high and low threshold are still processed with an LLM). The Nested Loops Join is generally expensive and accurate, while the Embedding Join rule is cheaper but less accurate.

These rules give Abacus $\sim 3,000$ physical operators when configured with access to all supported LLMs. For our experiments, unless stated otherwise, we provided Abacus with access to GPT-4o, GPT-4o-mini, Llama-3.1-8B, Llama-3.3-70B, Mixtral-8x7B, and DeepSeek-R1-Distill-Qwen-1.5B.

4.2 Benchmarks and Implementations.

Benchmarks. We evaluate Abacus on three benchmarks for processing unstructured documents. Each input in the **BioDEX** benchmark [6] is a document describing adverse reaction(s) experienced by a patient in response to taking a drug. In line with prior work [5, 28, 33], we focus on the task of producing a ranked list of the adverse reactions experienced by the patient. Success on this task is measured by the rank-precision (RP) of the output rankings at a specified threshold K (i.e. RP@K). Each input in the **CUAD** [12] benchmark is a legal contract. Given a set of 41 contract clauses, the task is to predict the span(s) in the contract which correspond to each clause (the ground truth for a single clause spans $\sim 0.25\%$ of the document on average). Success on this task is measured by the F1-score of the clause predictions. Finally, The **MMQA** dataset [35] contains questions involving reasoning over images, text, and/or tables. Success is measured by F1-score on question answers, as the ground truth label for every question is a list of outputs.

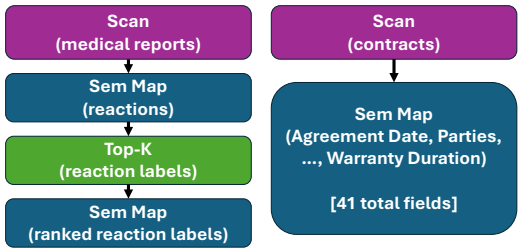

Figure 4: Query plans for BioDEX (left) and CUAD (right).

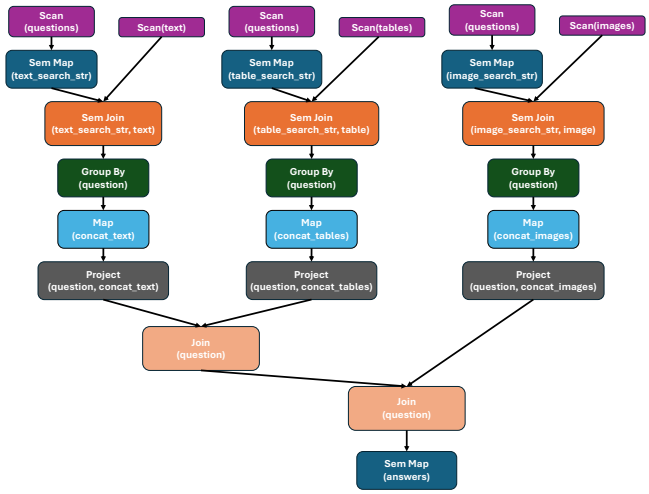

Figure 5: The query plan for the MMQA benchmark.

Implementations. For BioDEX, we use code from the authors of DocETL and LOTUS to evaluate their systems. Both LOTUS and DocETL compute a semantic join between each input medical document and the list of reaction labels before using a semantic map to rerank the labels. For Abacus, we implement a pipeline in Palimpzest which joins input medical documents to the most similar reaction labels using a semantic map and semantic top-k operator, before reranking the labels using a semantic map (Figure 4, left). For CUAD, each framework computes the 41 output clauses with a semantic map (or semantic extract). The code for DocETL was provided by the authors, while the code for LOTUS and Abacus were implemented using a single operator (Figure 4, right).

Finally, for MMQA we implement a simple baseline which asks `GPT-4o-mini` to answer each question without any relevant image, text, or table content. This represents a lower bound on expected performance. DocETL does not support image inputs so we omit it from our evaluation. For LOTUS and Abacus, we implemented

Table 2: Performance on the BioDEX, CUAD, and MMQA benchmarks for systems optimized to maximize quality. Quality is measured using RP@K for BioDEX and F1 score for CUAD and MMQA. Mean values are shown with their standard deviation.

