Optimizing Machine Learning Inference Queries with Correlative Proxy Models

Zhihui Yang*
Zhejiang Lab, Hangzhou, China
zhyang14@zhejianglab.com

Zuozhi Wang
UC Irvine, CA, USA
zuozhiw@ics.uci.edu

Yicong Huang
UC Irvine, CA, USA
yicongh1@ics.uci.edu

Yao Lu
Microsoft Research, WA, USA
luyao@microsoft.com

Chen Li
UC Irvine, CA, USA
chenli@ics.uci.edu

X. Sean Wang
Fudan University, Shanghai, China
xywangcs@fudan.edu.cn

ABSTRACT

We consider accelerating machine learning (ML) inference queries on unstructured datasets. Expensive operators such as feature extractors and classifiers are deployed as user-defined functions (UDFs), which are not penetrable with classic query optimization techniques such as predicate push-down. Recent optimization schemes (e.g., Probabilistic Predicates or PP) assume independence among the query predicates, build a proxy model for each predicate offline, and rewrite a new query by injecting these cheap proxy models in the front of the expensive ML UDFs. In such a manner, unlikely inputs that do not satisfy query predicates are filtered early to bypass the ML UDFs. We show that enforcing the independence assumption in this context may result in sub-optimal plans. In this paper, we propose CORE, a query optimizer that better exploits the predicate correlations and accelerates ML inference queries. Our solution builds the proxy models online for a new query and leverages a branch-and-bound search process to reduce the building costs. Results on three real-world text, image and video datasets show that CORE improves the query throughput by up to 63% compared to PP and up to 80% compared to running the queries as it is.

PVLDB Reference Format:
Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. PVLDB, 15(10): 2032 - 2044, 2022.
doi:10.14778/3547305.3547310

1 INTRODUCTION

Modern DBMS systems apply machine learning (ML) inference as user-defined functions (UDFs) for complex analytics over unstructured texts, images, and videos [3, 11, 22, 24]. Example models include those extracting user sentiments from product reviews for market analysis [39] and those estimating vehicle counts from surveillance videos for traffic planning [13]. Consider the following query, where input tweets are processed by two ML UDFs, namely a geographic tagger ($\mathcal{F}_1$) and a sentiment analyzer ($\mathcal{F}_2$), to generate the predicate columns. These queries enable downstream visualization and statistics, such as word cloud that shows most frequent tokens, and users can accept approximate but fast results.

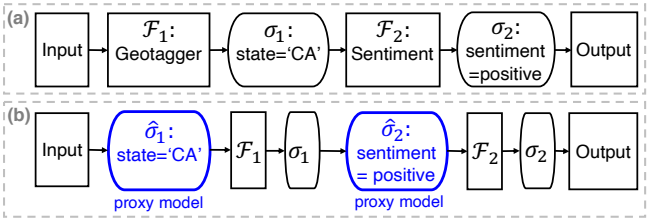


Figure 1: (a) An example query plan for tweet analysis. (b) An optimized query plan with proxy models.

```
SELECT F1(t) AS state, F2(t) AS sentiment
FROM Tweets AS t
WHERE state = 'CA' ∧ sentiment = positive;
```

Figure 1(a) demonstrates the plan of the above query, where σ_1 and σ_2 are the predicates `state = 'CA'` and `sentiment = positive`, respectively. ML queries are costly due to the expensive ML UDFs; improving the efficiency for ML inference has been a recent research focus [3, 11, 17, 21, 30] to provide an additional trade-off between accuracy and efficiency [9, 17, 30]. In our example, classic query optimization techniques such as predicate push-down cannot help much because σ_1 and σ_2 are stuck behind their corresponding ML UDFs regardless of their selectivity.

To optimize such ML inference queries, recent works [17, 30] propose to rewrite the query and insert a set of light-weight filters in front of the expensive ML UDFs, thus forming a *proxy model* [38]. Figure 1(b) demonstrates an example plan with two proxy models $\hat{\sigma}_1$ and $\hat{\sigma}_2$; they quickly discard input records that are unlikely to satisfy the predicates and thus improve the query performance.

In [30], a proxy model (i.e., "Probabilistic Predicate" or "PP") is specific to a predicate $c\phi v$, where c is a predicate column, ϕ is a comparison (e.g., $>$ or $=$), and v is a constant value. An independence assumption is made to train filters among different predicates directly using the raw input, regardless of the fact that each may have a different input relation. When ad-hoc queries with multiple predicates arrive, a query optimizer (QO) rewrites and accelerates the query by assembling individual filters and using them also in an independent manner. In many applications, query predicates are often correlated. In our example, sentiments may vary in different states – the sentiment in California can be different from that in Texas. As Section 2.2 will show, the QO in [30] overestimates the

*Part of the work was done at Fudan University and during a visit to UC Irvine.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547310

reduction when building the filters on the raw input and thus yields sub-optimal plans for a new query with correlated predicates.

Inspired by [17, 30] to optimize ML inference using proxy models, we intend to relax the independence assumption among different predicates. A proxy model hence is specific not only to a predicate but also to its input relation, i.e., prefix σ's and $\hat{\sigma}$'s, as well as parameter choices of prefix $\hat{\sigma}$'s. In Figure 1(b), $\hat{\sigma}_2$ learns upon filtering the raw input by $\hat{\sigma}_1 \wedge \sigma_1$[1]. Unlike [30] that builds a small number of independent filters, it is easy to see that relaxing the independence assumption may result in an untenable number of filters to build by enumerating their order and parameter choices.

We propose an optimizer called "CORE" that better exploits predicate correlations in ML inference. Given an ad-hoc query, CORE builds the proxy models *online* to avoid exhaustive offline filter construction. We describe a novel technique to accelerate such process at a small overhead (e.g., a few percent of the query processing) and a user-specified accuracy target. Extensive experiments for queries over datasets of tweets, images, and videos indicate that CORE improves the ML inference execution costs by up to 63% compared to [30] and up to 80% compared to running the workload as it is. Various downstream applications, such as interactive data exploration, can benefit from CORE due to a better resource utilization and a faster decision making.

To summarize, our key contributions are as follows:

- We show that correlations in predicates may harm the performance of a prior optimization scheme for ML inference [30].
- We propose CORE to accelerate ML inference and relax the independence assumption enforced by prior work. Our QO scheme prunes the space of candidate filters to build and incurs only a small computing overhead.
- Experiments on real-world ML-inference workloads and datasets show that CORE can achieve significant query-throughput improvements.

1.1 Related Work

Operator reordering in database optimization. [6, 10] studied the problem of reordering select-project-join operators in database systems. [2] studied how to order correlated predicates in streaming systems. It used a greedy algorithm for selection ordering and collected samples at runtime to estimate selectivity. Our query optimization algorithm gives an optimal solution and uses a branch-and-bound search to quickly prune plans in the space of proxy models. [34] studied various optimization techniques of complex user-defined functions on map-reduce-style big data systems, such as predicate simplification and UDF semantic inference. These techniques were orthogonal to our solution. Sampling-based approximate query processing techniques [5] provided approximate answers to queries by running queries on a small sampling subset of data. Our approach provides approximate answers by exploiting the accuracy of ML inference predicates.

Proxy models (a.k.a. cascaded filters) in machine learning. One of the first proxy models [38] cascaded a sequence of light-weight classifiers to discard background regions of an image to accelerate object detection. Later, proxy models were studied to improve the performance of classification [32], detection [4, 26], semantic image segmentation [27], and pose estimation [36]. Different from [4, 26, 27, 38] that used a cascade of classifiers to quickly reject sub-regions of an image, our CORE uses proxy models to reduce the size of records to be processed by ML UDFs. Unlike [32, 36] that integrated proxy models into DNN models to improve the performance during the training phase, our CORE uses proxy models as separate operators to accelerate ML inference.

Proxy models in databases. Recently proxy models have been applied in big-data systems to accelerate ML inference-based analysis tasks [12, 16–19, 23, 30, 40]. NoScope [17] firstly cascaded a cheap specialized model before expensive DNNs to accelerate selection video queries. After it, certain classes of video queries including selection without guarantees [12], selection with statistical guarantees [18], aggregation [16] and limit queries [16] was optimized using proxy models. A general index solution in [19] was proposed to accelerate these video queries over the schema induced by the target DNN. Probabilistic predicates (PP's) [30] optimized various domain queries by inserting multiple offline-built proxy models before expensive ML UDFs with an assumption of independence between predicates. Different from [12, 16–18, 23, 40], PP and our proposed CORE cascade general proxy models, which are applicable to a variety of domains. CORE follows this line of work and further relaxes the independence assumption of the predicates.

2 PROXY MODELS

We briefly review the background of proxy models and then study the impact of correlations to proxy models.

2.1 Background

Proxy models have been studied for decades to accelerate ML inference. Jones et al. [38] cascade weak classifiers as proxy models to speed-up face detection in images. Recently, techniques of using cheaper but less accurate ML models to accelerate ML models in [4, 26, 27, 32, 36, 38] attracted attention in big data systems. We briefly review two related solutions [17, 30] and refer the readers to their papers for more details.

NoScope (NS) [17] aims to process video queries such as "finding video frames with vehicles" and "finding video frames with pedestrians" using an object-detector UDF. It builds and applies a proxy model, i.e., a cheaper object detector using shallow Neural Networks (NNs), which has the same semantics as the object-detector UDF. NoScope has to train for each query predicate and thus has large building costs when the query predicates are ad-hoc or complex.

Probabilistic Predicate (PP) [30], as mentioned earlier, is another form of proxy models. Each PP is a cheap classifier to predict the likelihood of an input record matching a predicate clause. Easy inputs with a small likelihood will be discarded immediately, while hard inputs will be processed further by subsequent ML UDFs. For ad-hoc queries with complex predicates, a query optimizer assembles multiple PPs built offline, and a dynamic programming algorithm is leveraged to achieve a maximum reduction, under the independence assumption in queries. However, this assumption made in PP limits its use to broader applications. *Dependency between columns is the rule, rather than the exception, in the real*

[1] $\mathcal{F}_1$ is a row processor and does not filter as σ_1 and $\hat{\sigma}_1$ do.

world [14]. In the following, we conduct a controlled experiment to study the impact of correlations to proxy models.

