Data Systems for Explainable AI and Incorporating AI Infrastructure into Data Systems

Dong He

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Committee:
Magdalena Balazinska, Chair
Dan Suciu
Ranjay Krishna

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Data Systems for Explainable AI and Incorporating AI Infrastructure into Data Systems

Dong He

Chair of the Supervisory Committee:
Magdalena Balazinska
Paul G. Allen School of Computer Science & Engineering

Artificial Intelligence (AI) has become a cornerstone of modern computing, powering a wide range of applications in fields from face recognition and machine translation to medical diagnosis and autonomous driving. This transformation is advancing data-driven AI models that not only learn from vast amounts of data but can generate substantial data artifacts. These changes introduce significant data management challenges, particularly as AI models grow in architectural complexity and data intensity, because traditional data management systems were not designed to handle AI workloads.

Concurrently, the tremendous computational and memory demands of AI workloads have driven the rapid development of specialized AI hardware and software infrastructure. This evolution has created systems that not only accelerate AI workloads but offer new opportunities to improve the performance of relational query processing in data management systems. Despite these advances, the diversity in hardware characteristics and programming abstractions, coupled with the lack of native support in traditional data management systems, presents significant challenges for data management system builders to fully leverage the exciting potential of such AI infrastructure.

This dissertation aims to bridge the worlds of AI and data management by introducing new data systems that efficiently support explainable AI, a subset of especially data-intensive AI workloads, and leveraging AI infrastructure to accelerate relational query processing in

data management systems.

First, we introduce two data systems for efficient explainable AI: DeepEverest and MaskSearch. Each system supports a different type of AI model explanation task. DeepEverest accelerates neural network explanation queries that return input examples with certain neuron activation patterns; these queries help practitioners understand the functionality of groups of neurons in a neural network by tying that functionality to the input examples. MaskSearch enables efficient querying over databases of image masks generated by AI models (e.g., segmentation masks, saliency maps, etc.), supporting the retrieval of masks with particular characteristics that are crucial for applications such as identifying spurious correlations, detecting adversarial examples, and monitoring model errors.

Second, we introduce the Tensor Query Processor (TQP), the industry's first query processor that compiles SQL queries into tensor programs (i.e., PyTorch programs) and executes them on any hardware backend supported by the tensor runtime, including CPUs, GPUs, and TPUs. TQP demonstrates the potential of using AI infrastructure to accelerate relational query processing in data management systems by supporting the full TPC-H benchmark and outperforming state-of-the-art systems. Further, it bridges the gap between AI workloads and relational queries by providing a unified intermediate representation for efficient execution when both types of workloads are present in the same system.

While much work remains to be done, this dissertation contributes an important step towards improving data management systems in the novel era of AI.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables v

List of Algorithms vi

Acknowledgments vii

Dedication ix

Chapter 1: Introduction 1
- 1.1 DeepEverest: Accelerating Interpretation by Example Queries 4
- 1.2 MaskSearch: Querying Image Masks at Scale 8
- 1.3 Tensor Query Processing: Incorporating AI Infrastructure into Data Systems 12
- 1.4 Summary of Dissertation Contributions 15
- 1.5 Dissertation Organization 16

Chapter 2: Related Work 17
- 2.1 Related Work for DeepEverest and MaskSearch 17
- 2.2 Related Work for TQP 23

Chapter 3: DeepEverest: Accelerating Declarative Top-K Queries for Deep Neural Network Interpretation 28
- 3.1 Preliminaries 32
- 3.2 DeepEverest 33
- 3.3 Evaluation 53
- 3.4 Discussion 63

3.5 Summary 65

Chapter 4: MaskSearch: Querying Image Masks at Scale 66
4.1 Queries over Masks 70
4.2 MaskSearch 76
4.3 Evaluation 90
4.4 MaskSearch User Interface 105
4.5 Summary 107

Chapter 5: TQP: Query Processing on Tensor Computation Runtimes 108
5.1 Background 111
5.2 Query Processing on TCRs 113
5.3 Tensor Query Processor (TQP) 117
5.4 Operator Implementation in TQP 123
5.5 Evaluation 129
5.6 Limitations and Discussion 144
5.7 Summary 145

Chapter 6: Conclusion 147
6.1 Future Work Related to DeepEverest and MaskSearch 149
6.2 Future Work Related to TQP 149
6.3 Final Remarks 150

Bibliography 151

LIST OF FIGURES

Figure Number Page

1.1 An example image from ImageNet overlaid with a saliency map. 2
1.2 An example of a Convolutional Neural Network (CNN) architecture. 4
1.3 An example of a workflow that uses *interpretation by example* queries. 6
1.4 Examples of image masks produced by machine learning tasks. 9
1.5 System design of the Tensor Query Processor (TQP). 14

3.1 An example of building the Neural Partition Index. 37
3.2 $dPar$ and ord for the execution of NTA for an example query. 39
3.3 Intermediate variables for the execution of NTA for the example query. . . . 40
3.4 An example of constructing MAI and query execution for an example query. 50
3.5 Individual query times and storage sizes for DeepEverest and baselines. . 53
3.6 Cumulative total time of DeepEverest and baselines for multi-query workloads. 55
3.7 Query times of *SimHigh* queries. 59
3.8 Speedups of query times for DeepEverest against *ReprocessAll*. 59
3.9 Speedups against *ReprocessAll* by DeepEverest with different storage budgets. 62
3.10 Cumulative preprocessing times for *PreprocessAll* and DeepEverest. . . . 64
3.11 Speedups of query times by DeepEverest w/ IQA against w/o IQA. . . . 64

4.1 Example image masks: ImageNet images overlaid with saliency maps. 67
4.2 A toy image and its mask. 72
4.3 An example of CHI, `CP`, *available region*, and C. 79
4.4 Illustration of `CP` being a (finitely)-additive function. 81
4.5 An example of MaskSearch computing the upper bounds. 86
4.6 Individual query execution time based on motivation and related work. . . . 93
4.7 Query time of MaskSearch for different query types. 95
4.8 Relationship between query time and the fraction of masks loaded for a query. 96
4.9 Distributions of bounds of $\texttt{CP}(mask, roi, (lv, uv))$ computed by MaskSearch. 98
4.10 Cumulative total time for multi-query workloads for MaskSearch and baselines. 101

4.11 Example workflow of using MaskSearch's GUI. 106

5.1 TQP represents input tables in a columnar format with a 2d-tensor per column. 118
5.2 TQP's compilation phase. 120
5.3 An example of the sort-based join implementation in TQP. 126
5.4 Scalability on selected queries from TPC-H. 133
5.5 Cost/performance tradeoff for TQP on selected TPC-H queries. 136
5.6 Query time breakdown for tensor operators for selected TPC-H queries. . . . 137
5.7 GPU utilization breakdown for selected TPC-H queries. 138
5.8 Query time on a query mixing ML model prediction and relational operators. 141
5.9 End-to-end execution time breakdown for TQP on selected TPC-H queries. 142

LIST OF TABLES

Table Number Page

3.1 Query time breakdown for baselines for a *top-k most-similar* query. 35
3.2 Summary of frequently used notation in DeepEverest. 39
3.3 Number of inputs run by the DNN at query time for *SimHigh* queries. 61

4.1 Summary of frequently used notation in MaskSearch. 83
4.2 Summary of evaluated queries based on motivation and related work. 89
4.3 Number of masks loaded during query execution. 95

5.1 Execution times of filter over ∼6M elements with Torch and TorchScript. . . 115
5.2 Query execution time on the TPC-H benchmark. 132
5.3 Query time on selected TPC-H queries with hand-optimized tensor programs. 139
5.4 Query time of TPC-H Query 6 over different hardware and software backends. 143
5.5 Lines of source code for implementing relational operators. 145

LIST OF ALGORITHMS

3.1 The Neural Threshold Algorithm for *top-k most-similar* queries. 44

3.2 The main loop of the Neural Threshold Algorithm. 45

5.1 Sort-Based Join Implementation in TQP. 125

5.2 Hash-Based Join Implementation in TQP. 128

5.3 Aggregation Implementation in TQP. 129

ACKNOWLEDGMENTS

I am forever indebted to my advisor, Magda Balazinska, for introducing me to the fascinating world of data management and for guiding me through my journey to becoming an independent researcher. Magda's wise advice, limitless patience, unwavering encouragement, and relentless optimism have been instrumental not only in advancing my research but also in shaping my approach to life. I will always remember the countless opportunities she has provided and the generous support she has extended throughout my studies.

I extend my sincere thanks to Dan Suciu for his invaluable insight into my research and for being there for me for every milestone in my PhD journey. I will never forget the excitement when Dan talks about databases and the enthusiasm when he teaches.

I want to sincerely thank Ranjay Krishna for his mentorship and guidance. I enjoyed the fun and inspiring conversations we had about research and life. Thank you, Ranjay, for your valuable advice and consistently constructive feedback.

My thanks also go to my industry mentors and collaborators: Matteo Interlandi, Konstatinos Karanasos, Jesús Camacho-Rodríguez, and Carlo Curino at Microsoft Gray Systems Lab; Sravan Nandamuri, Tianshu Bao, and Jeffrey Geevarghese at Snowflake ML Platform; Jiaqi Yan and Sandeep Seethaapathy at Snowflake SQL Execution Platform. These internships have been invaluable experiences for me and I appreciate the opportunities to work with and learn from these talented researchers and engineers.

The database group at UW CSE has been my academic family, and I feel extremely fortunate to have been part of this wonderful group. Special thanks to Enhao Zhang, Guorui Xiao, Jieyu Zhang, and Junran Yang for being my great friends, and for all the camaraderie, laughter, and support we shared; to Maureen Daum for the fantastic support in my research

and for being a great officemate who listens to my complaints and worries; to Walter Cai and Brandon Haynes for offering me incredible help and advice when I needed it; I am also thankful to other members of the database group, past and present, whom I have had the pleasure to work with and learn from: Maaz Ahmad, Leilani Battle, Alvin Cheung, Shumo Chu, Kyle Deeds, Gibbs Geng, Cheng-Yu Hsieh, Shana Hutchison, Shrainik Jain, Moe Kayali, Jack Khuu, Jonathan Leang, Jeffrey Li, Yao Lu, Anton Lykov, Ryan Maas, Parmita Mehta, Laurel Orr, Jennifer Ortiz, Ameya Patil, Guna Prasaad, Alex Ratner, Max Schleich, Kurt Stockinger, Nicole Sullivan, Remy Wang, Chenglong Wang, Jingjing Wang, Jiacheng Wu, Cong Yan, Yihong Zhang, and Dongfang Zhao.

I was equally lucky to meet and make friends with many amazing people outside the database group. A heartfelt shoutout to them for making my PhD life more colorful, enjoyable, and memorable: Varich Boonsanong, Lequn Chen, Kaiming Cheng, Weixin Deng, Jiafei Duan, Anqi Gao, Ken Gu, Chenxu He, Haiyan He, Haotian Jiang, Ziheng Jiang, Liwei Jiang, Preston Jiang, Inna Lin, Kechun Liu, Oscar Liu, Zixuan Liu, Xin Liu, Jie Mei, Yuxuan Mei, Rock Pang, Lianhui Qin, Xinming Tu, Yuhao Wan, Yizhong Wang, Orson Xu, Lu Xu, Xieyang Xu, Shirley Xue, Zihao Ye, Wentao Yuan, Han Zhang, Xiangfeng Zhu, Mingyuan Zhong, and many others whom I inadvertently missed.

To Chenxi, thank you for always being there with me through the ups and downs. I am deeply grateful for the dedicated love, supportive company, and significant compromises you have given me, and I deeply regret all the times I have let you down and not been there for you. I promise I will become a better person.

Finally, to my parents, thank you for your unconditional love and support throughout my life. I hope I have made you proud.

DEDICATION

To mom and dad.

Chapter 1

INTRODUCTION

Artificial Intelligence (AI) now powers myriads of applications in fields such as computer vision [237], natural language processing [68], speech recognition [290], and autonomous driving [64]. AI models, central to these advancements, learn from large amounts of data and generate substantial amounts of data artifacts. For example, computer vision models, which take an image as input and identify all objects within, learn from large image datasets like ImageNet [90] and generate artifacts such as bounding boxes around objects and attributes that describe the objects and scenes in the images. Developing these models, training them on large datasets, investigating what the models are learning, improving them over time, and deploying them in production are tasks that AI practitioners perform regularly. These tasks are data-intensive and require significant computational resources, which make use of specialized infrastructure like GPUs and TPUs. As AI models grow in architectural complexity and data intensity, practitioners could benefit from more efficient data management systems to better support their work.

For many decades, data management systems have effectively helped users to work with their data across various domains. For example, data management systems power most modern businesses, enabling them to store, query, and analyze their data (e.g., tracking sales, managing inventory, and making business decisions by analyzing historical data). However, the advent of the AI era introduces new challenges in data management, particularly in data storage, indexing, and query processing for AI workloads.

Traditional data management systems were not designed to accommodate these unique AI workload requirements. First, they are not designed to process large volumes of high-dimensional data records, e.g., images, text, and model artifacts like activations and gradients

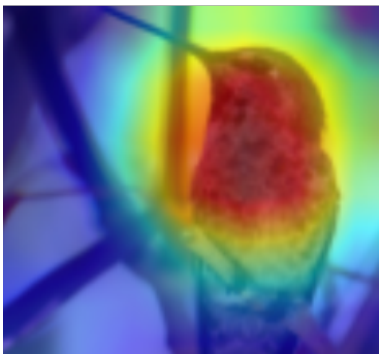


Figure 1.1: An example image from ImageNet [90] overlaid with a saliency map. The red regions indicate the pixels that are most important for the model's prediction.

that are transient for each query. Second, these systems do not address the important tradeoff between the amount of data (e.g., model artifacts) that needs to be stored and indexed on disk versus computed on-the-fly in order to answer queries specific to AI workloads that analyze large volumes of transient data. As a consequence, practitioners are finding it difficult to leverage current data management systems for their AI workloads.

Data management systems could help AI practitioners with many common workload tasks. One particularly data-intensive AI workload that has gained significant research attention is understanding what AI models have learned and identifying the data on which they exhibit specific behaviors [53, 202, 296, 104, 293, 54, 106, 284, 256, 103, 294, 279, 41, 179, 139, 72, 146, 200, 206, 268]. This task, known as *model explanation* or *interpretation*, or *explainable AI* in general, is essential for applications in high-stake domains such as healthcare and finance, and is crucial for debugging and improving models by providing insights into their prediction processes. For instance, Figure 1.1 shows a saliency map generated by a model that highlights the pixels in an image that are most important for its prediction. Explainable AI tasks pose significant challenges that involve analyzing voluminous amounts of data and model artifacts generated during modeling and prediction [240, 265]; this data can by orders of magnitude exceed the size of the original dataset and model themselves [182], complicating efforts to obtain timely insights.

The tremendous computational and memory demands of AI workloads are themselves driving the rapid development of specialized AI hardware and software infrastructure [111, 44, 276, 40, 49, 77, 138, 4, 96, 10, 36, 75]. Billions of dollars are being invested in this area [251], producing systems that offer new opportunities to both accelerate AI workloads as well as improve the performance of relational query processing in data management systems that may run in the same data centers where such AI infrastructure is deployed. Although data management system builders have effectively utilized multi-core and SIMD instructions [297, 216, 149], the proliferation of specialized AI hardware, each with its own characteristics and programming abstractions, and the lack of native support for these devices in traditional data management systems make it difficult to fully exploit their exciting potential for relational query processing.

This dissertation aims to bridge the worlds of AI and data management by developing new data systems that can efficiently support explainable AI workloads and leveraging the capabilities of AI hardware and software infrastructure to accelerate data management systems. Specifically, this dissertation introduces three innovative data systems designed to address these challenges: DeepEverest (Chapter 3) accelerates queries seeking to identify groups of examples in a dataset for which an AI model behaves similarly; MaskSearch (Chapter 4) accelerates queries that retrieve examples from a dataset based on the properties of the annotations generated by AI models; and Tensor Query Processor (TQP, Chapter 5) accelerates relational query processing by leveraging modern AI infrastructure. DeepEverest is published in PVLDB Volume 15 [115]. MaskSearch is in submission and a tech report is available on arXiv [117]. A demonstration paper on MaskSearch is published in VLDB 2024 [272]. TQP is published in PVLDB Volume 15 [116].

The remainder of this Introduction is organized into four sections. Section 1.1 and Section 1.2 highlight DeepEverest and MaskSearch, two data management systems designed to accelerate explainable AI tasks. Section 1.3 highlights TQP. Section 1.4 summarizes the contributions of this dissertation, and Section 1.5 outlines the organization of the following dissertation chapters.

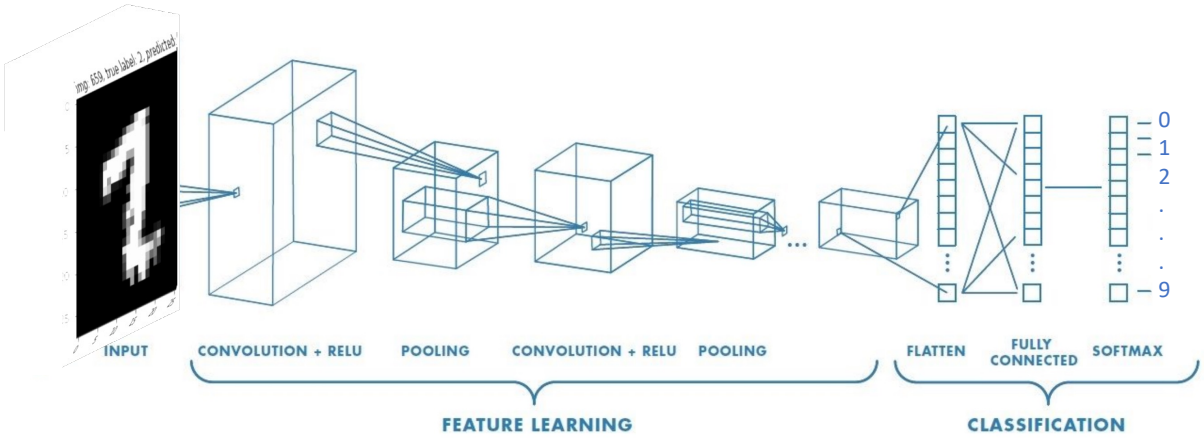


Figure 1.2: An example of a Convolutional Neural Network (CNN) architecture for image classification. The figure shows the input image, the convolutional layers, the pooling layers, the fully connected layers, and the output layer. The figure is adapted from [234].

1.1 DeepEverest*: Accelerating Interpretation by Example Queries*

Deep neural networks (DNNs) are a prominent class of AI models renowned for their ability to learn complex patterns in data, and they have been widely adopted in various AI applications such as image classification [119], object detection [118], and natural language processing [68]. Figure 1.2 shows an example of a *convolutional neural network* (CNN), a type of DNN, that performs image classification, where multiple layers of operations are connected with neurons. Each neuron outputs an activation value as the input is propagated through the network, and the final layer produces the model's prediction.

Explaining DNNs is crucial for practitioners to gain insights into the knowledge acquired by their models. Although numerous methods have been proposed, they often fail to scale effectively when processing large volumes of data artifacts for explanation purposes as models and datasets grow in size, leading to tedious model explanation and exploration processes for practitioners [240].

The fundamental building blocks of DNN interpretation are neurons. DNN interpretation techniques often perform analysis on the activation values of neurons [53, 202, 296, 104, 293, 54].

When DNNs are trained on tasks such as image classification or scene synthesis (generating images from textual descriptions), there emerge individual neurons and groups of neurons that match specific human-interpretable concepts [294, 104, 54], such as "human faces" and "trees".

To examine what individual neurons and groups of neurons learn and detect, researchers often ask *interpretation by example* queries [170], which are an important class of post-hoc interpretation methods that explain model predictions without clarifying the underlying mechanisms of the models. For example, a widely used *interpretation by example* query is, *"find the top-k inputs (e.g., images in a dataset) that produce the highest activation values for an individual neuron or a group of neurons"* [106, 284, 256, 103, 294, 279, 179, 139]. Another common query is, *"for any input and any group of neurons, use the activations of the neurons to identify the k nearest neighbors (e.g., images in a dataset) based on the proximity in the space learned by the neurons*" [72, 191, 146, 41, 200, 206, 268]. These queries help users to investigate and explain the functionalities of neuron groups by tying those functionalities to the input examples in a dataset. For instance, Mikolov et al. use the latter query to find the nearest neighbors of words in the latent space to examine the learned representations after training the word2vec model [191]. Below, we present an example of how a user might apply *interpretation by example* queries to understand a DNN.

Example 1.1.1. *As illustrated in Figure 1.3, consider a DNN trained to classify handwritten digit images from the MNIST dataset [164]. A user may want to know what parts of an image of digit 2 cause the DNN to mispredict it as digit 7 and what neurons detect these parts. To this end, the user may generate a saliency map and inspect the maximally activated neurons of different DNN layers based on the conjecture that some group of maximally activated neurons act as detectors of semantic features in the image (e.g., sharp angles that resemble the digit 7). In Figure 1.3, the user is interested in layer 12 and finds the three neurons (with activation values circled) that are most activated in the layer for the sample image. To investigate whether these neurons exhibit similar behavior for other images, the user may then ask for*

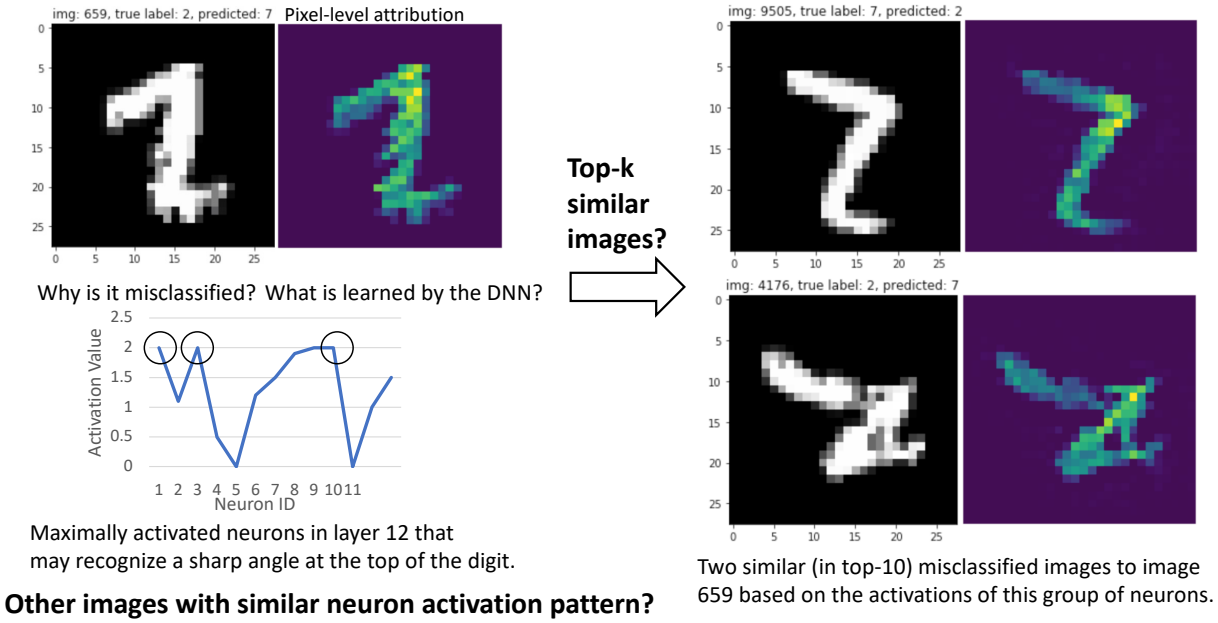


Figure 1.3: An example workflow that uses *interpretation by example* queries to explain a model's prediction. "image 659" comes from the MNIST dataset [164] and 659 is the ID of the image. Layer 12 and the activation values are hypothetical for the example.

the most similar images to the sample image based on the activation values of the group of neurons. As shown in Figure 1.3 (right), both these misclassified images returned by the query have sharp angles that explain the model's misclassification, and the three neurons with circled activations from layer 12 may be responsible for detecting these sharp angles.

Though many systems have been developed to enable various forms of DNN interpretation [41, 140, 139, 159, 227, 240, 182, 265], none supports flexible and efficient *interpretation by example* queries. There is therefore a need for a system that can efficiently support such queries to accelerate the DNN interpretation process.

To fill this gap, we design, implement, and evaluate a system called DeepEverest (Chapter 3). DeepEverest is specifically designed to accelerate *interpretation by example* queries that retrieve groups of examples in a dataset based on the activation values of neurons in DNNs. It helps researchers and practitioners efficiently and flexibly explore the inner

workings of DNNs through two types of queries: (1) *top-k highest* queries, which find the top-k input examples that produce the highest activation values for a user-specified group of neurons, and (2) *top-k most-similar* queries, which find the top-k most similar input examples based on a given input example's activation values for a user-specified neuron group. A group of neurons consists of one or more neurons within a DNN layer.

Executing these *interpretation by example* queries efficiently with low storage overhead is challenging and this is what DeepEverest addresses. A baseline approach would be to materialize the activation values for all inputs and neurons, but this requires significant storage. For example, storing uncompressed activations of a ResNet50 model for 10,000 images occupies 1.35 TB of disk space. At the other extreme, computing activation values at query time would avoid the substantial storage overhead but is compute-intensive and thus slow since it requires DNN inference for the entire dataset. Answering a *top-k most-similar* query targeting a late layer of ResNet50 on 10,000 images can take over 2 minutes, making the interpretation process tedious.

Further, although the target query is a k-nearest neighbor (KNN) search, existing KNN acceleration methods are not suitable. These methods rely on pre-built data structures, like trees or hash tables, to accelerate queries [58, 203, 174, 85, 43]. It is impractical to build a large multidimensional data structure for all neurons in each layer due to high dimensionality because DNN layers often have thousands of neurons. Constructing data structures for all possible neuron groups would either limit the user to a small set of possible queries or be extremely expensive in time and storage, with the number of possible neuron groups growing exponentially with the neuron count. In all cases, precomputing and storing all activation values in such data structures would add prohibitive storage overhead.

