RESEARCH-ARTICLE

Aerogel: Adaptive Query Processing of ML Queries

GAURAV TARLOK KAKKAR, Georgia Institute of Technology, Atlanta, GA, United States

JIASHEN CAO, Georgia Institute of Technology, Atlanta, GA, United States

AUBHRO SENGUPTA, Georgia Institute of Technology, Atlanta, GA, United States

JOY ARULRAJ, Georgia Institute of Technology, Atlanta, GA, United States

HYESOON KIM, Georgia Institute of Technology, Atlanta, GA, United States

Proceedings of the ACM on Management of Data, Volume 3, Issue 3 (June 2025)
https://doi.org/10.1145/3725408
EISSN: 2836-

Aero: Adaptive Query Processing of ML Queries

GAURAV TARLOK KAKKAR*, Georgia Institute of Technology, USA
JIASHEN CAO*, Georgia Institute of Technology, USA
AUBHRO SENGUPTA, Georgia Institute of Technology, USA
JOY ARULRAJ, Georgia Institute of Technology, USA
HYESOON KIM, Georgia Institute of Technology, USA

Query optimization is critical in relational database management systems (DBMSs) for ensuring efficient query processing. The query optimizer relies on precise selectivity and cost estimates to generate optimal query plans for execution. However, this static query optimization approach falls short for DBMSs handling machine learning (ML) queries. ML-centric DBMSs face distinct challenges in query optimization. First, performance bottlenecks shift to user-defined functions (UDFs), often encapsulating deep learning models, making it difficult to estimate UDF statistics without profiling the query. Second, optimal query plans for ML queries are data-dependent, requiring dynamic plan adjustments during execution.

To address these challenges, we introduce Aero, an ML-centric DBMS that utilizes adaptive query processing (AQP) for efficiently processing ML queries. Aero optimizes the evaluation of UDF-based query predicates by dynamically adjusting predicate evaluation order and enhancing UDF execution scalability. By integrating AQP, Aero continuously monitors UDF statistics, routes data to predicates in an optimal order, and dynamically allocates resources for evaluating predicates. Aero achieves up to 6.4× speedup compared to a state-of-the-art ML-centric DBMS across four diverse use cases, with no impact on accuracy.

CCS Concepts: • **Information systems** → **Data management systems**; **Database query processing**; **Query optimization**; • **Computer systems organization** → Distributed architectures; • **Computing methodologies** → *Artificial intelligence*.

Additional Key Words and Phrases: Adaptive Query Processing (AQP); Machine Learning Queries; Database Management Systems (DBMS); Resource Utilization

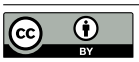
ACM Reference Format:
Gaurav Tarlok Kakkar, Jiashen Cao, Aubhro Sengupta, Joy Arulraj, and Hyesoon Kim. 2025. Aero: Adaptive Query Processing of ML Queries. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 174 (June 2025), 27 pages. https://doi.org/10.1145/3725408

1 Introduction

Modern database systems support ML-powered queries, enabling users to directly execute ML models within the database. These models are often wrapped within user-defined functions (UDFs). Since evaluating these functions is computationally expensive, they have become the central focus for query optimization. For example, video database management systems (VDBMSs), which heavily rely on computer vision algorithms, often encounter performance bottlenecks attributed to UDFs [14, 51, 60].

*Equal contribution (order determined by dice roll).

Authors' Contact Information: Gaurav Tarlok Kakkar, Georgia Institute of Technology, USA, gkakkar7@gatech.edu; Jiashen Cao, Georgia Institute of Technology, USA, jiashenc@gatech.edu; Aubhro Sengupta, Georgia Institute of Technology, USA, aubhros@gatech.edu; Joy Arulraj, Georgia Institute of Technology, USA, arulraj@gatech.edu; Hyesoon Kim, Georgia Institute of Technology, USA, hyesoon@cc.gatech.edu.



ACM 2836-6573/2025/6-ART174
https://doi.org/10.1145/3725408

State-of-the-art ML-centric DBMSs [14, 34, 51, 64] utilize a static query optimization approach, typically relying on UDF statistics derived from sample data. However, this approach struggles to accommodate the dynamic nature of UDF execution. Factors like varying data characteristics and system-level optimizations (*e.g.,* caching) cause UDF cost, resource utilization, and selectivity to fluctuate significantly during query execution [60], motivating the need for adaptive query processing (AQP).

AQP leverages runtime feedback to dynamically adjust query plans [9, 13, 22]. While successful in traditional databases, its application to ML-centric queries remains largely unexplored. This paper introduces Aero, a novel ML-centric DBMS designed to leverage AQP for efficient processing of ML queries. Aero dynamically refines query plans and resource allocation, adapting to the runtime characteristics of UDFs. This eliminates reliance on potentially inaccurate prior statistics and enables more efficient hardware utilization, leading to improved query performance.

Motivation. Consider a manufacturing facility's monitoring system detecting unsafe situations, such as workers without hard hats.

```
SELECT id FROM video
WHERE ['person'] <@ ObjectDetector(data).labels
AND ['no hardhat'] <@ HardHatDetector(data).labels;
```

Listing 1. Query to capture unsafe operating situations.

The query in Listing 1 identifies potentially unsafe video segments within a VDBMS. The query applies two predicates: one to determine if a worker is present in the scene (`['person'] <@ ObjectDetector(data).labels`), and another to verify if a hard hat is missing (`['no hardhat'] <@ HardHatDetector(data).labels`). From a query optimization standpoint, the evaluation order of these computationally expensive UDF predicates is critical. Choosing the optimal order can drastically reduce query execution time. Traditional predicate reordering technique [27] relies on selectivity and cost estimates, often ranking predicates using a scoring function (*e.g.,* $\frac{\text{cost}}{1-\text{selectivity}}$). However, this classical approach suffers from two limitations.

I - Unreliable Statistics. Prior ML-centric DBMSs either assume that accurate selectivity and cost statistics for UDFs are readily available [60] or estimate them by running the UDFs over sample data [14, 51]. However, such estimates can be inaccurate as the UDF evaluation cost may vary throughout the lifespan of a query. For example, the execution cost of `ObjectDetector` is correlated with the number of bounding boxes present in a frame, which can vary significantly across frames. Additionally, other system-level optimizations like caching [60] further reduce the accuracy of these estimates because caching and reusing UDF outputs directly impacts the cost.

Moreover, estimating statistics during query optimization increases both optimization time and the complexity of implementing the query optimizer. For example, to estimate the selectivity of `HardHatDetector` during the query optimization phase, the optimizer must execute the query on sample data, re-optimize the plan based on these estimates, and then pass the optimized plan to the query executor. This introduces additional overhead and complicates the design of query optimizer.

II - Unscalable UDF Execution. Achieving optimal performance with computationally expensive UDFs requires careful hardware resource management. Because UDFs often exhibit diverse characteristics, the query execution engine must monitor each UDF's resource usage throughout execution and allocate resources accordingly to achieve optimal performance. Static query optimization struggles to accommodate this level of runtime adaptation.

Additionally, UDFs can be processed by multiple workers concurrently, so a significant load imbalance between workers can lead to lower throughput and longer processing time. While simple policies like round-robin are adequate for preventing workload imbalance in common scenarios,

certain queries demand advanced load-balancing policies that consider data characteristics for workload distribution. For example, in large language model (LLM) question answering, execution latency is often strongly correlated with the lengths of both the input and output. Therefore, to avoid load imbalance, the DBMS must account for this data-dependent workload disparity, unlike a round-robin policy, which merely distributes requests evenly among workers without considering workload variation. We will later demonstrate that adopting a data-aware load balancing policy results in up to 1.3× speedup. Static query processing cannot accommodate these adaptive, on-the-fly optimizations.

OUR APPROACH. To address the limitations of static optimization for ML-centric queries, we introduce AERO, an ML-centric DBMS that implements a novel AQP framework. AERO dynamically adapts to the runtime characteristics of UDFs in three ways.

First, AERO eliminates the reliance on often inaccurate a priori UDF statistics by employing runtime monitoring that considers data characteristics and system optimizations like caching. This enables AERO to optimize predicate evaluation order on the fly using novel routing policies (implemented in the EDDY framework) that adapt to the observed runtime statistics of UDFs. Second, to efficiently manage hardware resources, AERO introduces LAMINAR, an automated scaling component that uses a low-overhead resource scaling strategy, called greedy-allocation-conservative-use (GACU). GACU achieves high hardware utilization through proactive and rapid scaling in response to monitored runtime GPU utilization. Finally, because UDF execution time can vary significantly depending on the input data, AERO uses data-aware load balancing in LAMINAR to distribute workloads evenly across workers, minimizing overall query time.

CONTRIBUTIONS.

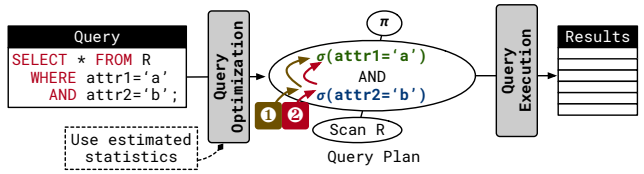
- We present AERO, a novel ML-centric DBMS leveraging AQP for efficient processing of ML queries. AERO leverages runtime UDF statistics monitoring and adaptive routing policies (via EDDY) to optimize predicate evaluation order.
- We propose LAMINAR, a novel automated scaling component designed for optimizing UDF hardware utilization within AQP framework. LAMINAR incorporates two key innovations: (1) GACU, a novel low-overhead resource scaling strategy to maximize hardware utilization; and (2) data-aware load balancing to effectively handle the uneven execution times often observed with data-dependent UDFs.
- We evaluate AERO across four use cases, including two in video analytics, one in image analysis, and one leveraging LLMs for analytics. Our results demonstrate that AERO achieves up to 6.4× speedup compared to state-of-the-art static optimization approaches, without impacting accuracy.

2 Background

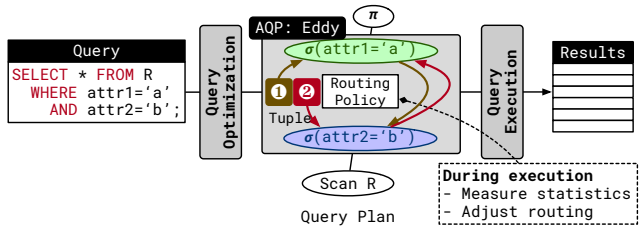
We first explain the SOTA adaptive query processing mechanisms in § 2.1. We then introduce how ML functions are typically integrated into DBMSs in § 2.2.

2.1 AQP in Relational DBMS

Traditional relational DBMSs commonly use a static query execution pipeline, which consists of query parsing, query optimization, and query execution, as shown in Fig. 1a. In this example, the query optimizer uses the estimated statistics to decide the optimal predicate ordering – whether to evaluate the predicate on attribute one or attribute two first. These estimated statistics can be inaccurate, often leading to sub-optimal query plans. To tackle the shortcomings mentioned above, many prior efforts [9, 10, 13, 22, 33] have proposed adaptive query processing that leverage runtime statistics to adjust the query plan.