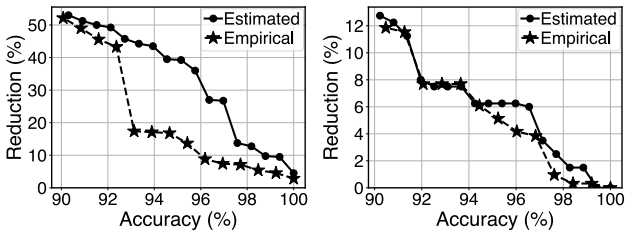
2.2 Impact of Correlations

To better understand the impact of correlations in processing ML inference queries, we leverage the correlation score provided by CORDS [14]. Specifically, let d_1 and d_2 be the distinct counts in a pair of columns. The correlation score is computed by a chi-squared test upon a sample of n-rows:

$$\hat{\kappa}^2 = \frac{1}{n(\min(d_1, d_2) - 1)} \sum_{i=1}^{d_1} \sum_{j=1}^{d_2} \frac{(n_{ij} - n_{i\cdot} n_{\cdot j})^2}{n_{i\cdot} n_{\cdot j}},$$

where n_{ij} is the frequency of distinct tuple i, j, and $n_{i\cdot}$, $n_{\cdot j}$ are the marginal frequency. A larger $\hat{\kappa}^2$ value indicates a stronger correlation between the columns. For example, we can follow CORDS to use a sample of 10K rows and normalize the correlations scores by the maximum number in all the predicate pairs. All other algorithmic details follow the CORDS paper [14].

Why correlation matters for PP**?** We explain the reason using the Twitter dataset and two queries, q and q', each with two predicates of different kinds of correlation. We illustrate these two queries in Appendix A.1 in the technical report [42].The correlation between the q predicates is stronger (2.5 ×) than that of the q' predicates. The PP filters are trained offline for each predicate without considering the context in which the predicate is applied. We collect the estimated accuracy-reduction curves for the *second* PP in q and q' during the training phase and illustrate them in Figures 2a and 2b, respectively. Two proxy models $\hat{\sigma}_1$ and $\hat{\sigma}_2$ are connected for the predicate $\sigma_1 \wedge \sigma_2$.



(a) Strongly correlated query q. **(b) Weakly correlated query q'.**

Figure 2: The estimated and empirical accuracy-reduction curves of the *second* PP filters in a *strongly* correlated query q and a *weakly* correlated query q'. Correlation results in overestimated reductions offline in PP.

When σ_1 and σ_2 are correlated and $\hat{\sigma}_1$ discards a row that matches σ_1, the discarded row is also likely to match σ_2 because of the correlation. In general, the empirical reduction produced by $\hat{\sigma}_2$ is less than the estimated reduction as shown in Figure 2, because there are fewer input rows for σ_2 after $\hat{\sigma}_1$. When there is a strong correlation, the reductions can be overestimated. For example, as shown by q with a strong correlation in Figure 2a, when the accuracy is 95%, the estimated data reduction is 40%, and the empirical value is 15%. At the same accuracy, the difference of the reduction ratio for q' in Figure 2b with a weak correlation is at most 2%. As a result, with strong correlations, PP unnecessarily routes more inputs to the expensive ML UDFs and thus yields a lower performance speedup. This example shows that the optimizer in previous work overestimates the reduction of the proxy models built offline, thus yielding suboptimal query plans and less performance improvement for a new query with correlated predicates; this limits the use of PPs to broader applications.

3 CORE OVERVIEW

In this section we give an overview of CORE and formally define its optimization problem.

3.1 System Architecture

In Figure 3, the input of CORE is a query that includes multiple ML inference UDFs. These UDFs, as seen in the previous section, depict row manipulators; they produce one output row per input row. ML UDFs wrap operations such as feature extraction or classification. CORE optimizes the input query by building proxy models online and generates a more efficient plan q^*. We build proxy models for predicates of the form $\mathtt{c}\phi\mathtt{v}$. Meanwhile, a query can have one or more predicate clauses in conjunction: $\bigwedge \mathtt{c}\phi\mathtt{v}$. A small portion of the input data (e.g., k%) is used to build proxy models, and the remaining data is processed by the optimized plan q^*. We follow the scope of previous papers such as NoScope [17] and PP [30] to focus on approximate selection queries.

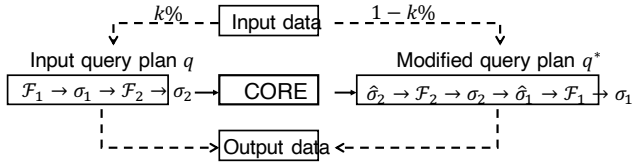


Figure 3: Given a query plan q, CORE generates an optimized plan q^* by applying proxy models. Part of the input data (k%) is used for building proxy models, and the remaining data is processed by q^*.

Definition 1. *A proxy model $\hat{\sigma}$ is characterized by a tuple*

$$\{d, \sigma, M, L, R\},$$

where d is an input relation (i.e., applying a sequence of prefix filters on the raw input), and σ is a target predicate that $\hat{\sigma}$ aims to improve; M is a regression model used by $\hat{\sigma}$ to produce a scoring function for each input record; L is a labeled sample from the input relation d to build M; and R is a mapping from an accuracy α to a reduction r. For the example in Figure 1(b), $\hat{\sigma}_1$ is built for the input relation $d_1 = \emptyset$ (raw input) and the predicate σ_1 : `state='CA'`, while $\hat{\sigma}_2$ is built for $d_2 = (\hat{\sigma}_1, \sigma_1)$ and σ_2 : `sentiment=positive`. The mapping R will be explained shortly.

Building proxy models online consists of collecting L and then training M. We leverage the initial stream of the input data for L (e.g., a few thousand rows). The labeled sample L is obtained by applying the filters specified in d upon the raw input and then labeling by predicate σ. The label is +1 if σ is satisfied, and -1 otherwise. Next, we use light-weight regression models such as linear SVMs [15] or shallow NNs [25] to train M.

Given an input record $\mathbf{x}$, a proxy model predicts a score $M(\mathbf{x})$. For example, for linear SVM, $M(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$, where $\mathbf{w}$ is a weighted

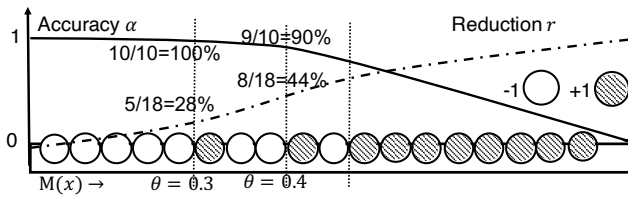


Figure 4: Relationship between an accuracy α and a reduction ratio r in a proxy model. Records are ranked in ascending order according to their $M(x)$ scores along the x-axis. White and dark circles represent records with -1 and +1 labels, respectively.

matrix and b is a bias term. Record $\mathbf{x}$ will be discarded if $M(\mathbf{x}) < \theta$ (for a threshold θ), and in this case the record is called a *negative example*. As in [30], the accuracy is the percentage of positive records being passed by a proxy model relative to all positive records. The data reduction is the percentage of records being discarded relative to all input records. In Figure 4, setting $\theta = 0.3$ results in all positive records being passed (i.e., the accuracy is 100%), and 5 out of 18 total records being discarded (i.e., the reduction is 28%). Setting $\theta = 0.4$ results in 9 records of 10 total positive records being passed (i.e., the accuracy is 90%), and 8 out of 18 records being discarded (i.e., the reduction is 44%). It is clear that a higher θ yields a lower accuracy and a higher data reduction. Such early filtering is a trade-off between accuracy and data reduction. Note that the mapping between α and r given θ can be evaluated using a validation set. In the rest of the paper we denote such a relationship as R. We can compute it by evaluating $\hat{\sigma}$ on a validation set from the initial stream of the input records.

Then, our developed query optimizer injects $\hat{\sigma}$ into the query plan right before the corresponding ML UDF that generates the σ predicate column (Figure 1(b)) for the remaining input records.

Query optimization by applying proxy models. We borrow the AQP-style query interface in [30]. Specifically, the user issues a query and specifies a global target accuracy $\mathscr{A}$ that depicts the level of false negatives of the proxy models in addition to those caused by the UDF. Note that the UDFs themselves produce false positives and negatives and we do not intend to break the black boxes to improve their accuracy and performance. $\mathscr{A}$ is the percentage of the output of an original query q kept by its optimized query q^* (Figure 3). It is a value between 0 and 1. It sets the trade-off goals between additional errors and query-processing speedups. Our QO builds the proxy models, considers their combinations, allocates their accuracy parameters, and injects them into the modified query plan q^*. To reduce the computing overhead and latency of building the proxy models before the input query can be accelerated, the QO reuses intermediate results during the filter construction and prunes candidate plans using a branch-and-bound search.

3.2 Formulation of Optimization Problem

Given an ML query q with UDFs $\mathscr{F}_1, \ldots, \mathscr{F}_n$, predicate filters $\sigma_1, \ldots, \sigma_n$, and a query-level target accuracy $\mathscr{A}$, we aim to build proxy models $\hat{\sigma}_1, \ldots, \hat{\sigma}_n$ with their accuracy parameters $\alpha_1, \ldots, \alpha_n$ so that $\mathscr{A}$ is met. Let the execution costs of applying $\hat{\sigma}_i$ and the ML UDF $\mathscr{F}_i$ be $\hat{c}_i$ and c_i, respectively. For a pair of a proxy model $\hat{\sigma}_i$ and its corresponding ML UDF $\mathscr{F}_i$ (i.e., $\hat{\sigma}_i \wedge \mathscr{F}_i$), its input cardinality is $\prod_{j=1}^{i-1} s_j \cdot \alpha_j$. The execution cost of the pair is

Table 1: Notations used in this paper.

Notation	Meaning
σ	A filter predicate after an ML UDF.
$\hat{\sigma}$	A cheap proxy model that has the same semantics as σ.
d	The input relation of a proxy model $\hat{\sigma}$.
L, M, R	The labeled sample, trained classifier, and accuracy-reduction curve for a proxy model, respectively.
α, r	A proxy model's accuracy and the achieved reduction ratio.
$q, \mathscr{A}$	A query and a query-level target accuracy specified by a user.
s_i	The selectivity of σ_i on the condition of prefix $\hat{\sigma}_1, \ldots, \hat{\sigma}_{i-1}$ and $\sigma_1, \ldots, \sigma_{i-1}$, i.e., $\sigma_i \vert (\hat{\sigma}_1, \ldots, \hat{\sigma}_{i-1}, \sigma_1, \ldots, \sigma_{i-1})$.
$\hat{c}_i, c$	The execution cost for $\hat{\sigma}$ and an ML UDF $\mathscr{F}$.
π	An order of proxy models.
C_i^l, C_i^u	Lower and upper bounds of execution cost for a pair $(\hat{\sigma}_i, \mathscr{F}_i)$.

$$C(\hat{\sigma}_i, \alpha_i) = (\prod_{j=1}^{i-1} s_j \cdot \alpha_j) \cdot (\hat{c}_i + (1 - r_i) \cdot c_i), \tag{3.1}$$

where α_i is the accuracy of $\hat{\sigma}_i$, r_i is the reduction of $\hat{\sigma}_i$, and s_i is the conditional selectivity of predicate σ_i with prior filters $\hat{\sigma}_1, \ldots, \hat{\sigma}_{i-1}$, $\sigma_1, \ldots, \sigma_{i-1}$.