In DeepEverest, we introduce the neural threshold algorithm (NTA) and the neural partition index (NPI) to efficiently execute our target queries with low storage overhead. NTA, based on the classic threshold algorithm (CTA) [92], supports top-k queries for arbitrary neuron groups. Unlike CTA, which requires computing activations for all inputs at query time, NTA reduces the number of activations that must be computed during query execution,

thus significantly improving query time. During preprocessing, NPI partitions input examples and stores a small amount of information per partition that is later used by NTA to determine which activation values to compute at query time. During query execution, NTA uses insights from CTA and information from NPI to decide when to terminate as it incrementally computes activation values using DNN inference only for small subsets of inputs as needed to answer the query. Additionally, DeepEverest includes optimizations—such as incremental indexing, the maximum activation index (MAI) for *top-k highest* queries, automatic configuration selection, and inter-query acceleration (IQA)—to accelerate related query sequences.

We evaluate DeepEverest on benchmark datasets and models, and demonstrate that it *significantly accelerates individual interpretation by example queries by up to 63× and consistently outperforms other competing methods on multi-query workloads that simulate DNN interpretation processes.*

This work first appeared in PVLDB Volume 15 [115]. The code is open-source and available at `https://github.com/uwdb/DeepEverest`.

1.2 MaskSearch*: Querying Image Masks at Scale*

Many machine learning (ML) tasks over image databases commonly generate masks (2D arrays of floating-point numbers) that annotate individual pixels in images. For instance, model explanation techniques [254, 248, 241, 292, 246] generate saliency maps to highlight the significance of individual pixels to a model's output. In image segmentation tasks [118, 151, 229], *masks* denote the probability of pixels being associated with a specific class or an instance. Depth estimation models [59, 208] yield masks reflecting the depth of each pixel, while human pose estimation models [71, 109] provide masks indicating the probability of pixels corresponding to body joints. Figure 1.4 shows some examples.

Exploring the properties of these masks unlocks a plethora of applications. For instance, in the context of model explanation, examining saliency maps is the most common approach to understanding whether a model is relying on spurious correlations in the input data, i.e., signals that deviate from domain knowledge [201, 215, 61, 274, 89, 192]. Other applications based

(a) Segmentation mask (b) Depth estimation mask (c) Saliency map

Figure 1.4: Examples of image masks that annotate image content for ImageNet [90] images produced by machine learning tasks.

on the properties of masks include identifying maliciously attacked examples using saliency maps [278, 270, 285], out-of-distribution detection using saliency maps [124], monitoring model errors [31, 143, 16] using segmentation masks, traffic monitoring and retail analytics using segmentation masks [84, 83], and others.

These wide-ranging applications underscore an emerging necessity for AI practitioners: the ability to efficiently query and retrieve examples from image databases together with their masks, based on properties of the latter [215, 82, 151]. Today, practitioners lack a system to support this task efficiently and at scale.

Consider the following scenario inspired by the literature:

Example 1.2.1. *(inspired by [89]): Alice is a scientist developing a model to detect COVID-19 based on chest X-rays. She has trained a model that achieves high accuracy on both the training and validation sets from a public dataset. However, when the model is deployed to local hospitals, its predictions often contradict the diagnoses based on PCR tests. Eager to understand why her high-accuracy model is failing in real-world settings, Alice examines the model saliency maps for the chest X-rays from the training set. She discovers that the high-value pixels in the saliency maps are concentrated on the markers around the peripheries, i.e., markers on the edges of the X-rays that do not contain any medical information, instead of*

the lung regions. This observation suggests that the model is learning the confounding factors in the images (i.e., the lateral markers) rather than the medical pathology of the lungs. Figure 3 in [89] shows example X-rays with their saliency maps that exhibit this phenomenon. To further investigate, Alice wants to retrieve more examples that exhibit similar mask properties. This process often requires multiple iterations of querying and analyzing the returned examples, each time adjusting the region of interest (where the high-value pixels are expected to be) and/or pixel value range (the range of high-value pixels) specified in the query.

As this example illustrates, querying databases of masks is important in ML applications. Unfortunately, there is a lack of system support to efficiently execute these queries [123]. According to [215], to identify examples for which the model relies on spurious correlations, researchers must manually examine the model saliency maps for each image. This tedious approach is clearly untenable and invites a system that efficiently supports mask-based queries.

We therefore design, implement, and evaluate MaskSearch (Chapter 4), a system that efficiently retrieves examples based on mask properties. To build MaskSearch, we first formalize a novel and broadly applicable class of queries that retrieve images (and their masks) from image databases based on the properties of masks computed over those images. At the core of these queries are predicates on image masks that apply filters and aggregations (i.e., count of pixels) on the values of pixels within regions of interest (ROIs). We further extend the queries to support aggregations across masks and top-k computations to enhance the versatility of the supported queries. Aggregations across masks serve as a powerful tool for comparing trends of different masks, e.g., studying the difference between model saliency maps and human attention maps [82]. Top-k computations are also widely used; for example, Alice might be interested in finding the top-k X-rays whose saliency maps have the least number of high-value pixels in the lung regions.

Efficiently executing the formulated queries is challenging: The database of masks is too large to fit in memory; loading all masks from disk is slow and dominates query execution

time; compressing masks does not help due to the overhead of decompression. Existing methods, including vanilla NumPy and PostgreSQL, take over 30 minutes for queries filtering masks by pixel count within an ROI and pixel value range on ImageNet. Existing multi-dimensional indexing techniques also do not improve execution times because they cannot handle mask-specific ROIs within a single query and their complexity is high because mask data is dense. Array databases such as SciDB [67] and TileDB [205], though designed to process multi-dimensional dense arrays, are not optimized for efficiently searching through large collections of small arrays, as required in our target queries. While masks can be flattened as vectors and stored in vector databases, MaskSearch differs significantly because it targets a fundamentally different type of query.

MaskSearch accelerates the aforementioned queries with no loss in query accuracy by introducing *a new type of index and an efficient filter-verification query execution framework.* Both techniques work in tandem to reduce the number of masks that must be loaded from disk during query execution while guaranteeing the correctness of the query result. The indexing technique, which we call the cumulative histogram index (CHI), provides bounds on the pixel counts within an ROI and a pixel value range in a mask. It is designed to work with arbitrary ROIs (both mask-specific and constant) and pixel value ranges specified by the user at query time. These bounds are used during query execution when deciding whether to load and process a mask while guaranteeing the correctness of the query result.

MaskSearch's filter-verification execution framework leverages CHI to bypass the loading of masks guaranteed to satisfy or not satisfy the query predicate. Only the masks that cannot be filtered out are loaded from disk and processed. By doing so, MaskSearch overcomes the limitation of existing systems by reducing the number of masks that must be loaded to process a query. Moreover, it includes an incremental indexing approach that avoids potentially high upfront indexing costs and enables it to operate in an online setting.

We evaluate MaskSearch on large image datasets and demonstrate that it *achieves up to two orders of magnitude speedup over existing methods for individual queries and consistently outperforms existing methods on various multi-query workloads that simulate*

model explanation and dataset exploration processes.

Overall, MaskSearch is a significant next step toward the seamless and rapid exploration of a dataset based on masks generated by AI models. We believe it will become an important component in a toolbox of methods for model explanation and dataset exploration. A paper on this work is in submission, and a tech report is available on arXiv [117]. The MaskSearch code is open-source and available at `https://github.com/uwdb/MaskSearch`. A demonstration of the system is accepted at VLDB 2024 [272].

1.3 Tensor Query Processing*: Incorporating AI Infrastructure into Data Systems*

Data management system vendors have delivered consistent performance improvements for decades by evolving their software to keep pace with Moore's law while influencing hardware development through close relationships with manufacturers. Though data volumes and demand for analytics are growing faster than ever [250], performance improvements on CPUs have slowed down [261] even as the count of processor transistors continues to grow. After adopting multi-core CPU architectures, hardware manufacturers began to augment their computing platforms with specialized components such as GPUs, FPGAs, compression and encryption chips, digital signal processors (DSPs), and neural network (NN) accelerators. Although data management system builders have taken advantage of multi-core and SIMD instructions effectively [297, 216, 149], the explosive number of specialized hardware components, each with different characteristics and programming abstractions, makes it difficult to support all the exciting capabilities that these new powerful devices can offer.

On the other hand, the tremendous demand for memory and computation in AI [102], combined with the market fever for AI, is driving unparalleled investments in new AI hardware and software. Hardware makers (e.g., Intel [111], Apple [44], Xilinx [276], AMD [40]), cloud vendors and other tech giants (e.g., Amazon [49], Microsoft [77], Google [138], Meta [96]), startups (e.g., Graphcore [6], Sambanova [12], Cerebras [4]), and car companies like Tesla [259]

are investing heavily in this space. Venture capitalists alone are pouring nearly $2B a quarter into specialized hardware for AI, aiming for a market expected to exceed $200B a year by 2025 [251]. On the software side, companies and open source communities are rallying behind a small number of big efforts (e.g., PyTorch [10], TensorFlow [36], TVM [75]). The combination of investments in specialized hardware devices and large software communities focusing on performance allows these efforts to thrive.

It thus appears that the AI community has made hardware accelerators accessible to nonspecialists (e.g., data scientists). The fact that the most popular AI frameworks are open-source creates a virtuous cycle, wherein any hardware vendor interested in the AI space must support these frameworks well to get adoption. At the same time, these large open source communities successfully tackle the labor-intensive problem of providing specialized kernels for various hardware devices, e.g., a month after Apple M1 was announced, TVM outperformed Apple's CoreML by 2× [258]. Hardware vendors can also directly improve the kernels' performance or the hardware itself [23, 24, 27], which further helps adoption since the performance improves at each new software and hardware release.

We propose to leverage the groundswell of new hardware devices and software targeting AI workloads. To demonstrate the viability of this idea, we propose and prototype a new query processor that runs SQL queries atop tensor computation runtimes (TCRs) such as PyTorch, TVM, and ONNX Runtime [25]. We call our prototype *Tensor Query Processor* (TQP) (Chapter 5) and illustrate its system design in Figure 1.5. TQP transforms a SQL query into a tensor program that is executable on TCRs. To our knowledge, TQP is the first query processor built atop TCRs. Careful architectural and algorithmic design enables TQP to: (1) improve *performance* significantly relative to popular CPU-based data systems and match or outperform custom-built solutions for GPUs, (2) demonstrate *portability* across a wide range of target hardware and software platforms, and (3) achieve the above with *parsimonious* and *sustainable engineering effort*.

These achievements may seem unexpected since specialized hardware accelerators are notoriously difficult to program, requiring considerable customization to extract optimal

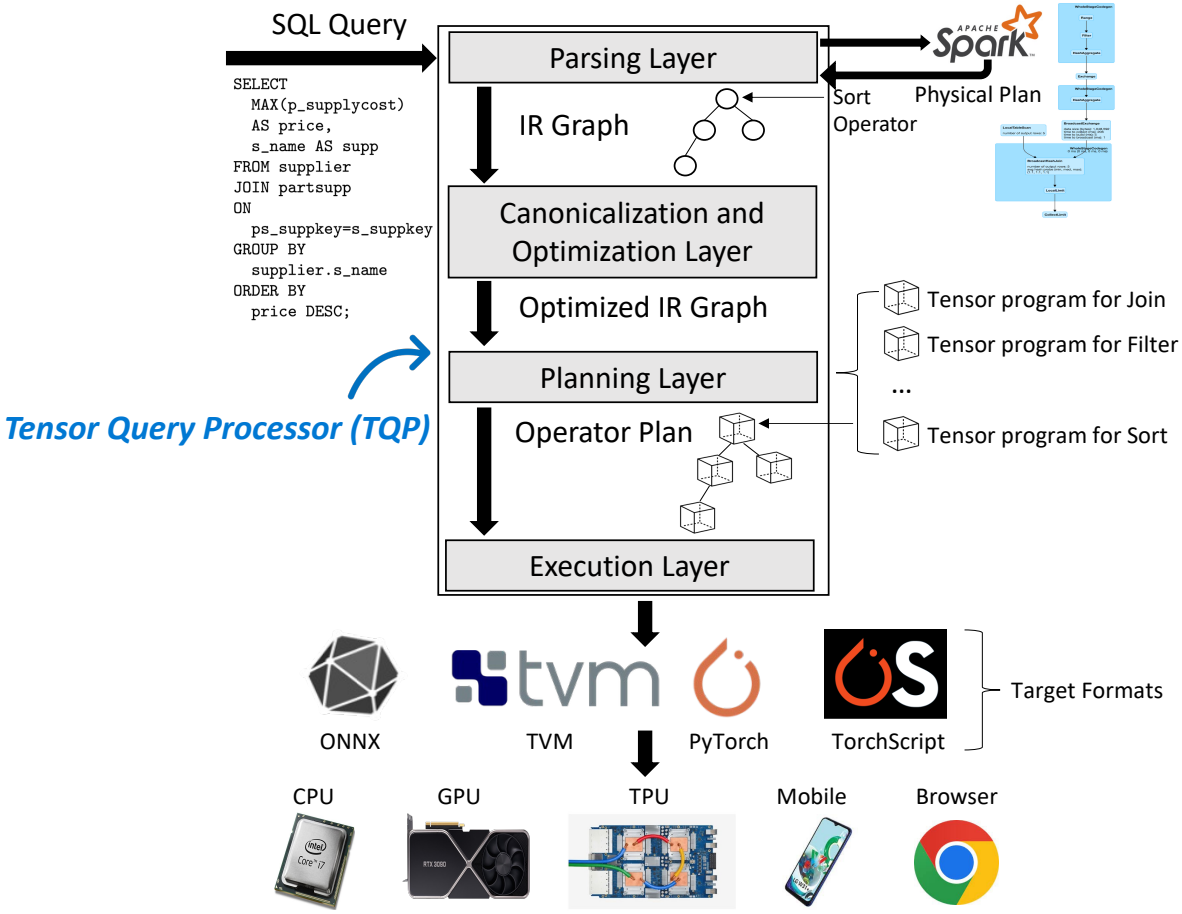


Figure 1.5: The system design of the Tensor Query Processor (TQP). IR stands for Intermediate Representation.

performance. Furthermore, their programming abstractions differ sufficiently to make our goals of *performance* (G1), *portability* (G2), and *parsimonious engineering effort* (G3) seemingly hard to reconcile. However, our innovation is a *compilation layer* and *a set of novel algorithms* that map the classical database abstraction to the prevalent one in AI, essentially *mapping relational algebra to tensor computations*. This lets us leverage labor-intensive efforts from the AI community to port and optimize TCRs across all new specialized hardware platforms.

Pursuing our goals of portability and parsimonious engineering effort, we make a deliberate decision to target existing tensor APIs rather than customize lower-level operators. Though

this decision may not initially optimize performance, it leads to sustainability over the longer term since TQP benefits from any performance enhancement and optimization added to the underlying software and hardware devices [23]. To validate this proposition, we run TQP on different hardware settings: from CPUs, to discrete GPUs, to integrated GPUs (Intel and AMD), to NN accelerators (e.g., TPUs [138]), and web browsers. Furthermore, TQP is able to run the full TPC-H benchmark on both CPU and GPU with only around 8000 lines of code, an achievement in itself considering that until 2021, no GPU database could run all 22 TPC-H queries [165].

Key evaluation results show that, *on GPUs,* TQP *outperforms open-source GPU databases in terms of query execution time; on CPUs,* TQP *outperforms Spark [282] and is comparable to a state-of-the-art vectorized engine, DuckDB [224], for several queries. Furthermore, when ML model inference is used in concert with SQL queries,* TQP *provides end-to-end acceleration for a 9× speedup over CPU baselines.*

The paper on this work is published in PVLDB Volume 15 [116], and its demo paper [47] won the Best Demo Award at VLDB 2022. A patent has also been issued for this work [129].

1.4 Summary of Dissertation Contributions

This dissertation introduces techniques and systems to address two core data management challenges in the novel era of AI: (1) query performance issues in AI model explanation tasks, and (2) the inability of existing data management systems to efficiently leverage the powerful capabilities of new AI hardware/software infrastructure. Specifically, our core contributions are three systems that introduce novel techniques to address these challenges: (1) DeepEverest, a system that accelerates *interpretation by example* queries for DNN interpretation; (2) MaskSearch, a system that accelerates queries over databases of image masks for model explanation and dataset exploration; and (3) TQP, a query processor that compiles SQL queries into tensor programs and executes them on any hardware devices supported by the tensor computation runtime. In summary, this dissertation contributes new approaches to productively connect the two worlds of AI and data management by improving

data management systems for explainable AI and leveraging AI infrastructure for accelerating data management systems.

1.5 Dissertation Organization

Chapter 2 discusses related work. Chapter 3 presents DeepEverest, and Chapter 4 presents MaskSearch. We discuss TQP in Chapter 5. Finally, Chapter 6 concludes the dissertation.

Chapter 2
RELATED WORK

This chapter presents an overview of the related work for the three components of this dissertation: DeepEverest, MaskSearch, and TQP. It is divided into two parts: one for DeepEverest and MaskSearch, covering related work for data systems for explainable AI, and the other for TQP, covering related work for relational query processing on AI hardware accelerators.

2.1 Related Work for DeepEverest *and* MaskSearch

The main areas of related work for DeepEverest and MaskSearch are DNN interpretation, image masks in AI tasks, data systems for AI model management and debugging, nearest neighbor search, and top-k query processing, multimedia databases and querying, and vector databases. This section presents each area in turn.

2.1.1 DNN Interpretation.

A variety of approaches have been proposed to interpret the internals of DNNs [53, 202, 296, 104, 293, 54, 106, 284, 256, 103, 294, 279, 41, 179, 139, 72, 146, 200, 206, 268]. Approaches like CAM [295], Grad-CAM [241], and Integrated Gradients [254] generate saliency maps that highlight the regions of the input image that contribute the most to the model's prediction. Other approaches like LIME [227] generate local explanations by approximating the model's behavior around a specific input by simpler and more interpretable surrogate models.

Many approaches and systems in this space ask *interpretation by example* queries that return the most similar inputs with respect to the activations of a group of neurons of a given input, or inputs that maximally activate a group of neurons [170, 191, 72, 146, 41, 200, 206,

268]. They motivate the design of DeepEverest, which does not invent new interpretation methods but rather builds novel indexes and algorithms that accelerate the execution of these commonly asked queries for DNN interpretation.

2.1.2 Image Masks in AI Tasks.

Masks are widely used in AI tasks to annotate image content, e.g., saliency maps [254, 248, 241, 292] and segmentation maps [118, 151, 229]. Practitioners use them for a variety of applications, including identifying maliciously attacked examples [278, 270, 285], detecting out-of-distribution examples [124], monitoring model errors [31, 143, 16], and performing traffic and retail analytics [84, 83].

These applications motivate the design of MaskSearch and could utilize MaskSearch's efficient query execution to quickly retrieve examples that satisfy the desired properties. For example, in [201], the authors found that models trained to detect pneumothorax (collapsed lungs) rely on the presence of chest drains, a device used during treatment, rather than the actual pathology. MaskSearch could be used to quickly retrieve images for which the model relies on the presence of chest drains to make predictions.

2.1.3 Data Systems for AI Model Management and Debugging.

Many systems have been proposed to support AI model management [266, 186, 240, 265, 182, 275]. ModelDB [266] is designed to automatically track and index AI models. It contains native client libraries for various machine learning environments (e.g., spark.ml, scikit-learn), a backend for model storage, and a web interface for exploring and analyzing AI models and pipelines. ModelHub [186] is similarly designed and optimized for deep learning models and workflows. It enables users to store, version, snapshot, query, and share deep learning models and their associated data artifacts. Both ModelDB and ModelHub are designed to support a different set of functionalities than DeepEverest and MaskSearch, focusing on model versioning and search functionality rather than querying over neuron activation patterns or image masks.

DeepEverest and MaskSearch fall into the group of systems that support efficient model diagnosis and interpretation. A number of systems like ModelTracker [41], CNNVis [172], and others [70, 140, 158, 139, 177, 161, 279] support visual inspection of AI models and features, such as visualizing model performance and discrepancies between prediction results on different input examples. However, these systems do not support querying over neuron activation patterns or image masks that are the focus of DeepEverest and MaskSearch. They could utilize DeepEverest and MaskSearch to accelerate some of the queries used to build the visualizations.

DeepBase [240] abstracts model diagnostic queries as hypotheses verification tasks, e.g., do neurons learn language nuances such as relational dependencies? It lets users identify neurons that have statistical dependencies with user-specified hypotheses. It takes as input a trained neural network, a test dataset, and a set of hypotheses represented as Python functions, and outputs the affinity between the neuron activations and the hypotheses. DeepBase allows users to quickly identify neurons that have statistical dependencies with user-specified hypotheses by techniques such as caching, early stopping, and streaming execution. However, DeepBase does not support *interpretation by example* queries that are the focus of DeepEverest; it also does not support querying over image masks, which is the focus of MaskSearch.

Rain [275] proposes complaint-driven training data debugging where a user specifies errors in the model's inference results and the system ranks the training examples by using influence functions [153] such that fixing the top-ranked examples will improve the model's performance the most. Rain is designed to support a different set of queries that retrieve examples in a dataset than DeepEverest (retrieving examples based on neuron activation patterns) and MaskSearch (retrieving examples based on properties of image masks).

Parmita et al. [182] proposes a system that uses sampling techniques for model diagnosis. It utilizes model decision boundaries to select samples to diagnose the model's behavior. However, it only focuses on aggregate queries, which are different from the queries supported by DeepEverest and MaskSearch.

MetaStore [288] is a system designed to efficiently collect, store, and analyze gradients in

deep learning models. It supports efficient analysis on gradients by storing compact intermediates called prefix and suffix gradients, which can exactly reconstruct the original gradients when needed, and by enabling direct execution of analytical operations on these compact intermediates. MetaStore is designed to support a different set of queries than DeepEverest and MaskSearch, focusing on gradient analysis rather than neuron activation pattern analysis.

MISTIQUE [265] is designed to capture, store, and query model artifacts, such as neuron activations, for efficient model diagnosis. Its technical contributions include activation quantization for neural networks to minimize storage costs without significant accuracy loss and similarity-based compression to eliminate redundant data. These techniques are orthogonal to DeepEverest's and could be incorporated in DeepEverest to further reduce the storage overhead. MISTIQUE also proposes a cost model that captures the tradeoff between materialization and recomputation of the activations for different layers and makes materialization decisions accordingly. It is possible to use this cost model in a model artifact caching algorithm, and we compare it with DeepEverest in our experiments (Section 3.3). None of these systems have addressed the problem of accelerating *interpretation by example* queries well because none of them reduce the number of activation values computed during query execution as DeepEverest does.

Meta's Segment Anything [151] allows users to query for images and segmentation masks based on the mask area and the number of masks per image. This system is designed for querying over segmentation masks, which is similar to MaskSearch. However, Segment Anything does not support retrieving masks based on filter conditions on aggregations over pixel values in arbitrary regions of interest, which is the focus of MaskSearch.

In summary, DeepEverest is the first system to formalize and support querying over neuron activation patterns for arbitrary groups of neurons; MaskSearch is the first system to formalize and support querying over image masks for arbitrary regions of interest and pixel value ranges.

2.1.4 Nearest Neighbor Search.

DeepEverest's target query is a k-nearest neighbor (KNN) search. While many methods exist for exact [58, 110, 203, 174] and approximate [85, 43, 273, 131, 137] KNN, the challenge in DeepEverest is different. These KNN methods must know what dimensions will be queried before constructing their corresponding data structures in that space. In DeepEverest's problem setting, the dimensions of the KNN search are not known ahead of time; they are defined by the neuron group specified by the user only at query time. Further, the groups of neurons targeted by different queries are probably different. Therefore, if one were to use traditional KNN methods, one would have to construct a separate index for each group of neurons at query time, which would be prohibitively expensive. DeepEverest's novel indexing technique and query execution algorithm are designed to address this challenge by constructing a single index that can be used to answer queries for arbitrary groups of neurons.

2.1.5 Top-K Query Processing.

DeepEverest's target query is essentially a top-k query which retrieves the k input examples with the highest activation values for a group of neurons or the lowest distance to a sample example based on the neuron activations. Top-k query processing is formalized by the seminal work on the threshold algorithm [92]. The algorithm scans multiple sorted lists and maintains an upper bound for the aggregate score of unseen objects. Each newly seen object is accessed (by random access) in every other list, and the aggregate score is computed by applying the scoring function to the object's value in every list. The algorithm terminates after k objects are seen with scores greater than or equal to the upper bound.

Many follow-up approaches propose approximation, optimizations, and extensions [262, 128, 51, 38, 204, 113, 289], but they assume that accesses are available to the underlying data sources. This assumption does not hold in the problem setting of DeepEverest. The activations required for the top-k query cannot be stored on disk because of the prohibitive storage overhead. Computing them at query time also incurs significant computation overhead.

DeepEverest builds on the seminal work on the threshold algorithm [92]. What is novel in DeepEverest is that it avoids computing as many activation values as possible at query time, while keeping the storage overhead low, by building the indexes we design and using these indexes in a modified threshold algorithm for query execution. The modified threshold algorithm in DeepEverest is proved to be instance-optimal in Section 3.2.5.

2.1.6 Multimedia Databases and Querying.

Many systems and techniques support efficient queries over multimedia databases [55, 226, 97, 60, 236, 87, 114, 88, 287, 286, 144, 52, 141, 86, 291, 171]. However, these methods are not optimized for the target queries of MaskSearch. For example, VDMS [226] focuses on retrieving images based on metadata, while DeepLake [112] supports content-based queries but lacks support for querying based on aggregations over pixels.

Array databases [67, 205] are designed for handling multi-dimensional dense arrays. They support efficient operations and queries like slicing, dicing, and aggregation over large arrays, which are essential for applications where data is represented as array data (e.g., satellite imagery, climate data). However, they do not efficiently support searching through large numbers of arrays. In contrast to MaskSearch, these existing systems do not reduce the work required to execute the target queries. Incorporating MaskSearch's techniques could enhance the efficiency of existing systems.

Existing image indexing techniques [97, 60, 236] primarily support similarity search queries based on visual features such as color, shape, and texture, but do not cater to MaskSearch's target queries. Moreover, existing multi-dimensional indexes, as further discussed in Section 4.1.2, are ill-suited for the target queries of MaskSearch. There are two reasons: (1) they do not support mask-specific ROIs within a single query; (2) their complexity is high because mask data is dense. Assuming a constant ROI for all masks, these techniques require representing each mask's pixel as a point in the space of $(x, y, \text{pixel value})$, where x and y are coordinates. In this space, MaskSearch's target query is an orthogonal range query followed by an aggregation by `mask_id`. The best known algorithm [73, 74], range

trees, has a query time of $O(k + \log^2 n)$ and a preprocessing time of $O(n \log^2 n)$, where n is the number of total mask pixels in the dataset, and k is the number of pixels in the cuboid defined by `roi` and `(lv, uv)`. n is extremely large because mask data is dense, which makes using these indexes infeasible.