(a) Static query processing pipeline.



(b) AQP query processing pipeline.

Fig. 1. Query Execution Pipelines – In static query processing, the predicate ordering is determined based on statistics estimated during query optimization. In contrast, adaptive query processing dynamically governs the predicate ordering during query execution.

Eddy. Among these efforts, Eddy [9] is a pioneer in proposing a systematic AQP framework that continuously reorders the application of pipeline operators in a query plan on a tuple-by-tuple basis. The query optimizer constructs an Eddy operator during optimization. As shown in Fig. 1b, the Eddy operator contains two inner operators (a.k.a selection predicates), σ(R.attr1) = 'a' and σ(R.attr2) = 'b'. The routing policy inside the Eddy determines which predicate to evaluate first during execution. Eddy also monitors execution statistics like execution cost and selectivity to adjust the routing table for optimal routing. In the original paper [9], a routing policy based on both execution cost and predicate selectivity demonstrated good performance. Intuitively, the predicate that runs faster and filters more tuples is prioritized. Eddy maintains an input queue for each inner operator, from which the execution cost is inferred based on the average queue length. It also uses a lottery system [56] to infer the selectivity of each inner operator.

While Eddy highlights the potential of adaptive routing in relational processing, its overhead often outweighs the benefits in practice. In contrast, Aero builds on the foundational concepts of Eddy and applies them to ML-centric workloads, where UDFs dominate query execution costs. Aero demonstrates that adaptive routing can yield significant performance gains in this novel context. Furthermore, Aero shows that the cost-driven routing policy outperforms the traditional score-driven approach for ML workloads. To address the unique challenges of ML workloads, such as the high computational cost of UDFs and the need for efficient GPU utilization, Aero incorporates Laminar. This automated scaling framework optimizes GPU utilization via dynamic resource scaling and data-aware load balancing.

Content-based Routing. Bizarro *et al.* [13] observed that the routing based on average statistics, as suggested in the Eddy paper, can be significantly improved using data or content-based routing. Their mechanism is built on top of the Eddy adaptive query processing framework. The key idea is that a tuple may have some attributes that strongly correlate with the predicate selectivity (*e.g.*, time of the year and the temperature). By examining the value of those attributes, the Eddy operator can determine the optimal predicate ordering. A drawback of this approach, when applied solely to relational operators, is the potentially high overhead of checking attribute values and determining the corresponding routing, which can outweigh the execution time of the relational operators.

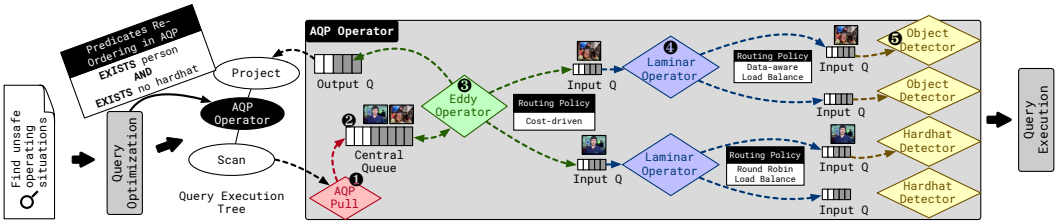


Fig. 2. Detailed AQP execution plan and its internal – left shows the execution tree with the AQP operator. *Diamonds* represent physical processes separate from the main process. All physical queues facilitate communication between data producers and consumers. *Rectangles* represent the routing policies attached to their respective processes. The AQP operator includes an Eddy operator, which determines the optimal execution order, and two Laminar operators (one for each predicate) to manage runtime scaling for improved hardware utilization.

2.2 ML Functions as UDFs

As the demand for intelligent analytics increases, standard SQL syntax has become insufficient to express complex business logic. To address this, DBMSs now commonly support UDFs, allowing execution engines to leverage third-party libraries for parts of query execution. The logic of UDF is typically unknown to the DBMS. Many recent DBMSs [5, 34, 65] further enable easy integration with ML algorithms through Python UDF interfaces. In this paper, we assume that ML algorithms are packaged in UDFs by users of DBMSs and their characteristics are transparent to the DBMS.

3 Aero

In this section, we first explain the principles that we follow in designing the Aero in § 3.1. We then explain the internal architecture of the AQP operator in § 3.2 using example query Listing 1.

3.1 Design Principles

Though the traditional static query optimizer is very powerful with many effective optimization rules, it cannot perform any optimizations or resource management during runtime. Thus, to address these shortcomings, when designing the AQP operator, we follow two important principles: 1. seamless and transparent integration with the existing systems; and 2. automatic optimization for predicate ordering and resource utilization during runtime.

Seamless Integration. Almost all DBMSs follow the workflow of parsing, optimizing, and executing queries. Our design leverages the observation that query optimization commonly performs static optimizations, especially rule-based optimization. To introduce the AQP operator, Aero instrumentw a new rule to rewrite any plans that involve multiple UDFs in a conjunction statement to an AQP plan. The AQP plan can then be translated to an AQP operator in the query execution tree as depicted in Fig. 2. This design ensures our design is extensible to existing static optimization infrastructure.

Dynamic Optimization. The most prominent advantage of Aero is its ability to automatically optimize predicate execution order and resource utilization, transparently to the existing static query optimization. In our design, we delegate these two tasks to Eddy operator and Laminar operator, respectively. As mentioned, the static query optimization simply sets up the AQP operator, containing Eddy operator and Laminar operator internally, without dictating any ordering or degree of parallelism of workers. As depicted in Fig. 2, during the query execution, the Eddy component is responsible for identifying the optimal predicate ordering (*e.g.,* prioritize `ObjectDetector` over `HardHatDetector`) based on monitored statistics. Additionally, the Laminar operator takes care

of scaling up the number of workers for predicates when the hardware is underutilized. To ensure Aero can obtain the most optimal ordering in both Eddy operator and Laminar operator, we propose using the cost-driven routing and data-aware load balancing in the system. To ensure Aero can maximize hardware utilization without significant overhead, we propose a scheme called GACU for the Laminar operator.

3.2 Architecture of AQP Operator

We take a deep dive into the internal architecture of the AQP operator, following our query example in Listing 1. Once the query plan transitions to an execution tree, the execution tree executes in a top-down fashion. The parent operator (*i.e.,* projection) of the AQP operator pulls a batch of data from its output queue, while the AQP operator pulls data from its child (*i.e.,* scan operator) and performs all heavy computation asynchronously.

AQP Operator Internals. In this example, the AQP operator, consisting of both Eddy and Laminar operators, is responsible for evaluating the `ObjectDetector` and `HardHatDetector` predicates.

❶ The aqp pull dictates the data flow from AQP's child operators. It pulls data batches from the child operator and inserts them into the Central queue. In this example, the aqp pull gets frames from the input video.

❷ The Central queue serves as a buffer for batches from the child operator and after predicate computation.

❸ The Eddy operator gets data from the Central queue and orchestrates the data flow within the AQP operator. In this scenario, it must determine whether to route the frame to the `ObjectDetector` or `HardHatDetector` predicate first. A routing policy (§ 4.1) guides this decision. For example, it may prioritize routing the frame to the UDF with a lower cost. Additionally, it redirects completed data batches to the output queue.

❹ The Laminar operator gets data from the Eddy operator. Each predicate is associated with a Laminar operator. So in this example, the AQP operator contains a total of two Laminar operators, one for each `ObjectDetector` and `HardHatDetector`. The Laminar operator is tasked with monitoring hardware usage (*e.g.,* GPU utilization), determining the number of workers to spawn, and performing load balancing between workers. Here, it opts to spawn two workers (illustrated as ❺) for both `ObjectDetector` and `HardHatDetector`. Additionally, it employs a routing policy to balance the load among multiple workers. For instance, it may employ round-robin for simple load balancing, or opt for an advanced data-aware load balancing approach (§ 5.2) when the characteristics of data batches heavily influence the workload.

❺ The Laminar worker is responsible for evaluating the predicate. In this example, the Laminar operator spawns two workers each for both `ObjectDetector` and `HardHatDetector`. After evaluating a data batch, it is inserted back into the Central queue. Once a batch completes all predicates, it will be routed to the output queue by the Eddy operator. The parent executor of the AQP operator pulls data from the output queue iteratively.

4 Eddy Operator

In this section, we explain the design of the eddy operator in Aero. Its choice of routing policy (§ 4.1) and the adaptiveness of its statistics monitoring (§ 4.2) are particularly significant, both of which are pivotal in achieving optimal predicate ordering.

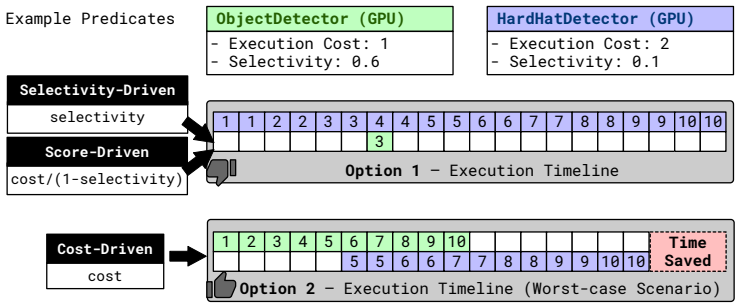


Fig. 3. Execution timeline of different routing policies – a comparison of selectivity-driven, score-driven, and cost-driven routing policies. One box represents a time unit. The Cost-driven routing policy offers a 3 time-unit saving even in the worst-case scenario.

4.1 Cost-Driven Routing

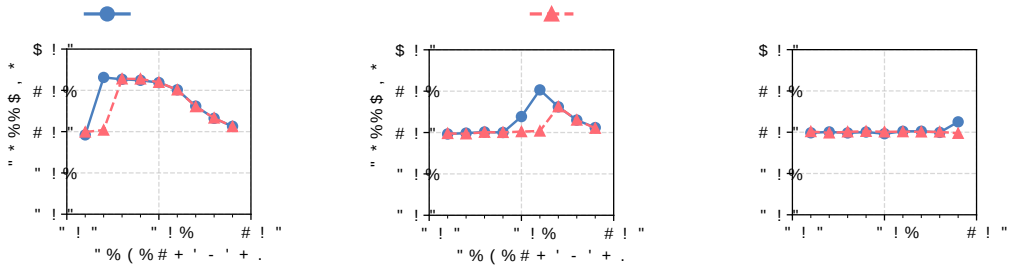
In AERO, the eddy operator primarily monitors two key statistics: the cost and selectivity of predicates, drawing inspiration from prior works [9, 13]. Routing decisions are then made based on these collected statistics.

ROUTING POLICY PRINCIPLE. An optimal routing policy is crucial for the AQP execution framework. Previous studies exploring predicate reordering [9, 10, 13] emphasize the significance of both the cost and selectivity of predicates for performance. Ideally, a faster predicate capable of filtering a substantial amount of data is preferred to run first, leading to a significant reduction in the invocation of slower predicates. Consequently, a score function, $\frac{\text{cost}}{1-\text{selectivity}}$, is commonly used to rank each predicate [27]. The predicate with the lowest score is prioritized for execution first, contributing to optimal performance.