In an original query q, let $\bar{s}_i$ be the conditional selectivity of σ_i with prior $\sigma_1, \ldots, \sigma_{i-1}$. In an optimized query q^*, let $\hat{s}_i$ be the conditional selectivity of $\hat{\sigma}_i \wedge \sigma_i$ with prior $\hat{\sigma}_1, \ldots, \hat{\sigma}_{i-1}, \sigma_1, \ldots, \sigma_{i-1}$. According to the accuracy definition in [30], the accuracy of $\hat{\sigma}_i$ can be computed as:

$$\alpha_i = \hat{s}_i / s_i, \tag{3.2}$$

which is the percentage of the output by σ_i kept by $\hat{\sigma}_i \wedge \sigma_i$. The output selectivity of the original query q is $\prod_{i=1}^{n} \bar{s}_i$, and the output selectivity of an optimized plan q^* is $\prod_{i=1}^{n} \hat{s}_i$. The query accuracy $\mathscr{A}$ can be computed as $\mathscr{A} = \prod_{i=1}^{n} (\hat{s}_i / \bar{s}_i)$. When building proxy models, their accuracy parameters and $\mathscr{A}$ satisfy

$$\prod_i \alpha_i \cdot \delta_i = \mathscr{A},$$

where $\delta_i = s_i / \bar{s}_i$. δ_i is at most $1/(\prod_{j=1}^{i-1} (\alpha_j \cdot \delta_j))$ and its value is always smaller than $1/\mathscr{A}$. The detailed derivation of a lower bound and an upper bound of δ_i is in Appendix A.2 in [42]. For simplicity, we use α_i to refer $\alpha_i \cdot \delta_i$ in the following sections.

Example. We demonstrate the number of passing records by each filter for the example query in Figure 5. In Figure 5(a), $\delta_2 = s_2 / \bar{s}_2$, where $\bar{s}_2 = 60/100$ is the conditional selectivity of the predicate `sentiment=positive` with a prior conditional predicate `state="CA"` (i.e., σ_1); $s_2 = 56/96$ is the conditional selectivity of the same predicate with a prior condition $\hat{\sigma}_1 \wedge \sigma_1$ in Figure 5(b). Hence, $\delta_2 = s_2 / \bar{s}_2 = (56/96)/(60/100) = 0.972$, which measures the changes of the input of σ_2 after adding its prefix proxy model $\hat{\sigma}_1$. This proxy model changes the input data size of σ_2 from 100 to 96 because $\hat{\sigma}_1$ discards 4 tweets satisfying `state="CA"`. Similarly, $\delta_1 = s_1 / \bar{s}_1 = (100/200)/(100/200) = 1$, since σ_1 is the first filter and there is no prefix proxy model changing the input of σ_1.

To this end, the target accuracy $\mathscr{A}$ is calculated as $\mathscr{A} = 54/60 = 0.9$, which is the percentage of the output of the original query in Figure 5(a) (i.e., 60 tweets) kept by its optimized plan in Figure 5(c) (i.e., 54 tweets). For each proxy model $\hat{\sigma}_i$, α_i is the percentage of the output by σ_i kept by $\hat{\sigma}_i \wedge \sigma_i$. In Figure 5(b), $\alpha_1 = 96/100 = 0.96$, as $\hat{\sigma}_1 \wedge \sigma_1$ keeps 96 tweets in Figure 5(b) and σ_1 keeps 100 tweets in Figure 5(a). Similarly, $\alpha_2 = 54/56 = 0.964$. As mentioned before,

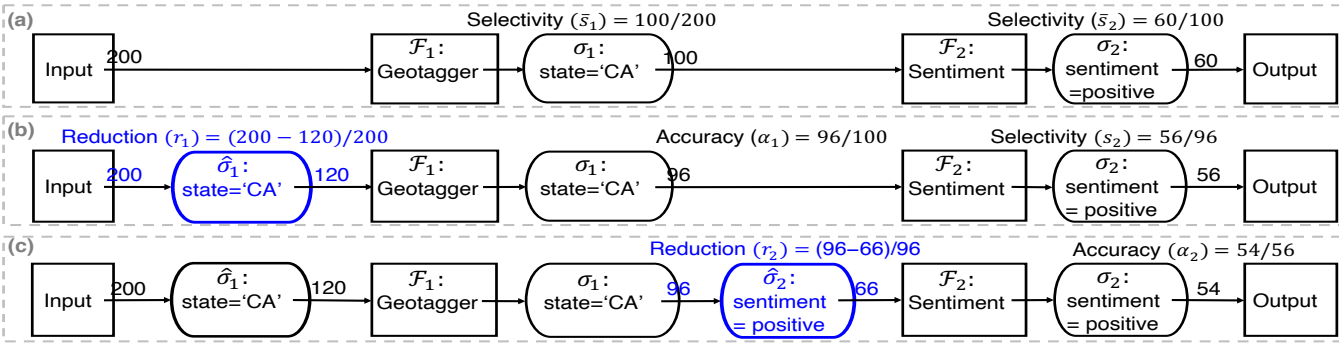


Figure 5: Step-by-step demonstration of inserting two proxy models to optimize a query. (a) An original query plan; (b) A query plan with $\hat{\sigma}_1$ inserted; (c) A query plan with $\hat{\sigma}_1$ and $\hat{\sigma}_2$ inserted. Each edge depicts the number of passing tweets. Selectivity (i.e., $\bar{s}_i, s_i$), reduction (i.e., r_i), and accuracy (i.e., α_i) values are illustrated. The overall query accuracy is $\mathcal{A} = 54/60$.

$\delta_1 = 1$ and $\delta_2 = 0.972$. Both of them measure the input relation changes for σ_1 and σ_2 respectively when applying proxy models. Finally, we have $\alpha_1 \cdot \delta_1 \cdot \alpha_2 \cdot \delta_2 = 0.9 = \mathcal{A}$. In general, relaxing the independence assumption among predicates results in introducing a input relation change factor δ caused by its prefix proxy model.

Problem Statement. Let π be an order of the ML UDFs and predicate filters. Let $\hat{\sigma}_{\pi_i}$ denote the π_i-th proxy model. Our QO finds the following optimal query plan in the order space $\pi \in \mathbb{H}$ and the accuracy space $\mathbb{A}$:

$$\arg \min_{\pi \in \mathbb{H}, \alpha \in \mathbb{A}} \sum_i C(\hat{\sigma}_{\pi_i}, \alpha_{\pi_i}), s.t. \prod_i \alpha_{\pi_i} = \mathcal{A}. \tag{3.3}$$

Finding an optimal order π of $\hat{\sigma}$ and allocating their parameter α, simultaneously, is NP-hard, as shown in Theorem 1 in [42].Since both r and s depend on d and the input relation of $\hat{\sigma}$ (i.e., prefix σ, $\hat{\sigma}$, and α choices), building $\hat{\sigma}$ offline by enumerating possible d incurs large computing costs. We seek a solution such that each $\hat{\sigma}$ is built on-the-fly on a materialized sample L of its input relation d. A main challenge is that, given the accuracy target, how to efficiently build $\hat{\sigma}$ with a small computing overhead with taking its input relation into account. We describe our solution to find an optimal set of accuracy parameters $\alpha \in \mathbb{A}$ given an order π in Section 4, and study how to find an optimal order $\pi \in \mathbb{H}$ in Section 5. Both sub-problems exhibit unique structures that can be leveraged for acceleration. Table 1 summarizes the notations used in the paper.

4 CORE: ACCURACY ALLOCATION

In this section, we present an efficient algorithm in CORE for deriving an optimal accuracy allocation $\alpha_{\pi_1}, \ldots, \alpha_{\pi_n}$ among different $\hat{\sigma}_{\pi_i}$ for a given order π to achieve a minimum cost $\sum_i C(\hat{\sigma}_{\pi_i}, \alpha_{\pi_i})$.

4.1 A Basic Approach and its Challenge

One approach to allocating the accuracy is as follows. We first discretize $\mathbb{A}$ with a fixed step size. For each candidate α_{π_i} satisfying $\prod_i \alpha_{\pi_i} \geq \mathcal{A}$, we build a proxy model in the order of π. We obtain a labeled sample given its input relation, train a classifier, and derive reduction as mentioned in Section 3. After building $\hat{\sigma}_{\pi_i}$, we compute its cost using Equation 3.1, and find an optimal α for a minimal cost. A main challenge is that building proxy models online is time-consuming for two reasons. (i) There are an exponential number of candidates $\hat{\sigma}_{\pi_i}$'s. (ii) For each proxy model, generating a labeled sample and training a classifier can be computationally costly.

To solve this problem, we present Algorithm 1, which accelerates the construction given input relations specified in π by reusing previously materialized samples and trained models. Next we will present the details of the algorithm.

Algorithm 1: Accuracy allocation

```
1:  procedure ACCURACY_ALLOCATION(π, 𝒜)
2:      L'_{π_0} ← raw input;
3:      for α = ⟨α_{π_1}, ..., α_{π_n}⟩ in discretized 𝔸, s.t. ∏_i α_{π_i} = 𝒜:
4:          for i ∈ {1, ..., n}:
5:              if L'_{π_i} is not materialized:
6:                  L'_{π_i} ← Apply σ_{π_i} on L'_{π_{i-1}};
7:              L_{π_i} ← Apply σ̂_{π_1}, ..., σ̂_{π_{i-1}} on L'_{π_i} with α;
8:              Reuse σ̂*_{π_i} if ϵ-approx on L_{π_i} else retrain;
9:              Compute C(σ̂_{π_i}, α_{π_i});
10:         Compute cost ∑_i C(σ̂_{π_i}, α_{π_i});
11:     Pick α* in 𝔸 with a minimum cost;
12:     Retrain σ̂_{π_1}, ..., σ̂_{π_n} with α*;
13:     return σ̂_{π_1}, ..., σ̂_{π_n} and α*_{π_1}, ..., α*_{π_n}.
```

4.2 Search Framework

As shown in the following example, the objective function (the cost $\sum_i C(\hat{\sigma}_{\pi_i}, \alpha_{\pi_i})$ subject to α) is non-convex, which means there could be multiple locally optimal solutions. In order to find a globally optimal solution, we use an exhaustive search framework in the algorithm (lines 3 ∼ 4). If a locally optimal solution is acceptable by the user, the algorithm can be easily extended to other search frameworks, such as hill climbing, by replacing lines 3 ∼ 4.