2.1.7 Vector Databases.

Vector databases [32, 33] have recently gained popularity. While mask data can be represented as vectors, MaskSearch is different from a vector database because it targets a fundamentally different workload. Vector databases specialize in vector similarity searches: given a vector as input, they return the top-k most similar vectors. In contrast, MaskSearch supports queries that investigate model performance or input data properties: queries that contain predicates over pixel counts (e.g., find all masks with large salient regions), predicates over aggregations of pixel counts (e.g., find all masks with few salient pixels in their foreground regions), and top-k computations over pixel counts within areas of interest (e.g., find masks with the most salient pixels inside their foreground regions). MaskSearch also supports queries that combine data generated by different sources (e.g., find all images where a model saliency map deviates the most from the corresponding human attention map). Vector databases do not support such queries.

2.2 Related Work for TQP

The main areas of related work for TQP are common representations for relational and AI workloads, GPUs and hardware accelerators for relational queries, query processing on heterogeneous hardware, vectorized execution, query compilation, and columnar databases. This section presents each area in turn.

2.2.1 Common Representations for Relational and AI Workloads.

Since the 1990s [196], there have been many works trying to integrate relational queries with data science and AI workloads [162, 121, 95, 239, 238, 183, 235, 219, 145, 69, 255, 17, 271,

155, 213, 247, 130, 280, 63, 127, 80, 62, 187, 147, 39]. To our knowledge, TQP is the first to propose executing relational queries over tensor computation runtimes (TCRs). Earlier attempts tried to run a few relational operators on the TPU using TensorFlow [122]. TQP is orthogonal to previous efforts to optimize relational and tensor algebra (e.g., [127, 271]), and we believe TQP can leverage them to improve its performance further. An analysis of matrix query languages can be found in [101]. In this dissertation, we focus on TCRs' tensor interface, which is more flexible than a linear algebra API. Nevertheless, it will be interesting to study whether we can effectively restrict TQP to only use linear algebra operations to improve relational query performance.

SciDB [252, 228] is a database using arrays as the base data representation. TensorDB [150] further proposes support for tensor data and decomposition operations inside databases. SciDB [252], TensorDB [150], and TQP suggest using a format closer to data science and AI to represent data. However, TQP further exploits TCRs to run both relational and AI workloads on hardware accelerators. DuckDB [224] is a state-of-the-art vectorized database system that can be seamlessly embedded within data science pipelines. It does not support running relational operators on hardware accelerators, but allows users to register and natively run relational queries over Pandas dataframes on CPUs. TQP can also be integrated with data science pipelines, and it is further able to operate on hardware accelerators and natively supports AI workloads.

TQP is the culmination of a series of works around using TCRs for workloads beyond AI. Hummingbird [195] was the first system showing that it is possible to run traditional machine learning (ML) models on hardware accelerators using TCRs. Raven [145] showed that it is possible to optimize relational and ML operators end-to-end by casting them into a unified intermediate representation (IR) built on top of the ONNX format [8]. TQP is the latest work along this line. With TQP, we demonstrate that it is possible to build a query processor on top of TCRs and the tensor abstraction. As a byproduct, TQP compiles machine learning and relational workloads into a unified intermediate representation, i.e., tensor programs that can be optimized and accelerated end-to-end.

2.2.2 GPUs and Hardware Accelerators for Relational Queries.

In the last decade, several systems have explored running relational queries over GPUs [243, 165, 281, 178, 209, 210, 218]. Paul et al. [211] provides a comprehensive survey of GPU-accelerated databases. With this trend, there emerge several startups and open-source projects [3, 7]. However, the majority of them focus mostly on microbenchmarks, while, to our knowledge, only RateUpDB [165] can support the full TPC-H benchmark. RateUpDB is a heterogeneous HTAP database system whose query engine is inherited GPUDB [281]. It has built-in query operators using CUDA/OpenCL and a code generator to generate CUDA or OpenCL code for query execution. In contrast, TQP compiles relational queries into the tensor abstraction, which can be run on any hardware platform supported by TCRs. TQP is able to run the full TPC-H benchmark on both CPU and GPU, thanks to TCRs' flexibility to support different hardware backends.

TCUDB [126] suggests using the Tensor Core Unit (TCU) of GPUs for accelerating relational operators. TCUDB requires an expensive transformation from tables to matrices and also uses low-level CUDA kernels, while TQP takes advantage of the high-level tensor interface of TCRs.

While GPUs are the default hardware for running neural network models, there has also recently been a rise in custom ASICs [138, 6, 4, 12, 44, 96] purposely built for ML workloads. We believe that ML workloads will become ubiquitous both on the cloud and the edge, whereby the availability of hardware accelerators and custom ASICs will be widespread. With TQP, we proposed a solution allowing us to run relational queries on any hardware platform supported by TCRs, since many ASICs [11, 5, 138] provide high-level interfaces directly through TCRs or are targetable through tensor compilers [75, 163].

2.2.3 Query Processing on Heterogeneous Hardware.

Several recent efforts have started to explore query execution over heterogeneous hardware, such as CPU-GPU co-execution [230, 269, 66, 76, 214, 120, 231, 99]. Many of them rely on

OpenCL [9] to target different hardware platforms. However, targeting a common language (or similarly a generic compiler, e.g., MLIR [163]), requires non-trivial engineering effort since each device requires proper tuning [214], algorithms, and data structures (as well as abstractions/dialects in the MLIR case). In contrast, TQP can natively run on any hardware platform supported by TCRs, and uses TCRs' tensor operation implementations and compilation stacks. Currently, the user has to specify which fragment of the query should run on which hardware device, but we are exploring how to automate this and enable co-execution.

A trend arises recently that suggests splitting relational operators into smaller functions that can be easily composed and efficiently dispatched over heterogeneous hardware [50, 267, 156]. TQP fits in this trend, whereby tensor operations are sub-components.

2.2.4 Vectorized Execution, Query Compilation, and Columnar Databases.

Interpreted (volcano-style [108]) query execution has been the default for databases for several decades. MonetDB/X100 [65] pioneered the vectorized execution model as well as the columnar data layout [253]. TQP follows a similar design as columnar databases, where data is stored in a columnar format with virtual IDs [35], but each column is represented as a tensor.

Recent works, such as HyPer [197] and others [242, 185, 198], have focused on query compilation. Nevertheless, since (1) there is no clear winner between query compilation and vectorized execution [148]; (2) many industry-grade systems use vectorized execution because it is easier to debug and profile [57]; and (3) compiled systems start to move to vectorized execution (e.g., Spark with Photon), we evaluate TQP against a state-of-the-art vectorized engine, DuckDB [224].

On the ML systems side, TensorFlow initially embraced a compiled (graph) execution model [36], while PyTorch pioneered interpreted (eager) execution [207]. As of version 2.0 eager execution is also the default on TensorFlow. Compilers [75, 15, 30, 163, 152, 93, 94] and optimization techniques [133, 134, 132] for neural networks have been developed to

optimize the execution of ML models. With TQP, we aim to ride the wave of innovation in this domain. While TQP is already integrated with the PyTorch stack and it can generate TorchScript and TVM models, we are planning to also add the ability to generate TensorFlow programs and integrate with MLIR. For TQP, interpreted vs. compiled execution is just another point in the query optimization space, since TCRs allow switching between them seamlessly.

Chapter 3

DEEPEVEREST: ACCELERATING DECLARATIVE TOP-K QUERIES FOR DEEP NEURAL NETWORK INTERPRETATION

Deep neural networks (DNNs) are increasingly used by AI applications. When training and deploying DNNs, interpretation is important for researchers to understand what their models learn. DNN interpretation is a relatively new field of research, and techniques are evolving. While many new approaches are developed, they often do not scale with the size of the datasets and models [240]. The problem we address in this chapter is the efficient execution of a common class of DNN interpretation queries.

The fundamental building blocks of DNN interpretation are neurons. Each neuron outputs an activation value as the input is propagated through the network. DNN interpretation techniques often perform analysis on these activation values of neurons [53, 202, 296, 104, 293, 54]. When DNNs are trained on tasks such as image classification or scene synthesis, there emerge individual neurons and groups of neurons that match specific human-interpretable concepts [294, 104, 54], such as "human faces" and "trees".

To understand what individual neurons and groups of neurons learn and detect, researchers often ask *interpretation by example* queries [170], which are important constituents of the class of post-hoc interpretation methods that are applied to trained models, as opposed to methods that achieve interpretability by restricting model complexity [193]. A widely used *interpretation by example* query is, *"find the top-k inputs that produce the highest activation values for an individual neuron or a group of neurons"* [106, 284, 256, 103, 294, 279, 179, 139]. Another common query is, *"for any input, find the k-nearest neighbors in the dataset using the activation values of a group of neurons based on the proximity in the latent space defined by*

the group of neurons" [72, 191, 146, 41, 200, 206, 268]. These queries help with investigating and understanding the functionalities of neuron groups by tying those functionalities to the input examples in the dataset. Moreover, these queries are staple techniques for verifying hypotheses of what groups of neurons learn and detect. For instance, Mikolov et al. ask the latter query to find the nearest neighbors of words in the latent space to examine the learned representations after training the word2vec model [191]. As a concrete example, consider a DNN trained to classify images. A user may be interested in understanding what parts of a dog image cause the DNN to predict its class. The user may inspect the maximally activated neurons of different layers in the DNN based on the conjecture that groups of maximally activated neurons act as semantic detectors of features in the image (e.g., floppy ears). To investigate whether these neurons exhibit similar behavior for other images, the user may then ask for the most similar images to the sample image based on the activation values of a group of neurons.

This chapter presents a system called DeepEverest that focuses on accelerating the aforementioned two kinds of queries: (1) find top-k inputs that produce the highest activation values for a user-specified group of neurons, and (2) find the top-k most similar inputs based on a given input's activation values for a user-specified neuron group. A group of neurons consists of one or more neurons within a layer of the DNN. We call the first type of query the *top-k highest* query and the second type of query the *top-k most-similar* query.

Executing these *interpretation by example* queries efficiently with low storage overhead is challenging. A baseline approach is to materialize the activation values for all inputs and all neurons. However, this approach requires significant storage space. For example, storing all the activations uncompressed of ResNet50 for 10,000 images occupies 1.35 TB of disk storage. At the other extreme, computing all activation values at query time imposes no storage overhead, but is compute-intensive and extremely slow because it requires DNN inference to compute the activation values on the entire dataset at query time. For instance, answering a *top-k most-similar* query that targets a relatively late layer of ResNet50 on a dataset of 10,000 images takes more than 120 seconds, which renders the DNN interpretation process

tedious.

Further, although the target query is a k-nearest neighbor (KNN) search, existing approaches that accelerate KNN queries are not applicable. KNN methods rely on building efficient data structures such as trees [58, 203, 174] or hash tables [85, 43] in advance for faster query execution later. One could try to build a single, large, multidimensional data structure for all neurons in each layer. However, such an index would not perform well because of its very large dimensionality. DNNs frequently have layers with multiple thousands of neurons, thus dimensions. One could build data structures for all possible neuron groups that a user could query. However, this would either limit the user to a small set of possible queries or would be prohibitively expensive both in time and storage because the number of possible neuron groups grows exponentially with the number of neurons in each layer. Additionally, in all cases, precomputing and storing all activation values in such data structures would add prohibitive storage overhead.

While many systems have been developed to enable various forms of DNN interpretation [41, 140, 139, 159, 227, 240], none supports flexible and efficient *interpretation by example* queries. In prior work [182], we investigated the use of sampling for model diagnosis. That work, however, focused only on aggregate queries. The closest work to DeepEverest is MISTIQUE [265]. It introduces storage techniques such as compression and quantization, which are orthogonal to DeepEverest and could complement our approach. It is, however, possible to use some of MISTIQUE's techniques as a caching algorithm, which we compare against in our experiments.

In DeepEverest, we design an index called the Neural Partition Index (NPI), and an efficient query execution algorithm, called the Neural Threshold Algorithm (NTA), which has low storage overhead, reduces the number of activation values that must be computed at query time, and guarantees the correctness of the query results. NTA builds on the classic threshold algorithm (CTA) [92], which supports top-k queries that target arbitrary neuron groups. However, CTA requires the computation of the activations of all inputs in the dataset at query time. Because the time to compute the activations by performing DNN inference

(not the calculation of the top-k result) dominates the end-to-end query time, CTA would not accelerate our target queries. We argue that for any algorithm to improve query time, it must reduce the number of inputs on which DNN inference is run at query time. DeepEverest achieves this reduction of DNN inference at query time while keeping the storage overhead low by building NPI and using it in NTA. Rather than store the raw activations for all neurons, NPI partitions the inputs and stores a small amount of information per-partition that is useful when deciding which activation values to recompute at query time. NTA then uses insights from CTA to decide when to terminate as it incrementally computes activation values using DNN inference only for small subsets of inputs as needed to answer the query. We analyze NTA and show that it is instance optimal for finding the query results of our target queries.

In addition to its fundamental approach, DeepEverest also includes several important optimizations: (1) incremental indexing to avoid large preprocessing overhead; (2) Maximum Activation Index (MAI) to accelerate *top-k most-similar* queries that target maximally activated neurons and *top-k highest* queries; (3) automatic configuration selection; and (4) Inter-Query Acceleration (IQA), which further speeds up sequences of related queries.

Contributions. In summary, the contributions of this chapter are:

- We propose, design, and implement a system called DeepEverest that includes an efficient index structure and an instance optimal query execution algorithm that accelerates *interpretation by example* queries for DNN interpretation while keeping the storage overhead low.
- We develop additional optimizations for DeepEverest that further accelerate individual queries, automatically select a good configuration for the system, and accelerate sequences of related queries.
- We evaluate DeepEverest on benchmark datasets and models. We demonstrate that DeepEverest, using less than 20% of the storage of full materialization, significantly accelerates individual *interpretation by example* queries by up to 63.5× and consistently

outperforms other methods on multi-query workloads that simulate DNN interpretation processes.

Organization. The rest of this chapter is organized as follows. Section 3.1 introduces some background and defines the target queries. Section 3.2 presents DeepEverest, including its index structure, query execution algorithm, and optimizations. The evaluation of DeepEverest is presented in Section 3.3. Section 3.4 discusses some potential optimizations and extensions for DeepEverest. Finally, Section 3.5 summarizes the chapter.

3.1 Preliminaries

A DNN consists of layers composed of units, called neurons, connected by edges with associated weights. Inputs to a DNN are propagated through the layers. The output of a neuron is a linear combination of its inputs and their associated edge weights that is optionally transformed by a nonlinear activation function. For example, an activation function may output only non-negative values by mapping negative values to 0 [194], or scale inputs to values in the range $(0, 1)$. The output of each neuron for a given input is called its *activation value* or *activation*. DNN interpretation often involves the study of these activations. Typical questions that researchers may ask include the *interpretation by example* queries. These queries enable researchers to reason about what the DNN learns and identify how groups of neurons match human-interpretable concepts. In this chapter, we address the problem of enabling fast queries over activations in a DNN. Conceptually, a DNN and an input dataset can be described by the relations `Neuron(neuronID, layerID, ...)` and `Artifact(inputID, neuronID, activation)`.

DeepEverest supports two fundamental classes of queries over activations: *top-k highest* queries that find the top-k inputs that produce the highest activations for a user-specified group of neurons and *top-k most-similar* queries that find the top-k inputs that are most similar to a user-specified target input based on the activations of a user-selected group of neurons. The rank of an input is decided by a user-specified distance function (or a default function), dist. Based on the user-selected neuron group, for *top-k highest* queries, dist takes

as input a set of activations and measures their magnitude. For *top-k most-similar* queries, DIST measures the distance between the input and the target input, and it takes as input a set of absolute differences between the input's activations and the target input's activations. Note that *top-k highest* queries can be considered as *top-k most-similar* queries with a hypothetical target input whose activations are infinite for all neurons. This distance function DIST must be monotonic, i.e., $\text{DIST}(x_1, x_2, \ldots, x_n) \leq \text{DIST}(x'_1, x'_2, \ldots, x'_n)$ whenever $x_i \leq x'_i$ for each i. Monotonicity is satisfied by common distance functions, such as l_p-distances, cosine distance (once transformed to normalized l_2-distance), and weighted distances like Mahalanobis distance, among others. The default distance function in DEEPEVEREST is l_2-distance.

3.2 DeepEverest

In this section, we first consider baseline approaches, then describe how DEEPEVEREST improves upon these baselines to accelerate query execution while keeping the storage overhead low.

3.2.1 Baselines

We first discuss baseline approaches and explain why applying CTA or any KNN algorithm would not improve the query time.

PreprocessAll. The first baseline, *PreprocessAll*, has a high storage cost. It performs DNN inference on the entire dataset and stores all the activations for all neurons ahead of time. It executes queries by loading the previously stored activations of the neuron group for all inputs from disk and maintaining a top-k result set.

ReprocessAll. The second baseline, *ReprocessAll*, has a high computation cost. It has no storage overhead and performs no preprocessing. It executes queries by computing the activations of the layer being queried by DNN inference on all inputs and maintaining a top-k result set as it performs the computation.

LRU Cache. The third baseline, *LRU Cache*, is a disk cache that has a fixed storage budget with a least-recently-used (LRU) replacement policy. This strategy strikes a balance between the storage overhead of *PreprocessAll* and the computation overhead of *ReprocessAll*. *LRU Cache* maintains a fixed-sized disk cache that stores the activations for queried layers. A query is executed as in *PreprocessAll* if the activations of the queried layer are present in the disk cache. Otherwise, it is executed as in *ReprocessAll*. After that, the activations of the queried layer are persisted to the disk cache. When the size of the disk cache exceeds the storage budget, the cache evicts the activations of the least recently used layer.

Priority Cache. The final baseline, *Priority Cache*, is a technique adapted from MISTIQUE [265]. It has a fixed-sized disk cache to store the activations for some layers. As a preprocessing step, it uses the storage cost model from [265] to pick which layers to store, assuming that each layer will be queried the same number of times. Under the storage budget, this cost model prioritizes the layers that save the most query time per GB of data stored. It performs DNN inference on every input and stores the activations for the layers selected ahead of time. A query is executed as in *PreprocessAll* if the activations of the queried layer are present in the disk cache. Otherwise, the query is executed as in *ReprocessAll*.

CTA could be applied to each baseline by first using the materialized or recomputed activations to construct the `Artifact` table defined in Section 3.1. `Artifact` is then used to construct a relation in which each row represents an input, and each column represents a neuron and contains the absolute difference between the activation of that row's input and the target input's activation on the column's neuron. CTA can run after sorting the absolute differences in each column in ascending order, using any monotonic norm of the absolute differences as the aggregation function.

However, applying CTA (or any KNN algorithm) on top of each of the baselines would not improve the query time. Using it along with *ReprocessAll* would not help because *ReprocessAll* requires running DNN inference on the entire dataset to compute `Artifact` at query time. Similarly, applying it on *PreprocessAll* would not improve the query time because generating the relation of absolute differences requires a full scan over `Artifact` before CTA can be

Table 3.1: Query time breakdown for baselines for a *top-k most-similar* query on *ImageNet-ResNet50* (query: *SimHigh*, neuron group size: 3, layer: *late*; detail in Section 3.3.1).

Method	*ReprocessAll*	*CTA* [92]	*K-D Tree* [58]	*Ball Tree* [203]
Total query time	121.6 s	121.6 s	121.8 s	121.7 s
DNN inference time	121.4 s	121.4 s	121.4 s	121.4 s

applied. The top-k result could already be computed during the full scan. Further, applying it on *PreprocessAll* incurs this strategy's prohibitive storage overhead. Applying it on top of *LRU Cache* and *Priority Cache* would not improve the query time for the same reasons as *PreprocessAll* (for layers in the cache) and *ReprocessAll* (for layers not in the cache). Given the prohibitive storage overhead of *PreprocessAll*, any existing method that supports queries for arbitrary neuron groups must compute the activations of the neuron group for all inputs at query time. Table 3.1 shows the query time breakdown for various baselines. The total query time consists of the time for DNN inference to compute the activations of the queried neuron group, the time for building the data structure required for each method (it cannot be computed ahead of time because the neuron group for a query can be arbitrary), and the time to obtain the top-k result. As the results show, DNN inference is the bottleneck of query execution. Therefore, any method that does not reduce the number of inputs fed into the DNN at query time will perform similarly to *ReprocessAll*. Hence, the query time of *ReprocessAll* can represent that of these more advanced methods.

3.2.2 Overview of DeepEverest

As described above, directly applying CTA does not improve the query time because `Artifact` must be fully computed for the neuron group at query time, which requires DNN inference on all inputs. Query execution can be significantly accelerated by avoiding running the DNN on inputs that will not be one of the top-k results.

We design and build a novel index, called the Neural Partition Index (NPI) (Section 3.2.3),

and a query execution algorithm, called the Neural Threshold Algorithm (NTA) (Section 3.2.4). NTA is a modified threshold algorithm. In contrast to CTA, NTA does not require all activations of all inputs before it starts. It utilizes NPI to progressively access and perform DNN inference on only the inputs that are possibly in the top-k results. NTA overcomes the bottleneck of query execution, DNN inference, by reducing the number of inputs on which DNN inference is performed at query time, while guaranteeing the correctness of the top-k results and introducing only tolerable storage overhead. Moreover, we show the instance optimality of NTA (Section 3.2.5) and propose various additional optimizations (Sections 3.2.6 and 3.2.7).

3.2.3 Neural Partition Index (NPI)

A key goal of DeepEverest is to support queries that target arbitrary neuron groups. Existing partitioning or indexing methods (e.g., *K-D Tree, locality-sensitive hashing*) must know which neuron group will be queried ahead of time and construct data structures in that space. In contrast, DeepEverest constructs indexes for each neuron separately and therefore is able to answer queries for arbitrary neuron groups. The activations in a DNN can conceptually be represented by the `Artifact` relation introduced in Section 3.1. Recall that `neuronID` is the identifier for a neuron in the DNN and `inputID` is the identifier for an input in the dataset. DeepEverestconceptually builds an index on the search key `(neuronID, activation)` and supports queries that return the `inputID`s for a given `neuronID` and range of `activation` values. To avoid materializing activations, DeepEverest builds an index on `(neuronID, PID)` instead, where `PID` (`partitionID`) is the identifier of a range-partition over activation values for a neuron. DeepEverest builds equi-depth partitions instead of equi-width partitions because the activation values are usually highly skewed, and equi-depth partitions adapt better to skewed distributions. Partition 0 contains the largest activations. The index then supports efficient lookups for a given `(neuronID, PID)` combination. The index returns the set of `inputID`s whose activations for the given `neuronID` belong to the partition identified with `PID`. Moreover, the index also supports queries that return the `PID` for

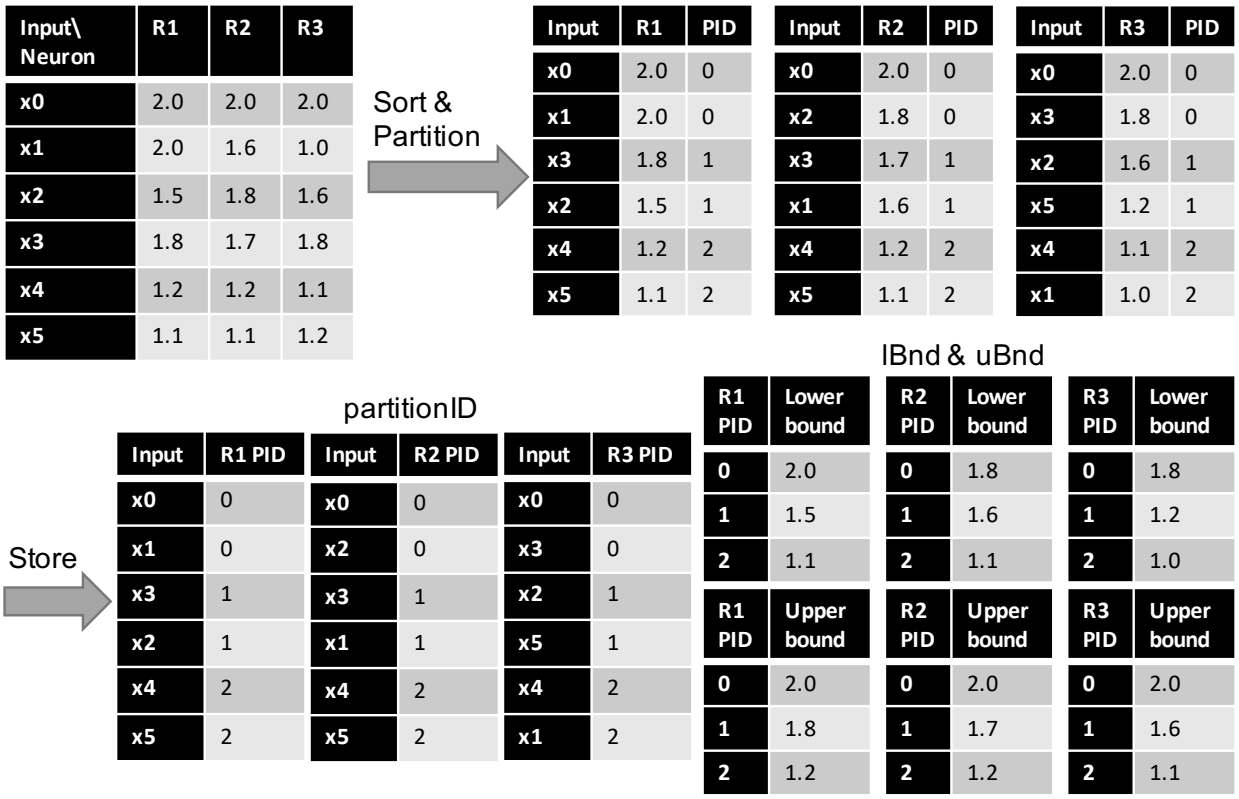


Figure 3.1: An example of building the Neural Partition Index of three neurons, $R1, R2, R3$, for six inputs, $x0, \ldots, x5$.

a given `(neuronID, inputID)` combination. We call this structure the Neural Partition Index (NPI) and denote the queries with `getInputIDs(neuronID, PID)` and `getPID(neuronID, inputID)`. Additionally, in NPI, for each partition, DeepEverest stores the *lower bound* and *upper bound* of the activations in that partition and supports queries that ask for them. We denote such queries with `lBnd(neuronID, PID)` and `uBnd(neuronID, PID)`. The number of partitions, $nPartitions$, is configurable and discussed further in 3.2.7.