However, our findings indicate that the scoring function ($\frac{\text{cost}}{1-\text{selectivity}}$) for predicate ordering is not optimal when dealing with concurrent execution. Selectivity is critical in sequential execution because higher selectivity of a predicate can significantly shorten the execution of subsequent predicates, resulting in a better overall performance. Whereas, in the concurrent setting, different predicates operate in a pipeline parallelism fashion. Because their executions are overlapped, the selectivity of a predicate plays a less important role in the overall performance. In contrast, a slower predicate becomes the pipeline's bottleneck, blocking the execution of subsequent predicates and degrading overall performance. We argue that in the case of ML queries, concurrent predicate execution is common. One example is that two predicates are evaluated by workers (❺ in Fig. 2) running concurrently on two different GPUs. The same holds when one predicate is evaluated on the CPU and the other on the GPU, allowing workers associated with these predicates to execute concurrently. In such concurrent settings, our empirical results (presented in Fig. 4) show that the cost-driven routing policy outperforms the score-driven routing policy.

REASONING WITH EXAMPLES. Our example query (Listing 1) has two predicates: `ObjectDetector` and `HardHatDetector`. Assuming the system has access to adequate hardware resources, these two predicates can be executed concurrently on the GPU. As mentioned earlier, in a concurrent setting, we notice a shift where the selectivity of a predicate no longer impacts query performance. Instead, only the execution cost of a predicate affects optimal execution efficiency. We illustrate this shift in our reasoning using a simple example (Fig. 3).

For both predicates in the example, we assign relative execution costs: `ObjectDetector` with a cost of 1, and `HardHatDetector` with a cost of 2. `HardHatDetector` has a higher execution cost due to more complex algorithms. We set the selectivity of `ObjectDetector` to 0.6 and the selectivity of



(a) **Selectivity of B** = 0.1 (b) **Selectivity of B** = 0.5 (c) **Selectivity of B** = 0.9

Fig. 4. Controlled experiment – speedup comparison of the cost-driven routing policy over other policies under different selectivities.

`HardHatDetector` to 0.1. To examine the different routing policies, consider a total of 10 units of data for evaluation.

For the selectivity-driven routing, `HardHatDetector` is preferred over `ObjectDetector` as $0.1 < 0.6$. Similarly, for score-driven routing, `HardHatDetector` is preferred over `ObjectDetector` as $\frac{2}{1-0.1} < \frac{1}{1-0.6}$. In this case, the execution spends 20 time units on the `HardHatDetector` predicate, taking 2 time units to process each data unit (1 box represents a time unit in Fig. 3). Since only one out of ten data units (0.1 selectivity) is passed to the `ObjectDetector` predicate for further evaluation, and it executes concurrently, the overall cost is 20 time units. With the cost-driven routing policy (option 2), data is routed to the `ObjectDetector` first. Considering the worst-case scenario, the execution spends 10 time units on the `ObjectDetector`, and then six out of ten data points are passed to `HardHatDetector` for further evaluation, resulting in a total time cost of 17 units. Despite the higher selectivity of `ObjectDetector`, leading to more data for further evaluation by the other predicate, the overall execution time is shorter due to better computation overlap between the two predicates. In the average-case scenario, there will be even more overlap of computation, resulting even shorter execution time. The intuition is that since `HardHatDetector` serves as the bottleneck in the pipeline due to its high execution cost, executing the faster predicate first helps alleviate the bottleneck, ultimately improving query performance.

Our Approach. Motivated by the above example and our observation, we contribute a concurrent-execution-tailored routing policy in Aero. We choose a cost-driven routing policy in Aero when two predicates can run concurrently. This approach sets Aero apart from other existing AQP systems. For other situations where predicates must run on the same hardware resource and require synchronization between processes, Aero falls back to using the classic score-based approach.

Theoretical Analysis. To justify our cost-based policy, we use the queueing theory [1, 6] to conduct the theoretical study. When two predicates are executed concurrently, and one depends on the other (pipeline parallelization), one predicate becomes the pipeline's throughput bottleneck. Queueing theory helps identify this bottleneck, whose throughput determines the pipeline's overall throughput. We find that prioritizing the predicate with the lower cost first consistently leads to higher pipeline throughput and shorter query processing time.

We present the proof below. Given two predicates P_1 and P_2 with selectivities s_1 and s_2 (where $0 \leq s_1, s_2 \leq 1$) and costs c_1 and c_2 (where $c_1, c_2 > 0$), respectively. According to the queueing theory [1, 6], the service rate (*i.e.,* throughput) for a worker executing a predicate with cost c is $\lambda = \frac{1}{c}$. To process N data, the processing time (T_1) when P_1 runs before P_2 is:

$$T_1 = \max\left\{\frac{N}{\lambda_1}, \frac{s_1 \cdot N}{\lambda_2}\right\} = N \cdot \max\left\{c_1, s_1 \cdot c_2\right\},$$

and when P_2 runs before P_1, the processing time (T_2) is:

$$T_2 = \max\left\{\frac{N}{\lambda_2}, \frac{s_2 \cdot N}{\lambda_1}\right\} = N \cdot \max\left\{c_2, s_2 \cdot c_1\right\}.$$

We prefer P_1 over P_2 if $T_1 - T_2 < 0$, which implies:

$$\max\left\{c_1, s_1 \cdot c_2\right\} - \max\left\{c_2, s_2 \cdot c_1\right\} < 0.$$

Now, consider the following cases to confirm that $c_1 < c_2$ always leads to an optimal ordering:

1. $c_1 \geq s_1 c_2$ and $c_2 \geq s_2 c_1 \Rightarrow c_1 < c_2$
2. $c_1 \geq s_1 c_2$ and $c_2 < s_2 c_1 \Rightarrow s_2 > 1$ (infeasible)
3. $c_1 < s_1 c_2$ and $c_2 \geq s_2 c_1 \Rightarrow s_1 < 1$ (trivially true)
4. $c_1 < s_1 c_2$ and $c_2 < s_2 c_1 \Rightarrow c_2 > c_1$ and $c_1 > c_2$ (contradiction)

Therefore, $T_1 < T_2$ holds if and only if $c_1 < c_2$, proving our claim that the cost-based policy always produces an optimal ordering.

Sensitivity Analysis using Controlled Queries. We conduct an investigation comparing the performance of cost-driven routing with both score-driven and selectivity-driven routing. We define two concurrently running predicates, denoted as A and B, with execution costs of 10 ms and 20 ms, respectively. The selectivity of predicate B is configured to be 0.1, 0.5, and 0.9 as shown in Fig. 4a, Fig. 4b, and Fig. 4c, respectively. We then vary the selectivity of A from 0.1 to 0.9. We measure the query processing speedup of the cost-driven routing policy over both the score-driven and the selectivity-driven routing policies. The results show that the cost-driven routing policy never yields a worse query processing time compared to the other routing policies. Moreover, the cost-driven routing policy tends to outperform the score-driven and selectivity-driven routing policies when the high-cost predicate has low selectivity. Among all routing policies, the solely selectivity-driven routing policy exhibits the worst performance. Our empirical study verifies our argument that cost-driven routing is preferred for concurrent execution.

Impact of predicate reordering on accuracy. Although UDFs utilize ML models, their outputs are invariant to predicate reordering. This is because the UDFs themselves are deterministic, consistently returning the same output labels for a given input. Consequently, the final query accuracy is unaffected.

Consider our example (Listing 1), which has `ObjectDetector` and `HardHatDetector` predicates with a logical `AND`. The final set of video frames will always be the same, as a frame must satisfy *both* predicates to be included in the result regardless of predicate order. Therefore, the accuracy of the final results remains unaffected by the evaluation order. We empirically confirm this by showing the final results are identical across different predicate orderings (§ 7.2).

4.2 Adaptive Routing

In the previous section, we study the Aero's cost-driven routing, which leverages the cost of predicates to determine their appropriate order during the execution. Since the cost of the predicates does not change during execution, the optimal ordering of the predicates, and consequently the query plan, remains fixed. However, our exploration of real-world exploratory analysis queries reveals that the cost of a predicate can fluctuate significantly during execution due to additional optimizations or changes in data characteristics. This motivates us to examine the benefits of using Aero for execution-time adaptive routing.

```
-- Q1: Initial exploratory query
SELECT id, ObjectDetector(data).labels FROM video WHERE id > 1000 AND id < 7000;
```

```
-- Q2: Initial exploratory query
SELECT id, HardHatDetector(data).labels FROM video WHERE id > 8000 AND id < 14000;

-- Q3: Recurrent query
SELECT id FROM video
 WHERE ['person'] <@ ObjectDetector(data).labels
 AND ['no hardhat'] <@ HardHatDetector(data).labels;
```

Listing 2. Query to identify unsafe situations in a warehouse.

Motivating Example. Listing 2 illustrates a motivating example of exploratory data analysis that involves multiple queries exploring video footage from a construction site. For simplicity, we consider simplified predicate conditions. In realistic scenarios, users may conduct deeper analyses of video segments based on factors such as time, weather conditions, and working locations.

In $Q1$, the user examines objects in the video between frame ids 1000 and 7000 using `ObjectDetector`. Next, in $Q2$, the user focuses on identifying hardhats using `HardHatDetector` between frame ids 8000 and 14000. Subsequently, in $Q3$, they run a query involving the previous UDFs to identify unsafe situations in the video where workers are not wearing hard hats when they should be. Given that evaluating the same UDF on the same data across queries is a common occurrence in real-world exploratory use cases, prior work [60] incorporates optimizations like result caching and reusing to expedite these queries. Here, the results of `ObjectDetector` for the range `id > 1000 AND id < 7000` and `HardHatDetector` for the range `id > 8000 AND id < 14000` are cached after executing $Q1$ and $Q2$. Consequently, when $Q3$ is executed, it efficiently reuses the cached results.

For the range `id > 1000 AND id < 7000`, the cost of `ObjectDetector` will be significantly lower than `HardHatDetector` because reusing cached results eliminates rerunning `ObjectDetector`. Therefore, the optimal query plan should prioritize `ObjectDetector` predicate for this range. Similarly, for the range `id > 8000 AND id < 14000`, the optimal plan should instead prioritize the `HardHatDetector` predicate because it will reuse cached results. In conclusion, optimizations like the use of partial caches introduce variability in the optimal predicate ordering during query execution.

reuse-aware Routing. To enhance our routing logic, we propose an reuse-aware routing algorithm, which builds upon the cost-driven routing discussed in § 4.1. In addition to the statistics collected for cost-driven routing, our algorithm ensures that the routing policy takes into account the cache hit statistics. The cache hit rate is used to adjust the estimated cost of evaluating a predicate. During routing, the algorithm first checks the potential cache hit rate for a batch before making routing decisions. After obtaining the cache hit rate and the actual predicate evaluation cost, the routing algorithm estimates the execution cost for a routing batch using the following equation.

$$\text{estimated cost} = (1 - \text{cache hit rate}) \cdot \text{actual cost of a predicate}$$

We assume that the cache access overhead is negligible compared to the cost of evaluating the predicate. Lastly, the routing algorithm prioritizes scheduling data to the predicate with the lowest estimated cost for optimal performance.

