AIDB: Unstructured Data Queries via Fully Virtual Tables

Daniel Kang

ABSTRACT

Analysts and scientists are increasingly interested in automatically analyzing the semantic contents of unstructured, non-tabular data (videos, images, text, and audio). In order to extract the semantic contents, analysts have turned to machine learning (ML) methods, which are most commonly used in unstructured analytics systems as user-defined functions (UDFs). Unfortunately, UDFs can be difficult to implement, unintuitive to application users, and challenging for systems to automatically optimize queries over.

Instead of specifying ML models via UDFs, we instead propose specifying mappings between *virtual columns* in a structured table, where *virtual rows* are materialized via ML models on-demand. Users are able to directly query columns as in standard structured tables, removing the need to reason about opaque UDFs. Furthermore, query engines are able to leverage standard techniques for query optimization instead of having to optimize over opaque UDFs. We implement these fully virtualized tables in a novel query engine, AIDB. We show that virtualized tables allow novel caching and sampling optimizations, which can improve query execution speeds by up to 300×.

ACM Reference Format:
Daniel Kang. 2023. AIDB: Unstructured Data Queries via Fully Virtual Tables. In *Proceedings of Conference on Innovative Data Systems Research (CIDR '23).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, analysts and scientists are increasingly interested in analyzing unstructured data in the form of videos, image, 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 the 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 [19]. 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. This body of work has also proposed a number of optimizations to reason about these opaque functions, such as using embeddings [18] or indexes [13] to group similar rows together. Other work focuses on accelerating approximate queries over unstructured data [3, 15–17, 23].

In this work, we instead propose a novel data model, DataModel, that allows application users to directly query ML model outputs as standard SQL tables, through *virtual columns* and *virtual rows*. We further propose novel caching and parallelization optimizations for ML-based queries. Finally, we show that a range of prior optimizations can be implemented under DataModel efficiently. We implement DataModel and these optimizations in AIDB, a novel, open-source system for ML-based queries. To the best of our knowledge, AIDB is the first open-sourced, fully general ML-based query engine.

In DataModel, 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 the unstructured data blob table, which contains a unique identifier for every unstructured data blob. Then, any number of tables can be derived from existing tables. These tables are generated by taking as input any other tables' column and outputs one or more rows against a fixed schema, generated from an ML model.

As an example of a DataModel schema, consider the case of video analytics. The unstructured data blob identifier might be the frame id (which has a one-to-one correspondence with timestamps). Then, the first ML model may be an object detection model, which takes as input the frame id and outputs object types and positions. The second model could be a color identification model, which takes in a frame id and car position and outputs a color. Finally, the system administrator can associate metadata with any table, such as timestamps with frame ids.

The overhead for specifying the mappings via ML methods is low: as few as 10 lines of configuration code per table. Furthermore, once the database administrator defines the unstructured blob table and the derived tables, application users can simply specify queries against the schemas of the tables. In particular, the application user does not need to reason about opaque UDFs.

Given the mappings between tables via the ML models, we propose novel caching and parallelization techniques. As mentioned, the outputs of ML models are often *deterministic*, up to floating point error. 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

CIDR '23, 2023, Amsterdam
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

trivially. We show that AIDB can leverage these techniques for up to 3× improved query execution costs after executing even a single query.

In addition to caching and parallelization optimizations, we propose a method of generating query execution plans for arbitrary approximate linear aggregation and selection queries. Since each virtual row can be deterministically generated, AIDB generate 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 [16]. We further propose a novel, 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 evaluate 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 300×.

2 BACKGROUND

In this manuscript, we refer to *unstructured data* as data where the information of interest is not natively present in a schema. This 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. Due to the sheer volumes of data, it is infeasible to manually analyze this data.

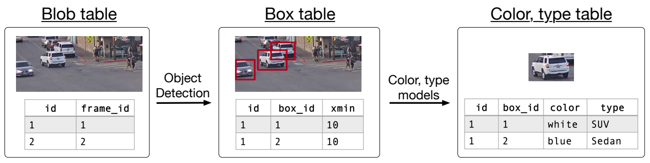
As a result, the data systems community has been building systems and algorithms to leverage ML methods to automatically serve queries over unstructured data. These systems and algorithms include query processing algorithms (for selection [3, 16, 23], aggregation [14], limit queries [13, 14], etc.), indexes [18], execution engines [19], and others [4, 11]. In particular, 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 database administrator 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.