To illustrate that the objective function is non-convex, we construct an example with $n = 2$. The cost of applying each proxy model before its corresponding ML UDF could be any non-decreasing function over its accuracy. This is because the reduction decreases with the increase of accuracy [30]. Two example costs are the following:

$$C(\hat{\sigma}_1, \alpha_1) = 1 - (\alpha_1 - 1)^2, \alpha_1 \in [0, 1].$$

$$C(\hat{\sigma}_2, \alpha_2) = e^{-(2\mathcal{A}/\alpha_2 - 1)^3}, \alpha_2 \in [0, 1].$$

Both $C(\hat{\sigma}_1, \alpha_1)$ and $C(\hat{\sigma}_2, \alpha_2)$ increase monotonically when $\alpha_1 \in [0, 1]$ and $\alpha_2 \in [0, 1]$. The cost function $f = \sum C$ is

$$e^{-(2x-1)^3} + 1 - (x-1)^2, x \in [0, 1].$$

If the function f is convex on an interval $[0, 1]$, by definition [8], for any two points x_1 and x_2 in $[0, 1]$ and any λ where $0 < \lambda < 1$,

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2).$$

However, when $x_1 = 0.1, x_2 = 0.5$ and $\lambda = 1/2$, $f(\frac{x_1+x_2}{2}) = 1.17$; $\frac{f(x_1)+f(x_2)}{2} = 1.12$. So f does not satisfy $f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$. Thus f is not convex.

4.3 Reusing Samples to Reduce Labeling Costs

We first give a theorem about the proxy models, then show how the algorithm leverages the theorem to reuse samples.

4.3.1 Commutative proxy models. We note that the order of prefix filters is interchangeable as shown in Theorem 2 in in Appendix A.4 in the technical report [42]. In Figure 5(b), the 96 output tweets after $\hat{\sigma}_1 \wedge \sigma_1$ with $\alpha_1 = 0.96$ are the same as the output tweets of applying $\hat{\sigma}_1$ with $\alpha_1 = 0.96$ on the 100 output tweets after σ_1 in Figure 5(a). That is, with $\alpha_1 = 0.96$, applying $\sigma_1 \wedge \hat{\sigma}_1$ and applying $\hat{\sigma}_1 \wedge \sigma_1$ have the same results. To prove the theorem, we introduce Lemma 1 to prove a base case that a pair of $\hat{\sigma} \wedge \sigma$ are commutative, and Lemma 4 (in [42]) to prove an inductive case that two pairs of $\hat{\sigma} \wedge \sigma$ are still commutative with the same prefix filter and the same suffix filter, respectively.

Lemma 1. *Given a list of records L, a filter σ, and a proxy model $\hat{\sigma}$ with a parameter α, σ and $\hat{\sigma}$ with α are commutative, i.e., the results after applying $\hat{\sigma} \wedge \sigma$ are the same as those after applying $\sigma \wedge \hat{\sigma}$. We denote $\hat{\sigma} \wedge \sigma = \sigma \wedge \hat{\sigma}$.*

Proof. We first prove that $\hat{\sigma}$ with a specific α parameter is a selection predicate, and $\hat{\sigma}$ predicts the same output for a record x_1 independent of different orders of x_1 (x_1, x_2 or x_2, x_1) and different orders of $\hat{\sigma}$ ($\hat{\sigma} \wedge \sigma$ or $\sigma \wedge \hat{\sigma}$). According to Definition 1, a proxy model $\hat{\sigma}$ is built based on its input relation d and a target predicate. After building $\hat{\sigma}$ and allocating an accuracy α, $\hat{\sigma}$ is a selection predicate with fixed values of α, r, and M. When applying $\hat{\sigma}$, any input record cannot change $\hat{\sigma}$. Consider two records x_1 and x_2, where $\hat{\sigma}$ passes x_1 and discards x_2. The output of $\hat{\sigma}$ with different input orders (x_1, x_2 and x_2, x_1) is the same record x_1. For $\sigma \wedge \hat{\sigma}$, an unseen record x for $\hat{\sigma}$ is the one passed by σ. If $\hat{\sigma}$ passes x, then x is in the output of $\sigma \wedge \hat{\sigma}$ and also in the output of $\hat{\sigma} \wedge \sigma$. Otherwise, x is not in their outputs. For $\hat{\sigma} \wedge \sigma$, $\hat{\sigma}$ takes more input records, compared to $\sigma \wedge \hat{\sigma}$. There is no unseen record for $\hat{\sigma}$.

As selection predicates are commutative in general, σ and $\hat{\sigma}$ with α are commutative. □

4.3.2 Reusing samples. The algorithm improves the performance by reusing early samples (lines 5 to 7). L_{π_i} is the sampled input to build $\hat{\sigma}_{\pi_i}$ by applying predicate σ_{π_i} on the input relation d_{π_i}. In Figure 5(b), the labeled sample L_2 for $\hat{\sigma}_2$ has 96 tweets, which are filtered by $\hat{\sigma}_1 \wedge \sigma_1$ on the raw input and then labeled using the predicate `sentiment=positive`. It is easy to see that L_{π_i} changes when accuracies assigned to its prefix proxy models (i.e., $\alpha_{\pi_1}, \ldots, \alpha_{\pi_{i-1}}$) change. For example, in Figure 5(b), L_2 changes from 97 tweets to 96 tweets when the accuracy parameter of its prefix $\hat{\sigma}_1$ changes from $\alpha_1 = 0.97$ to $\alpha_1 = 0.96$.

By leveraging Theorem 2, we can improve the performance by materializing samples L' after σ, and applying $\hat{\sigma}$ on L' during the search, since common L' can be shared for different α choices. L_{π_i} can be obtained by applying $\hat{\sigma}_{\pi_1}, \ldots, \hat{\sigma}_{\pi_{i-1}}$ on a pre-computed sample L'_{π_i} that is computed by applying $\sigma_{\pi_1}, \ldots, \sigma_{\pi_{i-1}}$ on the raw input. Lines 5 to 7 illustrate this process of quickly deriving L for each α search. For the proxy model $\hat{\sigma}_2$, we materialize its corresponding sample L'_2 containing 100 tweets filtered by σ_1 in Figure 5(a) to be reused. When $\alpha_1 = 0.97$, the labeled sample L_2 can be obtained by applying prefix $\hat{\sigma}_1$ with $\alpha_1 = 0.97$ on the 100 materialized tweets and producing 97 tweets. Similarly, when α_1 changes to 0.96 in Figure 5(b), the labeled sample L_2 can be obtained by applying $\hat{\sigma}_1$ with $\alpha_1 = 0.96$ on the already materialized sample L'_2 of 100 tweets and producing 96 tweets. This solution is simple but effective, since applying $\hat{\sigma}$ is cheap and doing so allows us to evaluate each expensive $\mathcal{F}$ and σ only once.

4.4 Reusing Classifiers to Reduce Training Costs

The algorithm adopts a classifier-reusing scheme (line 8) to avoid repeated training classifiers when the prefix proxy models change their accuracy assignments. Specifically, let $\hat{\sigma}^*$ trained on L^* with α from a previous iteration (line 3) be ϵ-approximate [1] to $\hat{\sigma}$ trained on L. That is:

$$(1-\epsilon)\phi^*(L^*) \leq \phi^* L \leq (1+\epsilon)\phi^* L^*, \tag{4.1}$$

where ϕ is the objective function of the regressor model used by the proxy model. ϕ can be computed using a scoring function, such as F1 score or coreset [1]. Take the F1 scoring function as an example. We efficiently compute ϕ by evaluating $\hat{\sigma}^*$ from a previous iteration and measuring its F1 score on its labeled sample L^* and current L [1]. $\hat{\sigma}^*$ can be reused if it is ϵ-approximate under the current accuracy setting. In Figure 5(b), suppose we want to build the proxy model $\hat{\sigma}_2$ for the predicate `sentiment=positive` on its 96 labeled tweets with prefix $\alpha_1 = 0.96$. If there is a proxy model $\hat{\sigma}_2^*$ trained on 97 tweets with prefix $\alpha_1 = 0.97$ satisfying Equation 4.1, we reuse the classifier in $\hat{\sigma}_2^*$ (i.e., M_2^*) without training a new classifier on the 96 tweets. In Equation 4.1, we compute $\phi^*(L^*)$ by evaluating the F1 score of M_2^* on the 97 tweets, while $\phi^*(L)$ is on the 96 tweets.

We next discuss how to compute $C(\hat{\sigma}_i, \alpha_i)$ (line 9). The per-row cost $\hat{c}$ for $\hat{\sigma}$ and c for $\mathcal{F}$ can be profiled during training or by counting the FLOPS of the ML model, while r can be obtained from R, and s can be measured by applying the prefix filters on a sample of the raw input. Since applying the proxy models is computationally cheap, C can be computed efficiently. In Figure 5, the cost of the ML UDF `Geotagger` is 20ms per tweet in our experiments, while that of the proxy model $\hat{\sigma}_1$ is 0.01ms per tweet. The proxy model $\hat{\sigma}_1$ with $\alpha_1 = 0.96$ pays the cost of processing 200 tweets and saves the cost of the 80 discarded tweets, which no longer need to be processed by the ML UDF `Geotagger`. Therefore, using Equation 3.1, we have $C(\hat{\sigma}_1, \alpha_1) = \hat{c}_1 + (1 - r_1) \cdot c_1 = 0.01 + (1 - 80/200) \cdot 20 = 12.01$.

5 CORE: REORDERING PROXY MODELS

In this section we study how to reorder proxy models to find an optimal order $\pi \in \mathbb{H}$ to minimize the cost $\sum C$. For different orders, proxy models built on input relations and predicates are different and they have different costs. For instance, in Figure 5(c), for the order `state = "CA"`$\wedge$`sentiment = positive`, the proxy model for predicate `state = "CA"` is built on the original input data. For the order `sentiment = positive`$\wedge$`state = "CA"`, the proxy model for the same predicate is built on records satisfying the predicate `sentiment = positive`. Because different orderings affect the input data to the proxy model, these two proxy models have different execution costs for the same ML UDF `Geotagger`.

The number of query plans in $\mathbb{H}$ is exponential in terms of the number of UDFs and filters. We construct a search tree to represent them by merging common prefixes of query plans. For example, let X, Y, and Z be three ML UDFs. There are six potential plans in $\mathbb{H}$ (e.g., XYZ and XZY). Figure 6 shows a snippet of the search tree starting from node X, where each tree node represents an ML UDF $\mathcal{F}$ and its corresponding $\hat{\sigma}$ and σ. In general, building all proxy models for the plans can be computationally prohibitive. To find an optimal order π efficiently, we propose a search algorithm based on branch-and-bound [20, 29] to prune candidate plans.

5.1 Bounded Cost

For a specific order of proxy models, we can compute a lower bound and an upper bound of the cost $\sum C$. Intuitively, an initial lower bound corresponds to the case when all proxy models discard everything. An initial upper bound corresponds to the case when all proxy models discard nothing. For example, for the order XYZ in Figure 6, the cost function reaches a lower bound when the first proxy model $\hat{\sigma}_X$ discards all its input records. It reaches an upper bound when all proxy models $\hat{\sigma}_X$, $\hat{\sigma}_Y$, and $\hat{\sigma}_Z$ discard nothing.

