ENHANCE DATABASE QUERY PERFORMANCE THROUGH SOFTWARE OPTIMIZATION AND HARDWARE ADAPTATION

A Dissertation
Presented to
The Academic Faculty

By

Jiashen Cao

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing
School of Computer Science

Georgia Institute of Technology

August 2025

ENHANCE DATABASE QUERY PERFORMANCE THROUGH SOFTWARE OPTIMIZATION AND HARDWARE ADAPTATION

Thesis committee:

Dr. Joy Arulraj
School of Computer Science
Georgia Institute of Technology

Dr. Kexin Rong
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Rathijit Sen

Microsoft Gray System Lab

Dr. Divya Mahajan
School of Electrical and Computer Engineering and Computer Science
Georgia Institute of Technology

Dr. Tarique Siddiqui

Microsoft Research

Date approved: April 25th, 2025

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to Dr. Joy Arulraj for his invaluable mentorship, which has made me an independent researcher. From my very first paper submission, he provided meticulous guidance—reviewing papers word by word and offering detailed feedback on figures. These experiences laid a strong foundation for me.

Secondly, I would like to thank Dr. Hyesoon Kim for her enduring support and guidance throughout my PhD journey. I began as an undergraduate research assistant under her supervision, and it was her encouragement that initially sparked my interest in research. I am incredibly grateful for her understanding and support when I chose to pursue a different research direction during my PhD. Without her continued encouragement, I would not have been able to explore GPU research as deeply as I have.

I would also like to express my sincere gratitude to my committee members, Dr. Divya Mahajan and Dr. Kexin Rong, for their invaluable feedback on my proposal and for helping shape my future research directions. In addition, I am thankful to Dr. Kexin Rong for organizing the database reading group, which provided a welcoming space to exchange ideas.

I am also grateful to Dr. Rathijit Sen for his support and guidance during my internship at Microsoft Gray Systems Lab. It is his dedication and the commitment he made to help me run experiments remotely that make our VLDB paper possible. I would like to express my gratitude to Dr. Tarique Siddiqui for his mentorship during my internship at Microsoft Research. It is his guidance that enables me to contribute meaningfully to an entirely new research domain.

I am thankful to Ramyad Hadidi, Bharadwaj Pudipeddi, Angela Yi for going above and beyond to mentor me during my studies and internships.

My PhD journey would not have been as smooth and pleasant without the support of my friends. I am grateful for all the laughter and memorable moments shared with members of the HPArch and GT Database groups, including Hyojong Kim, Prasun Gera, Bahar Asgari, Yonghae Kim, Seonjin Na, Euijun Chung, Euna Kim, Ruobing Han, Liam Cooper, Nicholas Parnenzini, Saurabh Singh, Anurag Kar, Xueyang Liu, Chihyo Ahn, Andrei Bersatti, Eric Lorimer, Shinnung Jeong, Michael Goldstein, Gaurav T Kakkar, Xinyu Liu, Pramod Chunduri. Special shout-out to Jaewon Lee and Sam Jijina for all the adventures that we shared. I am also especially grateful to Wan Shen Lim for all the chatting and ranting. I want to also thank my high school friend Shenghao Jiang for the constant memeing.

Most importantly, it is my family's unwavering support that has made it possible for me to study abroad. Their sacrifices and understanding have allowed me to fully dedicate myself to research.

Lastly, I want to thank my wife Mengtian Jin for her love, understanding, encouragement, and enlightenment. This journey would be incomplete without you by my side.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables xi

List of Figures xiii

List of Acronyms xviii

Summary xix

Chapter 1: INTRODUCTION 1

1.1 Thesis Statement 1

1.2 FiGO: Fine-grained Query Optimization for ML Queries 1

1.3 GPU Database Characterization and Optimization 2

1.4 Aero: Adaptive Query Processing for ML Queries 2

1.5 Buffer Management for Out-of-GPU LLM Execution 3

1.6 Thesis Outline 3

Chapter 2: BACKGROUND AND RELATED WORKS OF DATABASE SYSTEMS AND GPUS 5

2.1 DBMS Optimization and Execution 5

2.2 Executing Queries on GPUs 7

2.3 Executing ML Queries in DBMSs 9

Chapter 3: FiGO: Fine-grained Query Optimization for ML Queries 11

3.1 Introduction 11

3.1.1 Limitations 12

3.1.2 FiGO 15

3.2 Our Approach 17

3.2.1 Fine-Grained Optimization over Ensemble 17

3.2.2 System Architecture 21

3.3 Video Chunking 22

3.3.1 Variable Chunk Size 22

3.3.2 Sample Size Lower Bound 24

3.3.3 Cost Model 24

3.4 Model Ensemble Pruning 26

3.4.1 Connection to Multi-Armed Bandit 26

3.4.2 Ensemble Pruning via Thompson Sampling 27

3.5 OPTIMIZER Implementation 30

3.6 Evaluation 32

3.6.1 Evaluation Setup 32

3.6.2 End-to-End Performance 34

3.6.3 Complex Queries Performance 36

3.6.4 Impact of Accuracy Threshold 38

3.6.5 Query Optimization Time 39

3.6.6 Plan Optimality 41

3.6.7 Impact of Sampling Rate 43

3.6.8 Generalization to Other Ensembles 44

3.7 Related Work 45

3.8 Conclusion 45

Chapter 4: GPU Database Characterization and Optimization 47

4.1 Introduction 47

4.2 Background of Performance Modeling and GPU-tailed DBMSs 50

4.2.1 Roofline Performance Modeling 50

4.2.2 GPU Database Systems 51

4.3 Experimental Setup 53

4.3.1 Hardware 53

4.3.2 Workloads 54

4.3.3 Profiling Toolchains 55

4.3.4 Warm *vs.* Cold Execution Scenarios 55

4.4 Performance And Resource Analysis 56

4.4.1 Query Execution Performance 57

4.4.2 GPU Resource Utilization 58

4.5 Bottleneck Investigation 62

4.5.1 Operation and Data Efficiency 62

4.5.2 Warp Execution Efficiency 66

4.5.3 Memory Efficiency 68

4.6 Implementation Optimization 68
4.6.1 Inefficiencies and Optimizations 69
4.6.2 Performance Speedup 70
4.7 Resource Utilization Optimization 72
4.7.1 Model-Driven Resource Allocation 72
4.7.2 Model-Driven Concurrent Scheduling 76
4.7.3 Resource Allocation Study 77
4.7.4 Benefit of Concurrent Query Execution 79
4.7.5 Model Accuracy on Degree of Concurrency 80
4.7.6 MIG and MPS Trade-offs 81
4.8 Related Work 81
4.9 Conclusion 83
Chapter 5: Aero: Adaptive Query Processing for ML Queries 84
5.1 Introduction 84
5.2 AERO 87
5.2.1 Design Principles 87
5.2.2 Architecture of AQP Operator 90
5.3 EDDY Operator 91
5.3.1 Cost-Driven Routing 91
5.3.2 Adaptive Routing 96
5.4 LAMINAR Operator 98
5.4.1 Hardware Utilization and Scalability 99

5.4.2 Data-Aware Load Balancing 103

5.5 Implementation 104

5.6 Evaluation 106

5.6.1 Experimental Setup 106

5.6.2 End-to-End Performance 110

5.6.3 Impact of Routing Policies 112

5.6.4 Impact of LAMINAR on Performance Scaling 116

5.6.5 Performance Benefit Breakdown 119

5.6.6 Query Processing Breakdown Analysis 120

5.7 Limitations And Discussions 121

5.8 Related Work 122

5.9 Conclusion 125

Chapter 6: Buffer Management for Out-of-GPU LLM Execution 126

6.1 Introduction 126

6.2 Background of LLMs Architecture and Out-of-GPU LLMs Execution . . . 127

6.2.1 LLM Architecture 128

6.2.2 GPU Memory and Interconnect 129

6.3 Buffer Management Overview 130

6.3.1 Thrashing-Resistant Policies 131

6.3.2 Inconsistent Parameter Access Pattern 132

6.3.3 LIRS-M: MRU-based Low Inter-reference Recency Set Replacement Policy 133

6.4 Implementation 133

6.5 Evaluation 134

6.5.1 Buffering *vs.* Offloading 135

6.5.2 Impact of Buffer Management Policy 136

6.6 Conclusion 138

Chapter 7: Conclusion 139

Appendices 141

Appendix A: Proof of Sample Size Lower Bound 142

Appendix B: Fine-grained Query Optimization Algorithm 145

Appendix C: Buffer Management Eviction Algorithm Details 147

Appendix D: Tensor Sizes Distribution 149

References 151

LIST OF TABLES

3.1 **Query characteristics** – properties of queries and their associated video datasets. We first report the average size of videos and the selectivity of the queries. We then present the average chunk size and standard deviation. We finally report the overall query processing, optimization, and execution time. 31

3.2 **FiGO *vs.* FiGO – Join** – F-1 score and query processing time metrics of: (1) FiGO (evaluating predicates together) and (2) FiGO – Join (evaluating predicates separately). 38

4.1 **Qualitative comparison of GPU database systems** – We consider the following characteristics. **Coverage**: whether the system is general purpose or only specific queries are supported. **Data Format**: formats for data input to the system. **Backend**: how data operators are compiled for execution. **Open Source**: if the system is publicly available. 51

4.2 **Qualitative summary of the queries from SSB** – Key characteristics of queries in SSB: **Number of Joins**, the **Aggregation** type, whether they require **Sorting**, and query **Selectivity**. 54

4.3 **Profiled metrics to construct the roofline model.** 59

4.4 **Summary of most time-consuming kernels** – three most time-consuming GPU kernels for executing query Q21 on all systems. 65

4.5 **GPU scheduler statistics for different warp states** – ACTIVE: total warps allocated. ELIGIBLE: warps ready to issue instruction (includes issued warps). STALLED: warps cannot issue instruction. ISSUED: warps selected by the scheduler. Table units are *average number of warps per scheduler per cycle*. 67

4.6 **GPU memory statistics** - Register usage, cache hit rate and number of cache requests for CRYSTAL and HEAVYDB. 68

4.7 **Performance statistics of CRYSTAL and CRYSTAL-Opt** for Q21—Total bytes, total operations, cache hit rate, number of cache requests. 70

5.1 **Predicate statistics** – average selectivity and cost. 110

LIST OF FIGURES

2.1 **Query execution pipelines** – In static query processing, the predicate order is determined based on statistics estimated during query optimization. In contrast, adaptive query processing dynamically governs the predicate ordering during query execution. 6

2.2 **GPU background and concurrency mechanisms** – NVIDIA GPUs memory hierarchy, and supported concurrency mechanisms. 7

3.1 **Techniques for accelerating queries** – two approaches taken for accelerating queries in state-of-the-art video analytics systems. 12

3.2 **Comparison of video analytics systems** – F-1 score and query processing time associated with handling the query in § 3.1.1 across different video analytics systems. 14

3.3 **Fine-grained optimization over a model ensemble** – illustration of how FiGO processes the query in § 3.1.1. 19

3.4 **FiGO** – system architecture of FiGO. 21

3.5 **Video chunking** – illustration of how the OPTIMIZER chunks a given video in FiGO. 23

3.6 **End-to-end performance** – F-1 and query processing time of all the system across four queries. 35

3.7 **End-to-end performance on complex queries** – F-1 and query processing time of all the system across two complex queries. 37

3.8 **Impact of accuracy threshold** – F-1 score and query processing time metrics under different accuracy thresholds. 39

3.9 **Optimization time** – comparison of optimization overhead of three systems: ME-COARSE, FiGO without model pruning, and FiGO. 40

3.10 **Plan optimality** – comparison of query execution time with respect to that of the FiGO plan across four systems. 41

3.11 **Model usage distribution** – the distribution of usage of models in the ensemble across all queries. 41

3.12 **Impact of sample rate** – comparison of the impact of sample rate on: (1) F-1 score, and (2) query processing time. 43

3.13 **Generalization to other ensembles** – FiGO on different model ensembles. 44

4.1 **Roofline model with different resource bounds.** 50

4.2 **Query execution models** – Data mapping for query execution. 53

4.3 **Cold execution characterization over Q21 at SF=16 in HEAVYDB** – We split the end-to-end performance of cold query execution into six components: *Compute* time, *Host-to-Device* data transfer (HtoD), *Device-to-Host* data transfer (DtoH), *Compilation* time, and *Other* CUDA context setup and memory management time. Similar results apply to other queries and database systems. 55

4.4 **Warm execution characterization** – End-to-end query execution time and GPU execution time for all queries (BLAZINGSQL does not support queries in group 4; those queries are skipped when calculating the average.). The hatched area represents time spent besides the actual GPU execution. 56

4.5 **DRAM roofline model** – for SSB queries (SF=16). 59

4.6 **L2 cache roofline model** – for SSB queries (SF=16). 61

4.7 **GPU execution efficiency** – total integer operations (top) and number of bytes from DRAM (bottom). 63

4.8 **Warp stall causes breakdown** – Q21 for all systems. 67

4.9 **CRYSTAL vs CRYSTAL-Opt. query execution.** 70

4.10 **Resource utilization between CRYSTAL and CRYSTAL-Opt** – DRAM and L2 cache utilization. 71

4.11 **Memory-bound queries** – Performance impact on memory-bound queries (Q11 and Q31) for CRYSTAL. 73

4.12 **Compute-bound queries** – Performance impact on compute-bound queries (Q34 from BLAZINGSQL). 74

4.13 **Throughput *vs.* degree of concurrency** – Comparison between actual and estimated throughput of running queries concurrently (we omit data points for BLAZINGSQL and TQP where they run into out-of-memory issues at larger scale factors). For sub-figures (a), (b), (c), (d), and (e), the top plot shows the actual throughput whereas the bottom plot shows the estimated throughput. 78

4.14 **Trade-offs between MIG and MPS** – Throughput *vs.* degree of concurrency with different mechanisms tested on CRYSTAL-Opt. 81

5.1 **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. 89

5.2 **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. 92

5.3 **Controlled experiment** – speedup comparison of the cost-driven routing policy over other policies under different selectivities. 94

5.4 **LAMINAR workflow for listing 5.1** – 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. 101

5.5 **End to end performance** – a comparison between EVADB, VIVA, and AERO on Q1-Q6. 110

5.6 **Query processing time for Q1 w/o LAMINAR** – comparison of four EDDY routing options: no reordering, cost-driven, score-driven, and selectivity-driven. 112

5.7 **Query processing time for Q3 and Q5 with LAMINAR enabled** – comparison between score-driven and cost-driven EDDY policies under parallelization. 112

5.8 **Adaptive routing strategy impact on Q4** – comparison among baseline (no routing), cost-driven, and REUSE-AWARE cost-driven routing. 113

5.9 **Average cost of predicates** – the average cost of different predicates over video frame ID with different routing policies. 114

5.10 **Data-aware load balancing in LAMINAR** 115

5.11 **Query processing time of Q3** – a comparison with different auto-scaling backends. 117

5.12 **GPU utilization** – average GPU utilization of Q3 across different auto-scaling backends on short and long videos. 118

5.13 **Bottleneck study of AERO** – the average occupancy of CENTRAL QUEUE for running Q3 with the long video on 1 and 2 GPUs. 119

5.14 **Performance breakdown** – comparative efficiency of AERO variants (EVADB, EVADB +EDDY, EVADB +EDDY & LAMINAR) on Q3 and Q5. 120

5.15 **Query processing time breakdown** – time breakdown for static optimization, startup profiling, startup scaling, and execution phases. 120

6.1 **Comparison between offloading and buffering** – buffering allows faster access to data cached in the buffer. 127

6.2 **Internal architecture and execution flow of LLMs** – including dense and MoE LLMs. 128

6.3 **Time breakdown of out-of-GPU execution** – model parameters offloading *vs.* computation for running OPT-30B on a single A40 GPU. 129

6.4 **Buffer hit under different buffer management policy** – the example demonstrates a case in which the working set size of access is much larger than the buffer size and presents limited temporal reuse opportunities. . . . 130

6.5 **Implementation of buffer manager using PyTorch framework** – illustrates how we control the interaction between the PyTorch program with our buffer manager and how the buffer manager controls the data movement in the hardware. 134

6.6 **Llama-30B inference time** – demonstrate the inference time of Llama-30B under different execution settings, including FlexGen, LIRS, LIRS-M, 2Q, MRU, LRU, and FIFO with the option of KVRecomp 135

6.7 **Buffer hit rate of different buffer management policies over different buffer capacity** – buffer hit rate of Oracle, LIRS, LIRS-M, MRU, LRU, and FIFO over buffer capacities from 8 GB to 64 GB. 136

6.8 **Buffer hit rate of different buffer management policies over different buffer capacity** – buffer hit rate of Oracle, LIRS, LIRS-M, MRU, LRU, and FIFO over buffer capacities from 2 GB to 32 GB. 138

D.1 **Tensor sizes distribution** – the CDF of the sizes of tensor for Llama-30B and Qwen-MoE models. 150

LIST OF ACRONYMS

AQP adaptive query processing

DBMS database management system

GPU graphic processing unit

LLM large language model

MIG multi-instance GPU

ML machine learning

MPS multi-process service

SM streaming multiprocessor

StoA state-of-the-art

UDF user-defined function

SUMMARY

As data in a database management system (DBMS) becomes increasingly complex, there is a growing need for sophisticated analytical techniques—such as deep neural networks—capable of extracting meaningful insights. However, these methods are often compute-intensive, necessitating the use of specialized accelerators such as the graphic processing unit (GPU) to achieve practical performance.

Given this trend, modern DBMSs frequently integrate GPU-based query execution. For a good performance, we argue that query planning must be approached judiciously, ensuring that advanced algorithms are employed cautiously so as not to abuse the underlying hardware. Once an optimal plan is devised, the next critical step is to refine its implementation to leverage the GPU's computational resources fully. By combining judicious query planning with optimized execution, we can unlock the full potential of modern hardware accelerators in database systems.

My first research endeavor focuses on improving the query plan to avoid overusing compute intensive algorithms for machine learning-based queries, particularly queries in video DBMSs. FiGO proposes a fine-grained approach where just-enough accurate deep learning algorithms are used to analyze the data without compromising accuracy. To pick the right algorithm, FiGO uses an ensemble of algorithms, evaluates all algorithms on sampled data, and applies the fastest but yet still accurate algorithm to analyze. Aero proposes to apply the classic adaptive query processing (AQP) framework on those emergent queries. It utilizes runtime statistics to dynamically reorder predicates that involve deep learning to minimize the query processing time.

My second research endeavor focuses on improving hardware utilization, particularly GPUs in DBMSs. The GPU database systems characterization and optimization paper analyzes the resource utilization of GPUs during query execution and their performance bottlenecks. Based on the roofline analysis, we propose optimizations to reduce the total data

access and increase the query execution concurrency to improve the performance, delivering better performance than the state-of-the-art systems. As part of Aero, we also extend the AQP framework to also promote the GPU utilization during machine learning (ML) query executions, which previously was not done through the AQP framework. Lastly, to facilitate the recent demand of using the large language model (LLM) in DBMSs, we propose tailored buffer management policy called LIRS-M to better utilize the GPU memory resource for a good performance.

CHAPTER 1
INTRODUCTION

As modern database systems evolve to handle increasingly complex data with more complicated algorithms, achieving high-performance query processing has become very challenging. Traditional query processing pipelines and relational operators are often inadequate when targetting more compute-intensive tasks such as video analytics or large-scale machine learning queries. In parallel, advances in specialized hardware—particularly GPUs—have opened new challenges for accelerating data processing, where the underlying hardware can be easily under-utilized. Thus, making efficient use of GPUs in a database context, especially for queries that integrate machine learning models, requires careful software optimizations and specialized runtime management strategies.

1.1 Thesis Statement

In this thesis, we aim to develop advanced software optimizations and specialized runtime management strategies that fully leverage GPU acceleration in modern DBMSs, enabling high-performance query processing for increasingly complex, compute-intensive workloads such as video analytics and large-scale machine learning.

1.2 FiGO: Fine-grained Query Optimization for ML Queries

FiGO is a query optimization approach that tackles the inefficiencies of existing video analytics systems. Prior systems [1, 2, 3, 4] mostly adopt the idea of using a single specialized model or a cascaded approach that chains multiple models together for the entire dataset. Those systems often suffer from (1) inaccurate specialized models, and (2) poor performance improvement for not selective queries. FiGO instead uses an ensemble of off-the-shelf object-detection models with accuracy and performance tradeoffs. It adopts a fine-grained optimization approach where it splits data into multiple "chunks" and it searches the most optimal model to use by drawing samples from each chunk and evaluating models on those samples, FiGO then provides data analysis using carefully selected just-enough accurate models from the ensemble for every chunk. By applying the most cost-effective model per chunk (or sometimes skipping chunks entirely), FiGO shows it can retain high accuracy while significantly reducing runtime. To keep the query optimization overhead to its minimum, it also introduces a novel technique for model pruning based on Thompson sampling to ensure that optimization overhead does not outweigh execution gains.

1.3 GPU Database Characterization and Optimization

§ 4 offers a comprehensive characterization of GPU-based database systems. It applies a roofline analysis to characterize the resource utilization during query execution. It then attempts to explain the reason for underutilization and pinpoints the performance bottlenecks for popular systems (*e.g.,* Crystal and HeavyDB) using detailed performance metrics (*e.g.,* warp stall causes, cache hit rates, instruction mixes). Drawing on the insights from this analysis, we propose new implementation optimizations—such as reducing data movement and fine-tuning concurrency settings—to deliver marked performance gains over state-of-the-art (StoA) GPU database engines.

1.4 Aero: Adaptive Query Processing for ML Queries

Combining our vision of both software optimization and hardware resource optimization, we continue the work on Aero. Our analysis of previous systems reveals two findings. First of all, the predicate ordering during query execution is critical for performance, which is typically determined using estimated predicate statistics, always leading to suboptimal ordering and poor performance. Secondly, ML queries often need accelerator like GPUs, but the query system alone is challenging to maximize out the utilization of underlying hardware because that often requires runtime statistics to achieve. Thus, we propose Aero, which is a system built on top of classic adaptive query processing (AQP) strategies (*i.e.,* eddy routing) to accelerate ML-based queries. We adopt the idea of AQP to collect accurate runtime statistics about predicates to determine their optimal ordering and resource allocation. Within this, we propose a cost-driven routing policy, that evaluates not just operator selectivity but also operator execution time, a key concern for expensive ML inferences. We also introduce a laminar operator, which dynamically scales the degree of parallelism for ML models during query execution to maintain high GPU utilization. Through these strategies, Aero can significantly reduce query latencies for tasks involving video frames or other data with expensive ML predicates.

1.5 Buffer Management for Out-of-GPU LLM Execution

Lastly, § 6 attempts to accelerate the large language models (LLMs) analytics, where the size of the LLM cannot fit into a single GPU. Prior work proposes an offloading technique for this specific scenario, but our finding indicates the offloading technique significantly under-utilizes the GPU memory resource, leading to suboptimal performance. We instead adopt a buffer management mechanism for this use case. We propose a new buffer management replacement policy to ensure a high cache hit rate on critical model parameters, thereby reducing transfer overhead. The policy named LIRS-M (a modification of the Low Inter-reference Recency Set policy) works particularly well for inconsistent parameter access patterns—common in modern LLMs—and offers better performance than standard caching eviction policies like LRU or FIFO.

1.6 Thesis Outline

In § 2, we first introduce the necessary background about database systems (query optimization and execution details), GPU hardware, and DBMSs that specialize in video analytics and large-scale machine learning analytics. We then go over prior works that investigate similar research problems as we and motivate the need of building newer systems. § 3 presents FiGO, a fine-grained query optimization approach to accelerate video analytics without comprising accuracy. § 4 presents our work of investigating resource underutilization problems in GPU DBMSs and our endeavor to improve their performance. § 5 present Aero, which combines optimization from both software optimization and resource management into a uniform adaptive query processing framework and delivers better performance. Lastly, § 6 targets emergent LLM analytics and applies the idea of an innovative buffer replacement policy to speed up its process.

CHAPTER 2

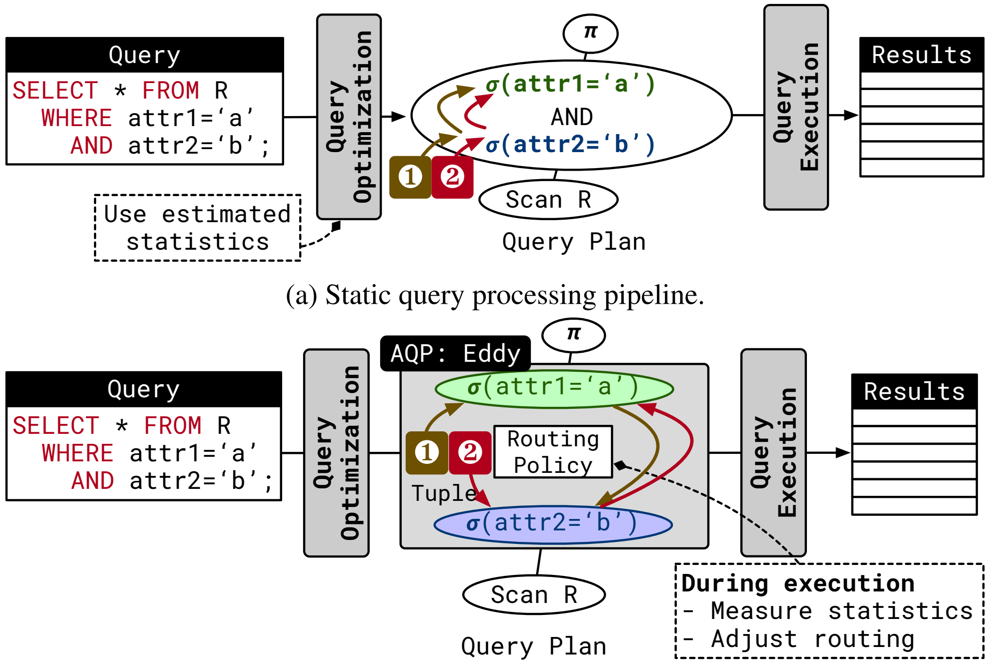
BACKGROUND AND RELATED WORKS OF DATABASE SYSTEMS AND GPUS

2.1 DBMS Optimization and Execution

DBMSs are widely used in many applications. To process a query in a DBMS, it often involves two steps: (1) optimizing the query execution plan; and (2) executing the query based on the optimized query plan. Additionally, to process complex logic that is not supported in SQL syntax, most DBMSs also allow users to supply a user-defined function (UDF) as part of the query. In this section, we explain the query processing mechanisms first. We then introduce how ML functions are typically integrated into DBMSs.

Static vs. Adaptive Query Optimization 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. 2.1a. 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 [5, 6, 7, 8, 9] have proposed AQP that leverage runtime statistics to adjust the query plan.

EDDY– *A Framework for* AQP. Among these efforts, EDDY [7] 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. 2.1b, 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



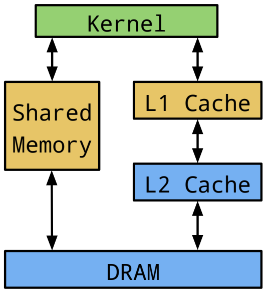
(a) Static query processing pipeline.

(b) AQP query processing pipeline.

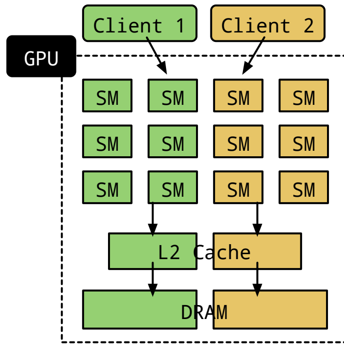
Figure 2.1: **Query execution pipelines** – In static query processing, the predicate order is determined based on statistics estimated during query optimization. In contrast, adaptive query processing dynamically governs the predicate ordering during query execution.

for optimal routing. In the original paper [7], 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 [10] to infer the selectivity of each inner operator.

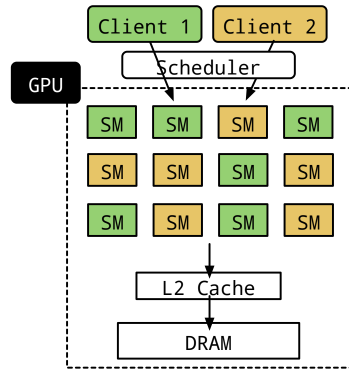
***Content-based* AQP.** Bizarro *et al.* [8] observed that the routing based on average statistics, as suggested in the EDDY paper, can be significantly improved using data or content-based routing. Their mechanism is built on top of the EDDY adaptive query processing framework. The key idea is that 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



(a) **GPU Memory hierarchy.**



(b) **MIG** – resource partitioning for multi-instance GPU.



(c) **MPS** – resource partitioning in multi-process service.

Figure 2.2: **GPU background and concurrency mechanisms** – NVIDIA GPUs memory hierarchy, and supported concurrency mechanisms.

can outweigh the execution time of the relational operators.

UDFs in DBMSs. 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 [11, 12, 13] further enable easy integration with ML algorithms through Python UDF interfaces. In later of this section, we will go over other specialized DBMSs that leverage the power of UDFs to process rich data including images and videos.

2.2 Executing Queries on GPUs

As the GPU hardware becomes more general-purpose and queries become more computationally intensive, there is a growing need of running queries on GPU hardware. Many relational operators can be easily implemented to use GPU hardware. On top of that, many complex UDFs, especially those that require running ML algorithms, need to be executed on GPU hardware to have a good performance. In this section, we go over the details of GPU hardware and its resource management mechanisms.

GPU Architecture. As Fig. 2.2a shows, kernel execution can access data stored either in the shared memory or in the L1 cache within a GPU. Shared memory is fully managed by the user, whereas the L1 cache is managed by the hardware. Each GPU streaming multiprocessor (SM) has a private L1 cache and shared memory region, but the L2 cache and DRAM are shared across all GPU SMs. The memory hierarchy is consistent for all NVIDIA GPUs, but the specific values for capacity and bandwidth vary across GPUs.

GPU Resource Management and Sharing. NVIDIA GPUs support two ways for allocating resources on the GPU, and providing concurrent GPU execution capability to processes: multi-instance GPU (MIG) [14] and multi-process service (MPS) [15]. MIG is a new feature supported only on the A100, A30, and H100 GPUs [16].

MIG enables physical partitioning of GPU resources—SMs, L2 cache, DRAM capacity, and bandwidth—which creates full isolation between concurrent processes. Fig. 2.2b shows an example of resource allocation through MIG to support two concurrent clients with equal allocation ($\frac{1}{2}$ GPU resources) in this example. MIG also supports heterogeneous resource partitions to meet the varying needs of different clients. At its finest granularity, MIG currently supports up to seven concurrent clients: each partition gets about $\frac{1}{8}$ of compute and memory resources, while nearly $\frac{1}{8}$ of the resources is reserved for the MIG controller. MIG currently offers a total of 18 choices for resource partitions on the A100.

MPS employs a logical resource partitioning to support concurrent execution. In previous GPU generations, MPS allows only time-sharing of the GPU. Since Volta, MPS allows concurrent execution through lightweight partitioning of SMs. In MPS, the L2 cache and DRAM are still unified resources without any isolation. When MPS starts, the GPU creates a scheduler to manage resource allocation among different CUDA applications, ensuring that each application receives the requested resources. MPS also allows specifying memory capability limits since CUDA 11.5 [17].

2.3 Executing ML Queries in DBMSs

We first go over systems that support running ML queries. One common example is visual DBMSs. VDBMSs specializes in processing visual data like images and videos. In VDBMSs, running deep learning models on every video frame is computationally expensive. To address this, researchers have proposed techniques to accelerate the computationally expensive portion of the system, where either better query optimization is done to use less computationally intensive but still accurate models or to use system techniques like filtering irrelevant data or caching results.

PP [2], BLAZEIT [4], and NOSCOPE [1] accelerate queries using model specialization. They construct lightweight models in an ad-hoc manner to answer the query. To choose which specialized model to use, they evaluate the models on a set of sampled frames during query optimization.

TAHOMA [3] is another VDBMS variant. It constructs a model cascade by combining a chain of models (*e.g.,* image classification or object detection models) and determines when to short-circuit the inference based on the confidence score of the prediction of each model in the chain. TAHOMA speeds up queries by skipping compute-intensive models that appear at the end of the chain. PANORAMA [18] is another state-of-the-art VDBMS that uses a single cascaded model to solve the unbounded vocabulary problem in object recognition. Similar to TAHOMA, it offers a set of performance-accuracy tradeoffs. The model generates multiple feature embeddings for an input with different levels of quality. PANORAMA determines the appropriate performance-accuracy tradeoff point based on the delta between the representative embedding of a category and the embedding of the input. MIRIS [19] is a VDBMS focused on multi-object tracking. It uses a recurrent neural network and a graph neural network to mark the tracking trajectories between objects. MIRIS also adopts a fine-grained approach to tuning the tracking accuracy. It starts sampling at a low frame rate to gain a high-level perspective of the video at the beginning and then gradually increases the sampling rate to improve the accuracy of tracking. Chameleon [20] selects the most optimal algorithm through sliding window-based profiling.

CHAPTER 3

FIGO: FINE-GRAINED QUERY OPTIMIZATION FOR ML QUERIES

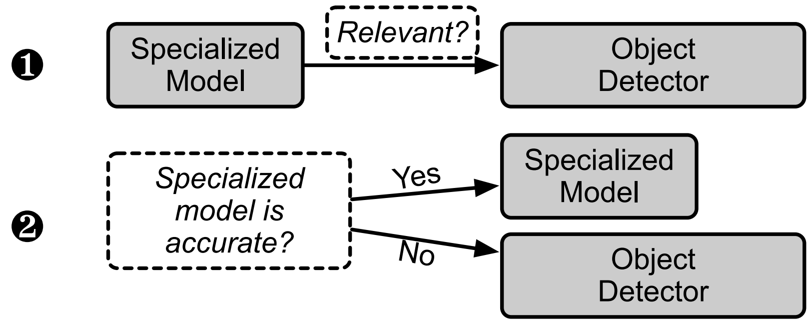
3.1 Introduction

By leveraging recent advances in deep learning (DL), video database management systems (VDBMSs) enable automated analysis of videos at scale. These systems use 2-D object detection models for locating objects of interest in the videos. The runtime performance of these systems is constrained by the computational overhead associated with processing the frames using the DL model. For example, the Det-7 model from the EFFICIENTDET family of object detectors [21] only processes 4 frames per second on Titan XP GPU.

