AIDB: a Sparsely Materialized Database for Queries using Machine Learning

Akash Mittal
University of Illinois (UIUC)
akashm3@illinois.edu

Chenghao Mo
University of Illinois (UIUC)
cmo8@illinois.edu

Jiahao Fang
University of Illinois (UIUC)
jiahaof3@illinois.edu

Timothy Dai
Stanford University
timdai@stanford.edu

Daniel Kang
University of Illinois (UIUC)
ddkang@illinois.edu

ABSTRACT

Analysts and scientists are interested in automatically analyzing the semantic contents of unstructured, non-tabular data (videos, images, text, and audio). In order to extract semantic information, analysts have turned to machine learning (ML), which can be used in unstructured data analytics systems. The most common method of using ML in analytics systems is to call them as user-defined functions (UDFs). Unfortunately, UDFs can be difficult for query optimizers to reason over. Furthermore, they can be difficult to implement and unintuitive to application users.

Instead of specifying ML models via UDFs, we instead propose specifying mappings between *virtual columns* in a structured table, where *virtual rows* are sparsely materialized via ML models. Querying sparsely materialized tables has unique challenges: even the cardinality of tables is unknown ahead of time, rendering a wide range of standard optimization techniques unusable. We propose novel optimizations for accelerating approximate and exact queries over sparsely materialized tables to address these challenges, providing speedups of up to 2-350×. Users are further able to directly query columns as in standard structured tables, removing the need to reason about opaque UDFs. We implement our techniques in AIDB and deploy them in four real-world datasets. Several of these datasets were constructed with collaborators including law professors studying court cases, showing AIDB's wide applicability.

PVLDB Reference Format:
Akash Mittal, Chenghao Mo, Jiahao Fang, Timothy Dai, and Daniel Kang. AIDB: a Sparsely Materialized Database for Queries using Machine Learning. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

In recent years, analysts and scientists are increasingly interested in analyzing unstructured data in the form of videos, images, text, and audio. Application users ranging from scientists to business analysts can query the *semantic contents* of this data to understand the real world. A traffic analyst could query video data to understand traffic patterns; a social scientist can query newspaper scans to track sentiment over historical news events; a business analyst could query retail store footage to optimize store layout.

Increasingly, these application users are leveraging machine learning (ML) to extract semantic information. For example, the urban planner could use an expensive deep neural network (DNN) such as Mask R-CNN to find object types and locations, which can subsequently be used to count cars or perform other traffic analyses. Unfortunately, these ML methods can be incredibly expensive: analyzing a small town's worth of video (100 camera-months) could cost millions in cloud compute credits [28]. Furthermore, these ML methods are difficult to implement for non-experts.

To address the cost and usability of ML-based queries, recent work has exposed these ML methods as *user-defined functions* (UDFs) in query systems, in which ML methods are called as opaque functions [11]. This body of work has also proposed many optimizations to reason about these opaque functions, such as using embeddings [27] or indexes [22] to group similar rows. Other work focuses on separate tables for ML models [34] or accelerating approximate queries over unstructured data [5, 24–26, 33].

Unfortunately, UDFs (and user-defined table functions, UDTFs) have two major drawbacks. Consider a query that uses an object detection model to extract a car's position and subsequently executes a color classification model based on this data (Figure 1). The first drawback is that it is difficult to optimize over opaque UD(T)Fs, especially those expressed in nested queries. As mentioned, full materialization over moderate-sized datasets could cost millions of dollars, so is infeasible. Second, writing queries often requires complex, nested table expressions and understanding opaque UDFs.

To address these drawbacks, we propose a novel data model, AIDM, that allows application users to directly query ML model outputs as standard SQL tables, through *virtual columns* and *virtual rows*. AIDM is enabled by the key observation that the output of ML models is *deterministic*[1] and expensive to compute. We further propose novel techniques to optimize both approximate and exact queries over AIDM schemas, despite the data not being materialized. We implement AIDM and these optimizations in AIDB, a novel, open-source system for ML-based queries.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1]The output of ML models is deterministic when the random seed is fixed, up to floating point error.

```
WITH object_detection_table AS (
  SELECT
    videoName, frameNum,
    explode(detectObjects(videoName,frameNum)) AS objects
  FROM vieo_table
), car_color_table AS (
  SELECT
    *,
    identifyCarColor(videoName,frameNum,objects.*)
      AS carColor
  FROM object_detection_table
)
SELECT * FROM car_color_table
```

(a) Example of using UDFs and UDTFs in Spark to answer a query about car colors. Executing this query requires complex nested table expressions and opaque UDFs.

```
-- color_table is sparsely materialized
SELECT * FROM color_table;
```

(b) Example answering the same query with AIDB.

Figure 1: Example of a query using Spark vs AIDB.

In AIDM, a schema consists of a table identifying the underlying unstructured data, derived tables generated by ML methods, and user-defined metadata. The only required table is a table containing a unique identifier for every unstructured data blob, e.g., the frame number in a video or a Tweet identifier. The primary key must be the blob id. Then, the data engineer can construct any number of derived tables, which take in as input any other tables' column and outputs one or more rows against a fixed schema, generated from an ML model. Concretely, consider the task of identifying cars in frames of a video and their colors. The first model may be an object detection model, which takes as input the frame id and outputs object types and positions. The second model may be a color identification model, which takes in a frame id and car position and outputs a color. Finally, the data engineer can associate metadata with any table, such as timestamps with frame ids.

The overhead for specifying the mappings via the ML methods often is as few as 10 lines of configuration code per table. Furthermore, once the data engineer defines the unstructured blob table and the derived tables, application users (e.g., data scientists) can simply specify queries against the schemas of the tables. In particular, the application user does not need to reason about opaque UDFs, allowing for much simpler queries (Figure 1).

Due to the cost of ML models, we first propose novel methods of answering arbitrary approximate linear aggregation [26] and selection queries in sparsely materialized tables (i.e., over AIDM schemas). Since each virtual row can be deterministically generated, AIDB generates query plans by sampling unstructured blob ids and re-weighting samples to obtain an unbiased estimate of the statistic of interest. We extend this method to arbitrary approximate selection queries [25]. We further propose an incremental query processing technique in which rows are incrementally generated as queries are executed and used in subsequent queries for improved performance. Finally, we show that a range of prior optimizations can easily be implemented in AIDB. These optimizations include embedding-based indexes to query processing algorithms for approximate selection and aggregation.

We further propose methods to accelerate exact queries with complex predicates. Traditional query planning executes the predicates in a fixed ordering. We show that executing the predicates by cost results in efficient queries. Instead, we can optimize the predicate ordering on a per-row basis to execute the ML models most likely to match a predicate first. We show that our optimization matches optimal static ordering without knowledge of the optimal ordering.

Finally, we propose methods for efficient physical query execution via caching and parallelization. As the mapping of the ML models is fully specified, given a set of input rows, we can assume that the output is fixed. Thus, AIDB can simply cache the output of any ML model execution, partially materializing the virtual rows. In particular, many previous systems require complex reasoning about UDFs for caching, which is challenging for systems builders to implement and for users to reason about performance. Furthermore, since the generation across rows is independent, AIDB can parallelize execution trivially.

We deployed AIDB on a wide range of applications, including social science, life science, and business applications, to show its applicability. Our workloads include those from law professors at Stanford University and the University of California, Berkeley, economists at Harvard, and urban planning. By deploying on a wide range of use cases, we have found that AIDB is both flexible enough for a wide range of scenarios and efficient.

We further evaluated AIDB on video, image, and text datasets, showing that it can handle a wide range of scenarios. We compare AIDB to existing systems for ML-based queries and show that it can answer a wider range of queries and outperform these systems by up to 2-350×.

In summary, our contributions are:

1. AIDM, a data model for ML-based queries that does not require users to reason about opaque UDFs.
2. AIDB, an open-sourced query engine for executing arbitrary AIDM queries.
3. Novel optimizations for query processing for ML-based queries, which can deliver up to 2-350× improvement over existing systems.