5 Laminar Operator

In the previous section, we explored the benefits of the Eddy operator, which determines the optimal predicate execution order based on their runtime statistics. Beyond the order of predicate execution, another pivotal factor influencing query performance is the proper utilization of underlying hardware resources. This is particularly crucial for operators and functions that heavily depend on GPUs (*e.g.*, `ObjectDetector`). For example, as shown in § 7.5, under-utilization of GPUs can degrade performance by a factor of 6.0×. To address this, we introduce the Laminar operator following

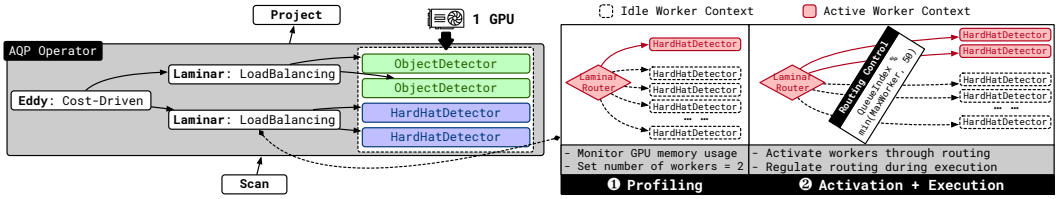


Fig. 5. Laminar workflow for Listing 1 – the left side illustrates two-way parallelization enabled by the Laminar operator with a single GPU. On the right, the figure shows the query execution beginning with a profiling phase to assess worker hardware utilization, followed by the activation of more workers.

the Eddy operator, illustrated in Fig. 2. The Laminar operator enhances query performance by (1) ensuring optimal hardware utilization, particularly for GPU resources (§ 5.1); (2) facilitating robust scalability as the system increases the number of resources (§ 5.1); and (3) achieving effective load balancing among multiple backend workers during query execution (§ 5.2).

5.1 Hardware Utilization and Scalability

The utilization of the underlying hardware significantly impacts query performance. In queries like Listing 1, where the GPU-intensive parts (*e.g.,* `ObjectDetector`) are the bottleneck, the efficiency of GPU usage determines the final query execution performance. Prior work in real-time serving for deep neural networks [23, 59] emphasizes the importance of GPU utilization for achieving high throughput. While most works [21, 29, 52, 54, 58] aim to improve throughput without compromising the Service Level Objective (SLO) latency, they also introduce techniques to improve GPU utilization for maximal query throughput. One of the most effective and straightforward of these techniques is adaptive runtime batching. This method determines the ideal batch size during runtime and performs model inference with the determined batch size.

Challenges. However, the adaptive runtime batching does not fully apply to our use case for two reasons.

- First, batching assumes uniform dimensions across all input tensors, which is not always feasible in many practical applications. As the query becomes more complicated, inputs to the UDF predicate do not always have the same dimension. Consider an example where a dog owner attempts to locate their lost pet, based on its color and breed. The query finds existing dog objects first and performs additional breed and color classifications. The dimensions of inputs to the breed classification UDF can vary based on the bounding boxes of the detected dog objects, which makes it challenging to utilize data batching for better GPU utilization. This has been confirmed by other works [29] as well.
- Second, Aero heavily relies on a UDFs interface to allow users to use any third-party ML library. Many such libraries do not support execution with a batch size other than one, which inherently prevents Aero from employing adaptive runtime batching. For example, the YOLOv8 API [55] exposes an API interface where users pass a single image as a parameter. This can lead to serious GPU under-utilization issues, necessitating a solution to overcome the fixed batch based on the actual GPU usage.

Batch-agnostic Parallelization. Inspired by the spatial-multiplexing approach from previous works [29, 59], we implement a batch-agnostic parallelization approach for the Laminar operator. Each predicate worker only evaluates a single batch of data, but multiple predicate workers perform evaluation simultaneously. When GPU utilization is low, the Laminar operator spawns multiple concurrent predicate workers, enabling the overlap of data movement, and computation on

both CPUs and GPUs. The query plan for Listing 1 with two-way parallelization is depicted in Fig. 5. In this example, the Laminar operator spawns two workers for each predicate and assumes the system has only one GPU.

GACU: Greedy-Allocation-Conservative-Use. Even though the spatial-multiplexing approach has been well explored, we note that our key contribution is to support this feature in the context of AQP framework. The Laminar operator manages parallelism by spawning workers during execution. Ideally, parallelism should be maximized. However, allocating too many workers to the same GPU risks out-of-memory errors, resulting in DBMS crashes and stalled queries. To mitigate this, the Laminar operator has been designed to dynamically adjust the number of workers based on their memory usage, which is collected during the profiling phase. However, dynamically increasing the number of workers during execution presents challenges, as the framework must acquire additional processes, create queues, and adjust the pipeline while the query is being processed.

To tackle this, we introduce greedy-allocation-conservative-use (gacu). It allows the Laminar operator to allocate more workers through routing as needed, requiring no change to the framework. The core design of gacu involves greedily allocating multiple worker contexts at the start of a query, but only activating a small subset of those workers during execution, based on runtime statistics (*e.g.*,, GPU memory usage). For our experiments, we configure gacu to initialize 50 workers at the start of query execution, allowing us to utilize GPU resources effectively.

Predicate workers allocate GPU resources lazily; only activated workers will occupy GPU resources (*i.e.,* only when requests are present in their queues). To scale up, the Laminar operator activates more predicate workers by routing data to them, without needing to adjust the query pipeline during execution.

As demonstrated in Fig. 5, during startup profiling phase, the Aero conservatively only activates one worker. The Laminar operator monitors worker memory usage to decide how many workers to activate. After the profiling, the Laminar operator updates the number of workers to be $\lfloor \frac{\text{Total GPU memory}}{\text{Used GPU memory}} \rfloor$. It subsequently activates the remaining workers until the number of active workers reaches the determined threshold. Throughout query execution, the routing logic directs data exclusively to active workers, avoiding the activation of new ones. Other inactive worker contexts will stay idle until they are cleaned up at the end of the query execution.

Scaling Out. When the DBMS has multiple GPUs, all GPUs will be assigned workers to run the same predicates. We follow a similar approach to determine the number of workers per GPU as discussed, whose upper-bound is also set to 50. The key difference is that the Laminar operator now manages workers situated on different physical GPUs. To ensure a good query execution performance, we adopt a GPU-aware Laminar routing policy, in which we configure the routing logic to alternate between GPUs when routing a continuous data sequence. Through our experiment(§ 7.5), we find this approach can achieve a good load balancing between GPUs and consequently improve the overall GPU utilization.

5.2 Data-Aware Load Balancing

Lastly, using data-agnostic routing policies, such as round-robin, for Laminar routing can lead to suboptimal performance due to workload imbalance when scaling up.

Workload Imbalance. Data-dependent predicate costs contribute significantly to workload imbalance, as discussed in § 1. For example, the execution cost of `ObjectDetector` varies with input frame size, with larger frames incurring higher costs. This is amplified for language-based UDFs, such as LLMs, which generate results token by token. Their execution cost is significantly influenced by both the length of the input and the output. Moreover, batching can exacerbate this imbalance

when a query contains multiple predicates. Earlier predicates may filter out rows from the batch leading to variable batch sizes for the subsequent predicates, creating an uneven workload.

Data-agnostic routing policies, like round-robin, distribute data to workers without accounting for such variations in execution cost due to differences in input data. This may lead to overloading busy workers and underutilizing idle ones, as these routing policies overlook data characteristics.

Data-Aware Load Balancing. To mitigate the workload imbalances arising from data-dependent predicate costs, we propose a data-aware load balancing (DLB) within Laminar routing. Unlike reactive policies like power-of-two choice [44] or least connection [2], which react to observed queue congestion and often act too late, the DLB proactively balances load using data characteristics. It makes routing decisions based on heuristics derived from actual data, such as image sizes. Users can define custom heuristics for the UDFs. By default, Aero uses input size as a proxy for execution cost. For language-based UDFs, this corresponds to text length, and for vision models, it corresponds to input frame size.

6 Implementation

Aero augments EvaDB [34], an ML-centric database system, to support AQP. We implement routers and workers using the Ray [46] framework. Routers and workers that redirect and process data are implemented as Ray remote functions, while queues that buffer data are implemented using Ray queues.

Metadata for Data Routing. The AQP executor requires associating metadata with each batch. To uniquely identify each batch, it assigns a unique ID to every routing batch. Furthermore, the Eddy router maintains a hash table to track predicates visited by each batch using this ID. This generic metadata is essential regardless of the routing policy used in the Eddy router. Before routing the batch, the Eddy router checks the hash table to decide whether to skip or run the predicate based on its visitation status, preventing redundant computation. Once a batch has visited all predicates, it is routed to the output queue.

Additionally, the Eddy router can maintain extra metadata for different routing policies. For example, the cost-driven routing policy (§ 4.1) necessitates monitoring input queue length and execution time for each predicate as part of its cost. The Eddy router tracks these statistics as additional metadata and uses them to update routing decisions.

Eager Materialization. During predicate evaluation, we use an eager materialization approach, promptly discarding tuples that do not satisfy the predicate condition. This simplifies the predicate short-circuiting logic, eliminating the need to track information at the row level within the batch.

Model Serving. In Aero, users have flexibility in implementing models as UDFs, which may incorporate model-serving frameworks. However, our system prioritizes models that do not rely on external model-serving frameworks for performance optimization. Laminar is designed to provide auto-scaling with minimal resource allocation overhead, managed by gacu. Model serving frameworks like Ray Serve allocate resources on demand to minimize pending requests in the queue. They additionally require user resource usage hints for auto-scaling. Our empirical study § 7.5 shows that its reactive scaling scheme is slower to adapt to the demand of the query and does not maximize resource usage of hardware resources.

Startup Profiling Phase. To route data batches based on their execution cost, Aero requires initial cost data, which is only available after running a few initial batches. We design a profiling phase during when batches are initially routed to all workers to gather execution cost data for each predicate. After this phase, batches are routed based on the chosen policy. To minimize the

overhead, we route just enough batches to execute all predicates, slightly delaying other batches until the phase completes.

7 Evaluation

In this section, we answer the following research questions:

RQ1: How does the query processing performance of Aero compare to other baseline systems (§ 7.2)?
RQ2: How effective are Aero's routing policies, including cost-driven routing, adaptive routing, and data-aware routing(§ 7.3)?
RQ3: How efficient is Aero when scaling up resources compared to other model-serving frameworks (§ 7.4)?
RQ4: What is the performance contribution of Eddy and Laminar in Aero (§ 7.5)?
RQ5: What are the overheads of Aero (§ 7.6)?

7.1 Experimental Setup

Hardware. We conducted experiments on a server with an AMD EPYC 7452 32-core processor with 256 GB of memory. The server has an NVIDIA A40 GPU, which has 48 GB of GPU memory. The server runs a Ubuntu 22.04.3 LTS operating system, and the GPU library is compiled with NVIDIA CUDA 12.0.