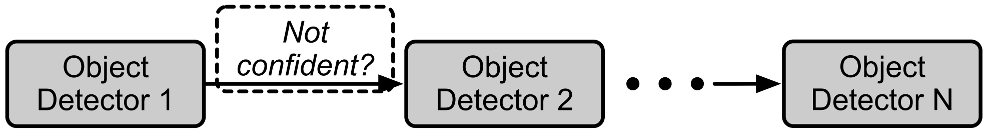
To speed up query execution, researchers have proposed two techniques for reducing the invocation of the heavyweight, object detection model (*a.k.a.,* reference model [1]) in the VDBMS. The first technique consists of using a lightweight, specialized model [2, 4, 1] for quickly filtering out irrelevant frames. The second technique consists of using a sequence of lightweight models [3]. We next describe these two approaches in detail:

Model Specialization (MS). This technique consists of using a lightweight, specialized model to accelerate query processing. As shown in Fig. 3.1a, a VDBMS may use a specialized model to filter out irrelevant frames or to even directly answer the query. By reducing the number of invocations of the heavyweight, reference model, the VDBMS accelerates the query with a tolerable drop in accuracy. Two exemplars of this technique are PP [2] and BLAZEIT [4]. PP filters out irrelevant frames using the specialized model (❶ in Fig. 3.1a). The frames that are considered relevant are subsequently processed using the reference model. We refer to this technique of using specialized models as MS-FILTER. In contrast, BLAZEIT uses the specialized model to directly answer the given query (❷ in Fig. 3.1a). It

[1] We refer to the most accurate (and often also the most compute-intensive model) in the model pipeline as the reference model.



(a) **Model specialization** – configurations associated with MS-FILTER and MS-SKIP.



(b) **Model cascade** – configuration associated with MC.

Figure 3.1: **Techniques for accelerating queries** – two approaches taken for accelerating queries in state-of-the-art video analytics systems.

only uses the reference model if the specialized model is not accurate enough. We refer to this technique of using specialized models as MS-SKIP.

Model Cascade (MC). Another approach for efficiently processing video analytics queries consists of using a sequence of models, called a model cascade, as shown in Fig. 3.1b. While processing the query, the VDBMS short-circuits the inference based on the feedback returned by each model (*e.g.,* confidence score). An exemplar of this technique is the Tahoma system [3].

3.1.1 Limitations

While these techniques successfully accelerate queries, they still suffer from four limitations. We highlight these limitations using the following illustrative query:

```
SELECT frameID FROM UA-DeTrac WHERE Count(Bus)≥1;
```

(1) – Model Specialization Overhead. In MS-FILTER, since each filter detects only one object category (*e.g.,* bus), the VDBMS needs to train several specialized models (*i.e.,* filters) at runtime. In contrast, with BLAZEIT, since a specialized model directly returns the number of buses in an image, it must maintain a separate model for each predicate (*e.g.,* `Count(Bus)`). Thus, with model specialization, VDBMSs need to train and maintain a large collection of models for different objects and predicates, respectively. The overhead of training a specialized model while processing a previously unseen query is significant. For instance, training a ResNet-34 model on 100 sampled video frames takes 32 seconds.

(2) – High Selectivity Query. The speedup obtained with the filtering technique used in MS-FILTER relies on the data reduction rate. Consider a video with N frames. Let the fraction of frames discarded by the specialized model be r, and the costs of running the filter and running the reference model be C_f and C_o per frame, respectively. To accelerate the query, r must satisfy this constraint:

$$N(C_f + (1 - r) \cdot C_o) < NC_o$$
$$r > \frac{C_f}{C_o}$$

This constraint is not met by queries with high selectivity. To illustrate this problem, we replicate MS-FILTER using a model from the EFFICIENTDET family of object detectors [21]. We defer a description of the empirical setup to § 3.6. The results are shown in Fig. 3.2. MS-FILTER takes more time to process queries with high selectivity (*e.g.,* r = 0.2). Thus, when the data reduction rate is low, the performance gap between MS-FILTER and NAIVE (*i.e.,* naively running object detector on every frame) is minimal.

We replicate the model cascade (MC) approach using eight models from EFFICIENTDET. This approach also does not work well on queries with high selectivity. This is because a large fraction of video frames *cannot* be filtered out by earlier models in the cascade leading to slower query processing.

(3) – Difficult-to-Detect Predicates. MS-SKIP [4] uses a specialized model to directly return aggregates (*e.g.,* number of cars in an image). This approach does not generalize to all predicates. First, the specialized model is designed to be shallow for faster execution.

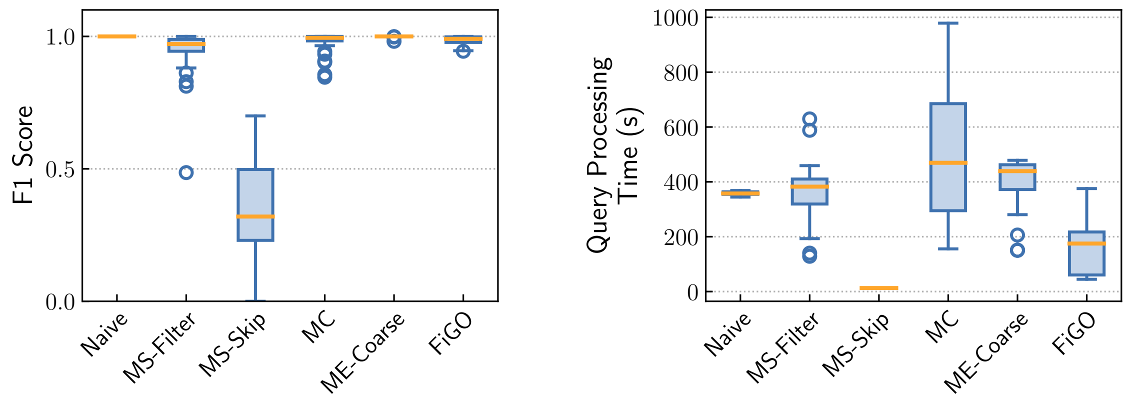


Figure 3.2: **Comparison of video analytics systems** – F-1 score and query processing time associated with handling the query in § 3.1.1 across different video analytics systems.

So, it is unable to deliver high accuracy for harder predicates (*e.g.,* small objects in the background of the frame). Second, it relies on a subset of the videos for training. Lack of positive examples in the selected subset greatly affects the quality of the model. We replicate MS-SKIP by training a ResNet-34 model [22]. As shown in Fig. 3.2, MS-SKIP is faster than NAIVE. But, its F-1 score is worse than its counterparts. While it delivers 0.7 F-1 score on some videos, it fails to provide useful results on others.

We observe that the model cascade approach also suffers from accuracy loss. First, short-circuiting based on confidence scores does not provide a reliable way to achieve good accuracy. Second, the confidence score thresholds fail to generalize to the entire video.

IV – Coarse-Grained Optimization. Another drawback of state-of-the-art DBMSs is that they adopt a coarse-grained approach towards finding the *model configuration* (*e.g.,* a filter followed by the reference model or a model cascade) to process the query. We refer to the chosen configuration as a *query plan*. For instance, the OPTIMIZER may uniformly sample 10% of the frames from the video and select a fixed configuration for the entire video. Given an accuracy constraint, coarse-grained optimization leads to higher query processing time. Positive events tend to not appear in every segment of the video, especially with low-selectivity queries. If the OPTIMIZER picks the same query plan for the entire video, video segments that are less likely to contain positive events or those that contain easy-to-detect events are processed using the same slow model configuration tailored for important segments.

3.1.2 FiGO

We present FiGO (**Fi**ne-**G**rained query **O**ptimization in video analytics), a VDBMS that addresses the limitations highlighted above using a novel model ensemble approach together with fine-grained optimization. In FiGO, the OPTIMIZER first splits the given video into a sequence of *chunks* of varying sizes. It then picks an appropriate model from a collection of models for each chunk. Unlike the model cascade approach, the EXECUTION ENGINE in FiGO only processes a chunk with one model picked from the ensemble.

Prior efforts do not focus on query optimization across a model ensemble. This is critical since there are several suitable models for a given vision task. For instance, Faster-RCNN [23] and SSD [24] object detectors offer different performance-accuracy tradeoffs. We present a novel technique for optimizing queries using a collection of models. Another limitation of the model specialization and cascade approaches is that they require modifications to the neural network's architecture or the model configuration. In contrast, FiGO uses off-the-shelf models to accelerate queries. It does not train any specialized models to process ad-hoc queries. This enables it to seamlessly work across diverse queries and video datasets.

FiGO takes a fine-grained approach to query optimization. Since the content of a video often changes, the optimal model for processing the chunk also varies. FiGO tailors the plan for each chunk. It uses slower, more accurate models for processing important chunks. It either skips or applies faster, less accurate models for processing irrelevant chunks. FiGO uses a novel accuracy-driven sample size bound to determine the sizes of chunks [2]. The OPTIMIZER initially treats the entire video as a single chunk and then iteratively splits it into smaller chunks depending on the accuracy constraint.

[2] Like other VDBMSs, we measure accuracy with respect to the most accurate model (*a.k.a.,* reference model) in the ensemble.

We illustrate the benefits of the fine-grained approach by constructing a baseline that couples a model ensemble approach with coarse-grained optimization. In this case, the VDBMS picks a single optimal model from the ensemble to process the entire video. We refer to this approach as ME-COARSE. As shown in Fig. 3.2, to meet the target accuracy constraint, ME-COARSE often picks a compute-intensive model, even though many segments in the video does not need such a model. This leads to a higher query processing time on most of the videos. In contrast, FiGO tailors the model configuration for each chunk in the video, thereby delivering lower query processing time while still meeting the accuracy constraint.

Coupling a model ensemble approach with fine-grained optimization leads to a higher query optimization time. The OPTIMIZER must profile the sampled frames using all the models in the ensemble. To lower this optimization overhead that prior VDBMSs do not suffer from, we introduce a novel technique for pruning out a subset of models with limited utility. We demonstrate that FiGO outperforms state-of-the-art VDBMSs across diverse queries and video datasets with respect to both accuracy *and* performance.

Contributions. FiGO makes the following contributions.

- FiGO adopts a novel model ensemble approach coupled with fine-grained optimization to accelerate query processing.
- FiGO leverages an accuracy-driven bound for chunking the given video that strikes a balance between query optimization time and query execution time.
- FiGO prunes the set of models using a variant of Thompson sampling to reduce optimization time.
- FiGO is $3.3\times$ faster on average compared to the state-of-the-art VDBMSs across diverse queries. It generalizes to four different video datasets by not relying on ad-hoc, specialized models.

3.2 Our Approach

In this section, we first describe how FiGO couples fine-grained optimization with a model ensemble approach in § 3.2.1. We discuss the importance of adaptively changing the optimal model for each video chunk in lowering query processing time and improving accuracy. We conclude with an overview of FiGO in § 3.2.2.

3.2.1 Fine-Grained Optimization over Ensemble

Chunk. A chunk consists of a contiguous sequence of video frames. We denote a chunk by V_i. Chunks may vary in their size (*i.e.,* number of frames). The maximum size of a chunk is the size of original video V. A video V with R chunks may be specified as:

$$|V| = \sum_{i=1}^{R} |V_i|, \quad 0 < |V_i| \leq |V|$$

Two key design decision in FiGO are: (1) it operates at the chunk-granularity for both optimization and execution. (2) it always picks a *fixed* number of samples from a chunk for query optimization. To obtain more samples in an interesting part of the video, it splits that part into a larger set of fine-grained chunks. Similarly, it maps an uninteresting part of the video to a single, coarse-grained chunk. The OPTIMIZER uses sample size estimation and cost estimation to determine how to split the video into a sequence of chunks (§ 3.3).

Query Execution. The EXECUTION ENGINE uses a model ensemble M with $|M|$ models (*e.g.,* object detectors). Unlike the model specialization approach, there is no online training overhead in FiGO since it does not construct ad-hoc models. Unlike the model cascade approach, the EXECUTION ENGINE only processes each chunk with exactly one model. FiGO delivers lower query processing time as it only uses the optimal model for processing the chunk. The EXECUTION ENGINE completely skips processing irrelevant chunks (*i.e.,* does not run any of the models in the ensemble over them). Thus, FiGO accelerates queries by processing less interesting chunks using faster models and skipping irrelevant chunks.

Query Optimization. The OPTIMIZER is responsible for picking the optimal model configuration for each chunk. In FiGO, the query plan consists of exactly one model or skipping the chunk. Unlike other VDBMSs, FiGO splits the video into a set of chunks. The size of each chunk depends on: (1) the estimated number of required samples, and (2) the estimated execution time for the chunk. After splitting, the OPTIMIZER evaluates potential plans for each chunk. Given a chunk, it first picks a few samples $\hat{V}_i$ and then evaluates each model m' from the ensemble M over $\hat{V}_i$ to obtain accuracy and processing time metrics $f(x, m')$:

$$\hat{V}_i \rightarrow \{ f(x, m') \mid x \in \hat{V}_i, m' \in M \}$$

For simplicity, FiGO uses uniform sampling during this *profiling* step. Lastly, the OPTIMIZER constructs the set of optimal plans P for all of the chunks:

$$P \sim \{ [V_1, m_2], [V_2, m_1] \ldots [V_R, m_3] \}$$

Since the OPTIMIZER needs to profile multiple models during optimization, the profiling step increases query optimization time. We will later present a model ensemble pruning technique for lowering this overhead.

Case Study. We illustrate how FiGO processes the query in Fig. 3.3. In this example, we assume the VDBMS only has access to an ensemble with three models (*e.g.,* Det-0, Det-1, and Det-2 from EFFICIENTDET [21]). Det-0 is the fastest model and Det-2 is the most accurate model (*a.k.a.,* the reference model). We use the same query that checks for the existence of a bus object in a frame.

The OPTIMIZER must make two decisions. First, it must determine how to split the video into chunks. Second, it must decide which model to use for each chunk or to skip that chunk. To make these decisions, the OPTIMIZER always picks the same number of video

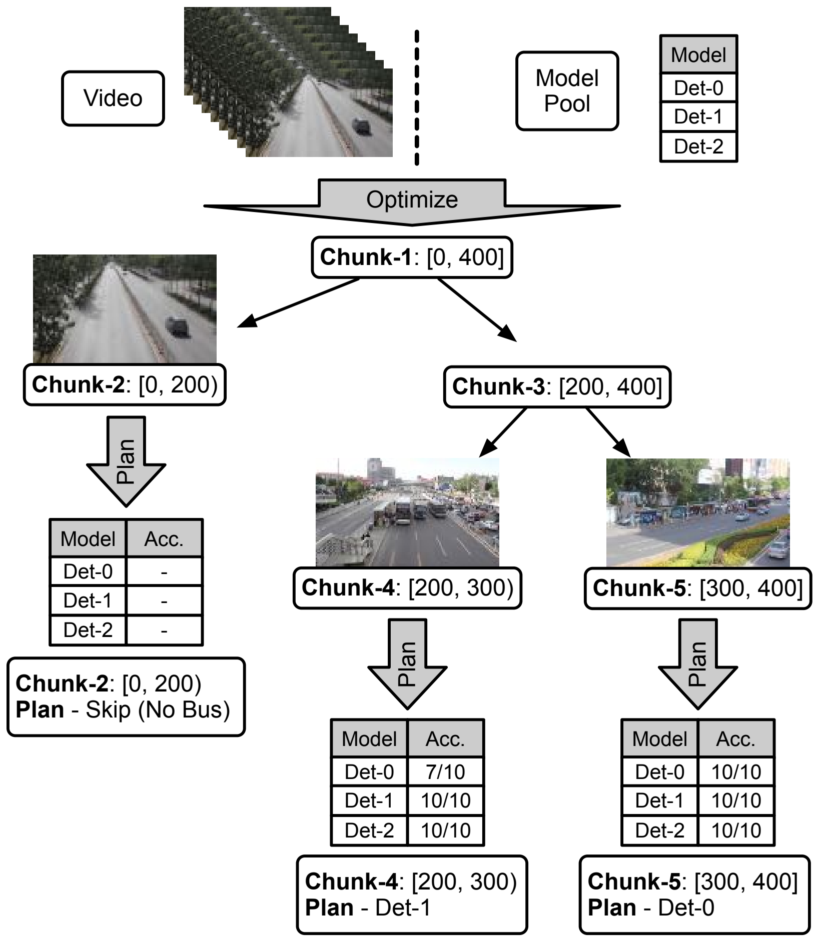


Figure 3.3: **Fine-grained optimization over a model ensemble** – illustration of how FiGO processes the query in § 3.1.1.

frames from each chunk (denoted by λ, *e.g.,* $\lambda = 10$ frames). It then estimates the number of required samples to find the optimal model (elaborated in § 3.3). After confirming that the sample size in a given chunk is sufficient, it picks the model to use based on the accuracy of each model over the sampled frames from that chunk. Like other VDBMS, we define the accuracy of a given model m based on the consensus between m and the reference model (typically also the most computationally-expensive model).

In the example shown in Fig. 3.3, the OPTIMIZER first splits the video into two chunks: $[0, 200]$ and $[200, 400]$. The OPTIMIZER next determines to further split the $[200, 400]$ into two chunks, based on the sample size bound and the cost model (elaborated in § 3.3). The profiling results for each chunk are shown in the tables. For example, in the table associated with chunk-4, the accuracy of Det-0 is $\frac{7}{10}$ (*i.e.,* it agrees with Det-2 on seven out of ten frames). Based on collected metrics, the OPTIMIZER constructs the query plan.

Assume that the $[0, 200]$ chunk does not contain any bus. So, all the models do not find a bus object in any of the sampled frames. Then the OPTIMIZER decides to completely skip processing the other 190 frames in this chunk. Assume that the $[200, 300]$ chunk contains a bus in the background that is difficult to detect. For this chunk, the accuracy of the Det-0 model is lower than that of Det-1 and Det-2. Det-1 provides the same accuracy as Det-2 but is faster than Det-2. So, the OPTIMIZER picks Det-1 for processing the remaining frames in this chunk. Assume that the $[300, 400]$ chunk contains a bus in the foreground that is easy to detect. For this chunk, Det-0 is sufficient to accurately detect the existence of the bus. Hence, the OPTIMIZER picks Det-0 for processing the rest of this chunk.

Theoretical Cost. We now provide a theoretical analysis of the benefits of fine-grained optimization over other approaches by quantifying only the execution time without the optimization overhead. To simplify our analysis, we assume that the VDBMS only uses two models $m1$ and $m2$. Let us denote the cost of running these models by C_{m1} and C_{m2}, respectively (where $C_{m1} < C_{m2}$). In the given video, let us denote the length of the video that does not contain any positive events by α. Let the length of video that may be correctly processed by either model be β. Let the length of video that may only be correctly processed by the most accurate model $m2$ be γ. Then, the execution time of FiGO to process this video is:

$$\beta C_{m1} + \gamma C_{m2}$$

This is because the OPTIMIZER skips chunks that are unlikely to contain positive events. In addition, it picks $m1$ to process the chunks that are easier to analyze.

With the model specialization approach, we only obtain the execution cost of the filtering technique since it is hard to guarantee the accuracy of a specialized model that directly answers the given query (*i.e.,* MS-SKIP). The theoretical execution time of VDBMS that

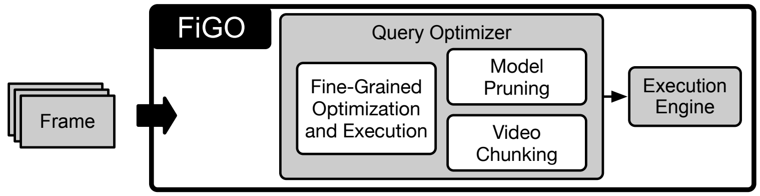


Figure 3.4: **FiGO** – system architecture of FiGO.

uses a specialized model for filtering out irrelevant frames is:

$$\beta C_{m2} + \gamma C_{m2} + \epsilon$$

This is because MS-FILTER is likely to use $m2$ to process the entire video to obtain higher accuracy. The specialized model in front of $m2$ adds computational overhead (ϵ). The value of ϵ depends on the specialized model. We observe that this cost is already higher than that of FiGO as this approach does not leverage both models.

With the model cascade approach, the execution cost is:

$$\alpha C_{m1} + \beta C_{m1} + \gamma (C_{m1} + C_{m2})$$

This is because a VDBMS using a model cascade approach coupled with coarse-grained optimization never skips processing a chunk. Furthermore, with a model cascade, the VDBMS always runs the faster model before running the slower model, which increases the execution time associated with the γ subset of the given video.

3.2.2 System Architecture

We now present an overview of the system architecture of FiGO. As shown in Fig. 3.4, the OPTIMIZER is responsible for chunking the video and pruning the ensemble to lower optimization time.

Video Chunking. Video chunking consists of two parts: (1) required sample size estimation, and (2) cost estimation. The OPTIMIZER must pick the optimal model to process a given chunk. By estimating the number of required samples, the OPTIMIZER determines whether it must further split the chunk based on the given sample size bound. A query plan may be sub-optimal with respect to query execution time even if it meets the target accuracy. The OPTIMIZER relies on a cost model to estimate the execution cost of a plan (*i.e.,* the selected model for a particular chunk). If the execution cost is higher than the cost of additional chunking, the OPTIMIZER further splits the chunk to lower the query execution cost.

Model Ensemble Pruning. If the OPTIMIZER profiles all the models in the ensemble over the sampled frames in a chunk, the optimization overhead may outweigh the benefits of reduced query execution time. FiGO uses an online ensemble pruning technique that is based on Thompson sampling. This allows the OPTIMIZER to only consider a smaller subset of models in the ensemble that have higher utility, thereby lowering the query optimization time.

3.3 Video Chunking

In this section, we first motivate the need for chunking in § 3.3.1. We then present a theoretical analysis of the accuracy of a given query plan based on the number of samples in § 3.3.2. Lastly, we present a cost model for estimating the execution cost of a query plan in § 3.3.3.

3.3.1 Variable Chunk Size

In FiGO, the OPTIMIZER picks a model for processing a chunk by first evaluating a set of models over the sampled frames from the chunk. This profiling step is important to ensure the runtime performance and accuracy of FiGO. First, if a query is evaluated over an *insufficient* number of sampled frames from a chunk, then the query accuracy with respect to that chunk may be low. This is because the OPTIMIZER may miss positive events that do not show up in the sampled frames (but are present in remaining frames of the chunk).

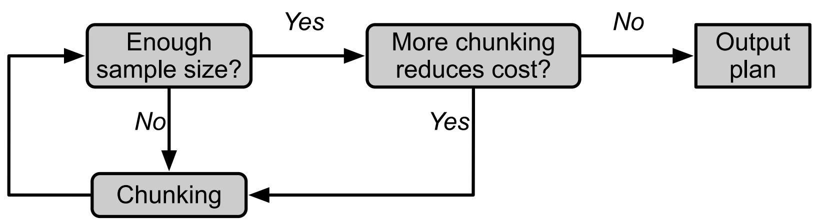


Figure 3.5: **Video chunking** – illustration of how the OPTIMIZER chunks a given video in FiGO.

Second, increasing the number of sampled frames leads to a higher query optimization time due to the profiling cost. If the plan is nearly optimal, and the VDBMS continues to collect more samples by further splitting the chunk, then it increases the overall query processing time.

To overcome these problems, FiGO splits the video into a sequence of differently-sized chunks. Since FiGO uniformly pick a fixed number (λ) of samples from each chunk, this approach is equivalent to varying the sampling rate across the video. A larger chunk size maps to a lower sampling rate. In contrast, a smaller-sized chunk maps to a higher sampling rate.

During optimization, λ is always fixed. FiGO instead varies each individual chunk size to strike a balance between accuracy and query execution time. λ is a configurable parameter.

Fig. 3.5 illustrates the chunking process. The OPTIMIZER first treats the entire video as a single chunk. If the number of samples (λ) obtained from a given chunk c is lower than the estimated number of required samples or the OPTIMIZER determines that c may be further split using the cost model, then it continues to further split c. We discuss how the OPTIMIZER estimates the lower bound of the number of required samples and how it uses the cost model in § 3.3.2 and § 3.3.3, respectively. Once the OPTIMIZER determines that the actual sample size (λ) is higher than the estimated lower bound, and that the plan cannot be further optimized, it picks the model for processing the remaining frames within c.

3.3.2 Sample Size Lower Bound

We now explain how the OPTIMIZER determines if the chunk is split enough to meet the accuracy constraint. Our key idea is that given the configurable sample size for each chunk λ, given an accuracy threshold $\mathbb{A}$, and an acceptable error range t, we estimate the probability that the accuracy of the query execution plan is within the acceptable error range based on the observed average accuracy and the average accuracy deviations across *all* the models in the ensemble. We then use the probability to obtain a lower bound of sample size K [3]. Using the *sample size lower bound*, the OPTIMIZER determines whether it needs to further split the chunk, so that actual sample size λ surpass the estimated lower bound K. We defer the threotical proof where we obtain the value of K to § A.

3.3.3 Cost Model

Lastly, we discuss how the OPTIMIZER uses the cost model. The idea is to compare the cost of further splitting the chunk (*i.e.,* more optimization time) against the estimated reduction in query execution time due to more fine-grained plans. If the estimated reduction in query execution time outweighs the additional optimization overhead, then the OPTIMIZER proceeds to further split the chunk.

We model the execution cost of a specific chunk as the product of the cost of the assigned model C_m and number of frames in a specific chunk $|V_i|$:

$$|V_i|C_m$$

Thus, the total cost of splitting the chunk is given by the sum of additional optimization costs and the cost of executing the new plan. Since splitting a chunk into two sub-chunks leads to doubling the number of samples, the optimization cost is given by the cost of evaluating all models in a collection M over the additional samples (λ samples for each

[3] Like other VDBMSs, FiGO computes accuracy based on consensus with respect to the reference model. This metric is highly correlated with a canonical F-1 score.

chunk):

$$2\lambda \sum_{m' \in M} C_{m'} - \lambda \sum_{m' \in M} C_{m'}$$

To estimate the execution cost of the new plan, the challenge here is that which models will be selected for the sub-chunks is unknown. However, we may obtain a lower bound on the execution cost. We denote the estimated cost of model profiling for left and right sub-chunks by C_{m_l} and C_{m_r}, respectively. The models found by the next iteration of optimization to use for the new sub-chunks should be at least as good as the model for left and right sub-chunks found during current iteration for providing good accuracy. Their costs are denoted by $C_{m_{lc}}$ and $C_{m_{rc}}$. So C_{m_l} and C_{m_r} must be greater than or equal to $C_{m_{lc}}$ and $C_{m_{rc}}$ (more expensive or slower model provides better accuracy). Since the size of the new sub-chunk is half of that of the original chunk, we estimate the execution cost by calculating its lower bound:

$$\frac{|V_i|}{2} C_{m_l} + \frac{|V_i|}{2} C_{m_r} = \frac{|V_i|}{2} C_{m_{lc}} + \frac{|V_i|}{2} C_{m_{rc}}$$

So, the potential reduction in query execution time is given by:

$$\frac{|V_i|}{2} C_{m_l} + \frac{|V_i|}{2} C_{m_r} - |V_i| C_m$$

To justify the optimization overhead, the reduction in execution time must be higher than the increase in optimization time:

$$\lambda \sum_{m' \in M} C_{m'} \leq \frac{|V_i|}{2} C_{m_l} + \frac{|V_i|}{2} C_{m_r} - |V_i| C_m$$

The OPTIMIZER computes this bound to strike a balance between query execution time and optimization time. If the estimated increase in optimization time is justified, it continues to split the chunk even if K is less than λ frames.

3.4 Model Ensemble Pruning

In this section, we present the model ensemble pruning technique that OPTIMIZER uses to lower the optimization overhead. We first motivate the need for ensemble pruning and its connection to multi-bandit problem and Thompson sampling [25]in § 3.4.1. We conclude with a discussion on how we tailor the Thompson sampling algorithm to the ensemble pruning problem in § 3.4.2.

FiGO uses a collection of models to accelerate queries. In particular, the ensemble consists of eight object detection models from the EFFICIENTDET family [21]. Each model offers a unique tradeoff between the accuracy and query execution time. To maximize speedup, FiGO supports as many models as possible. However, during the profiling step, the OPTIMIZER spends a significant amount of time evaluating all the models over the sampled frames. To tackle this problem, we leverage the observation that even though different models are used across different videos, it is often the case that only a subset of models is used while processing a given video. Thus, the OPTIMIZER prunes out the subset of models that is not likely to be used for a given query or video dataset, thereby lowering the overhead of the profiling step.

3.4.1 Connection to Multi-Armed Bandit

In reinforcement learning, the multi-armed bandit problem demonstrates [26] the dilemma between exploration and exploitation [27, 28]. A player has access to a set of slot machines and the goal is to maximize gain by playing on these machines. The dilemma lies in whether: (1) the player should try out other slot machines to discover machines with higher rewards (exploration), or (2) the player should stick to playing on a certain machine (exploitation).

The OPTIMIZER faces a similar challenge while pruning the model ensemble. To identify which models to prune, the OPTIMIZER must evaluate them on the sampled frames during profiling. While this will eventually lead to better ensemble pruning, the profiling overhead may outweigh those benefits. On the other hand, if the optimizer only evaluates a subset of models that delivers high accuracy, that would lead to imperfect model pruning as the OPTIMIZER is unable to leverage faster, less accurate models.

Thompson Sampling. This is a widely-used technique [29, 30] for solving the multi-armed bandit problem. It begins with no assumption about the reward of different actions. It then refines the estimation of the reward associated with each action by exploring the action space. A key benefit of this technique is that it returns the reward estimate along with a confidence score. We tailor this algorithm to tackle the ensemble pruning problem in FiGO.

3.4.2 Ensemble Pruning via Thompson Sampling

The goal of the OPTIMIZER is to find the performant model that makes the most correct predictions and discard other models. We may view the process of evaluating a specific model as an action a. The reward associated with a model Q_a is given by the average number of correct predictions made by that model. Thus, we formulate the ensemble pruning problem as finding the model among all models that maximizes the number of correct predictions (Q).

$$\underset{a \in A}{argmax}(Q_a)$$

In reality, the Q may only be estimated because the VDBMS does not know how a model performs a priori. The OPTIMIZER can estimate Q at time t by using the average of the observed reward value R [4] across all the sampled video frames examined before t.

$$Q_t(a) = \frac{1}{n} \sum_{i=1}^{n} R_i$$

[4] In FiGO, the observed reward value is 1 if the model is in consensus with the reference model. Otherwise, it is 0.

However, this greedy algorithm requires the OPTIMIZER to evaluate every model for the same number of times, leading to a higher optimization overhead.

To overcome this problem, we tailor the Thompson sampling algorithm to estimate the average number of correct predictions of each model. The estimator allows the OPTIMIZER to quickly converge to evaluating only a smaller number of models. As Thompson sampling, the estimator uses three variables: (1) n records how many times a model is evaluated. (2) τ represents the confidence of the estimation (higher confidence level if a model is evaluated more times), so it is correlated to n. (3) μ represents the expected value of estimated reward $E[Q]$. For each model, the estimated reward Q is updated as a running average. The OPTIMIZER updates the expected value of estimated reward weighted on variable τ as shown below:

$$Q_{t+1} = \Big(1 - \frac{1}{n}\Big)Q_t + \frac{1}{n}R$$
$$\mu_{t+1} = \frac{\tau_t \mu_t + nQ_{t+1}}{\tau_t + n}$$

When the OPTIMIZER checks the expected reward of a model, at time t, instead of returning Q directly, it returns a random value that is sampled from a probability distribution based on μ and τ:

$$Q_t \sim \mathcal{N}\Big(\mu_t, \sqrt{\frac{1}{\tau_t}}\Big)$$

This distribution not only accounts for the encountered values of Q, but also incorporates confidence of the evaluation.

Performance Cost. Typically, in the multi-armed bandit problem, the system only cares about an action with the highest reward. However, in FiGO, the OPTIMIZER needs to consider both the accuracy and execution cost of a particular model. So, we include the execution cost of a model C_a in the estimator. We use the profiled inference time per frame of a model as its execution cost.

$$\underset{a \in A}{argmin}(C_a) \wedge \underset{a \in A}{argmax}(Q_a)$$

This ensures that if two models have similar accuracy, then the OPTIMIZER picks the faster model.

Exploration Expansion. The canonical Thompson sampling algorithm only updates the estimated reward value of the best action. However, in FiGO, pruning all models in the model ensemble to one model does not minimize the query processing time. We empirically find that pruning the ensemble to three models provides sufficient flexibility for the OPTIMIZER. Pruning to fewer models lowers accuracy, and pruning to more models increases optimization overhead. So, we configure the OPTIMIZER to only consider the top three models during query optimization. However, for the pruning algorithm, OPTIMIZER must initially evaluate *all* the models over the first chunk to determine their accuracy and inference time metrics. For those selected models, the estimator evaluates all of them on sampled frames and updates their reward.

Ensemble Pruning + Video Chunking. We adapt the estimator to operate on fine-grained chunks. First, it is important to let the OPTIMIZER do sufficient exploration at very beginning. So, the OPTIMIZER evaluates all the models on the very first chunk. Otherwise, it may prune certain models without sufficient analysis. Second, the OPTIMIZER prunes and updates the estimated reward values for models at chunk granularity. For a given chunk, the OPTIMIZER first decides which models to prune based on their estimated reward value. It then evaluates those models to construct the plan. Based on the profiling step, it updates the estimated reward value of the evaluated models. For the next chunk, the OPTIMIZER prunes the models based on their updated reward estimates. This enables fine-grained, local adaptation of ensemble pruning to each chunk.

3.5 OPTIMIZER Implementation

We defer the discussion of the query optimization algorithm details to § B. We also add the following two optimization techniques to reduce the query optimization overhead and strike a balance between query optimization cost and execution performance.

Affinity-based Sampling for Reusing Results. Evaluating the consensus between models is computationally expensive as the VDBMS must evaluate those models. We use an affinity-based sampling technique that enables the OPTIMIZER to reuse profiling results. Instead of using a strict uniform random sampling algorithm, the OPTIMIZER prioritizes frames over which other models have already been evaluated. This allows the OPTIMIZER to reuse profiling results, thereby reducing the number of model invocations.