Let C^l and C^u be the lower and upper bounds of the cost for a node, respectively. As shown in Equation 3.1, the cost C of a proxy model $\hat{\sigma}$ is bounded by accuracy α, reduction r, and selectivity s, where (i) $\alpha \in [\mathscr{A}, 1]$, (ii) $s \in [0, 1]$ and (iii) $r \in [0, 1]$. C increases when s and α increase and r decreases. To calculate a lower bound of node t at depth i assuming the depth of the root is 0, we use the minimal value of the accuracy $\alpha_i^l = \mathscr{A}$, the minimal value of the selectivity $s_i^l = 0$, and the maximum value of the reduction $r_i^u = 1$. Similarly, to compute an upper bound of t, we use the maximum value of the accuracy $\alpha_i^u = 1$, the maximum value of the selectivity $s_i^u = 1$, and the minimal value of the reduction $r_i^l = 0$. Based on the analysis, we present a lower bound and an upper bound of the cost C of a node t in Lemma 2. Additionally, a lower bound of the cost for a plan is the sum of the lower bound of the cost for each node in the plan, and an upper bound for a plan is the sum of the upper bound for each node in the plan. That is, the bounds of $\sum C$ for a plan are $\sum C^l$ and $\sum C^u$, respectively.

Lemma 2. *For a node t of depth i, a lower bound of its cost C_t is*

$$\Big(\prod_{j=1}^{i-1} s_j^l \cdot \alpha_j^l\Big) \cdot \big(\hat{c}_i + (1 - r_i^u) \cdot c_i\big). \tag{5.1}$$

Algorithm 2: QO by branch-and-bound pruning

```
1:  procedure BB_PRUNING(q, 𝒜)
2:      Construct a search tree based on ℍ from q;
3:      Q = {q_π | ∀π ∈ ℍ}; visited=∅;
4:      for each node t in the search tree:
5:          C^l, C^u ← initialize(t);
6:      while |Q| > 1:
7:          t ← pop_unvisited(Q, visited);
8:          σ̂*, α* ← accuracy_allocation(t, 𝒜);
9:          update_node(t, σ̂*, α*);
10:         visited = visited ∪ {t};
11:         sort_and_prune(Q, ∑C^l, ∑C^u);
12:     return (π, α) that minimizes ∑C.
```

An upper bound is

$$\Big(\prod_{j=1}^{i-1} s_j^u \cdot \alpha_j^u\Big) \cdot \big(\hat{c}_i + (1 - r_i^l) \cdot c_i\big). \tag{5.2}$$

Example. In Figure 6, the lower bound of node 1 is the cost of applying a proxy model. $C_X^l = \hat{c}_X$ using Expression 5.1 with $\alpha_X^l = \mathscr{A}$, $s_X^l = 0$, and $r_X^u = 1$. The upper bound C_X^u is the cost of a proxy model $\hat{c}_X$ plus that of the ML UDF c_X with $\alpha_X^u = 1$, $s_X^u = 1$, and $r_X^l = 0$. For the plan XYZ in Figure 6, the lower bound of the plan is $C_X^l + C_Y^l + C_Z^l$, and the upper bound is $C_X^u + C_Y^u + C_Z^u$.

5.2 Branch-and-bound Search

We present a general pruning framework in Algorithm 2. Its main idea is that the upper and lower bounds can be improved as we collect information during the search process, such as selectivity and reduction. The search builds necessary proxy models and prunes the search tree to reduce the optimization overhead. For each node t, according to Lemma 2, we initialize the lower and upper bounds of $\hat{\sigma}$ using C^l and C^u, respectively (lines 4~ 5). We then progressively build proxy models (lines 6~11). For each search step, we find optimal α parameters for t and prefix nodes using Algorithm 1. We compute the cost $\sum C$ of these nodes after using Algorithm 1, and tighten the bounds of costs for t's leaf nodes. The search yields an order π that minimizes the overall cost $\sum_i C(\hat{\sigma}_{\pi_i}, \alpha_{\pi_i})$. We next explain several specific functions used in the algorithm.

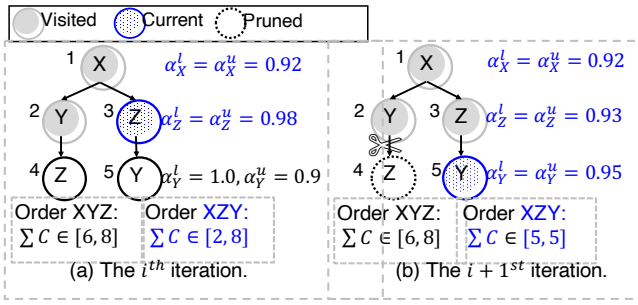


Figure 6: Two iterations in branch-and-bound search on a tree starting from node 1 with $\mathscr{A} = 0.9$. The blue text is updated information such as accuracies, lower bounds, and upper bounds after calling the function `update_node()`.

Initialization (line 5): We initialize the lower and upper bounds for each node according to Lemma 2. The query accuracy $\prod \alpha$ in Equation 3.3 is within $[\mathscr{A}^n, 1]$. For example, for the plan XYZ in Figure 6, we initialize the lower and upper bounds for each node with $\alpha^l = \mathscr{A}$, $s^l = 0$, $r^u = 1$ and $\alpha^u = 1$, $s^u = 1$, $r^l = 0$, respectively. The query accuracy $\prod \alpha$ is within $[0.9^3, 1]$ initially, where 0.9 is the query target accuracy $\mathscr{A}$.

Choosing the next candidate node. (line 7): We find the first unvisited tree node t from π that is in the front of the queue. In Figure 6(a), $\pi = XZY$ is in the front of the queue Q according to `sort_and_prune()`, which will be explained later. `pop_unvisited()` yields $\pi = XZY$ and node 3, since node 1 has been visited. Similarly, `pop_unvisited()` yields $\pi = XZY$ and node 5 in Figure 6(b). If all the nodes for the head plan in the queue have been visited, we look for the next $\pi \in Q$.

Tightening cost bounds. (line 8∼line 9): We first call `accuracy_allocation()` to build an optimal proxy models $\hat{\sigma}^*$ with an optimal α^* from the root till the current node t at depth i. The `update_node()` function updates $\alpha^l = \alpha^u = \alpha^*$ for nodes from the root till t. Similarly, $s^l = s^u = s^*$, and $r^l = r^u = r^*$. This process improves the bounds of $\sum C$ for plans under node t (with untrained $\hat{\sigma}$s) and in turn tightens the query accuracy $\prod \alpha$ to $[\mathscr{A}^{n-i+1}, \mathscr{A}]$. In Figure 6(a), for node 3, we call `accuracy_allocation()` for the sub-query XZ and find the optimal $\alpha_X^l = \alpha_X^u = 0.92$ and $\alpha_Z^l = \alpha_Z^u = 0.98$ for node 1 and node 3, respectively. The `update_node()` tightens the query accuracy $\prod \alpha$ for the plan XZY from $[0.9^3, 1]$ to $[0.9^2, 0.9]$, and tightens the lower and upper bounds of $\sum C$ to $[2, 8]$.

Pruning plans. (line 11): After the bounds are updated, we sort and prune $\pi \in Q$. The following rules are used to determine the sort order of π as well as to prune unnecessary plans.

- When $[\sum C^l, \sum C^u]$ for two π's have overlap, the one with a lower mean cost $\frac{\sum C^l + \sum C^u}{2}$ has a higher priority and is likely to yield more gains. Such a plan should be explored first. In Figure 6(a), the mean cost for the plan XZY is 5, which is less than that of the plan XYZ. Therefore, the plan XZY has a higher priority than the plan XYZ.
- When $[\sum C^l, \sum C^u]$ for two π's have no overlap, we prune the one with a higher value range from the search tree, since it provides greater cost. In Figure 6(b), $[\sum C^l, \sum C^u]$ for the plan XZY is lower than that of the plan XYZ, and they have no overlap. Then the plan XYZ is removed from Q, i.e., the edge connecting node 2 and node 4 is deleted.

The above comparisons are done for each pair of π's until Q is fully sorted. The lower bound and upper bound are equal to the exact cost once $\hat{\sigma}$ is built. Pruned π's are removed from Q.

5.3 Improvement Using a Fine-grained Tree

The branch-and-bound search discussed above involves generating labeled samples L, followed by training classifiers M and deriving C for each node in $\mathbb{H}$. To further speedup the search, we split one node into two: an L-node to generate labeled samples, and an M-node to train classifiers M and derive R and C. An L-node has to be placed before its corresponding M-node, i.e., labeling happens before training. For instance, the node X in Figure 7(a) is split into an L_X node to generate the labeled sample for $\hat{\sigma}_X$ and an M_X node to train the classifier for $\hat{\sigma}_X$ in Figure 7(b). We call this new tree a *fine-grained* search tree $\mathbb{H}^+$.

Compared to the original search tree discussed in the previous section, $\mathbb{H}^+$ provides more opportunities to tighten the cost bounds. For example, we can prune the search tree at an L-node without executing its corresponding M-node. The search algorithm is similar to Algorithm 2, except a new `update_node()` function. Its update scheme now depends on the type of node t, discussed below.

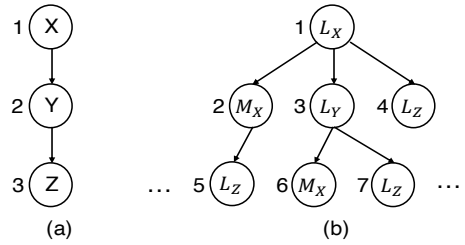


Figure 7: (a) A snippet of the search tree in Figure 6; (b) A fine-grained tree of (a).

L-node. We update the lower and upper bounds of selectivity s because we generate labeled samples and compute s at L-node. For an L-node t, a proxy model $\hat{\sigma}$ is called *available for* t if its corresponding M-node is an ancestor of t; otherwise, $\hat{\sigma}$ is called *unavailable for* t. We compute lower and upper bounds of s_t by applying all available prefix $\hat{\sigma}$ and σ on the raw input to obtain a labeled sample L_t^*, and its selectivity is denoted as s_t^*. In Figure 7(b), $\hat{\sigma}_X$ is available for node 5 because we build $\hat{\sigma}_X$ at node 2, which is an ancestor of node 5, while it is unavailable for node 3 because M_X is not an ancestor of node 3. The labeled sample L_Y^* for node 3 is labeled by σ_Y after σ_X on the raw input without applying $\hat{\sigma}_X$. Let the selectivity on L_Y^* be s_Y^*. We compute C_t^l and C_t^u as follows:

- A lower bound C_t^l can be computed when its unavailable proxy models have $\alpha^l = \mathscr{A}$ and discard records that satisfy σ_t from L_t^*. In this case, the selectivity s becomes $(s_t^* - (1 - \mathscr{A})^k) / \mathscr{A}^k$, where k is the number of unavailable prefix proxy models. This selectivity is used to estimate C_t^l using Expression 5.1. For node 3 in Figure 7(b), we compute C_Y^l using $s_Y^l = (s_Y^* - (1 - \mathscr{A}))/\mathscr{A}$ when the unavailable $\hat{\sigma}_X$ with $\alpha = \mathscr{A}$ discards records satisfying σ_Y from L_Y^*.
- An upper bound C_t^u can be computed when unavailable proxy models do not discard any records in L_t^* (i.e., $\alpha = 1.0$). Its selectivity is s_t^* in this case. We compute C_t^u using $s_t^u = s_t^*$ in Expression 5.2. In Figure 7(b), at node 3, when $\hat{\sigma}_X$ is unavailable and we use $\alpha = 1.0$, the selectivity $s_Y^u = s_Y^*$ is used to estimate C_Y^u.