2 BACKGROUND

Analyzing unstructured data. In this manuscript, we refer to *unstructured data* as data where the information of interest is not natively present in a schema. This unstructured data is typically video, image, text, or audio data. As these unstructured data volumes have been growing, so has the demand for analyzing such data. It is infeasible to manually analyze this data, so practitioners are increasingly using ML.

The data systems community has been building systems and algorithms to leverage ML methods to automatically serve queries over unstructured data. These systems and algorithms include query processing algorithms (for selection [5, 25, 33], aggregation [23], limit queries [22, 23], etc.), indexes [27], execution engines [28], and others [7, 21]. Many of these optimizations focus on *approximate*

answers, as exhaustively executing ML models on all of the data can be prohibitively expensive for many applications.

To execute ML models, many of these systems expose the models via *user-defined functions* (UDFs). These UDFs are typically executed externally via some service or some custom function implementation. For example, the data engineer could call Amazon Rekognition Video's or Google Cloud Video Intelligence's API to obtain object types and positions in a video.

Although these UDFs are extremely flexible, they are difficult for both users and query optimizers to reason about performance. As the UDFs are typically externally executed, they cannot reason about the inputs. Thus, several research groups have developed techniques for optimizations over these opaque UDFs, such as indexing techniques [22, 27]. As a result, query execution costs can be difficult for end users to estimate.

Deployment patterns. Typically, the person or team that deploys the ML models is not the same as the person or team that performs the analysis. In corporate settings, typically a data engineer deploys the ML models and a data scientist performs the analysis. Similar patterns happen in academic settings. We refer to the person that sets up the ML models as the data engineer and the person that performs the analysis as the data scientist or application user.

ML models and APIs. In recent years, ML models have greatly expanded in their capabilities. In tandem, many ML models are increasingly becoming accessible (sometimes exclusively) behind APIs. For example, OpenAI's best language models are only accessible behind APIs at the time of writing. These APIs make it easier for non-experts to use ML since they do not need to set up remote servers with GPUs or install complex packages.

In addition to being simple to use, these APIs also have predictable costs. For example, all of Google Cloud Vision [18], OpenAI [37], AWS Rekognition [4], and many others have costs per example (e.g., image) or token (i.e., piece of text). The predictability of costs makes it easier for users to reason about their usage.

However, the convenience comes at a hefty price. As an example, consider Google Cloud Vision's object detection capabilities. The base price is $2.25 per 1,000 images. Computing object detection over a month of video would cost $177,400, far outside of the budget of many applications. As a result, reducing the cost of answering queries is critical.

3 USE CASES

We describe several use cases for AIDB. These use cases were inspired by collaborations with domain experts from Stanford Law School, Harvard Economics, and others.

3.1 Analyzing Newspaper Scans

Economists, political scientists, and other social scientists at Harvard and other institutions have collected over 10TB of historical newspaper scans. The social scientists are interested in answering questions pertaining to social science. As a concrete example, one question may be "how did trust in science evolve after the introduction of the polio vaccine." In order to answer such questions, the social scientists must extract information from these newspaper scans.

To do so, they first apply a layout detection model to the scans [39] to determine which parts of the newspaper contains text. This model takes as input a PDF scan (for the time being we assume a single page) and outputs a list of boxes and their content type (e.g., cartoon, headline, or article body). Then, for all of the headlines and article bodies, they apply an OCR model [35]. The OCR model takes the region of the scan and outputs the text within the image scan. Finally, they can use named-entity recognition [36] and sentiment detection [38] to find headlines and article bodies about the polio vaccine and determine its sentiment.

We show an example of a potential data model corresponding to this use case in Figure 3. As shown, these mappings can be complex and nested. They further require many ML models. The queries span multiple tables and would require multiple UD(T)Fs if implemented in the current standard method of using ML models.

3.2 Video Analytics Example

Consider an example of an urban planner studying traffic patterns. Suppose the urban planner has access to two video feeds, numbered 1 and 2, with 10,000 frames each. The urban planner is interested in studying traffic patterns across these two cameras.

The data engineer first specifies the base tables referencing the underlying video. The primary key for the base table is a composite key with the video feed identifier and the frame number.

Then, the data engineer specifies that an object detection model takes as input a unique video frame, as specified by the composite key. Given the frame, the object detection model outputs zero or more rows with the following schema: an opaque object identifier (such as an auto-increment primary key), object type (among a fixed list), xy coordinates (four floating point numbers), and the confidence (a floating point number between 0 and 1).

As a final extension, the urban planner may be interested in the make and model of cars. To classify the cars, the data engineer can register a classification model which takes as input the frame identifier, object identifier, and xy coordinates. Given these inputs, the model will output no rows if the object type is not a car and will output a single row with the predicted make and model if the object type is a car.

We show a diagram of the schema and mappings in Figure 2.

4 AIDM

To address the difficulties of querying unstructured data with UDFs, we propose AIDM, a data model for interacting with ML models via declarative queries. AIDM consists of three components: base table(s) with references to the underlying unstructured data, mappings of ML model input columns and output columns, and user-specified metadata. By fully specifying ML models through the schema, application users need not reason about opaque UDFs.

In working with our collaborators, we have found the separation of the ML model execution and the schema (from the perspective of the application user) to be critical: a data engineer can simply set up AIDM and any non-expert user can query the virtual tables. In contrast, we have found UDFs to be difficult to use.

Figure 2: Example of a data model for a traffic camera dataset. An object detection model takes a frame (associated with a blob id) and produces a list of object types and positions. Given the list of cars, a separate ML model produces the car type (e.g., sedan vs SUV) and color.

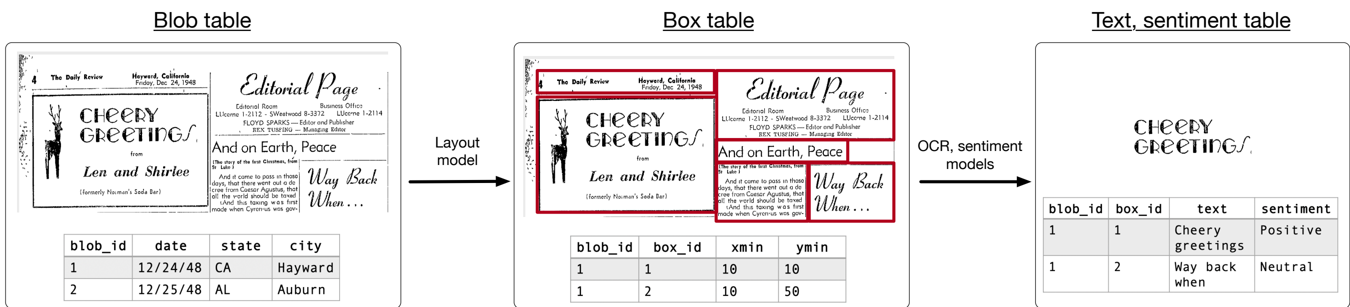

Figure 3: Example of a data model corresponding to a real-world social science analysis. Given a set of historical scans of newspapers, we can use a layout detection model that segments the pages, producing a list of boxes and types (e.g., image, text). Then, we can use an OCR model along with a sentiment detection model to determine the contents of the boxes.

We now describe the full specification of AIDM. As mentioned, AIDM has three components: the base table(s) referencing the underlying unstructured data, the mappings between ML models, and user-specified metadata. We describe each in turn.

Base tables. AIDM's first component is base tables that provide identifiers for the underlying unstructured data. Each base table contains a primary key that references an unstructured data blob. The base tables are fully materialized. For example, the base table in the video example has a composite primary key that references the camera id and the frame id.

An AIDM schema can have more than one base table if there are multiple sources of underlying data. In the case of analyzing a video conferencing meeting, the data engineer may have one table for the screen share and one table for the participants' video.