Chunk Size Limit. We also constrain the size of the smallest chunk to 100 frames. This corresponds to a maximal sampling rate of 10% (λ is set to 10 frames based on a sensitivity analysis. The reasons are two-fold. First, we configure the other baselines we compare FiGO against to use 10% of the video frames. To ensure a fair comparison, we also constrain the maximal sampling rate of FiGO. Second, we empirically found that this maximal sampling rate strikes a balance between accuracy and query processing time.

Table 3.1: **Query characteristics** – properties of queries and their associated video datasets. We first report the average size of videos and the selectivity of the queries. We then present the average chunk size and standard deviation. We finally report the overall query processing, optimization, and execution time.

Query	Dataset	Predicate	Avg. Video Size	Avg. Sel.	Chunk Size Avg.	Chunk Size Std.	Qry Proc. Time (s)	Qry Exec. Time (s)	Qry Opt. Time (s)
Q1.1	UA-DeTrac	`Count(Car)≥4`	1404	0.95	273	276	128.8	99.3	29.5
Q1.2	BDD	`Count(Traffic Light)≥1`	1205	0.47	163	69	187.1	149.4	37.7
Q1.3	VIRAT	`Count(Person)≥2`	1316	0.76	149	99	208.7	161.0	47.7
Q1.4	Jackson	`Count(Car)≥1`	10000	0.23	194	515	815.2	571.9	243.3
Q2.1	UA-DeTrac	`Count(Car)≥4 AND Count(Bus)≥1`	1404	0.73	166	150	186.3	141.6	44.7
Q2.2	BDD	`Count(Traffic Light)≥1 AND Count(Car)≥1`	1205	0.46	165	67	190.2	152.3	37.9

3.6 Evaluation

We seek to answer the following questions in our evaluation:

RQ1 – How effective is FiGO compared to the state-of-the-art techniques for accelerating queries in VDBMSs?

RQ2 – How does FiGO perform on complex queries with multiple atomic predicates?

RQ3 – What is the impact of the accuracy threshold?

RQ4 – How much does pruning lower optimization time?

RQ5 – How optimal is the plan found by the OPTIMIZER?

RQ6 – How does sample size affect accuracy & processing time?

RQ7 – How does FiGO generalize to another model ensemble?

3.6.1 Evaluation Setup

Evaluation metrics. Like others VDBMSs [1, 4, 20], we measure accuracy with respect to the reference model (*i.e.,* Det-7). We compute F-1 score of a baseline relative to the results of the reference model. We report query optimization time, execution time, and overall processing time (*i.e.,* optimization time + execution time).

Queries. Similar to the query shown in § 3.1, we evaluate the baselines on queries that focus on finding frames containing target object(s). We vary the dataset and the predicate to construct these queries. We report the average selectivity of these queries in table 3.1 (*i.e.,* the fraction of frames that satisfy the predicate(s)).

Evaluated Techniques. We reimplement these three key techniques used in state-of-the-art DBMSs: (1) model specialization with filtering (MS-FILTER), (2) model cascade (MC), and (3) model ensemble with coarse-grained optimization (ME-COARSE).

Model Ensemble. FiGO uses the EFFICIENTDET family of object detection models [21]. In particular, the ensemble consists of eight models (Det-0∼Det-7) with different accuracy-execution time tradeoffs. All the models are pre-trained on COCO dataset [31]. FiGO uses these off-the-shelf models to robustly answer diverse queries over different datasets. We configure the sample size of each chunk (λ) used by FiGO to 10 frames based on a sensitivity analysis (§ 3.6.7).

MS-FILTER. With model specialization, we construct a baseline inspired by PP. The model configuration consists of a lightweight filter that discards irrelevant frames followed by the heavyweight, reference model. We configure MS-FILTER to picks the optimal filter based on evaluation on 10% of the frames (elaborated in § 3.6.7). It picks a filter from Det-0 through Det-6 models, and uses Det-7 as the reference model. We found that directly returning answers using a lightweight model often returns inaccurate results (MS-SKIP). So, we do not compare FiGO against this baseline.

MC. We replicate the model cascade approach by connecting all the eight models in EFFICIENTDET in a sequence. We first optimize MC on 10% video frames to determine the optimal confidence thresholds to use for a given query. During query execution, MC decides to when to short-circuit the inference based on confidence threshold.

ME-COARSE. This baseline also uses a model ensemble similar to FiGO. Unlike FiGO, ME-COARSE takes a coarse-grained approach to optimization. It uniformly samples 10% of the frames from the entire video and profiles all the models over the sampled frames. It then picks exactly one model from the ensemble to process the remaining frames in the video.

Datasets. As shown in table 3.1, we evaluate these baselines on four representative video datasets: (1) UA-DeTrac [32], (2) Jackson-Town dataset from [1], (3) a subset of BDD100K dataset [33], and (4) a subset of VIRAT dataset [34].

UA-DeTrac and Jackson-Town. These datasets are obtained from traffic surveillance cameras. The majority of the objects in their videos are vehicles: cars, trucks, or buses. These datasets differ in terms of video length and content. Videos in UA-DeTrac are relatively short. They are often only 30 seconds long (*i.e.,* 1K video frames). In contrast, videos in the Jackson-Town dataset are longer.

BDD. Unlike the previous dataset, the BDD dataset consists of vidoes obtained from dash-cams. Since the videos are obtained from a *moving* camera, it is more challenging to deliver accurate answers on them. Videos in the BDD dataset are also comparatively short ($\sim$ 1K frames). Besides vehicles, the videos contain many traffic lights.

VIRAT. To ensure that FiGO works well on objects other than vehicles, we evaluate it on the VIRAT dataset for pedestrian detection. We pick a subset of videos from VIRAT that mainly contains difficult-to-detect human objects in the background of the frames. These videos range from 400 to 3K frames.

Software and Hardware. We implement FiGO with the PyTorch [35] framework. We use a server with 44 CPU cores and 256 GB memory along with one Titan Xp GPU with 12 GB memory.

3.6.2 End-to-End Performance

In this experiment, we first compare the F-1 score and query processing time of all the systems. The results are shown in Fig. 3.6. With FiGO, we set the accuracy threshold α to 0.95 and the error range t to be 0.03. We evaluate the query on the target video dataset five times and take the average of collect the metrics. We separately plot the F-1 score and query processing time on each video. We also mark the centroid across all videos for a given system using a scatter plot. The most notable observation is that FiGO outperforms other systems on both accuracy and query processing time.

Q1.1. The UA-DeTrac dataset contains videos of busy traffic intersections. So the selectivity of this query is high, as shown in table 3.1. Due to this, MS-FILTER has a very high query processing time demonstrated in Fig. 3.6. In contrast, both ME-COARSE and MC accelerate the query on some videos. While ME-COARSE offers a high F-1 score as it is easier to optimize, it leads to higher query processing time (*e.g.,* 632 s) on videos that re-

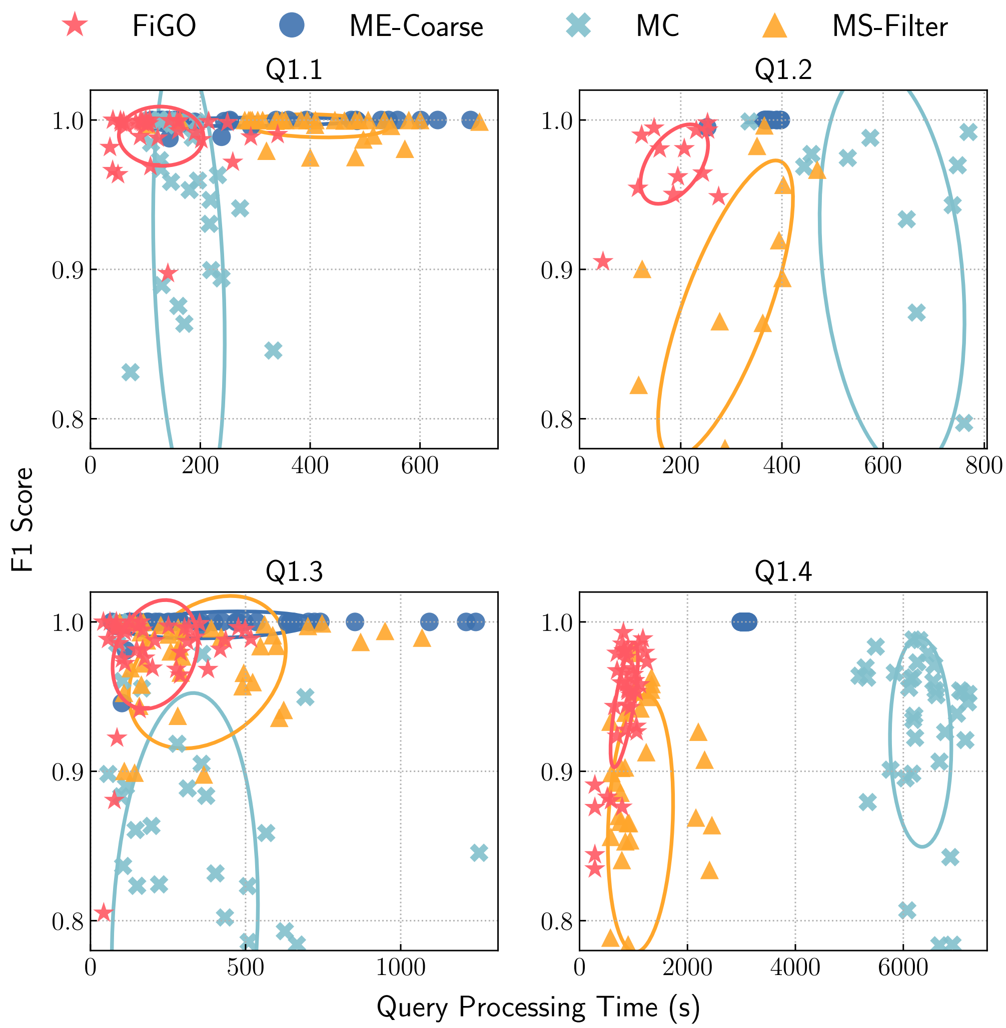


Figure 3.6: **End-to-end performance** – F-1 and query processing time of all the system across four queries.

quire more compute-intensive models. MC is faster but its F-1 score is lower (*e.g.,* it drops to 0.82 on a video). Compared to other systems, FiGO delivers better F-1 and processing time. We attribute this to two factors. First, FiGO finds better fine-grained plans that often use faster models. Second, FiGO uses a sample size bound and ensemble pruning to lower optimization time, which leads to faster query processing.

Q1.2 and Q1.3. Besides vehicles, we compare the performance of FiGO against other baselines on more complex objects like traffic lights and pedestrians. Since these objects are relatively smaller in the frame, the VDBMS often requires slower, compute-intensive models to provide accurate predictions. As shown in Fig. 3.6, FiGO still consistently beats ME-COARSE and MC on both queries ($1.8\times$ and $3.2\times$ on Q1.2 and $1.9\times$ and $1.4\times$ on Q1.3). MS-FILTER is comparable to FiGO, as its filter is able to discard many irrelevant video frames on these low selectivity queries. FiGO is $1.5\times$ and $1.2\times$ faster than MS-FILTER and delivers 0.06 and 0.01 higher average F-1 score on Q1.2 and Q1.3, respectively.

Q1.4. Lastly, we compare FiGO against other baselines on a query that operates on much longer videos. In this case, the processing time difference between FiGO and other systems is more prominent. This illustrates the importance of using fine-grained optimization in tandem with a model ensemble. MS-FILTER also has lower query processing time as the filter is effective in this low query selectivity (table 3.1). In this case, FiGO still shows is $1.4\times$ faster than MS-FILTER and delivers 0.08 higher F-1 score on average across all videos.

Chunk Size. We report the average chunk size and the standard deviation of chunk size for each query when the videos are processed by FiGO. The results indicate that the OPTIMIZER is effectively adjusting the chunk size based on the queries and the contents of the videos. For instance, the average chunk size decreases from 299 frames in Q1.1 to 243 frames in Q2.1 due to harder predicate.

3.6.3 Complex Queries Performance

In this experiment, we examine the performance of FiGO and other systems on two representative complex queries with multiple predicates (Q2.1 and Q2.2 in table 3.1). For example, the relational algebraic plan for Q2.1 is shown below:

$$\sigma_{Classification < \text{Count(Car)} \geq 4 \wedge \text{Count(Bus)} \geq 1 >}(\text{UA-DeTrac})$$

We tailor the OPTIMIZER to find a model that works well for *both* predicates together.

As shown in Fig. 3.7, on both queries, FiGO delivers a lower query processing time

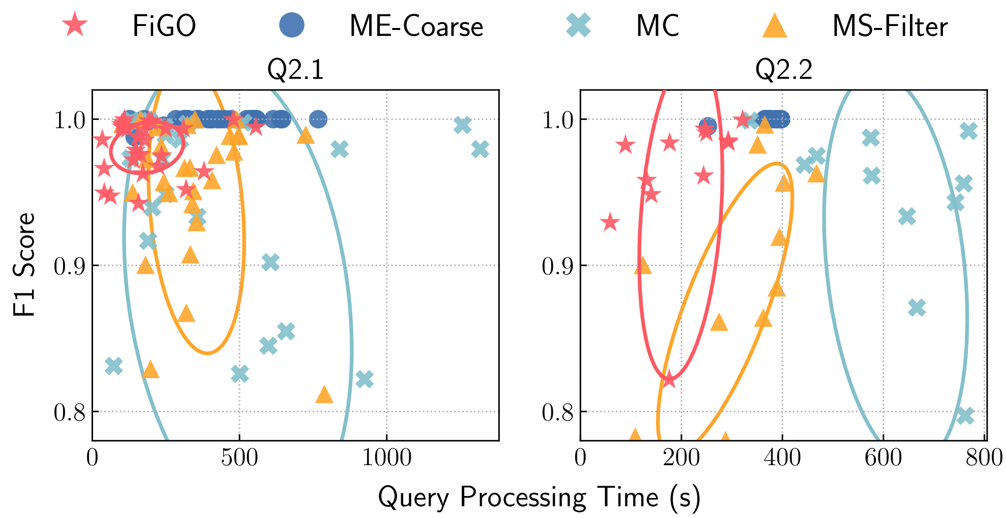


Figure 3.7: **End-to-end performance on complex queries** – F-1 and query processing time of all the system across two complex queries.

than other systems. As the predicates become more complex, MC requires significantly longer query processing time than others. In the case of MS-FILTER, complex predicates lead to lower selectivity. So the filter is able to discard more frames using faster models. However, we observe that this approach suffers from a lower F-1 score compared to FiGO (0.04 and 0.06 points on Q2.1 and Q2.2, respectively). ME-COARSE also suffers from higher query processing time since it does not leverage intra-video opportunities.

Execution with Join. We next compare the current technique for jointly evaluating a complex predicate against another technique that *separately* evaluates the predicates and joins the results based on the frame number. With this technique, the algebraic plan for Q2.1 is given by:

$$(\sigma_{Classification<\text{Count(Bus)}\geq 1>}(\text{UA-DeTrac}))$$
$$\bowtie_{\text{frameID}} (\sigma_{Classification<\text{Count(Car)}\geq 4>}(\text{UA-DeTrac}))$$

In this case, the OPTIMIZER in FiGO separately finds the optimal model for each atomic predicate. After obtaining results from both models, it joins them to return the final set of

Table 3.2: **FiGO *vs.* FiGO – Join** – F-1 score and query processing time metrics of: (1) FiGO (evaluating predicates together) and (2) FiGO – Join (evaluating predicates separately).

	Query	F-1 Score	Qry Proc. Time (s)	Qry Exec. Time (s)	Qry Opt. Time (s)
ME-COARSE-Join	Q2.1	0.99	691.69	495.38	196.31
	Q2.2	0.99	588.30	435.52	152.78
FiGO-Join	Q2.1	0.99	295.17	242.19	52.98
	Q2.2	0.99	275.88	238.66	37.22
FiGO	Q2.1	0.98	186.31	141.69	44.62
	Q2.2	0.95	190.23	152.36	37.87

frames that satisfies both atomic predicates.

We compare these two techniques in Q2.1 and Q2.2. We also implement join with ME-COARSE for comparison (as it is the strongest baseline). We report the average F-1 score and query processing time metrics associated with these systems in table 3.2. Our results demonstrate that there is a tradeoff between these two approaches. First, evaluating each predicate separately improves F-1 score. This is because combining multiple predicates together complicates the process of estimating the sample size bound. In contrast, evaluating multiple predicates together in the OPTIMIZER lowers query processing time (due to lower optimization time and execution time). Better support for complex predicates is beyond the scope of this paper. We plan to explore this problem in the future.

3.6.4 Impact of Accuracy Threshold

In this experiment, we investigate how the accuracy threshold affects the F-1 score and query processing time. We use Q1.4 which operates on longer videos. This allows us to better illustrate the impact of the accuracy threshold and the associated sample size bound. We evaluate FiGO across three accuracy thresholds: 0.95, 0.9, and 0.8. Across all cases, we configure the tolerable error range t to 0.03. FiGO executes Q1.4 100 times. We report the min, 25% percentile, median, 75% percentile, and max of results of all executions.

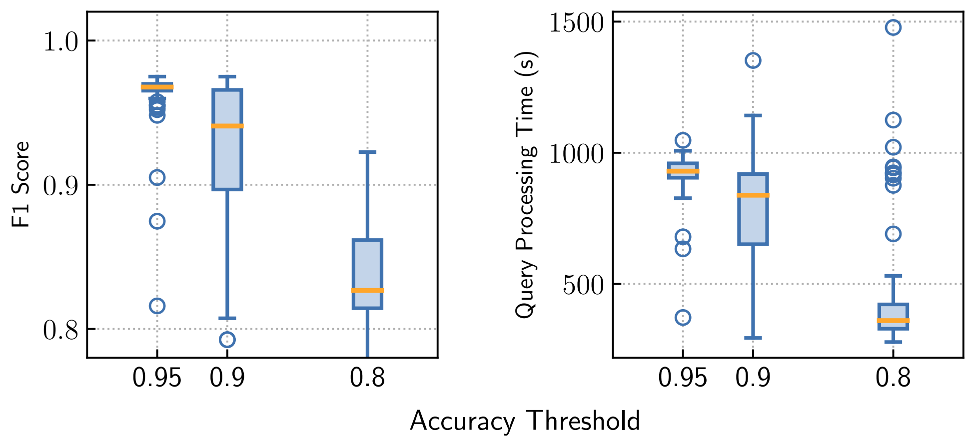


Figure 3.8: **Impact of accuracy threshold** – F-1 score and query processing time metrics under different accuracy thresholds.

As shown in Fig. 3.8, when we lower the accuracy threshold, the final query F-1 score is reduced as expected. This is because the sample size bound computed by the OPTIMIZER is highly correlated with the final F-1 score. We note that FiGO does not provide a strict accuracy guarantee. In addition, we observe that the variance of F-1 score increases when the accuracy threshold is lowered. This is because while the OPTIMIZER may find a good plan with fewer samples, it often ends up with a sub-optimal plan.

Besides F-1 score, we also evaluate the query processing time of FiGO under different accuracy thresholds. We discover that speedup increases when the threshold is lowered. The reasons are two-fold. First, fewer samples are needed by the OPTIMIZER when the system has a lower accuracy threshold, leading to lower optimization time. Second, with fewer samples, the OPTIMIZER often picks a faster, less accurate model for processing the chunk. Thus, a sub-optimal plan also results in lower execution time.

3.6.5 Query Optimization Time

In this experiment, we examine the efficacy of model pruning in reducing the optimization time. We focus on three systems: ME-COARSE, FiGO - no pruning, and FiGO. FiGO - no pruning differs from FiGO in that it picks the best model from all the models in the ensemble. Similar to FiGO, it also chunks the video during optimization. We measure the

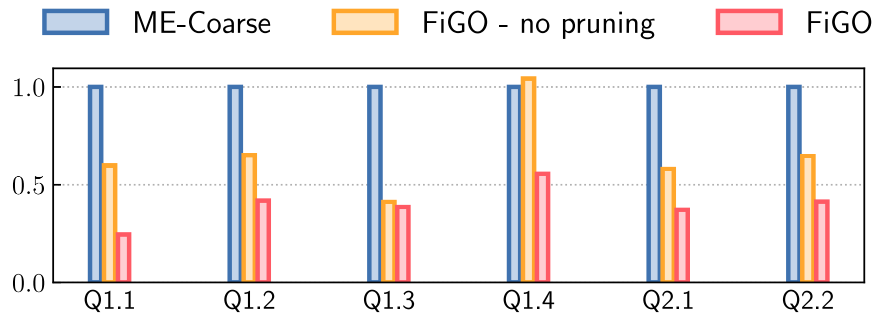


Figure 3.9: **Optimization time** – comparison of optimization overhead of three systems: ME-COARSE, FiGO without model pruning, and FiGO.

optimization time of all the queries across these three systems. We report the average time spent on query optimization over five runs. We normalize the optimization time of each system against that of ME-COARSE.

The most notable observation shown in Fig. 3.9 is that model pruning further reduces the optimization overhead. The reduction in optimization time depends on the query (*e.g.,* Q1.1 *vs.* Q1.3). This is because optimization time is unevenly distributed over different models (*i.e.,* evaluating a compute-intensive model costs more than evaluating a faster model). If the system prunes more compute-intensive models, the reduction is more significant.

Q1.1. This query only requires a faster model. So FiGO prunes all the compute-intensive models. Thus, ensemble pruning lowers optimization time by $2\times$ as shown in Fig. 3.9.

Q1.3. Correctly detecting pedestrian objects in Q1.3 is much harder. This is only feasible with compute-intensive object detection models. So, on this query, FiGO only prunes the faster models (must keep the slower models to meet the accuracy constraint). So, the impact of pruning on optimization time is smaller than that in Q1.1.

Discussion. As listed in table 3.1, optimization time accounts for $\sim 25\%$ of the total query processing time. So, the optimization overhead is non-trivial compared to the total processing time. To further reduce the optimization overhead, we consider training a *selector model* that is a fast deep neural network to directly estimate which model to use (instead

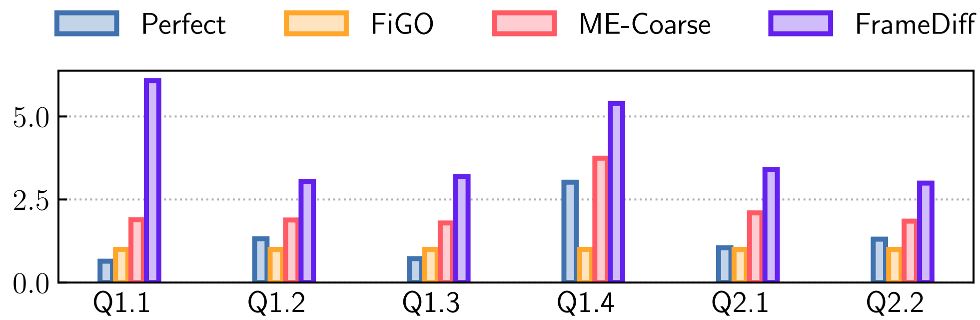


Figure 3.10: **Plan optimality** – comparison of query execution time with respect to that of the FiGO plan across four systems.

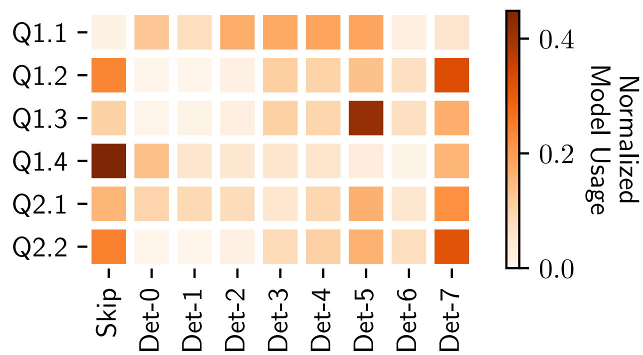


Figure 3.11: **Model usage distribution** – the distribution of usage of models in the ensemble across all queries.

of relying on the profiling step). Nevertheless, this approach suffers from two limitations. First, it is challenging to obtain enough training data. Second, the collected training dataset often has a skewed distribution. Due to its ineffectiveness, we do not include its results in this paper.

3.6.6 Plan Optimality

We refer to the plan that delivers the shortest query processing time with no loss in accuracy with respect to the reference model as the *perfect plan*. In this experiment, we study the optimality of plans generated by ME-COARSE, FiGO, and another baseline by comparing their query execution time against that of the corresponding perfect plan. We examine the FRAMEDIFF baseline in this experiment. FRAMEDIFF leverages the frame skipping technique presented in NoScope [1]. It uses a traditional frame-wise structural similarity index to determine whether inference with the reference model is necessary for the given frame. We identify the perfect plan by profiling all the models over every chunk. While this leads to high optimization overhead, the goal of this experiment is to only compare the query execution time associated with the resulting plan. We normalize the execution time metrics of these systems against that of FiGO.

The results shown in Fig. 3.10 demonstrate that the plans provided by ME-COARSE is sub-optimal compared to the corresponding perfect plans resulting in large performance gaps. FiGO consistently outperforms ME-COARSE. The execution time of FiGO may be further improved on Q1.1 and Q1.3. However, getting closer to the perfect plan will also increase the optimization overhead as it corresponds to smaller-sized chunks (*i.e.,* high sampling rate). On queries like Q1.2, FiGO even outperforms the perfect plan. This is because the OPTIMIZER selects faster models than that required for accurate predictions leading to a tolerable drop in F-1 score. FRAMEDIFF has a higher normalized execution time than other systems. The reasons are two-fold. First, the target objects may be very small with respect to the entire frame. In this case, the frame difference calculated by the traditional algorithm is not able to detect any significant change in the frame. Second, it must configure the threshold value used to determine whether the frame difference is significant enough to require inference with the reference model. To ensure tolerable accuracy, we configure a conservative threshold value, that further increases the execution overhead.

Model Usage Distribution. Lastly, we examine the distribution of usage of models in the ensemble across all queries. The results are shown in Fig. 3.11. The distribution changes across queries that operate on different datasets. On queries with low selectivity (*e.g.,* Q1.4), the OPTIMIZER skips several chunks. On queries with hard-to-detect events (*e.g.,* Q1.2 and Q2.2), it frequently uses the reference model to reach a high F-1 score. This illustrates that OPTIMIZER adapts the plan to different queries and datasets.

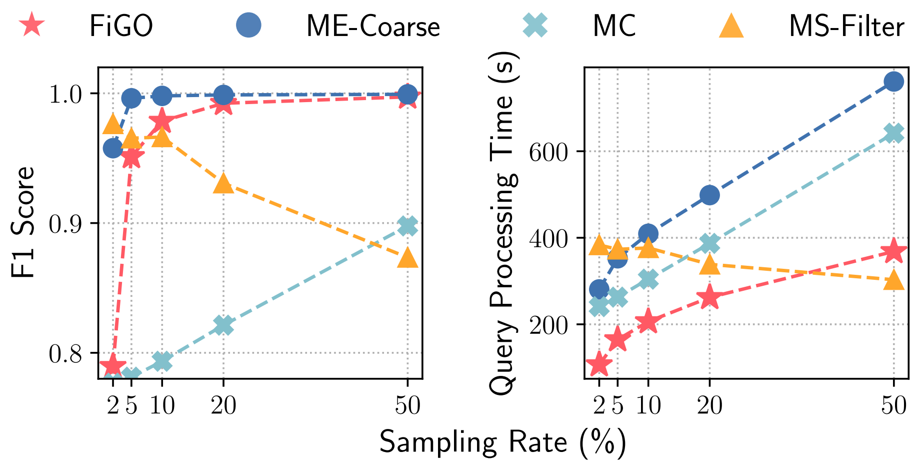
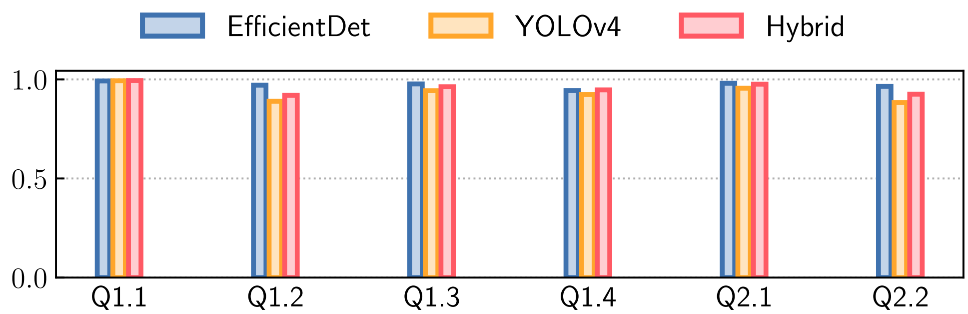


Figure 3.12: **Impact of sample rate** – comparison of the impact of sample rate on: (1) F-1 score, and (2) query processing time.

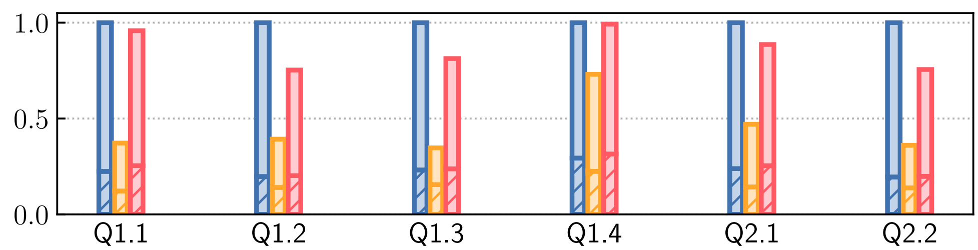
3.6.7 Impact of Sampling Rate

In this experiment, we investigate the impact of sampling rate. We examine these settings for sampling rate: 2%, 5%, 10%, 20% and 50%. Recall that FiGO always picks a fixed number (λ) of samples from each chunk but varies the size of each chunk. We instead vary the value of λ: 2, 5, 10, 20, and 50 samples in each chunk.

The results for a representative query (Q1.3) are shown in Fig. 3.12. With FiGO, if very small number of samples are picked from each chunk, then the sample size lower bound does *not* work due to high variance. So, this leads to a lower F-1 score ($\lambda == 2$). With ME-COARSE, the sampling rate has a significant impact on both the F-1 score and query processing time. It lowers processing time by lowering the sampling rate, but that also lowers the F-1 score. F-1 score of ME-COARSE does not improve when it surpass 10% sampling rate. FiGO delivers lower query processing time under the same F-1 score. With MS-FILTER, a higher sampling rate causes a lower F-1 score. The reason is that in our setting, MS-FILTER is able to pick a more aggressive filtering threshold value, which can incorrectly filter out positive events. Based on these results, we configure the sampling rate to 10% in all the other experiments to do a fair comparison between systems. While other systems may also reduce the sampling rate to lower query processing time (*e.g.,* 5% for



(a) **Generalization to other ensembles: F-1 score** – comparison of F-1 score between different model ensembles.



(b) **Generalization to other ensembles: processing time** – comparison of processing time between different model ensembles. We also report optimization time (hatched pattern) *vs.* execution time breakdown.

Figure 3.13: **Generalization to other ensembles** – FiGO on different model ensembles.

ME-COARSE), FiGO is still significantly faster.

3.6.8 Generalization to Other Ensembles

We next study the ability of FiGO to generalize to other model ensembles. We first evaluate FiGO over the SCALED-YOLOv4 ensemble [36] with five object detection models. The models in the SCALED-YOLOv4 ensemble deliver similar accuracy to those in EFFICIENTDET. They support faster inference but only offer a few performance-accuracy tradeoffs. We also evaluate FiGO over a hybrid ensemble (*i.e.,* HYBRID) that combines all the models in the EFFICIENTDET and SCALED-YOLOv4 ensembles (13 models in total).

In Fig. 3.13a, we report the average F-1 score comparison between three model ensembles. The F-1 score of all model ensembles is evaluated against the reference model in EFFICIENTDET. In Fig. 3.13b, we report the query processing time of three model ensembles. The processing time is normalized to the EFFICIENTDET model ensemble. We report the time breakdown by showing the optimization time (hatched pattern) and execution time.

The most notable observation is that the idea of fine-grained query optimization generalizes to different model ensembles. FiGO delivers good F-1 scores with all the ensembles. SCALED-YOLOV4 ensemble has a lower F-1 score compared to other ensembles, because of limited model diversity. So, it does not provide smooth tradeoffs between processing time and F-1 score. In the case of HYBRID, FiGO is able to pick an optimal query execution plan. The F-1 score of HYBRID is higher than the SCALED-YOLOV4 ensemble, and it has a lower query processing time than the EFFICIENTDET ensemble.

3.7 Related Work

We present a brief review of related work on query optimization. DB2's LEarning Optimizer (Leo) was a pioneering effort in improving the efficacy of query optimizers [37]. Leo learns from its mistakes by adjusting its statistical estimates over time. More recently, Neo [38] adopts a novel deep neural network-based approach for finding the optimal plan. It uses a value network for accurately predicting the latency of partial and complete query plans.

The overhead of query optimization in production DBMSs has increased over time. Developers of MemSQL reported that optimization time in analytical workloads may even be higher than query execution time [39]. It is critical for the OPTIMIZER to strike a balance between optimization and execution time. We discuss how FiGO tackles this problem in § 3.4.

3.8 Conclusion

We presented, FiGO, a video analytics system for efficiently processing visual data at scale. FiGO couples a model ensemble approach with fine-grained query optimization. Its OPTIMIZER first splits the video into a sequence of differently-sized chunks based on a sample size bound. It then picks a model from the ensemble that delivers the lowest query execution time while meeting the target accuracy constraint. Lastly, its EXECUTION ENGINE processes the remaining frames within the chunk using the selected model. FiGO prunes the model ensemble to lower query optimization time. We empirically show that these techniques enable FiGO to outperform the state-of-the-art approaches for processing queries over videos by $3.3\times$ on average across four video datasets.