M-node. As in Section 5.2, we call Algorithm 1 to compute α^*, train $\hat{\sigma}$, and estimate C. We also update the bounds for all its ancestor nodes. In Figure 7(b), after we train $\hat{\sigma}_X$ for node 5, we update the selectivity of node 3 by applying $\hat{\sigma}_X^l$ on its labeled sample L_Y'.

The above search on the fine-grained tree is efficient, as illustrated in our experiments. For a query on the Twitter dataset, the search algorithm prunes 37% of the nodes on the original search tree, and 85% of the nodes on the fine-grained tree.

6 EXPERIMENTS

6.1 Setup

Datasets. We used three datasets with text, images, and videos.

Twitter text dataset. It contained 2M tweets from January 2017 to September 2017 in the United States randomly sampled using the Twitter sampled stream API [37]. Each tweet was a string with a maximum of 140 characters. This dataset supported text analysis and retrieval by utilizing various NLP modules such as entity recognition, sentiment analysis, and part-of-speech (PoS) tagger.

COCO image dataset. COCO [28] was a public dataset collected online. It contained 123K images and 80 object classes such as "person", "bicycle", and "dog". Each image was labeled with multiple objects for their class labels and bounding box positions. The dataset was used for retrieving images that contained one or more object classes specified in user queries.

UCF101 video dataset. The UCF101 activity recognition dataset [35] contained 13K videos collected from YouTube. Each video was labeled with one of 101 action categories such as "applying lipstick" and "baby crawling". It supported video retrieval using labels generated by object detection and action recognition models.

Workloads. To our best knowledge, there is no off-the-shelf benchmark for ML inference with comprehensive ML operators and predicates. To solve the problem, we generated 10 queries for each dataset in the experiments. Table 2 illustrates some of them, and Figure 8 shows a sample workflow. The workloads retrieved texts, images, and videos that matched given query predicates, which were conjunctions of multiple clauses with different selectivity values. Each predicate clause was an equality condition on an ML-generated label column. We refer the readers to a full list of the queries as well as snapshots of the datasets in [41]. Each query also specified a target query accuracy $\mathcal{A}$, indicating how much accuracy loss the user was willing to pay relatively to the original query.

Table 2: Some of ML queries used in the experiments.

Dataset	Q#	Query semantics	Selectivity	Correlation
Twitter	q_1	Sentiment('negative' or 'neutral') & PoS Tagger('VBD' or 'WRB' or 'IN')	0.49	0.55
	q_2	Sentiment('negative' or 'neutral') & PoS Tagger('PRP')	0.35	0.41
COCO	q_6	Object detection (person) & (car or chair or cup or tv or bed or . . .)	0.13	0.99
UCF101	q_2	Activity Recognition (archery or balance beam or biking or . . .) & Object detection (chair or sports ball or bird or . . .)	0.17	1.00

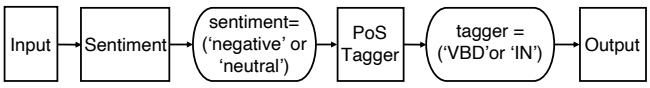


Figure 8: A sample ML workflow on the Twitter dataset.

Metrics. We measured (1) the end-to-end total processing time that included the query optimization, training of necessary models, and processing the query given an optimized plan; (2) the accuracy of our query processing relatively to the original ML inference queries; (3) the query execution cost (milliseconds per record); and (4) the decomposition of the optimization costs (minutes).

CORE. We implemented a query execution engine and the CORE optimizer in Python that enabled ML inference queries on various unstructured texts, images, and videos. We also implemented several ML UDFs using the Stanford NLP [31] and spaCy packages for text analysis, YOLOv3 [33] for object detection in images, and an activity recognition model [7] for recognizing activities in videos.

To build a proxy model, we generated the labeled sample L for $\hat{\sigma}$ by pulling initial records from the input, filtering these records by its condition d, and then labeling L using its predicate σ. L was divided into a training set, a testing set, and a validation set. We re-sampled the training data to ensure a label balance. The classifier M for $\hat{\sigma}$ was trained on the training set and the testing set using light-weight classification algorithms, such as a linear SVM [15] and a shallow NN [25]. During training, we leveraged a grid-search on the F1-score to decide the best hyper-parameters and a cross-validation to train a classifier using the set of hyper-parameters. After training M, we derived its accuracy vs. reduction curve R using the validation set.

Baselines. We compared CORE against the following baseline approaches. (i) ORIG was a baseline that ran the original query as it is. (ii) NS was a baseline based on NoScope [17]. It trained a single light-weight model and inserted it early in a plan to quickly filter input records that did not match the query predicate so that the entire query could be accelerated. (iii) PP (short for Probabilistic Predicates [30]) built a light-weight filter for each predicate offline and injected them early in a plan with an independence assumption of predicates, given an ad-hoc query. The experiments were run on a c5.4xlarge AWS instance with 280GB SSD storage, 16 vCPUs, and 32GB memory, running a Ubuntu Linux 16.04.

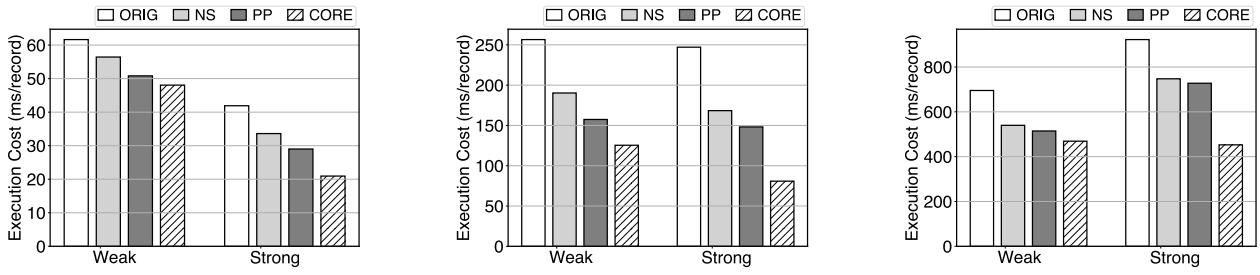
6.2 Effect of Predicate Correlation

To understand the effect of correlations of UDFs in a query, we used the three datasets and 20 test queries with two or three predicates for each dataset. These queries were divided by their correlation score $\hat{\kappa}^2$ at a cutoff score of 0.2 on the Twitter dataset, 0.9 on the COCO dataset, and 0.5 on the UCF101 dataset. As a result, each query was classified as *weakly* or *strongly* correlated among the predicates. Table 3 shows the correlation score.

We collected the execution costs of these weakly and strongly correlated queries with a query accuracy $\mathcal{A}$ = 90%. We ran these queries using ORIG, NS, PP, and CORE to generate optimal plans, and tested the execution cost of an optimal plan by executing the plan on a sample of data. Figure 9 shows the execution costs. From Figure 9, we can see that (i) NS, PP, and CORE reduced the execution cost compared to ORIG, and (ii) compared to PP, CORE reduced the execution cost more on strongly correlated queries than weakly correlated queries. In general, NS improved over ORIG using cheap filters to quickly discard irrelevant inputs, and PP further boosted the performance by decomposing the filters according to the predicate clauses. There was still room for improvements for queries with more correlations and CORE filled this gap as expected.

6.3 Time Reduction of CORE

To study the performance improvements of CORE over existing solutions, we tested the total times of strongly correlated queries with $\mathcal{A}$ = 90% on the three datasets. For query optimization to



(a) Twitter. (b) COCO. (c) UCF101.

Figure 9: Average execution costs over strongly correlated queries and weakly correlated queries on the three datasets, respectively.

Table 3: The correlation scores for 10 strongly correlated queries $q_1 \sim q_{10}$ (marked as "Strong") and 10 weakly correlated queries $q'_1 \sim q'_{10}$ (marked as "Weak") on the three datasets.

(a) Weakly correlated queries.

Dataset	q'_1	q'_2	q'_3	q'_4	q'_5	q'_6	q'_7	q'_8	q'_9	q'_{10}
Twitter	0.15	0.15	0.15	0.15	0.16	0.16	0.16	0.16	0.16	0.16
COCO	0.87	0.88	0.87	0.87	0.86	0.88	0.87	0.87	0.88	0.88
UCF101	0.40	0.40	0.40	0.40	0.41	0.41	0.41	0.41	0.41	0.41

(b) Strongly correlated queries.

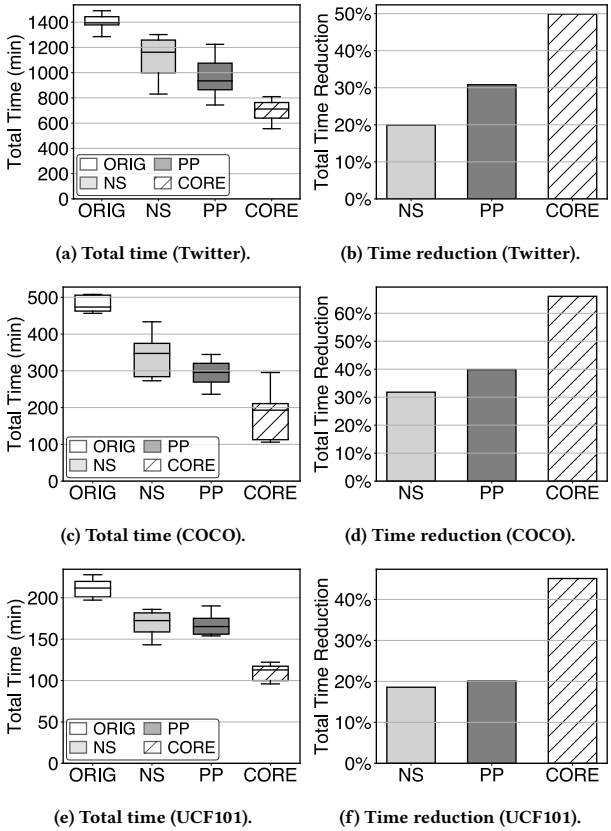
Dataset	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
Twitter	0.55	0.41	0.55	0.42	0.41	1.00	0.80	0.96	0.80	0.93
COCO	0.99	0.98	0.98	0.98	0.98	0.99	0.99	0.99	0.99	0.99
UCF101	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.82	0.82

generate an optimal query plan, we used 0.34% of the input data on the Twitter dataset, 0.84% of the input data on the COCO dataset, and 14.86% of the input on the UCF101 dataset (due to its smaller size). After generating the optimal plan, we ran it on the rest of the input. The total time included the optimization time and the time of processing all the records. We used the same setting for NS and PP, which built proxy models online.