There are two approaches to implementing the index. The first approach would be to maintain a set of buckets, each identified with a unique `(neuronID, PID)` combination as the key, and, for each bucket, maintain a list of `inputID`s. The second approach, which DeepEverest uses, is to maintain a list of `(neuronID, inputID)` pairs as keys, and, for

each entry, store the `PID`. `neuronID`, `inputID`, and `PID` are integers, so rather than building a B-tree or a hash index over the keys, we create an optimized index structure using an array where `neuronID` and `inputID` act as offsets for lookups in the array. This enables DeepEverest to only store the values and therefore avoid the cost of storing the keys. Figure 3.1 illustrates NPI for an example dataset with three partitions.

During preprocessing, DeepEverest runs DNN inference on all inputs once to build NPI for every neuron, which requires sorting. The time complexity to compute NPI once the activations are computed is $O(nNeurons \cdot nInputs \cdot log_2(nInputs))$, but the main source of overhead is DNN inference, which is a cost proportional to the number of inputs. The method that DeepEverest adopts is more space-efficient than building an index over `(neuronID, PID)` pairs because it costs $nNeurons \cdot nInputs \cdot \log_2(nPartitions)$ bits rather than $nNeurons \cdot nInputs \cdot \log_2(nInputs)$ bits, where $nPartitions << nInputs$. NPI also has much smaller storage overhead compared to fully materializing all activation values. A `PID` takes less storage than an activation value because a `PID` only costs $log_2(nPartitions)$ bits, while an activation value is usually a 32-bit floating point. For example, if DeepEverest has 8 partitions for each neuron, representing a `PID` costs 3 bits, which is less than 10% of the storage cost of full materialization. Storing the lower and upper bounds costs $nNeurons \cdot nPartitions \cdot 2 \cdot 32$ bits, which is normally negligible compared to the cost of storing the `PID`s.

3.2.4 Neural Threshold Algorithm (NTA)

Notation. We denote with N the set of all neurons in the DNN and with D the input dataset. A neuron is denoted with $n \in N$ and an input with $x \in D$. x is the `inputID`. The user issues a query: `topk`$(s, G, k,$ DIST$)$, where $s \in D$ is the sample input (also known as the target input) of interest to the user. $G \subseteq N$ is a set of neurons from a single layer in N. k is the desired number of query results, and DIST is the distance function. This function computes the distance between the set of activations of s and x looking only at the neurons in G. Table 3.2 lists our frequently used notation.

dPar

PID\Neuron	R1	R2	R3
0	0.9	0.7	0.6
1	0.4	0.5	0.0
2	0.0	0.0	0.1

ord

c\Neuron	R1	R2	R3
0	2	2	1
1	1	1	2
2	0	0	0

Figure 3.2: *dPar* and *ord* for the execution of NTA for the example query that finds the most similar inputs to $x5$ based on the activations of $\{R1, R2, R3\}$. For neuron i, c is the index of the PIDs in $ord(i)$, e.g., when c=0, $ord(R1, c)$=2.

Table 3.2: Summary of frequently used notation in DeepEverest.

Symbol	Meaning
s	Sample input (or target input) from D
G	Group of neurons from N
`topk`$(s, G, k,$DIST$)$	*Top-k most-similar* query
DIST	Function to compute distances between inputs
g_i (or i when clear)	The i-th neuron in G
`act`(i, x)	Activation value of g_i for input x
$dist(s, x, G)$	Distance between s and x based on G
top	Set of top-k inputs that are closest to s
P_n	Set of partitions for neuron n
`getInputIDs`(n, p)	`inputID`s of a single partition $p \in P_n$
`sPID`(n)	`PID` to which s belongs for neuron n
`lBnd`(n, p)	Lower bound of a single partition $p \in P_n$
`uBnd`(n, p)	Upper bound of a single partition $p \in P_n$

NTA returns the set of top-k inputs that are closest to the target input when considering only the neurons in G. This set of top-k inputs can be defined as a set $top \subseteq D$ of k inputs. *top* is initially empty and is conceptually built incrementally by identifying and adding to *top* the next input that satisfies:

$$\arg \min_{x \in D \setminus top} \{dist(s, x, G)\} \tag{3.1}$$

toRun (c = 0)

Neuron	R1	R2	R3
toRun	{x4, x5}	{x4, x5}	{x2, x5}

toRun (c = 1)

Neuron	R1	R2	R3
toRun	{x2, x3}	{x1, x3}	{x1, x4}

(a) Build *toRun*

dist(s, x, G) (c = 0)

Input newly computed	x2	x4
dist	1.5	0.3

dist(s, x, G) (c = 1)

Input newly computed	x1	x3
dist	1.6	1.9

top (c = 0)

(x, dist(s, x, G))	(x4, 0.3)	(x2, 1.5)

top (c = 1)

(x, dist(s, x, G))	(x4, 0.3)	(x2, 1.5)

(b) Perform DNN inference to compute *dist* and update *top*

minBoundary, maxBoundary (c = 0)

Neuron	R1	R2	R3
(min, max)	(1.1, 1.2)	(1.1, 1.2)	(1.2, 1.6)

minBoundary, maxBoundary (c = 1)

Neuron	R1	R2	R3
(min, max)	(1.1, 1.8)	(1.1, 1.7)	(1.0, 1.6)

F, V (c = 0)

Neuron	R1	R2	R3
(F, V)	(∞, 1)	(∞, 1)	(1, 1)

F, V (c = 1)

Neuron	R1	R2	R3
(F, V)	(∞, 1)	(∞, 1)	(∞, 1)

minDist (c = 0)

Neuron	R1	R2	R3
minDist	0.1	0.1	0.0

1.5 > t = 0.1 + 0.1 + 0.0 = 0.2 (c = 0)

minDist (c = 1)

Neuron	R1	R2	R3
minDist	0.7	0.6	0.4

1.5 < t = 0.7 + 0.6 + 0.4 = 1.7 (c = 1)

(c) Check termination

Figure 3.3: Intermediate variables for the execution of NTA for the example query. The values when c=0 are shown on the left, and the values when c=1 are shown on the right.

We further denote with g_i (or i when clear) the i-th neuron in set G, and with $\texttt{act}(i, x)$ the activation of neuron g_i on input x. g_i is the `neuronID`. For each neuron $n \in N$, NPI includes the set of partitions, P_n. We denote a single partition for neuron n with $p \in P_n$, and p is the `PID`. We denote the lower and upper bounds of this partition $p \in P_n$ with $\texttt{lBnd}(n, p)$ and $\texttt{uBnd}(n, p)$.

NTA proceeds as follows,

Step 1: Load indexes. We assume that NPI is initially on disk. This step reads the NPI for the neurons $g_i \in G$ from disk. The index holds the set of partitions, their lower and upper bounds, and the `PID`s of each input for each $g_i \in G$.

Step 2: Compute target activations. For each $g_i \in G$, compute $\texttt{act}(i, s)$, the activation value for input s and neuron g_i by running DNN inference on input s. A single inference pass is sufficient to compute the activations for all neurons in G.

Step 3: Order partitions. This step computes the order by which the partitions are accessed by NTA for each neuron. Let $\texttt{sPID}(i)$ denote the `PID` to which s belongs for neuron g_i. For each neuron $g_i \in G$ and partition $p \in P_i$, compute $dPar(i, p)$ as,

$$dPar(i,p) = \begin{cases} 0, & p = \texttt{sPID}(i) \\ \texttt{lBnd}(i,p) - \texttt{act}(i,s), & p < \texttt{sPID}(i) \\ \texttt{act}(i,s) - \texttt{uBnd}(i,p), & p > \texttt{sPID}(i) \end{cases} \tag{3.2}$$

which is the distance between the target input's activation value for neuron g_i and the closest activation value in partition p. For each neuron g_i, sort the partitions in P_i on their $dPar(i, p)$ values in ascending order and put them in a list, denoted with $ord(i)$. Later steps will process the partitions in the order specified by $ord(i)$.

Example: To illustrate the first three steps, consider a query $\texttt{topk}(x5, \{R1, R2, R3\}, 2, l1\text{-}\texttt{distance})$ *that finds the top-2 most similar inputs to* $x5$ *based on the activations of* $\{R1, R2, R3\}$*, using the example dataset in Figure 3.1. In this example,* $s{=}x5$, $G{=}\{R1, R2, R3\}$. *Step 1 reads from disk PID, lBnd, and uBnd shown in Figure 3.1. Step 2 runs DNN inference to compute the activations for* $x5$, $(\texttt{act}(R1, x5), \texttt{act}(R2, x5), \texttt{act}(R3, x5)){=}(1.1, 1.1, 1.2)$. *Step 3 computes dPar and ord for* $\{R1, R2, R3\}$*, as shown in Figure 3.2.*

Step 4: Find top-k. This step runs the modified threshold algorithm. It starts with the partitions to which the target input belongs, and it expands its search from there. Unlike CTA, this step incrementally computes the activations for candidate inputs and does so in batches to get good GPU performance. This step proceeds as follows,

Starting with an index $c = 0$:

Step 4 (a): For each neuron g_i, maintain a set $toRun_i$ that contains the inputs whose activations should be computed. Access $ord(i, c)$ to get the partition that contains the next most similar inputs. Query NPI to get the `inputID`s that belong to this partition $ord(i, c)$ and add them to $toRun_i$, i.e., $toRun_i \leftarrow$`getInputIDs`$(i, ord(i, c))$.

Example: as shown in Figure 3.3a, when $c = 0$, $toRun_{R1,0} = \{x4, x5\}$ *because* $ord(R1, 0) = 2$.

Step 4 (b): For each neuron g_i, compute the activations for the inputs in $toRun_i$ (excluding those that have already been computed) by running DNN inference in batches. $toRun_i$ is cleared after DNN inference. Note that this inference step computes the activations for *all* neurons in the neuron group being queried. Compute the distance between each newly computed input x and the target input s as $dist(s, x, G) = \text{DIST}(|\texttt{act}(0, x) - \texttt{act}(0, s)|, \ldots, |\texttt{act}(|G| - 1, x) - \texttt{act}(|G| - 1, s)|)$. Update *top* if $dist(s, x, G)$ is one of the k-smallest NTA has seen so far, i.e., input x is one of the k-most similar inputs to the target input s seen so far. Ties are broken arbitrarily.

Example: as shown in Figure 3.3b, when c=0, *the activations of inputs* $x2, x4$ *are computed (*$x5$ *was computed in Step 1). The distances from* $x2$ *and* $x4$ *to* $x5$ *are* 1.5 *and* 0.3, *respectively.*

Step 4 (c): Maintain a range of seen activations for inputs from $toRun_i$ for each neuron g_i, which is the range of activations such that NTA has seen every input with an activation in the open interval of this range. It is possible that NTA has seen one or more inputs from other neurons' *toRun* sets with activations outside of this range. However, the open interval of this range denoted by $(minBoundary_i, maxBoundary_i)$ only contains the activations for the inputs that NTA is guaranteed to have seen.

Let $minDist_i$ be the shorter distance from the boundaries of this range to the target input for each neuron g_i: $minDist_i = \min\{F_i \cdot |minBoundary_i - \texttt{act}(i, s)|, V_i \cdot |maxBoundary_i - \texttt{act}(i, s)|\}$, where F_i is an indicator function that indicates whether NTA has seen the last partition (inputs with the lowest activations) of neuron g_i, and V_i is another indicator function that

indicates whether the 0-th partition (inputs with the highest activations) of neuron g_i has been seen. Specifically, F_i=∞ when the last partition of neuron g_i has been seen; F_i=1 otherwise. V_i=∞ when the first partition of neuron g_i has been seen; V_i=1 otherwise. Define the threshold to be,

$$t = \text{DIST}(minDist_0, minDist_1, \ldots, minDist_{|G|-1}) \tag{3.3}$$

The threshold, t, represents the smallest possible distance to s from any unseen input. Hence, the termination condition is,

$$\max_{x \in top} \{dist(s, x, G)\} \leq t \tag{3.4}$$

where $\max_{x \in top} \{dist(s, x, G)\}$ represents the maximum distance to the target input s in the current top-k result set. As soon as this inequality holds, halt and return *top* as the query results.

Example: as shown in Figure 3.3c, $minBoundary_i$, $maxBoundary_i$ and $minDist_i$ are maintained and calculated for $\{R1, R2, R3\}$. For example, when $c = 0$, $minBoundary_{R1} = 1.1$, $maxBoundary_{R1} = 1.2$. Since NTA has seen the last partition (2) and has not seen the first partition (0), $F_{R1} = \infty, V_{R1} = 1$. Therefore, $minDist_{R1} = |maxBoundary_{R1} - act_{R1,x5}| = |1.2-1.1| = 0.1$.

When $c = 0$, $t = 0.2 < 1.5 = \max_{x \in top} \{dist(s, x, G)\}$, so NTA does not halt. When $c = 1$, $t = 1.7 \geq 1.5 = \max_{x \in top} \{dist(s, x, G)\}$, so NTA halts and returns top as the query result. It is worth noting that the cost of DNN inference on $x0$ is not incurred because it is impossible for $x0$ to be one of the top-2 results.

Step 4 (d): Increment c by 1. Repeat Step 4 (a) - (d) until all partitions have been seen or the halting condition in Step 4 (c) is satisfied.

The pseudocode is shown in Algorithm 3.1 and Algorithm 3.2.

The key innovation of NTA compared to CTA is in its processing of the inputs partition-by-partition using NPI until the termination condition is met. NTA runs DNN inference on only the necessary partitions of inputs for it to be certain that it has the precise top-k results when it terminates. This approach significantly reduces the number of inputs on which DNN

Algorithm 3.1 The Neural Threshold Algorithm for *top-k most-similar* queries.

function ANSWERQUERY($model, D, s, G, k,$ DIST) ▷*model: the DNN, D: dataset, s: sample image, G: neuron group, k: number of results to return,* DIST*: function to compute distances between inputs*

$layer \leftarrow$ GETLAYER(G)

$P, \texttt{lBnd}, \texttt{uBnd} \leftarrow$ LOADINDEX($layer$) ▷*Load indexes*

for all $g_i \in G$ **do** ▷P_i *contains the partitions for neuron* g_i

$P_i \leftarrow$ GETPARTITIONS(P, g_i)

$sampleAct \leftarrow$ MODELINFERENCE($model, layer, s$) ▷*Compute the activations for* s *by DNN inference*

Initialize `act` to an empty map that contains the activations of the neuron group for accessed inputs

for all $g_i \in G$ **do**

$\texttt{act}(i, s) \leftarrow sampleAct(i)$

for all $g_i \in G$ **do**

$\texttt{sPID}(i) = P \twoheadrightarrow \texttt{getPID}(g_i, s)$

Initialize the list $dPar(i)$

for all $p \in P_i$ **do** ▷$dPar(i, p)$*: the distance from each partition* p *for neuron* g_i *to* s

if $p == \texttt{sPID}(i)$ **then** $dPar(i, p) \leftarrow 0$

else if $p < \texttt{sPID}(i)$ **then** $dPar(i, p) \leftarrow \texttt{lBnd}(i, p) - \texttt{act}(i, s)$

else $dPar(i, p) \leftarrow \texttt{act}(i, s) - \texttt{uBnd}(i, p)$

for all $g_i \in G$ **do** ▷$ord(i)$*: the order by which the partitions for neuron* g_i *are accessed*

$ord(i) \leftarrow$ ARGSORT($dPar(i)$)

for all $g_i \in G$ **do** ▷*Initialization of some variables*

$F_i \leftarrow 1, V_i \leftarrow 1$

$minBoundary_i \leftarrow \infty, maxBoundary_i \leftarrow -\infty$

$c \leftarrow 0, top \leftarrow \emptyset$ ▷*Starting with* $c = 0$*; top: current top-k result set*

$inputRun \leftarrow \{s\}$ ▷*inputRun: set of inputs that have been run for DNN inference*

NTA-Loop ▷*The main loop of the Neural Threshold Algorithm, detailed in Algorithm 3.2*

return *top*

inference is performed at query time compared to computing the activation values for all inputs at query time. It further improves query performance by utilizing batch processing on GPUs. Inputs that share a partition are sent to the DNN for inference all at once. NTA along with NPI also has much smaller storage overhead compared to fully materializing the

Algorithm 3.2 The main loop of the Neural Threshold Algorithm.

while True do
  for all $g_i \in G$ **do**
    if $ord(i, c)$ does not exist **then return** *top* ▷*Return if all partitions have been seen*
    $toRun_i \leftarrow P \twoheadrightarrow \texttt{getInputIDs}(i, ord(i, c))$
  if $exitFlag$ **then break**
  $toRunUnion \leftarrow \bigcup_{g_i \in G} toRun_i \setminus inputRun$
  $toRunAct \leftarrow$ MODELINFERENCE$(model, layer, toRunUnion)$ ▷*Run DNN inference in batches*
  for all $x \in toRunUnion$ **do**
    Initialize the list $diff$
    for all $g_i \in G$ **do**
      $\texttt{act}(i, x) \leftarrow toRunAct_{i,x}$
      $diff_i \leftarrow |\texttt{act}(i, x) - \texttt{act}(i, s)|$
    $dist(s, x, G) \leftarrow$ DIST$(diff)$ ▷*Compute the distance between x and s*
    if $|top| < k$ **or** $dist(s, x, G) <$ GETMAXDIST(top) **then**
      UPDATE$(top, x, dist(s, x, G))$ ▷*Update top if x is one of the k-most similar seen*
  for all $g_i \in G$ **do**
    for all $x \in toRun_i$ **do**
      $minBoundary_i \leftarrow$ MIN$(minBoundary_i, \texttt{act}(i, x))$
      $maxBoundary_i \leftarrow$ MAX$(maxBoundary_i, \texttt{act}(i, x))$
    if $ord(i, c + 1)$ does not exist **then** $F_i \leftarrow \infty$
    if $ord(i, c) == 0$ **then** $V_i \leftarrow \infty$
    $minDist_i \leftarrow$ MIN$(F_i \cdot |minBoundary_i - \texttt{act}(i, s)|, V_i \cdot |maxBoundary_i - \texttt{act}(i, s)|)$
  $t \leftarrow$ DIST$(minDist)$ ▷*Calculate the threshold t*
  if $|top| == k$ **and** GETMAXDIST$(top) \leq t$ **then break** ▷*Termination condition*
  $inputRun \leftarrow inputRun \cup toRunUnion$
  $c \leftarrow c + 1$

activations for all inputs.

3.2.5 Instance Optimality of NTA

In this section, we investigate the instance optimality of NTA, which corresponds to the optimality in every instance, rather than just the worst or average case. Fagin's paper [92] shows that CTA is instance optimal for finding the top-k items over all algorithms (excluding

those that make very lucky guesses) and over all legal databases. We build on that proof to show the same for NTA.

Following the definitions in [92], let $\mathbb{A}$ be a set of algorithms that answer our target queries, and $\mathbb{D}$ be a set of combinations of datasets and DNN models that are legal inputs. Let $cost(\mathcal{A}, \mathcal{D})$ be the number of inputs in the dataset of $\mathcal{D}$ accessed by $\mathcal{A} \in \mathbb{A}$ when running $\mathcal{A}$ on $\mathcal{D} \in \mathbb{D}$. An algorithm $\mathcal{B}$ is instance optimal over $\mathbb{A}$ and $\mathbb{D}$ if $\mathcal{B} \in \mathbb{A}$ and if for every $\mathcal{A} \in \mathbb{A}$ and every $\mathcal{D} \in \mathbb{D}$,

$$cost(\mathcal{B}, \mathcal{D}) = O(cost(\mathcal{A}, \mathcal{D})) \tag{3.5}$$

We show the following theorem.

Theorem 1. *NTA is instance optimal for finding the top-k inputs over all algorithms (excluding those that make very lucky guesses) and over all legal combinations of datasets and DNN models.*

The proof follows similarly to that of Theorem 6.1 in [92]. We bound the maximal number of inputs accessed by NTA with an additive constant over CTA's maximal sequential access depth.

Proof. For any $\mathcal{D}\in\mathbb{D}$, assume that we already have the relation of sorted absolute differences (denoted with `AbsDiff`) between the activations of all inputs and the activation of the target input. Assume that CTA halts at depth d_i for each neuron g_i when running on `AbsDiff`, i.e., d_i is the number of inputs seen via sequential accesses for neuron g_i. As in [92], it suffices to bound the maximal number of inputs accessed by NTA for any $g_i \in G$. Let $d = \max_i d_i$. For the duration of this proof, let x_j denote the input at depth j in `AbsDiff` for a neuron g_i that reaches depth d when running CTA. Let R be the partition size in NPI. We will show that NTA will have accessed $x_j (\forall 1 \leq j \leq d)$ and terminate in at most $d + 2R$ accesses.

Recall that we denote a single partition for neuron g_i with $p \in P_i$. p is the partition identifier, `PID`, and $\texttt{sPID}(i)$ is the `PID` of the target input s for neuron g_i. In NPI, for $g_i \in G$, we say that a partition $p \in P_i$ is an *above* partition if $p < \texttt{sPID}(i)$; a partition $p \in P_i$ is an *under*

partition if $p \geq \texttt{sPID}(i)$. We say that an input is an *above* input if it belongs to an *above* partition; an input is an *under* input if it belongs to an *under* partition. As in Equation (3.2), an *above* partition uses the input whose activation is the lower bound of the partition as its representative; an *under* partition uses the input whose activation is the upper bound as its representative (the only exception is that the partition $p = \texttt{sPID}(i)$ uses the target input, s, as its representative).

Without loss of generality, assume that x_d is an *above* input. Let a be the greatest $j<d$ where x_j is an *under* input. Let b be the least $j>d$ where x_j is an *under* input. When NTA has accessed $x_j(\forall 1 \leq j \leq d)$ and confirmed that x_b is more distant from s than x_d, it knows that it has the correct top-k results and can then terminate. This is equivalent to the termination condition in Section 3.2.4.

Case 1: When x_a is accessed by NTA after x_d, we will show at most R *above* inputs, $x_j(j>d)$, are accessed before x_a. Assume towards contradiction that more than R *above* inputs $x_j(j>d)$ are accessed before x_a. There would be a representative of an *above* partition, $x_{d'}$ where $d' \geq d$. Let $x_{a'}$ be the representative of the partition containing x_a. Thus, $a' \leq a < d \leq d' \implies a' < d'$. This is a contradiction because the *above* partition with $x_{d'}$ as its representative was chosen to be accessed earlier than the *under* partition with $x_{a'}$ as its representative, which implies $d' \leq a'$.

Since *under* partitions are accessed in order w.r.t. x_a, the number of *under* inputs, $x_j(j>d)$, accessed by NTA is also at most R, i.e., the *under* inputs co-located on the partition containing x_a.

Note that if x_a and x_b belong to the same partition, NTA will terminate after this partition. If x_a and x_b belong to different partitions, then x_b is the representative of its partition. After confirming that x_b is more distant from the target input s than x_d, NTA knows that it does not need to access any further partitions (including the partition containing x_b) and terminates. In either scenario, the number of *above* $x_j(j>d)$ accessed by NTA is at most R, and the number of *under* $x_j(j>d)$ accessed is also at most R. Hence, the total number of accesses made by NTA is at most $d + 2R$.

Case 2: When x_b is accessed by NTA before x_d, there can be at most R *under* inputs, $x_j(j{>}d)$, that are accessed before x_d. Furthermore, the number of *above* inputs, $x_j(j{>}d)$, accessed is at most R, i.e., the *above* inputs co-located on the partition containing x_d Hence, the total number of accesses made by NTA is at most $d + 2R$. The proof follows similarly to Case 1.

Case 3: When x_a, x_b are accessed by NTA in order w.r.t. x_d, x_b must be the representative for its partition. After NTA processes the partition containing x_d, it will recognize that x_b is more distant than x_d and terminate. Thus no *under* $x_j(j{>}d)$ will be explicitly accessed. The *above* partitions are accessed in order w.r.t. x_d, so the number of *above* inputs, $x_j(j{>}d)$, accessed is at most R, i.e., the inputs co-located on the partition containing x_d. Hence, the total number of accesses made by NTA is at most $d + R$.

Since the three cases above exhaust all possibilities, this proves that for each neuron in G, NTA will have accessed $x_j(\forall 1{\leq}j{\leq}d)$ and terminate in at most $d + 2R$ accesses. □

3.2.6 Incremental Indexing

As shown in Section 3.3, the DeepEverest approach described so far achieves excellent query execution times with low storage overhead. This approach, however, incurs a potentially high preprocessing cost, especially for large datasets and models. Before executing any query, DeepEverest needs to compute the activations for all neurons and all inputs by running DNN inference. It then needs to construct the indexes for all layers and persist them to disk.

To address this challenge, we propose building the indexes incrementally as queries execute so that preprocessing is performed *only for the layers users query.* With this approach, DeepEverest performs no preprocessing ahead of time. When the user submits a query, if the indexes of the queried layer are available on disk, DeepEverest proceeds as described in Sections 3.2.4 and 3.2.7; otherwise, DeepEverest computes the activations of the queried layer by running DNN inference on all inputs. While doing so, it computes the query answer and returns it to the user. DeepEverest then constructs the indexes for

the layer and persists them to disk. Note that DNN inference is performed starting from the first layer instead of a previously queried layer each time DeepEverest constructs the indexes for a layer because only the indexes of the queried layers are stored on disk. With this approach, the costs of index computation and persistence for each layer are incurred once the first time that layer is queried, and only if that layer is queried.

In Section 3.3.3, we show that DeepEverest with this incremental approach significantly outperforms other methods on multi-query workloads. While DeepEverest must do extra preprocessing to build and store its indexes compared with caching the activations directly, it accelerates significantly more queries because it is able to store the indexes for significantly more layers given a storage budget.

3.2.7 Optimizations

In this section, we present several important optimizations that further improve the performance of DeepEverest. The first optimization, described in Section 3.2.7, accelerates two common types of top-k queries. The second optimization, described in Section 3.2.7, automatically selects DeepEverest's configuration. The third optimization, described in Section 3.2.7, accelerates sequences of related queries, as may occur during data exploration.

Maximum Activation Index (MAI)

For a given target input and a layer in a DNN, the *maximally activated neurons* are those neurons in the layer for which the activation values for the target input are the highest. DNN interpretation often involves examining such maximally activated neurons [249, 284, 294, 53] because they respond to the input the most and have the greatest impact on the DNN output. A common set of *top-k most-similar* queries ask to find the top-k similar inputs to a target input based on a neuron group consisting of these maximally activated neurons.