Dataset. We evaluate Aero using four datasets spanning video, image, and text modalities: The Dog dataset consists of a 19k-frame video sourced from YouTube, showcasing dogs of various breeds, sizes, and colors. The Warehouse dataset comprises a 14k-frame video from YouTube, depicting warehouse activities, including workers with and without safety gears. The IMDB dataset consists of 15k face images of actors sourced from the IMDB dataset [53]. Lastly, the FoodReview dataset consists of 600 McDonald's Google review comments, each accompanied by its respective rating.

Workload. The evaluation consists of six queries, each tailored to evaluate specific aspects of Aero's performance. These aspects include diverse data modalities (video, image, and text), predicates with varying selectivity and cost characteristics, routing policies, resource utilization, partial caching during exploratory analysis, and worker load imbalances.

Q1. This query retrieves dogs of specific breeds and colors, evaluating predicate reordering and routing policies.

```
SELECT * FROM Dog
JOIN LATERAL UNNEST(ObjectDetector(frame)) AS Obj(label, bbox, score)
WHERE Obj.label = 'dog'
 AND DogBreedClassifier(Crop(frame, bbox)) = 'labrador retriever'
 AND DogColorClassifier(Crop(frame, bbox)) = 'black';
```

Q2. To evaluate the impact of increasing the number of predicates, this query extends Q1 by including an additional predicate (identify larger dogs using the `Area`).

```
SELECT * FROM Dog
JOIN LATERAL UNNEST(ObjectDetector(frame)) AS Obj(label, bbox, score)
WHERE Obj.label = 'dog'
 AND DogBreedClassifier(Crop(frame, bbox)) = 'labrador retriever'
 AND DogColorClassifier(Crop(frame, bbox)) = 'black'
 AND Area(frame, bbox) > 0.4;
```

Q3. This query identifies unsafe warehouse operating conditions, focusing on workers operating machinery without hard hats. Specifically, it retrieves frames where a person is present but not wearing a hard hat. We evaluate this query to understand how LAMINAR impacts resource utilization.

```
SELECT * FROM Warehouse
WHERE ['person'] <@ ObjectDetector(frame).labels
 AND ['no hardhat'] <@ HardHatDetector(frame).labels;
```

Q4. Inspired by [57, 60], which emphasizes exploratory analysis, Q4 extends Q3 by assuming the user runs a couple of exploratory queries before executing the last retrieval query (refered as Q4). In this scenario, the results of the `ObjectDetector` and `HardHatDetector` are cached, and Q4 leverages these cached results to detect unsafe conditions efficiently. This query evaluates partial caching and the impact of adaptive routing policies.

```
-- Initial exploratory analysis
SELECT ObjectDetector(frame).labels FROM Warehouse
WHERE id > 1000 AND id < 7000;

SELECT HardHatDetector(frame).labels FROM Warehouse
WHERE id > 8000 AND id < 14000;

-- Q4: Last query using cached results.
SELECT * FROM Warehouse
WHERE ['person'] <@ ObjectDetector(frame).labels
 AND ['no hardhat'] <@ HardHatDetector(frame).labels;
```

Q5. Expanding beyond video use cases, we evaluate the performance of a query that analyzes the IMDB image dataset to retrieve images based on the depicted actor's age and emotion.

```
SELECT * FROM IMBD
WHERE AgeClassifier(frame) = '20-29'  AND EmotionClassifier(frame) = 'happy';
```

Q6. Lastly, this query examines how load imbalance among UDF workers affects performance. It analyzes food reviews using two predicates: a UDF employing a language model and a rating filter.

```
SELECT * FROM FoodReview
WHERE LLM('What is the following review about? Only choose "food" or "service"
    ', review) = 'food'
 AND rating <= 1;
```

PREDICATE MODEL AND STATISTICS. Our query suite consists of multiple predicates backed by different algorithms. We use the SOTA YoloV5 model [24] as `ObjectDetector`. For `DogColorClassifier`, we implement a simple heuristic, which classifies the object color like red, black, and other based on the HSV range. `DogBreedClassifier` utilizes a ViT-based transformer for identifying 120 dog breeds [62]. `HardHatDetector` detects hard hats with a fine-tuned YOLOv8s model. `AgeClassifier` and `EmotionClassifier` employ ViT-based transformers for age [50] and emotion [30] classification, respectively. The `LLM` predicate uses Orca, a local large language model with 13 billion parameters from the GPT4All [7] library, optimized for CPU execution with multi-threading capabilities. Table 1 lists the average selectivity and cost for the predicates in the workload.

BASELINES.

Table 1. Predicate Statistics – Average selectivity and cost.

Predicate	Avg. Selectivity	Avg. Cost (ms)
`DogBreedClassifier = 'labrador retriever'`	0.06	30.35
`DogColorClassifier = 'black'`	0.15	0.95
`['person'] <@ ObjectDetector(frame).labels`	0.83	21.79
`['no hardhat'] <@ HardHatDetector(frame).labels`	0.25	35
`AgeClassifier(frame) = '20 -29'`	0.57	12.2
`EmotionClassifier(frame) = 'happy'`	0.4	16

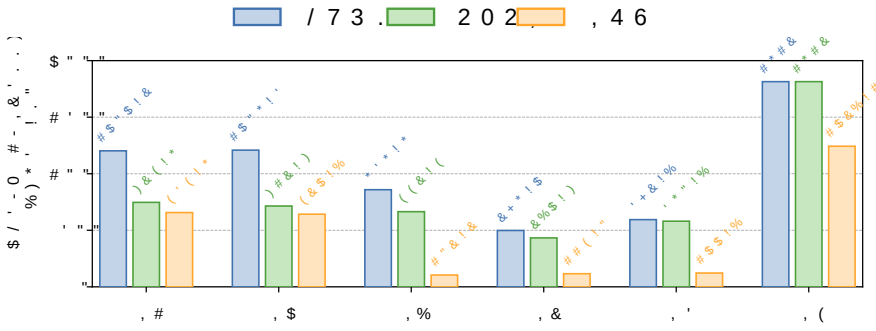


Fig. 6. End to end performance – a comparison between EvaDB, Viva, and Aero on Q1-Q6.

- EvaDB: an open-source video analytics system employing static, cascades-style query optimizer. It includes optimizations for exploratory video analytics, such as caching and reusing results of expensive UDFs, and predicate reordering.
- Viva: a recently proposed video analytics framework that also uses static query optimization. It estimates UDF statistics (*e.g.,* selectivity) using 3% of the input data. Viva proposes three optimizations: input filtering, model replacement, and predicate reordering. However, we focus solely on predicate reordering because the other two can negatively impact end-to-end accuracy. These other two optimizations, which trade off accuracy for improved performance, are orthogonal to our approach.
- Ray Serve: We also compare the Laminar in Aero with Ray Serve autoscaling. We defer the setup details of Ray Serve to § 7.5.

7.2 End-to-End Performance

In this section, we compare the end-to-end query performance of Aero against two baselines, EvaDB and Viva. For Q1, Aero outperforms EvaDB and Viva by 1.83× and 1.14×, respectively. This is because executing `DogColorClassifier` before `DogBreedClassifier` is beneficial, but EvaDB fails to reorder predicates, affecting its performance. Viva correctly identifies predicate ordering but incurs a statistics collection overhead.

To assess Aero's adaptability with an increased number of predicates, Q2 introduces one more predicate. Again, EvaDB fails to reorder predicates, resulting in 1.88× slower execution compared to Aero. While Viva efficiently determines the correct predicate ordering, it faces the statistics collection overhead. Note, for both Q1 and Q2, owing to the significant difference in the cost of

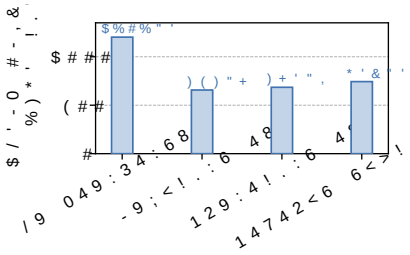


Fig. 7. Query processing time for Q1 w/o LAMINAR – comparison of four EDDY routing options: no reordering, cost-driven, score-driven, and selectivity-driven.

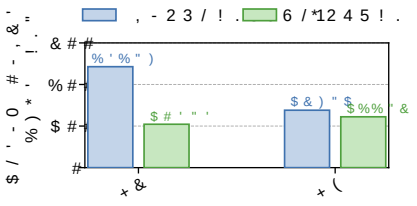


Fig. 8. Query processing time for Q3 and Q5 with LAMINAR enabled – comparison between score-driven and cost-driven EDDY policies under parallelization.

the UDFs (30.35 ms vs 0.95 ms, Table 1), it is feasible to estimate the correct ordering with just 3% data samples. Therefore, even with inaccurate estimates, VIVA can determine the optimal ordering. Additionally, as both queries are bottlenecked by `ObjectDetector`, the LAMINAR component of AERO provides a negligible advantage.

In Q3, AERO outperforms EVADB and VIVA by 8.22× and 6.37×, respectively, due to their failure to maximize GPU utilization. GACU enables AERO to dynamically increase worker numbers, thereby maximizing GPU utilization. EVADB's suboptimal predicate ordering further impacts its performance.

Similarly, in Q4, AERO outperforms both baselines, demonstrating its ability to adapt to fluctuations in the cost of predicates introduced by UDF caching. However, it should be noted that Q4 has a slightly higher execution time than Q3, despite utilizing cached results. This increase in execution time is due to caching, which leads to suboptimal GPU utilization and additional lookup overhead.

In Q5, AERO outperforms EVADB and VIVA by 4.86× and 4.75×, respectively. This is due to AERO's ability to adapt to fluctuating predicate statistics and its efficient GPU utilization, achieved through dynamic worker allocation via its LAMINAR component. Specifically, predicates `EmotionClassifier(frame) = 'happy'` and `AgeClassifier(frame) = '20-29'` exhibit significantly varying selectivity across the image dataset; the former showing higher selectivity in earlier images, the latter in later images. These fluctuations negatively impact EVADB and VIVA, leading to suboptimal predicate ordering.

In Q6, AERO outperforms both baselines by a factor of 1.46×, thanks to improved CPU utilization by data-aware load balancing.

QUERY ACCURACY. We compare query results from EVADB and AERO. Both systems produce identical query results, indicating that the optimizations in AERO do not impact the query accuracy. Because UDFs behaviors are deterministic, the execution order of UDFs does not affect query accuracy.