As exemplified by the previous example, the data engineer has flexibility in deciding the base tables' schemas. The analyst may wish to treat all videos identically or may be interested in studying the videos in a time-synced manner. Depending on the requirements, the data engineer could also have the screen share video and participants' video be in one table.

ML model mappings. AIDM's second component is the mappings between ML model inputs and outputs. In AIDM, every ML model can have one or more columns (possibly across tables) as input. The output is one or more columns (also possibly across tables). Every row as input produces zero or more rows as output.

If an ML model has input that has columns spanning multiple tables, AIDM has the following constraints to ensure that inputs for ML models can be deterministically generated. The input set of tables must have primary-foreign key relationships and the input set of columns must contain these keys.

AIDM also requires that the mappings between columns be acyclical. In particular, construct a graph $\mathcal{G}$ with a directed edge from column c_i to c_j if there is an ML model mapping with input c_i and output c_j. AIDM requires that $\mathcal{G}$ be a directed, acyclic graph (DAG).

Finally, AIDM requires that every generated column have a parent and requires that every column with no parents be fully materialized. The columns with no parents are typically base table columns.

In the urban planning example, the object detection model maps the frames to objects. As a frame may contain no object or many objects, zero or more rows can be generated. The car make/model classification model maps the frame and object to the prediction of the make/model.

User-defined metadata. AIDM's final component is user-defined metadata columns and tables. In many cases, the application user is interested in analyzing the semantic contents of the unstructured

data in conjunction with metadata, such as timestamps. As a result, AIDM allows every table to have additional columns with user-defined metadata and additional tables that are user-defined.

This metadata is often application specific. As mentioned, timestamps are a common piece of metadata. Other metadata could include the object types of an object detection model or metadata about video conferencing participants.

We have implemented a prototype system, AIDB, for answering AIDM queries. In this work, we focus on and develop novel algorithms for efficient aggregation queries but also develop optimizations for exact queries. We describe the system design and novel algorithms below.

5 AIDB OVERVIEW

We have created AIDB, an open-source system for implementing and executing AIDM queries. AIDB supports both exact queries and approximate queries (which are of interest due to the cost of ML models).

AIDB consists of many of the same components a traditional query engine contains, including a query parser, optimizer, and execution engine. However, the cost of ML models necessitates changes in the architecture, particularly for approximate queries. One major difference of note is that AIDB *contains a structured query engine* as part of its architecture.

AIDB is implemented in Python for ease of integration with ML model frameworks. AIDB also provides a command line interface for application users to use standard SQL for specifying queries.

In the remainder of this section, we describe common features for executing exact and approximate queries in AIDB. We further describe how to execute exact queries in AIDB. In the next several sections, we describe AIDB's novel optimizations.

5.1 AIDB Architecture

As mentioned, AIDB contains many of the same architectural components a standard query engine contains, including a query parser, optimizer, and execution engine. Several components are similar to standard structured query engines, such as query parsing. However, due to the cost of ML model execution, AIDB aggressively caches ML model outputs; we describe the caching algorithm below. This caching causes several architectural differences, which we highlight below.

First, AIDB contains a structured query engine for its caching layer. As part of query execution, AIDB will typically *execute a structured query* against the structured query engine as its first step. To understand why, consider a limit query searching for 10 common events. If AIDB has executed previous queries that match the predicate, AIDB can directly return cached results for the limit query. We describe AIDB's caching and query execution below.

Second, AIDB will execute ML models by calling external functions. To do so, AIDB provides an API for arbitrary ML models. Data engineers must provide the callback or external service. We describe the API below.

5.2 Specifying AIDM Schemas

To specify AIDM schemas, AIDB accepts standard configuration formats (YAML) [9]. The data engineer must specify:

(1) Table schemas and column types,
(2) Input and output columns for the ML models,
(3) Whether or not the ML model is a one-to-one mapping or one-to-many mapping, and
(4) ML model execution logic

in the specification. For all base tables, the data engineer must provide the full table (i.e., the universe of unstructured data blob ids).

In order to specify the ML model mapping and execution, the data engineer must specify the input/output columns for ML models and the ML model execution logic. The blob id is a required input to all ML models. To specify the ML model execution logic, the data engineer must either implement a Python or REST API. The Python API takes as input a Pandas dataframe with the input columns and their values and outputs the output columns and their values. The REST API takes the same inputs and outputs but as JSON instead.

Finally, we optionally allow the user to specify cost functions and preferred batch sizes for models. These costs can either be the cost of the API call (e.g., the cost of calling a Google Cloud Vision model) or the estimated cost of a self-hosted model (e.g., the GPU cost). These costs are used to provide cost estimates to users.

5.3 Interface for ML Models

There is a wide range of ML models that users of AIDB will be interested in deploying. These include API-gated models (e.g., GPT-3, Google Cloud Vision) and self-hosted models (e.g., BERT, Mask R-CNN). AIDB aims to support a wide range of models efficiently.

In order to support this wide range of models, AIDB supports two interfaces: a direct Python interface and a REST interface. In both cases, the data engineer must implement a wrapper that takes blob ids and any derived rows/columns and returns a dataframe with the results conforming to the schema.

Each ML model in our real-world deployments requires only a few lines of code (LOC), with some as little as 37 LOC. These include both external APIs and self-hosted models.

5.4 Naive Generation of AIDM Rows

Given an AIDM specification, we describe how a system could materialize the rows against the schema. Recall that the graph $\mathcal{G}$ specifies the relationships between the columns. The graph $\mathcal{G}$ has nodes c_i which represent the columns in the schema. We further associate every edge e_j with an ML model m_k.

Suppose the system wishes to materialize rows against a set of columns $C = \{c_i\}$. The first step is to recursively find all parents of $c_i \in C$. Denote the parents and the columns to materialize as C'. The next step is to collect the minimal set of edges between C' and the ML models associated with these edges, $\mathcal{M} = \{m_k\}$. Then, for every base table row, we execute the ML models in order of a topological sort of the columns C' by walking down the edges.

In the next two sections, we describe how to optimize query execution for both approximate and exact queries in AIDB.

6 EXECUTING APPROXIMATE QUERIES

In addition to answering exact queries, we provide novel algorithms for answering approximate queries. In contrast to many structured approximate query systems, the rows are not materialized ahead of time (see Section 9 for an extended discussion). Furthermore, in contrast to many unstructured approximate query systems that answer specific queries [5, 23, 24, 26, 33], AIDB is designed to answer arbitrary approximate linear aggregation[2] [26] and selection queries [25].

Because AIDM rows are not materialized ahead of time, standard techniques for pre-computing statistics or summaries cannot be used. As such, AIDB must decide how to sample without this pre-computed information.

One possible solution would be to uniformly sample from the base table records and compute the average from the sampled rows. However, as we describe, standard forms of uniform sampling can result in *arbitrarily bad query results.*

In the remainder of this section, we describe why uniform sampling fails and our novel stratified sampling algorithms for answering AIDM queries. We further describe how to incorporate prior techniques into AIDB.

6.1 Naive Uniform Sampling Gives Incorrect Results

To understand the intuition behind why various forms of uniform sampling would give incorrect results, we first highlight the semantics behind approximate queries in AIDB. There is a range of error semantics for approximate queries, ranging from best-effort to confidence intervals with guarantees of validity.

For the settings we consider, where scientific inferences or business decisions are made based on the results of queries, it is critical to have *confidence intervals with guarantees of validity.* In particular, best-effort systems can return query results that are arbitrarily far from the true answer.

In order for uniform sampling to give confidence intervals, the uniform sample must be drawn from the correct distribution. Consider a simple query with two blob ids (1 and 2) where blob id 1 has one derived row with value 0 and blob id 2 has two derived rows of values 3 and 6 respectively. The correct average of the derived rows is 3, but if we placed uniform weights on blob ids 1 and 2, this would result in 2.25. As we can see from this example, the sampling bias is unknown when performing naive uniform sampling from blob ids.