To accelerate these *top-k most-similar* queries that target maximally activated neurons as well as *top-k highest* queries, we introduce a straightforward yet effective optimization. The idea is for DeepEverest to store, for each neuron, a fraction of the highest activation

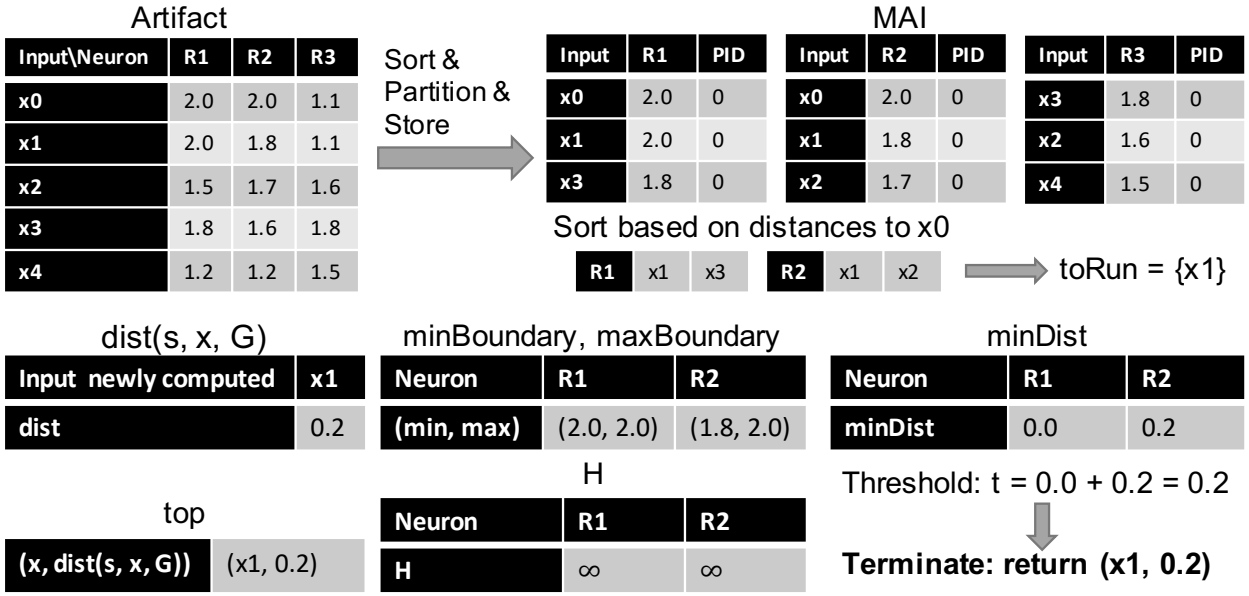


Figure 3.4: An example of constructing MAI ($ratio$=0.6) and query execution for `topk`$(x0, \{R1, R2, R3\}, 1, l1)$-`distance` ($batchSize$=1). Despite $x0$ only being in MAI for $R1$ and $R2$, DeepEverest leverages MAI to answer the query after only running DNN inference on $x0$ and $x1$.

values together with the corresponding `inputID`s. We call this data structure the Maximum Activation Index (MAI), and denote this fraction of `(activation, inputID)` pairs for a given `neuronID` with `MAI(neuronID)`. This fraction automatically becomes each neuron's 0-th partition. We denote the fraction of the inputs stored in MAI with *ratio*, which is a configurable parameter and discussed further in Section 3.2.7.

DeepEverest utilizes MAI during query execution to further reduce the number of inputs on which DNN inference is performed, if possible. DeepEverest now has more detailed knowledge of which inputs are most similar to the target input, rather than just the high-level knowledge that the inputs are in the same partition. We observe empirically that the activation values of the *maximally activated neurons* for an input are often likely to be in the top activations stored in MAI, and thus MAI is effective in improving the query time. DeepEverest modifies query execution described in Section 3.2.4 of partition 0 to

incorporate this information as follows: it first finds the neurons for which the target input is in MAI. For these neurons, DeepEverest sorts the other inputs in the 0-th partition by their distances to the target input s. Rather than performing DNN inference on all inputs in MAI for each neuron, DeepEverest builds a global *toRun* set by adding the most similar inputs from all of these neurons until the batch size is reached. Step 4(c) in Section 3.2.4 is modified to compute $minDist_i$ as $\min\{|minBoundary_i - act_{i,s}|, H_i \cdot |maxBoundary_i - act_{i,s}|\}$, where H_i is an indicator function that indicates whether NTA has seen the input with the highest activation in `MAI(i)`. $H_i{=}\infty$ when highest activation of neuron g_i has been seen; $H_i{=}1$ otherwise. The neurons g_i for which s is not in `MAI(i)` contribute 0 to the threshold calculation. Figure 3.4 illustrates an example.

Automatic Configuration Selection

Given a storage budget, DeepEverest must allocate it between NPI and MAI. It uses a heuristic algorithm to achieve this. A greater $nPartitions$ leads to smaller partitions, which is key to generally better performance (see Section 3.3.4) on queries that target any kind of neuron group (see Section 3.3.1), while a larger *ratio* accelerates only the two types of queries mentioned in Section 3.2.7. Hence, DeepEverest first picks a value for $nPartitions$ and then sets the value of *ratio*.

Intuitively when partitions are smaller, DNN inference is performed on fewer inputs not in the top-k because DeepEverest processes inputs partition-by-partition. However, if the partitions are smaller than the optimal batch size, DeepEverest will not leverage the full GPU parallelism.

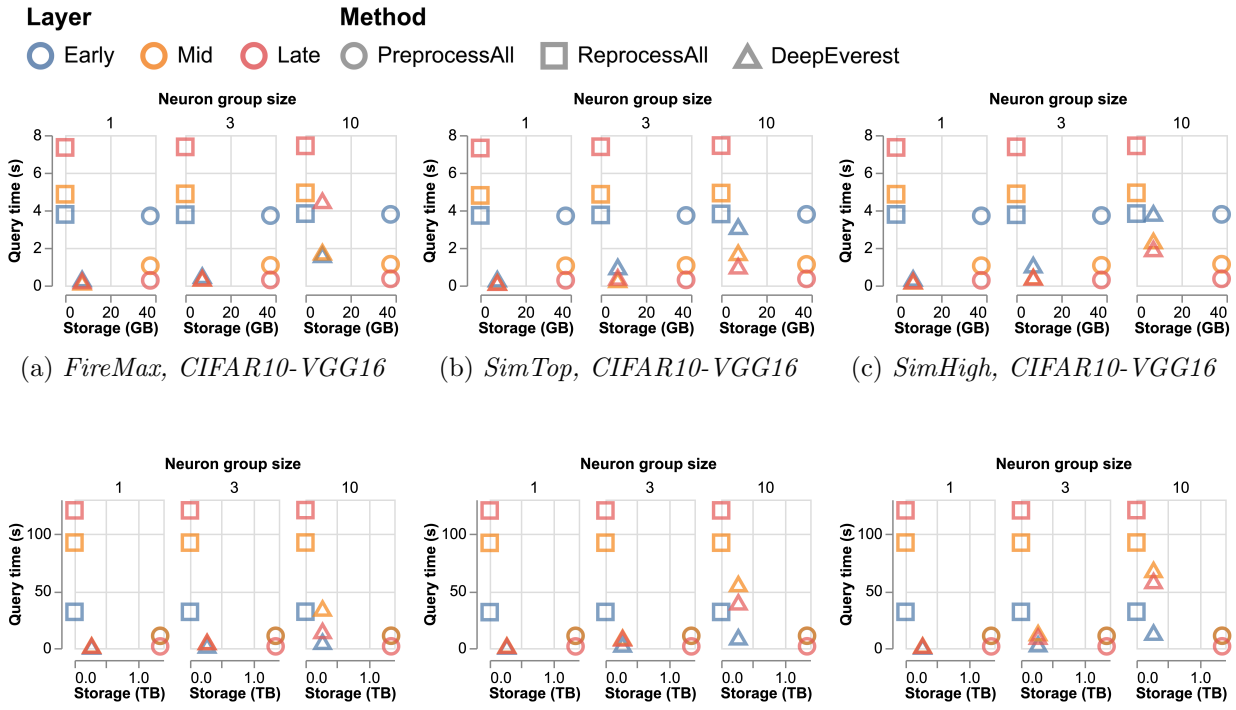
Given a storage budget, *budget* (in bytes), and a batch size, $batchSize$, DeepEverest sets $nPartitions$ to be the maximum power of two (to utilize all bits) that satisfies $nPartitions \leq nInputs/batchSize$ and $cost(nPartitions) < budget$. $cost(nPartitions)$ is the bytes consumed by storing NPI and is calculated as $nNeurons \cdot nInputs \cdot \log_2(nPartitions)/8$. $batchSize$ is set to the value that achieves the highest throughput for the DNN. Given the remaining storage budget, DeepEverest sets *ratio* to be the maximum value that

satisfies $cost(ratio) \leq budget - cost(nPartitions)$, where $cost(ratio)$ is the bytes consumed by storing MAI and calculated as $ratio \cdot nInputs \cdot nNeurons \cdot 4 \cdot 2$, since `activation` and `inputID` are 4 bytes each. When there is no remaining storage budget after selecting $nPartitions$, DeepEverest sets $ratio$ to 0.

Inter-Query Acceleration (IQA)

Inter-Query Acceleration (IQA) is an optimization technique to accelerate sequences of related queries, as may occur during DNN interpretation. As an example, imagine that a user finds a misclassified image. The user may want to first see the maximally activated neurons in a layer for the image and then find images with similar maximally activated neurons. The user may then decide to change how many neurons they are looking at, e.g., go from the top-3 neurons to the top-4 neurons. These exploration queries can be related in different ways. For example, the neuron group of a query could overlap with a subset of the neurons from recent queries, or the user-specified sample input in a query could be one of the top-k results of recent queries. Queries that overlap in neurons present an opportunity for further optimization as activation values can be reused for related queries.

With IQA, DeepEverest leverages an in-memory cache that contains recently used activation values to reduce the number of inputs that it must run DNN inference on at query time. Note that this in-memory cache is different from the disk caches described in Section 3.2.1. During query execution, DeepEverest inserts the activation values of each input processed by NTA into the cache. Instead of caching only the activation values for the neuron group being queried, it caches the activations for *all* neurons in the queried layer. This enables DeepEverest to utilize the cache for future related queries that target a different group of neurons in the same layer. DeepEverest adopts a most recently used (MRU) replacement policy for the in-memory cache. This is because DeepEverest processes partitions in order from most similar to the target input to least similar, and it seeks to prioritize keeping the activations from the most similar partitions in the cache. We show in Section 3.3.6 that given a small in-memory cache budget, DeepEverest with IQA



(a) *FireMax, CIFAR10-VGG16* (b) *SimTop, CIFAR10-VGG16* (c) *SimHigh, CIFAR10-VGG16*

(d) *FireMax, ImageNet-ResNet50* (e) *SimTop, ImageNet-ResNet50* (f) *SimHigh, ImageNet-ResNet50*

Figure 3.5: End-to-end individual query times and storage sizes on *CIFAR10-VGG16* and *ImageNet-ResNet50*. *nPartitions* and *ratio* of DeepEverest are selected by our heuristic algorithm given a storage budget of 20% of full materialization.

achieves up to 8× faster query times than DeepEverest without it.

3.3 Evaluation

DeepEverest is implemented in Python, using C++ to build the indexes. We evaluate it against the baselines described in Section 3.2.1.

3.3.1 Evaluation Setup

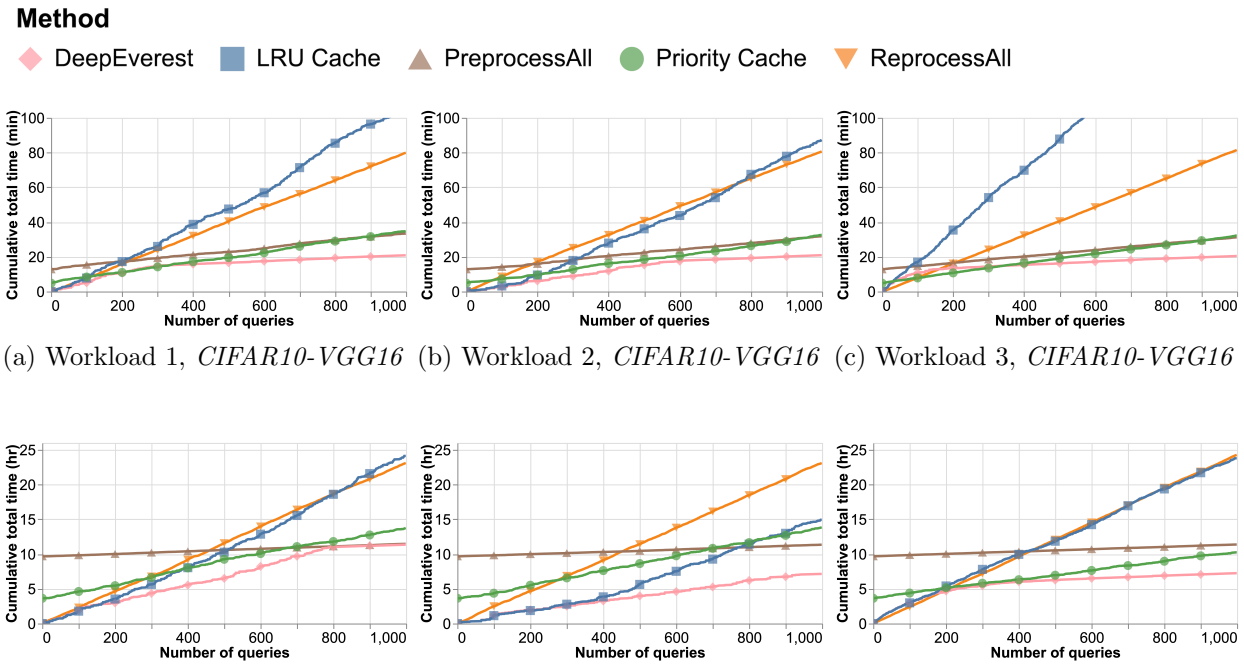
Datasets and models. We evaluate DeepEverest on two sets of well-known datasets and models. The first, called *CIFAR10-VGG16*, uses as inputs 10,000 images from the test

set of CIFAR10 [160], and uses a VGG16 network [245, 173]. The second, called *ImageNet-ResNet50*, uses as inputs 10,000 images from the validation set of ImageNet [233], and uses a ResNet50 model [119]. These two sets of models and datasets complement each other in terms of model and input size and DNN inference cost. In all experiments, we preload the entire input dataset into memory. We set $batchSize$ for each model as the value that achieves the highest inference throughput (128 for *CIFAR10-VGG16*; 64 for *ImageNet-ResNet50*).

Query generation. To generate queries, we consider 3 types of layers: *early*, *mid*, and *late*. For *CIFAR10-VGG16*, these correspond to activation layers 2, 7, and 13. For *ImageNet-ResNet50*, we use activation layers 2, 25, and 48. Given an input and a layer, we consider the following types of neuron groups: (a) *Top*: the maximally activated neurons for the given input in the layer; and (b) *RandHigh*: neurons randomly picked from the top half of non-zero neurons for the given input. We further consider *small*, *medium*, and *large* neuron groups consisting of 1, 3, and 10 neurons. Finally, based on the neuron groups, we use the following query types: (a) *FireMax*: *top-k highest* query; (b) *SimTop*: *top-k most-similar* query based on a *Top* neuron group; and (c) *SimHigh*: *top-k most-similar* query based on a *RandHigh* neuron group. We randomly select inputs from each dataset to generate *SimTop* and *SimHigh* queries.

In all experiments, we set k=20, which is a reasonable number of results for a user to inspect for a query. With a smaller k, we expect DeepEverest to achieve larger speedups because it will process fewer inputs and therefore return the results faster, while the query times of baselines will remain similar since they still need to recompute or load all the activations and maintain the query results. With a larger k, the overall speedups could degrade, but DeepEverest can incrementally return the top-k query results, as discussed in Section 3.4. Therefore, the perceived query time is still significantly improved. We use l_2-distance as the distance function. All numbers reported are median values of five queries on random inputs for each query configuration (e.g., query type: *FireMax*, neuron group size: 3, layer: *late*).

Machine configuration. All experiments are run on an AWS EC2 p2.xlarge instance, which



(a) Workload 1, *CIFAR10-VGG16* (b) Workload 2, *CIFAR10-VGG16* (c) Workload 3, *CIFAR10-VGG16*

(d) Workload 1, *ImageNet-ResNet50* (e) Workload 2, *ImageNet-ResNet50* (f) Workload 3, *ImageNet-ResNet50*

Figure 3.6: Cumulative total time (preprocessing time plus query execution time) for various multi-query workloads.

has an Intel Xeon E5-2686 v4 CPU running at 2.3 GHz, with 61 GB of RAM, an NVIDIA K80 GPU with 12 GB of GPU memory, and EBS gp3 volumes for disk storage.

3.3.2 Fundamental Space-Time Tradeoff

We first evaluate the fundamental tradeoff that DeepEverest achieves in terms of storage space and query execution time for individual queries. In this experiment, the only optimization DeepEverest uses is MAI described in Section 3.2.7. We first precompute and store the indexes for all layers before executing the benchmark queries. DeepEverest has a storage budget of 20% of *PreprocessAll*, and selects $nPartitions$ and $ratio$ using the algorithm described in Section 3.2.7. For *CIFAR10-VGG16*, $nPartitions = 64$, $ratio = 0.0046$; for *ImageNet-ResNet50*, $nPartitions = 64$, $ratio = 0.0074$. We compare DeepEverest against

PreprocessAll and *ReprocessAll*. Figure 3.5 shows the results.

As the figures show, *PreprocessAll* has the highest storage cost (37.8 GB for *CIFAR10-VGG16*, and 1.35 TB for *ImageNet-ResNet50*) since it stores all activations for every input. However, scanning the precomputed activation values generally leads to the fastest query times. The query times of *PreprocessAll* are slower for the early layer of *CIFAR10-VGG16* because it has a large number of neurons, and thus it takes longer to load all the activations. *ReprocessAll* has the lowest storage cost since it does not precompute or store anything ahead of time. Its query times are slow because of the DNN inference on the entire dataset at query time.

DeepEverest achieves the best of both worlds: low storage overhead and fast query times. For *CIFAR10-VGG16*, compared with *ReprocessAll* DeepEverest is 1.65× to 31.1× faster for *FireMax*, 1.22× to 50.8× faster for *SimTop*, and up to 31.5× faster for *SimHigh*. For *ImageNet-ResNet50*, compared with *ReprocessAll*, DeepEverest is 2.67× to 62.8× faster for *FireMax*, 1.65× to 63.5× faster for *SimTop*, and 1.35× to 63.1× faster for *SimHigh*.

Compared to *PreprocessAll* for both *CIFAR10-VGG16* and *ImageNet-ResNet50*, DeepEverest achieves comparable and sometimes even faster query times for queries that target small and medium-size neuron groups despite using only 20% of *PreprocessAll*'s storage overhead. For queries that target large neuron groups, DeepEverest's query times are slower. We observe this phenomenon again in Figure 3.9 (discussed in Section 3.3.4). Due to the curse of dimensionality, there is little difference in the distances between different pairs of inputs. As a result, DeepEverest is not able to reduce the number of inputs run by the DNN at query time as it does for small and medium neuron groups. Table 3.3 shows the number of inputs run by the DNN at query time to compute the activation values for *SimHigh* queries. We find that the number of inputs run by the DNN at query time for queries on larger neuron groups is higher than that of queries on smaller neuron groups.

3.3.3 Multi-Query Workloads

In this section, we evaluate DeepEverest on multi-query workloads using incremental indexing described in Section 3.2.6 that avoids start-up overhead. We construct various query workloads to represent possible DNN interpretation patterns and compare DeepEverest against other on-disk caching techniques. All workloads consist of 1,000 the most general *SimHigh* queries that target neuron groups of medium size. The first query of each workload targets a *RandHigh* neuron group from a randomly selected layer. Each later query has probabilities p_{same} of querying the same layer as the previous query, p_{prev} to query one of the previously queried layers (excluding the layer queried by the previous query), and p_{new} to query a layer that has not been queried yet. Workload 1 sets these to p_{same}=0.5, p_{prev}=0.3, p_{new}=0.2. Workload 2 sets these to p_{same}=0.5, p_{prev}=0.4, p_{new}=0.1. Workloads 1 and 2 are intended to simulate the exploration process of users that are likely to initially target layers they are interested in, and gradually explore more layers. Additionally, we construct Workload 3 in which queries are independent of each other; each layer is targeted uniformly at random by each query. This is not a realistic interpretation pattern but is meant to show the worst-case workload for DeepEverest.

We measure the cumulative total time, which includes the time for both preprocessing and query execution, and cumulative storage for each method. DeepEverest is given a storage budget of 20% of full materialization, and *nPartitions* and *ratio* are selected by our heuristic algorithm. *LRU Cache* and *Priority Cache* have the same 20% storage budget. The time to initially compute and store the data on disk is included with the 0-th query for *PreprocessAll* and *Priority Cache*. The results for cumulative total time are shown in Figure 3.6. We report the storage results in the text.

DeepEverest consistently performs the best for Workloads 1 and 2 using less than 20% of the storage of full materialization. We observe that after some number of queries, the cumulative total time of DeepEverest grows more slowly. For *CIFAR10-VGG16*, it plateaus after around 300 queries for Workload 1 and around 550 queries for Workload 2.

This indicates that DeepEverest has built and stored NPI and MAI for all layers in the DNN. All later queries are much faster because they benefit from these indexes and NTA. For *ImageNet-ResNet50*, DeepEverest completes building and storing the indexes for all layers after around 780 queries for Workload 1 and never completes for Workload 2, as we observe that DeepEverest's storage consumption is only 11.3% of full materialization after 1,000 queries. DeepEverest finishes building its indexes after fewer queries in Workload 1 than in Workload 2 since Workload 1 has a higher probability of querying new layers.

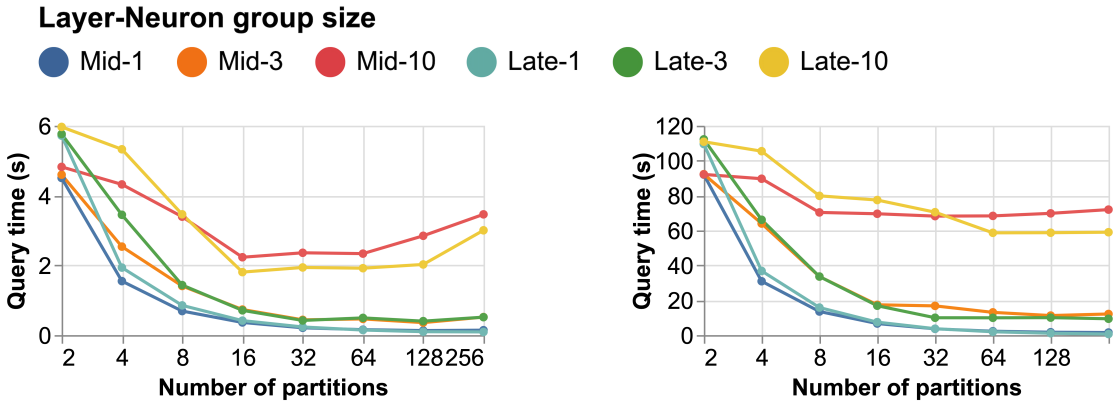
While DeepEverest has the fastest query times, its storage also grows more slowly than the baseline approaches (except for *ReprocessAll* which does not have any storage overhead). For both datasets and models, *PreprocessAll* uses full storage after its preprocessing step. Similarly, *Priority Cache* consumes its 20% storage budget after preprocessing. *LRU Cache* consumes its storage budget after around 50 to 200 queries. As discussed above, DeepEverest finishes building the indexes for all layers and consumes its storage budget after around 300 to 500 queries on *CIFAR10-VGG16*, and for *ImageNet-ResNet50* it fills its storage after 780 queries for Workload 1 and never does so for Workload 2.

For Workload 3, which is unlikely an interpretation pattern, DeepEverest is slightly worse than the best performing method for the first 200 to 300 queries on both datasets and models because during that time DeepEverest builds indexes for many new layers that have not been queried before. However, DeepEverest performs the best after 400 queries because more queries target previously seen layers and benefit from its indexes and NTA.

We further observe that users typically pause between queries. DeepEverest can use that time to compute and persist its indexes to disk, which would yield even better user-perceived query times.

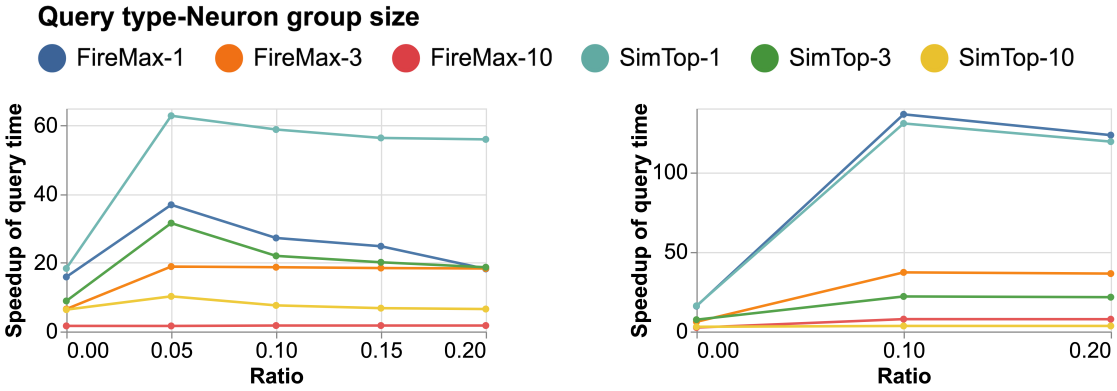
3.3.4 DeepEverest*'s Configuration Selection*

We now study the effectiveness of NPI and MAI, and the impact of DeepEverest's configurable parameters, as well as how DeepEverest performs using the configuration selection heuristic described in Section 3.2.7 with different storage budgets.



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 3.7: Query times of *SimHigh* queries when varying *nPartitions*. Note the log scale on the x-axis.



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 3.8: Speedups of query times (layer: *late*) against *ReprocessAll* when varying *ratio* with *nPartitions* set to 16.