7.3 Impact of Routing Policies

COST-DRIVEN ROUTING. In this experiment, we assess the benefits of the cost-driven routing policy in EDDY. For this micro-experiment, the LAMINAR component is disabled. We examine four variants: the NO REORDERING variant, which avoids predicate reordering during query optimization or execution, and three other variants that utilize different EDDY routing policies: COST-DRIVEN, SCORE-DRIVEN, and SELECTIVITY-DRIVEN. These variants prioritize routing batches based solely on

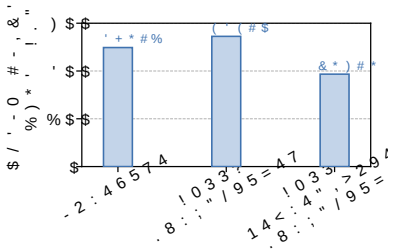
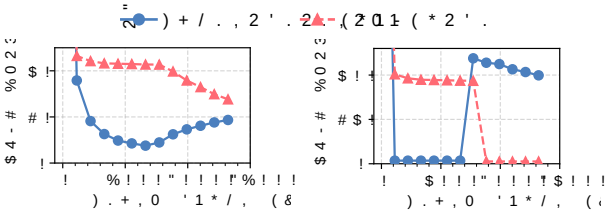


Fig. 9. Adaptive routing strategy impact on Q4 – comparison among baseline (no routing), cost-driven, and REUSE-AWARE cost-driven routing.



(a) Cost-driven. **(b)** REUSE-AWARE cost-driven.

Fig. 10. Average cost of predicates – the average cost of different predicates over video frame ID with different routing policies.

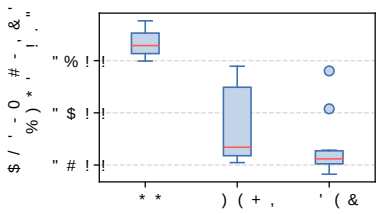
cost, $\frac{\text{cost}}{1-\text{selectivity}}$, and solely on selectivity, respectively. For both SCORE-DRIVEN and SELECTIVITY-DRIVEN routing methods, the EDDY router tracks the number of filtered rows for each predicate and calculates selectivity.

We run Q1 with all these variants and demonstrate the results in Fig. 7. COST-DRIVEN, SCORE-DRIVEN, and SELECTIVITY-DRIVEN routing offer 1.83×, 1.76×, and 1.62× speedup, respectively. Profiling data (Table 1) shows that `DogBreedClassifier` takes 30.35 ms per tuple with a selectivity of 0.06, while `DogColorClassifier` takes 0.95 ms per tuple with a selectivity of 0.15. Given these metrics, both COST-DRIVEN and SCORE-DRIVEN policies should ideally run `DogColorClassifier` first. However, fluctuations in statistics during query execution cause SCORE-DRIVEN to route in a suboptimal order for certain segments of the data. SELECTIVITY-DRIVEN policy prioritizes `DogBreedClassifier` due to its higher selectivity and therefore results in slightly higher query processing time (743.4 s) compared to the other two routing policies.

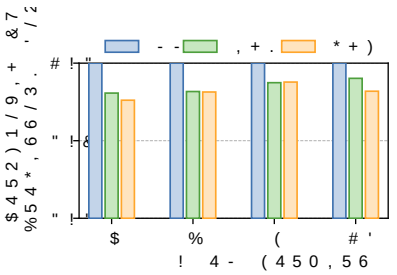
Next, we evaluate the benefits of the COST-DRIVEN routing policy when both EDDY and LAMINAR are enabled. This evaluation provides insights into how the routing policies perform in the presence of parallelization introduced by LAMINAR. We execute queries Q3 and Q5, comparing the execution times of the two policies. As shown in Fig. 8, the COST-DRIVEN policy outperforms the SCORE-DRIVEN policy by 2.3× for Q3 and marginally by 1.12× for Q5. Profiling data (Table 1) indicates that SCORE-DRIVEN prioritizes `HardHatDetector` over `ObjectDetector`, whereas COST-DRIVEN favors `ObjectDetector` due to its lower cost. Running `ObjectDetector` before `HardHatDetector` results in better overlap between the predicate workers, which in turn leads to improved GPU utilization and lower execution times.

ADAPTIVE ROUTING. As discussed in § 4.2, adaptive routing is crucial, especially in the presence of existing optimizations such as reusing results of expensive UDFs. To showcase the effectiveness of adaptive routing, we focus on query Q4 and examine the performance of eddy routing in three scenarios: (i) baseline (no routing), (ii) cost-driven routing, and (iii) REUSE-AWARE cost-driven routing. We disable the LAMINAR component of AERO to isolate the impact of EDDY routing. Fig. 9 illustrates the query processing time for the three scenarios. First, +EDDY REUSE-AWARE cost-driven routing achieves a speedup of 1.29× over the baseline. On the other hand, +EDDY cost-driven routing has a longer query processing time (545 s) than the baseline (498.2).

To understand the results shown in results in Fig. 9, we examine the estimated costs of predicates across the two routing policies over the video (Fig. 10). During routing, predicates with lower estimated costs are prioritized. As depicted in Fig. 9a, the estimated cost of `ObjectDetector` significantly decreases from frame `id > 1000` to `id < 7000` due to caching, making `ObjectDetector` the



(a) Query processing time of different LAMINAR routing policies for Q6 – comparison among LAMINAR with round-robin (RR), queue length-driven (QLEN), and data-aware (DLB) load balancing.



(b) Normalized query processing time with varying numbers of workers for Q6 – comparison between routing policies.

Fig. 11. Data-aware load balancing in LAMINAR

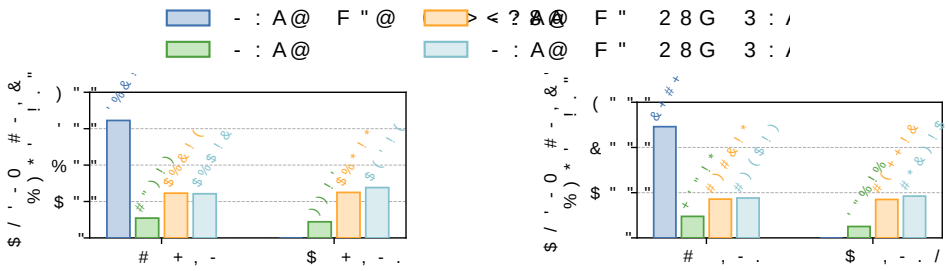
optimal choice for this video segment. However, because cost-driven routing lacks REUSE-AWARE, it cannot adjust the estimated cost for later frames (`id > 8000`). Consequently, it continues to prioritize `ObjectDetector`, resulting in a sub-optimal plan for the latter part of the video.

In contrast, EDDY REUSE-AWARE cost-driven routing adjusts the predicate's estimated cost dynamically for different video segments, as demonstrated in Fig. 9b. Specifically, the estimated cost of `ObjectDetector` is adjusted to a very low value for the range of `id > 1000 AND id < 7000` due to cache. Likewise, the estimated cost of `HardHatDetector` is also adjusted for the later part of the video. Consequently, when EDDY REUSE-AWARE cost-driven routing is enabled, data is consistently routed to the lower-cost predicate, resulting in a better query execution plan compared to EDDY cost-driven routing.

Last, we also observe that the baseline (*i.e.,* no routing) is slightly faster than EDDY cost-driven in Fig. 9. This is because the baseline setting always goes with the static plan that has a fixed predicate order. However, EDDY cost-driven has a startup profiling phase, during which it routes data to all predicates regardless of their actual cost to gather some initial statistics about both predicates, this causes some data to be routed in a sub-optimal order compared to the baseline, so EDDY cost-driven has slightly higher query processing time in this case.

DATA-AWARE LOAD BALANCING. To demonstrate the performance benefits of data-aware load balancing, we focus on Q6. We compare the runtime of AERO on Q6 using three different LAMINAR routing policies: (i) round-robin policy (RR), (ii) queue length-driven load balancing (QLEN) adapted from popular policies [2, 44], (iii) data-aware load balancing policy (DLB). Since query execution involves multiple workers and queues, which can randomize data order and affect performance, we present query processing time based on 10 runs for each policy. As depicted in Fig. 11a, the LAMINAR with RR policy has a higher median query processing time than both QLEN and DLB. While QLEN has a comparable median to DLB, it exhibits greater performance degradation in the worst-case scenario. This is because QLEN only aims to balance the number of pending requests between workers instead of balancing the actual loads, because it does not consider the load of individual requests.

We also evaluate how this benefit generalizes when the number of workers increases with a control experiment shown in Fig. 11b. For this experiment, we statically control the number of predicate workers and their hardware utilization. The result shows that as the number of workers increases, the performance difference between RR and DLB decreases, suggesting that the uneven load is amortized. But, RR still performs worse than the other two policies. Also, we can see in worst case scenarios, DLB is still critical to prevent performance degradation from load imbalance.



(a) Q3 – short video **(b)** Q3 – long video

Fig. 12. Query processing time of Q3 – a comparison with different auto-scaling backends.

7.4 Impact of Laminar on Performance Scaling

We investigate how well Aero does when scaling for better hardware resource utilization. We compare Aero with the Ray Serve model serving framework using its auto-scaling feature.

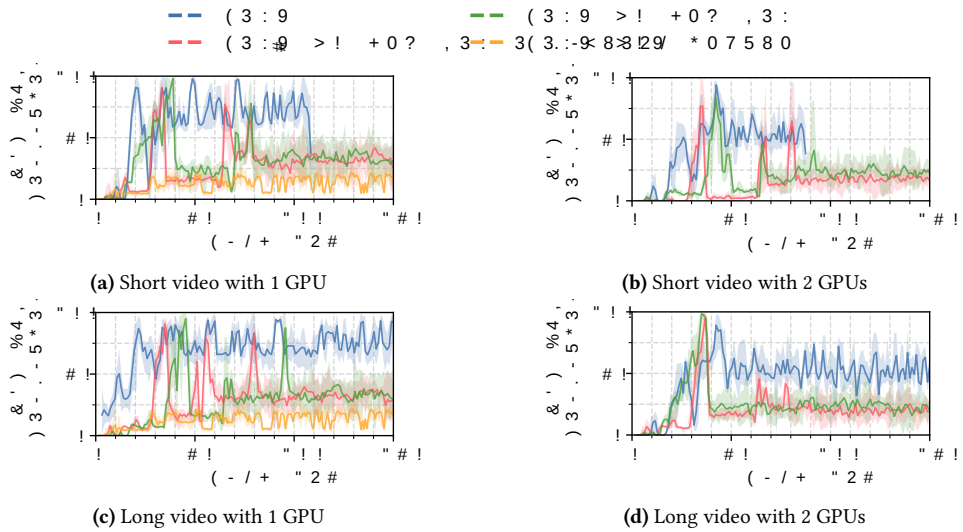
Ray Serve Setup. To assess the performance of our proposed Laminar auto-scaling component, we introduce two variants: Aero w/ Ray Serve [Default] and Aero w/ Ray Serve [Tuned], where Laminar in Aero is replaced with Ray Serve. Aero w/ Ray Serve automatically scales up the number of workers in the cluster in response to congested incoming requests. Aero w/ Ray Serve [Default] uses the default auto-scaling configuration. For Aero w/ Ray Serve [Default], we tune the auto-scaling configuration, where we set the max number of pending requests to 3, the max number of worker replicas to 100, and the target number of pending requests to 1 (*i.e.,* encouraging rapid scaling). Additionally, we create a longer version of Warehouse by duplicating the frames in the video. The new long video has a total of 112912 frames, while the short video has 14114 frames. We omit plotting Aero w/o Laminar in the two-GPU scenario, as the system cannot scale beyond one GPU without Laminar.