	System	Quality	Cost ($) Opt.	Cost ($) Exec.	Cost ($) Total	Time (s) Opt.	Time (s) Exec.	Time (s) Total
BioDEX	DocETL	0.193 ± 0.032	$3.50 ± 3.04	$3.04 ± 2.51	$6.54 ± 5.53	427 ± 130	1,008 ± 249	1,435 ± 238
	LOTUS	0.216 ± 0.042	–	–	$18.9 ± 12.8	–	–	2,348 ± 1,489
	Abacus	**0.261 ± 0.026**	$0.18 ± 0.02	$0.70 ± 0.12	$0.89 ± 0.11	303 ± 48	147 ± 22	450 ± 47
CUAD	DocETL	0.475 ± 0.106	$6.04 ± 2.52	$1.01 ± 0.330	$7.05 ± 2.63	1,540 ± 511	280 ± 128	1,820 ± 594
	LOTUS	0.234 ± 0.005	–	–	$0.20 ± 0.02	–	–	125 ± 19
	Abacus	**0.662 ± 0.010**	$0.19 ± 0.05	$0.51 ± 0.01	$0.69 ± 0.05	318 ± 61	132 ± 13	450 ± 67
MMQA	GPT-4o-mini	0.160 ± 0.33	–	–	$< \$3 \cdot 10^{-3}$	–	–	78.0 ± 4.9
	LOTUS	0.284 ± 0.046	–	–	$14.3 ± 5.8	–	–	1,208 ± 347
	Abacus	**0.304 ± 0.079**	$0.17 ± 0.01	$12.9 ± 10.6	$13.1 ± 10.6	598 ± 152	550 ± 299	1,149 ± 300

Table 3: Abacus' performance on BioDEX, CUAD, and MMQA when optimizing to minimize cost (top) and latency (bottom). Reduction measures how much cheaper / faster the optimized plan is relative to the max quality equivalent.

(MinCost)	Quality	Exec. Cost ($)	Reduction
BioDEX	0.21 ± 0.02	$0.28 ± 0.10	2.50x
CUAD	0.05 ± 0.02	$0.12 ± 0.01	4.25x
MMQA	0.31 ± 0.05	$16.0 ± 9.7	0.81x

(MinTime)	Quality	Exec. Time (s)	Reduction
BioDEX	0.21 ± 0.03	128 ± 50	1.15x
CUAD	0.10 ± 0.05	55 ± 18	2.4x
MMQA	0.28 ± 0.07	540 ± 382	1.02x

a complex query plan (Figure 5) that uses three semantic maps to generate search strings for querying relevant images, text, and tables. Next, each data modality is semantically joined to the questions based on their relevance to the search string. The retrieved data is then manipulated with relational group by, map, project, and join operations to obtain a dataset with one row per question, where each row contains a list of the joined images, text, and tables. Finally, the question is answered using a semantic map. (To keep the computation tractable, we limited the datasets to only include data items related to at least one question in the ground truth.)

4.3 Abacus Outperforms Prior Work

To evaluate our first experimental claim, we compare Abacus to DocETL [33] and LOTUS [28]. In order to maintain parity with their evaluations, we restrict each system to using `GPT-4o-mini`, `text-embedding-3-small`, and `clip-ViT-B-32`.

Setup. We executed each system 10 times on the BioDEX, CUAD, and MMQA benchmarks with the objective of maximizing output quality. For each of the 10 trials, we sampled different examples from the benchmarks' test datasets. These "splits" (each containing 250 samples for BioDEX and 100 samples for CUAD and MMQA) were drawn to make the evaluation computationally tractable while still accounting for diversity in test examples. We ran each system on each split and measured the output quality, execution cost in dollars, and latency in seconds. We report the mean and standard deviation of these measurements. For DocETL and Abacus, which have distinct optimization and execution stages, we also break out the cost of optimization and the cost to execute the final optimized plan. Finally, for Abacus we used the default values of $k = 6$ and $j = 4$, while setting the sample budget equal to 50× the number of semantic operators in the query plan. This heuristic ensures that roughly half of the sample budget ($50 - 6 \cdot 4 = 26$) is used for exploration by the MAB algorithm.