Impact of Number of Partitions. We first examine how the number of partitions, *nPartitions*, affects the query times. We measure the query times of DeepEverest on *SimHigh* queries after building NPI with varying *nPartitions*. MAI is disabled for this experiment. The results are shown in Figure 3.7. We also measure the number of inputs on which DeepEverest performs DNN inference during query processing. Table 3.3 shows the

results for *CIFAR10-VGG16*. Similar trends are observed for *ImageNet-ResNet50*.

The query time initially decreases as $nPartitions$ increases. This is because when partitions are larger, inputs that do not contribute to the result end up being processed by DeepEverest. Hence, as partitions get smaller, the number of inputs run by the DNN at query time decreases. Then, after $nPartitions$ increases past a certain value (64 for *CIFAR10-VGG16*; 128 for *ImageNet-ResNet50*), the query time no longer decreases despite the number of inputs run by the DNN at query time continuing to decrease. Recall that NTA runs inference on all inputs in a partition as it processes that partition. When $nPartitions$ is so large that the partition size is below the optimal $batchSize$ for DNN inference, the parallelism of the GPU is not fully utilized, which causes some queries to slow down. Therefore, a good value of $nPartitions$ creates partitions whose sizes are similar to the optimal $batchSize$.

Effectiveness of MAI. This experiment evaluates the effectiveness of MAI. We measure the speedups of query times compared with *ReprocessAll* when varying $ratio$, which determines the fraction of inputs with activation values materialized in MAI. Recall that when MAI is non-empty, it becomes the 0-th partition. For this experiment, we set $nPartitions = 16$, which is a setting that performs well (see Figure 3.7). As discussed in Section 3.2.7, MAI is designed to accelerate *FireMax* and *SimTop* queries. We measure the speedups of such queries on neuron groups of different sizes.

Figure 3.8 shows the results. Note that when $ratio$=0, DeepEverest runs without MAI. The speedups of query times are generally much higher when $ratio$ is any non-zero value. This is because MAI enables DeepEverest to return the query results after processing a subset of the inputs from MAI (partition 0), rather than processing the entire partition. We also observe that the speedups of query times plateau or drop as $ratio$ further increases. This is because loading MAI from disk takes longer as $ratio$ increases. When a small index provides enough information for DeepEverest to find the top-k results after processing only some inputs from MAI, increasing $ratio$ degrades the speedups; the additional inputs in MAI do not improve the query times, and loading a larger index takes longer. The best value of $ratio$ in practice depends on the queries and the distributions of the activations of

Table 3.3: Number of inputs run by the DNN at query time for *SimHigh* queries on *CIFAR10-VGG16*.

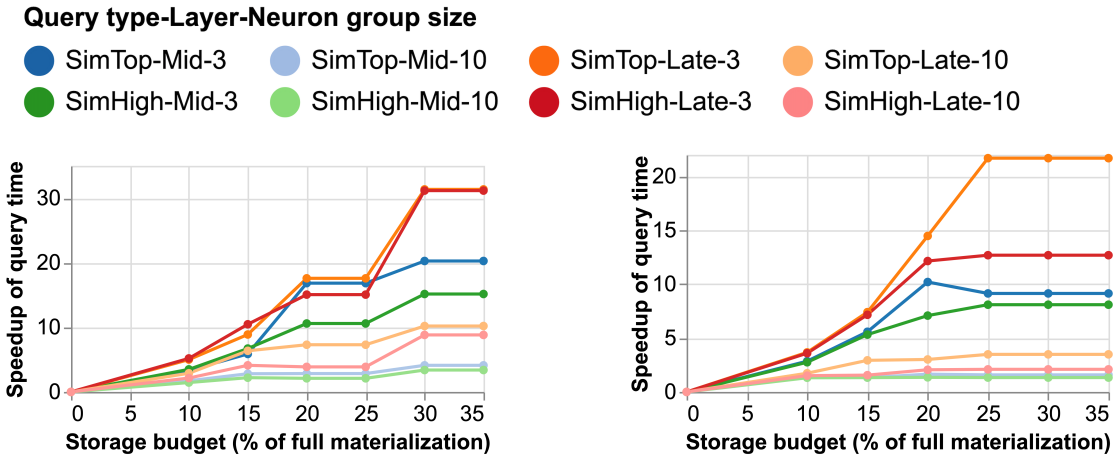
Layer-Neuron group size	Number of partitions						
	4	8	16	32	64	128	256
mid-1	3334	1429	667	323	159	79	40
mid-3	5462	2902	1441	736	727	390	390
mid-10	8941	6869	4339	4215	3515	3492	3316
late-1	3334	1429	667	323	159	79	40
late-3	5968	2372	1106	618	618	388	391
late-10	9008	5565	2870	2745	2227	1956	1919

the queried neuron group. Empirically, we observe that a small value of *ratio* (e.g., 0.05) is good for *FireMax* and *SimTop* queries on the two datasets and models.

Impact of Storage Budget. In previous sections, we examined the performance of DeepEverest with a storage budget of 20% of full materialization. Here we examine how well DeepEverest performs when the configuration selection algorithm has different storage budgets. We measure the speedups of query times compared with *ReprocessAll* for *SimTop* and *SimHigh* queries that target medium and large neuron groups, as shown in Figure 3.9. We observe that empirically DeepEverest delivers high speedups across different storage budgets, which also suggests that our configuration selection algorithm is robust. With a larger storage budget, DeepEverest performs better. We also observe that the speedups of queries on medium neuron groups are generally greater than the speedups for queries on large neuron groups due to the curse of dimensionality.

3.3.5 Preprocessing Costs

This experiment evaluates the costs of preprocessing in an extreme case where the user queries all layers in the DNN. Note that with incremental indexing, the costs of index computation and persistence for each layer are incurred once *only* if that layer is queried. In this experiment, DeepEverest is given a storage budget of 20% of full materialization and preprocesses all



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 3.9: Speedups against *ReprocessAll* by DeepEverest when given different storage budgets.

layers for each dataset and model from the first layer to the last layer. Convolutional layers, activation layers, and batch normalization layers are considered separate layers. We measure the cumulative times for each component in preprocessing: DNN inference, data persistence (for *PreprocessAll*, persisting the activations to disk; for DeepEverest, persisting NPI and MAI to disk), and index computation. We force-write the data to disk when measuring the time for data persistence. Figure 3.10 shows the results. DeepEverest has similar preprocessing times compared with *PreprocessAll*. The time for building NPI and MAI and persisting them to disk is similar to the time for *PreprocessAll* to persist the activations to disk. Note that DNN inference takes longer for late layers than for early layers. We also observe that data persistence and index computation for early layers takes longer than for late layers, since the sizes of early layers are usually greater than that of late layers. Considering these results along with the results shown in Section 3.3.2, DeepEverest achieves comparable and sometimes better query times than *PreprocessAll*, with only 20% of its storage overhead and similar preprocessing times.

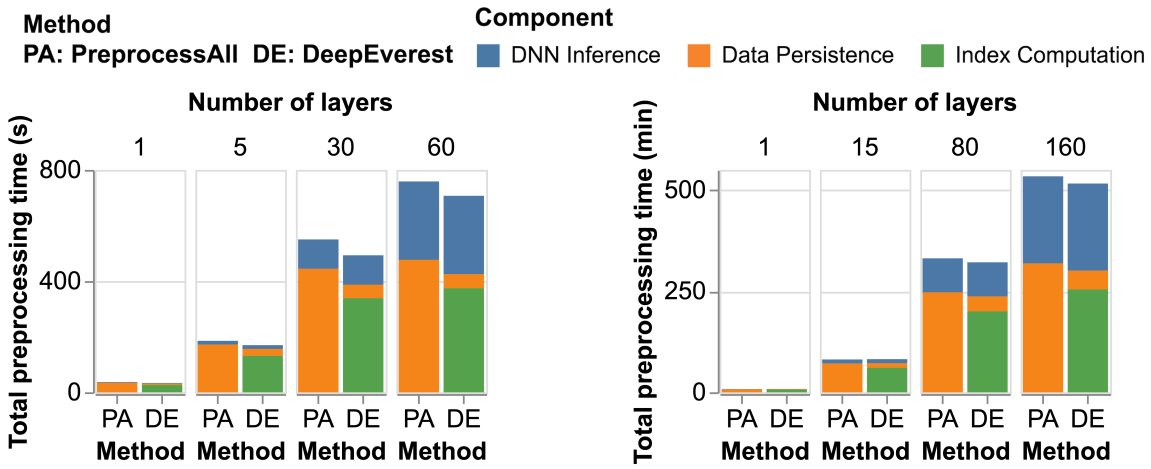
3.3.6 Effectiveness of IQA

Finally, we evaluate the effectiveness of Inter-Query Acceleration (IQA) for sequences of related queries. We randomly select five inputs and construct two sequences of related queries for each input on various layers. Both sequences consist of 1,000 *SimHigh* queries. The first query of each sequence targets a *RandHigh* neuron group containing n_{size} neurons. Each later query randomly replaces $n_{replace}$ neurons in the neuron group of the previous query by $n_{replace}$ randomly selected *RandHigh* neurons. Sequence 1 sets n_{size}=5 and $n_{replace}$=1. Sequence 2 sets n_{size}=10 and $n_{replace}$=2.

We measure the speedups for query times of DeepEverest with IQA against DeepEverest without IQA for each query. Figure 3.11 shows the median of the speedups for each query on *CIFAR10-VGG16* given an in-memory cache budget of 1 GB for IQA, with $nPartitions$=16 and $ratio$=0 set for DeepEverest. We observe that even with this small budget, IQA consistently improves DeepEverest's query times across different layers. Not shown in this chapter, we also experiment with different $nPartitions$ and $ratio$ and different cache budgets. We find that IQA always consistently speeds up related queries and larger budgets generally lead to larger speedups. In Figure 3.11, the speedups for the first query are around 1× since the in-memory cache is initially empty. For later queries when the cache is populated, for Sequence 1, DeepEverest with IQA achieves speedups of 2.65× to 8.73× for the late layer, 3.97× to 8.08× for the mid layer, and 1.53× to 3.38× for the early layer. For Sequence 2, DeepEverest with IQA achieves speedups of 4.00× to 8.06× for the late layer, 4.29× to 7.86× for the mid layer, and 1.48× to 1.82× for the early layer. The speedups for the early layer are smaller because this layer is larger, and hence the in-memory cache can hold fewer inputs' activations of the full layer. We observe larger speedups for the early layer with larger cache budgets.

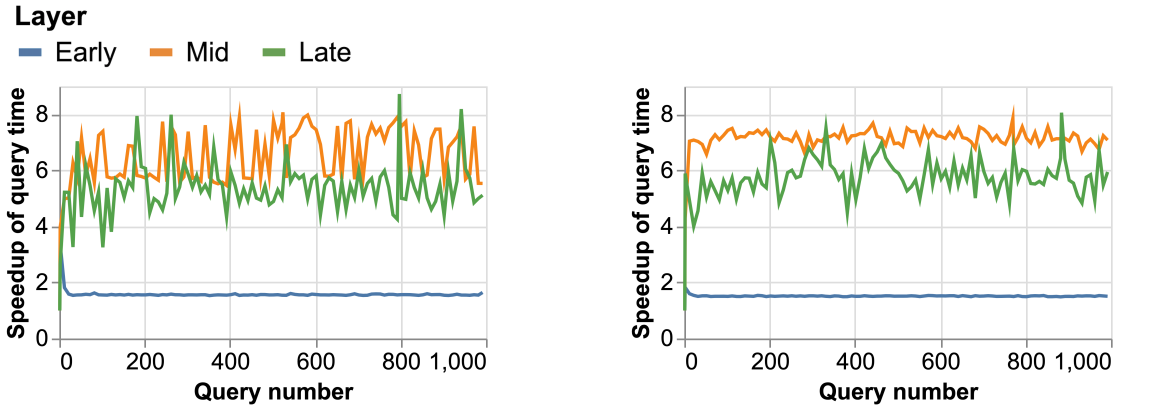
3.4 Discussion

This section discusses some possible optimizations and extensions for DeepEverest.



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 3.10: Cumulative preprocessing times from the first layer to the last layer for *PreprocessAll* and DeepEverest.



(a) Sequence 1 (b) Sequence 2

Figure 3.11: Speedups of query times by DeepEverest with IQA (1 GB cache budget) against DeepEverest without IQA.

Incrementally Returning Query Results. NTA runs until it has found k inputs whose distances to the sample s are at most the threshold value, t. However, NTA may be certain that some inputs are part of the top-k set before it has found the complete set. For queries

where $k > 1$, after each round of the algorithm, DeepEverest returns inputs in $Y \subseteq U$, where for all $y \in Y$, $dist(s, y, G) \leq t$, and continues running to find the rest of the $k - |Y|$ results. DeepEverest's optimizations enable it to incrementally return the top-k results quickly, and therefore reduces the time required to return the first part of the answer to the user.

Approximation. Modifying DeepEverest to give approximate results is straightforward. Following the definition of the θ-approximation in Fagin's paper [92], a θ-approximation (let $0 < \theta < 1$ be given) to the top-k answers is a collection of k inputs, U, (and their distances to the sample input) such that for each $y \in U$ and each $z \in D \setminus U$, $\theta * dist(s, y, G) \leq dist(s, z, G)$. Let t be the threshold value from Equation (3.3). DeepEverest can find a θ-approximation to the top-k answers by modifying the termination condition in Equation (3.4) to be,

$$\max_{x \in top} \{dist(s, x, G)\} \leq t/\theta \tag{3.6}$$

Early Stopping. DeepEverest can be further modified into an interactive process in which it can show the user the current top-k results with a guarantee about the degree of approximation to the correct top-k results. Based on this guarantee, the user can decide whether they would like to stop the process at any time. Let b be the largest distance to the sample input from the current top-k results, let t be the current threshold value, and let $\theta = t/b$. If the algorithm is stopped early, we have $0 < \theta < 1$ because $b > t$. Therefore, the current top-k results is then a θ-approximation to the correct top-k answers. Thus, the user can be shown the current top-k results and the number θ, with a guarantee that they are being shown a θ-approximation.

3.5 Summary

In this chapter, we presented DeepEverest, a system that accelerates top-k queries for DNN interpretation. DeepEverest, with various optimizations, reduces the number of activations computed at query time with low storage overhead, while guaranteeing correct query results. With less than 20% of the storage of full materialization, DeepEverest

accelerates individual queries by up to 63.5× and consistently outperforms other methods over various multi-query workloads.

Chapter 4

MASKSEARCH: QUERYING IMAGE MASKS AT SCALE

Many machine learning (ML) tasks over image databases commonly generate masks that annotate individual pixels in images. For instance, model explanation techniques [254, 248, 241, 292, 246] generate saliency maps to highlight the significance of individual pixels to a model's output. In image segmentation tasks [118, 151, 229], masks denote the probability of pixels being associated with a specific class or an instance. Depth estimation models [59, 208] yield masks reflecting the depth of each pixel, while human pose estimation models [71, 109] provide masks indicating the probability of pixels corresponding to body joints. Figure 1.4 shows some examples.

Exploring the properties of these masks unlocks a plethora of applications. For instance, in the context of model explanation, examining saliency maps is the most common approach to understanding whether a model is relying on spurious correlations in the input data, i.e., signals that deviate from domain knowledge [201, 215, 61, 274, 89, 192]. Other applications based on the properties of masks include identifying maliciously attacked examples using saliency maps [278, 270, 285], out-of-distribution detection also using saliency maps [124], monitoring model errors [31, 143, 16] using segmentation masks, traffic monitoring and retail analytics using segmentation masks [84, 83], and others.

The wide-ranging applications underscore an emerging necessity for AI practitioners: the capability to efficiently query and retrieve examples from image databases together with their masks, based on properties of the latter [215, 82, 151]. Today, AI practitioners lack a system that would support this task efficiently and at scale.

Consider the following two scenarios inspired by the literature:

Scenario 1 *(inspired by [278]): Bob is an engineer responsible for monitoring the*

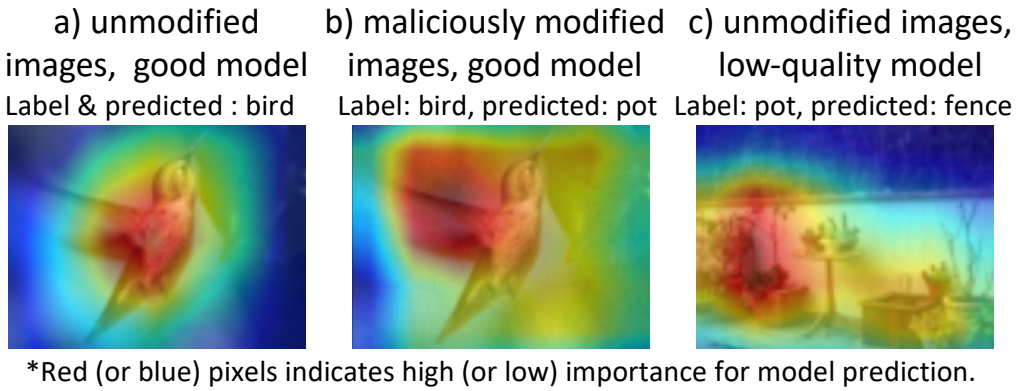


Figure 4.1: Example image masks: ImageNet [90] images overlaid with saliency maps. Saliency maps in columns b) and c) reveal that the models rely on irrelevant pixels to make predictions. Retrieving more examples with similar mask properties helps to better investigate the model's behavior.

performance of an image classification model. Recently, he notices a significant drop in the model's accuracy. To investigate, Bob examines the saliency maps for the misclassified images and finds that the high-value pixels are not concentrated on the foreground objects, but rather diffused across irrelevant background regions (see Figure 4.1). Suspecting these misclassifications may be due to malicious modifications that mislead the model to focus on irrelevant pixels, Bob wishes to identify and retrieve other masks (with the corresponding images) where high-value pixels are dispersed. This process often requires multiple iterations of querying and analyzing the returned examples, each time adjusting the regions of interest (where the high-value pixels are expected to be) and pixel value ranges (the range of high-value pixels) specified in the queries. Through this process, Bob could better understand the extent of the malicious modifications and work towards improving the model's resilience to such attacks.

Scenario 2 *(inspired by [89], detailed in Example 1.2.1): Alice, a scientist, developed a model to detect COVID-19 from chest X-rays, achieving high accuracy on training and validation sets. Despite this, the model often contradicts PCR test diagnoses in local hospitals.*

Investigating this discrepancy, Alice found that the model's saliency maps focused on peripheral markers, i.e., markers on the edges of the X-rays that do not contain any medical information, rather than the lungs, indicating it learned confounding factors. To identify other examples where the model relies on irrelevant pixels, Alice iteratively queries her dataset for masks with high-value pixels in the peripheral regions.

As the above examples illustrate, querying databases of masks is important in ML applications. Unfortunately, there is a lack of system support to efficiently execute these queries [123]. According to [215], to identify examples for which the model relies on spurious correlations, researchers have to manually examine the explanation maps for each image. This tedious approach is clearly untenable and calls for a system that efficiently supports mask-based queries.

In light of existing challenges, we propose MaskSearch, a system that efficiently retrieves examples based on mask properties. To build MaskSearch, we first formalize a novel, and broadly applicable, class of queries that retrieve images (and their masks) from image databases based on the properties of masks computed over those images. At the core of these queries are predicates on image masks that apply filters and aggregations (i.e., count of pixels) on the values of pixels within regions of interest (ROIs). We further extend the queries to support aggregations across masks and top-k computations to enhance the versatility of the supported queries. Aggregations across masks serve as a powerful tool for comparing trends of different masks, e.g., studying the difference between model saliency maps and human attention maps [82]. Top-k computations are also widely used. For example, Alice might be interested in finding the top-k X-rays whose saliency maps have the least number of high-value pixels in the lung regions.

Efficiently executing the formulated queries is challenging: The database of masks is too large to fit in memory; loading all masks from disk is slow and dominates the query execution time; compressing masks does not help due to the overhead of decompression. Existing methods do not support these queries efficiently. Using either NumPy or PostgreSQL to load and process the masks, a query that filters masks based on the number of pixels

within an ROI and a pixel value range takes more than 30 minutes to complete on ImageNet (Figure 4.6). Existing multi-dimensional indexing techniques also do not provide better execution times because masks are dense arrays. Array databases such as SciDB [67] and TileDB [205], though designed to process multi-dimensional dense arrays, are not optimized for efficiently searching through large collections of small arrays, as required in our target queries (Figure 4.6). While masks can be flattened as vectors and stored in vector databases, MaskSearch differs significantly because it targets a fundamentally different type of queries. A detailed discussion is provided in Section 4.1.2.

MaskSearch accelerates the aforementioned queries without any loss in query accuracy by introducing a new type of index and an efficient filter-verification query execution framework. Both techniques work in tandem to reduce the number of masks that must be loaded from disk during query execution while guaranteeing the correctness of the query result. The indexing technique, which we call the Cumulative Histogram Index (CHI), provides bounds on the pixel counts within an ROI and a pixel value range in a mask. It is designed to work with arbitrary ROIs (both mask-specific and constant) and pixel value ranges specified by the user at query time. These bounds are used during query execution when deciding whether a mask should be loaded from disk and processed while guaranteeing the correctness of the query result.

MaskSearch's query execution employs the idea of pre-filtering. Using pre-filtering techniques to avoid expensive computation or disk I/O has been explored and proven to be effective in many other problems, such as accelerating similarity joins [180, 135] and queries that contain ML models [142, 176, 42, 125] in cases where computing the similarity function or running model inference is expensive during query execution. MaskSearch's filter-verification execution framework leverages CHI to bypass the loading of the masks that are guaranteed to satisfy or not satisfy the query predicate. Only the masks that cannot be filtered out are loaded from disk and processed. By doing so, MaskSearch overcomes the limitation of existing systems by reducing the number of masks that must be loaded to process a query. Moreover, MaskSearch includes an incremental indexing approach that

avoids potentially high upfront indexing costs and enables it to operate in an online setting.

Contributions. In summary, the contributions of this chapter are:

- We formalize a novel, and broadly applicable, class of queries that retrieve images and their masks from image databases based on the properties of the latter, and further extend the queries to support aggregations across masks and top-k computations.
- We develop a novel indexing technique and an efficient filter-verification query execution framework.
- We implement the algorithms in a prototype system, MaskSearch, and demonstrate that it achieves up to two orders of magnitude speedup over existing methods for individual queries and consistently outperforms existing methods on various multi-query workloads that simulate dataset exploration and analysis processes.
- We develop a user interface for MaskSearch that allows users to interactively explore their datasets and models by issuing queries and visualizing the results.

Overall, MaskSearch is an important next step toward the seamless and rapid exploration of a dataset based on masks generated by AI models. It is an important component in a toolbox of methods for model explanation and debugging.

Organization. The remainder of this chapter is organized as follows. Section 4.1 provides background, formalizes the queries that MaskSearch supports, and discusses the challenges for their efficient execution. Section 4.2 presents the design of MaskSearch, including the indexing technique and the query execution framework. Experiments are presented in Section 4.3. Section 4.4 describes the user interface of MaskSearch. The chapter is summarized in Section 4.5.

4.1 Queries over Masks

This section formalizes the queries that MaskSearch supports and discusses the challenges associated with their efficient execution.

4.1.1 Data and Query Model

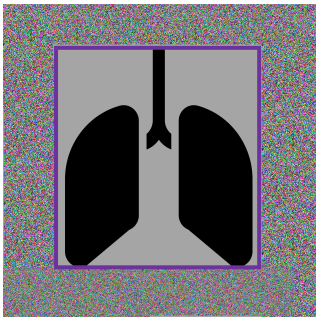
Data Model. An image is a 2D array of pixel values. A mask over an image is also a 2D array. The values in a mask, however, are limited to the range $[0, 1.0)$. Figure 4.2 shows an illustrative example of a toy x-ray image and an associated mask. The example shows a saliency map in which a higher value indicates that the pixel is more important to the model's decision. We can capture this data model with the following conceptual relational view,

```
MasksDatabaseView (
  mask_id INTEGER PRIMARY KEY,
  image_id INTEGER, -- Image from which the mask was derived
  model_id INTEGER, -- Model that generated the mask
  mask_type INTEGER, -- Type of mask (e.g., saliency map)
  mask FLOAT[][],
  ...);
```

where `mask_id`, `image_id`, and `model_id` store the unique identifiers of the mask, image, and model that generate the mask, respectively. `mask_type` is the identifier of the type of mask (an `ENUM` type), e.g., saliency map, human attention map, segmentation mask, depth mask, etc. The `mask` column stores the mask itself. Each mask is a 2D array of floating points in the range of $[0, 1)$. Additional columns can store other information, such as ground-truth labels, predicted labels, and image capture times. With some abuse of notation, an example tuple in the above view could be $(6, 4, \text{ResNet50}, \text{SaliencyMap}, [[0.9, 0.5, \ldots], \ldots])$, referring to a saliency map (mask #6) computed for image #4 using ResNet50 [119]. Note that `mask_id` does not have a direct relationship with `image_id` because an image can have multiple or no masks.

An ROI is a bounding box which can either be user-specified or computed by a query. Figure 4.2 shows a user-specified ROI that corresponds to the part of the image with the lungs. ROIs are query-dependent, so they are not included in `MasksDatabaseView`.

Basic Queries. MaskSearch supports queries that specify: (1) regions of interest within



(a) Example image

0.1	0.2	0.4	0.1	0.1
0.4	0.8	0.5	0.1	0.1
0.5	0.9	0.5	0.1	0.1
0.1	0.9	0.6	0.1	0.1
0.3	0.3	0.5	0.1	0.1

(b) Mask

Figure 4.2: A toy image motivated by [89] and its mask. The purple box is the ROI. Predicates on masks often involve counting the number of pixels in the ROI with values in a range, e.g., # pixels in the ROI with values in $(0.85, 1.0)$ is 2.

images (e.g., where the user expects the lungs to be located), (2) filter predicates over the pixel values in a mask (e.g., all pixel values above a threshold, indicating importance), and aggregates over those pixels that satisfy the predicates (i.e., count of pixels). A query over a mask can be expressed with the following query, where concepts like `CP(...)` will be explained in detail below,

```
SELECT *, CP(mask, roi, (lv, uv)) AS val
FROM MasksDatabaseView
WHERE <filter on CP(...)> [AND | OR] ...        -- optional
ORDER BY val [ASC | DESC] [LIMIT K]             -- optional
```

Region of interest (ROI). The ROI, `roi`, is a bounding box represented by pairs of coordinates that are the upper left and lower right corners of the box. It can be constant for all masks or different for each mask, e.g., the bounding box of the foreground object in each image computed by an off-the-shelf model. The ROI is specified by the user at query time or obtained from another table (e.g., a table containing bounding boxes) joined with `MasksDatabaseView`.