We now formally describe why two forms of uniform sampling do not draw samples from the correct distribution.

Setting. Consider the setting of computing the mean of a single, derived column $C = \{C_1, ..., C_n\}$. Formally, we wish to compute $\mathbb{E}[C] = \frac{1}{N}\sum C_i$. Further consider the blob id column $B = \{1, ..., m\}$. There is a one-to-many mapping $f : B \to C$ from the blob ids to rows in the column C. Each blob id can generate zero or many rows in C.

We consider two sampling algorithms and show that they do not achieve valid estimates of $\mathbb{E}[C]$. Although we consider the case of an average query without predicates, our analysis easily extends to the setting of predicates and other linear statistics.

Example. Before we formally prove that the naive uniform sampling of blob ids gives incorrect results, we first provide an example. Consider the setting with two blob ids, one that generates a single derived row with value 10 and the other that generates two derived rows with value 1. The true average is 4.

However, if we sample by blob id and average by the average of derived rows, we obtain the estimate $\frac{10+1}{2} = 5.5$. Furthermore, sampling a blob id, then sampling a randomly derived row will also fail for the same reason: the weights on the rows would be 1/2, 1/4, and 1/4 respectively (for 10, 1, and 1), giving an estimate of 5.5.

We now formally prove why uniform sampling fails.

Uniformly sampling blob ids. We first consider the strategy of uniformly sampling from blob ids, then computing all of the statistic C_i for the sampled blob id, and averaging the results.

Formally, we compute

$$\mathbb{E}_{j\sim B}\left[\frac{1}{|f(j)|}\sum_{i\in f(j)} C_i\right]$$

which is not equivalent to $\mathbb{E}[C]$. To see this, consider replacing the expectation with the exhaustive sum over B:

$$\begin{aligned}\mathbb{E}_{j\sim B}\left[\frac{1}{|f(j)|}\sum_{i\in f(j)} C_i\right] &= \frac{1}{|B|}\sum_{j\in B}\frac{1}{|f(j)|}\sum_{i\in f(j)} C_i \\ &= \sum_{i\in C}\frac{C_i}{|B|\cdot|f(j) : C_i \in f(j)|} \\ &\neq \sum \frac{C_i}{N}\end{aligned}$$

As such, uniformly sampling from blob ids does not result in valid estimates of the statistic of interest.

Uniformly sampling C_i after uniformly sampling blobs. Another sampling strategy would be to uniformly sample among the C_i after uniformly sampling a blob id. However, this has the same expectation as the prior sampling distribution, which immediately follows by observing that:

$$\mathbb{E}_{j\sim B}\left[\mathbb{E}_{i\sim f(j)} C_i\right] = \mathbb{E}_{j\sim B}\left[\frac{1}{|f(j)|}\sum_{i\in f(j)} C_i\right]$$

due to the definition of the expectation.

Discussion. The core problem with uniform sampling strategies is that we do not have access to direct access to samples from C. Naive adjustments based on only local information from the blob id are insufficient to correct for the incorrect sampling distribution. This problem motivates our stratified sampling algorithm.

6.2 Stratified Sampling

Algorithm. AIDB uses stratified sampling by default to answer approximate queries. Stratified sampling splits the samples into disjoint sets (call *strata*), computes the statistic of interest among the strata, and combines the results [29].

[2] Linear aggregation queries are aggregation queries where the result is a linear combination of the statistics, which includes sums, counts, and averages. It does not include max, min, or count distinct.

Algorithm 1 AIDB's sampling algorithm

```
1:  function StratifiedSampling(T)
2:      S = {∅ : k = 1, ..., K}
3:      while Budget T not exhausted do
4:          B_i ← SampleBlob(B)
5:          s ← |f(B_i)|
6:          C_j ← SampleColumn(B_i)
7:          S_k = S_k ∪ {C_j}
8:      U = Σ_k |S_k|
9:      Estimate = (1/U) Σ_k |S_k| Ŝ_k
10:     return Estimate
```

Denote the strata as S_k, so that $C = \cup_k S_k$. Standard stratified sampling requires knowledge of the strata sizes and computes

$$\mathbb{E}[C] = \sum_k \frac{|S_k|}{|C|} \mathbb{E}[C_i : i \in S_k].$$

In our setting, we do not have access to direct samples from C_i (or underlying statistics about C_i, such as $|C|$), which prevents us from using standard stratified sampling.

Instead, we construct a novel stratified sampling algorithm. AIDB's stratified sampling algorithm works by sampling random blob ids, then random C_i from the blob ids. After exhausting its budget, AIDB will stratify C_i such that $S_k = \{C_i : |f(B_j)| = k\}$. Denote p_k to be the true proportion of each stratum and $\hat{p}_k$ to be the plug-in estimator. AIDB will then return the estimate

$$\sum_k k \cdot \hat{p}_k \sum_{i \in S_k} \frac{C_i}{|S_k|}.$$

Importantly, the strata are weighted appropriately, which gives unbiased estimates (assuming samples from every stratum). We present a formal algorithm in Algorithm 1.

Intuitively, AIDB's stratified sampling algorithm works by naturally grouping relevant records by the number of derived records. This allows us to draw C_i uniformly at random *within* each strata, *without* knowing the strata ahead of time.

Validity. Proving the validity of AIDB's estimator requires one additional assumption: that each blob id generates at most K derived rows. To understand why this assumption is necessary, suppose there were 1 billion blob ids, where every blob id generated one derived row except one blob id which generated 1 trillion rows. Then, if the single blob id is missed, the estimate can be arbitrarily far off.

Given our assumption, proving the validity of AIDB's estimator requires two parts: proving that the bias of AIDB is small (from the probability of missing a stratum) and that AIDB's estimates for individual strata are unbiased.

We formalize the validity statement with the following theorem. Our theorem *controls* the bias of AIDB's estimator, which is in contrast to uniform sampling, which can give arbitrarily far estimates.

Theorem 6.1. *The bias of AIDB's estimator decays exponentially fast in the number of samples n:*

$$\mathbb{E}[\hat{C} - C] \leq K \cdot \max_i |C_i| \cdot \left(1 - \min_k p_k\right)^n$$

where $p_k = \frac{|S_k|}{|C|}$.

Proof. We first show that AIDB's per-strata estimator is unbiased conditional on drawing samples. Namely,

$$\mathbb{E}[\hat{C}_k - C_k | |\hat{S}_k| \geq 1] = 0.$$

This follows immediately since the probability of drawing a blob id within S_k is uniform and the probability of drawing a sample C_i given a blob id is also uniform.

We can then decompose the bias of AIDB's estimator as follows:

$$\begin{aligned}
\mathbb{E}[\hat{C} - C] &= \sum_k \mathbb{E}[\hat{p}_k \hat{C}_k - p_k C_k] \\
&= \sum_k \mathbb{E}[\hat{p}_k \hat{C}_k - p_k C_k | |\hat{S}_k| \geq 1] P(|\hat{S}_k| \geq 1) \\
&\quad + \mathbb{E}[\hat{p}_k \hat{C}_k - p_k C_k | |\hat{S}_k| = 0] P(|\hat{S}_k| = 0) \\
&= \sum_k \mathbb{E}[\hat{p}_k \hat{C}_k - p_k C_k | |\hat{S}_k| = 0] P(|\hat{S}_k| = 0)
\end{aligned}$$

where the last line follows because the per-strata estimator is unbiased. We can then bound the remainder term by considering each term:

$$\begin{aligned}
\mathbb{E}[\hat{p}_k \hat{C}_k - p_k C_k | |\hat{S}_k| = 0] P(|\hat{S}_k| = 0) &\leq \max_i |C_i| P(|\hat{S}_k| = 0) \\
&= \max_i |C_i| \, (1 - p_k)^n
\end{aligned}$$

The theorem follows from summing the k terms. □

Computing confidence intervals. As we have shown, the bias in AIDB's estimator decreases exponentially with the number of samples. However, we must still compute valid confidence intervals for scientific and high-stakes business decisions.