CHAPTER 4
GPU DATABASE CHARACTERIZATION AND OPTIMIZATION

4.1 Introduction

Graphics Processing Units (GPUs) have attracted significant interest in the realm of accelerating data analytics due to their potential for massively parallel computing, high-bandwidth memory access capability, and ease of programming as an accelerator. In recent years, a number of GPU database management systems (DBMSs) have been developed in both academic and industrial settings [40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52]. Recent advances in interconnect [53, 54, 55, 56] and GPU architecture have made GPUs more attractive as accelerators for data analytics. As a result, we anticipate a rise in the popularity of GPU DBMSs and a proliferation of research into improving their efficiency in the future.

Several factors contribute to query performance in GPU DBMSs, including (1) the resource capacity (including both compute and memory resources) of the GPU; (2) the implementation of the DBMS; (3) the characteristics of the query; and (4) the size of the database. A deeper understanding of GPU resource utilization and encountered bottlenecks is crucial in designing better GPU DBMSs.

Prior Work. While previous studies (*e.g.,* [57, 58, 40]) have compared query performance across GPU DBMSs, a cross-stack analysis that connects query performance with the GPU resource utilization along with microarchitectural metrics is lacking. In contrast, similar studies [59, 60, 61, 62] exist for CPU DBMSs. We aim to address this gap by conducting both resource-utilization and performance-bottleneck analysis for GPU DBMSs. Based on the findings from our studies, we propose optimizations to improve both the system implementations and GPU resource allocations.

With several GPU instance types now available, each with different performance and cost trade-offs, choosing the most suitable GPU for a particular workload can be challenging. For example, a T4 GPU can run the TPC-H benchmark 20% slower compared to a P100, but at $\frac{1}{5}$th of the cost [40]. To complicate things further, starting from the Ampere generation, NVIDIA now allows different resource-allocation mechanisms through the Multi-Instance GPU (MIG) capability [16]. This new capability allows partitioning GPU hardware resources to increase concurrency by sharing the GPU. Developers are now faced with several choices on how to schedule their workloads over GPUs, which requires a deep understanding of both workload characteristics and hardware properties.

Characterization. In this paper, we conduct an in-depth study on both GPU resource utilization and performance bottlenecks. We build on the roofline model [63] to identify the underutilized resources in GPUs. Our findings indicate that many queries underutilize either DRAM or L2 cache bandwidth, even in highly optimized systems. Since GPU resource utilization is dependent on the implementation of each system, we conduct a GPU execution performance bottleneck study by profiling microarchitectural performance counters to fully comprehend the reasons for the underutilization of resources in each system. The study reveals that several systems still have opportunities for further optimization due to a lack of kernel fusion, inefficient threads termination, and cache underutilization. Our approach can be readily adopted by other systems to identify and address potential performance issues.

Optimization. Based on the insights we gained, we propose remedies to improve the queries' performance. Our strategy focuses on two aspects: efficient system implementation and optimal resource allocation. To optimize system implementations, we improve terminations of idle threads and cache hit rates for the best-performing (academic) GPU database (CRYSTAL). To improve resource allocation, we employ an analytical model to determine the ideal resources to allocate so that more query executions can be concurrently accommodated by the same GPU. This approach improves the overall throughput without modifying any system implementation. As far as we know, we are the first to show how

MIG and MPS technology can be used to improve GPU databases throughput.

Contributions. The contributions of this work are as follows:

1. **Characterization of GPU DBMSs** (§ 4.4 and § 4.5): To close the gap in GPU DBMS characterizations, we conduct two original studies: (1) GPU resource-utilization analysis, and (2) GPU DBMS bottleneck analysis based on GPU microarchitectural metrics. Our studies show that many GPU DBMSs underutilize GPU resources. Additionally, we discover previously unknown performance bottlenecks (*e.g.,* inefficient memory bandwidth utilization) existing in the State-of-the-Art (SotA) GPU DBMSs.

2. **Performance optimization of GPU DBMSs** (§ 4.6 and § 4.7): To demonstrate that the analyses are useful for guiding optimizations, we showcase implementation optimizations on CRYSTAL, resulting in an improved single-query execution performance with an average speedup of $1.9\times$. This, we believe, makes it the fastest open-source implementation of SSB queries on GPUs. Furthermore, we demonstrate GPU resource allocation optimizations guided by our proposed analytical model. The analytical model can reason about GPU resource utilization, while concurrently running multiple queries with up to $6.5\times$ throughput improvement. This tool provides system administrators with effective guidelines on how to allocate GPU resources to queries.

Outline. We introduce the background information and experimental setup in § 4.2 and § 4.3. In § 4.4, we compare the performance among various GPU DBMSs. Motivated by the performance gap between different GPU DBMSs, we use the roofline model to understand the GPU resource utilization of those systems. Then, to understand the causes of GPU resource underutilization, we profile GPU microarchitectural metrics to understand existing bottlenecks in § 4.5. Finally, we showcase our proposed optimizations in § 4.6 and § 4.7 to address our identified performance bottlenecks.

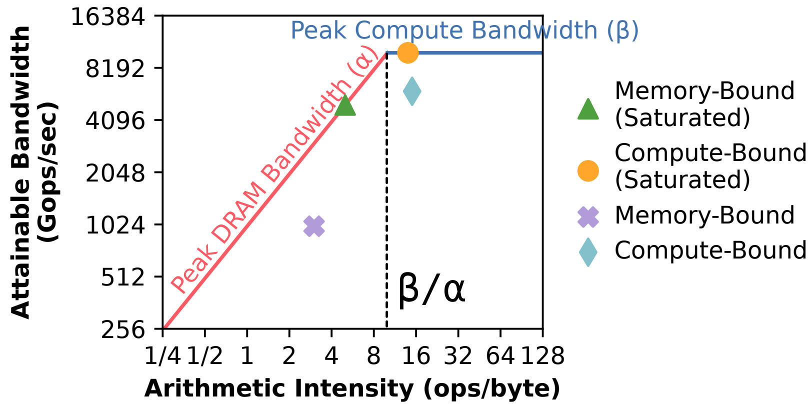


Figure 4.1: **Roofline model with different resource bounds.**

4.2 Background of Performance Modeling and GPU-tailed DBMSs

We provide information about the roofline performance analysis model in § 4.2.1. We then present an overview of existing GPU database systems in § 4.2.2.

4.2.1 Roofline Performance Modeling

The roofline model [63] is a well-known performance-modeling method to study resource bottlenecks for existing algorithms. The roofline model assumes that any execution on specific hardware is bounded either by its memory resources or by its compute resources. Visually, as shown in Fig. 4.1, the model contains two lines to indicate the peak memory bandwidth (α) and the peak compute bandwidth (β). Any execution on this hardware will correspond to a point within the space bounded by these lines—those two lines are considered the performance ceilings for that hardware. The X-axis represents the arithmetic intensity (AI), calculated by dividing the number of operations (*e.g.,* integer or floating-point operations) by the number of bytes read during execution. The Y-axis indicates the achieved throughput, calculated as the operations per second.

Conventionally, a query could be either memory-bound (AI $< \frac{\beta}{\alpha}$), or compute-bound (AI $> \frac{\beta}{\alpha}$) [64]. The performance of algorithms that already saturate the bandwidth of either of the resources will be impacted by changes in the allocation of the corresponding

Table 4.1: **Qualitative comparison of GPU database systems** – We consider the following characteristics. **Coverage**: whether the system is general purpose or only specific queries are supported. **Data Format**: formats for data input to the system. **Backend**: how data operators are compiled for execution. **Open Source**: if the system is publicly available.

System	CRYSTAL	HEAVYDB	BLAZINGSQL	TQP
Coverage	SSB [65] only	General purpose	General purpose	General purpose
Data Format	Binary array	CSV Parquet	CSV, DF Parquet	CSV, DF Parquet
Backend	Hardcoded CUDA	LLVM to PTX	Thrust cuDF	PyTorch
Open Source	Yes	Yes	Yes	No

resource (*e.g.,* memory-bound saturated or compute-bound saturated in Fig. 4.1). Algorithmic or compiler inefficiencies will increase the AI and make an otherwise memory-bound execution compute-bound, causing the query to take more time to complete.

4.2.2 GPU Database Systems

table 4.1 summarizes the key characteristics of four database systems. **CRYSTAL.** CRYSTAL [43] is a recently proposed SotA GPU database system that delivers superior query-execution performance compared to other systems. It currently supports only queries from the Star Schema Benchmark (SSB) [65]. All the queries are written in CUDA and have hard-coded parameters for the size of the hash table, the number of groups, and the size of the output table [66]. While this is feasible for these queries in SSB, it is not feasible to predetermine these parameters for arbitrary SQL queries. CRYSTAL also assumes that each column is a binary array generated after preprocessing. String columns in CRYSTAL are converted to binary arrays using dictionary encoding on the CPU.

HEAVYDB. HEAVYDB [42] is a widely used GPU database system that supports many types of SQL queries. Besides the query executor, it contains other components like query parser and query optimizer. HEAVYDB takes various data formats as input, such as CSV and Parquet. Data in HEAVYDB is grouped into fragments and transferred to the GPU using different CUDA streams. Internally, it uses LLVM [67] to generate PTX [68] code for query execution. Additionally, HEAVYDB leverages a custom LLVM pass for more flexible and optimized execution strategies.

BLAZINGSQL. BLAZINGSQL [41] is another open-source GPU database system. Similar to HEAVYDB, BLAZINGSQL is also very flexible and handles different types of queries. Unlike HEAVYDB, it uses Thrust [69] and cuDF [70] as its backend for query execution. Reusing the APIs of Thrust and cuDF has both pros and cons. While less engineering is needed to support various operations, certain operators cannot be implemented in an efficient way (*e.g.,* cannot avoid the unnecessary generation of intermediate results) due to the limited functionalities of existing APIs. BLAZINGSQL also supports different input data formats, while data moves between CPU and GPU through different CUDA streams.

TQP. TQP [40] is a recently-presented GPU database system from Microsoft. It is designed to be general-purpose (*e.g.,* it supports the full TPC-H benchmark). The interesting aspect of TQP is that internally it uses the PyTorch [71] framework as its backend for executing relational operations. This design choice allows it to quickly support many different operations with existing GPU kernels that are already optimized. As the PyTorch framework already supports various hardware platforms (*e.g.,* NVIDIA GPUs, AMD GPUs), TQP inherits the portability and extensibility of the PyTorch framework.

Query Optimizations. All of these systems have query optimization and query compilation phases except for CRYSTAL, where the query plans are hard-coded and predetermined based on the selectivity of each operator (*i.e.,* selective operators are executed earlier). HEAVYDB and BLAZINGSQL rely on the Apache Calcite framework for query optimization. HEAVYDB also uses LLVM to further optimize the physical execution of GPU code. TQP utilizes SparkSQL [72] to optimize the SQL queries and then translates the Spark SQL physical plans to an intermediate representation (IR). Based on the IR, TQP assem-

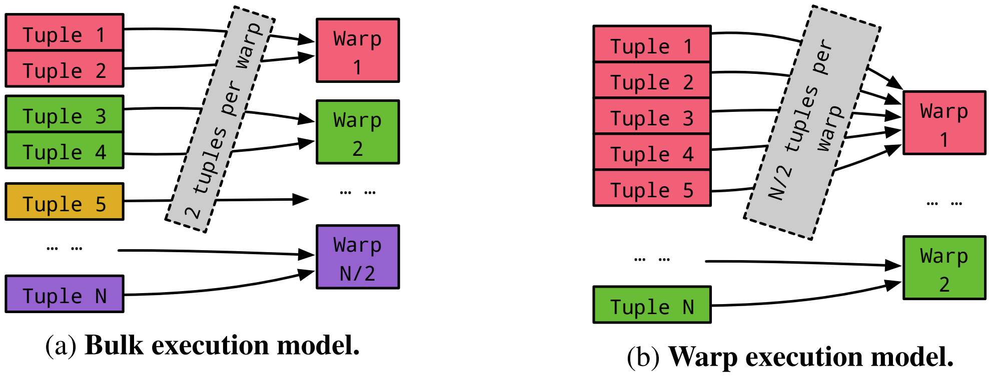


Figure 4.2: **Query execution models –** Data mapping for query execution.

bles a PyTorch program as a composition of predefined tensor programs, one for each operator in the IR. The implementation is later optimized by the PyTorch compiler.

Query Executions. There are two common execution models, as shown in Fig. 4.2. The bulk execution model (Fig. 4.2a) has a fixed number of tuples per warp. The GPU will launch a variable number of warps to accommodate different data sizes. CRYSTAL, BLAZINGSQL, and TQP all adopt this execution model. On the other hand, HEAVYDB uses a warp execution model, in which the GPU always launches a fixed number of warps (2 as shown in Fig. 4.2b). Consequently, each warp gets a variable number of tuples to process based on the total amount of work. Those two different execution models can also have an impact on GPU memory efficiency.

4.3 Experimental Setup

We discuss the hardware, query workloads, and profiling toolchains used in this paper in § 4.3.1, § 4.3.2, and § 4.3.3, respectively. We then describe the two execution scenarios that we consider in § 4.3.4.

4.3.1 Hardware

We use an NVIDIA A100 GPU [73] with the Ampere architecture, 40GB of GPU memory, and 108 SMs on a dual-socket AMD EPYC 7313 machine with 16 physical (32 logical)

Table 4.2: **Qualitative summary of the queries from SSB** – Key characteristics of queries in SSB: **Number of Joins**, the **Aggregation** type, whether they require **Sorting**, and query **Selectivity**.

Query Group	Group 1			Group 2			Group 3				Group 4		
	Q11	Q12	Q13	Q21	Q22	Q23	Q31	Q32	Q33	Q34	Q41	Q42	Q43
# Joins	1			3			3				4		
Aggregation	Sum			Group By			Group By				Group By		
Sorting	No			Yes			Yes				Yes		
Selectivity	*1.9e-2*	*6.5e-4*	*7.5e-5*	*8e-3*	*1.6e-3*	*2e-4*	*3.4e-2*	*1.4e-3*	*5.5e-5*	*7.6e-7*	*1.6e-2*	*4.6e-3*	*9.1e-5*

cores per socket and 3 GHz base clock frequency. Each SM can schedule up to 64 *warps*. A *warp* is the basic execution unit of the NVIDIA GPU. A warp consists of 32 threads scheduled and executing in a *single instruction, multiple threads* (SIMT) fashion (*i.e.,* each thread executes the same instruction on different data but allows divergence between threads). The A100 GPU supports newer features like MIG and MPS, as introduced in § 4.3.1. The GPU is connected to the CPU via PCI-e 4 protocol, which provides up to 32 GB/s of bandwidth. For consistent performance, we set the GPU clock frequency to be 1410 MHz.

4.3.2 Workloads

Since CRYSTAL currently only supports SSB [65], in our experiments we decided to evaluate SSB queries over the four different GPU database systems. Even if SSB is simpler than other benchmarks like TPC-H and TPC-DS, its queries are complex enough that system characterization on this benchmark enables us to find several interesting insights. We are confident that these insights generalize to other workloads as well.

table 4.2 presents a summary of the queries in SSB. Queries in group 1 have only one join, and the results are aggregated into a single scalar value. All the other queries have multiple joins (up to 4), and the final results are aggregated into a table with multiple groups. Based on our profiling, we found that 16 is the largest scale factor on which all DBMSs can run queries without moving data back and forth between GPUs and CPUs

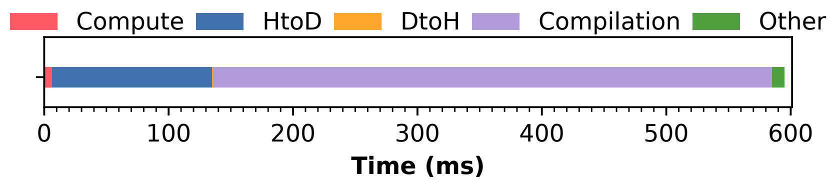


Figure 4.3: **Cold execution characterization over Q21 at SF=16 in HEAVYDB** – We split the end-to-end performance of cold query execution into six components: *Compute* time, *Host-to-Device* data transfer (HtoD), *Device-to-Host* data transfer (DtoH), *Compilation* time, and *Other* CUDA context setup and memory management time. Similar results apply to other queries and database systems.

or encountering out-of-memory issues on the GPU. So, we present results for most of the experiments with a scale factor of 16 unless specifically mentioned.

4.3.3 Profiling Toolchains

We extensively use NVIDIA Nsight System [74], Nsight Compute [74], and nvidia-smi [75] tools built with CUDA11.6. The NSight System provides system-wide time breakdown, including time spent on data transfer, memory allocation, kernel execution, etc. We also use Nsight Compute to obtain detailed kernel execution metrics, which include achieved instruction per cycle, cache utilization, etc.

4.3.4 Warm *vs.* Cold Execution Scenarios

In the *warm scenario*, we assume that the data has already been loaded (or cached) in GPU memory, the device has warmed up, and the query has been parsed, optimized, and compiled (*e.g.,* the physical plan is available in the plan cache). In the *cold scenario*, we assume that the data resides in CPU memory and needs to be transferred to the device, and all the other overheads associated with query parsing, optimization, and compilation are considered.

Cold Execution Overhead. In this paper, our focus is on warm query execution, where the majority of the time is spent on GPU computations. However, here we briefly report the timing breakdown for cold query execution to provide an overview of where time is

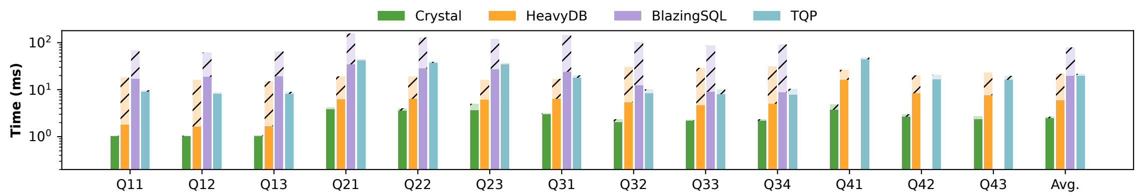


Figure 4.4: **Warm execution characterization** – End-to-end query execution time and GPU execution time for all queries (BLAZINGSQL does not support queries in group 4; those queries are skipped when calculating the average.). The hatched area represents time spent besides the actual GPU execution.

typically spent. With cold query execution, there are several non-negligible overheads from both query optimization and compilation, as well as from data transfer. Fig. 4.3 shows that these overheads are always greater than the actual query execution time (Compute). Data needs to be moved to the GPU before query execution, and this incurs significant overhead. Data transfer and kernel computation can be overlapped, but the time saving is small due to data transfer overhead being the dominant bottleneck. During data transfer, most systems copy only the columns needed for the query to the GPU to reduce the host-to-device (HtoD) data transfer overhead. HtoD overhead in Fig. 4.3 reflects the time spent on copying the required columns.

Systems also need to move results from the device back to the host. However, as all the queries compute aggregated results that are very small, the device-to-host (DtoH) overhead is generally minimal. Query plan optimization and compilation also introduce non-trivial overhead. This is unavoidable when the system receives a query for the first time. Most systems implement plan caching, so recurrent queries do not have the same overhead.

4.4 Performance And Resource Analysis

In this section, we first provide a high-level overview of execution time for both end-to-end and GPU computation, in all systems, in § 4.4.1. We then conduct a resource-utilization analysis to reason about performance differences between systems in § 4.4.2.

4.4.1 Query Execution Performance

We present the end-to-end warm query execution performance in Fig. 4.4 [1]. The most notable observation is that CRYSTAL represents the upper bound in execution performance among all considered systems. Both HEAVYDB and TQP are on average $8\times$ slower than CRYSTAL. BLAZINGSQL is even slower, being $30\times$ slower than CRYSTAL. Next, we provide our analysis of how time is spent on each system by using the time breakdown information.

CRYSTAL *and* TQP. For both of these systems, the end-to-end time mostly represents the actual GPU compute time, with some small overheads besides GPU execution. In CRYSTAL, each CUDA source file implements one query from SSB. Once the source file is compiled, there is no additional compilation overhead associated with the query. Unlike CRYSTAL, TQP is a general-purpose system that supports a wider range of SQL queries. In TQP, the workflow for executing queries is done in two phases: the input query is first parsed, optimized, and compiled into a PyTorch model object; then, the already-compiled model is executed over the input data. Because of this workflow, the query execution time does not contain optimization and compilation time. In addition, TQP lazily caches the query results in the GPU until they are requested. This approach is beneficial for reducing end-to-end query execution time if the results are not immediately required, or if there is a subsequent query executing on the previously generated output results.

HEAVYDB *and* BLAZINGSQL. For HEAVYDB and BLAZINGSQL, their end-to-end time consists not only of the GPU compute time but also of other overheads. For example, HEAVYDB uses Calcite for query plan optimization, and LLVM to compile the query into executable code. Although HEAVYDB implements a plan cache, it still has non-negligible overhead for query parsing and GPU setup. Similar to HEAVYDB, BLAZINGSQL also has non-trivial overheads beyond the GPU compute time. In practice, if the system is able to

[1] Most systems can successfully execute SSB queries except BLAZINGSQL, which cannot run Q41, Q42, and Q43 due to errors during dataframe encodings.

look up the cached physical query plan without even parsing the query, like TQP, then it can avoid many unnecessary overheads.

Next, we study each system's GPU execution time and its efficiency. While some works benchmark different GPU database systems [58], studies on only GPU execution efficiency have not been done before. The GPU execution time of the studied systems is shown in Fig. 4.4 with non-hatched bars.

For GPU execution time, CRYSTAL is still the fastest system. However, unlike the end-to-end query time, HEAVYDB performance is closer to CRYSTAL ($2\times$ slowdown, as opposed to $8\times$ slowdown, reported in the end-to-end comparison). TQP is still $8\times$ slower than CRYSTAL. BLAZINGSQL has the slowest performance among these systems and is unable to run queries in group 4.

4.4.2 GPU Resource Utilization

The significant performance gap among different systems motivates us to investigate the reasons behind it. In this section, we attempt to visualize the resource utilization of each system and rely on it to reason about the root causes of the performance gap. During our investigation, we explain the reasons for GPU resource underutilization by connecting them to query characteristics and system implementations.

DRAM Utilization

We first present the DRAM-level resource utilization for all systems. We obtain the metrics described in table 4.3, from which we derive the AI and the attainable bandwidth. We use the theoretical GPU DRAM bandwidth [73] for the DRAM bandwidth ceiling. We note that GPUs also have other functional units like floating-point operation units. However, because most OLAP queries require only integer operations, those functional units are not needed for constructing the roofline model.

The challenge for this type of modeling is that each query consists of multiple kernels.

Table 4.3: **Profiled metrics to construct the roofline model.**

	Metric Name	Description
DRAM	gpu__time_duration.sum	Execution duration
	dram__bytes.sum	Total bytes from DRAM
	smsp__sass_thread_inst_executed_op_integer_pred_on.sum.per_cycle_elapsed	Achieved compute bandwidth
	sm__sass_thread_inst_executed_op_integer_pred_on.sum.peak_sustained	Peak compute bandwidth
L2 Cache	lts__t_requests_srcunit_tex_op_read.sum	Total requests to L2 cache

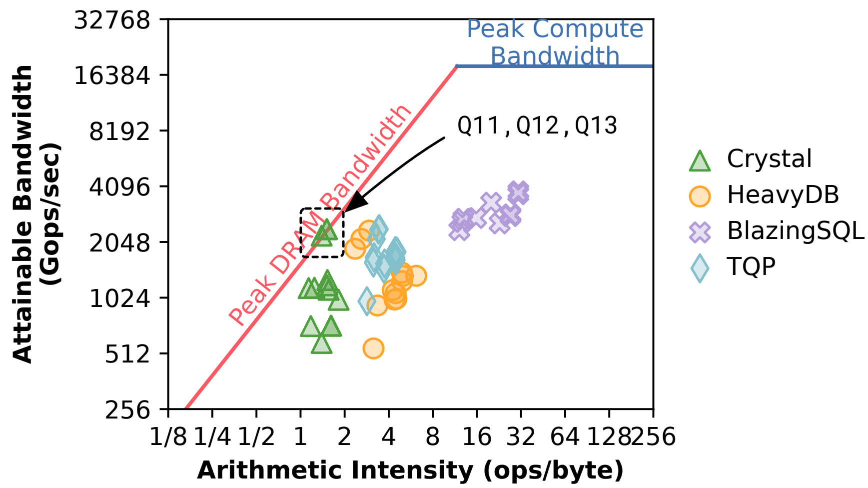


Figure 4.5: **DRAM roofline model** – for SSB queries (SF=16).

Our approach is to aggregate scalar metrics such as the execution duration, total bytes, and total integer operation instructions. We then use the aggregated metrics to obtain the required metrics for constructing the roofline model. Fig. 4.5 shows where all 13 SSB queries are located with respect to the roofline model, for each of the four GPU DBMSs.

Query Specific. We observe that most queries do not fully use the provided DRAM bandwidth because they all have hash-join operator causing many random accesses. Queries like Q11, which have only one hash-join operator, are likely to have higher achieved DRAM bandwidth.

Implementations Specific. Although query characteristics influence both AI and achieved bandwidth, they are also largely dependent on the system implementations. As shown, the AIs of CRYSTAL, HEAVYDB, and TQP are relatively low. Compared to the other three systems, BLAZINGSQL is instead more compute intensive. This shows that even simple OLAP queries can be compute intensive depending on the query implementation in the DBMS.

Even the same queries have different achieved bandwidths in different implementations. For example, three queries have already saturated the peak GPU DRAM bandwidth in CRYSTAL. This is because CRYSTAL implements the hash join as a filter for Q11, Q12, and Q13 (§ 4.3.2). For those queries, the hash join is just a one-to-one mapping between rows from the fact table and the dimension table. CRYSTAL pushes down the predicate selection from the dimension table to the fact table, which skips the hash join and simply runs a predicate selection (*i.e.,* filter) on the fact table. As the filter operator involves only running a table scan, it is feasible to saturate the GPU DRAM bandwidth (unlike hash join). Even though this may not be applicable to all queries, it is still an interesting optimization to do from the resource-utilization perspective.

L2 Cache Utilization

We also discovered that the GPU DRAM bandwidth is *not* the only resource constraint during query execution. Especially for optimized systems like CRYSTAL and HEAVYDB, they are more likely to saturate the peak L2 cache bandwidth. This motivates us to extend the roofline modeling methodology to the L2 cache as well. In prior work, Ilic [76] also proposed to make the roofline model cache-aware for CPUs.

Model Innovation. On top of the previous study, we find that *the AIs of different resources are very different*. For example, when a query has a good L2 cache hit rate, most of the memory requests are handled by the L2 cache. So, the number of bytes loaded from the L2 cache are high. As a result, the AI relative to the L2 cache is low. On the other hand, because there are fewer bytes loaded from the GPU DRAM, the AI is high with a fixed

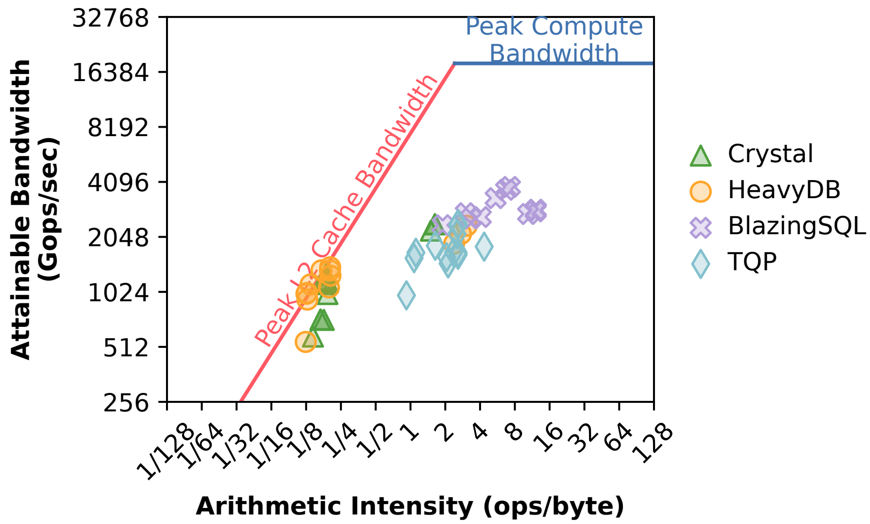


Figure 4.6: **L2 cache roofline model** – for SSB queries (SF=16).

number of integer operation instructions. This requires us to have a separate roofline model to characterize the same query for a different memory resource.

To construct the roofline model for the L2 cache, we reuse most metrics profiled in table 4.3 except for the total bytes read from DRAM. To estimate the bytes read from the L2 cache, we profile the number of L2 requests (shown in table 4.3) that the kernel loads, which can be used to calculate the total bytes loaded from the L2 cache by knowing that the cache line size is always 128 bytes. We present our profiling results in Fig. 4.6.

Query Specific. We first observe that the AI of queries on the DRAM roofline model is higher than their corresponding AI on the L2 cache roofline model. This is reasonable, especially in the case in which queries have good utilization of the L2 cache bandwidth. Because most memory requests are completed at the L2 cache level, the GPU has less data to handle at the DRAM level. We also observe that many of those queries have reached the peak L2 cache bandwidth. We discover that queries with hash joins are more likely to saturate the L2 cache bandwidth. Because SSB has relatively small dimension tables that can fit into the L2 cache of a high-end GPU (*e.g.,* 40 MB), queries may still exhibit good L2 cache utilization due to the spatial locality of data accesses from the small working set size even though hash joins involve random accesses. In Fig. 4.5, some queries saturate the peak DRAM bandwidth—queries with simple filters are more likely to be bound by DRAM bandwidth. Since there is minimal data reuse, most data is streamed to the kernel, and cache utilization is lower. As a result, visually, those queries move away from the peak L2 bandwidth ceiling.

Implementations Specific. Because CRYSTAL and HEAVYDB are very optimized, their queries saturated the L2 cache bandwidth. In BLAZINGSQL and TQP, the query implementations are far from the peak L2 cache bandwidth ceiling. Those two systems now have clearly separated clusters from the one from CRYSTAL and HEAVYDB, indicating lower cache efficiency in BLAZINGSQL and TQP.

4.5 Bottleneck Investigation

In § 4.4, we provide an overview of the performance and resource utilization of various GPU database systems. GPU resource underutilization can be caused by many different factors. Therefore, in this section, we aim to examine the existing implementation bottlenecks, by using hardware performance counters, and offer insights into how to reduce those bottlenecks.

4.5.1 Operation and Data Efficiency

As mentioned in § 4.2.1, the roofline model is derived from compute operations and data load. Thus, we first study the operation and data efficiency by profiling the following hardware counters: (1) the number of integer operations, and (2) bytes loaded from DRAM.

Integer Operations and Bytes.

First, we profile the total integer operations for the four systems, as shown in Fig. 4.7 (top chart). We find that HEAVYDB performs slightly more integer operations than CRYSTAL. Conversely, BLAZINGSQL and TQP execute significantly more integer operations com-

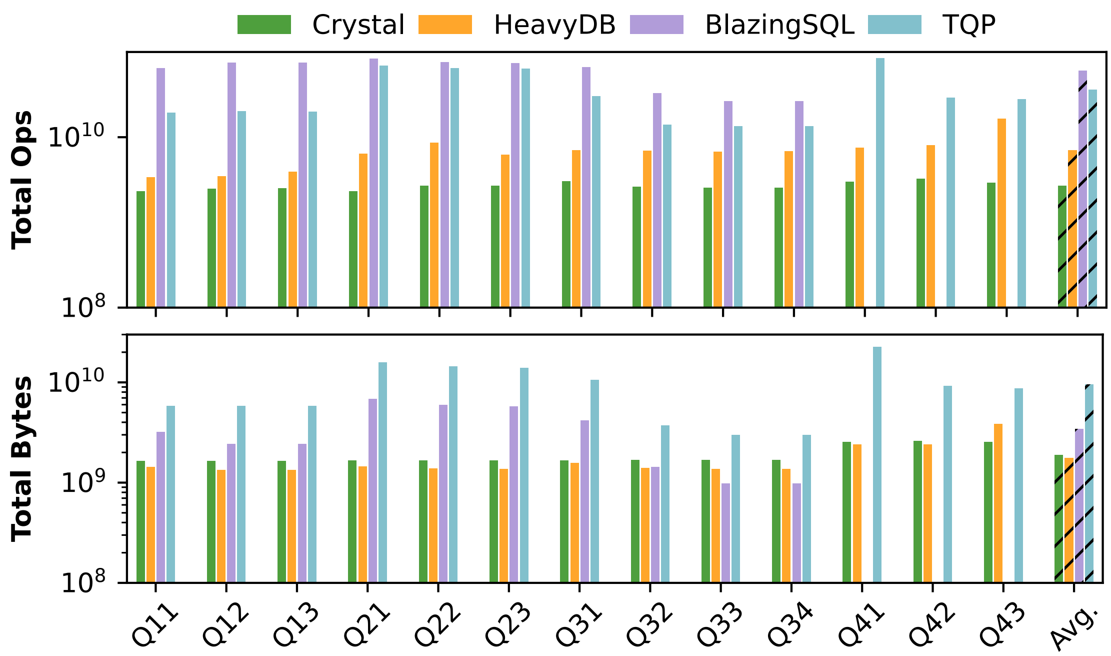


Figure 4.7: **GPU execution efficiency** – total integer operations (top) and number of bytes from DRAM (bottom).

pared to CRYSTAL and HEAVYDB. We also examine the number of bytes each system reads from the GPU DRAM (Fig. 4.7, bottom chart). Similar to the number of integer operations each system executes, CRYSTAL and HEAVYDB read less data from the GPU DRAM than the other two systems. For most queries besides Q43, CRYSTAL reads slightly more data from the GPU DRAM than HEAVYDB due to its bulk execution model. However, BLAZINGSQL and TQP read significantly more data from the GPU DRAM in most cases. Queries Q32, Q33, and Q34 are exceptions where the amount of data loaded from the GPU DRAM by BLAZINGSQL is equal to or even lower than that by CRYSTAL and HEAVYDB. Our investigation reveals that CRYSTAL does not fully avoid unnecessary data loading after predicates and joins.

General Purpose. Most of the systems are designed to be general-purpose, so they have additional operations other than the actual algorithm. For example, even though HEAVYDB is already very close to the optimal, it still has GPU algorithms to calculate the hash-table size for hash join on-the-fly, while CRYSTAL has a hard-coded hash-table size. Other systems have similar operations to set up needed data structures and meta-data.