`CP` function. At the core of the query is the `CP` function. It stands for "Count Pixels".

It takes in a mask, an ROI, a lower bound (`lv`), and an upper bound (`uv`) as input, and returns the number of pixels in the ROI of the mask with values in the range of $[\texttt{lv}, \texttt{uv})$. `CP` is formally defined as follows,

$$\texttt{CP}(mask, roi, (lv, uv)) = \sum_{(x,y) \in roi} \mathbb{1}_{lv \leq mask[x][y] < uv}$$

where $\mathbb{1}_{\text{condition}}$ is an indicator function that is 1 if the condition is true and 0 otherwise. The output of `CP` is a scalar value and arithmetic operations can be applied to it. In our queries, `CP` is often present in the filter predicate, e.g., $\texttt{CP}(mask, roi, (lv, uv)) > T$, and in the `ORDER BY` clause, e.g., `ORDER BY` $\texttt{CP}(mask, roi, (lv, uv))$ `ASC`. Multiple `CP` functions can be used in a query, e.g., to specify multiple ROIs, or to compute multiple ratios of pixels in different ranges. The `CP` function is abstracted from the applications that motivated MaskSearch (examples from the beginning of the chapter and below) and is based on the observation that they can be expressed as predicates or aggregations together with predicates over pixel values and pixel counts.

Example 1*: Consider Scenario 2 from the beginning of the chapter. Alice, the scientist, is building a model that takes X-ray images as input and classifies them as COVID-19 vs. non-COVID. Her model does not work well once deployed. To investigate the problem, Alice wants to verify that the model is focusing its attention on the region in the images that corresponds to the lungs. Hence, she writes a query that computes the number of salient (i.e., important, e.g., with value* > 0.85*) pixels within the ROI that corresponds to the lungs, which she specifies manually as a bounding box,* `roi`[1]*. She retrieves all the images where the number of salient pixels is less than 10,000 by,*

```
SELECT image_id
FROM MasksDatabaseView
WHERE CP(mask, roi, (0.85, 1.0)) < 10000;
```

[1]For readability, we specify the ROI as the variable, `roi`. This would normally be a set of four numbers specifying the coordinates of the bounding box.

She can also compute the ratio of the number of salient pixels within the lung region to the total number of salient pixels in the image. She queries the top-25 images with the lowest ratios by,

```
SELECT image_id, CP(mask, roi, (0.85, 1.0)) / CP(mask, -, (0.85,1.0)) AS r
FROM MasksDatabaseView
ORDER BY r ASC LIMIT 25;
```

Complex Queries. MaskSearch further supports aggregations over pixel counts and pixel counts over aggregated masks. These more complete queries can be expressed with the following SQL,

```
SELECT [mask_id | image_id | model_id | ...],
       [SCALAR_AGG(CP(mask, roi, (lv, uv)))
       | CP(MASK_AGG(mask), roi, (lv, uv))] AS aggregate
FROM MasksDatabaseView
WHERE <filter on CP(...)> [AND | OR] ...       -- optional
GROUP BY [image_id | model_id | mask_type]     -- optional
HAVING <filter on aggregate> [AND | OR] ...    -- optional
ORDER BY aggregate [ASC | DESC] [LIMIT K]      -- optional
```

Scalar aggregation. The user can aggregate the outputs of `CP` functions for masks of the same image, model, or mask type, by defining the `SCALAR_AGG` function, which aggregates the outputs of `CP` functions. MaskSearch supports common functions such as `SUM`, `AVG`, `MIN`, and `MAX`, e.g., the average of multiple `CP` functions over masks produced by different models grouped by `image_id`.

Mask aggregation. `MASK_AGG` is used to aggregate masks themselves. It is a user-defined function that takes in a list of masks as input and returns a mask: `MASK_AGG` $\rightarrow$ `FLOAT[][]`. An example of `MASK_AGG` is `INTERSECT`$(m_1 > 0.8, ..., m_n > 0.8)$, i.e., the intersection of n masks after thresholding at 0.8.

Example 2*: Consider a case where our user in Scenario 2 from the beginning of the chapter, Alice, would like to understand if her model focuses on the same parts of the X-ray images*

as human experts. After setting **`roi`** *to the full mask, she can write the query below, where saliency maps have* **`mask_type`** $= 1$ *and human attention maps have* **`mask_type`** $= 2$*,*

```
SELECT image_id, CP(INTERSECT(mask > 0.7), -, (0.7, 1.0)) AS s
FROM MasksDatabaseView
WHERE mask_type IN (1, 2)
GROUP BY image_id
ORDER BY s DESC LIMIT 10;
```

4.1.2 Challenges

Processing the above queries efficiently is challenging. A baseline approach of loading masks from disk into memory before query processing is extremely slow because it saturates disk read bandwidth. A single query on ImageNet [90] takes more than 30 minutes to complete (Figure 4.6). Alternatively, storing compressed masks reduces data loaded from disk but moves the bottleneck to decompression, so a single query on ImageNet still takes around 30 minutes.

Existing systems, such as PostgreSQL, have the same bottleneck of loading masks from disk. Multi-dimensional indexing techniques do not efficiently support our target queries for two reasons: (1) they cannot handle mask-specific ROIs within a single query; (2) their complexity is high because mask data is dense. Assuming a constant ROI for all masks, these techniques require representing each mask's pixel as a point in the space of $(x, y,$ pixel value$)$. In this space, our query is an orthogonal range query followed by an aggregation by `mask_id`. The best known algorithm [73, 74], range trees, has a query time of $O(k + \log^2 n)$ and a preprocessing time of $O(n \log^2 n)$, with a space complexity greater than $O(n)$. Here, n is the number of total mask pixels in the dataset, and k is the number of pixels in the cuboid defined by `roi` and `(lv, uv)`. n is extremely large because mask data is dense (e.g., 65 billion for ImageNet), which makes using these indexes infeasible.

Vector databases, both functionally and practically do not support our target queries. Vector databases are designed for similarity searches, where an input vector's similarity to

stored vectors is calculated to find the closest matches. In contrast, MaskSearch targets queries that retrieve masks based on the number of pixels within ROIs and pixel value ranges, and optionally computes aggregations and top-k results. Storing the counts of pixels within ROIs and pixel value ranges as metadata in vector databases is impractical because the ROIs and pixel value ranges are specified at query time and can be arbitrary. Furthermore, vector databases often have dimensionality limits, such as 32,768 [33] in Milvus and 20,000 [32] in Pinecone, while a mask of 224×224 pixels has 50,176 dimensions, exceeding these limits.

Array databases [67, 205] are designed to work with dense arrays, but they are optimized for complex computations over small numbers of large arrays rather than efficiently searching through large numbers of arrays. While they can load specific slices within a desired ROI rather than entire arrays, MaskSearch avoids loading any pixels at all for a large fraction of masks, as we explain next.

4.2 MaskSearch

MaskSearch efficiently executes queries over a database of image masks while guaranteeing the correctness of query results. As presented above, the fundamental operations in our target queries involve filtering masks based on pixel values within ROIs, followed by performing optional aggregations, sorting, or top-k computations. The key challenge when performing these operations is that the database of masks is too large to fit in memory, and scanning, loading, and processing all masks is slow.

To accelerate such queries, MaskSearch introduces a novel type of index, called the Cumulative Histogram Index (CHI) (Section 4.2.1), and an efficient filter-verification query execution framework (Section 4.2.2). The CHI technique indexes each mask by maintaining pixel counts for key combinations of spatial regions and pixel values. CHI constructs a compact data structure that enables fast computation of upper and lower bounds on `CP` functions for arbitrary ROIs and pixel value ranges. These bounds are used during query execution to efficiently filter out masks that are either guaranteed to fail the query predicate or guaranteed to satisfy it without loading them from disk. The query execution framework comprises two

stages: the filter stage and the verification stage. During the filter stage, the framework utilizes CHI to compute bounds on `CP` functions to filter out the masks without loading them from disk. Then, during the verification stage, the framework verifies the remaining masks by loading them from disk and applying the full predicate. This framework guarantees the correctness of the query results and overcomes the bottleneck of query execution by significantly reducing the number of masks that must be loaded from disk.

4.2.1 Cumulative Histogram Index (CHI)

The key goals of CHI are to: (**G1**) support arbitrary query parameters lv and uv that specify the range of pixel values, which are unknown to MaskSearch ahead of time, and (**G2**) support arbitrary regions of interest, roi, and allow mask-specific rois in a single query. The rois are also unknown ahead of time because the user can specify rois arbitrarily at query time.

Key Idea. MaskSearch achieves both goals by building CHI to maintain pixel counts for different combinations of spatial locations and pixel values for each mask. Conceptually, MaskSearch builds an index on the search key $(mask\_id, roi, \text{pixel value})$. For each search key, CHI conceptually holds the number of pixels in the mask with the specified pixel value within the specified roi.

Building an index on every possible combination of $(mask\_id, roi, \text{pixel value})$ is infeasible both in terms of space and time complexity because the number of possible rois for each mask is quadratic in the number of pixels in the mask, let alone the number of masks and the number of possible pixel values.

Instead, CHI is a data structure that efficiently provides upper and lower bounds on predicates, rather than exact values. This approach leads to a small-sized index while still effectively pruning masks that are either guaranteed to fail the predicate or guaranteed to satisfy it. Only a small fraction of masks must then be loaded from disk and processed in full to verify the predicate.

CHI Details. CHI leverages two key ideas: discretization and cumulative counts. Discretiza-

tion reduces the total amount of information in the index, while cumulative counts yield highly efficient lookups. We explain both here.

To build a small-sized index, MaskSearch partitions masks into disjoint regions and discretizes pixel values into disjoint intervals. It then builds an index on the combinations of $(mask\_id, \text{region}, \text{pixel value interval})$, i.e., for each mask, it maintains pixel counts for combinations of partitioned spatial regions and pixel value intervals. Spatially, MaskSearch partitions each mask into a grid of cells, each of which is w_c by h_c pixels in size. Pixel value-wise, MaskSearch discretizes the values into b buckets (bins). MaskSearch could use either equi-width or equi-depth buckets. Our prototype uses equi-width buckets.

After discretization, there are several options for implementing the index. A straightforward option is to build an index on the search key $(mask\_id, cx\_id, cy\_id, bin\_id)$, where cx_id, cy_id, and bin_id identify the coordinates of each unique combination of grid and pixel-value range (e.g., cx_id of 3 corresponds to the grid cell that starts at pixel $w_c * 3$, similarly for cy_id and bin_id). For each such key, the index could store the number of pixels in the mask whose coordinates are in the cell identified by (cx_id, cy_id) and with values in the range $[p_{min} + bin\_id \cdot \Delta, p_{min} + (bin\_id + 1) \cdot \Delta)$, where p_{min} is the lowest pixel value across all masks and Δ is the width of each bucket. This option would require MaskSearch to identify all the cells that intersect with roi and all the bins that intersect with (lv, uv) and perform our query execution technique (discussed in Section 4.2.2) on the pixel counts of these cells and bins. A more efficient approach, which we adopt, is to build an index on the search key $(mask\_id, cx\_id, cy\_id, bin\_id)$, but, for each key, store the reverse cumulative sum of pixel counts in the mask with values in the range $[p_{min} + bin\_id \cdot \Delta, p_{max}]$ and coordinates in the region of $((1, 1), (cx\_id \cdot w_c, cy\_id \cdot h_c))$. This index is denoted with $H(mask\_id, cx\_id, cy\_id, bin\_id)$. We will also use $H(mask\_id, cx\_id, cy\_id)$ to denote the array of cumulative sums for all bins, i.e., $H(mask\_id, cx\_id, cy\_id)[bin\_id] = H(mask\_id, cx\_id, cy\_id, bin\_id)$. Recall that a $mask\_id$ uniquely identifies a $mask$. The index can be formally expressed as,

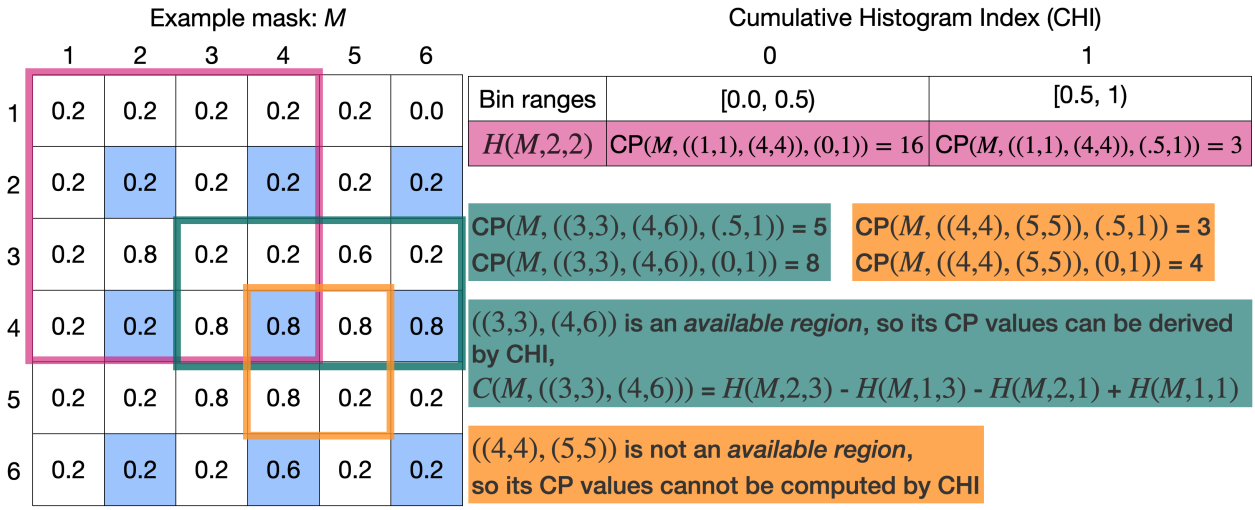


Figure 4.3: An example of CHI, CP, *available region*, and C.

$$H(mask\_id,cx\_id, cy\_id, bin\_id) = \\ \texttt{CP}(mask, ((1,1), (cx\_id \cdot w_c, cy\_id \cdot h_c)), (p_{min} + bin\_id \cdot \Delta, p_{max})) \tag{4.1}$$

***Example**: In Figure 4.3,* MaskSearch *builds CHI for an example mask, M, with $w_c = 2$, $h_c = 2$, and $b = 2$. Hence, each blue mask cell, (x_c, y_c), marks the corner of a discretized region. With $b = 2$, the pixel value range is discretized into 2 bins, $[0, 0.5)$ and $[0.5, 1)$.* MaskSearch *builds $H(M, x_c/w_c, y_c/h_c)$ for each blue mask cell (x_c, y_c). For example, the blue mask cell $(2, 2)$, corresponds to index entry $(x_c = 1, y_c = 1)$, and we have $H(M, 1, 1)[0] = 4$ (all four pixels are in (p_{min}, p_{max})) and $H(M, 1, 1)[1] = 0$ (no pixels are in the 0.5 to p_{max} range). For cell $(4, 4)$, $H(M, 2, 2) = [16, 3]$.*

In essence, $H(mask\_id, cx\_id, cy\_id, bin\_id)$ stores a cumulative sum of pixel counts, considering both spatial and pixel value dimensions. Storing cumulative sums offers greater efficiency compared to storing raw values, as it enables rapid evaluation of pixel counts within a specific range, in terms of both spatial and pixel value dimensions, by only performing simple arithmetic operations without having to access all the bins within the desired pixel value range for all the cells in the desired spatial region. To illustrate this, we first introduce

the concept of *available regions*.

Definition 4.2.1. *Let* X_c *denote* $\{x_c | x_c \in [w_c, 2w_c, 3w_c \ldots, w]\}$ *and* Y_c *denote* $\{y_c | y_c \in [h_c, 2h_c, 3h_c, \ldots, h]\}$. *A region* $((x_1, y_1), (x_2, y_2))$ *is available in the CHI of a mask if* $(x_2, y_2) \in X_c \times Y_c$ *and* $(x_1 - 1, y_1 - 1) \in (X_c \cup \{0\}) \times (Y_c \cup \{0\})$.

Example: *Available regions in Figure 4.3 are bounding boxes that start from the bottom-right corner of a blue cell*[2] *and end at the bottom-right corner of a blue cell, e.g.,* $((3,3),(4,6))$ *is an available region, highlighted with a dark green bounding box;* $((4,4),(5,5))$ *is not an available region, highlighted with an orange bounding box.*

Pixel counts within *available regions* are used to compute bounds on `CP` functions for arbitrary ROIs and pixel value ranges during query execution (Section 4.2.2). Before we get to these bounds, we explain how to compute pixel counts within an *available region* with pixel values within the range of two bin boundaries. MaskSearch performs two steps: (1) compute the reverse cumulative sums (C below) for the specified region by looking up the CHI entries (H); (2) calculate pixel counts between the two bin boundaries by subtracting the relevant cumulative sums. The details are explained below.

Let $C(mask\_id, r)$ denote the histogram of the reverse cumulative pixel counts of region r in mask $mask\_id$, where $C(mask\_id, r)[i] = \texttt{CP}(mask, r, (p_{min} + i\Delta, p_{max}))$. MaskSearch can compute $C(mask\_id, ((x_1, y_1), (x_2, y_2)))$ for any *available region* $((x_1, y_1), (x_2, y_2))$ (step (1) above). Let M denote $mask\_id$,

$$\begin{aligned} & C(M, ((x_1, y_1), (x_2, y_2))) \\ = \; & H(M, x_2/w_c, y_2/h_c) - H(M, (x_1 - 1)/w_c, y_2/h_c) \\ - \; & H(M, x_2/w_c, (y_1 - 1)/h_c) + H(M, (x_1 - 1)/w_c, (y_1 - 1)/h_c) \end{aligned} \tag{4.2}$$

where $-$ and $+$ are element-wise subtraction and addition, respectively, for two arrays of the same size. Equation (4.2) holds because $C(mask\_id, region)$ is a (finitely)-additive function over disjoint spatial partitions since each bin of $C(mask\_id, region)$ is a `CP` function which is

[2] $(0,0)$, not shown in the figure, is considered as a blue cell as well.

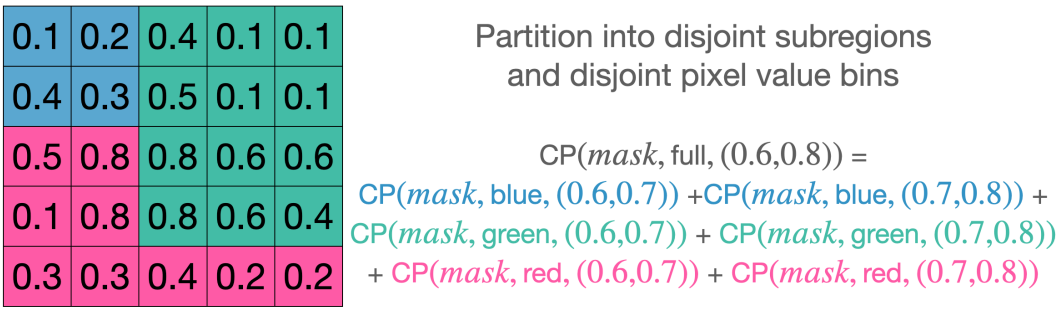


Figure 4.4: Illustration of CP being a (finitely)-additive function.

(finitely)-additive. Figure 4.4 shows an illustrative example of this additive property. Note that for any $mask\_id$ and roi, $C(mask\_id, roi)[\lceil p_{max}/\Delta \rceil]$ is always 0 for notation simplicity.

***Example**: Figure 4.3 shows how $C(M, ((3,3),(4,6)))$ is computed.*

After MaskSearch computes the reverse cumulative sums of pixel counts, C, for a region r, the pixel counts between any two bin boundaries (for pixel value discretization) can be obtained by subtracting the cumulative sums of the two bins (step (2) above).

Given a predicate $\texttt{CP}(mask, roi, (lv, uv)) > T$, MaskSearch uses CHI to check whether the predicate is satisfied. At a high level, MaskSearch identifies *available regions*, r_1 and r_2, in the CHI of the mask, such that r_1 is the smallest region that covers roi and r_2 is the largest region that is covered by roi. Then, MaskSearch computes $C(mask, r_1)$ and $C(mask, r_2)$ using Equation (4.2) and uses them to compute the lower and upper bounds of $\texttt{CP}(mask, roi, (lv, uv))$. Finally, MaskSearch checks whether $mask$ is guaranteed to satisfy or guaranteed to fail the predicate by comparing the lower and upper bounds with T. The details are further explained in Section 4.2.2.

Since $mask\_id$, cx_id, cy_id, and bin_id are all integers, rather than building a B-tree index or a hash index over the keys, we create an optimized index structure using an array where $mask\_id$, cx_id, cy_id, and bin_id act as offsets for lookups in the array. We call this structure the Cumulative Histogram Index (CHI) and $H(mask\_id)$ the CHI of mask $mask\_id$. There are several advantages of this optimized index structure. First, it enables MaskSearch to

only store the values of CHI and avoid the cost of storing the keys of CHI and the overhead of building a B-tree or hash index. Second, for any lookup key, the lookup latency is of constant complexity and avoids pointer chasing which is common in other index structures.

The time complexity for computing CHI for N masks of size $w \times h$ is $O(N \cdot w \cdot h)$, and this cost is amortized over time with the incremental indexing technique described in Section 4.2.6. The number of CHI that MaskSearch builds for N masks is $N \cdot w \cdot h/(w_c \cdot h_c)$. Each CHI has b bins, thus taking $4 \cdot b$ bytes. Hence, the set of CHI for N masks takes $4 \cdot b \cdot N \cdot w \cdot h/(w_c \cdot h_c)$ bytes in space. With a reasonable configuration of b, w_c, and h_c, CHI can be held in memory for a moderately-sized dataset, and MaskSearch can achieve good query performance with it (see Section 4.3.2).

4.2.2 Filter-Verification Query Execution

Without loss of generality, in this section, we will show how MaskSearch accelerates the execution of a one-sided filter predicate $\texttt{CP}(mask, roi, (lv, uv)) > T$, denoted with P, as multiple one-sided filter predicates can be combined to form a complex predicate. In Section 4.2.3, we will show that our technique applies to accelerating predicates that are in the form of $\texttt{CP}(...) < T$ or involve multiple different `CP` functions, e.g., `CP(...) < CP(...)`. Aggregations and top-k queries are discussed in Section 4.2.4 and Section 4.2.5, respectively.

MaskSearch takes as input a filter predicate P, and its goal is to find and return the *mask_id*s of the masks that satisfy P. At a high level, MaskSearch executes the following workflow:

- **Filter stage**: filter out the masks that *are guaranteed to fail* the predicate P, and add the masks that *are guaranteed to satisfy* P directly to the result set, before loading them from disk.
- **Verification stage**: load the remaining unfiltered masks from disk to memory and verify them by applying predicate P. If a mask satisfies P, add it to the result set.

It is worth noting that MaskSearch guarantees the correctness of the query results with

Table 4.1: Summary of frequently used notation in MaskSearch.

Symbol	Meaning
$\texttt{CP}(mask, r, (lv, uv))$	Count of pixels in region r of $mask$ with pixel values in range (lv, uv)
P	Predicate $\texttt{CP}(mask, roi, (lv, uv)) > T$
$mask\_id$	Identifier of $mask$
θ	Actual value of $\texttt{CP}(mask, roi, (lv, uv))$
$\bar{\theta}$	Upper bound of θ
$\underline{\theta}$	Lower bound of θ
p_{min}	Minimum pixel value
p_max	Maximum pixel value
Δ	Size of a pixel value bin
$C(mask\_id, r)$	Histogram of reverse cumulative pixel counts
$C(mask\_id, r)[i]$	$\texttt{CP}(mask, r, (p_{min} + i\Delta, p\_max))$
roi	Region of interest specified by the user
$\overrightarrow{roi}$	Smallest region *available* in CHI covering roi
$\underline{roi}$	Largest region *available* in CHI covered by roi

this two-stage execution framework.

Filter Stage

At a high level, the algorithm works as follows, for each mask, MaskSearch uses the CHI of the mask to compute bounds of $\texttt{CP}(mask, roi, (lv, uv))$, and it then uses the bounds to determine whether the mask will satisfy P or not. In this manner, MaskSearch reduces the number of masks loaded from disk during the verification stage (Section 4.2.2) by pruning the masks that are guaranteed to fail P and adding the masks that are guaranteed to satisfy P directly to the result set R. Deriving the bounds of $\texttt{CP}(mask, roi, (lv, uv))$ is challenging because roi and (lv, uv) can be arbitrary and not known in advance. MaskSearch addresses this challenge by leveraging the CHI of masks and the (finitely)-additive property of CHI to derive the bounds for arbitrary roi and (lv, uv).

Notation. P denotes $\texttt{CP}(mask, roi, (lv, uv)) > T$. $mask$ is uniquely identified by $mask\_id$.

θ denotes the actual value of $\texttt{CP}(mask, roi, (lv, uv))$. $\bar{\theta}$ and $\underline{\theta}$ denote the upper bound and the lower bound on θ computed by MaskSearch, respectively. $C(mask\_id, r)$ denotes the histogram of reverse cumulative pixel counts of the pixel value bins of region r in mask $mask\_id$, where $C(mask\_id, r)[i] = \texttt{CP}(mask, r, (p_{min} + i\Delta, p_{max}))$. When clear from context, we use $C(r)$ to denote $C(mask\_id, r)$. Frequently used notation is summarized in Table 4.1.

When a session of MaskSearch starts, the CHI of each mask is loaded from disk to memory and will be held in memory for the duration of the system run time. In cases where CHI cannot be held in memory, MaskSearch loads the CHI of a mask from disk on demand during query execution. Note that the size of the CHI of a mask is much smaller than the size of the mask itself, and therefore, even if the CHI of a mask is on disk, computing the bounds is much less expensive than loading the masks from disk to memory and evaluating the predicate P on them.

Given a predicate P, MaskSearch processes each mask targeted by the filter predicate in parallel. For each $mask$ uniquely identified by $mask\_id$, MaskSearch proceeds as follows:

Step 1: Compute $\bar{\theta}$ and $\underline{\theta}$. In this step, MaskSearch computes $\bar{\theta}$ and $\underline{\theta}$ by using the CHI of $mask\_id$. MaskSearch uses two approaches to computing two upper bounds, $\bar{\theta}_1$ and $\bar{\theta}_2$, on θ, and uses the smaller one as $\bar{\theta}$. The two approaches are effective in yielding bounds in different scenarios (details below).