3 DATAMODEL

To address the difficulties of querying unstructured data with UDFs, we propose DataModel, a data model for interacting with ML models via declarative queries. DataModel consists of three components: base table(s) with references to the underlying unstructured data, mappings of ML model input columns and output columns,



id	frame_id
1	1
2	2

id	box_id	xmin
1	1	10
1	2	10

id	box_id	color	type
1	1	white	SUV
1	2	blue	Sedan

Figure 1: Example of a DataModel schema for traffic analysis. The analyst can define an object detection table and a car make/color table. These tables will be populated via ML models.

and user-specified metadata. By fully specifying ML models through the schema, application users need not reason about opaque UDFs.

In the remainder of this section, we first walk through a concrete video analytics example. We then describe the exact semantics of DataModel. We conclude this section by describing how to generate rows against DataModel schemas and further examples.

3.1 Video Analytics Example

We first describe an example of an urban planner studying traffic patterns. Concretely, suppose the urban planner has access to two video feeds, numbered 1 and 2. The urban planner wishes to batch analyze the data from Monday and saves 10,000 frames from each feed, to study traffic patterns across these cameras.

The database administrator 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 database administrator 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 row 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 database administrator 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 1.

3.2 DataModel Specification

We now describe the full specification of DataModel. As mentioned, DataModel 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. DataModel's first component are base tables that provide identifiers for the underlying unstructured data. Each base table contains a primary key which 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. A DataModel schema can have more than one base table if there are multiple sources of underlying data.

ML model mappings. DataModel's second component are the mappings between ML model inputs and outputs. In DataModel, every ML model can have one or more columns (possibly across tables) as input. The output are 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, DataModel has the following constraints. The input set of tables must have primary-foreign key relationships and the input set of columns must contain these keys. These relationships are required to ensure that inputs for the ML models can be deterministically generated.

DataModel also requires that the mappings between columns be non-cyclical. 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. DataModel requires that $\mathcal{G}$ be a directed, acyclic graph (DAG).

Finally, DataModel requires that every generated column have a parent and 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. DataModel's final component are 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, DataModel 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.

Design considerations. DataModel is designed to be flexible, as different applications have different requirements. For example, consider the use case of analyzing a video conferencing meeting. The database administrator may have one table for the screenshare and one table for the participants' video. As exemplified by the previous example, the database administrator has flexibility in deciding the base tables' schemas. The database administrator could also have the screenshare video and participants' video be in one table. This flexibility is by design as various applications have different requirements. In the video conferencing example, the analyst may wish to treat all video identically or may be interested in studying the videos in a time-synced manner. The database administrator can choose the base table schemas depending on the application at hand.

4 AIDB'S OVERVIEW

We have created AIDB, an open-source system for implementing and executing DataModel 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.

4.1 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 a number of architectural differences, which we highlight below.

First, AIDB contains a structured database 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. Database administrators must provide the callback or external service. We describe the API below.

4.2 Specifying DataModel Schemas

To specify DataModel schemas, AIDB accepts standard configuration formats (YAML) [6]. The database administrator 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 database administrator 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 database administrator must specify the input/output columns for ML models and the ML model execution logic. To specify the ML model execution logic, the database administrator 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 its values. The REST API takes the same inputs and outputs but as JSON instead.

4.3 Generating DataModel Rows

Given a DataModel 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.

5 OPTIMIZED QUERY EXECUTION

Given a DataModel schema, AIDB can execute any query by first fully materializing all rows in all tables in the schema and using a standard structured query execution engine. However, this is prohibitively expensive in many applications. To address the cost of executing queries, we describe AIDB's novel optimizations for executing exact and approximate queries below.

As described above, AIDB could fully materialize all rows to execute exact queries. However, this process is inefficient for two reasons: it will re-materialize previously materialized rows and not all rows are necessary for answering queries. As such, AIDB will cache the results of ML model execution and execute all computation on structured metadata before executing ML models.