Algorithmic Complexities. Optimal implementations like CRYSTAL mostly operate with a bitmap vector. It marks the bit to indicate whether the tuple satisfies predicates or joins without materializing the data. In contrast, systems like BLAZINGSQL largely depend on tuple indices. For example, it outputs the matched indices of the two tables when it does a join. As a result, it must materialize the tuple data again based on the indices. Additionally, general-purpose systems implement more complicated algorithms, such as MurmurHash for hashing operations. This is crucial to distribute data evenly, especially for an open-addressing linear hash table. Instead, CRYSTAL implements a perfect hash table, which does not make it practical to handle general-purpose use cases.

Kernel Fusion. There are two general approaches for kernel fusion. Systems like CRYSTAL and HEAVYDB write their own customized execution engine, allowing them to do rigorous kernel fusion. As a result, they can avoid many intermediate results, which in turn avoid a large number of bytes read from DRAM. On the other hand, BLAZINGSQL and TQP rely on existing third-party APIs for query execution. Those APIs are designed to be modular but are not customized for query execution. So, generating many intermediate results is often not avoidable. It is also very challenging to do rigorous kernel fusions if a system chooses to use this approach.

Extensibility. As previously mentioned, BLAZINGSQL and TQP have performance overheads since they reuse functions from other libraries to construct queries. Nevertheless, this approach allows the system to be easily extensible and portable. For instance, TQP uses PyTorch as its backend and can run on any hardware platforms that PyTorch supports (e.g., NVIDIA and AMD GPUs), integrates with ML tools [77], and supports multimodal and trainable queries [78].

Table 4.4: **Summary of most time-consuming kernels** – three most time-consuming GPU kernels for executing query Q21 on all systems.

		Top 1	**Top 2**	**Top 3**
CRYSTAL	Kernel	*probe_ht*	*build_htp*	*build_hts*
	Time (ms)	3.96 (99.35%)	0.01 (0.37%)	0.01 (0.37%)
HEAVYDB	Kernel	*multifrag*	*fill_hj*	*init_hj*
	Time (ms)	5.86 (91.78%)	0.51 (7.96%)	0.01 (0.19%)
BLAZINGSQL	Kernel	*probe_ht*	*comp_hj_output*	*parallel_fn*
	Time (ms)	13.42 (38.26%)	12.27 (34.97%)	5.55 (15.83%)
TQP	Kernel	*collect_fn*	*idx_select*	*gather_fn*
	Time (ms)	16.88 (40.88%)	9.55 (23.13%)	5.72 (13.86%)

Kernel Execution Time Breakdown (Q21)

Following our observation, we study GPU compute-time breakdown for query Q21 over the four systems as a micro-benchmark. Although we only report one query result, we find that the time breakdown between different queries is very consistent. Due to space limitations, we report only the three most time-consuming kernels. We note that BLAZINGSQL and TQP run 172 and 190 kernels for Q21, respectively. In contrast, both CRYSTAL and HEAVYDB run only four kernels.

CRYSTAL *and* HEAVYDB. These systems spend most of the query execution time on only the top kernel. As shown in table 4.4, more than 90% of the total compute time is spent on the top kernel. As discussed, this is because both systems implement kernel fusion, through which they execute as many operators as possible in each kernel to avoid the generation of intermediate results. Since only the hash-join operation is a *pipeline breaker* [79] for this particular query, they implement the hash-join build phase in a separate kernel (*e.g., build_htp* and *fill_hj*).

BLAZINGSQL. In this case, one query is split into many kernels. Among the different kernels, generated data (*e.g.,* bitmap to identify the selected rows) needs to be materialized in order to be visible to other kernels. This causes higher overhead compared to the other systems that do not need to materialize intermediate results. Results show that even the hash-table probing phase already exceeds the execution time of both CRYSTAL and HEAVYDB due to the algorithm complexity issue mentioned above.

TQP. Similar to BLAZINGSQL, TQP executes many kernels to complete one query, and in fact, it spends more time on intermediate results materialization than on actual query execution due to the limited functionalities of PyTorch. For example, PyTorch documentation indicates that *idx_select* function (second kernel) indexes a tensor to create a new tensor. It is used to create a new column after a predicate evaluation, but its overhead is high compared to the other kernels. The creation of intermediate results not only causes storage overhead due to the limited GPU device memory space but also incurs significant costs for the entire query execution.

4.5.2 Warp Execution Efficiency

We have quantified the number of operations and the amount of data each system processes for each query. We now turn to analyzing the impact of different implementations on execution efficiency by presenting microarchitecture-level performance statistics.

Warp Bottleneck

One advantage of GPUs is their ability to hide different types of execution stalls with their massive parallelism. If warp x encounters any stopping event (*e.g.,* waiting for instruction fetch or waiting for compute resources to become free), the GPU will quickly schedule other warps that are ready to be executed and have available resources. This approach improves the number of executed instructions per cycle. table 4.5 presents the average number of warps per scheduler per cycle in the GPU. For NVIDIA GPUs, each scheduler issues one warp per cycle at its most optimal state. Based on the table, we see that most systems have enough occupancy (*i.e.,* the GPU is occupied with enough warps). However, most systems do not have enough warps that are ready in time. For example, the eligible

Table 4.5: **GPU scheduler statistics for different warp states** – ACTIVE: total warps allocated. ELIGIBLE: warps ready to issue instruction (includes issued warps). STALLED: warps cannot issue instruction. ISSUED: warps selected by the scheduler. Table units are *average number of warps per scheduler per cycle.*

System	Active	Eligible	Stalled	Issued
CRYSTAL	11.08	0.16	10.92	0.1
HEAVYDB	7.94	0.16	7.78	0.14
BLAZINGSQL	8.33	1.97	6.36	0.72
TQP	12.52	0.45	12.07	0.18

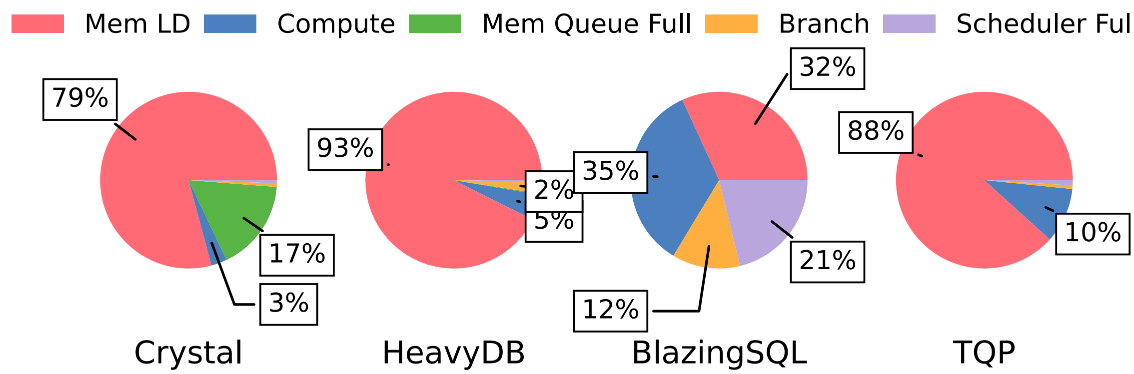


Figure 4.8: **Warp stall causes breakdown** – Q21 for all systems.

warp per cycle for CRYSTAL is 0.16. In other words, the scheduler does not have anything to schedule for five out of six cycles. We conclude that since most systems are issue-bound, computation resources after the issue stage are underutilized for OLAP queries. One exception is BLAZINGSQL, which has a high number of both eligible and issued warps.

Warp Stall Breakdown

We then attempt to reason about warp stalls by presenting a breakdown of their causes. The most notable observation, based on Fig. 4.8, is that most systems have the majority of stalls on memory load requests. For HEAVYDB and TQP, they have a small fraction of stalls on computation, meaning that the compute unit is saturated for very few operations. However, for BLAZINGSQL, due to its compute-intensive nature, it becomes more bottlenecked by the computing unit instead. For example, its warps cannot be scheduled due to the busy

Table 4.6: **GPU memory statistics** - Register usage, cache hit rate and number of cache requests for CRYSTAL and HEAVYDB.

System	Register per Thread	L1 Cache Hit (%)	L1 Cache Req. (M)	L2 Cache Hit (%)	L2 Cache Req. (M)
CRYSTAL	40	0.47	19	51.38	127
HEAVYDB	64	17.47	66	89.54	397

compute backend and also due to unavailable slots in the scheduler.

4.5.3 Memory Efficiency

The investigation in the last section shows that most systems are bottlenecked by memory requests. Thus, we analyze the memory efficiency of different systems. Our study focuses on CRYSTAL and HEAVYDB since they represent the most optimal implementations. Additionally, they have different execution approaches, so it is insightful to understand their trade-offs on memory efficiency.

table 4.6 shows the register and cache usage of both systems. The most notable observation is that the CRYSTAL (bulk execution model) has both worse L1 and L2 cache hit rates compared to HEAVYDB (warp execution model). Nevertheless, in theory, the bulk execution model should have better memory efficiency because of better memory coalescing and better temporal cache reuse. Our study identifies that CRYSTAL has two limitations that cause worse memory efficiency. First, CRYSTAL loads unnecessary data from DRAM, which disrupts its cache locality. Secondly, for CRYSTAL (bulk execution model), it is also more favorable to not bypass the L1 cache, so that data loaded together (*i.e.,* a bulk or a tile) can be reused temporally (CRYSTAL chooses to bypass L1 cache).

4.6 Implementation Optimization

In this section, we discuss optimizations that we have implemented based on the insights gained from the previous analysis.

4.6.1 Inefficiencies and Optimizations

As mentioned in § 4.2.2, CRYSTAL uses a bulk execution model to improve cache locality. A column is partitioned into multiple tiles, with each tile containing four tuples. This yields the most optimal query execution efficiency.

Extra Bytes and Operations. The original CRYSTAL work implements a tile-based data loading primitive that loads all tuples for a tile from the GPU DRAM. After evaluating a predicate over a column, it is common for many tuples to become unnecessary, as they do not satisfy the predicate. In this case, the design of the CRYSTAL load primitive is suboptimal as it loads all tuples from the GPU DRAM, even the ones that are not needed. Considering that analytics query execution is mostly stalled due to memory load (§ 4.5.2), the extra data loading can cause a performance slowdown. Furthermore, as discussed in § 4.3.1, GPU execution operates in warp granularity, with all threads (*i.e.,* 32) executing instructions in lockstep. During query execution, many threads in a warp find that not all of their assigned tuples satisfy predicates after evaluating a few where clauses, especially for very selective predicates. CRYSTAL does not have any primitive to terminate those threads in time, which causes unnecessary instruction execution on the GPU.

To address this issue, we implement two primitives in CRYSTAL: a predicated loading primitive (*PredLoad*) and a voluntary runtime thread termination primitive (*Term*). Our primitives keep track of valid tuples and idle threads for each tile using a bitmap. During execution, our new primitive performs on-demand tuple loading based on the validity of the tuple stored in the bitmap. In doing so, we avoid loading tuples that have already been invalidated by predicates on other columns. We also monitor the liveness of the thread through the bitmap. If all tuples of a thread become invalid, the execution chooses to end the thread immediately to avoid executing further instructions. This optimization is particularly beneficial when the entire warp can be terminated early.

Cache Locality. Historically, GPU L1 caches have not been used (L1 cache bypass) dur-

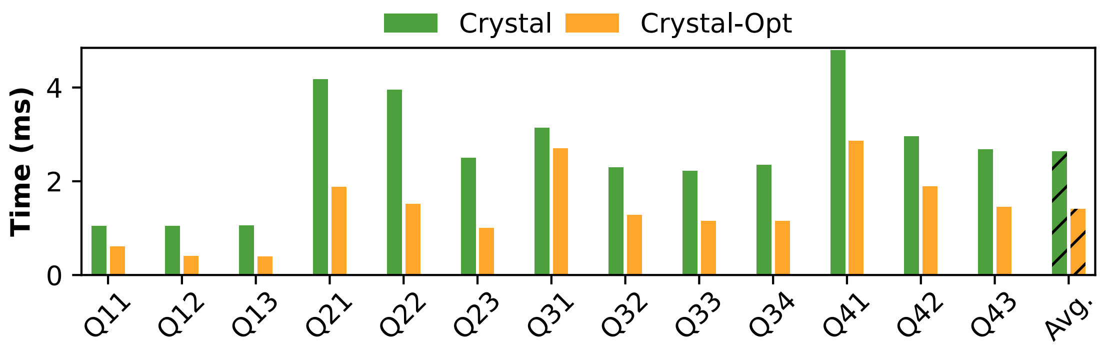


Figure 4.9: **CRYSTAL vs CRYSTAL-Opt. query execution.**

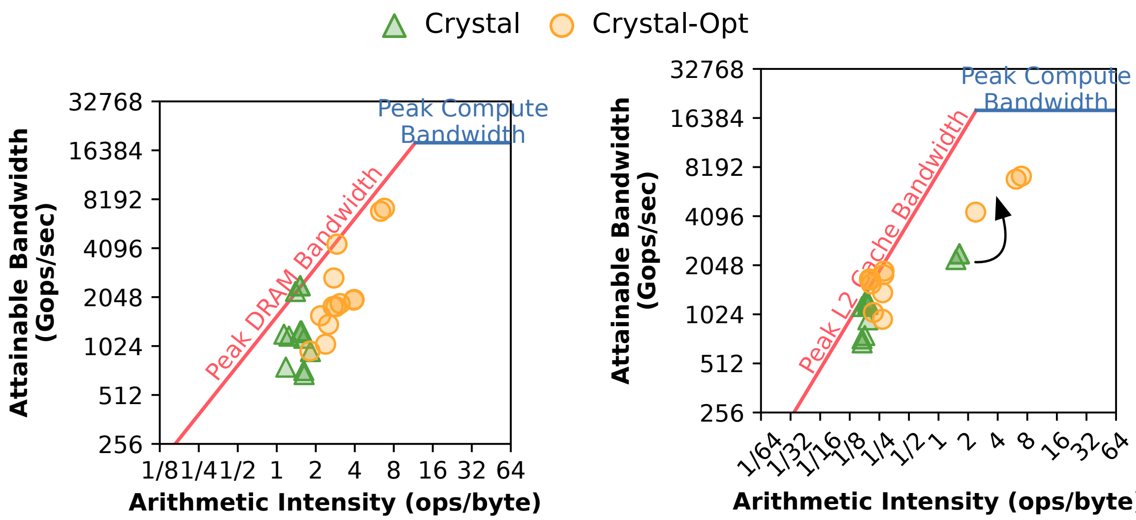
Table 4.7: **Performance statistics of CRYSTAL and CRYSTAL-Opt** for Q21—Total bytes, total operations, cache hit rate, number of cache requests.

System	Total Bytes (M)	Total Ops (M)	L1 Cache Hit (%)	L1 Cache Req. (M)	L2 Cache Hit (%)	L2 Cache Req. (M)
CRYSTAL	1653	2704	0.47	18	51.49	127
CRYSTAL-Opt	902	2291	41.31	14	67.48	66

ing execution because of their limited capacity, particularly for workloads involving many random accesses. However, in our experiments, we discover that the increased L1 cache capacity can bring additional performance benefits by improving the temporal data locality. This is especially true since data in a tile tends to be reused often in the bulk execution model. In contrast to CRYSTAL's approach, we enable the L1 cache by adjusting the compilation flag supported by CUDA GPUs.

4.6.2 Performance Speedup

We combine all proposed optimizations as CRYSTAL-Opt. In Fig. 4.9, we compare the query execution time of both CRYSTAL and CRYSTAL-Opt. CRYSTAL-Opt achieves an average speedup of $1.9\times$ over CRYSTAL. The speedup is more significant for selective queries, but not as prominent for less selective queries (*e.g.,* Q31). We also conduct a case study for Q21 to understand the reasons for the speedup (shown in table 4.7). First, we



(a) DRAM roofline Model.

(b) L2 roofline Model.

Figure 4.10: **Resource utilization between CRYSTAL and CRYSTAL-Opt** – DRAM and L2 cache utilization.

can see that by using the newly introduced primitives, CRYSTAL-Opt significantly reduces the total bytes and operations compared to CRYSTAL. As a result of the reduced total data read from DRAM, the number of requests to the cache system is also reduced. Second, by enabling the L1 cache, CRYSTAL-Opt increases the cache hit rates for both L1 and L2 caches. Due to the improved L1 cache hit rate, the total number of requests to the L2 cache is also reduced.

We also investigate the impact of CRYSTAL optimization on resource utilization of both DRAM and L2 cache. Fig. 4.10 demonstrates that the AI for GPU DRAM is increased because our optimization reduces unnecessary data brought from GPU DRAM. As a result, the maximum possible attainable bandwidth is also increased, especially for queries Q11, Q12, and Q13. For L2 cache utilization, our optimization on cache locality also improves the attainable bandwidth of queries that have hash join. Compared to CRYSTAL, CRYSTAL-Opt now has more queries that fully saturate the maximum L2 cache bandwidth. Even for queries that do not have hash join (the right-most three dots in Fig. 4.10), their attainable bandwidths are also improved by having better cache locality.

4.7 Resource Utilization Optimization

Our characterization in § 4.5 demonstrates that many queries in GPU DBMSs do not fully saturate the existing GPU resources. As shown in § 4.6, one way to improve resource utilization is to optimize the system implementation. However, with the resource-allocation features in newer generation GPUs (e.g., MIG [16]), DBMSs can also improve resource utilization by running multiple queries concurrently inside a single GPU, in an implementation-agnostic way.

One unsolved challenge of doing concurrent query execution is that it does not always offer performance speedup, especially when a GPU resource is already saturated. Therefore, we need an analytical model to reason about performance improvements given the available GPU resources. In this section, we demonstrate that we can use the roofline model as a guideline to reason about performance speedup when the allocated GPU resources are adjusted.

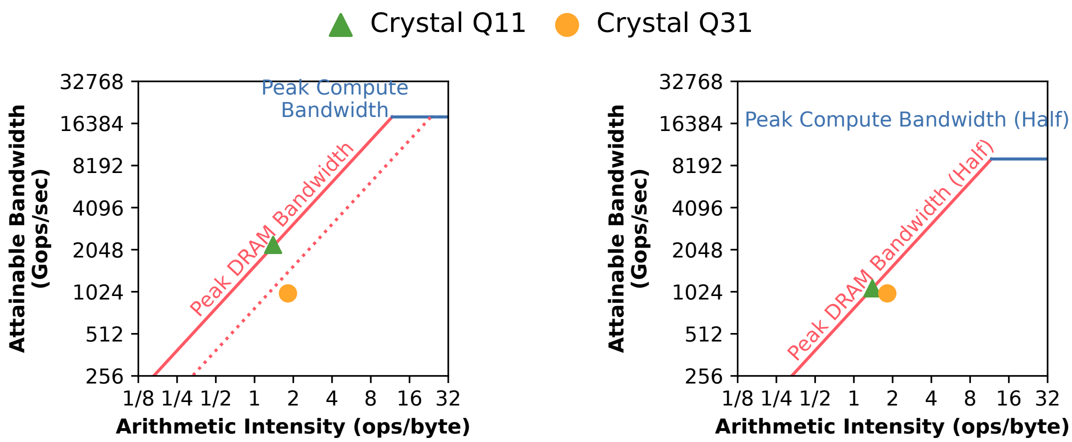
4.7.1 Model-Driven Resource Allocation

We show how to use the roofline model to estimate the GPU execution performance changes with respect to resource allocation, which can then be used to determine the optimal configuration. The model uses runtime information, which can be obtained from prior executions of recurring queries or from a representative workload.

We use GPU DRAM as the target resource for illustration. Let t denote the query time under the current resource allocation and let $\mathrm{Bandwidth_{DRAM'}}$ denote the new DRAM bandwidth. We predict the new query GPU execution time $\mathrm{t'}$ by using the following equation.

$$\mathrm{t'} = \max\left(\mathrm{t}, \frac{\text{\# of Int. Ops}}{\mathrm{AI_{DRAM}} \times \mathrm{Bandwidth_{DRAM'}}}\right)$$

This equation is proposed based on the observation that *AI is determined by the implemen-*



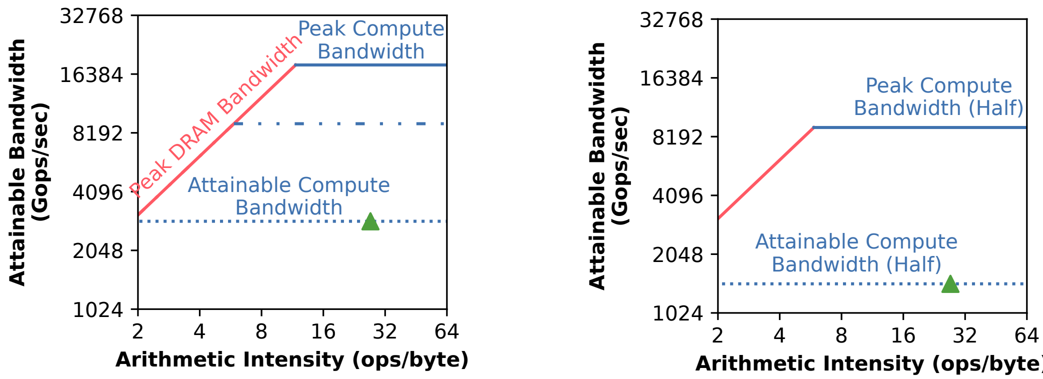
(a) Query performance, memory bandwidth, and compute bandwidth of *full* GPU resource (*dashed line is projected memory bandwidth of half GPU resource*).

(b) Query performance, memory bandwidth, and compute bandwidth of *half* GPU resource allocation.

Figure 4.11: **Memory-bound queries** – Performance impact on memory-bound queries (Q11 and Q31) for CRYSTAL.

tation of the query and is unlikely to change when the resource allocation changes. Our model picks the maximum among the two terms in the equation—the first term for scenarios where bandwidth is not a bottleneck (and so, the time remains unchanged), and the second for scenarios where the change in memory bandwidth hurts performance. For the latter case, the denominator in the fraction is the maximum throughput at the given AI and allocated bandwidth. In this case, the query time is the time to execute the total integer operations at that throughput. Fig. 4.11 shows these two scenarios using representative queries (Q11, Q31) from CRYSTAL, and full $\rightarrow$ half allocation change. Q31 underutilizes the DRAM bandwidth and has no performance impact, whereas Q11 loses throughput (geometrically, the point shifts downwards) since it saturates the bandwidth. Finally, we compute $\mathrm{Slowdown_{DRAM}} = \mathrm{t}'/\mathrm{t}$. Visually, in Fig. 4.11, the green dot moves down but not the yellow dot (no slowdown).

L2 Bandwidth. As previously discussed, the latest GPUs [73] support both L2 and DRAM bandwidth partitioning. Similar to the above method, we can estimate the execution time due to L2 resource allocation using the L2 roofline model. One challenge is how to com-



(a) Query performance, memory bandwidth, and compute bandwidth of *full* GPU resource (*dashed-dotted line projects compute bandwidth of half GPU resource*).

(b) Query performance, memory bandwidth, and compute bandwidth of *half* GPU resource allocation.

Figure 4.12: **Compute-bound queries** – Performance impact on compute-bound queries (Q34 from BLAZINGSQL).

bine both the DRAM and L2 roofline models into a unified model to estimate the query slowdown. We solve this by using a max function to estimate the total slowdown as follows.

$$\text{Slowdown} = \max\left(\text{Slowdown}_{\text{DRAM}}, \text{Slowdown}_{\text{L2Cache}}\right) \tag{4.1}$$

The rationale is that we empirically find that *queries can rarely be bottlenecked by both memory resources (L2 and DRAM)*. For example, if a query has very high utilization of the L2 cache bandwidth (*i.e.,* it is bottlenecked by the L2), it generates only minimal traffic to DRAM, so it will not be affected by changes in DRAM bandwidth. Hence, one of the estimated slowdown terms is likely to be 1 (no slowdown). Thus, we only need to use the $\max$ function to get the dominating slowdown value.

Compute-Bound. Next, we explain slowdown estimation for compute-bound queries using BLAZINGSQL, because it has the most compute-bound implementations compared to the other systems.

Fig. 4.12 left shows Q34 attainable bandwidth (*i.e.,* throughput) and AI (2886.74 Gops/sec and 27.15 ops/byte respectively). The peak compute bandwidth of the full GPU is 18247.00 Gops/sec and that for half of the GPU resources is near 9123.50 Gops/sec (dashed and dotted line in Fig. 4.12 left). It is clear that even the peak compute bandwidth for half of the GPU is beyond the attainable bandwidth of Q34, so the traditional roofline model would predict no performance slowdown. Nevertheless, this is not the case as a result of the attainable compute bandwidth per SM being less than its peak due to execution inefficiencies (*e.g.,* memory stalls). The overall compute bandwidth (attainable or peak) is the per-SM value $\times$ the number of SMs. When the GPU allocates fewer compute resources, it reduces the number of SMs allocated, but it cannot improve the execution efficiency (*i.e.,* attainable compute bandwidth) of each SM. As a result, the overall attainable compute bandwidth will decrease (Fig. 4.12(b)). To estimate the resulting slowdown, we can simply use the ratio of resource allocations reduction as follows.

$$\mathrm{Slowdown}_{\mathrm{Compute}} = \frac{1}{\mathrm{ComputeAllocationRatio}} \tag{4.2}$$

For example, if the GPU compute resources are halved, the attainable bandwidth can be calculated as half of the original attainable bandwidth with full GPU resources.

Unified Model. Now that we have proposed two models for estimating slowdowns with changing allocations—one for memory resources and one for compute resources, the last step is to determine which model to use. We use a simple, commonly-used heuristic [80, 64] to determine whether an application is compute- or memory-bound. As shown below, we can determine if an application tends to be compute-bound based on the AI and peak compute and DRAM bandwidths of the GPU. The final $\mathrm{Slowdown}$

$$= \begin{cases} (2), \text{if } \mathrm{AI}_{\mathrm{DRAM}} > \frac{\mathrm{Bandwidth}_{\mathrm{Compute}}}{\mathrm{Bandwidth}_{\mathrm{DRAM}}} \text{or } \mathrm{AI}_{\mathrm{L2Cache}} > \frac{\mathrm{Bandwidth}_{\mathrm{Compute}}}{\mathrm{Bandwidth}_{\mathrm{L2Cache}}} \\ (1), \text{otherwise} \end{cases}$$

If the application is more compute-bound, then we use the model to account for compute bandwidth reduction. Otherwise, we apply the model for DRAM or L2 cache bandwidth reduction.

This approach can be more easily used to (1) estimate the GPU execution performance impact of downsizing both memory and compute resources, or (2) reason about the performance impact of upsizing compute resources. However, it could have inaccurate estimations for the performance impact of upsizing memory resources when they are no longer a bottleneck.

4.7.2 Model-Driven Concurrent Scheduling

Next, we extend the model to estimate the end-to-end performance impact for different degrees of concurrency. Using our analytical model, users can accurately predict the resulting end-to-end performance when running queries concurrently on multiple GPU partitions. Our model assumes that the scheduler uses a round-robin scheduling algorithm for load balancing.

CPU and Constant Overhead. To construct the model, we first need to consider additional overheads for query executions. For CPU overhead, our model includes the overhead of query optimization and compilation. For some systems (*e.g.,* HEAVYDB, BLAZINGSQL), even though the same query has already been optimized and compiled to binary form, each query invocation still introduces some constant overhead on the CPU side. For all systems, we also consider those overheads. Later, we show insights into how concurrency is also beneficial to alleviate those overheads. We consider two additional major overheads—GPU setup overhead, which includes GPU context initialization and memory allocations, and data transfer overhead. All systems cache tables on the GPU for future query executions, so the data transfer overhead is also only a one-time cost.

End-to-End Performance. We can now estimate the end-to-end query execution time individually for each process. When the system varies the resources, the model will adjust the query GPU execution time. Other overheads will remain unchanged. Now, to consider the time from multiple concurrent processes P, we use a $\max$ function because the longest-running process will determine the end-to-end query execution time for concurrent executions.

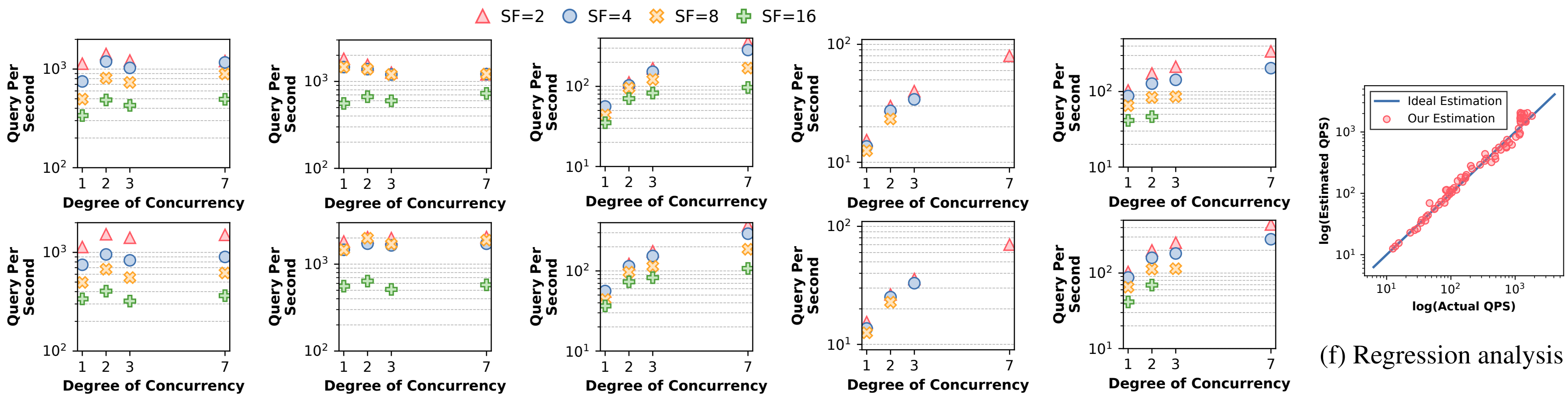
$$\mathrm{ExecTime} = \max\left(\mathrm{ExecTime}_{\mathrm{P}_1}, \mathrm{ExecTime}_{\mathrm{P}_2} \ldots \mathrm{ExecTime}_{\mathrm{P}_\mathrm{n}}\right)$$

Our model can also be used to estimate query performance *vs.* degree of concurrency for MPS (§ 2.2), excluding accounting for interference in accessing the shared L2 cache and DRAM.

4.7.3 Resource Allocation Study

In this study, we focus on answering the following questions:

1. **RQ1** – How beneficial is concurrent query execution?
2. **RQ2** – How good is our model for estimating the query execution performance *vs.* the degree of concurrency?
3. **RQ3** – What are the trade-offs between MIG and MPS?



(a) CRYSTAL (b) CRYSTAL-Opt (c) HEAVYDB (d) BLAZINGSQL (e) TQP (f) Regression analysis

Figure 4.13: **Throughput *vs.* degree of concurrency** – Comparison between actual and estimated throughput of running queries concurrently (we omit data points for BLAZINGSQL and TQP where they run into out-of-memory issues at larger scale factors). For sub-figures (a), (b), (c), (d), and (e), the top plot shows the actual throughput whereas the bottom plot shows the estimated throughput.

4.7.4 Benefit of Concurrent Query Execution

We implemented concurrent query execution on the four systems and designed the following experiment to evaluate the benefits with Degree of Concurrency (DoC) = 2, 3, and 7, over DoC=1 (no concurrency), for SSB queries [2]. We configured the GPU, through MIG, to support the required DoC and started the corresponding number of system instances, one on each GPU partition. We construct a simple scheduler to dispatch SSB queries in a randomized sequence to each system instance, repeated 1000 times. We then measure the overall throughput (queries per second *i.e.,* QPS) for the GPU.

Fig. 4.13 (top) shows the actual (measured) throughput for different DoC. For large-scale factors (*e.g.,* 8 and 16), we omit query throughput for systems that run into out-of-memory issues. For CRYSTAL, the throughput initially increases by an average of $1.5\times$ as the DoC changes from 1 to 2. Because CRYSTAL spends the majority of the time on GPU execution, the performance speedup mostly comes from the improved resource utilization of the GPU hardware. However, as the DoC increases beyond 2, it does not gain further speedup, because resources are already very close to being fully saturated when running two processes concurrently. In fact, for DoC=3, the throughput is lower than for DoC=2. A further reason is that MIG cannot evenly divide the GPU resource into three partitions. For instance, some of the GPU DRAM banks and L2 cache slices are not being used at all when DoC=3.

CRYSTAL-Opt achieves a $1.3\times$ throughput improvement compared to CRYSTAL-Opt without MIG at scale factor 16, while it does not gain better throughput at smaller scale factors. This is because GPU execution time dominates other overheads for larger scale factors. Increasing the DoC initially reduces GPU execution time, but since GPU resource utilization is quickly saturated, a higher DoC cannot further improve the overall throughput. In contrast, other overheads like data and results movement dominate end-to-end execution

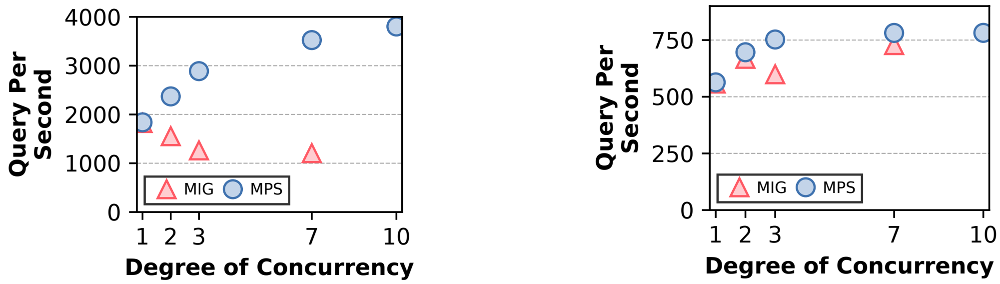
[2]For A100 MIG partitions [14], compute resource has a total of 7 slices but memory resource has a total of 8 slices. When the GPU is partitioned, some resources will end up being idle because they cannot be evenly distributed. We choose the DoC that minimizes the amount of idle GPU resources when partitioned.

for smaller scale factors. Partitioning the GPU also partitions the PCI-e bandwidth, so the overhead grows proportionally as the DoC increases. We also observe that CRYSTAL-Opt with MIG has a $2.2\times$ speedup over CRYSTAL with no MIG, while CRYSTAL only has a $1.5\times$ speedup after enabling MIG. This demonstrates that both system implementations and resource allocations are critical.