We first measure the query processing time for Aero, Aero w/o Laminar, Aero w/ Ray Serve [Default], and Aero w/ Ray Serve [Tuned] on both short and long videos (Fig. 12). Aero consistently delivers better performance than Aero w/o Laminar, Aero w/ Ray Serve [Default], and Aero w/ Ray Serve [Tuned] on both short and long videos. When more GPU resources are available, both Ray Serve variants do not offer better performance. Aero w/ Ray Serve [Tuned] offers slightly better performance than Aero w/ Ray Serve [Default] by scaling faster. Aero provides a 1.3× and 1.9× speedup on short and long videos, respectively, when comparing two GPUs to one GPU.

We also analyze the GPU utilization analysis across different systems (Fig. 13). Aero achieves nearly 100% utilization with a single GPU. However, the utilization drops when Aero scales to two GPUs. On the other hand, the GPU utilization of both Ray Serve variants is lower than Aero. Aero w/o Laminar has the lowest GPU utilization compared to others.

Our findings demonstrate that using Laminar provides greater benefits to the overall system than directly leveraging an existing model serving framework for auto-scaling. One overhead of using a model-serving framework for auto-scaling is the need to offload data from the DBMS to the serving framework in customized formats. This incurs additional data de/serialization overheads, which do not exist if the DBMS provides efficient UDFs execution natively.

Moreover, Ray Serve is designed to meet the user-specified number of pending requests. Its scaling behavior depends on the user's manual configuration. As demonstrated, Aero w/ Ray Serve [Tuned] provides faster scaling and better performance than Aero w/ Ray Serve [Default]. Additionally, due to its reactive nature, Ray Serve experiences significant delays in responding



(a) Short video with 1 GPU

(b) Short video with 2 GPUs

(c) Long video with 1 GPU

(d) Long video with 2 GPUs

Fig. 13. GPU utilization – average GPU utilization of Q3 across different auto-scaling backends on short and long videos.

to a sudden increase in queued requests. For example, in Fig. 13a, AERO w/ RAY SERVE [TUNED] upscales twice, but the second upscaling occurs 70 seconds into the query—a quarter of the total query execution. Lastly, RAY SERVE stops scaling once the target number of pending requests is met, even though the underlying GPU may remain under-utilized. In contrast, LAMINAR focuses on maximizing underlying GPU utilization because it adopts a more proactive mechanism by spawning the maximum possible number of workers to reduce the overall query execution time.

BOTTLENECK STUDY. We further investigate the relatively low GPU utilization for AERO with 2 GPUs by profiling the queue occupancy (Fig. 14). We attribute the cause to a more congested CENTRAL QUEUE when scaling out, as shown in Fig. 14 where the queue occupancy rate with 2 GPUs is higher than running with 1 GPU. In AERO, each predicate worker operates as a single-threaded loop. It retrieves data from its dedicated input queue, processes it, and then inserts the results back into the CENTRAL QUEUE. A congested CENTRAL QUEUE delays the insertion of completed data, preventing the predicate workers from quickly processing the next input. This leads to a longer idle time (*i.e.,* lower average GPU utilization) from the earlier data to the latest one.

7.5 Performance Benefit Breakdown

In this section, we demonstrate the benefits of EDDY and the LAMINAR operator in AERO. We explore three configurations: EVADB, EVADB +EDDY, and EVADB +EDDY &LAMINAR, assessing their impact on performance across queries Q3 and Q5. Fig. 15 illustrates the comparative performance of these setups. For Q3, integrating EDDY improves performance by 1.36× over the baseline EVADB, showcasing its dynamic predicate reordering capabilities. Adding LAMINAR to EVADB +EDDY further accelerates performance by 6×, owing to the effectiveness of the LAMINAR component in optimizing GPU utilization. Similarly, in Q5, EDDY enhances performance by 1.27× compared to EVADB, adapting well to skewed predicate distributions. Extending to EVADB +EDDY &LAMINAR enhances performance by 3.8× over EVADB +EDDY, emphasizing LAMINAR's role in maximizing GPU resources.

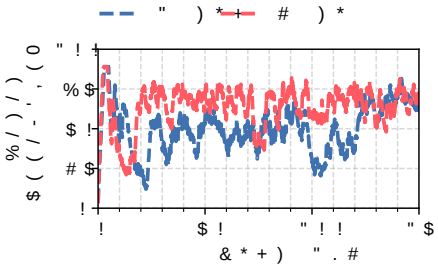


Fig. 14. Bottleneck study of Aero – the average occupancy of Central queue for running Q3 with the long video on 1 and 2 GPUs.

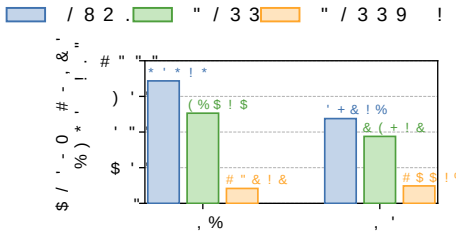


Fig. 15. Performance breakdown – comparative efficiency of Aero variants (EvaDB, EvaDB +Eddy, EvaDB +Eddy & Laminar) on Q3 and Q5.

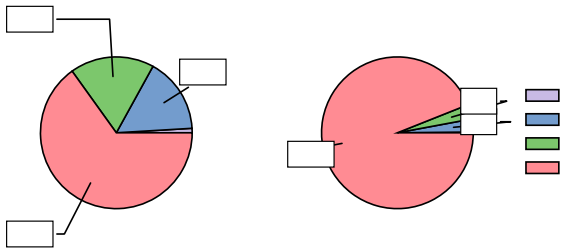


Fig. 16. Query processing time breakdown – time breakdown for static optimization, startup profiling, startup scaling, and execution phases.

7.6 Query Processing Breakdown Analysis

We further conduct query processing time breakdown analysis for Q3 with both short and long videos to understand the significance of each phase in Aero. As other DBMSs, Aero first does rule-based optimization (*i.e.*,, static optimization). Before the actual query execution, it first collects statistics for predicates, deemed as the startup profiling phase, and then it scales out the number of workers for predicates, deemed as the startup scaling phase. After those phases, Aero executes the query. Based on the results in Fig. 16, we notice that time spent on static optimization is almost negligible. For the short video, our previous result (Fig. 12a) demonstrates that Aero does not offer linear scalability when adding more resources (only 1.2× speedup for 2 GPUs). This is attributed to both the startup overheads of Aero, which reflect 34% of query processing time. Nevertheless, both those startup overheads stay constant when the data size increases (*e.g.,* video becomes longer). As shown in Fig. 16, they only account for 6% of query processing time, being amortized out. Thus, for a longer video (Fig. 12b), Aero offers better scalability when adding more GPU resources.

8 Limitations And Discussions

In this section, we discuss limitations and potential mitigations.

Startup Overhead. The evaluation shows that the current approach incurs high startup profiling and scaling overhead. During the startup profiling phase, Aero delays scheduling data to predicates until it collects accurate statistics. A potential solution is to start scheduling data prematurely without waiting for precise statistics. While this approach could reduce startup profiling overhead, it might slow down query execution due to suboptimal predicate ordering. Although a tradeoff is possible, we consider this outside the scope of this paper. During the startup scaling phase, Aero activates additional concurrent predicate workers. As the number of concurrent workers increases,

the time spent on activating them also increases. As shown in (Fig. 16), the overhead is negligible for large sizes (*i.e.,* the overhead is amortized). However, it becomes a bottleneck when the data size is small. A potential solution is to reduce the number of predicate workers to activate for small datasets. While this would lower the startup scaling overhead, it could also slow down query execution. Addressing this requires balancing data size and worker parallelism intelligently.

Queuing Congestion and Delay. Furthermore, the current approach relies on a central queue to route data to different destinations such as predicate workers and output queues. Every batch of data needs to go through the Central queue multiple times before its execution completes. This could become a contention point as the system scales up or out further and the service rate of the Central queue cannot catch up with the processing rate of backend workers, eventually affecting the underlying hardware utilization. Having multiple Central queue can improve its service rate, but the incoordination between different Central queues can lead to sub-optimal predicate ordering for Eddy.

Future Enhancements. Aero currently focuses on applying AQP to provide efficient predicate evaluation. However, the idea of AQP can be generalized to other optimizations like join ordering and choosing the plan execution orders when they involve compute-intensive UDFs. We plan to extend AQP to more complex queries that have UDFs on joins in the future. In addition, while Aero leverages concurrency across workers, it does not currently utilize native GPU concurrency features such as multi-stream, multi-process service (MPS) [4], or multi-instance GPU (MIG) [3]. Multi-stream could enable within-worker concurrency but requires UDF implementation modifications, which Aero avoids. Similarly, MPS and MIG could potentially improve performance through spatial multiplexing. Incorporating these features in future designs could further enhance Aero's performance and scalability.

9 Related Work

Visual DBMSs. In VDBMSs, running deep learning models on every video frame is computationally expensive. To address this, researchers have proposed techniques [8, 35–38, 41, 42, 61] using lightweight, specialized models to reduce the frames processed by the resource-intensive oracle model. Xu *et al.* [60] uses materialized views to store the results of expensive UDFs, thereby accelerating queries in exploratory video analytics settings where queries have overlapping computation. VIVA [51] optimizes queries using user-specified relational hints, such as replacement or filtering hints. It uses these hints to select the optimal query plan that meets an accuracy constraint. ExSample [45] supports distinct object queries and introduces an adaptive sampling algorithm to select frames from video segments likely to contain the object of interest. FiGO [14] harnesses the power of executing queries using a suite of vision algorithms, focusing on selecting the most optimal algorithm through an adaptive query optimization approach. Similarly, Chameleon [32] selects the most optimal algorithm through sliding window-based profiling. Focus [28], Seiden [11], and TASTI [39] explore classical techniques like indexing to speed up video analytics queries. Other works focus on specific tasks, like object tracking [12], out-of-domain vocabulary classification [68], and action detection [19]. These works trade off accuracy for efficiency, while Aero prioritizes system-level optimizations without compromising query accuracy.

Equi-Vocal [66] presents a new interface through which users can find events in the video corpora by providing positive and negative examples. Skyscraper [40] introduces a video extract-transform-load (ETL) problem, focusing on transforming video streams into application-specific formats by applying UDFs during ingestion. Another line of research focuses on building indexes over

precomputed object detections and trajectories to efficiently execute spatio-temporal queries [15–17]. These works assume that the relevant UDFs are known priori, while Aero does not have this requirement but accelerates any ad-hoc queries.

Deep Neural Network Serving. Clipper [21] is a framework that does deep neural networks (DNN) serving equipped with a model abstraction layer. It applies adaptive and delayed batching techniques to improve the throughput of model serving without violating the latency requirement. INFaaS [52] instead specializes in improving the model selection logic for the model serving framework. It automatically determines the model variant and underlying hardware architecture based on performance and accuracy requirements. Inferline [20] monitors runtime traffic and conducts dynamic scaling during execution to save model serving costs.