Results. The results of our evaluation are shown in Table 2. Overall, Abacus is able to maximize quality better than all competing systems. On BioDEX, CUAD, and MMQA, Abacus achieves 20.3%, 18.7%, and 39.2% better mean quality than the next best system, respectively. Furthermore, Abacus's plans are on-average 12.6x cheaper and 2.7x faster than the next best system (in terms of quality). The key drivers of Abacus's performance improvements vary across benchmarks. However, a common theme is that Abacus succeeds when it (1) has access to implementation rules (i.e. optimizations) that are cost-effective for the given task and (2) is able to obtain good estimates of these optimizations' performance, which help the MAB algorithm search for high-quality optimizations.

For example, on the BioDEX benchmark, the Reduced-Context Generation rule works well for extracting reactions from the medical reports and re-ranking the final reaction labels (the first and second map in Figure 4, respectively). This is because the optimization discards text from the medical report which is not directly related to the patient's adverse reaction(s), thus saving money on token processing while also helping the LLM focus on the most relevant data when performing the map. LOTUS optimizes semantic joins by sampling join tuples in order to learn thresholds for a cascade. However, the quality of the cascade is subject to variance and depends on how well the sampled join tuples represent the overall join. In the worst case, LOTUS produces joins which require $> 100,000$ LLM calls, leading to high runtime and cost. For DocETL, we inspect the final pipelines it generates with GPT-4o and find that it often implements the query by (1) using a map to extract reactions from the underlying document, (2) joining these reactions to the reaction labels, and (3) reranking reaction labels based on

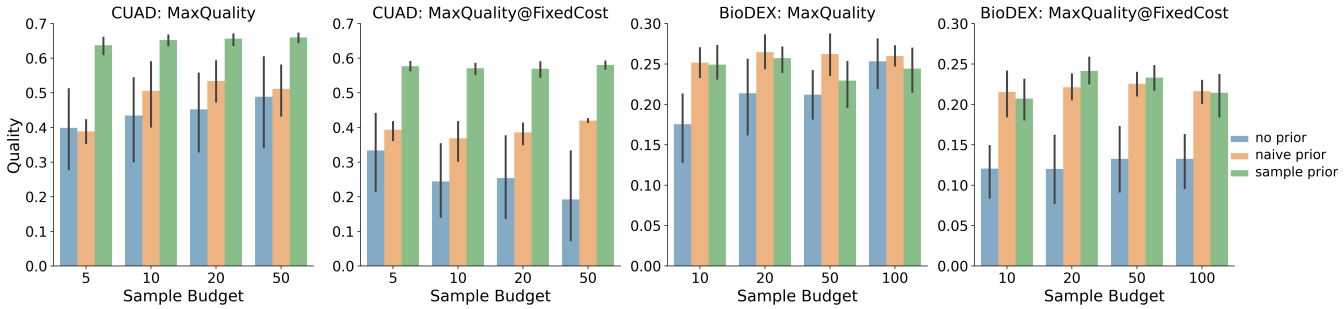

Figure 6: System output quality as a function of the sample budget when optimizing with (1) no priors, (2) naive priors computed from MMLU-Pro performance, and (3) priors computed with samples from each benchmark's train split. Overall, Abacus yields better plans in the constrained and unconstrained settings when leveraging prior beliefs on operator performance.

document relevance. DocETL's LLM-based optimizer implements the join using a combination of heuristics (e.g. checking that all words in the reaction label appear in the document) and thresholding based on embedding similarity between the reaction label and the extracted reactions. Similar to LOTUS, imprecision in the choice of heuristic and similarity threshold can degrade performance.

On CUAD, Abacus implements the semantic map with a Mixture-of-Agents operator in all 10 trials. The Mixture-of-Agents operator does a good job of extracting legal clauses from the documents with high precision (avg. 87.2%) and decent recall (avg. 53.0%), likely because its proposer-aggregator architecture enables the aggregator LLM to synthesize a final response which only includes the proposed text spans which are correct with high probability. By comparison, DocETL's LLM optimizer spends anywhere from 20 - 40 minutes incurring high optimization cost as it decomposes the map operation into a multi-stage pipeline. In 10 trials, we observe that DocETL rewrites the map into a pipeline with anywhere from 2 to 7 operations, which ultimately leads to large variance in its performance. Interestingly, we find that DocETL's pipelines perform best (achieving up to 63.7% F1-score) when it composes a 3-step pipeline and perform much worse (as low as 35.3% F1-score) on its deeper 7-step pipelines. (LOTUS does not optimize map operators so its implementation is cheap and fast but achieves low quality).