Concurrent execution provides sub-linear performance improvement for both HEAVYDB and BLAZINGSQL. For those two systems, queries at small scale factors do not fully saturate resources. More queries can finish within a fixed time with concurrent execution, so the overall throughput increases. In contrast, queries are more easily affected by resource reduction at larger scale factors, leading to a significant increase in query latency that overshadows the gains of concurrent execution. However, concurrency is still able to hide CPU latency overhead, leading to substantial throughput improvements for SF=8 and SF=16, respectively, at DoC=7 (over DoC=1). Concurrent query executions also provide sub-linear performance speedup for TQP. TQP is shown to have low GPU utilization, so most of the performance gain comes from better utilization.

4.7.5 Model Accuracy on Degree of Concurrency

Fig. 4.13 (bottom) shows the estimated overall throughput vs. DoC for the above experimental setting. Fig. 4.13 (f) indicates the prediction accuracy of the model compared to the ideal prediction (estimated = actual). We use our models to estimate end-to-end query execution time for all DoC using profiling data from a single run of the query with full GPU allocation (DoC=1). Our estimated throughputs are close to the actual, for both absolute values and scaling trends. The correlation coefficient between actual and estimated throughput is 0.95, thereby indicating strong correlation. The 50^{th} and 95^{th} percentile of relative errors ($= \frac{|\text{Actual}-\text{Estimated}|}{\text{Actual}}$) are ≤ 0.11 and ≤ 0.46, while the average is 0.15. The evaluation demonstrates that our model is able to help system administrators determine how to allocate GPU resources and configure the level of query execution concurrency to



(a) SF=2

(b) SF=16

Figure 4.14: **Trade-offs between MIG and MPS** – Throughput *vs.* degree of concurrency with different mechanisms tested on CRYSTAL-Opt.

achieve the best performance.

4.7.6 MIG and MPS Trade-offs

We present an analysis (Fig. 4.14) to compare the trade-offs of executing a batch of queries between MIG and MPS on CRYSTAL-Opt. For SF=2, MPS offers better scalability than MIG. Since all clients share the same PCI-e interconnect, MPS allows different processes to overlap compute with data transfer with the full interconnect bandwidth, considering the fact that the bandwidth is not fully saturated temporally. In contrast, the PCI-e bandwidth is partitioned among different processes in MIG, and thus, each process gets reduced PCI-e bandwidth. For SF=16, MIG and MPS show similar performance. This is because compute overhead dominates at larger scale factors, and thus the benefit of interleaving reduces. In general, MIG is favored for scaling performance without interference between partitions, especially when GPU execution time dominates. In contrast, MPS provides better data and computation overlap when data movement overhead dominates the end-to-end execution time.

4.8 Related Work

Suh *et al.* [58] is the most recent study to compare different GPU database systems. They only focus on end-to-end execution time comparison of different systems on different

GPUs. However, our paper shares more insights into the internals of GPU execution efficiency. Yin and Yang [81] study GPU execution of SSB [82], and present the time breakdown between kernels and data transfers, but never examine the efficiency of each kernel. Pump up the Volume [83] and Triton join [84] study the performance impact from newer CPU-GPU interconnects on out-of-GPU hash-join. GPL [48] also observes that the GPU can be underutilized during query execution. Furst *et al.* [57] study GPU kernel efficiency but only focus on comparing GPU occupancy *vs.* instruction types. Funke *et al.* [49, 85] and Paul *et al.* [86] optimize JIT compilers for better GPU execution efficiency through kernel fusions and thread divergence elimination. Sioulas *et al.* [87] implement partitioned hash join in GPU. Shanbhag *et al.* [88] also extend the CRYSTAL library for in-GPU compression. Instead, HippogriffDB [89] accelerates query performance by supporting GPU execution on compressed data. Mordred [90] and HetExchange [91] explore CPU-GPU query executions. Rosenfeld *et al.* [92] provide an in-depth overview of CPU-GPU database systems. GaccO [93] studies transactional query processing on GPUs. MGJoin [94] and Maltenberger *et al.* [95] evaluate join and sorting algorithms for multi-GPU systems. Doraiswamy and Freire [96] propose using GPUs to process spatial data (*e.g.,* geometric objects) in database systems. Our work provides an in-depth microarchitectural analysis of existing systems.

GPU performance modeling. Hong and Kim [97] model the performance of early generation GPUs, where GPU caches were still not mature, so the focus is on GPU DRAM and compute bandwidths. Zhang *et al.* [98] provide performance optimization suggestions to developers through GPU performance modeling with micro-benchmarking profiling. Wu *et al.* [99] instead propose using a machine learning approach to predict kernel performance, which also requires profiling performance counters on real hardware. Baghsorkhi *et al.* [100] use code analysis to consider the performance impact from control flow divergence and memory bank conflicts. Our approach of using the roofline models provides a simple way to analyze the performance impact for different GPU resources without requiring code analysis or machine learning models.

Gables [101] uses roofline models to study SoC platforms with multiple accelerators. Ding *et al.* [102] and Lopes *et al.* [103] also apply roofline models to GPUs. Ding *et al.* capture all instructions for AI, which can lead to inaccurate estimations. Lopes *et al.* also explore the cache-aware aspect but do not differentiate between AI at the cache level and at the DRAM level, in contrast to our work. Additionally, we extend and apply the roofline model to estimate query performance for different resource allocations on the GPU. This has not been explored in prior work.

Concurrent execution in GPUs. Yu *et al.* [104] have also surveyed the trade-off between MIG and MPS. Their paper shares the potential opportunities and use cases of using concurrent execution in GPUs. Tan *et al.* [105] explore accelerating deep neural network (DNN) inference by using MIG. Kass *et al.* [106] instead investigate DNN training in MIG. In our work, we support relational query operations in the form of concurrent execution, which is often limited by the GPU memory resource. Instead, DNN inference is more compute-bound. Our work uses simple models to estimate performance with different GPU resource limits, whereas others [105] require profiling the execution for different configurations.

4.9 Conclusion

In this work, we provide microarchitectural insights for existing GPU database systems. We demonstrate how using our insights, we improve the performance of a state-of-the-art database system by $1.9\times$. Additionally, we show how these insights can be further leveraged into an analytical model—predicting resource utilization under concurrent query execution—delivering up to a $6.5\times$ better throughput. We expect these contributions to drive further research and development in the field of GPU-accelerated data analytics.

CHAPTER 5

AERO: ADAPTIVE QUERY PROCESSING FOR ML QUERIES

5.1 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 [107, 108, 109].

State-of-the-art ML-centric DBMSs [12, 108, 109, 110] 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 [107], motivating the need for adaptive query processing (AQP).

AQP leverages runtime feedback to dynamically adjust query plans [7, 8, 5]. 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 5.1: Query to capture unsafe operating situations.

The query in listing 5.1 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 [111] 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 [107] or estimate them by running the UDFs over sample data [108, 109]. 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 [107] 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 $1.3\times$ 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 make the following contributions in this work.

- 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 $6.4\times$ speedup compared to state-of-the-art static optimization approaches, without impacting accuracy.

5.2 AERO

In this section, we first explain the principles that we follow in designing the AERO in § 5.2.1. We then explain the internal architecture of the AQP operator in § 5.2.2 using example query listing 5.1.

5.2.1 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 instruments 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. 5.1. 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. 5.1, 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.

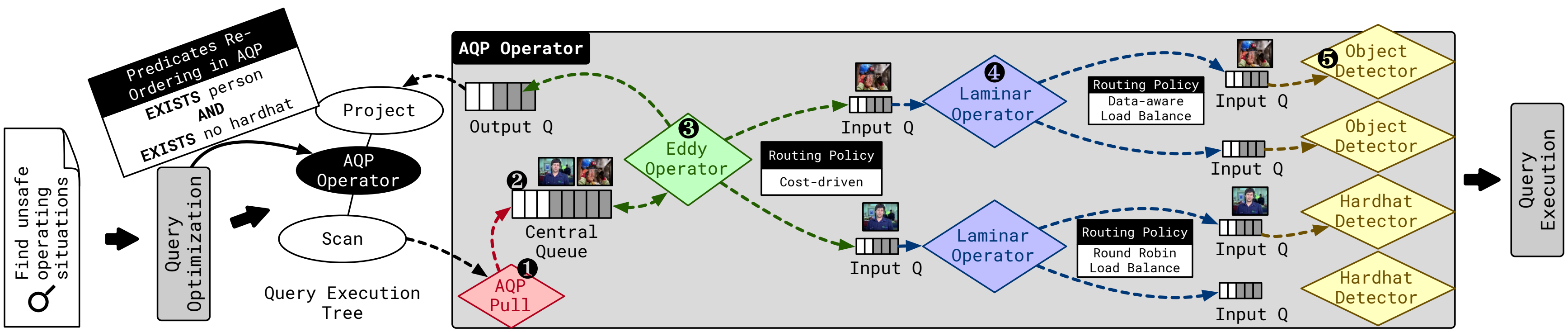


Figure 5.1: **Detailed AQP execution plan and its internal** – left shows the execution tree with the AQP operator. *Diamonds* represent physical processes separate from the main process. All physical queues facilitate communication between data producers and consumers. *Rectangles* represent the routing 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.

5.2.2 Architecture of AQP Operator

We take a deep dive into the internal architecture of the AQP operator, following our query example in listing 5.1. 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 (§ 5.3.1) 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 (§ 5.4.2) 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.

5.3 EDDY Operator

In this section, we explain the design of the eddy operator in AERO. Its choice of routing policy (§ 5.3.1) and the adaptiveness of its statistics monitoring (§ 5.3.2) are particularly significant, both of which are pivotal in achieving optimal predicate ordering.

5.3.1 Cost-Driven Routing

In AERO, the eddy operator primarily monitors two key statistics: the cost and selectivity of predicates, drawing inspiration from prior works [8, 7]. 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 [7, 9, 8] 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 [111]. The predicate with the lowest score is prioritized for execution first, contributing to optimal performance.

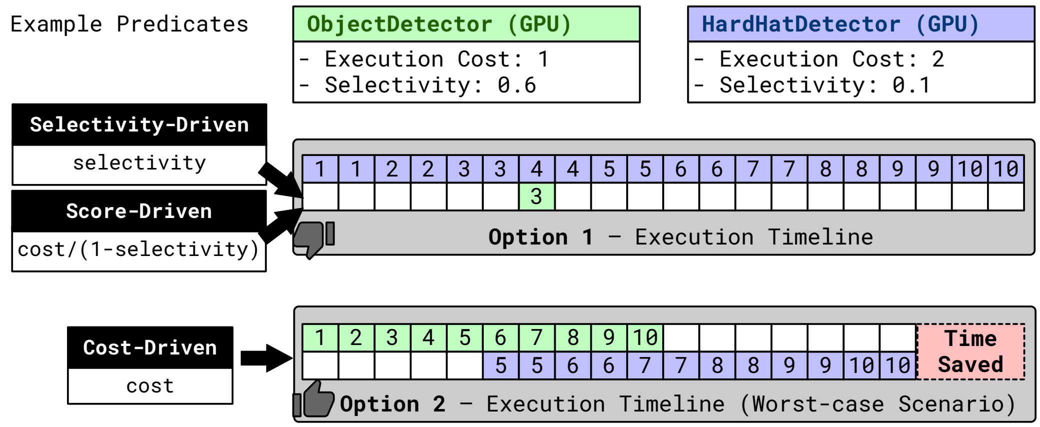


Figure 5.2: **Execution timeline of different routing policies –** a comparison of selectivity-driven, score-driven, and cost-driven routing policies. One box represents a time unit. The Cost-driven routing policy offers a 3 time-unit saving even in the worst-case scenario.

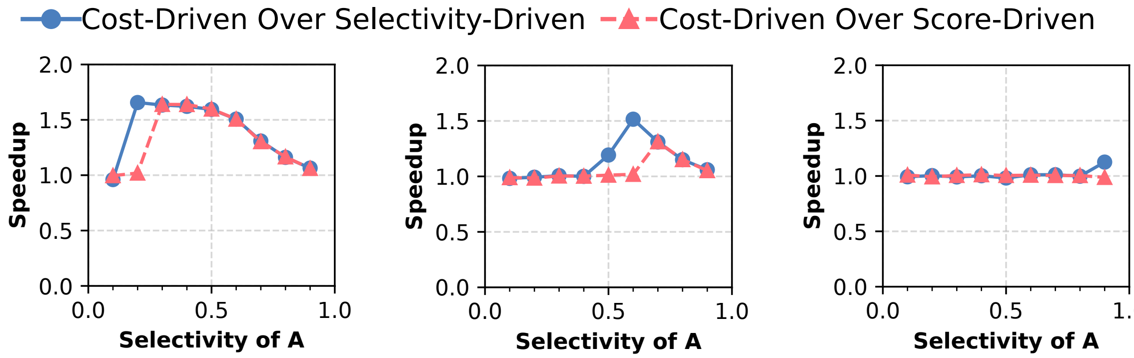
However, our findings indicate that the scoring function ($\frac{\text{cost}}{1-\text{selectivity}}$) for predicate ordering is not optimal when dealing with concurrent execution. Selectivity is critical in sequential execution because higher selectivity of a predicate can significantly shorten the execution 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. 5.1) 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. 5.3) show that the cost-driven routing policy outperforms the score-driven routing policy.

Reasoning with Examples. Our example query (listing 5.1) 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. 5.2).

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 `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. 5.2). 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



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

Figure 5.3: **Controlled experiment** – speedup comparison of the cost-driven routing policy over other policies under different selectivities.

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 [112, 113] 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 [112, 113], 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. 5.3a, Fig. 5.3b, and Fig. 5.3c, 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 5.1), 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 (§ 5.6.2).