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, DataModel specifications contain the exact relationship between input columns and output columns. Since we assume ML models are deterministic, AIDB can simply cache the output of ML models as materialized rows. However, ML models can potentially output zero rows. For example, an object detection method will output zero rows for a frame of an empty street (i.e., a street with no objects). Similarly, a named entity recognition model may output zero rows for a sentence with no named entities.

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 simplicity in caching design is a result of foregoing opaque UDFs.

Execution. To execute exact queries efficiently, AIDB chooses queries plans that places all structured computation first and performs lazy execution. AIDB does in this in three ways.

First, when filtering by a structured metadata column over the unstructured blobs, AIDB will first perform this filtering operation before executing the remainder of the computation required to answer the query. For example, if a query over a video only requires information in the first 10 minutes of the video, AIDB will only materialize columns and rows associated with the first 10 minutes of the video.

Second, AIDB will first check its cache for any cardinality limited queries (i.e., queries with a `LIMIT` clause) to see if the query can be answered without further ML model execution. By doing so, AIDB can reuse work and avoid expensive materialization.

Finally, AIDB will only execute the ML models necessary to answer a given query. For example, in the traffic analysis use case, if the query does not touch the car color column, the color classification model will not be executed.

5.1 Optimized Approximate Queries

In addition to executing exact queries, AIDB supports approximate queries. In contrast to many structured approximate query systems, the rows are not materialized ahead of time. Furthermore, in contrast to many unstructured approximate query systems that answer specific queries [3, 14, 15, 17, 23], AIDB is designed to answer arbitrary approximate linear aggregation[1] [17] and selection queries [16].

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

One possible solution would be to uniformly sample from the base table records and computing the average from the sampled rows. However, *uniform sampling can be biased* for queries over derived columns. This is because ML models can generate a variable number of rows, so sampling uniformly from the base tables does not necessarily result in a uniform sample among the derived rows. For example, sampling uniformly at random from *frames* will not give a uniform sampling of *cars* in a video, as a frame can have multiple cars, when computing a statistic about cars in a video.

To address this issue, we develop novel, unbiased stratified sampling algorithms that can provide standard confidence intervals. We further describe how to incorporate prior sampling algorithms with AIDB's stratified sampling algorithms.

Stratified sampling. 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 [20].

AIDB leverages stratified sampling to produce unbiased results with valid confidence intervals. To do so, it groups the rows of interest by the unstructured blob ids that generated those rows. It then groups these groups by the number of records each blob id generates. The final set of records (per blob id with the same number of generated records) consists of the final set of strata. By groupings such records together, AIDB avoids the potential bias from sampling different numbers of records per unstructured data blob.

Incorporating prior work. In addition to its stratified sampling algorithm, AIDB can incorporate optimizations from prior work.

[1]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.

As a concrete example, consider TASTI [18], which is an index for unstructured data. TASTI generates proxy scores for downstream algorithms, such as ABAE [17] or SUPG [16]. TASTI does this by grouping records together 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.

6 EVALUATION

To evaluate AIDB, we deployed it on a standard video analytics dataset. We used the widely studied `night-street` dataset [7, 15, 25]. We show that AIDB can answer arbitrary approximate queries that prior work does not support and can accelerate exact queries with its optimized plans.

Experimental setup. We deployed two ML models on `night-street` to demonstrate the versatility of AIDB. We first deployed a state-of-the-art object detection model to extract object types and positions in the video [12]. We then deployed a model to determine car color by averaging the pixel values.

The primary metric we use is total dollar cost. AIDB executes models via external services, which are typically far more expensive than the cost to host AIDB. Furthermore, external services often do not have any latency service-level agreements [10], so latency is not a primary constraint for unstructured data queries. We use the dollar costs of the Google vision API for our experiments.

Approximate queries. We first show that AIDB can answer approximate queries that prior systems cannot answer with valid confidence intervals. We consider two queries: the average x-position of cars and the number of blue cars, both at the box level. An example query is as follows:

```
SELECT AVG(xmin)
FROM objects
WHERE object_class = 'car';
```

We compare against exhaustive materialization as prior work in approximate queries over unstructured data do not support these queries. For both queries we target an error of 5%. As shown in Figure 2, AIDB outperforms exhaustive materialization by up to 300×.

Exact queries. We also show that AIDB can effectively answer exact queries with its caching and exact query planning algorithms. To do so, we consider a sequence of queries: executing the two queries above, then executing a selective limit query searching for 10 instances of red cars.