Figures 10a, 10c and 10e show the total times of ten queries in each dataset, and Figures 10b, 10d and 10f show the average total-time reductions for the ten queries using NS, PP, and CORE compared to ORIG. We also presented the total time of each individual query in the Twitter dataset in Figure 11. These results show that CORE had a better performance than the baseline approaches in general. Specifically, CORE achieved up to a 61% reduction on the Twitter dataset compared to ORIG. For NS and PP, the reductions were about 44% and 50%, respectively. We observe similar reductions on other datasets as well. For example, on the COCO dataset, CORE had a reduction of up to 73% compared to ORIG, while NS and PP achieved a reduction of 35% and 44%, respectively. As discussed in Section 2.2, CORE achieved more gains over PP when the queries had predicates with a stronger correlation.

6.4 Optimization Cost of CORE

To better understand the detailed optimization cost of CORE, we collected the time to generate labeled samples, the time to train classifiers, and the time of search frameworks for each query. The optimizer CORE used multiple threads to label training samples. Each ML model processing unstructured texts used ten threads in parallel. The YOLOv3 model and the image feature model used two



(a) Total time (Twitter). (b) Time reduction (Twitter).

(c) Total time (COCO). (d) Time reduction (COCO).

(e) Total time (UCF101). (f) Time reduction (UCF101).

Figure 10: The total time over ten queries for each dataset using CORE and baseline approaches. $\mathbb{A} = 90\%$. For (a), (c), and (e), we show the 1^{st} and 99^{th} percentiles on the bars and 1^{st} quartile, median, and 3^{rd} quartile on the boxes. For (b), (d) and (f), we present the average total time reductions relative to ORIG.

processes in parallel, and the activity recognition model used six processes in parallel. During the phase of building proxy models, the size of labeled sample L was empirically set to 2,000. The training set, testing set, and validation set were split in a 6:2:2 ratio. We used scikit-learn to train a linear SVM classifier M on the labeled sample for text analytic queries, and used keras to train a shallow NN classifier for analytic queries on images and videos.

Table 4 shows the results of the ten queries over each dataset, including the time reduction compared to ORIG. On the Twitter

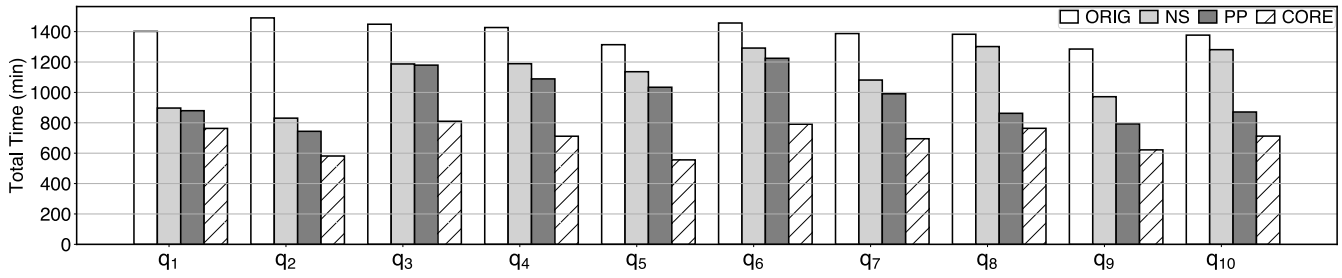


Figure 11: The total time of each query in the Twitter dataset using ORIG, NS, PP, and CORE.

dataset, the optimization time was 0.70% of the total time, and the total time reduction was 49.87% on the average. On the COCO dataset, the optimization time was 5.67% of the total time, and the total time reduction was 66.07% on the average. UCF101 was relatively smaller, and 14.86% of the data was used for optimization. The optimization time was 21.80% of the total time, and the total time reduction was 49.49% on the average. Overall, the query optimization cost of CORE was a small portion of the total processing time, and it achieved significant performance improvement compared to ORIG. When the dataset was small (e.g., the UDF101 dataset) or queries had many ML operators and predicates (e.g., q_8 and q_{10} on the Twitter dataset), the query optimization costs were larger.

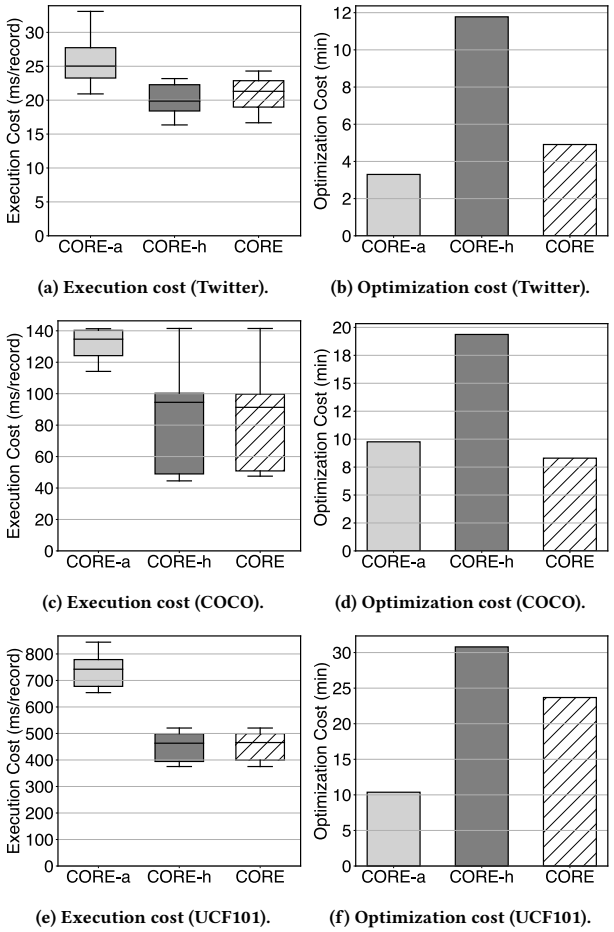
Table 4: Optimization costs and the total processing time for ten queries over each dataset using CORE with $\mathcal{A}$ = 90%. The "labeling time" is the time to generate labeled samples. The "training time" is the time to train classifiers. The "searching time" is the elapsed time for the search framework. The "QO time" is the total time of the labeling, training and searching times. The "QO Time pct." is the percentage of the QO time over the total processing time. Total Time Reduction = (ORIG-CORE)/ORIG.

Dataset	ID	#preds	Labeling Time (min)	Training Time (min)	Searching Time (min)	QO Time (min)	QO Time pct.	Total Time (min)	Total Time Reduction (%)
Twitter	q_1	2	0.93	0.10	0.17	1.20	0.16%	763	45.56
Twitter	q_2	2	1.22	0.09	0.14	1.46	0.25%	581	60.99
Twitter	q_8	3	1.53	0.75	3.28	5.58	0.73%	764	44.77
Twitter	q_{10}	3	1.76	0.75	2.93	5.47	0.77%	712	48.26
Twitter	Avg.	2.5	1.84	0.44	2.61	4.91	0.70%	700	49.87
COCO	Avg.	2	6.00	2.06	0.24	8.30	5.67%	173	66.07
UCF101	Avg.	2	23.40	0.08	0.20	23.68	21.80%	110	49.49

6.5 Effectiveness of CORE Components

CORE searched an optimal query plan in both the accuracy space $\mathbb{A}$ and the order space $\mathbb{H}$. We evaluated the effectiveness of different components in CORE using two variants, namely CORE-a and CORE-h. CORE-a represented the setting with the reordering step disabled during optimization and constrained the search space to solely $\mathbb{A}$ (Section 4). It used the input-query order and derived an optimal set of accuracy values in $\mathbb{A}$ using Algorithm 1. CORE-h applied Algorithm 1, and exhaustively searched an optimal order in $\mathbb{H}$ instead of performing the pruning in Algorithm 2.

We ran ten queries for each dataset using CORE-a, CORE-h, and CORE with $\mathcal{A}$ = 90%, and collected the execution costs for optimized plans and the average optimization costs to generate optimal plans. Figure 12 shows the results. We can see that CORE-a had the worse execution cost compared to CORE because CORE-a did not use the optimal order. CORE had similar execution costs to CORE-h, but CORE-h had much larger query optimization costs.



(a) Execution cost (Twitter). (b) Optimization cost (Twitter).

(c) Execution cost (COCO). (d) Optimization cost (COCO).

(e) Execution cost (UCF101). (f) Optimization cost (UCF101).

Figure 12: The execution costs and average optimization costs for queries over three datasets using CORE, CORE-a and CORE-h.

Table 5: Optimization costs of CORE variants on the Twitter dataset.

	Labeling Time (min)	Training Time (min)	Searching Time (min)	QO Time (min)	QO Time pct.(%)
CORE-a	1.37	0.15	1.78	3.30	0.38
CORE-h	6.51	0.57	4.69	11.78	1.74
CORE	1.84	0.44	2.61	4.91	0.70

Table 5, shows the average optimization cost including labeling, training, and searching using CORE-a, CORE-h, and CORE. We can see that CORE reduced the labeling, training and searching times compared to CORE-h. This result indicated that the branch-and-bound search algorithm in CORE successfully pruned some nodes in the tree and reduced the optimization overhead. In general, the branch-and-bound search algorithm found the optimal order. Therefore, both the Algorithm 1 for $\mathbb{A}$ and Algorithm 2 for $\mathbb{H}$ successfully accelerated the ML inference process.

6.6 Scalability

We evaluated the scalability of CORE by increasing the number of records in the Twitter dataset. We started with 0.2 million tweets and gradually increased the data size to 2 million tweets. We ran the ten queries with $\mathcal{A} = 90\%$ using ORIG, NS, PP, and CORE, and collected the total processing times at different data sizes. Figure 13 shows the average total processing time using ORIG, NS, PP, and CORE. We also presented the total times for two example queries using CORE at different data sizes. The results show that CORE scaled up well, and outperformed the other three baseline approaches at all data sizes.

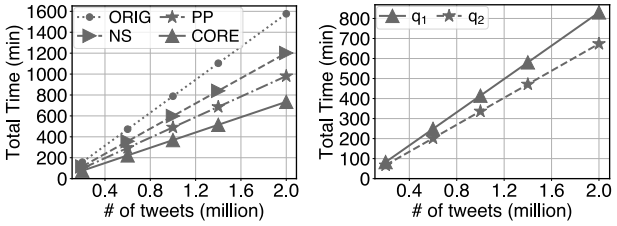


Figure 13: (Left) The average total processing time (including optimization cost) using CORE, ORIG, NS, and PP on ten queries over the Twitter dataset with different input sizes. (Right) The total times of two sample queries: q_1 and q_2, with different input sizes.