5.3.2 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;
```

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

Listing 5.2: Query to identify unsafe situations in a warehouse.

Motivating Example. listing 5.2 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 [107] 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 § 5.3.1. 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.

5.4 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 § 5.6.5, under-utilization of GPUs can degrade performance by a factor of $6.0\times$. To address this, we introduce the LAMINAR operator following the EDDY operator, illustrated in Fig. 5.1. The LAMINAR operator enhances query performance by (1) ensuring optimal hardware utilization, particularly for GPU resources (§ 5.4.1); (2) facilitating robust scalability as the system increases the number of resources (§ 5.4.1); and (3) achieving effective load balancing among multiple backend workers during query execution (§ 5.4.2).

5.4.1 Hardware Utilization and Scalability

The utilization of the underlying hardware significantly impacts query performance. In queries like listing 5.1, 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 [114, 115] emphasizes the importance of GPU utilization for achieving high throughput. While most works [116, 117, 118, 119, 120] 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 [120] 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 [121] 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.

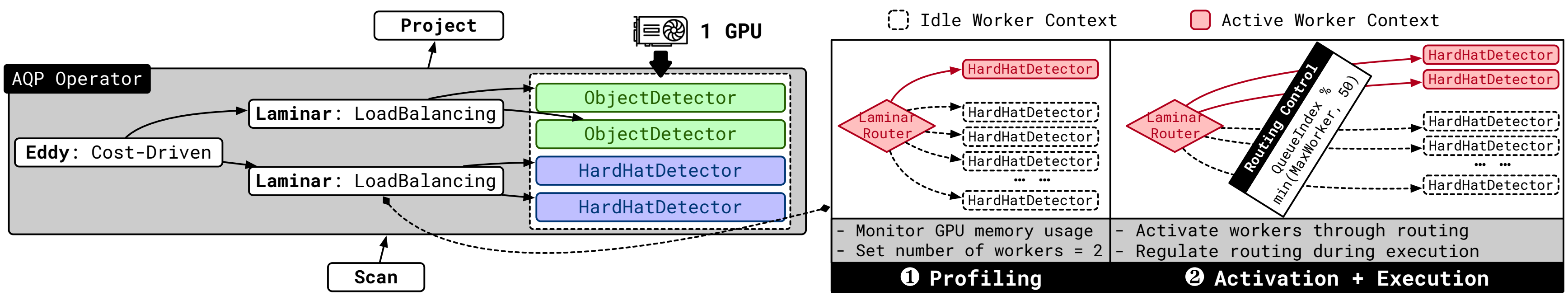


Figure 5.4: **LAMINAR workflow for listing 5.1** – the left side illustrates two-way parallelization enabled by the LAMINAR operator with a single GPU. On the right, the figure shows the query execution beginning with a profiling phase to assess worker hardware utilization, followed by the activation of more workers.

Batch-agnostic Parallelization. Inspired by the spatial-multiplexing approach from previous works [114, 120], 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 5.1 with two-way parallelization is depicted in Fig. 5.4. 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 LAM-

INAR operator activates more predicate workers by routing data to them, without needing to adjust the query pipeline during execution.

As demonstrated in Fig. 5.4, 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(§ 5.6.5), we find this approach can achieve a good load balancing between GPUs and consequently improve the overall GPU utilization.

5.4.2 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 § 5.1. 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 [122] or least connection [123], 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.

5.5 Implementation

AERO augments EVADB [12], an ML-centric database system, to support AQP. We implement routers and workers using the Ray [124] 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 (§ 5.3.1) 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 § 5.6.5 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 which 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.

5.6 Evaluation

In this section, we answer the following research questions:

RQ1 – How does the query processing performance of AERO compare to other baseline systems (§ 5.6.2)?

RQ2 – How effective are AERO's routing policies, including cost-driven routing, adaptive routing, and data-aware routing(§ 5.6.3)?

RQ3 – How efficient is AERO when scaling up resources compared to other model-serving frameworks (§ 5.6.4)?

RQ4 – What is the performance contribution of EDDY and LAMINAR in AERO (§ 5.6.5)?

RQ5 – What are the overheads of AERO (§ 5.6.6)?

5.6.1 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 [125]. 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 [107, 126], 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 [127] 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 [128]. `HardHatDetector` detects hard hats with a fine-tuned YOLOv8s model. `AgeClassifier` and `EmotionClassifier` employ ViT-based transformers for age [129] and emotion [130] classification, respectively. The `LLM` predicate uses Orca, a local large language model with 13 billion parameters from the GPT4All [131] library, optimized for CPU execution with multi-threading capabilities. table 5.1 lists the average selectivity and cost for the predicates in the workload.

Baselines. We use the following baselines to compare with this work.

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 § 5.6.5.

Table 5.1: **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

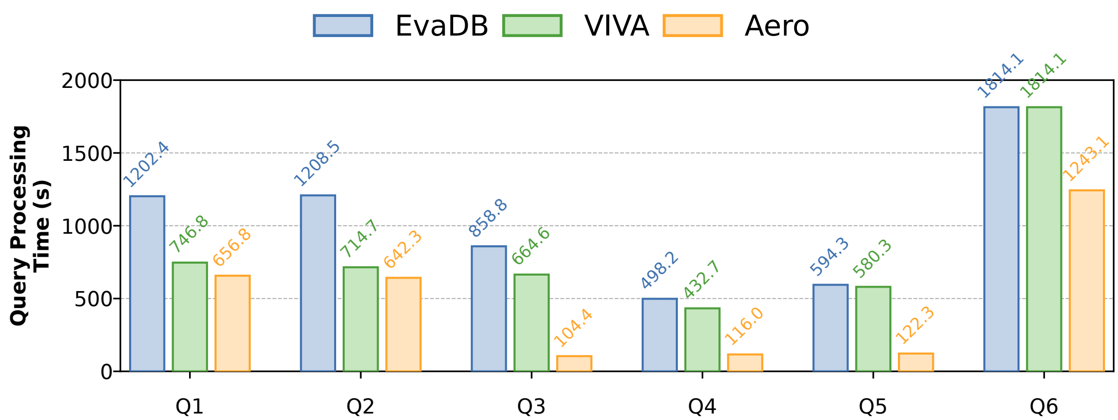


Figure 5.5: **End to end performance** – a comparison between EVADB, VIVA, and AERO on Q1-Q6.

5.6.2 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 $1.83\times$ and $1.14\times$, 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 $1.88\times$ 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 the UDFs (30.35 ms vs 0.95 ms, table 5.1), 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 $8.22\times$ and $6.37\times$, 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 $4.86\times$ and $4.75\times$, 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 $1.46\times$, 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.

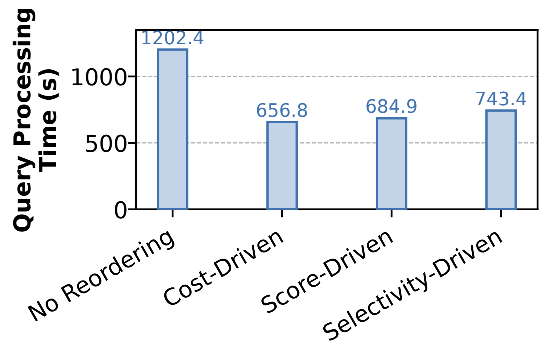


Figure 5.6: **Query processing time for Q1 w/o LAMINAR** – comparison of four EDDY routing options: no reordering, cost-driven, score-driven, and selectivity-driven.

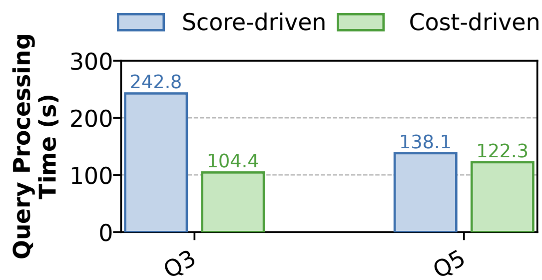


Figure 5.7: **Query processing time for Q3 and Q5 with LAMINAR enabled** – comparison between score-driven and cost-driven EDDY policies under parallelization.

5.6.3 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 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. 5.6. COST-DRIVEN, SCORE-DRIVEN, and SELECTIVITY-DRIVEN routing offer $1.83\times$, $1.76\times$, and

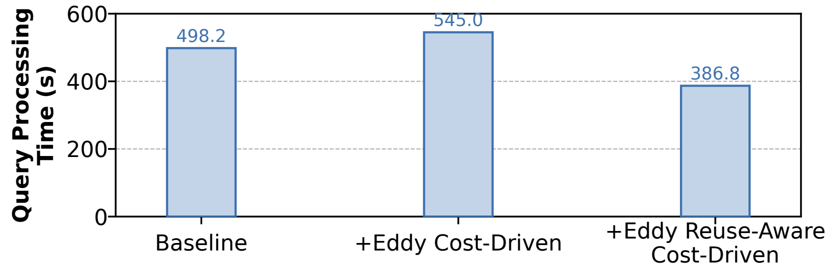
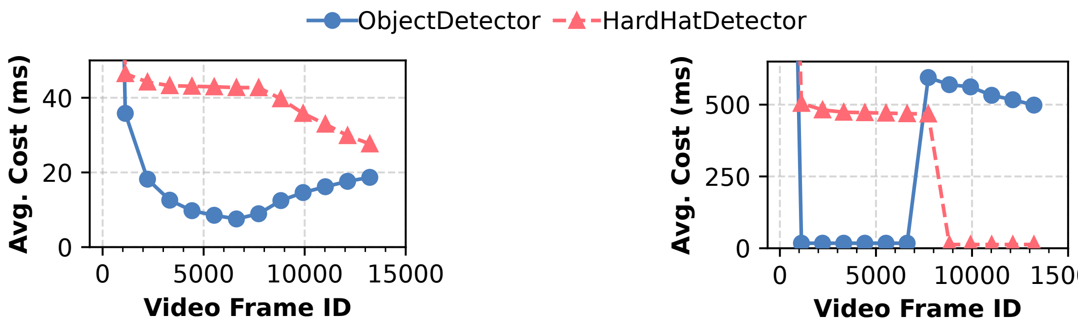


Figure 5.8: **Adaptive routing strategy impact on Q4** – comparison among baseline (no routing), cost-driven, and REUSE-AWARE cost-driven routing.

1.62× speedup, respectively. Profiling data (table 5.1) 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 sub-optimal 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. 5.7, the COST-DRIVEN policy outperforms the SCORE-DRIVEN policy by 2.3× for Q3 and marginally by 1.12× for Q5. Profiling data (table 5.1) 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 § 5.3.2, adaptive routing is crucial, especially in the presence of existing optimizations such as reusing results of expensive UDFs. To showcase



(a) Cost-driven.

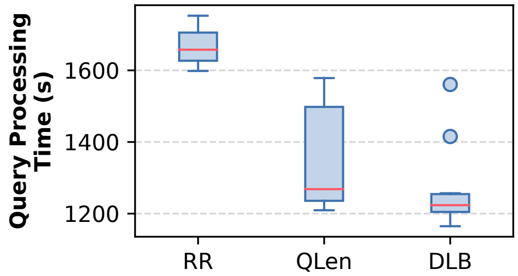
(b) REUSE-AWARE cost-driven.

Figure 5.9: **Average cost of predicates –** the average cost of different predicates over video frame ID with different routing policies.

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. 5.8 illustrates the query processing time for the three scenarios. First, +EDDY REUSE-AWARE cost-driven routing achieves a speedup of $1.29\times$ 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. 5.8, we examine the estimated costs of predicates across the two routing policies over the video (Fig. 5.9). During routing, predicates with lower estimated costs are prioritized. As depicted in Fig. 5.9a, the estimated cost of `ObjectDetector` significantly decreases from frame `id > 1000` to `id < 7000` due to caching, making `ObjectDetector` the 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. 5.9b. 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`



(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.

Figure 5.10: **Data-aware load balancing in LAMINAR**

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. 5.8. 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.

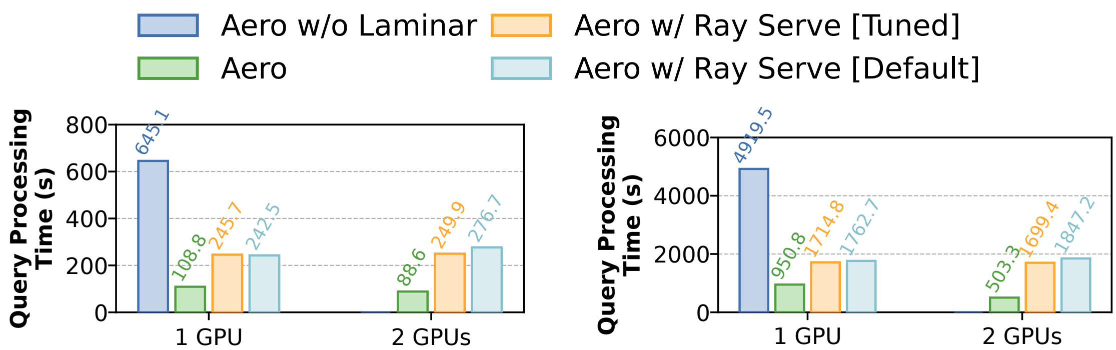
Data-aware Load Balancing. To demonstrate the performance benefits of data-aware load balancing, we focus on Q6. We compare the runtime of AERO on Q6 using three different LAMINAR routing policies: (i) round-robin policy (RR), (ii) queue length-driven load balancing (QLEN) adapted from popular policies [122, 123], (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. 5.10a, 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. 5.10b. 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.

5.6.4 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



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

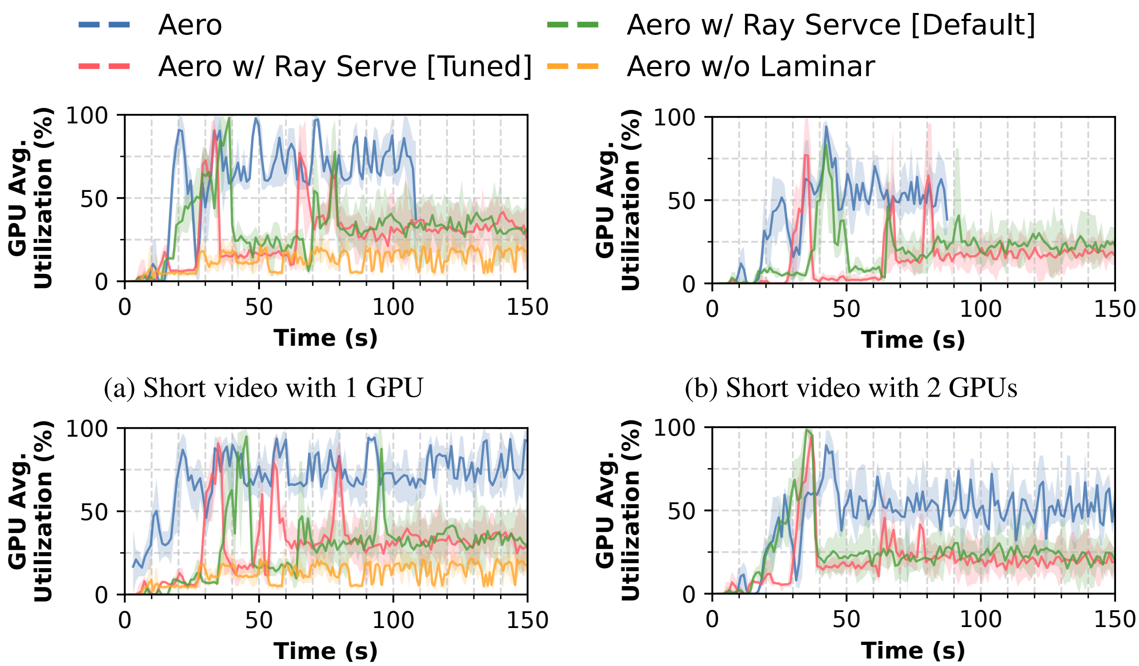
Figure 5.11: **Query processing time of Q3** – a comparison with different auto-scaling backends.

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. 5.11). 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 $1.3\times$ and $1.9\times$ 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. 5.12). 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



(a) Short video with 1 GPU

(b) Short video with 2 GPUs

(c) Long video with 1 GPU

(d) Long video with 2 GPUs

Figure 5.12: **GPU utilization** – average GPU utilization of Q3 across different auto-scaling backends on short and long videos.

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 to a sudden increase in queued requests. For example, in Fig. 5.12a, 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

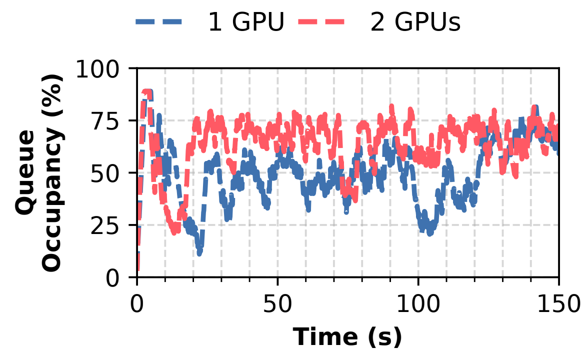


Figure 5.13: **Bottleneck study of AERO** – the average occupancy of CENTRAL QUEUE for running Q3 with the long video on 1 and 2 GPUs.

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. 5.13). We attribute the cause to a more congested CENTRAL QUEUE when scaling out, as shown in Fig. 5.13 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.

5.6.5 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. 5.14 illustrates the comparative performance of these setups. For Q3, integrating EDDY improves performance by $1.36\times$ over the baseline EVADB, showcasing its dynamic predicate reordering

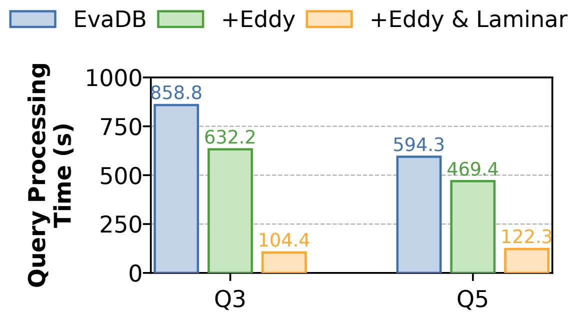


Figure 5.14: **Performance breakdown** – comparative efficiency of AERO variants (EVADB, EVADB +EDDY, EVADB +EDDY & LAMINAR) on Q3 and Q5.

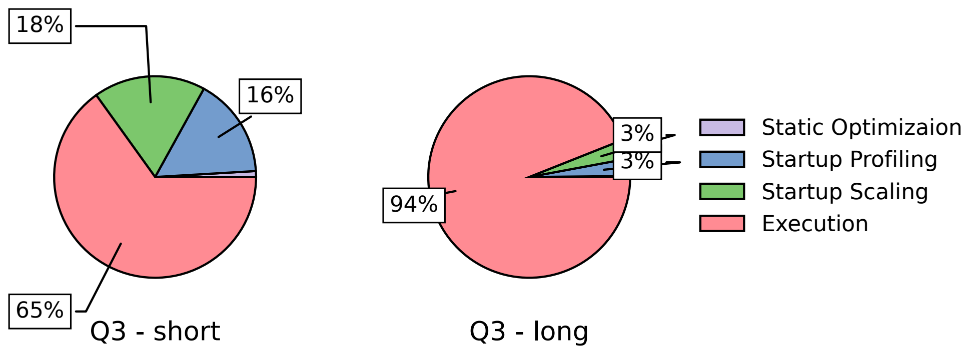


Figure 5.15: **Query processing time breakdown** – time breakdown for static optimization, startup profiling, startup scaling, and execution phases.

capabilities. Adding LAMINAR to EVADB +EDDY further accelerates performance by $6\times$, owing to the effectiveness of the LAMINAR component in optimizing GPU utilization. Similarly, in Q5, EDDY enhances performance by $1.27\times$ compared to EVADB, adapting well to skewed predicate distributions. Extending to EVADB +EDDY &LAMINAR enhances performance by $3.8\times$ over EVADB +EDDY, emphasizing LAMINAR's role in maximizing GPU resources.

5.6.6 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. 5.15, we notice that time spent on static optimization is almost negligible. For the short video, our previous result (Fig. 5.11a) demonstrates that AERO does not offer linear scalability when adding more resources (only $1.2\times$ 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. 5.15, they only account for 6% of query processing time, being amortized out. Thus, for a longer video (Fig. 5.11b), AERO offers better scalability when adding more GPU resources.

5.7 Limitations And Discussions

In this section, we discuss limitations and potential mitigation.

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 sub-optimal 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. 5.15), 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) [132], or multi-instance GPU (MIG) [133]. 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.

5.8 Related Work

Visual DBMSs. In VDBMSs, running deep learning models on every video frame is computationally expensive. To address this, researchers have proposed techniques [1, 4, 3, 134, 135, 136, 137, 138] using lightweight, specialized models to reduce the frames processed by the resource-intensive oracle model. Xu *et al.* [107] uses materialized views to store the results of expensive UDFs, thereby accelerating queries in exploratory video analytics settings where queries have overlapping computation. VIVA [109] 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 [139] supports distinct object queries and introduces an adaptive sampling algorithm to select frames from video segments likely to contain the object of interest. FiGO [108] 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 [20] selects the most optimal algorithm through sliding window-based profiling. Focus [140], Seiden [141], and TASTI [142] explore classical techniques like indexing to speed up video analytics queries. Other works focus on specific tasks, like object tracking [19], out-of-domain vocabulary classification [18], and action detection [143]. These works trade off accuracy for efficiency, while AERO prioritizes system-level optimizations without compromising query accuracy.

Equi-Vocal [144] presents a new interface through which users can find events in the video corpora by providing positive and negative examples. Skyscraper [145] 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 [146, 147, 148]. 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 [116] 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 [117] 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 [149] monitors runtime traffic and conducts dynamic scaling during execution to save model serving costs.

Clockwork [150] attempts to build a system with predictable latency to reduce tail latency. Scrooge [120] focuses on optimizing the cost of the deep learning inference by allocating just enough resources for inferencing without violating latency constraints. Tensorflow serving [151] and TensorRT inference server [152] are production-grade model serving systems. Nexus [118] proposes squishy bin packing to improve the utilization of GPUs and accurately avoid service-level objective violation. Cocktail [153] 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.* [154] study existing GPU spatial and temporal sharing mechanisms. To maximize spatial sharing, G-Net [155] proposes offloading network functions to GPU and allows multiple functions to share the GPU resource. Gandiva [114] 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 [156] 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 [157] focuses on designing a finish-time fair GPU resource-sharing mechanism. GSLICE [115] proposes GPU resource auto-provisioning along with NVIDIA MPS technology [158] to maximize the processing throughput. Choi *et al.* [159] 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.

5.9 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 $6.4\times$.

CHAPTER 6

BUFFER MANAGEMENT FOR OUT-OF-GPU LLM EXECUTION

6.1 Introduction

As large language models (LLMs) continue to advance, their parameter sizes have grown rapidly, exceeding the memory capacity of modern GPUs. For example, the state-of-the-art H100 GPU [160] is equipped with $80\,\text{GB}$ of high-bandwidth memory (HBM), whereas the latest breakthrough model, DeepSeek-R1 [161], contains 671B parameters—requiring about $1.22\,\text{TB}$ of memory when stored in `float16` format. While enterprises can leverage thousands of GPUs for distributed inference, non-commercial users typically have limited access to multiple GPUs. This situation creates a growing need to explore efficient execution strategies for LLMs that cannot fit entirely in a single GPU.

Existing approaches such as Orca [162] and PagedAttention [163] target inference optimizations only when the entire model fits on a GPU, and thus do not support models exceeding GPU memory capacity. FlexGen [164] and frameworks like DeepSpeed [165] can offload parameters to the CPU to accommodate larger models, but this offloading strategy often underutilizes GPU memory. As shown in Fig. 6.1, the offloading strategy uses CPU memory as the primary storage space, so all parameter accesses look up CPU first, which has high latency and low bandwidth.

In this work, we adopt a different perspective: we treat LLM parameters as data, GPU memory as a fast cache, and CPU memory as slower storage. Our goal is to build an efficient buffer manager that maximizes GPU memory usage. Different from the offloading strategy, the buffering strategy treats the GPU memory as the main storage, but the CPU memory as the alternative storage for parameters that cannot fit into the GPU memory. During the inference, the system always looks up the GPU buffer first. If the required

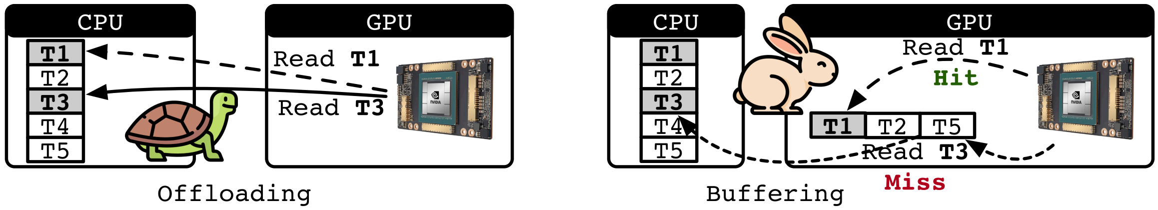


Figure 6.1: **Comparison between offloading and buffering** – buffering allows faster access to data cached in the buffer.

model parameters are already present in the GPU buffer (*i.e.,* "hit"), its value can be read from the buffer with low latency and high bandwidth (*e.g.,* `T1` in Fig. 6.1). When a "miss" occurs, the system looks up the CPU memory. With a good "hit" rate, the buffering strategy offers a better performance for out-of-GPU LLM inference.

For the buffering strategy, a good buffer replacement policy is critical to offer good performance, which could minimize the number of slow lookups to the CPU memory. Thus in this work, we analyze the model parameter access patterns of diverse LLM architectures. Based on the analysis, we propose a buffer management policy called LIRS-M that increases data hit rates in GPU memory, ultimately improving performance for out-of-GPU LLM execution. Our results demonstrate that buffering paired with LIRS-M offers $2.0\times$ speedup over the offloading strategy. LIRS-M policy itself offers a better buffer hit rate compared to other state-of-the-art buffer management policies.

Our main contributions are as follows:

- We present a proof-of-concept buffer manager that accelerates out-of-GPU LLM execution. Compared to the offloading strategy, our buffering strategy offers $2.0\times$ speedup.
- We design a new buffer management policy called LIRS-M that generalizes to state-of-the-art LLMs with diverse architectures. It demonstrates a better buffer hit rate than other state-of-the-art buffer management policies.

6.2 Background of LLMs Architecture and Out-of-GPU LLMs Execution

In this section, we review the background of LLMs and GPUs.

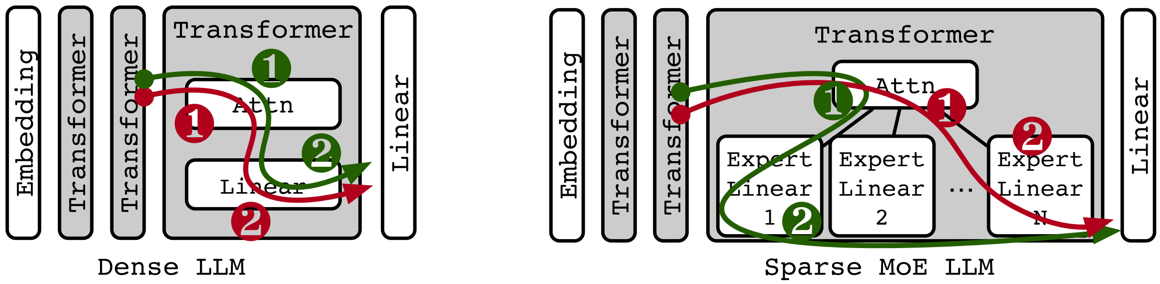


Figure 6.2: **Internal architecture and execution flow of LLMs** – including dense and MoE LLMs.

6.2.1 LLM Architecture

Internal of* LLMs*. The attention mechanism [166] has been widely adopted across various models—and LLMs are no exception. In an LLM, the attention kernel is typically encapsulated in a transformer layer, and the entire LLM is essentially a stack of multiple transformer layers with embedding and linear layers at the beginning and end (see Fig. 6.2). Each transformer layer includes model parameters (i.e., weights) used to transform its inputs (i.e., queries, keys, and values). Additionally, there is a key-value (*kv*) cache that stores the featurized outputs of previous keys and values required for inference. When the parameters from all transformer layers are combined, an LLM can become extremely large in terms of parameter size.

The description above outlines the architecture of dense LLMs. However, a more sparse architecture—known as the mixture of experts (MoE)—has recently gained popularity (e.g., Mixtral 8× 7B [167] and DeepSeek-R1 [161]). The key difference in an MoE is that each transformer layer contains multiple "expert" layers (often implemented as linear layers). During model inference, the specific expert layers used are chosen dynamically based on the input data, whereas in dense LLMs, all layers are always used (red and green execution path shown in Fig. 6.2).

LLM *Inference.* The execution of an LLM proceeds in a layer-by-layer fashion. Once

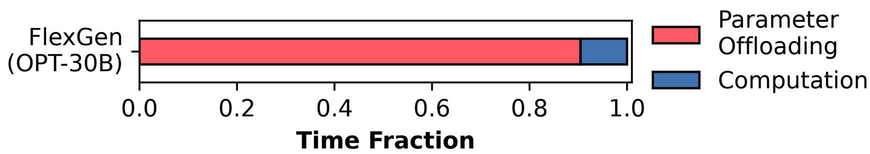


Figure 6.3: **Time breakdown of out-of-GPU execution** – model parameters offloading *vs.* computation for running OPT-30B on a single A40 GPU.

the final layer completes execution, the model produces a new token. In dense LLM, all parameters are consistently used, but in an MoE LLM, only a subset of expert layers is used during inference, chosen dynamically based on the input data.

LLM inference has prefilling and generation phases. During the prefilling phase, the input to the LLM is pre-existing tokens (*e.g.,* questions). Whereas, during generation, the generated new tokens are combined with pre-existing tokens and are fed into the LLM. For dense architecture LLMs, the inference phase does not change how model parameters are used. Nevertheless, for sparse MoE LLMs, how model parameters are accessed is different between the two phases. The prefilling phase results more expert layers being used, because input tokens are computed altogether. Fewer expert layers are used during the generation phase because the computation only runs on the new single token (computation of pre-existing tokens has already been done during the prefilling phase).

6.2.2 GPU Memory and Interconnect

State-of-the-art server-grade GPUs are equipped with dedicated memory that provides high bandwidth for GPU kernels to access data. Despite offering excellent bandwidth, GPU memory is limited in capacity. Consequently, data exceeding GPU memory capacity resides in CPU memory and must be transferred to the GPU on demand (*i.e.,* offloading). Because the bandwidth between CPU and GPU is significantly lower, these data transfers incur overheads.

To illustrate this overhead, we profiled the execution of the OPT-30B [168] model on a single A40 GPU using the FlexGen offloading framework. The A40 GPU has 48 GB of

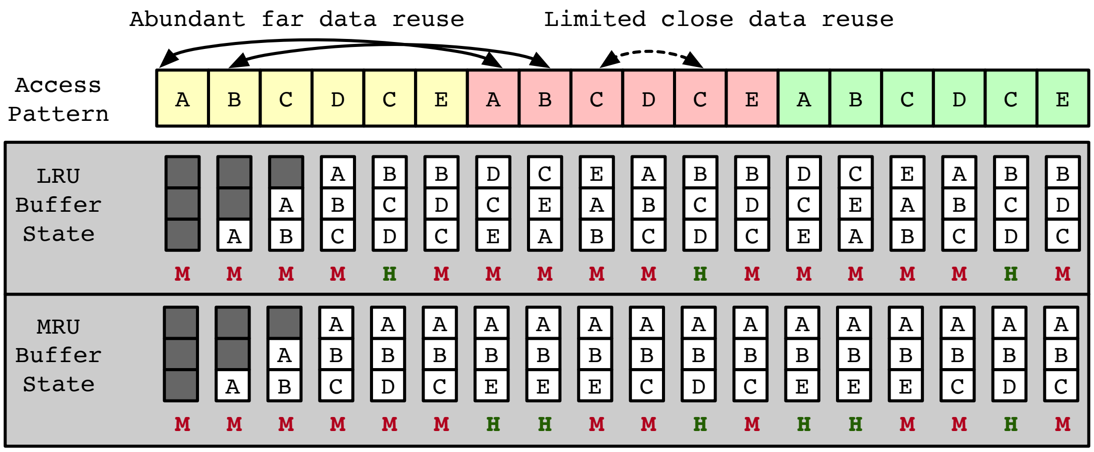


Figure 6.4: **Buffer hit under different buffer management policy** – the example demonstrates a case in which the working set size of access is much larger than the buffer size and presents limited temporal reuse opportunities.

memory with a bandwidth of approximately 700 GB/s, connected to CPU memory via a PCIe 4 interface that offers only a bandwidth around 26 GB/s. During the LLM execution, FlexGen stores all model parameters in CPU memory and transfers them to the GPU as needed. As shown in Fig. 6.3, more than 90% of the execution time is spent on offloading parameters, whereas the actual computation accounts for only 10%. These results highlight the severe CPU–GPU data transfer bottleneck and underscore the need to reduce data movement between.

6.3 Buffer Management Overview

During LLM execution, computation proceeds layer by layer, and only the model parameters required for the current layer need to reside in the GPU buffer. Parameters that are not present in the GPU buffer need to be fetched from CPU memory. To deliver good inference performance, the system must minimize slow parameter fetch from the CPU memory (*i.e.,* maximizing buffer hit).

6.3.1 Thrashing-Resistant Policies

Least-recently-used (LRU) replacement is a widely used buffer replacement policy. It prioritizes keeping the most recently accessed data in the buffer while evicting data that has not been accessed for the longest time. Although LRU excels when workloads exhibit strong temporal locality, it can underperform when the working set size greatly exceeds the available buffer capacity. In such cases, the long reuse distance causes data in the buffer to become stale rapidly, leading to buffer thrashing.

Fig. 6.4 shows a data access pattern that closely resembles the characteristics of LLM execution. Each color corresponds to generating a different token, and within a single token's generation, data **A** through **E** is accessed in near-sequential order. Although these data items exhibit strong reuse opportunities across token generations, their reuse distances are relatively high.

Now, assume the buffer can hold only three slots. Under the least-recently-used (LRU) policy, data that was accessed too far in the past is evicted first, retaining only the most recently accessed data. This strategy fails to capitalize on the longer-term reuse of data across different tokens. By contrast, the most-recently-used (MRU) policy prioritizes data that was accessed less recently, anticipating that it will be needed again sooner than data accessed more recently. Hence, it yields a higher buffer hit rate in this example.

KV Cache Recomputation. In many inference systems [163, 165, 164, 162], *kv* cache is enabled by default. However, we find that in out-of-GPU execution scenarios, if the *kv* cache resides in CPU memory, accessing it can be significantly slower than recomputing it. Furthermore, the *kv* cache involves both read and write operations: whenever a new token is generated, the updated *kv* cache must be flushed to CPU memory, incurring additional overheads. Consequently, recomputing the *kv* cache can be more efficient while also freeing GPU memory for other model parameters.

6.3.2 Inconsistent Parameter Access Pattern

The MRU policy shows significant performance gains over LRU in dense LLM architectures. At the same time, we also aim to enhance buffer management for sparse MoE LLMs. As discussed in § 6.2.1, the challenge arises from the fact that some model parameters (*e.g.,* attention layers) are consistently accessed across iterations, while others (e.g., expert layers) may not be accessed every iteration.

To understand its behavior, we profiled the Qwen MoE model [169] on a large-scale question-answer dataset, examining the activation frequency of each expert. Although the frequencies are not perfectly uniform, they are still quite evenly distributed with a slight skew. This suggests that virtually any expert layer can be activated during inference, making it difficult for buffer management alone to achieve substantially better performance than a baseline with no buffer management.

Instead of focusing on highly dynamic parameters, our approach prioritizes buffering parameters that exhibit consistent access patterns. In such a case, identifying these consistently accessed parameters is the core challenge. Prior work on the LIRS replacement policy [170] addressed a similar problem and outperformed LRU by tracking the historical reuse distance of each buffer entry and evicting entries with higher reuse distances. In our setting, reuse distance helps distinguish between parameters with consistent access and those with dynamic access. Parameters accessed in every iteration have reuse distances close to the total number of parameters in the model—a value that remains relatively constant. In contrast, parameters with dynamic access patterns show reuse distances that vary across iterations and are generally greater than the total parameter count. By leveraging this observation, we can prioritize buffering parameters that appear in every iteration (*i.e.,* lower reuse distance) and thereby improve the buffer hit rate.

6.3.3 LIRS-M: MRU-based Low Inter-reference Recency Set Replacement Policy

Inspired by this, we introduce LIRS-M, a new buffer management policy that integrates concepts from both LIRS and MRU. Like LIRS [170], LIRS-M maintains the reuse distance for every buffer entry. To calculate the virtual reuse distance, the buffer manager has a global virtual time counter vt, whose value is incremented every time the buffer is accessed. Entries that are inserted in the buffer for the first time have an infinite reuse distance. The buffer manager maintains metadata where the vt of every entry's last access is recorded. Once the entry is read again from the buffer, its reuse distance is updated to $vt_{entry_i,\ current} - vt_{entry_i,\ last\ access}$. Even after an entry is evicted from the buffer, we still keep its metadata for later use. The key difference between LIRS and LIRS-M lies in how eviction candidates are chosen: LIRS-M considers entries in most-recently-used order and selects the one with the greatest reuse distance, whereas LIRS considers entries in least-recently-used order. This design choice makes LIRS-M more resilient to thrashing than LIRS. We explain the algorithm details in § C.

6.4 Implementation

We implement the buffer manager in the PyTorch framework. For simplicity, the buffer manager operates at tensor granularity, where partial tensor placement in the GPU buffer is not supported. In § D, we report the tensor size distribution of studied LLMs.

Custom FxGraph Interpreter. PyTorch presumes the entire model resides in GPU memory at the start of execution and relies on users to explicitly manage parameter placement. To support out-of-GPU execution with automatic buffering, we use the `torch.compile` API to generate a static graph of the PyTorch program. We then annotate this static graph so that all parameter accesses go through our custom buffer manager. Specifically, within the generated static graph, model parameters are typically accessed via `fetch_attr` and `call_module` nodes; we monkey patch these nodes so that they retrieve parameters from

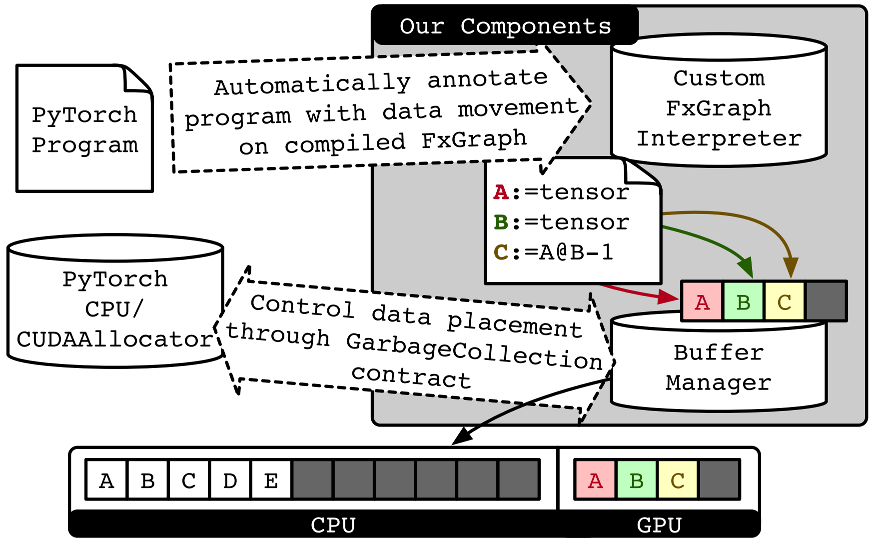


Figure 6.5: **Implementation of buffer manager using PyTorch framework** – illustrates how we control the interaction between the PyTorch program with our buffer manager and how the buffer manager controls the data movement in the hardware.

the buffer manager directly. For any other nodes that take model parameters as arguments, we override their behavior to reference those arguments from our buffer manager.

Buffer Manager and Garbage Collection. The PyTorch program follows the reference counting-based garbage collection protocol as the Python program. Users cannot directly control which tensors to evict from GPU memory; the Python garbage collector handles this based on reference counts, and users cannot freely manipulate those counts. To give the buffer manager explicit control over tensor placement, we modify the static graph to access tensors via `WeakRef`. Only the buffer manager holds actual references to tensors. As a result, the buffer manager's decisions translate directly into the physical placement of tensors in GPU memory, unaffected by references in the program graph.

6.5 Evaluation

In this section, we show our preliminary results of the benefit of using the buffer manager for out-of-GPU LLM execution.

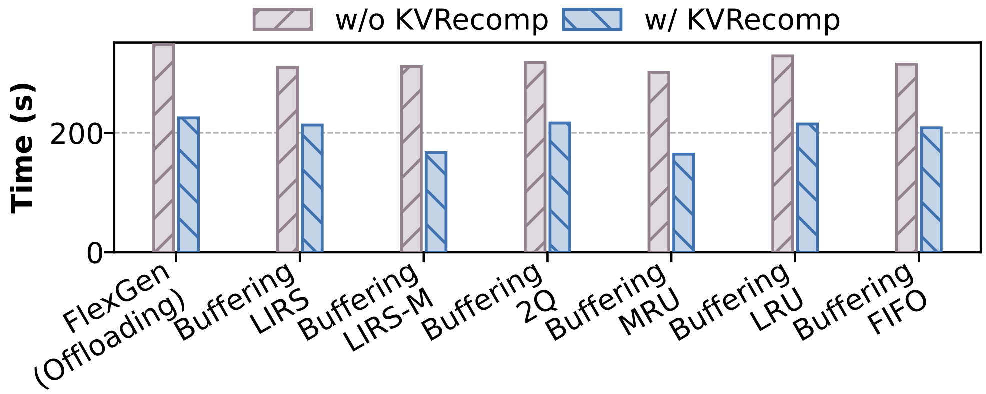


Figure 6.6: **Llama-30B inference time** – demonstrate the inference time of Llama-30B under different execution settings, including FlexGen, LIRS, LIRS-M, 2Q, MRU, LRU, and FIFO with the option of KVRecomp .

6.5.1 Buffering *vs.* Offloading

To demonstrate that using our buffer manager offers a performance advantage over straightforward model parameter offloading, we evaluate it on the Llama-30B model [171] running on an A40 GPU with 48 GB of memory. We feed random inputs to the model and measure the time required to generate 20 tokens. To isolate the impact of each technique, we create seven variants.

First, we implement FlexGen [164] for the Llama-30B model [171], relying on offloading and running it with a PyTorch-compiled graph. Next, we develop six additional variants that combine the compiled graph with buffering under different buffer management policies to gauge their relative benefits. Finally, we compare runs with and without *kv* cache recomputation to highlight the direct performance gains of recomputing the *kv* cache.

Fig. 6.6 shows that using our buffer manager with either the LIRS-M or MRU policy yields better performance compared to the baseline offloading mechanism (FlexGen). We also observe that when the LLM does not fit entirely in GPU memory, recomputing the *kv* cache is faster than accessing it from CPU memory—reducing FlexGen's runtime from 347.9 to 225.1 seconds. Moreover, combining the buffer manager (LIRS-M or MRU) with *kv* cache recomputation achieves the highest performance (i.e., 168.2 and 166.0 seconds,

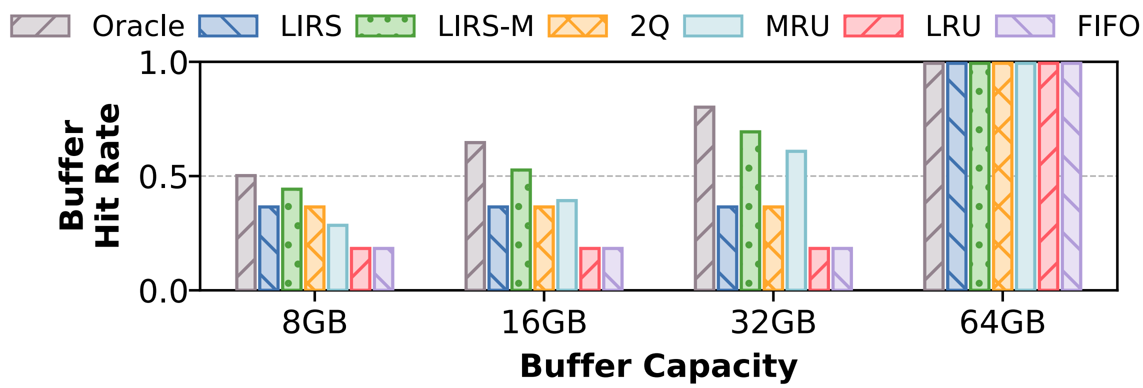


Figure 6.7: **Buffer hit rate of different buffer management policies over different buffer capacity** – buffer hit rate of Oracle, LIRS, LIRS-M, MRU, LRU, and FIFO over buffer capacities from 8 GB to 64 GB.

respectively). In the absence of *kv* cache recomputation, all buffer management policies still outperform offloading alone, although no single policy stands out as the clear winner.

Additionally, we analyzed the buffer hit rate by directly recording the model parameter access pattern and varying buffer capacities from 8 GB to 64 GB (Fig. 6.7). The model size for this experiment is 60 GB. When the buffer cannot accommodate the entire model, LIRS-M consistently achieves the highest hit rate (close to the oracle), with MRU performing nearly as well once the buffer size approaches the full model size. By contrast, the other policies exhibit noticeably lower hit rates. More interestingly, for this particular access pattern, other buffer management policies do not show any improving hit rate even when the buffer capacity increases. The buffer hit rate only improves when the buffer capacity surpasses the model size.

6.5.2 Impact of Buffer Management Policy

In this section, we explore how different buffer management policies perform when the model parameter access pattern is dynamic. We use the Qwen1.5-MoE-2.7B model [169], which has data-dependent expert usage, and run it on a randomly sampled question from the Google Natural Questions dataset [172]. We record the model parameter access pattern and study the buffer manager hit rate.[1]

Fig. 6.8 shows that LIRS-M consistently achieves the highest—or near-highest—hit rate across various buffer capacities. Recall that the Qwen1.5-MoE-2.7B model has 32.2 GB of total parameters, but only 5.4 GB of parameters are activated each inference iteration. Within that 5.4 GB subset, some parameters are used consistently across iterations, while others are not.

When the buffer capacity is below the activated size (e.g., 2 or 4 GB), LIRS-M significantly outperforms other policies. Though a gap still exists between the oracle and LIRS-M, we note the oracle does much better buffering for expert layers. This demonstrates that the reuse-distance-based approach allows it to focus on parameters consistently accessed each iteration, but prioritizes evicting parameters that are not consistently accessed over iterations. By contrast, 2Q, MRU, LRU, and FIFO have lower buffer hit rates. 2Q, for instance, can mistake infrequently accessed parameters for hot ones during the prefiling phase. This is because a great number of expert layers are activated and it is more likely that the same layer is activated repeatly during prefilling, resulting 2Q to misclassify them as hot entries. After the prefilling phase, most of the model parameters are only used once per iteration (*i.e.,* streaming), so it is very unlikely to get a hit in the FIFO queue. Consequently, very few parameters can be promoted from FIFO queue to LRU queue and the misclassified ”hot” entries stay in LRU queue, but does not help the buffer hit rate at all.

Once the buffer capacity exceeds the 5.4 GB activated set, most policies except MRU converge to a similar high hit rate. Although MRU is thrashing-resistant, it erroneously retains expert layers with very high reuse distance. While MRU excelled in the less dynamic scenario (Fig. 6.7), it underperforms when access patterns become more variable. Finally, as the buffer size approaches the entire model size, all policies achieve a hit rate of 1.

[1]Running out-of-GPU execution for this model on the entire dataset would take prohibitively long, so we only study the buffer hit rate.

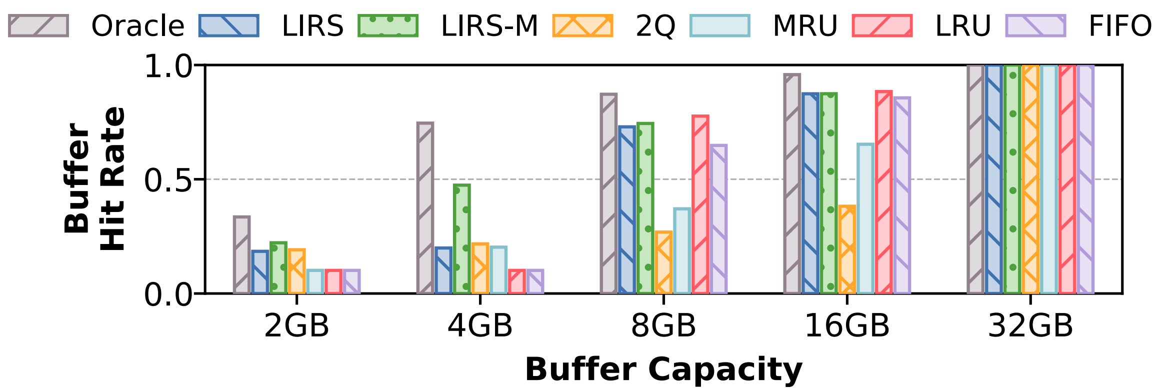


Figure 6.8: **Buffer hit rate of different buffer management policies over different buffer capacity** – buffer hit rate of Oracle, LIRS, LIRS-M, MRU, LRU, and FIFO over buffer capacities from 2 GB to 32 GB.

6.6 Conclusion

In this work, we explored the advantages of buffering over offloading for out-of-GPU LLM execution. Our preliminary findings show that buffering with the LIRS-M policy delivers a notable performance speedup compared to offloading, even across diverse model architectures (dense and sparse MoE). The LIRS-M policy proves effective in handling a range of parameter access patterns.

Moving forward, we plan to evaluate our approach on real workloads and hardware. Thus far, we have focused on single-GPU inference; however, in future work, we will extend our study to multi-GPU environments. Despite having more total memory across multiple GPUs, large models may still exceed available capacity. We have also conducted our experiments using single-batch inference; multi-batch settings—particularly for MoE models—could generate more varied parameter access patterns, making them an essential avenue for future investigation.

CHAPTER 7
CONCLUSION

To conclude, we establish a cohesive framework for enhancing query performance in modern databases by aligning advanced software optimization with careful hardware adaptation. At its core, this work recognizes that traditional static optimization—and simplistic reliance on monolithic ML models—cannot adequately handle the complexity and scale of today's workloads. These demands include video analytics, large-scale machine learning (ML), and, increasingly, large language model (LLM) inference.

First, the proposed fine-grained query optimization strategy shows that a single coarse-grained pipeline fails to deliver good performance on the variable difficulty within a dataset (*e.g.*,, video segments with differing content). By splitting data into chunks and picking just-enough accurate models for each segment, the system avoids abusing heavyweight algorithms and achieves significant performance gains without sacrificing correctness. This chunk-based approach transforms video analytics from a one-size-fits-all pipeline into a targeted strategy, leveraging an ensemble of off-the-shelf models and pruning away unhelpful ones.

Second, the in-depth GPU database characterization reveals that current GPU-accelerated systems leave substantial performance potential unrealized. Detailed metrics, including the roofline model analyses and warp stall breakdowns, identify memory-bound operations as the performance bottleneck. Insights from this analysis lead to our proposed optimizations—minimizing data movement, improving scheduling, and refining concurrency controls—that boost throughput and ensure more consistent utilization of GPU resources.

Building on these findings, the adaptive query processing (AQP) framework introduces dynamic execution plans that adapt to runtime statistics, which is especially critical for ML-based queries. By reordering predicates based on measured cost and selectivity, and by scaling GPU parallelism adaptively, the system maintains high hardware utilization. In a domain where deep-learning predicates can be orders of magnitude more expensive than standard relational operators, adaptive routing is key to keeping pipelines efficient.

Finally, the buffer management policy for out-of-GPU LLM execution extends the benefits of hardware-aware design to the large language models use cases. Instead of offloading model parameters in a naive fashion—resulting in thrashing and high data-transfer overhead—the proposed approach uses a novel eviction policy tailored to LLM access patterns. This yields significantly better caching behavior and narrows the performance gap for out-of-GPU LLM execution.

Taken as a whole, these contributions illustrate that software-hardware co-optimization is critical in achieving good performance for emerging workloads. This dissertation delivers a set of solutions by systematically studying and optimizing GPU-based systems—from the kernel level up to the query planner—and by smartly managing limited memory resources for large models.

Appendices

APPENDIX A

PROOF OF SAMPLE SIZE LOWER BOUND

We present a theoretical *sample size lower bound* of a chunk. To derive the bound, we first obtain the generalization error bound for a specific model m. We then obtain a probability bound for all the possible plans for a given accuracy threshold. By combining those two bounds, we compute the sample size lower bound.

Assume that the original video V is divided into R chunks.

$$V \rightarrow V_1 \, V_2 \ldots V_R$$

We assume that the OPTIMIZER needs at least K samples for each chunk, so that the OPTIMIZER is able to derive a query plan with small error. So, it picks a total of KR samples. We denote the collected samples of the i^{th} chunk by $\hat{V}_i$. We assume that the OPTIMIZER has a set of models in the ensemble (denoted by M). The OPTIMIZER evaluates model m on frame x to obtain an accuracy metric denoted by $f(x, m)$. $f(x, m)$ compares results with respect to the reference model. If they are in consensus, it is set to 1. Otherwise, it is set to 0.

For a given model m, $f(x, m)$ over sampled frames is likely to differ from that computed across the entire video. This error is given by:

$$\mathop{\mathbb{E}}_{x \in \hat{V}} f(x, m) - \mathop{\mathbb{E}}_{x \in V} f(x, m)$$

To facilitate the computation of sample size lower bound, the average variance is given by:

$$\sigma^2 = \mathop{\mathbb{E}}_{x \in \hat{V} m' \in M} \left[f(x, m') - \mathop{\mathbb{E}}_{m'' \in M} f(x, m'') \right]^2$$

It measures the average variance between the accuracy of a model and the mean accuracy of all models in the ensemble. Given an error threshold t, we next obtain the probability that error is within t using Bernstein's inequality [173]:

$$\mathbb{P}\Big[\mathop{\mathbb{E}}_{x\in\hat{V}} f(x,m) - \mathop{\mathbb{E}}_{x\in V} f(x,m) > t\Big] \leq exp\Big(\frac{KRt^2}{2\sigma^2 + 2t/3}\Big)$$

Next, we seek to bound the probability that a particular plan returned by the OPTIMIZER has better accuracy than the accuracy threshold $\mathbb{A}$. Let us first define μ to be the mean accuracy of all models from the model ensemble over the all sampled video frames.

$$\mu = \mathop{\mathbb{E}}_{x\in\hat{V},m'\in M} f(x,m')$$

The bound of a model m that has greater than $\mathbb{A}$ accuracy is:

$$\begin{aligned}
&\mathbb{P}\Big[\mathop{\mathbb{E}}_{x\in\hat{V}} f(x,m) > \mathbb{A}\Big] \\
&\quad = \mathbb{P}\Big[\mathop{\mathbb{E}}_{x\in\hat{V}} f(x,m) - \mathop{\mathbb{E}}_{x\in\hat{V},m'\in M} f(x,m') > \mathbb{A} - \mathop{\mathbb{E}}_{x\in\hat{V},m'\in M} f(x,m')\Big] \\
&\quad = \mathbb{P}\Big[\mathop{\mathbb{E}}_{x\in\hat{V}} f(x,m) - \mathop{\mathbb{E}}_{x\in\hat{V},m'\in M} f(x,m') > \mathbb{A} - \mu\Big]
\end{aligned}$$

We next use Bernstein's inequality again to bound the term on right-hand-side to obtain:

$$\mathbb{P}\Big[\mathop{\mathbb{E}}_{x\in\hat{V}} f(x,m) > \mathbb{A}\Big] \leq exp\Big(-\frac{KR(\mathbb{A}-\mu)^2}{2\sigma^2 + 2(\mathbb{A}-\mu)/3}\Big)$$

This represents the probability that a plan with a specific m has greater than $\mathbb{A}$ accuracy. Since there are $|M|$ models in the ensemble and R chunks in the video, the total number of possible plans is $|M|^R$. Using the previous bound, we obtain a bound on the number of execution plans that have accuracy greater than $\mathbb{A}$:

$$\Big|\Big\{m' \in M : \mathop{\mathbb{E}}_{x\in\hat{V}} f(x,m') > \mathbb{A}\Big\}\Big| \leq |M|^R exp\Big(-\frac{KR(\mathbb{A}-\mu)^2}{2\sigma^2 + 2(\mathbb{A}-\mu)/3}\Big)$$

So far, we have obtained the bound that the error of a plan is within t. We have also obtained the bound for the number of plans that have accuracy better than $\mathbb{A}$. By combining of these two bounds, we obtain a bound on the sample size K such that plans with accuracy better than $\mathbb{A}$ have an error within t. We also simplify the equation by Jensen's inequality [174]:

$$
\begin{aligned}
&\mathbb{P}\Big[\mathop{\mathbb{E}}_{x\in\hat{V}} f(x,m) - \mathop{\mathbb{E}}_{x\in V} f(x,m) > t\Big], \forall m \in M \\
&\qquad \leq |M|^R exp\Big(-\frac{KR(\mathbb{A}-\mu)^2}{2\sigma^2+2(\mathbb{A}-\mu)/3}\Big) exp\Big(-\frac{KRt^2}{2\sigma^2+2t/3}\Big) \\
&\qquad \leq |M|^R exp\Big(-\frac{KR(\mathbb{A}-\mu)^2}{2\sigma^2+2(\mathbb{A}-\mu)/3} - \frac{KRt^2}{2\sigma^2+2t/3}\Big) \\
&\qquad \leq |M|^R exp\Big(-\frac{KR(t+\mathbb{A}-\mu)^2}{4\sigma^2+2(t+\mathbb{A}-\mu)/3}\Big)
\end{aligned}
$$

We expect the value to reach 0 as R grows (*i.e.,* in the extreme case, generalization error should be 0 if query is optimized for every frame or chunk size is 1). To achieve that, the base should be less than 1 as expressed below:

$$
\begin{aligned}
1 &\geq |M| exp\Big(-\frac{K(t+\mathbb{A}-\mu)^2}{4\sigma^2+2(t+\mathbb{A}-\mu)/3}\Big) \\
\frac{K(t+\mathbb{A}-\mu)^2}{4\sigma^2+2(t+\mathbb{A}-\mu)/3} &\geq log(|M|) \\
\lambda \overset{?}{\geq} K &\geq \frac{log(|M|)(4\sigma^2+2(t+\mathbb{A}-\mu)/3)}{(t+\mathbb{A}-\mu)^2}
\end{aligned}
$$

Thus, given a desired error range t and expected accuracy $\mathbb{A}$, the OPTIMIZER calculates sample size lower bound K using the observed variance between models and the mean model accuracy.

APPENDIX B

FINE-GRAINED QUERY OPTIMIZATION ALGORITHM

We present the detailed query optimization algorithm here. As shown in Algo. 1, the OPTIMIZER operates on a chunk V_i defined by the start and end frames. It begins by treating the entire video V as a single chunk (line 2). Given a chunk c, the OPTIMIZER picks a λ number of samples from c (line 4). The OPTIMIZER then prunes the set of models using the estimator that is built using a variant of Thompson sampling (line 5). For the very first chunk, it does not prune any model. For later chunks, it prunes the ensemble into fewer number of models (*e.g.,* three).

Next, the OPTIMIZER computes reward by measuring consensus between a model and the reference model on sampled video frames. It then updates the reward estimates (line 11). Using the techniques presented in § 3.3, the OPTIMIZER estimates the required sample size bound (K) and the benefits of further splitting the chunk based on the current query plan. If the number of samples (λ) is greater than the lower bound K, and further splitting does not lower the query processing time, then the OPTIMIZER returns the current plan (line 16). The OPTIMIZER also stops searching for a better plan even when the chunk is too small (*i.e.,* less than 100 frames). Otherwise, the OPTIMIZER continues to recursively split the chunk and compute plans for the resulting sub-chunks (line 20).

Algorithm 1: Query optimization.