Since AIDB's estimator is biased, we cannot directly use standard confidence interval methods. Instead, AIDB will compute two terms: an upper bound on the bias and an upper bound on the error from the variance.

To compute the upper bound on the bias, AIDB can form an estimate of $\min_k p_k$ from the samples drawn using standard multinomial confidence intervals. To compute the error from the variance, AIDB can use standard tools for unbiased estimators, such as sub-Gaussian approximations or the bootstrap.

Accounting for queries with predicates. Our analysis above shows an optimized query execution procedure for queries without predicates. To extend AIDB's procedure to queries with predicates, we can first perform rejection sampling, where we discard samples that do not match the predicates. Our analysis directly follows from standard tools of rejection sampling.

6.3 Approximate Selection

Setting and challenge. We further propose methods of performing arbitrary approximate selection queries over AIDM schemas. Approximate selection queries (SUPG queries) were proposed by Kang et al. [25], in which the query returns a set of records that match the predicate with a specified recall or precision target. For brevity, we describe how to extend recall target queries for AIDM schemas when only the blob id is selected. Precision target queries

can be similarly implemented and selection of columns beyond the blob id is similarly simple to implement.

As in SUPG, we assume the existence of a proxy.

Extending SUPG. The core challenge behind the SUPG algorithm on AIDM schemas is that a blob id can generate many rows in a nested fashion. SUPG assumes that the blob id selected is associated with a single row. Roughly, SUPG approximates the probability that a blob id matches a predicate and uses this information to select a set of blob ids that satisfies the recall target.

In order to extend SUPG to AIDM, instead of estimating probabilities, we estimate the *number of rows* that a blob id produces that matches a predicate. To understand why this is sufficient, suppose we had a selection query that only selected blob ids and that we knew the exact number of rows generated per blob id. Then, AIDB could simply compute the cardinality of the exact query and greedily select blob ids until the recall target is met.

Because we do not know the number of records that match a predicate, we instead use our stratified sampling algorithm to estimate the following quantities: the cardinality of the result of the exact query, a lower bound on the cardinality of the set of positive records above the estimated cutoff (as in SUPG), and an upper bound on the cardinality of the set of positive records below the estimated cutoff. These quantities are sufficient to perform the confidence interval correction as SUPG uses.

6.4 Incorporating Prior Work

In addition to its stratified sampling algorithm, AIDB can incorporate optimizations from prior work. As a concrete example, consider TASTI [27], which is an index for unstructured data. TASTI generates proxy scores for downstream algorithms, such as ABae [26] or SUPG [25]. TASTI does this by grouping records that are semantically similar and labeling a small number of the records as cluster representatives.

To use TASTI in AIDB, we could create an auxiliary table with the TASTI groupings as a metadata table. This table would store the closest cluster representatives and their distances for every unstructured data blob. Then, downstream query processing could use these groupings to generate proxy scores via the TASTI algorithm. Finally, these proxy scores can be used for accelerated approximate queries.

Finally, AIDB's query processing system is flexible enough to accommodate other methods of sampling. As concrete examples, we have implemented BlazeIt's aggregation algorithm and the SUPG approximate selection algorithm in AIDB.

7 EXECUTING EXACT QUERIES

Given an AIDM specification, we describe how AIDB answers queries that require exact answers. We first describe an overview of how AIDB can answer exact queries using filtered full scans, which requires materializing any unmaterialized rows necessary for the query. We then describe novel optimizations in AIDB, including novel caching and cost estimation algorithms. We conclude this section by describing how AIDB optimizes exact queries, optimizing either for cost or latency.

To understand why new methods of caching and cost estimation are necessary, we re-emphasize two points about real-world analytics with ML. First, ML models can be incredibly expensive: invoking a single ML model on a single frame of video can be as expensive as a structured aggregation query over billions of rows. Second, due to the costs of executing ML models, many systems do not fully materialize all of the rows (in AIDM) or UDFs (in prior systems).

As a result, all rows generated from ML models in AIDM are initially virtual, until materialized. In order to reduce the cost of query execution, it is critical that AIDB cache results from ML model execution. Furthermore, the set of rows materialized for any given query will change as new queries are executed. Thus, cost estimation must take into account which rows are materialized for accurate estimates.

Given AIDB's novel caching and cost estimation procedures, we describe how AIDB optimizes query execution for exact queries. Intuitively, AIDB aims to use as many cached (i.e., materialized) rows as possible, before selecting the cheapest materialization plan on the remaining unmaterialized rows. Furthermore, AIDB can parallelize execution to reduce latency.

7.1 Full Scans with Filtering

As we have described in Section 5, AIDB can answer any query by first fully materializing all rows and using a standard DBMS to answer the query. However, this is inefficient and too expensive for many applications.

Recall that, to answer exact queries in an unoptimized way, AIDB can perform a full scan to materialize rows until the query is answered. To do so, denote the set of columns (across tables) that a given query, Q, touches to be $C = \{c_i\}$.

To optimize the full scan, AIDB can perform filtering based on metadata and blob ids. Because metadata and blob ids are fully materialized, AIDB will first execute a structured query to select these records and filter them based on any predicates in the query. Then, for the remaining columns, AIDB will perform a full table scan based on the row generation procedure above until the query is satisfied (i.e., AIDB will terminate early for limit queries).

As an example, consider the following query for the urban planning use case:

```
SELECT   c.color, COUNT(c.blob_id)
FROM     Colors c JOIN Blobs b
ON       c.blob_id = b.blob_id
WHERE    b.timestamp > 10AM and b.timestamp < 12PM
GROUP BY c.blob_id
```

In this query, the user counts the number of cars of a specific color per frame from 10 AM to 12 PM (this query is used for illustrative purposes).

The color query accesses three columns: the blob id, the timestamp, and the color. The blob id and timestamp are fully materialized, so AIDB can filter by timestamp before materializing the color column. Since the color column has a dependency on the Box table, AIDB must first fully materialize the relevant rows in the Box table.

7.2 Optimized Exact Queries

To further optimize exact queries, we implemented several other optimizations in AIDB. Our first optimization is to cache the results of ML model execution for reuse in subsequent queries. Our second optimization is to *order* the execution of ML models to reduce the

cost as much as possible. In particular, we perform the ordering of ML models on a *per-blob id basis*, which is in contrast to standard structured data query plans which decides at the query level. As we describe, this can have dramatic effects on overall query performance.

We now describe our caching algorithm and how to order ML model execution on a per-row basis.

7.2.1 Caching. As mentioned, prior work exposes ML model outputs as UDFs. Unfortunately, as UDFs are opaque functions, they are difficult to reason about. For example, UDFs in general may not be deterministic.

In contrast, AIDM specifications contain the exact relationship between input columns and output columns. Because ML models are deterministic up to floating point error,[3] AIDB can simply cache the output of ML models as materialized rows. However, since ML models can potentially output zero rows, AIDB must mark which input rows have been processed by which ML models. AIDB uses an auxiliary table per ML model to do so, where the table columns are the ML model input columns along with a Boolean column specifying if the ML model has been executed over the input row.

The overhead for caching is a boolean value per ML model execution. We have found that the ML model execution far dominates the storage overhead in every use case we consider.

7.2.2 Ordering ML Model Execution. We now describe the necessity to order ML model execution and how AIDB orders ML model execution.

Necessity of per-row ordering. Another critical decision AIDB must make is the order in which the ML models are executed. In standard structured query processing systems, the ordering is often decided at the *query* level. This is done because it is not worth the overhead of per-row decisions for structured data processing. However, ML models are typically expensive enough for the overhead to be worth the cost.