We show the cost of the final query compared to a sequential scan without caching in Figure 3. As shown, AIDB's exact query optimization techniques can improve performance by up to 3× after running even a *single* query.

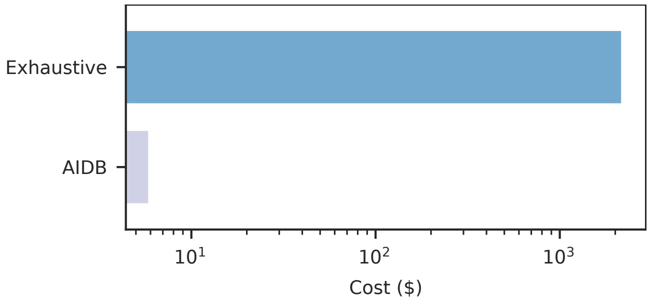


Figure 2: AIDB vs exhaustive materialization for an aggregation query. AIDB targeted and error of 5%. As shown, AIDB can outperform by up to 300× by supporting arbitrary linear aggregation queries.

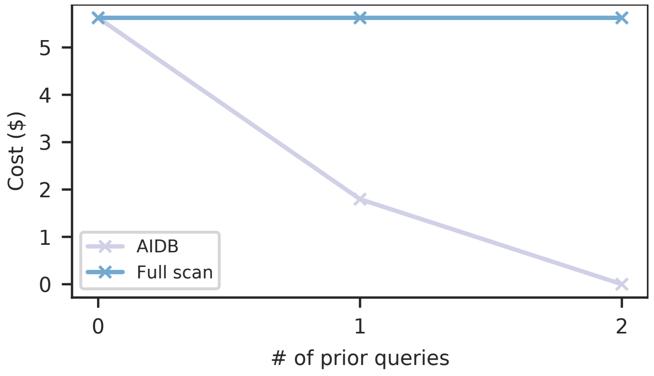


Figure 3: AIDB vs early termination with a full scan for a limit query. We show the query cost of AIDB after running zero, one, or two approximate queries first. As shown, the cost of limit queries for AIDB decreases as other queries are executed first.

7 RELATED WORK

Structured approximate query processing. Approximate query processing techniques can be divided into online and offline techniques [22]. Offline techniques generate summaries (e.g., sketches or pre-computed samples) to accelerate approximate queries [1, 2, 9]. These techniques assume the records are already present as structured data and can compute the summaries cheaply, e.g., based on query workloads [22]. Many online techniques also rely on precomputed information, e.g., indexes [21]. 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 a novel method of stratified sampling for expensive, ML-based queries.

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 [3, 13, 23]. UDFs require users and systems to reason about opaque functions, which can be challenging. Other systems, such as `BlazeIt` [14], have custom schemas, which are inflexible. In contrast, DataModel allows database administrators to set application-specific schemas over any ML-based queries.

Efficient ML execution. Several systems aim to efficiently execute ML models, such as Clipper and TFX Serving [5, 8, 24]. 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 [19]. 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 [3, 15, 23], aggregation queries [14], aggregation queries with predicate [17], limit queries [13, 14], tracking queries [4], and top-k queries [11]. As we describe in this work, these query processing algorithms can be implemented in AIDB.

8 CONCLUSION

We introduce AIDB to improve the usability and performance of unstructured data queries. As we have shown, AIDB's virtual tables provide simple ways for users to write standard SQL to query unstructured data. To execute queries efficiently, we provide novel caching and approximate query processing techniques. Using these techniques, AIDB can improve query performance by up to 300×. These results show the promise of AIDB as a platform for optimizing unstructured data queries.

REFERENCES

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David Andersen, Michael Kaminsky, and Subramanya Dulloor. 2019. Scaling Video Analytics on Constrained Edge Nodes. *SysML* (2019).

[8] 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.

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

[10] Google. 2018. Cloud Vision Service Level Agreement. (2018). https://cloud.google.com/vision/sla-20181119

[11] 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).

[12] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *ICCV*. IEEE, 2980–2988.

[13] 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.

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

[15] 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.

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

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

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

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

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

[21] 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.

[22] 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.

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

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

[25] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2019. VStore: A Data Store for Analytics on Large Videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 16.