```
Input  : V – Video.
         M – Model ensemble (e.g., 8 models from EFFICIENTDET).
         λ – Sample size for each chunk (e.g., 10 frames).
         𝔸 – User-specified query accuracy (e.g., 0.95).
         t – Tolerable error bound (e.g., 0.03).
         E – Estimator used for model pruning.
Output: Return a list of execution plans.
return GetQueryPlan(0, Length(V))

Function GetQueryPlan(start, end)
    Output: A collection of plans under 𝔸 constraint within t.
    V_i ← V [start, end] // Obtain chunk.
    V̂ ← UniformSample(V_i, λ) // Sample frames.
    M̂ ← PruneModel(E, M)
    R ← {}
    // Profiling step.
    for v ∈ V̂ do
        for m ∈ M̂ do
            R += { [v, m]: Predict(v, m) ==
                        Predict(v, m_reference) }
    UpdateEstimator(E, R, M̂)
    K ← EstimateSampleSize(R, M̂)
    FurtherChunkingCost, ExecCost ← EstimateCost(R, M̂)
    // Determine whether to continue splitting.
    plan ← {}
    if (K ≤ λ and ExecCost < FurtherChunkingCost)
      or ( |end - start| ≤ 100 frames) then
        plan += { [start,end]: PickBestModel(R, M̂, 𝔸, t) }
    else
        plan += GetQueryPlan(start, end/2)
        plan += GetQueryPlan(end/2, end)
    return plan
```

APPENDIX C

BUFFER MANAGEMENT EVICTION ALGORITHM DETAILS

We explain the eviction algorithm in Algo. 2. As shown, when the computation accesses the model parameter, it is always directed to the `Read` function. Every access increments the global virtual timestamp (*i.e.,* `VT`) by `1`. If the parameter is already in the buffer, it is simply returned. If it is not in the buffer, the buffer manager starts to evict entries in it until there is enough space in the buffer to hold the incoming parameter. The buffer manager relies on `GetEvictionEntry` to find an eviction candidate, which locates the entry with the highest reuse distance in the MRU order. Once the buffer has enough space for the parameter, it is inserted in the buffer. The timestamp when the parameter was lastly used and also its reuse distance are updated accordingly.

Algorithm 2: Buffer access and replacement policies.

```
Input  : Buf – Buffer that stores parameters in GPU
         MRUList – Buffer entry ordered from the most recent to the least recent order
         LastUseVT – Dictionary metadata to track the virtual timestamp (VT) when the entry is being used last time
         ReuseDis – Dictionary metadata to track the historical reuse distance of each buffer entry
         VT – Global virtual timestamp

Function Read(Param, ParamSize)
    Output: Read a parameter from the buffer
    VT += 1
    while Param not ∈ Buf and AvailableSize(Buf) < ParamSize do
        CandEntry ← GetEvictionEntry(Buf)
        Evict(Buf, CandEntry)
    // Update buffer and eviction order.
    if Param not ∈ Buf then
        Insert(Buf, Param)
        ReuseDis [Param] ← VT − LastUseVT [Param]
        LastUseVT [Param] ← VT
    return Buf [Param]

Function GetEvictionEntry()
    Output: Return buffer entry to evict
    CandEntry ← MRUList [0]
    CandReuseDis ← ReuseDis [CandEntry]
    for Entry ∈ MRUList do
        // Traverse in MRU order.
        if ReuseDis [Entry] > CandReuseDis then
            CandEntry ← Entry
            CandReuseDis ← ReuseDis [Entry]
    return CandEntry
```

APPENDIX D
TENSOR SIZES DISTRIBUTION

Fig. D.1 shows the CDF of tensor sizes in both Llama-30B and Qwen-MoE models. We notice that most of the parameters are in the order of MB. While only very few tensors have less than MB size.

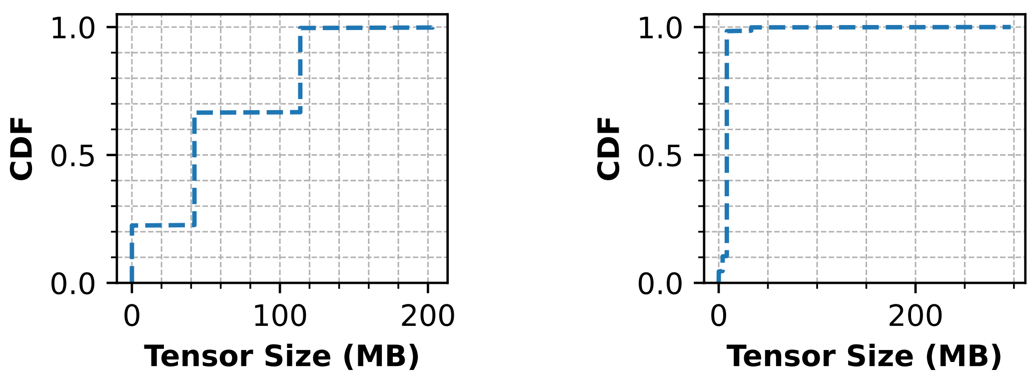


Figure D.1: **Tensor sizes distribution –** the CDF of the sizes of tensor for Llama-30B and Qwen-MoE models.

REFERENCES

[1] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing deep cnn-based queries over video streams at scale," in *VLDB*, 2017, pp. 1586–1597.

[2] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri, "Accelerating machine learning inference with probabilistic predicates," in *SIGMOD*, 2018, pp. 1493–1508.

[3] M. R. Anderson, M. Cafarella, G. Ros, and T. F. Wenisch, "Physical Representation-Based Predicate Optimization for a Visual Analytics Database," in *ICDE*, 2019, pp. 1466–1477.

[4] D. Kang, P. Bailis, and M. Zaharia, "Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics," in *VLDB*, 2019, pp. 533–546.

[5] A. Deshpande, Z. Ives, and V. Raman, "Adaptive Query Processing," *Foundations and Trends in Databases*, pp. 1–140, 2007.

[6] N. Kabra and D. J. Dewitt, "Mid-Query Re-Optimization of Sub-Optimal Execution Plans," *SIGMOD*, 1998.

[7] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," *SIGMOD*, p. 12, 2000.

[8] P. Bizarro, S. Babu, D. DeWitt, and J. Widom, "Content-Based Routing: Different Plans for Different Data," *PVLDB*, 2005.

[9] S. Babu, P. Bizarro, and D. DeWitt, "Proactive re-optimization," in *SIGMOD*, 2005, pp. 107–118.

[10] C. A. Waldspurger and W. E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," *OSDI*, 1994.

[11] M. Zaharia *et al.*, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, pp. 56–65, 2016.

[12] G. T. Kakkar *et al.*, "EVA: An End-to-End Exploratory Video Analytics System," in *DEEM*, 2023, pp. 1–5.

[13] *Using third-party packages*, 2024.

[14] NVIDIA, *Nvidia multi-instance gpu user guide*, [Online] Available from: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, 2022.

[15] NVIDIA, *Nvidia multi-process service introduction*, [Online] Available from: https://docs.nvidia.com/deploy/mps/index.html, 2021.

[16] NVIDIA, *NVIDIA Multi-Instance GPU*, [Online] Available from: https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2022.

[17] R. Armstrong, A. Sundaram, and F. Oh, *Revealing new features in the cuda 11.5 toolkit*, [Online] Available from: https://developer.nvidia.com/blog/revealing-new-features-in-the-cuda-11-5-toolkit/, 2021.

[18] Y. Zhang and A. Kumar, "Panorama: A data system for unbounded vocabulary querying over video," in *VLDB*, 2019, pp. 477–491.

[19] F. Bastani *et al.*, "MIRIS: Fast Object Track Queries in Video," in *SIGMOD*, 2020, pp. 1907–1921.

[20] J. Jiang, G. Ananthanarayanan, P. Bodík, S. Sen, and I. Stoica, "Chameleon: Scalable adaptation of video analytics," in *SIGCOMM*, 2018, pp. 253–266.

[21] M. Tan, R. Pang, and Q. V. Le, "EfficientDet: Scalable and Efficient Object Detection," in *CVPR*, 2020, pp. 10 778–10 787.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2015, pp. 770–778.

[23] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," in *NeurIPS*, 2015, pp. 91–99.

[24] W. Liu *et al.*, "SSD: Single shot MultiBox detector," in *ECCV*, 2016, pp. 21–37.

[25] W. R. Thompson, "On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika*, pp. 285–294, 1933.

[26] M. N. Katehakis and A. F. Veinott Jr, "The multi-armed bandit problem: Decomposition and computation," *Mathematics of Operations Research*, pp. 262–268, 1987.

[27] A. K. Gupta, K. G. Smith, and C. E. Shalley, "The interplay between exploration and exploitation," *Academy of management journal*, pp. 693–706, 2006.

[28] J. G. March, "Exploration and exploitation in organizational learning," *Organization science*, pp. 71–87, 1991.

[29] O. Chapelle and L. Li, "An Empirical Evaluation of Thompson Sampling," in *NeurIPS*, 2011.

[30] S. Agrawal and N. Goyal, "Analysis of thompson sampling for the multi-armed bandit problem," in *COLT*, 2012.

[31] T.-Y. Lin *et al.*, "Microsoft COCO: Common Objects in Context," in *ECCV*, 2014, pp. 740–755.

[32] L. Wen *et al.*, "UA-DETRAC: A new benchmark and protocol for multi-object detection and tracking," in *CVIU*, 2020.

[33] F. Yu *et al.*, "BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning," 2020.

[34] S. Oh *et al.*, "A large-scale benchmark dataset for event recognition in surveillance video," in *CVPR*, 2011, pp. 3153–3160.

[35] A. Paszke *et al.*, "Automatic differentiation in pytorch," in *NeuIPS, Autodiff Workshop*, 2017.

[36] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-YOLOv4: Scaling cross stage partial network," in *CVPR*, 2021, pp. 13 029–13 038.

[37] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO – DB2's LEarning Optimizer," in *VLDB*, 2001, pp. 19–28.

[38] R. Marcus *et al.*, "Neo: A learned query optimizer," in *VLDB*, 2019, pp. 1705–1718.

[39] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews, "The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database," *VLDB*, pp. 1401–1412, 2016.

[40] D. He *et al.*, "Query processing on tensor computation runtimes," *PVLDB*, pp. 2811–2825, 2022.

[41] BlazingSQL, *Blazingsql*, https://github.com/BlazingDB/blazingsql, 2021.

[42] HeavyDB, *Heavydb*, https://github.com/heavyai/heavydb, 2022.

[43] A. Shanbhag, S. Madden, and X. Yu, "A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics," in *SIGMOD*, 2020, pp. 1617–1632.

[44] K. Kohei, *Pg-strom*, https://github.com/heterodb/pg-strom, 2022.

[45] P. Bakkum and S. Chakradhar, *Efficient data management for gpu databases*, https://github.com/bakks/virginian/, 2010.

[46] J. Shen, Z. Wang, D. Wang, J. Shi, and S. Chen, *Aresdb*, https://github.com/uber/aresdb, 2019.

[47] A. Krolik, C. Verbrugge, and L. Hendren, "R3d3: Optimized Query Compilation on GPUs," in *CGO*, 2021, pp. 277–288.

[48] J. Paul, J. He, and B. He, "GPL: A GPU-based Pipelined Query Processing Engine," in *SIGMOD*, 2016, pp. 1935–1950.

[49] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner, "Pipelined Query Processing in Coprocessor Environments," in *SIGMOD*, 2018, pp. 1603–1618.

[50] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *PVLDB*, pp. 709–720, 2013.

[51] H. Wu *et al.*, "Red Fox: An Execution Environment for Relational Query Processing on GPUs," in *CGO*, 2014, pp. 44–54.

[52] S. Floratos *et al.*, "NestGPU: Nested Query Processing on GPU," in *ICDE*, 2021, pp. 1008–1019.

[53] *PCIe 4.0 specification finally out with 16 GT/s on tap*, [Online] Available from: https://techreport.com/news/32064/pcie-4-0-specification-finally-out-with-16-gts-on-tap/, 2017.

[54] *PCI-SIG achieves 32GT/s with new PCI Express 5.0 specification*, [Online] Available from: https://www.businesswire.com/news/home/20190529005766/en/PCI-SIG%C2%AE-Achieves-32GTs-with-New-PCI-Express%C2%AE-5.0-Specification, 2019.

[55] *PCI-SIG releases PCIe 6.0 specification delivering record performance to power Big Data applications*, [Online] Available from: https://www.businesswire.com/news/home/20220111005011/en/PCI-SIG%C2%AE-Releases-PCIe%C2%AE-6.0-Specification-Delivering-Record-Performance-to-Power-Big-Data-Applications, 2022.

[56] *PCI-SIG announces PCI Express 7.0 specification to reach 128 GT/s*, [Online] Available from: https://www.businesswire.com/news/home/20220621005137/en, 2022.

[57] E. Furst, M. Oskin, and B. Howe, "Profiling a GPU database implementation: A holistic view of GPU resource utilization on TPC-H queries," in *DaMON*, 2017, pp. 1–6.

[58] Y.-K. Suh, J. An, B. Tak, and G.-J. Na, "A Comprehensive Empirical Study of Query Performance Across GPU DBMSes," *SIGMETRICS*, pp. 1–29, 2022.

[59] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs On A Modern Processor: Where Does Time Go?" *PVLDB*, 1999.

[60] U. Sirin and A. Ailamaki, "Micro-architectural analysis of OLAP: Limitations and opportunities," *PVLDB*, pp. 840–853, 2020.

[61] R. Sen and K. Ramachandra, "Characterizing resource sensitivity of database workloads," in *HPCA*, 2018, pp. 657–669.

[62] R. Sen and Y. Tian, "Microarchitectural analysis of graph bi queries on rdbms," in *DaMoN*, 2023, pp. 102–106.

[63] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, pp. 65–76, 2009.

[64] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, "Applying the roofline model," in *ISPASS*, 2014, pp. 76–85.

[65] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *TPCTC*, 2009, pp. 237–252.

[66] A. Shanbhag, *Crystal gpu library*, https://github.com/anilshanbhag/crystal, 2020.

[67] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004, pp. 75–88.

[68] NVIDIA, *Parallel thread execution isa version 7.8*, [Online] Available from: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2022.

[69] NVIDIA, *Thrust*, [Online] Availble from: https://docs.nvidia.com/cuda/thrust/index.html, 2022.

[70] R. D. Team, *Rapids: Collection of libraries for end to end gpu data science*, 2018.

[71] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019, pp. 8024–8035.

[72] M. Armbrust *et al.*, "Spark SQL: Relational Data Processing in Spark," in *SIGMOD*, 2015, pp. 1383–1394.

[73] NVIDIA, *Nvidia a100 tensor core gpu unprecedented acceleration at every scale*, [Online] Available from: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf, 2020.

[74] NVIDIA, *Nvidia nsight systems user guide*, [Online] Available from: https://docs.nvidia.com/nsight-systems/UserGuide/index.html, 2022.

[75] NVIDIA, *Nvidia-smi documentation*, [Online] Available from: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, 2016.

[76] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware Roofline model: Upgrading the loft," *IEEE CAL*, pp. 21–24, 2014.

[77] Y. Asada *et al.*, "Share the tensor tea: How databases can leverage the machine learning ecosystem," *PVLDB*, pp. 3598–3601, 2022.

[78] A. Gandhi *et al.*, "The tensor data platform: Towards an ai-centric database system," in *CIDR*, 2022.

[79] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *PVLDB*, pp. 539–550, 2011.

[80] L. Mao, *Math-bound vs memory-bound operations*, https://leimao.github.io/blog/Math-Bound-VS-Memory-Bound-Operations/, 2021.

[81] Y. Yuan, R. Lee, and X. Zhang, "The Yin and Yang of processing data warehousing queries on GPU devices," *PVLDB*, pp. 817–828, 2013.

[82] T. Rabl, M. Poess, H.-A. Jacobsen, P. O'Neil, and E. O'Neil, "Variations of the star schema benchmark to test the effects of data skew on query performance," in *ICPE*, 2013, p. 361.

[83] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects," in *SIGMOD*, 2020, pp. 1633–1649.

[84] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects," in *SIGMOD*, 2022, pp. 1017–1032.

[85] H. Funke and J. Teubner, "Data-parallel query processing on non-uniform data," *PVLDB*, pp. 884–897, 2020.

[86] J. Paul, B. He, S. Lu, and C. T. Lau, “Improving execution efficiency of just-in-time compilation based query processing on GPUs,” *PVLDB*, pp. 202–214, 2020.

[87] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, “Hardware-Conscious Hash-Joins on GPUs,” in *ICDE*, 2019, pp. 698–709.

[88] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden, “Tile-based Lightweight Integer Compression in GPU,” in *SIGMOD*, 2022, pp. 1390–1403.

[89] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, “HippogriffDB: Balancing I/O and GPU bandwidth in big data analytics,” *PVLDB*, pp. 1647–1658, 2016.

[90] B. W. Yogatama, W. Gong, and X. Yu, “Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS,” *PVLDB*, pp. 2491–2503, 2022.

[91] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, “HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines,” *PVLDB*, pp. 544–556, 2019.

[92] V. Rosenfeld, S. Breß, and V. Markl, “Query Processing on Heterogeneous CPU/GPU Systems,” *ACM Computing Surveys*, pp. 1–38, 2023.

[93] N. Boeschen and C. Binnig, “GaccO - A GPU-accelerated OLTP DBMS,” in *SIGMOD*, 2022, pp. 1003–1016.

[94] J. Paul, S. Lu, B. He, and C. T. Lau, “MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures,” in *SIGMOD*, 2021, pp. 1413–1425.

[95] T. Maltenberger, I. Ilic, I. Tolovski, and T. Rabl, “Evaluating Multi-GPU Sorting with Modern Interconnects,” in *SIGMOD*, 2022, pp. 1795–1809.

[96] H. Doraiswamy and J. Freire, “A GPU-friendly Geometric Data Model and Algebra for Spatial Queries,” in *SIGMOD*, 2020, pp. 1875–1885.

[97] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *ISCA*, 2009.

[98] Y. Zhang and J. D. Owens, “A quantitative performance analysis model for GPU architectures,” in *HCPA*, 2011, pp. 382–393.

[99] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, “GPGPU performance and power estimation using machine learning,” in *HPCA*, 2015, pp. 564–576.

[100] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *PPoPP*, 2010, p. 10.

[101] M. Hill and V. Janapa Reddi, "Gables: A Roofline Model for Mobile SoCs," in *HPCA*, 2019, pp. 317–330.

[102] N. Ding and S. Williams, "An Instruction Roofline Model for GPUs," in *PBMS*, 2019, pp. 7–18.

[103] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, "Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling," in *ISPASS*, 2017, pp. 259–268.

[104] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, and X. Chen, *A Survey of Multi-Tenant Deep Learning Inference on GPU*, 2022. arXiv: 2203.09040 `[cs.DC]`.

[105] C. Tan *et al.*, *Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem*, 2021. arXiv: 2109.11067 `[cs.DC]`.

[106] A. F. Kaas, S. P. Paleykov, T. Robroek, and P. Tözün, *Deep Learning Training on Multi-Instance GPUs*, 2022. arXiv: 2209.06018 `[cs.LG]`.

[107] Z. Xu, G. T. Kakkar, J. Arulraj, and U. Ramachandran, "EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views," in *SIGMOD*, 2022, pp. 602–616.

[108] J. Cao, K. Sarkar, R. Hadidi, J. Arulraj, and H. Kim, "FiGO: Fine-Grained Query Optimization in Video Analytics," in *SIGMOD*, 2022, pp. 559–572.

[109] F. Romero, J. Hauswald, A. Partap, D. Kang, M. Zaharia, and C. Kozyrakis, "Optimizing Video Analytics with Declarative Model Relationships," *PVLDB*, pp. 447–460, 2022.

[110] S. Yu *et al.*, "Vqpy: An object-oriented approach to modern video analytics," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 279–295, 2024.

[111] J. M. Hellerstein, "Practical predicate placement," in *SIGMOD*, 1994, pp. 325–335.

[112] I. Adan and J. Resing, "Queueing theory," *Eindhoven University of Technology*, vol. 180, 2002.

[113] "Introduction to Queueing Theory," in *Introduction to Discrete Event Systems*, C. G. Cassandras and S. Lafortune, Eds., Springer US, 2008, pp. 429–497.

[114] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *OSDI*, 2018, pp. 595–610.

[115] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "GSLICE: Controlled spatial sharing of GPUs for a scalable inference platform," in *SoCC*, 2020, pp. 492–506.

[116] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency online prediction serving system," pp. 613–627, 2017.

[117] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," pp. 397–411, 2021.

[118] H. Shen *et al.*, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *SOSP*, 2019, pp. 322–337.

[119] Y. Wu, M. Lentz, D. Zhuo, and Y. Lu, "Serving and optimizing machine learning workflows on heterogeneous infrastructures," *PVLDB*, pp. 406–419, 2022.

[120] Y. Hu, R. Ghosh, and R. Govindan, "Scrooge: A Cost-Effective Deep Learning Inference System," in *SoCC*, 2021, pp. 624–638.

[121] Ultralytics, *Ultralytics yolov8*, [Online] Available from: https://github.com/ultralytics/ultralytics, 2023.

[122] M. Mitzenmacher, A. W. Richa, and R. Sitaraman, "The Power of Two Random Choices: A Survey of Techniques and Results," in *Handbook of Randomized Computing*, 2001, pp. 255–312.

[123] *Wayback Machine*, 2017.

[124] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," in *OSDI*, 2018, pp. 561–577.

[125] R. Rothe, R. Timofte, and L. V. Gool, "Deep expectation of real and apparent age from a single image without facial landmarks," *International Journal of Computer Vision*, vol. 126, no. 2-4, pp. 144–157, 2018.

[126] A. Wasay, X. Wei, N. Dayan, and S. Idreos, "Data canopy: Accelerating exploratory statistical analysis," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 557–572.

[127] J. Glenn, *Yolov5 by ultralytics*, 2020.

[128] S. Yau, *Dog breed classifier vit*, 2023.

[129] N. Raw, *Facial emotion detection vit*, 2021.

[130] D. Iakubovskyi, *Facial emotion detection vit*, 2023.

[131] Y. Anand, Z. Nussbaum, B. Duderstadt, B. Schmidt, and A. Mulyar, *Gpt4all: Training an assistant-style chatbot with large scale data distillation from gpt-3.5-turbo*, https://github.com/nomic-ai/gpt4all, 2023.

[132] *Nvidia multi-process service*, 2024.

[133] *Nvidia multi-instance gpu user guide*, 2024.

[134] D. Kang, E. Gan, P. Bailis, T. Hashimoto, and M. Zaharia, "Approximate selection with guarantees using proxies," *PVLDB*, pp. 1990–2003, 2020.

[135] D. Kang, J. Guibas, P. Bailis, T. Hashimoto, Y. Sun, and M. Zaharia, "Accelerating approximate aggregation queries with expensive predicates," *PVLDB*, pp. 2341–2354, 2021.

[136] N. Koudas, R. Li, and I. Xarchakos, "Video monitoring queries," in *ICDE*, 2020, pp. 1285–1296.

[137] Z. Lai, C. Han, C. Liu, P. Zhang, E. Lo, and B. Kao, "Top-k deep video analytics: A probabilistic approach," in *SIGMOD*, 2021, pp. 1037–1050.

[138] Z. Yang, Z. Wang, Y. Huang, Y. Lu, C. Li, and X. S. Wang, "Optimizing machine learning inference queries with correlative proxy models," *PVLDB*, pp. 2032–2044, 2022.

[139] O. Moll, F. Bastani, S. Madden, M. Stonebraker, V. Gadepally, and T. Kraska, "ExSample: Efficient Searches on Video Repositories through Adaptive Sampling," *arXiv:2005.09141 [cs]*, 2020.

[140] K. Hsieh *et al.*, "Focus: Querying large video datasets with low latency and low cost," in *OSDI*, 2018, pp. 269–286.

[141] J. Bang, G. T. Kakkar, P. Chunduri, S. Mitra, and J. Arulraj, "Seiden: Revisiting Query Processing in Video Database Systems," *PVLDB*, pp. 2289–2301, 2023.

[142] D. Kang, J. Guibas, P. D. Bailis, T. Hashimoto, and M. Zaharia, "TASTI: Semantic Indexes for Machine Learning-based Queries over Unstructured Data," in *SIGMOD*, 2022, pp. 1934–1947.

[143] P. Chunduri, J. Bang, Y. Lu, and J. Arulraj, "Zeus: Efficiently Localizing Actions in Videos using Reinforcement Learning," in *SIGMOD*, 2022, pp. 545–558.

[144] E. Zhang, M. Daum, D. He, B. Haynes, R. Krishna, and M. Balazinska, "EQUIVOCAL: Synthesizing Queries for Compositional Video Events from Limited User Interactions," *PVLDB*, pp. 2714–2727, 2023.

[145] F. Kossmann *et al.*, "Extract-transform-load for video streams," *PVLDB*, pp. 2302–2315, 2023.

[146] Y. Chen, X. Yu, and N. Koudas, "Ranked window query retrieval over video repositories," in *ICDE*, 2022, pp. 2776–2791.

[147] Y. Chen, X. Yu, N. Koudas, and Z. Yu, "Evaluating temporal queries over video feeds," in *SIGMOD*, 2021, pp. 287–299.

[148] Y. Chen, N. Koudas, X. Yu, and Z. Yu, "Spatial and temporal constrained ranked retrieval over videos," *PVLDB*, pp. 3226–3239, 2022.

[149] D. Crankshaw *et al.*, "InferLine: Latency-aware provisioning and scaling for prediction serving pipelines," in *SoCC*, 2020, pp. 477–491.

[150] A. Gujarati *et al.*, "Serving DNNs like clockwork: Performance predictability from the bottom up," pp. 443–462, 2020.

[151] C. Olston *et al.*, *TensorFlow-Serving: Flexible, High-Performance ML Serving*, 2017.

[152] NVIDIA, *Tensorrt inference server*, [Online] Available from: https://github.com/NVIDIA/tensorrt-inference-server, 2020.

[153] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Cocktail: A multidimensional optimization for model serving in cloud," in *NSDI*, 2022, pp. 1041–1057.

[154] P. Jain *et al.*, *Dynamic Space-Time Scheduling for GPU Inference*, 2018.

[155] K. Zhang *et al.*, "G-NET: Effective GPU sharing in NFV systems," in *NSDI*, 2018, pp. 187–200.

[156] P. Yu and M. Chowdhury, "Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications," in *MLSys*, 2020.

[157] K. Mahajan *et al.*, "Themis: Fair and efficient GPU cluster scheduling," in *NSDI*, 2020, pp. 289–304.

[158] NVIDIA, *Nvidia multi-process service introduction*, [Online] Available from: https://docs.nvidia.com/deploy/mps/index.html, 2021.

[159] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing," in *ATC*, 2022, pp. 199–216.

[160] *NVIDIA H100 Tensor Core GPU Architecture Overview*.

[161] DeepSeek-AI *et al.*, *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, 2025.

[162] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "ORCA: A Distributed Serving System for Transformer-Based Generative Models," *OSDI*, 2022.

[163] W. Kwon *et al.*, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *SOSP*, 2023, pp. 611–626.

[164] Y. Sheng *et al.*, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," *ICLR*, 2023.

[165] R. Y. Aminabadi *et al.*, *DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale*, 2022.

[166] A. Vaswani *et al.*, "Attention is all you need," in *NeurIPS*, 2017, pp. 6000–6010.

[167] A. Q. Jiang *et al.*, *Mixtral of Experts*, 2024.

[168] S. Zhang *et al.*, *Opt: Open pre-trained transformer language models*, 2022. arXiv: 2205.01068 `[cs.CL]`.

[169] J. Bai *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[170] S. Jiang and X. Zhang, "Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *SIGMETRICS*, 2002, pp. 31–42.

[171] H. Touvron, L. Martin, and K. Stone, "Llama 2: Open Foundation and Fine-Tuned Chat Models,"

[172] T. Kwiatkowski *et al.*, "Natural Questions: A Benchmark for Question Answering Research," *Transactions of the ACL*, pp. 453–466, 2019.

[173] S.N.Bernstein, "On a modification of chebyshev's inequality and of the error formula of laplace," *Ann. Sci. Inst. Sav. Ukraine, Sect. Math*, 1924.

[174] J. L. W. V. Jensen, "Sur les fonctions convexes et les inégalités entre les valeurs moyennes," *Acta Mathematica*, pp. 175–193, 1906.