To understand the relative costs, consider the cost of using a cached result compared to executing a typical ML model (e.g., to materialize another column). Fetching a cached result may take 50ms of compute time, which would cost $0.00000101388 using a standard AWS RDS instance (`db.t4g.medium`). In contrast, using Google Cloud Vision API for object detection would cost $0.00225, which is 2219× more expensive than retrieving a cached row. Even within ML models, the costs can vary by an order of magnitude or more: OpenAI's most powerful language model (`davinci`) is 50× more expensive than their cheapest language model (`ada`) at the time of writing.

In addition to considering costs, AIDB must consider selectivity as well. Suppose we have two predicates based on models A and B with selectivities of 100% and 50% respectively (which are unknown). Even if A is 100× cheaper than B, it would be optimal to run B first.

Optimization procedure. In order to optimize queries, AIDB will construct cost estimates, generate per-row selectivity estimates, and will then execute the query with cached results.

[3]Some models have stochastic execution which can be made deterministic by fixing the random seed.

The first step is to obtain cost estimates for executing ML models. If the user provides the costs (e.g., pricing of external APIs), AIDB will use these costs. Otherwise, AIDB will profile the ML model execution time. This step only needs to be done once. Although ML model execution costs can vary (e.g., with text sequence length), we have found that using average execution costs is sufficient for our queries. This is largely because the cost differentials between models are much higher than the cost differentials between data.

In order to account for caching, AIDB approximates the cost of retrieving from the cache as zero. As we have shown above, this heuristic is often close enough to reality.

The next step is to generate per-row selectivity estimates for the predicates. In order to do so, AIDB requires access to proxy scores [25]. AIDB will convert the condition in the query to conjunctive normal form (CNF). For each row (in the flattened schema), AIDB can compute the estimated cost and selectivity for every clause in the CNF. AIDB will order the ML model execution by the cost of the CNF clauses, where the cost is defined by the selectivity multiplied by the ML model costs. It will also update the cost of the formulas as the ML models are executed.

7.3 Efficient Physical Query Plans

Given an optimized logical query plan, AIDB must efficiently execute these queries. The overwhelming cost of query execution in AIDB is the ML models. Thus, we primarily focus on efficient ML model execution.

For API-gated ML models, the costs are often fixed. Reducing the total number of API calls results in the cheapest queries.

However, for self-hosted ML models, this is often not the case. One major performance consideration for self-hosted ML models is batching. The aggregate throughput (and therefore cost) of ML model execution scales down poorly: the throughput of optimized ResNet-50 execution can be 4× higher at batch size 32 compared to batch size 1 [40].

Thus, to ensure high-performance query execution, AIDB must batch ML model execution efficiently. To do so, AIDB leverages the preferred batch size that is specified in the configuration.

Since AIDB executes the computational DAG per blob id, it batches by collating inputs to an ML model grouped as closely by the blob id as possible. For example, consider a simple DAG where model A is executed on the blob id and always generates two rows that model B executes. Suppose the batch size is 32 for both models. AIDB will execute A on 32 blob ids at a time. It will collect the results for every 16 blob ids to execute model B. In general, this process is done at query execution time.

7.4 Cost Estimation for Exact Queries

A critical component in AIDB is cost estimation. Many application users, particularly scientists, have limited budgets. It is critical to ensure that queries are not executed outside of their budgets.

At first glance, cost estimation for queries with ML models may seem simple, as ML models have predictable costs. In fact, ML model serving systems that are agnostic to queries leverage this consistency to achieve tight SLAs [19].

Unfortunately, cost estimation for *queries* with ML models is not straightforward due to unmaterialized rows. Consider the urban

planning example: the color column has a variable number of rows generated per blob id since there are a variable number of cars per frame. Furthermore, a query may have filters on upstream columns, e.g., a query that only queries cars in the left-hand side of the frame need not materialize all rows in the color table. Finally, the cached rows can dramatically affect query costs.

AIDB performs cost estimation online for a given query. It estimates the cost of generating the derived rows and columns per blob id by randomly sampling blob ids and generating the information necessary for the query. By doing so per query, its estimates are valid and also account for cached rows. Furthermore, the estimation cost is independent of the query execution time. Under reasonable settings, it requires only sampling in the low hundreds of blob ids.

8 EVALUATION

We evaluate AIDB on four datasets, spanning text and video as modalities to show the generality of AIDB. We show that AIDB can accelerate approximate queries by up to 350× compared to prior work. We further show that AIDB's exact query execution optimizations can match optimal static ordering plans without knowledge of the optimal plans.

8.1 Experimental Setup

Datasets and models. We evaluated AIDB on four datasets:

(1) `night-street` [24], a widely studied video dataset [6, 24, 33]. We used three models: a weather detection model that classifies the weather in the scene, an object detection model that locates and classifies objects in the scene, and a color classification model that classifies cars detected by the object detection model.
(2) `tweets` [15], a dataset of tweets. We used topic detection, named entity recognition, sentiment detection, and hate detection models [10].
(3) `arxiv` [12], a dataset of arXiv papers. We used PDFMiner to perform layout extraction and OCR [1], and TextBlob to do sentiment detection [32]. The `arxiv` dataset was inspired by a similar analysis conducted by Harvard University economists.
(4) `law`, a custom dataset which we created with collaborators from Stanford and Berkeley law schools. The dataset consists of PDFs. We jointly perform layout extraction and OCR with PDFMiner [1] and perform sentiment detection with TextBlob [32].

For each model, we used the cost of a hosted API. For the vision models, we used the Google Cloud Vision API [18]. For the text models, we used the Google Natural Language API [17].

Baselines. To the best of our knowledge, the most comparable systems are using UD(T)Fs in Spark [42] and MindsDB [34]. Neither have optimizations implemented in AIDB for approximate or exact queries. As a result, they execute ML models statically over the entire database to answer queries. Due to the cost of executing ML-based queries, we emulated using MindsDB and Spark UDFs by using various static orderings of ML model execution.

```
SELECT SELECT frame, object_id
FROM car_color
WHERE color LIKE 'grayish_blue'
RECALL_TARGET 75
CONFIDENCE 95;
```

(a) Example of an approximate selection query selecting grayish blue cars in the `night-street` dataset. `car_color` is a table derived from the `objects` table, which is derived from the blob IDs.

```
SELECT AVG(sentiment)
FROM sentence_sentiments
ERROR_TARGET 5%
CONFIDENCE 95;
```

(b) Example of an approximate aggregation query selecting the average sentiment across sentences in the `law` dataset. `sentence_sentiment` is a derived table.

Figure 4: Examples of approximate queries that we use in the evaluation.

Queries. We executed a variety of exact and approximate queries with AIDB. The baselines do not support the approximate queries we consider in this work, so we compute exact results with the baselines for the approximate queries. For each optimization we consider (ML model ordering for exact queries, optimized approximate aggregation, and optimized approximate selection), we execute one query per dataset. These queries were derived from real-world workloads. We show examples in Figure 4.

Metrics. The primary metric we measure is the dollar cost of query execution. Because executing the queries multiple times is expensive, we pre-computed the answers and simulated query execution. We measure the dollar cost of the ML model execution by computing the number of ML model execution and computing the cost of using an external ML service (e.g., OpenAI, Google Cloud Vision API, etc.). We further measured the cost of running a cloud VM for the AIDB service, but this is negligible for all queries we consider in this work (<1% of the total cost).

We did not measure accuracy as all of our queries give statistical guarantees on accuracy.

8.2 AIDB Accelerates Approximate Queries

We then investigated how AIDB's optimizations for approximate queries affected query performance. There are a number of systems specialized for specific forms of approximate queries, such as approximate selection queries only on blob ids [25]. Since we have implemented these optimizations in AIDB, AIDB performs the same as these specialized systems on these specific queries. Instead, we focused on approximate queries that prior work does not support. We executed a complex approximate selection query on derived tables and aggregation queries on derived tables.

Approximate aggregation. We first investigated whether AIDB could accelerate approximate aggregation queries that prior work does not support. To do so, we executed four approximate aggregation queries on derived tables for the four datasets (one per dataset). We targeted a 5% error rate for all queries.