On MMQA, the implementation of the three semantic joins largely determines each system's performance. In particular, the semantic join for relevant images is the primary driver of plan cost and latency. If the joins successfully retrieve the relevant images, text, and/or tables, the final map is significantly more likely to answer the question correctly. Abacus implements the semantic image join with the EmbeddingJoin optimization on 75% of trials. Similar to LOTUS, the optimization's performance is sensitive to the join tuples sampled during optimization. However, its conservative thresholding leads the EmbeddingJoin to (on average) invoke the LLM more often, which leads to higher average quality with slightly larger cost and latency.

In Table 3 we examine Abacus's ability to minimize the cost and latency of query plans on each benchmark. For BioDEX and CUAD, Abacus reduces the average cost / latency of the optimized plans by implementing each map with Reduced-Context Generation rules that process only a fraction of the document. On BioDEX, Abacus already uses instances of this rule when optimizing for quality so the cost and latency savings are smaller than for CUAD. On CUAD, these savings come with a larger trade-off in quality, because text related to the 41 legal clauses are more evenly dispersed through the contract. Finally, Abacus struggles to minimize latency and cost on MMQA because 75% of the quality maximizing plans already use the EmbeddingJoin optimization, leaving little room for Abacus to improve. Furthermore, because the EmbeddingJoin exhibits high variance in its quality, cost, and latency, when averaged over 10 trials we can achieve results where the minimum cost / latency plans achieve higher cost / latency than their quality maximizing counterparts. In the future–if given better, lower variance optimizations–we expect Abacus could improve its cost and latency minimizing results in Table 3.

4.4 Performance Improves with Better Priors

For our second experimental claim, we ran Abacus on CUAD and BioDEX with and without prior beliefs while also varying the sample budget. We omitted MMQA from our evaluation as the number of physical operators for semantic joins is small enough to sample exhaustively. We examined maximizing quality with and without a cost constraint. The cost constraints for CUAD and BioDEX were set equal to the 25th percentile of plan execution costs we observed in the unconstrained setting, thus making them non-trivial to satisfy. We aimed to show that Abacus could leverage prior beliefs to identify more optimal plans with fewer samples.

For each benchmark, we used two sets of prior belief(s). The first "naive" prior estimated each operator's quality as an average of its model(s') performance on the MMLU-Pro benchmark [40]. It also estimated the cost of each operator by averaging its per-token input and output costs. This prior is cheap to compute and can be done offline, but lacks fidelity in the accuracy of its estimates. The second "sample-based" prior estimated operator performance by running each operator on 5 samples from the train split of the respective dataset. This prior is more expensive to compute and must be done online, but has higher fidelity in its estimates.

The results of our evaluation are shown in Figure 6. Overall, we observe that Abacus produces plans with higher quality when provided with prior beliefs. In the unconstrained setting, plans optimized with prior beliefs perform up to 1.60x and 1.43x better

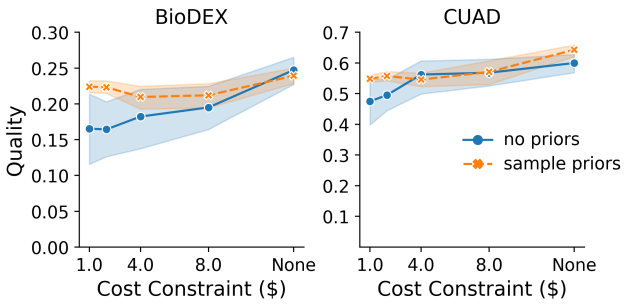

Figure 7: The performance of plans optimized for max quality subject to a cost constraint. Plan performance improves as constraints are relaxed. Prior beliefs help Abacus maintain better performance even with tight cost constraints.

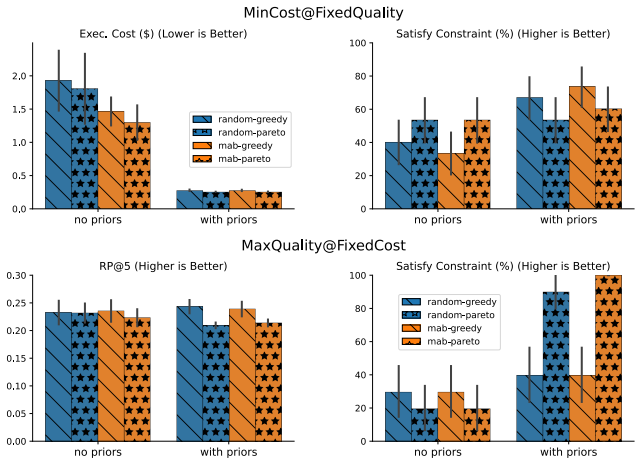

Figure 8: Ablation study isolating the effects of the Pareto-Cascades and MAB sampling algorithms and prior beliefs.