Clockwork [25] attempts to build a system with predictable latency to reduce tail latency. Scrooge [29] focuses on optimizing the cost of the deep learning inference by allocating just enough resources for inferencing without violating latency constraints. Tensorflow serving [49] and TensorRT inference server [47] are production-grade model serving systems. Nexus [54] proposes squishy bin packing to improve the utilization of GPUs and accurately avoid service-level objective violation. Cocktail [26] proposes to enhance the model selection logic of Clipper and also improves the resource auto-scaling mechanism to ensure requests are handled within SLO. These systems primarily focus on model serving given the user performance and accuracy requirements. This aspect is orthogonal to Aero, which specializes in optimizing long-running analytical queries on a database.

GPU Resource Management and Sharing. Jain *et al.* [31] study existing GPU spatial and temporal sharing mechanisms. To maximize spatial sharing, G-Net [67] proposes offloading network functions to GPU and allows multiple functions to share the GPU resource. Gandiva [59] develops a suspend-and-resume mechanism to allow temporal GPU resource sharing for DNNs, aiming for faster feedback on hyper-parameter tuning. On top of that, Salus [63] proposes a new DNNs execution preemption scheme, achieving a fine-grained GPU time-sharing without the need for data migration from GPU to CPU. Themis [43] focuses on designing a finish-time fair GPU resource-sharing mechanism. GSLICE [23] proposes GPU resource auto-provisioning along with NVIDIA MPS technology [48] to maximize the processing throughput. Choi *et al.* [18] instead study efficiently sharing multi-GPUs resources for model serving. Aero also adopts the idea of spatial sharing, but it focuses on adapting it to the AQP framework.

10 Conclusion

We presented Aero, an adaptive query processing framework tailored for ML queries. Aero eliminates the need to collect UDF statistics during query optimization. Instead, it leverages the Eddy operator to collect statistics during query execution and dynamically routes data to different predicates. Additionally, Aero takes advantage of the Laminar operator to ensure optimal hardware utilization, scalability, and efficient load balancing among multiple backend workers. Our empirical results demonstrate that Aero successfully optimizes query plans across four diverse use cases, achieving a speedup of up to 6.4×.

11 Acknowledgments

This work was supported in part by the U.S. National Science Foundation (IIS-2238431, IIS-2335881), Cisco, Dolby, and Adobe. We thank colleagues in Georgia Tech Database Group for their constructive feedback in improving the system.

References

[1] 2008. Introduction to Queueing Theory. In *Introduction to Discrete Event Systems*, Christos G. Cassandras and Stéphane Lafortune (Eds.). Springer US, 429–497.

[2] 2017. Wayback Machine. https://web.archive.org/web/20170430085805/https://f5.com/Portals/1/Cache/Pdfs/2421/load-balancing-101-nuts-and-bolts-.pdf

[3] 2024. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/

[4] 2024. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/

[5] 2024. Using Third-Party Packages. https://docs.snowflake.com/en/developer-guide/udf/python/udf-python-packages

[6] Ivo Adan and Jacques Resing. 2002. Queueing theory. *Eindhoven University of Technology* 180 (2002).

[7] Yuvanesh Anand, Zach Nussbaum, Brandon Duderstadt, Benjamin Schmidt, and Andriy Mulyar. 2023. GPT4All: Training an Assistant-style Chatbot with Large Scale Data Distillation from GPT-3.5-Turbo. https://github.com/nomic-ai/gpt4all.

[8] Michael R. Anderson, Michael Cafarella, German Ros, and Thomas F. Wenisch. 2019. Physical Representation-Based Predicate Optimization for a Visual Analytics Database. In *ICDE*. 1466–1477.

[9] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *SIGMOD* (2000), 12.

[10] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *SIGMOD*. 107–118.

[11] Jaeho Bang, Gaurav Tarlok Kakkar, Pramod Chunduri, Subrata Mitra, and Joy Arulraj. 2023. Seiden: Revisiting Query Processing in Video Database Systems. *PVLDB* (2023), 2289–2301.

[12] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. 2020. MIRIS: Fast Object Track Queries in Video. In *SIGMOD*. 1907—-1921.

[13] Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. 2005. Content-Based Routing: Different Plans for Different Data. *PVLDB* (2005).

[14] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2022. FiGO: Fine-Grained Query Optimization in Video Analytics. In *SIGMOD*. 559–572.

[15] Yueting Chen, Nick Koudas, Xiaohui Yu, and Ziqiang Yu. 2022. Spatial and Temporal Constrained Ranked Retrieval over Videos. *PVLDB* (2022), 3226–3239.

[16] Yueting Chen, Xiaohui Yu, and Nick Koudas. 2022. Ranked Window Query Retrieval over Video Repositories. In *ICDE*. 2776–2791.

[17] Yueting Chen, Xiaohui Yu, Nick Koudas, and Ziqiang Yu. 2021. Evaluating Temporal Queries Over Video Feeds. In *SIGMOD*. 287–299.

[18] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *ATC*. 199–216.

[19] Pramod Chunduri, Jaeho Bang, Yao Lu, and Joy Arulraj. 2022. Zeus: Efficiently Localizing Actions in Videos using Reinforcement Learning. In *SIGMOD*. 545–558.

[20] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC*. 477–491.

[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. (2017), 613–627.

[22] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Foundations and Trends in Databases* (2007), 1–140.

[23] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *SoCC*. 492–506.

[24] Jocher Glenn. 2020. YOLOv5 by Ultralytics. https://github.com/ultralytics/yolov5

[25] 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. (2020), 443–462.

[26] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *NSDI*. 1041–1057.

[27] Joseph M Hellerstein. 1994. Practical predicate placement. In *SIGMOD*. 325–335.

[28] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *OSDI*. 269–286.

[29] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 2021. Scrooge: A Cost-Effective Deep Learning Inference System. In *SoCC*. 624–638.

[30] Dmytro Iakubovskyi. 2023. Facial Emotion Detection ViT. https://huggingface.co/dima806/facial_emotions_image_detection

[31] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic Space-Time Scheduling for GPU Inference. http://arxiv.org/abs/1901.00041
[32] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodík, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *SIGCOMM*. 253—-266.
[33] Navin Kabra and David J Dewitt. 1998. Mid-Query Re-Optimization of Sub-Optimal Execution Plans. *SIGMOD* (1998).
[34] Gaurav Tarlok Kakkar, Jiashen Cao, Pramod Chunduri, Zhuangdi Xu, Suryatej Reddy Vyalla, Prashanth Dintyala, Anirudh Prabakaran, Jaeho Bang, Aubhro Sengupta, Kaushik Ravichandran, Ishwarya Sivakumar, Aryan Rajoria, Ashmita Raju, Tushar Aggarwal, Abdullah Shah, Sanjana Garg, Shashank Suman, Myna Prasanna Kalluraya, Subrata Mitra, Ali Payani, Yao Lu, Umakishore Ramachandran, and Joy Arulraj. 2023. EVA: An End-to-End Exploratory Video Analytics System. In *DEEM*. 1–5.
[35] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. In *VLDB*. 533—-546.
[36] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. In *VLDB*. 1586—-1597.
[37] Daniel Kang, Edward Gan, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Approximate Selection with Guarantees using Proxies. *PVLDB* (2020), 1990–2003.
[38] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating Approximate Aggregation Queries with Expensive Predicates. *PVLDB* (2021), 2341–2354.
[39] Daniel Kang, John Guibas, Peter D. Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. TASTI: Semantic Indexes for Machine Learning-based Queries over Unstructured Data. In *SIGMOD*. 1934–1947.
[40] Ferdi Kossmann, Ziniu Wu, Eugenie Lai, Nesime Tatbul, Lei Cao, Tim Kraska, and Sam Madden. 2023. Extract-Transform-Load for Video Streams. *PVLDB* (2023), 2302–2315.
[41] Nick Koudas, Raymond Li, and Ioannis Xarchakos. 2020. Video Monitoring Queries. In *ICDE*. 1285–1296.
[42] Ziliang Lai, Chenxia Han, Chris Liu, Pengfei Zhang, Eric Lo, and Ben Kao. 2021. Top-K Deep Video Analytics: A Probabilistic Approach. In *SIGMOD*. 1037–1050.
[43] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *NSDI*. 289–304.
[44] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. 2001. The Power of Two Random Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing*. 255–312.
[45] Oscar Moll, Favyen Bastani, Sam Madden, Mike Stonebraker, Vijay Gadepally, and Tim Kraska. 2020. ExSample: Efficient Searches on Video Repositories through Adaptive Sampling. *arXiv:2005.09141 [cs]* (2020).
[46] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*. 561–577.
[47] NVIDIA. 2020. TensorRT Inference Server. [Online] Available from: https://github.com/NVIDIA/tensorrt-inference-server.
[48] NVIDIA. 2021. NVIDIA Multi-Process Service Introduction. [Online] Available from: https://docs.nvidia.com/deploy/mps/index.html.
[49] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. http://arxiv.org/abs/1712.06139
[50] Nate Raw. 2021. Facial Emotion Detection ViT. https://huggingface.co/nateraw/vit-age-classifier
[51] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. 2022. Optimizing Video Analytics with Declarative Model Relationships. *PVLDB* (2022), 447–460.
[52] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. (2021), 397–411.
[53] Rasmus Rothe, Radu Timofte, and Luc Van Gool. 2018. Deep expectation of real and apparent age from a single image without facial landmarks. *International Journal of Computer Vision* 126, 2-4 (2018), 144–157.
[54] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *SOSP*. 322–337.
[55] Ultralytics. 2023. Ultralytics YOLOv8. [Online] Available from: https://github.com/ultralytics/ultralytics.
[56] Carl A Waldspurger and William E Weihl. 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. *OSDI* (1994).
[57] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 557–572.
[58] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. 2022. Serving and Optimizing Machine Learning Workflows on Heterogeneous Infrastructures. *PVLDB* (2022), 406–419.

[59] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*. 595–610.
[60] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *SIGMOD*. 602–616.
[61] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. *PVLDB* (2022), 2032–2044.
[62] Skylar Yau. 2023. Dog Breed Classifier ViT. https://huggingface.co/skyau/dog-breed-classifier-vit
[63] Peifeng Yu and Mosharaf Chowdhury. 2020. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *MLSys*.
[64] Shan Yu, Zhenting Zhu, Yu Chen, Hanchen Xu, Pengzhan Zhao, Yang Wang, Arthi Padmanabhan, Hugo Latapie, and Harry Xu. 2024. VQPy: An Object-Oriented Approach to Modern Video Analytics. *Proceedings of Machine Learning and Systems* 6 (2024), 279–295.
[65] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* (2016), 56–65.
[66] Enhao Zhang, Maureen Daum, Dong He, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. 2023. EQUI-VOCAL: Synthesizing Queries for Compositional Video Events from Limited User Interactions. *PVLDB* (2023), 2714–2727.
[67] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-NET: Effective GPU Sharing in NFV Systems. In *NSDI*. 187–200.
[68] Yuhao Zhang and Arun Kumar. 2019. Panorama: A Data System for Unbounded Vocabulary Querying over Video. In *VLDB*. 477–491.

Received October 2024; revised January 2025; accepted February 2025