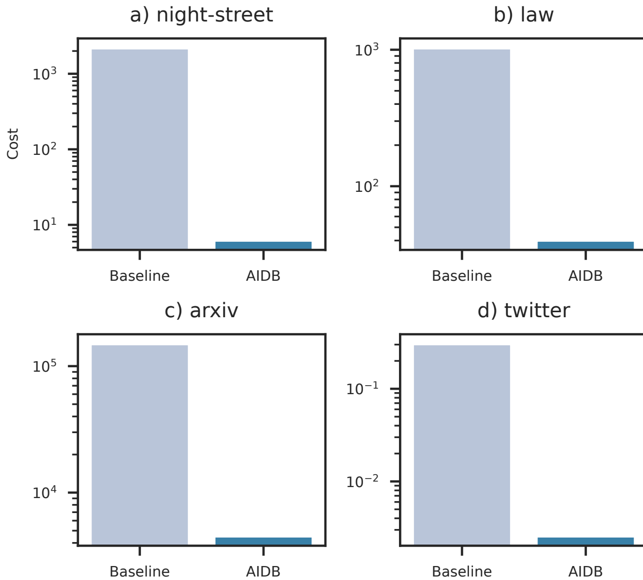

Figure 5: Dollar cost of aggregation queries for baselines and AIDB. Prior work does not support approximate aggregation queries on nested data, requiring full materialization. As shown, AIDB can achieve up to orders of magnitude improvement by supporting approximate aggregation.

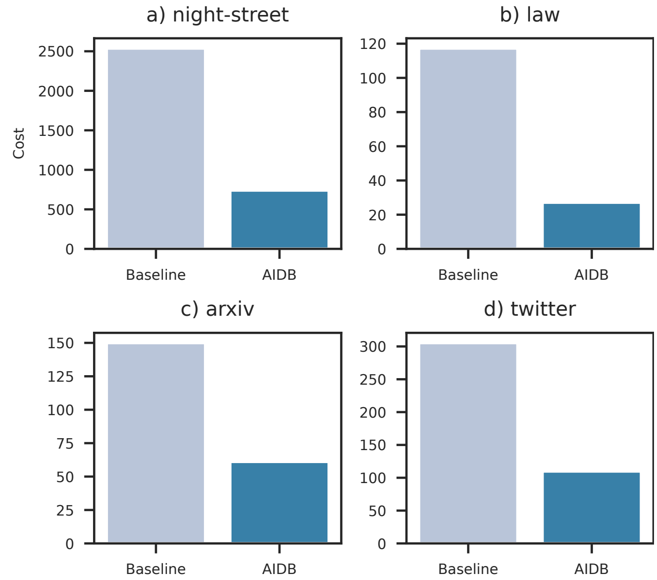

Figure 6: Dollar cost of selection queries for baselines and AIDB. Prior work does not support approximate selection queries with a recall target with nested data. As shown, AIDB can outperform baselines on recall target queries by up to 4.3×.

As shown in Figure 5, AIDB can dramatically reduce the cost of aggregation queries, producing 25-350× cheaper queries compared to exact queries. We emphasize that these queries are not supported by prior work. Furthermore, all queries achieved the confidence bound.

Approximate selection. We next investigated whether AIDB could accelerate approximate selection queries that are not supported by prior work. To do so, we executed four approximate selection queries on derived rows, where a single ML model produces multiple output rows.

As shown in Figure 6, AIDB outperforms baselines for recall target queries by up to 4.3×. Furthermore, AIDB achieves its statistical guarantees over many runs of query execution (e.g., with a 95% confidence interval, it achieves the recall target on at least 95% of the runs). As with aggregation queries, we emphasize that these queries are not supported by prior work.

8.3 AIDB Accelerates Exact Queries

We first investigated whether AIDB's optimizations for exact queries outperform static orderings. In choosing queries, we evaluated queries with predicates. To understand why, consider the simplest query of `SELECT * FROM TABLE`. This query *must* evaluate all ML models on all records of the table. Since ML model execution is the primary bottleneck, no optimization will decrease the total cost (except batching for locally executed models, but we focus on API-hosted models in this work).

As a result, we focus on queries with predicates. We executed a complex query with various selective predicates on the datasets (summarized in Table 1). We measured two static orderings of the predicate execution: as ordered by predicate cost and as ordered randomly. We compared these static orderings to AIDB's optimized query execution plans and measured the total cost of executing the ML models. We executed these queries assuming the queries were executed from scratch (i.e., with no caching beyond the construction of the TASTI index).

Dataset	Predicate
`night-street`	Red light, blue car
`law`	Positive sentiment, specific organizations
`arxiv`	Positive sentiment, early in document
`twitter`	Positive sentiment, specific organizations

Table 1: Predicates used for the exact queries.

We show the results in Figure 7. We first see that ordering the predicates by cost is not always optimal. Nonetheless, AIDB matches or outperforms the optimal static ordering plan for all queries, despite not having access to the optimal plan.

8.4 Caching Accelerates Queries

Finally, we measured the effects of caching on query performance. We first note that the exact experimental setup greatly affects the performance of subsequent queries that leverage caching. As a trivial example, suppose we executed a full scan (i.e., `SELECT *`). Then, *all* downstream queries would be cheap.

To make the comparison as fair as possible, we executed 0 to 2 aggregation queries before executing a limit query for the `night-street` dataset. We chose aggregation queries to execute

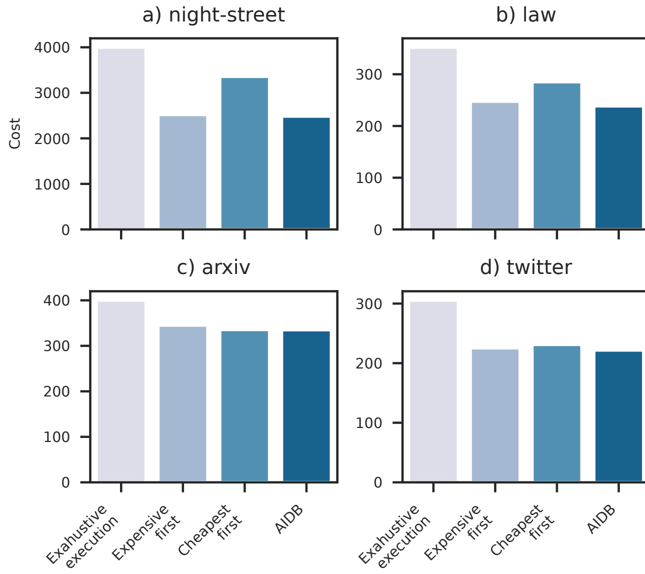

Figure 7: Exact queries with predicates for exhaustive execution, ordering the most expensive predicate first, the least expensive predicate first, and AIDB. As shown, AIDB matches or outperforms the optimal static ordering in all cases.

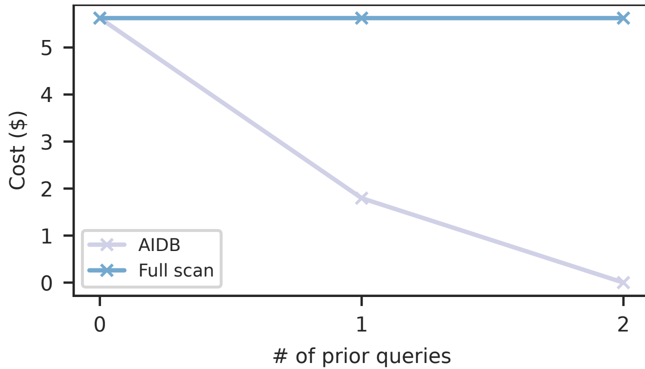

Figure 8: Cost of executing a limit query after executing 0, 1, or 2 prior aggregation queries on the `night-street` dataset. As shown, caching substantially reduces the cost of query execution.

first as they generally do not materialize a large fraction of the rows.