(at a fixed sample budget) than those optimized without priors on CUAD and BioDEX, respectively. This gap is even greater in the constrained optimization setting, where plans optimized with prior beliefs perform up to 3.02x and 2.01x better (at a fixed sample budget) than those optimized without priors on CUAD and BioDEX, respectively. This latter result comes from the fact that identifying a good Pareto frontier of operators is more difficult than identifying a single best operator, thus having a prior belief over the entire frontier provides greater benefit relative to sampling without priors.

Finally, due to (1) the higher average operator costs in this experiment and (2) the small sample budget sizes, average plan costs and latencies are higher than in Table 2. The average plan costs and latencies in the constrained and unconstrained settings, respectively, are ($4.42, 240s) and ($1.15, 211s) for BioDEX and ($6.20, 213s) and ($1.83, 214s) for CUAD. Still, 91.6% and 93.3% of the constrained BioDEX and CUAD plans satisfy the given constraints, suggesting the optimizer was willing to accept higher costs and runtimes in the pursuit of better plan quality.

4.5 Abacus Leverages Relaxed Constraints

For our third experimental claim, we used Abacus to optimize plans for BioDEX and CUAD with the objective of maximizing quality subject to a cost constraint. We varied the cost constraint from unconstrained optimization down to $1, which is 11.8% and 16.2% of the median cost of an unconstrained plan on BioDEX and CUAD, respectively. Tightening the cost constraint limits the space of systems available to Abacus. Thus, our goal was to demonstrate that Abacus responds to looser constraints by identifying more optimal plans, or vice-versa, that Abacus responds to tighter constraints with non-trivial system implementations.

For each cost constraint, we used Abacus to optimize plans for maximum quality with 10 different test splits of the BioDEX and CUAD datasets. We used the same sample budget at each cost constraint and optimized with and without prior beliefs. The results of our evaluation are shown in Figure 7. For optimization without prior beliefs, Abacus is generally able to identify plans which achieve better quality as the cost constraint is relaxed. Furthermore, Abacus still identifies plans which achieve non-trivial performance when optimizing under tight constraints.

When optimizing with priors, we see a smaller degradation in performance as the constraint is tightened. For example, without priors, performance on BioDEX decreases by 45.6% from having no constraint to a constraint of $1. However, with priors it only decreases by 12.5% at its lowest point with a constraint of $4. This is due to the fact that prior beliefs on operator performance help guide Abacus's MAB sampling algorithm to prioritize operators which lie on the entire Pareto frontier of the cost vs. quality trade-off.

4.6 Ablation Study

Finally, we ran an ablation study to isolate the benefits of Pareto-Cascades, our MAB sampling algorithm, and prior beliefs. We ran Abacus on the BioDEX benchmark with the default values of k and j and a sample budget of 150 samples. We executed two policies: minimizing cost with a lower bound on quality (Figure 8, top row) and maximizing quality with an upper bound on cost (Figure 8, bottom row). To make the optimization challenging, we set the quality and cost constraints equal to 80% and 50% of Abacus's mean quality and cost in Table 2, respectively.

When running Abacus without Pareto-Cascades, we replaced it with a modified traditional Cascades algorithm that greedily selects the optimal subplan for each group which does not violate the constraint. This algorithm represents a greedy scheme in which the plan is built myopically without considering how subsequent operators may need to satisfy the constraint. When running Abacus without the MAB sampling algorithm, we replaced it with a random sampling algorithm that sampled k physical operators and $j = B/k$ inputs. In theory, this algorithm should suffer from an inability to use some of its sample budget to search for new operators, which is a key benefit of the MAB approach.

The results of our ablation are shown in Figure 8. First, on-average, prior beliefs help Abacus optimize the objective 3.5x better and satisfy the constraint 1.8x more often. Second, we can see the benefits of the MAB algorithm in the cost minimization objective, where Abacus with the MAB algorithm minimizes cost by 1.4x and 1.3x for Pareto-Cascades and greedy optimized plans, respectively.