6.7 Effect of Target Query Accuracy

We evaluated the impact of the target accuracy $\mathcal{A}$ on CORE by increasing $\mathcal{A}$. We started from $\mathcal{A} = 90\%$, and linearly increased it to $\mathcal{A} = 98\%$. We collected the execution costs of optimized plans for the ten queries over the Twitter dataset using ORIG, NS, PP, and CORE with different target accuracy values. Figure 14 left shows the average execution costs for the ten queries using ORIG, NS, PP, and CORE. We also presented the execution costs for three example queries using CORE with different target accuracy values in Figure 14 right. The results indicated that CORE outperformed ORIG, NS, and PP in different accuracy settings. Moreover, the execution costs increased for all the baselines when the target accuracy increased. In addition, Table 6 shows the percentage of the query optimization time relative to the total processing time in the same setting. Similar to the observations in Section 6.3, the query optimization in CORE with different accuracy targets still had a smaller overhead relative to the total processing time.

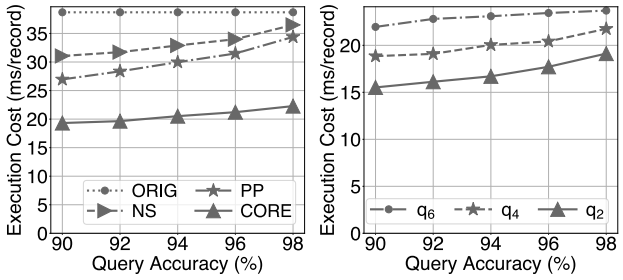


Figure 14: (Left) The average execution costs of optimized plans for ten queries over the Twitter dataset with different $\mathcal{A}$ values. (Right) The execution costs of three sample queries: q_2, q_4, and q_6, with different target accuracies.

Table 6: The optimization costs for q_2 and q_6 with different $\mathcal{A}$ values. Each cell contains the QO costs and the QO percentage relative to the total query-processing cost.

QO cost (min) / pct	$\mathcal{A} = 90\%$	$\mathcal{A} = 92\%$	$\mathcal{A} = 94\%$	$\mathcal{A} = 96\%$	$\mathcal{A} = 98\%$
q_2	1.50/0.11%	1.54/0.11%	1.50/0.11%	1.48/0.11%	1.48/0.11%
q_6	4.73/0.35%	5.28/0.39%	8.31/0.61%	6.03/0.45%	3.83/0.28%
avg.	4.57/0.36%	4.83/0.38%	5.07/0.40%	4.30/0.34%	3.24/0.25%

6.8 Effect of Sample Size Used in Training

To better understand the effect of the labeled sample size on CORE, we varied the sample size from 1K to 5K. Table 7 shows the execution costs for two example queries and the average execution costs (in milliseconds per tuple) over the 10 queries with different sample sizes on the Twitter dataset with $\mathcal{A} = 90\%$. The results showed that the execution costs decreased and the query optimization time percentage increased when the labeled sample size increased. When we set the sample size to 500, the query accuracy $\mathcal{A} = 90\%$ could no longer be guaranteed and decreased to 82% on average.

Table 7: Execution costs of q_2 and q_3 with different sample sizes. Each cell contains an execution cost and percentage of the QO cost.

Cost / QO pct	Sample size=1K	Sample size=2K	Sample size=3K	Sample size=4K	Sample size=5K
q_2	16.8/0.1%	15.8/0.1%	16.3/0.2%	16.7/0.3%	15.9/0.4%
q_3	21.9/0.1%	21.5/0.1%	21.5/0.2%	21.0/0.3%	19.6/0.5%
avg.	20.0/0.2%	19.3/0.4%	19.2/0.9%	19.4/1.0%	19.0/1.4%

7 CONCLUSIONS

We proposed a novel query optimizer, CORE, to accelerate ML inference queries. It improved state-of-the-art techniques by relaxing the independence assumption among query predicates. CORE incurs only a small overhead by leveraging a branch-and-bound search algorithm to prune the space of candidate filters and reusing intermediate results. A thorough experimental evaluation showed that CORE significantly reduced the ML inference execution cost.

ACKNOWLEDGMENTS

This work was partially supported by the National Key R&D Program of China (No. 2020AAA0103903), the NSFC (No. 61732004), the USA NSF award IIS-2107150, and the CSC studentship.

REFERENCES

[1] Pankaj K Agarwal, Sariel Har-Peled, and Kasturi R Varadarajan. 2005. Geometric approximation via coresets. *Combinatorial and computational geometry* 52 (2005), 1–30.

[2] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive Ordering of Pipelined Stream Filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 13-18, 2004*. ACM, Paris, France, 407–418.

[3] Shaofeng Cai, Gang Chen, Beng Chin Ooi, and Jinyang Gao. 2019. Model slicing for supporting complex analytics with elastic inference cost and resource constraints. *Proceedings of the VLDB Endowment* 13, 2 (2019), 86–99.

[4] Zhaowei Cai, Mohammad J. Saberian, and Nuno Vasconcelos. 2015. Learning Complexity-Aware Cascades for Deep Pedestrian Detection. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, December 7-13, 2015*. IEEE Computer Society, Santiago, Chile, 3361–3369.

[5] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, May 14-19, 2017*. ACM, Chicago, IL, USA, 511–519.

[6] Surajit Chaudhuri and Kyuseok Shim. 1999. Optimization of Queries with User-Defined Predicates. *ACM Trans. Database Syst.* 24, 2 (1999), 177–228.

[7] Xianshun Chen. 2020. Activity Recognition. https://github.com/chen0040/keras-video-classifier. last accessed: 2020-01-22.

[8] Izrail Solomonovich Gradshteyn and Iosif Moiseevich Ryzhik. 2014. *Table of integrals, series, and products*. Academic press, Cambridge, MA.

[9] Sona Hasani, Saravanan Thirumuruganathan, Abolfazl Asudeh, Nick Koudas, and Gautam Das. 2018. Efficient construction of approximate ad-hoc ML models through materialization and reuse. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1468–1481.

[10] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, May 26-28, 1993*. ACM Press, Washington, DC, USA, 267–276.

[11] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.

[12] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, October 8-10, 2018*. USENIX Association, Carlsbad, CA, USA, 269–286.

[13] Nacim Ihaddadene and Chabane Djeraba. 2008. Real-time crowd motion analysis. In *19th International Conference on Pattern Recognition (ICPR 2008), December 8-11, 2008*. IEEE Computer Society, Tampa, Florida, USA, 1–4.

[14] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 13-18, 2004*. ACM, Paris, France, 647–658.

[15] Thorsten Joachims. 2006. Training linear SVMs in linear time. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 20-23, 2006*. ACM, Philadelphia, PA, USA, 217–226.

[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 (2019), 533–546.

[17] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *PVLDB* 10, 11 (2017), 1586–1597.

[18] Daniel Kang, Edward Gan, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Approximate Selection with Guarantees using Proxies. *Proc. VLDB Endow.* 13, 11 (2020), 1990–2003.

[19] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Task-agnostic Indexes for Deep Learning-based Queries over Unstructured Data. *CoRR* abs/2009.04540 (2020).

[20] Walter H Kohler and Kenneth Steiglitz. 1974. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM (JACM)* 21, 1 (1974), 140–156.

[21] Sanjay Krishnan, Adam Dziedzic, and Aaron J. Elmore. 2019. DeepLens: Towards a Visual Data Management System. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, Asilomar, CA, USA.

[22] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An intermediate representation for optimizing machine learning pipelines. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1553–1567.

[23] Iosif Lazaridis and Sharad Mehrotra. 2007. Optimization of multi-version expensive predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 12-14, 2007*. ACM, Beijing, China, 797–808.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[25] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Handwritten Digit Recognition with a Back-Propagation Network. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, November 27-30, 1989]*. Morgan Kaufmann, Denver, Colorado, USA, 396–404.

[26] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, and Gang Hua. 2015. A convolutional neural network cascade for face detection. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, June 7-12, 2015*. IEEE Computer Society, Boston, MA, USA, 5325–5334.

[27] Xiaoxiao Li, Ziwei Liu, Ping Luo, Chen Change Loy, and Xiaoou Tang. 2017. Not All Pixels Are Equal: Difficulty-Aware Semantic Segmentation via Deep Layer Cascade. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, July 21-26, 2017*. IEEE Computer Society, Honolulu, HI, USA, 6459–6468.

[28] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *Computer Vision - ECCV 2014 - 13th European Conference, September 6-12, 2014, Proceedings, Part V (Lecture Notes in Computer Science)*, Vol. 8693. Springer, Zurich, Switzerland, 740–755.

[29] John DC Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. 1963. An algorithm for the traveling salesman problem. *Operations research* 11, 6 (1963), 972–989.

[30] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, June 10-15, 2018*. ACM, Houston, TX, USA, 1493–1508.

[31] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, System Demonstrations*. The Association for Computer Linguistics, Baltimore, MD, USA, 55–60.

[32] Venkatesh N. Murthy, Vivek Singh, Terrence Chen, R. Manmatha, and Dorin Comaniciu. 2016. Deep Decision Network for Multi-class Image Classification. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, June 27-30, 2016*. IEEE Computer Society, Las Vegas, NV, USA, 2240–2248.

[33] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *CoRR* abs/1804.02767 (2018).

[34] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. 2017. Optimization of Complex Dataflows with User-Defined Functions. *ACM Comput. Surv.* 50, 3 (2017), 38:1–38:39.

[35] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. 2012. UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild. *CoRR* abs/1212.0402 (2012).

[36] Alexander Toshev and Christian Szegedy. 2014. DeepPose: Human Pose Estimation via Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, June 23-28, 2014*. IEEE Computer Society, Columbus, OH, USA, 1653–1660.

[37] Twitter API 2019. Twitter API. https://developer.twitter.com/en/docs/twitter-api. last accessed: 2019-01-01.

[38] Paul A. Viola and Michael J. Jones. 2001. Rapid Object Detection using a Boosted Cascade of Simple Features. In *2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2001), with CD-ROM, 8-14 December 2001*. IEEE Computer Society, Kauai, HI, USA, 511–518.

[39] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: machine learning as an analytics service system. *Proceedings of the VLDB Endowment* 12, 2 (2018), 128–140.

[40] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, Fisher Yu, and Joseph E. Gonzalez. 2018. IDK Cascades: Fast Deep Learning by Learning not to Overthink. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, August 6-10, 2018*. AUAI Press, Monterey, California, USA, 580–590.

[41] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Correlative Proxy Models. https://github.com/ZhihuiYangCS/CorrProxies/wiki/Queries-and-Datasets. last accessed: 2022-02-22.

[42] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models (Technical Report). http://texera.ics.uci.edu/pdf/proxymodel/proxymodel-tech-report.pdf. last accessed: 2022-06-09.