We show the cost of the limit query after 0 to 2 previously executed queries in Figure 8. As shown, the cost of a limit query substantially reduces with previously executed queries.

9 RELATED WORK

Structured approximate query processing. Approximate query processing techniques can be divided into online and offline techniques [31]. Offline techniques generate summaries (e.g., sketches or pre-computed samples) to accelerate approximate queries [2, 3, 16]. These techniques assume the records are already present as structured data and can compute the summaries cheaply, e.g., based on query workloads [31]. Many online techniques also rely on pre-computed information, e.g., indexes [30]. As a result, these techniques do not directly apply to the setting where the overwhelming majority of the cost is in executing expensive ML methods. In AIDB, we provide novel methods for approximate aggregation and selection queries for expensive ML-based queries. We further provide an analysis of our stratified sampling algorithm, proving convergence.

Expressing ML-based queries. As the demand for ML-based queries has increased, systems builders have created a number of systems for expressing ML-based queries. The most common method of incorporating ML models for queries is to expose them as UDFs [5, 22, 33, 34]. UDFs require users and systems to reason about opaque functions, which can be challenging. Other systems, such as `BlazeIt` [23], have custom schemas, which are inflexible. In contrast, AIDM allows data engineers to set application-specific schemas over any ML-based queries. As we show, this enables real-world applications with much less overhead compared to using UDFs.

Efficient ML execution. Several systems aim to efficiently execute ML models, such as Clipper and TFX Serving [8, 13, 41]. Other work aims to store data efficiently [14, 20] for efficient downstream query execution. These systems are agnostic to analytics needs and AIDB can leverage these systems in its query execution engine. Several systems are specific to executing efficient queries, such as Smol [28]. These can also be used in conjunction with AIDB.

Other optimizations for ML-based queries. A large body of recent work aims to optimize ML-based queries. One line of work aims to optimize specific queries, ranging from selection queries [5, 24, 33], aggregation queries [23], aggregation queries with predicate [26], limit queries [22, 23], tracking queries [7], and top-k queries [21]. This work is largely orthogonal to AIDB: these optimizations can be implemented within AIDB. In this work, we focus on queries not supported by prior work, including complex approximate aggregation and selection queries.

10 CONCLUSION

In this work, we propose AIDM, a novel method for specifying ML-based queries. We implemented AIDB to execute AIDM queries and design several novel optimizations to accelerate such queries, including stratified sampling, parallelization, and caching algorithms. To demonstrate AIDM's flexibility, we deployed AIDB on a wide range of workloads, including social science, life science, and urban planning workloads. We further evaluated AIDB, showing that it can answer a wider range of queries compared to previous systems and that it can outperform prior systems by up to 350×. We hope that AIDB will serve as a platform for future research in unstructured data analytics.

ACKNOWLEDGMENTS

This work is funded in part by the Open Philanthropy project.

REFERENCES

[1] 2023. pdfminer.six. https://github.com/pdfminer/pdfminer.six

[2] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. 1999. Aqua: A fast decision support systems using approximate query answers. In *PVLDB*. 754–757.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*. ACM, 29–42.

[4] Amazon. 2023. Amazon Rekognition - Pricing. https://aws.amazon.com/rekognition/pricing/

[5] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. 2019. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1466–1477.

[6] Michael R Anderson, Michael Cafarella, Thomas F Wenisch, and German Ros. 2019. Predicate Optimization for a Visual Analytics Database. *ICDE* (2019).

[7] Favyen Bastani and Samuel Madden. 2022. OTIF: Efficient Tracker Pre-processing over Large Video Datasets. (2022).

[8] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[9] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2009. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008* 5 (2009), 11.

[10] Jose Camacho-Collados, Kiamehr Rezaee, Talayeh Riahi, Asahi Ushio, Daniel Loureiro, Dimosthenis Antypas, Joanne Boisson, Luis Espinosa-Anke, Fangyu Liu, Eugenio Martínez-Cámara, et al. 2022. TweetNLP: Cutting-Edge Natural Language Processing for Social Media. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Abu Dhabi, U.A.E.

[11] Marshall Carter and Sujoy Dutta. 2022. Parallel ML: How Compass Built a Framework for Training Many Machine Learning Models on Databricks. https://www.databricks.com/blog/2022/07/20/parallel-ml-how-compass-built-a-framework-for-training-many-machine-learning-models-on-databricks.html

[12] Colin B. Clement, Matthew Bierbaum, Kevin P. O'Keeffe, and Alexander A. Alemi. 2019. On the Use of ArXiv as a Dataset. arXiv:1905.00075 [cs.IR]

[13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A {Low-Latency} Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

[14] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, and Magdalena Balazinska. 2021. TASM: A tile-based storage manager for video analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1775–1786.

[15] Jonathan Falvey. 2023. ChatGPT Tweets. https://huggingface.co/datasets/deberain/ChatGPT-Tweets

[16] Edward Gan, Peter Bailis, and Moses Charikar. 2020. Coopstore: Optimizing precomputed summaries for aggregation. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2174–2187.

[17] Google. 2023. Pricing | Cloud Natural Language. https://cloud.google.com/natural-language/pricing

[18] Google. 2023. Pricing | Cloud Vision API. https://cloud.google.com/vision/pricing

[19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[20] Brandon Haynes, Maureen Daum, Dong He, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2021. Vss: A storage system for video analytics. In *Proceedings of the 2021 International Conference on Management of Data*. 685–696.

[21] Dong He, Maureen Daum, Walter Cai, and Magdalena Balazinska. 2021. DeepEverest: accelerating declarative top-K queries for deep neural network interpretation. *arXiv preprint arXiv:2104.02234* (2021).

[22] Wenjia He, Michael R Anderson, Maxwell Strome, and Michael Cafarella. 2020. A method for optimizing opaque filter queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1257–1272.

[23] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *PVLDB* (2019).

[24] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *PVLDB* 10, 11 (2017), 1586–1597.

[25] Daniel Kang, Edward Gan, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Approximate Selection with Guarantees using Proxies. *PVLDB* (2020).

[26] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating Approximate Aggregation Queries with Expensive Predicates. *PVLDB* (2021).

[27] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. Semantic Indexes for Machine Learning-based Queries over Unstructured Data. *SIGMOD* (2022).

[28] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. 2021. Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics. *PVLDB* (2021).

[29] Leslie Kish. 1965. *Survey sampling*. Number 04; HN29, K5.

[30] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.

[31] Kaiyu Li and Guoliang Li. 2018. Approximate query processing: What is new and where to go? *Data Science and Engineering* 3, 4 (2018), 379–397.

[32] Steven Loria. 2018. textblob Documentation. *Release 0.15* 2 (2018).

[33] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *SIGMOD*. ACM, 1493–1508.

[34] MindsDB. [n.d.]. Machine Learning in Your Database Using SQL. https://mindsdb.com/

[35] Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada. 1999. *Optical character recognition*. John Wiley & Sons, Inc.

[36] David Nadeau and Satoshi Sekine. 2007. A survey of named entity recognition and classification. *Lingvisticae Investigationes* 30, 1 (2007), 3–26.

[37] OpenAI. 2023. Pricing - OpenAI. https://openai.com/api/pricing/

[38] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. 2002. Thumbs up? Sentiment classification using machine learning techniques. *arXiv preprint cs/0205070* (2002).

[39] Zejiang Shen, Ruochen Zhang, Melissa Dell, Benjamin Charles Germain Lee, Jacob Carlson, and Weining Li. 2021. LayoutParser: A Unified Toolkit for Deep Learning Based Document Image Analysis. *arXiv preprint arXiv:2103.15348* (2021).

[40] Geoff Tate. 2019. Do Large Batches Always Improve Neural Network Throughput? https://semiengineering.com/do-large-batches-always-improve-neural-network-throughput/

[41] Han Vanholder. 2016. Efficient inference with tensorrt. In *GPU Technology Conference*, Vol. 1. 2.

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.