These benefits do not show up as clearly in the quality maximizing objective; this is in part because estimating quality is more difficult and may require more samples for this query. Third, on-average, the Pareto-Cascades algorithm helps identify plans which satisfy constraints 1.2x more often.

5 LIMITATIONS AND FUTURE WORK

Modeling Operator Dependencies. One limitation of Abacus's cost model is that it treats operator performance as being independent of the other operators in the plan. This assumption enables Abacus to produce cost estimates for logical plans which it has not sampled. However, it can also lead to errors in estimation. In future work we plan to explore using techniques from Bayesian optimization [7], which have recently been applied in similar declarative programming frameworks [19].

Sequential MAB Sampling. In Abacus' implementation of Algorithm 5, each operator frontier is updated in sequence in order to determine the new highest quality operator(s) on the frontier. The sample output(s) generated by the highest quality operator(s) are then used as input to the next operator frontier in Line 7 of Algorithm 1. A downside of this implementation is that the optimization algorithm is slowed by the sequential processing of operator frontiers (even though operators within a frontier may process sample inputs in parallel). In the future, we will explore pipelining execution of operator frontiers to decrease optimization overhead.

6 RELATED WORK

Optimizing Semantic Operator Systems. Recent work has investigated the optimization of semantic operator systems [2, 15, 22–24, 28, 31, 33, 37]. Prior to Abacus, Palimpzest [23] used sampling and heuristics to estimate and optimize semantic operator systems. LOTUS [28] optimizes semantic join, filter, group-by, and top-k operators by offloading data processing from an expensive "gold algorithm" to a cheaper proxy method while providing statistical guarantees on quality with respect to the gold algorithm. DocETL [33] uses LLMs to apply (and validate) query rewrites to data processing pipelines. In contrast to Abacus, LOTUS and DocETL only optimize for system quality and do not satisfy explicit constraints on system cost or latency.

Similar to DocETL, Aryn [2] uses an LLM to apply rewrites to query plans, however it focuses more on using a human-in-the-loop to validate the plans it generates. VectraFlow [24] built a stream processing engine with support for vector data and vector-based operations. Galois [31] introduced new logical and physical optimizations for answering queries with LLM-based operators.

Early work on semantic operators focused on adding machine learning classifiers to data systems for tasks such as image classification, object detection, sentiment analysis, and more [3, 16–18, 20, 29]. Caesura [37], EVA [15], and ZenDB [22] integrated semantic operators into systems which support SQL queries over multi-modal, video, and text data, respectively. In general, these systems support narrow optimizations over semantic operators, rather than building a new general-purpose optimizer for them.

Optimizing More General AI Systems. There is also a large body of work on building AI systems that go beyond using semantic operators. We focus our discussion on frameworks which treat the optimization of these systems as a primary challenge. DSPy [19, 27, 34] enabled users to construct and optimize "language model programs", i.e. workflows composed of modular operators which can be optimized in a declarative manner. The main levers of optimization included prompt optimization, parameter optimization, and model finetuning. More recently, AFlow [13, 43], Archon [30], and ADAS [14] all explored automatically constructing AI systems for a given workload. Each of these systems searches over a space of computation graphs and operator implementations, using Monte Carlo Tree Search, Bayesian optimization, and LLM-guided search, respectively. In contrast to these works, Abacus focuses solely on declarative optimization of semantic operator systems.

Relational Query Optimization. There is a long and rich literature on query optimization in relational database systems [8–10, 25, 26, 32]. From this line of work, Abacus most closely resembles a Cascades optimizer [8]. There are two key challenges which make optimizing semantic operator systems different from optimizing relational queries. First, the quality of a semantic operator is not guaranteed to be perfect. Thus, Abacus must be able to estimate the quality of an operator, possibly without the use of precomputed statistics. Second, in order to support constrained optimization Abacus cannot rely on the principle of optimality to prune sub-plans during its plan search. This necessitates Abacus's use of a new dynamic programming algorithm which maintains the Pareto frontier of physical plans for every subplan in its search.

7 CONCLUSION

We present Abacus, an extensible, cost-based optimizer for semantic operator systems which optimizes their quality, cost, and latency. Abacus modifies traditional multi-armed bandit and Cascades algorithms to overcome challenges in estimating the performance of semantic operators while supporting constrained optimization. We evaluate Abacus and its core algorithmic contributions on a diverse set of benchmarks. We demonstrate that its plans perform better than those produced by recent work, and its algorithmic contributions help decrease the number of samples required to achieve good performance and help satisfy optimization constraints.

ACKNOWLEDGMENTS

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, a Google Research Award, 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] Rajeev Agrawal. 1995. The Continuum-Armed Bandit Problem. *SIAM Journal on Control and Optimization* 33, 6 (1995), 1926–1951. https://doi.org/10.1137/S0363012992237273 arXiv:https://doi.org/10.1137/S0363012992237273

[2] Eric Anderson, Jonathan Fritz, Austin Lee, Bohou Li, Mark Lindblad, Henry Lindeman, Alex Meyer, Parth Parmar, Tanvi Ranade, Mehul A. Shah, Benjamin Sowell, Dan Tecuci, Vinayak Thapliyal, and Matt Welsh. 2025. The Design of an LLM-powered Unstructured Analytics System. CIDR.

[3] Michael R. Anderson, Michael J. Cafarella, Germán Ros, and Thomas F. Wenisch. 2018. Physical Representation-Based Predicate Optimization for a Visual Analytics Database. *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2018), 1466–1477. https://api.semanticscholar.org/CorpusID:48362547

[4] Song Dingjie, Shunian Chen, Guiming Hardy Chen, Fei Yu, Xiang Wan, and Benyou Wang. 2024. MileBench: Benchmarking MLLMs in Long Context. In *First Conference on Language Modeling*. https://openreview.net/forum?id=Uhwze2LEwq

[5] Karel D'Oosterlinck, Omar Khattab, François Remy, Thomas Demeester, Chris Develder, and Christopher Potts. 2024. In-Context Learning for Extreme Multi-Label Classification. *arXiv preprint arXiv:2401.12178* (2024).

[6] Karel D'Oosterlinck, François Remy, Johannes Deleu, Thomas Demeester, Chris Develder, Klim Zaporojets, Aneiss Ghodsi, Simon Ellershaw, Jack Collins, and Christopher Potts. 2023. BioDEX: Large-Scale Biomedical Adverse Drug Event Extraction for Real-World Pharmacovigilance. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 13425–13454. https://doi.org/10.18653/v1/2023.findings-emnlp.896

[7] Roman Garnett. 2023. *Bayesian Optimization*. Cambridge University Press, University Printing House, Shaftesbury Road, Cambridge, CB2 8BS, United Kingdom.

[8] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29. https://api.semanticscholar.org/CorpusID:260706023

[9] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*. IEEE Computer Society, USA, 209–218.

[10] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. 1989. Extensible query processing in starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 377–388. https://doi.org/10.1145/67544.66962

[11] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. *Architecture of a Database System*. Now Publishers Inc., Hanover, MA, USA.

[12] Dan Hendrycks, Collin Burns, Anya Chen, and Spencer Ball. 2021. CUAD: An Expert-Annotated NLP Dataset for Legal Contract Review. *NeurIPS* (2021).

[13] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VtmBAGCN7o

[14] Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated Design of Agentic Systems. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=t9U3LW7JVX

[15] Gaurav Tarlok Kakkar, Jiashen Cao, Pramod Chunduri, Zhuangdi Xu, Suryatej Reddy Vyalla, Prashanth Dintyala, Anirudh Prabakaran, Jaeho Bang, Aubhro Sengupta, Kaushik Ravichandran, Ishwarya Sivakumar, Aryan Rajoria, Ashmita Raju, Tushar Aggarwal, Abdullah Shah, Sanjana Garg, Shashank Suman, Myna Prasanna Kalluraya, Subrata Mitra, Ali Payani, Yao Lu, Umakishore Ramachandran, and Joy Arulraj. 2023. EVA: An End-to-End Exploratory Video Analytics System. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning* (Seattle, WA, USA) *(DEEM '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. https://doi.org/10.1145/3595360.3595858

[16] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 533–546. https://doi.org/10.14778/3372716.3372725

[17] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1586–1597. https://doi.org/10.14778/3137628.3137664

[18] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating approximate aggregation queries with expensive predicates. *Proc. VLDB Endow.* 14, 11 (July 2021), 2341–2354. https://doi.org/10.14778/3476249.3476285

[19] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *The Twelfth International Conference on Learning Representations*.

[20] Ferdi Kossmann, Ziniu Wu, Eugenie Lai, Nesime Tatbul, Lei Cao, Tim Kraska, and Sam Madden. 2023. Extract-Transform-Load for Video Streams. *Proc. VLDB Endow.* 16, 9 (May 2023), 2302–2315. https://doi.org/10.14778/3598581.3598600

[21] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 793, 16 pages.

[22] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G. Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. arXiv:2405.04674 [cs.DB] https://arxiv.org/abs/2405.04674

[23] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. CIDR.

[24] Duo Lu, Siming Feng, Jonathan Zhou, Franco Solleza, Malte Schwarzkopf, and Uğur Çetintemel. 2025. VectraFlow: Integrating Vectors into Stream Processing. CIDR.

[25] Thomas Neumann, Viktor Leis, and Alfons Kemper. 2017. The Complete Story of Joins (in HyPer). In *Datenbanksysteme für Business, Technologie und Web*. https://api.semanticscholar.org/CorpusID:32421956

[26] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[27] Krista Opsahl-Ong, Michael Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. 9340–9366. https://doi.org/10.18653/v1/2024.emnlp-main.525

[28] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proc. VLDB Endow.* 18, 11 (July 2025), 4171–4184. https://doi.org/10.14778/3749646.3749685

[29] Matthew Russo, Tatsunori Hashimoto, Daniel Kang, Yi Sun, and Matei Zaharia. 2023. Accelerating Aggregation Queries on Unstructured Streams of Data. *Proc. VLDB Endow.* 16, 11 (July 2023), 2897–2910. https://doi.org/10.14778/3611479.3611496

[30] Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Kumar Guha, E. Kelly Buchanan, Mayee F Chen, Neel Guha, Christopher Re, and Azalia Mirhoseini. 2025. An Architecture Search Framework for Inference-Time Techniques. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=EGrSMMj37o

[31] Dario Satriani, Enzo Veltri, Donatello Santoro, Sara Rosato, Simone Varriale, and Paolo Papotti. 2025. Logical and Physical Optimizations for SQL Query Execution over Large Language Models *(SIGMOD '25)*. Association for Computing Machinery, New York, NY, USA, 28. https://doi.org/10.1145/3725411

[32] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[33] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *Proc. VLDB Endow.* 18, 9 (May 2025), 3035–3048. https://doi.org/10.14778/3746405.3746426

[34] Dilara Soylu, Christopher Potts, and Omar Khattab. 2024. Fine-Tuning and Prompt Optimization: Two Great Steps that Work Better Together. 10696–10710. https://doi.org/10.48550/arXiv.2407.10930

[35] Alon Talmor, Ori Yoran, Amnon Catav, Dan Lahav, Yizhong Wang, Akari Asai, Gabriel Ilharco, Hannaneh Hajishirzi, and Jonathan Berant. 2021. MultiModal{QA}: complex question answering over text, tables and images. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ee6W5UgQLa

[36] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. https://openreview.net/forum?id=wCu6T5xFjeJ

[37] Matthias Urban and Carsten Binnig. 2024. Demonstrating CAESURA: Language Models as Multi-Modal Query Planners. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 472–475.

https://doi.org/10.1145/3626246.3654732

[38] Junlin Wang, Jue WANG, Ben Athiwaratkun, Ce Zhang, and James Zou. 2025. Mixture-of-Agents Enhances Large Language Model Capabilities. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=h0ZfDIrj7T

[39] Yizao Wang, Jean-Yves Audibert, and Rémi Munos. 2008. Algorithms for infinitely many-armed bandits. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems* (Vancouver, British Columbia, Canada) *(NIPS'08)*. Curran Associates Inc., Red Hook, NY, USA, 1729–1736.

[40] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024. MMLU-Pro: a more robust and challenging multi-task language understanding benchmark. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 3018, 25 pages.

[41] Yongwen Xu. 1998. EFFICIENCY IN THE COLUMBIA DATABASE QUERY OPTIMIZER. https://api.semanticscholar.org/CorpusID:60571693

[42] Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2024. MMMU: A Massive Multi-discipline Multimodal Understanding and Reasoning Benchmark for Expert AGI. In *Proceedings of CVPR*.

[43] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025. AFlow: Automating Agentic Workflow Generation. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=z5uVAKwmjf