Approach 1 first identifies the smallest *available region* (Definition 4.2.1) in the CHI that covers roi of $mask\_id$. For any $mask\_id$ and roi, an *available region* that covers roi always exists because the bounding box that covers the entire mask is always an *available region*. We denote this region with $\overrightarrow{roi}$. Then, $C(\overrightarrow{roi})$ (i.e., $C(mask\_id, \overrightarrow{roi})$) can be computed by CHI using Equation (4.2). Finally, $\bar{\theta}_1$ is computed as,

$$\bar{\theta}_1 = C(\overrightarrow{roi})[\lfloor lv/\Delta \rfloor] - C(\overrightarrow{roi})[\lceil uv/\Delta \rceil] \tag{4.3}$$

where $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the floor and ceiling of x, respectively.

Approach 2 first identifies the largest *available region* (Definition 4.2.1) covered by roi in

the CHI for each mask. We denote this region with $\underline{roi}$. Then, $C(\underline{roi})$ (i.e., $C(mask\_id, \underline{roi})$) can be computed using Equation (4.2). Finally, $\bar{\theta}_2$ is computed as,

$$\bar{\theta}_2 = \ C(\underline{roi})[\lfloor lv/\Delta \rfloor] - C(\underline{roi})[\lceil uv/\Delta \rceil] + |roi| - |\underline{roi}| \tag{4.4}$$

where $|\cdot|$ denotes the area of a region. Note that $\underline{roi}$ does not always exist in cases where w_c and h_c are large. In such cases, Approach 2 abstains from computing a bound and $\bar{\theta}_2$ is set to ∞.

Finally, $\bar{\theta}$ is computed by taking the minimum of $\bar{\theta}_1$ and $\bar{\theta}_2$. To show $\bar{\theta}$ is an upper bound of θ, we first show the following inequality. Because $(\lfloor lv/\Delta \rfloor * \Delta, \lceil uv/\Delta \rceil * \Delta)$ is a superset of (lv, uv), for any $mask\_id$ and roi, we have,

$$C(roi)[\lfloor lv/\Delta \rfloor] - C(roi)[\lceil uv/\Delta \rceil] \geq \theta \tag{4.5}$$

We now show the following theorem.

Theorem 2. *$\bar{\theta}$ is an upper bound of θ.*

We prove the theorem by showing both $\bar{\theta}_1 \geq \theta$ and $\bar{\theta}_2 \geq \theta$. For conciseness, we omit $mask\_id$ in $C(mask\_id, ...)$ and omit $mask$ in $\texttt{CP}(mask, ...)$ when clear from context, i.e., $C(Q)$ denotes $C(mask\_id, Q)$ and $\texttt{CP}(Q, (lv, uv))$ denotes $\texttt{CP}(mask, Q, (lv, uv))$. We also use $\texttt{CP}(Q \setminus W, (lv, uv))$ to denote the count of pixels in spatial region $Q \setminus W$ with pixel values in (lv, uv).

Proof. We first show $\bar{\theta}_1 \geq \theta$.

$$\bar{\theta}_1 = C(\overrightarrow{roi})[\lfloor lv/\Delta \rfloor] - C(\overrightarrow{roi})[\lceil uv/\Delta \rceil] \tag{4.6}$$

$$\geq \texttt{CP}(\overrightarrow{roi}, (lv, uv)) \tag{4.7}$$

$$= \texttt{CP}(roi, (lv, uv)) + \texttt{CP}(\overrightarrow{roi} \setminus roi, (lv, uv)) \tag{4.8}$$

$$\geq \texttt{CP}(roi, (lv, uv)) = \theta \tag{4.9}$$

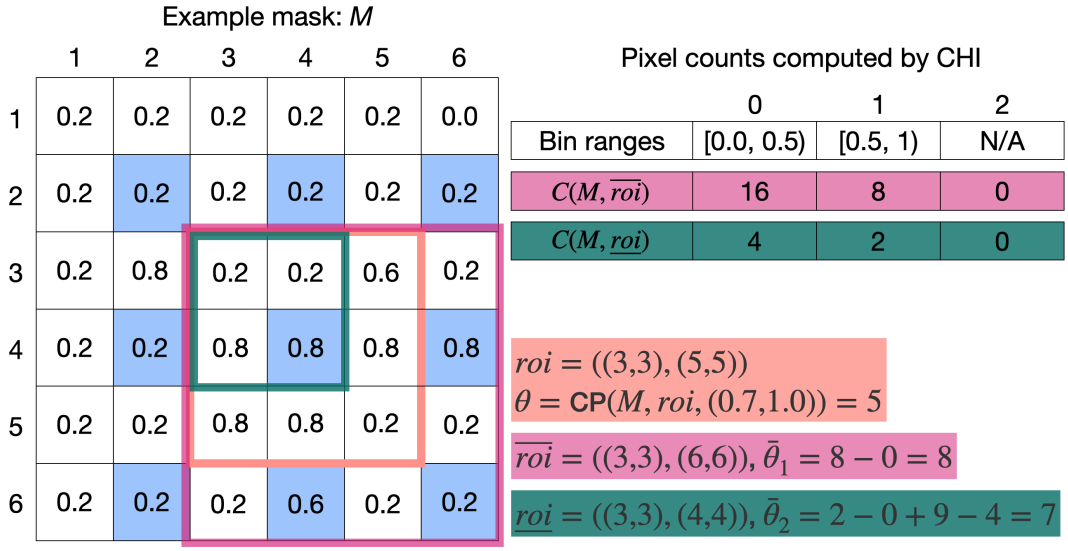


Figure 4.5: An example of MaskSearch computing the upper bounds, $\bar{\theta}_1$ and $\bar{\theta}_2$, given a mask, roi, and (lv, uv).

where Inequality (4.7) follows from Equation (4.5) and Equation (4.8) follows from CP is an additive function over disjoint spatial regions.

Let L denote $(\lfloor lv/\Delta \rfloor * \Delta, \lceil uv/\Delta \rceil * \Delta)$. We now show $\bar{\theta}_2 \geq \theta$.

$$\theta = \texttt{CP}(roi, (lv, uv)) \tag{4.10}$$

$$\leq \texttt{CP}(roi, L) \tag{4.11}$$

$$= \texttt{CP}(\underline{roi}, L) + \texttt{CP}(roi \setminus \underline{roi}, L) \tag{4.12}$$

$$\leq \texttt{CP}(\underline{roi}, L) + |roi| - |\underline{roi}| \tag{4.13}$$

$$= C(\underline{roi})[\lfloor lv/\Delta \rfloor] - C(\underline{roi})[\lceil uv/\Delta \rceil] + |roi| - |\underline{roi}| = \bar{\theta}_2 \tag{4.14}$$

where Equation (4.12) follows from the fact that CP is an additive function over disjoint spatial regions. Inequality (4.13) is because the count of pixels in any region with pixel values in any range is bounded by the total number of pixels in the region. □

***Example**: The two approaches are illustrated with an example mask in Figure 4.5. Mask data is the same as in Figure 4.3. The first approach identifies $\overline{roi}$, which is* $((3,3),(6,6))$,

and $C(M, \overrightarrow{roi})$ *is computed using Equation* (4.2)*. Then,* $\bar{\theta}_1$ *is computed using Equation* (4.3)*, i.e.,* $C(M, \overrightarrow{roi})[1] - C(M, \overrightarrow{roi})[2] = 8 - 0 = 8$*. The second approach identifies* $\underline{roi}$*, which is* $((3,3),(4,4))$*, and* $C(M, \underline{roi})$ *is computed using Equation* (4.2)*. Then,* $\bar{\theta}_2$ *is computed using Equation* (4.4)*, i.e.,* $C(M, \underline{roi})[1] - C(M, \underline{roi})[2] + |roi| - |\underline{roi}| = 2 - 0 + 9 - 4 = 7$*.*

The two approaches are effective in yielding bounds in different scenarios. Intuitively, the first approach is more effective when roi and $\overrightarrow{roi}$ are close to each other, which would result in a small difference between $\bar{\theta}_1$ and θ. The second approach is more effective when roi and $\underline{roi}$ are close to each other.

The lower bound, $\underline{\theta}$, can be computed similarly following the two approaches.

Step 2: Determine the relationship between $\bar{\theta}$ and $\underline{\theta}$ and T. In this step, MaskSearch determines whether the predicate P is satisfied by the mask based on the relationship between $\bar{\theta}$ and $\underline{\theta}$ and T. There are three cases:

- *Case 1:* $\bar{\theta} \leq T$. The mask is pruned because it is impossible for the mask to satisfy the predicate P.
- *Case 2:* $\underline{\theta} > T$. The mask is directly added to the result set R because the mask is guaranteed to satisfy the predicate P.
- *Case 3:* $\underline{\theta} \leq T < \bar{\theta}$. The mask is added to the candidate mask set S since it needs to be verified against P in the verification stage.

Verification Stage

The verification stage aims to verify each candidate mask in S that was neither pruned nor directly added to the result set. By loading it from disk and computing the actual value of θ, and then evaluating the predicate P, MaskSearch determines whether the mask satisfies the predicate P. If the mask satisfies the predicate P, it is added to the result set R.

4.2.3 Generic Predicates

So far, we have described how MaskSearch can efficiently process predicates in the form of $\texttt{CP}(mask, roi, (lv, uv)) > T$. Supporting predicates in the form of $\texttt{CP}(mask, roi, (lv, uv)) < T$ is similar to the previous case. The only difference is that in Step 2 of the filter stage, MaskSearch directly adds the mask to the result set R if $\bar{\theta} < T$ and prunes the mask if $\underline{\theta} \geq T$.

MaskSearch also supports generic predicates that involve multiple CP functions, i.e., $\texttt{CP}_1(...)\,\text{op}_1\,\texttt{CP}_2(...)\cdots\text{op}_{\text{n-1}}\,\texttt{CP}_n(...) > T$. Let $F = \texttt{CP}_1(...)\,\text{op}_1\cdots\text{op}_{\text{n-1}}\,\texttt{CP}_n(...)$. MaskSearch uses the lower and upper bounds of every CP function to derive the lower and upper bounds of F and use the bounds to efficiently prune the masks that are guaranteed to fail the predicate or guaranteed to satisfy it in the filter stage described in Section 4.2.2, as long as F is monotonic with respect to each $\texttt{CP}_i$ function. Common operators that make F monotonic include $+, -, \times$.

4.2.4 Aggregation

MaskSearch supports queries that contain scalar aggregates on CP functions or on the CP function over mask aggregations, as described in Section 4.1. For filter predicates on scalar aggregates, e.g., $\texttt{SUM}(\texttt{CP}(mask, roi, (lv, uv))) > T$ group by *image_id*, MaskSearch uses the same approach as in Section 4.2.3 to efficiently filter out groups of masks associated with the same *image_id* that are guaranteed to fail the predicate or guaranteed to satisfy it, since common scalar aggregate functions (SUM, AVG, and etc.) are monotonic with respect to the CP function. For filter predicates on mask aggregations, e.g., $\texttt{CP}(\texttt{MASK\_AGG}(mask), roi, (lv, uv)) > T$, MaskSearch treats the aggregated masks as new masks and uses the same approach described in Section 4.2.2 to process the query. The index for the aggregated masks is either built ahead of time or incrementally built (Section 4.2.6), which is a limitation of the current prototype. However, when the mask aggregation is monotonic, e.g., weighted sum, MaskSearch can be easily extended to support efficient filtering of the aggregated masks

Table 4.2: Summary of evaluated queries based on motivation and related work.

Query	Description
Q1	Returns masks s.t. $\texttt{CP}(mask, roi, (lv, uv)) > 5000$, $roi = ((50, 50), (200, 200))$, $(lv, uv) = (0.6, 1.0)$, $model\_id = 1$
Q2	Returns masks s.t. $\texttt{CP}(mask, roi, (lv, uv)) >$ 15,000, $roi =$ object, $(lv, uv) = (0.8, 1.0)$, $model\_id = 1$
Q3	Returns top-25 masks with largest $\texttt{CP}(mask, roi, (lv, uv))$, $roi = ((50, 50), (200, 200))$, $(lv, uv) = (0.8, 1.0)$, $model\_id = 1$
Q4	Returns top-25 images with largest $\texttt{mean}(\texttt{CP}(mask, roi, (lv, uv)))$ (groupby $image\_id$) for $mask$s associated with two models, $roi =$ object, $(lv, uv) = (0.8, 1.0)$
Q5	Returns top-25 images with largest $\texttt{CP}(\texttt{intersect}(mask), roi, (lv, uv))$ (groupby $image\_id$) for $mask$s associated with two models, $roi =$ object, $(lv, uv) = (0.8, 1.0)$

using indexes built for the individual masks.

4.2.5 Top-K

To answer top-k queries, MaskSearch follows a similar idea as described in Section 4.2.2, but it intertwines the filter and verification stages to maintain the current top-k result. Without loss of generality, let's consider the case of a top-K query seeking the masks with the highest values of the CP function. The set of top-k masks can be defined as a set, R, of k masks. R is initially empty and is conceptually built incrementally as the query is executed by identifying and adding to R the next mask, $mask$ (associated with its $\texttt{CP}(mask, roi, (lv, uv))$ value), that satisfies the following condition,

$$\texttt{CP}(mask, roi, (lv, uv)) > \min_{mask' \in R} \texttt{CP}(mask', roi, (lv, uv)) \tag{4.15}$$

MaskSearch sequentially processes the masks. For each mask, it computes the upper bound $\bar{\theta}$ and compares $\bar{\theta}$ with the CP values of the current R. If $\bar{\theta} \leq \min_{mask' \in R} \texttt{CP}(mask', roi, (lv, uv))$, the mask is pruned because it is impossible for the mask to be in the top-k result; otherwise, MaskSearch loads the mask from disk and computes the actual value of $\texttt{CP}(mask, roi, (lv, uv))$. It then updates R by adding the mask to R if it satisfies Inequality 4.15.

4.2.6 Incremental Indexing

As we show in Section 4.3.2 and Section 4.3.3, the vanilla MaskSearch system described so far achieves a significant query time improvement over the baselines with a small index size. The approach so far, however, incurs a potentially high overhead during preprocessing to build the index. Before processing any query, the vanilla approach must build the CHI for every mask in the database, which could lead to a long wait time for the user to get the first result.

To address this challenge, we propose building CHI incrementally as queries are executed so that only the masks that are necessary for the current query are indexed. Every time the user issues a query, as MaskSearch sequentially processes each mask as described in Section 4.2.2, it checks if the CHI of the mask is already built. If so, MaskSearch directly proceeds as described in Section 4.2.2. If not, MaskSearch executes the query by loading the masks from disk and evaluating whether they satisfy the query predicates. For each mask loaded from disk, MaskSearch then builds the CHI for the mask and keeps it in memory for future queries in the same session. When a MaskSearch session ends, the CHI for all the masks in the session is persisted to disk for future sessions. With this approach, the cost of building the CHI of a mask is incurred once the first time the mask is loaded from disk, and only if the mask is necessary for a query. In Section 4.3.5, we show that MaskSearch with such incremental indexing amortizes the cost of indexing quickly and significantly outperforms other baseline methods on multi-query workloads.

4.3 Evaluation

4.3.1 Experimental Setup

Implementation. MaskSearch is written in Python as a library and can work seamlessly with existing data management systems that stores and indexes the metadata of masks and images.

Dataset. We evaluate MaskSearch on two pairs of datasets and models. The first pair

of dataset and model, called *WILDS*, is from [154]. *WILDS* contains 22,275 images from the in-distribution and out-of-distribution validation sets of the iWildCam dataset [154]. For each image, we use GradCAM [241] to generate two saliency maps for two different ResNet50 [119] models obtained from [154]. Each saliency map is 448×448 pixels. The second, called *ImageNet*, contains 1,331,167 images from the ImageNet dataset [90]. We also use GradCAM [241] to generate two saliency maps for two different ResNet50 [119] models for each image, and use them as the masks for *ImageNet*. Each mask in *ImageNet* is 224×224 pixels. These two pairs of models and datasets complement each other in terms of the number of images (and masks) and the size of the masks.

MaskSearch configuration. Unless otherwise specified, we set b (the number of buckets for pixel value discretization) to 16 for both *WILDS* and *ImageNet*; then we set $w_c = h_c = 64$ (the cell size for spatial partitioning) for *WILDS* and $w_c = h_c = 28$ for *ImageNet*, such that the uncompressed index sizes for both datasets are around 5% of the compressed dataset sizes: the uncompressed index size is 6.5 GB for *ImageNet* and 88 MB for *WILDS*. The effect of more granular indexes is discussed in Section 4.3.4.

Baselines. As discussed earlier in the chapter, there is a lack of system support for the efficient processing of our targeted queries. To the best of our knowledge, no existing system reduces the work required, i.e., loading the masks from disk and computing the `CP` function values for them, to process a query. Thus, we compare MaskSearch to the following three baselines: (1) PostgreSQL 10. The masks are stored as 2D arrays of floating points as described in Section 4.1. The `CP` function is implemented as a user-defined function (UDF) written in C and compiled into a dynamically shared library. (2) TileDB 2.17.1 [205] with TileDB-Py 0.23.1. The masks are stored as a 3D array of floating points, with the first dimension being the mask ID, and the second and third dimensions being the height and width of the mask, respectively. The tile sizes for *WILDS* and *ImageNet* are set to 448×448 and 224×224, respectively because we found that these tile sizes provide the best performance for TileDB compared to other tile sizes. (3) NumPy 1.21.6. The masks are stored as NumPy arrays on disk. The `CP` function is implemented in Python and uses NumPy functions to

ensure vectorized computation.

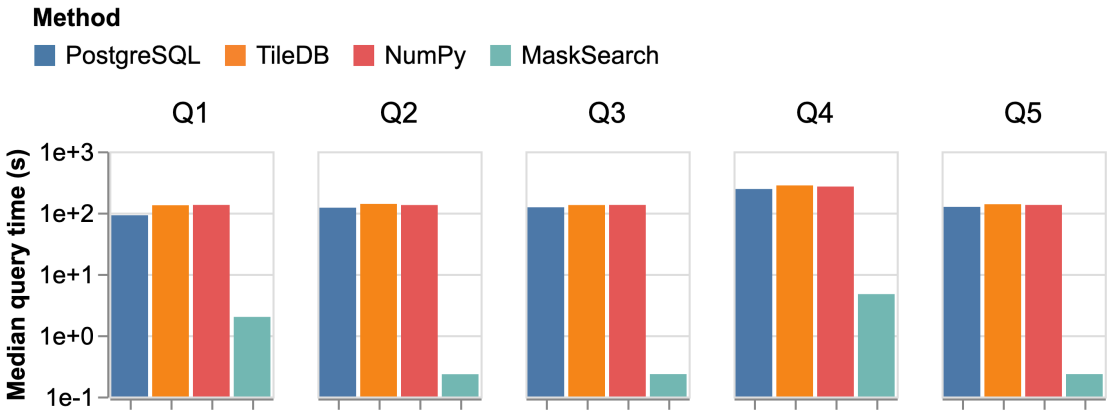
Machine configuration. All experiments were run on an AWS EC2 p3.2xlarge instance, which has an Intel Xeon E5-2686 v4 processor with 8 vCPUs and 61 GiB of memory, an NVIDIA Tesla V100 GPU with 16 GiB of memory, and EBS gp3 volumes provisioned with 3000 IOPS and 125 MiB/s throughput for disk storage. We evaluate MaskSearch on a single-node setup because most data scientists today work with a single machine [78]. Even in a multi-node setup, MaskSearch still reduces the number of masks loaded from disk (or over the network) and processed to answer a query, which is the dominant cost of query execution. The GPU was only used to compute the masks. All evaluated methods were using all vCPUs.

4.3.2 Individual Query Performance

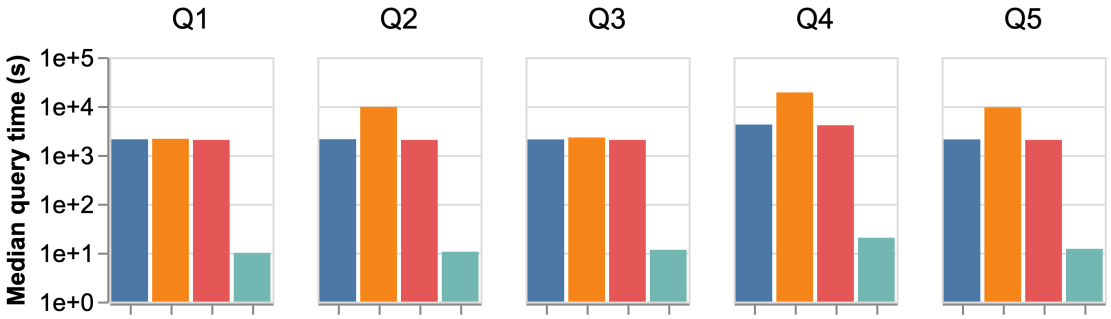
We first evaluate the performance of MaskSearch on 5 individual queries motivated by the examples and use cases from the beginning of this chapter and Section 4.1:

- Q1 (Filter, Scenario 2 from the beginning of this chapter): mask selection with a filter predicate on `CP` with a constant *roi* across all masks.
- Q2 (Filter, a variant of Q1): mask selection with a filter predicate on `CP` with different *roi*s for different masks.
- Q3 (Top-K, a variant of Example 1 in Section 4.1): top-k mask selection, ranked by `CP` with a constant *roi* across all masks.
- Q4 (Aggregation, a variant of Example 2 in Section 4.1): image selection with an aggregation over the `CP` values of masks associated with different models, with a filter predicate on the aggregated values.
- Q5 (Mask Aggregation, Example 2 in Section 4.1): image selection with a filter predicate on the `CP` value of the aggregated mask computed from the masks associated with different models.

The specific parameters for each query are shown in Table 4.2. k is set to 25 for top-k



(a) *WILDS*



(b) *ImageNet*

Figure 4.6: End-to-end individual query execution time based on motivation and related work. The index size for MaskSearch is $\sim 5\%$ of the original compressed dataset size for both datasets. Note the log scale on the y-axis.

queries because it is a reasonable number of masks to examine for a scientist. When *roi* is set to object, the *roi* is the bounding box of the foreground object in the image generated by YOLOv5 [136]. We build the CHI for all masks prior to executing the benchmark queries and clear the OS page cache before each query execution. The median execution time of 5 runs for each query is shown in Figure 4.6. In addition, Table 4.3 displays the number of masks loaded from disk by each system during query execution.

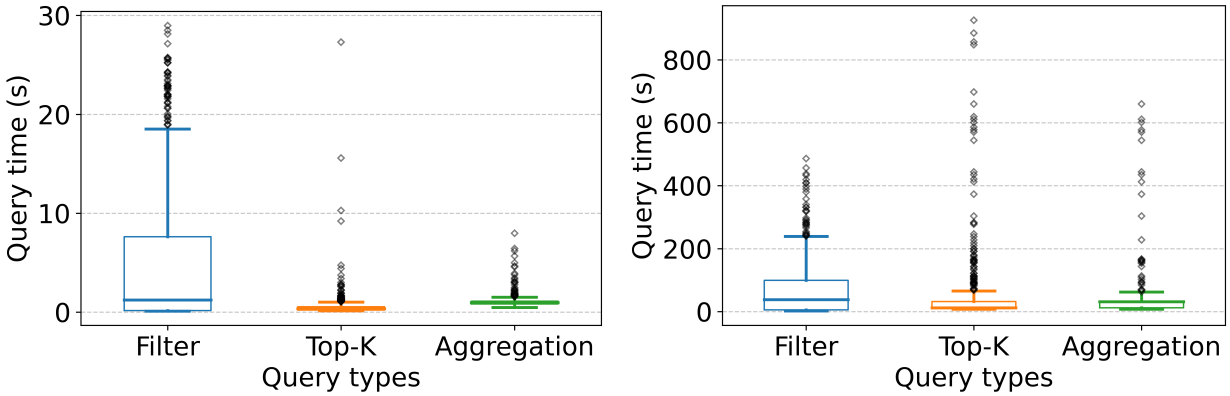
As Figure 4.6 shows, on *WILDS*, it takes PostgreSQL, TileDB, and NumPy around 2

minutes to answer each query; on *ImageNet*, it takes them more than 30 minutes to answer each query. Profiling these queries showed that mask-loading from disk dominates the query execution time. All baseline methods suffer from the same performance bottleneck: they all load all masks from disk and process them to generate the query results. Q4 notably takes more time than the other queries. This is because it demands the loading of two masks for every image due to its aggregation over the `CP` values of the masks. For Q2, Q4, and Q5 on *ImageNet*, TileDB is slower than the other two baselines. The reason is that TileDB has to sequentially load masks from the disk (instead of slicing the same ROI from multiple masks at once) because the ROIs in these queries are mask-specific. This results in suboptimal disk read bandwidth utilization. During the execution of all queries on PostgreSQL and NumPy and for the other queries on TileDB, we observed that the disk read bandwidth was fully utilized, reaching 125 MiB/s, the provisioned disk read bandwidth for our EBS volumes. This confirms that the query execution time is dominated by the time required to load the masks from disk. Therefore, any system that does not reduce the number of masks loaded from disk during execution can achieve, at best, a comparable query time to that of NumPy and PostgreSQL. And, while faster EBS volumes could enhance the baselines' performance, MaskSearch would still outperform them by reducing mask-loading during query execution.

MaskSearch executes each query in under 5 seconds on *WILDS* and in less than 20 seconds on *ImageNet*, providing query time speedups of up to two orders of magnitude over the baselines. This significant difference in performance is attributed to MaskSearch loading many fewer masks (shown in Table 4.3) because its filter-verification framework enables it to avoid loading from disk the masks that are guaranteed to satisfy the query predicate or guaranteed to fail it. On *ImageNet*, MaskSearch's query time for Q4 is longer compared to other queries, even though the number of masks loaded for Q4 is smaller. This discrepancy stems from the additional computation MaskSearch performs for Q4 (2× bound computation than other queries), as it contains an aggregation.

Table 4.3: Number of masks loaded during query execution. PG = PostgreSQL, TDB = TileDB, NP = NumPy.

Dataset	Method	Q1	Q2	Q3	Q4	Q5
WILDS	MaskSearch	407	40	32	874	48
	PG & TDB & NP	22,275	22,275	22,275	44,550	22,275
ImageNet	MaskSearch	2696	3849	2943	1494	2768
	PG & TDB & NP	1,331,167	1,331,167	1,331,167	2,662,334	1,331,167



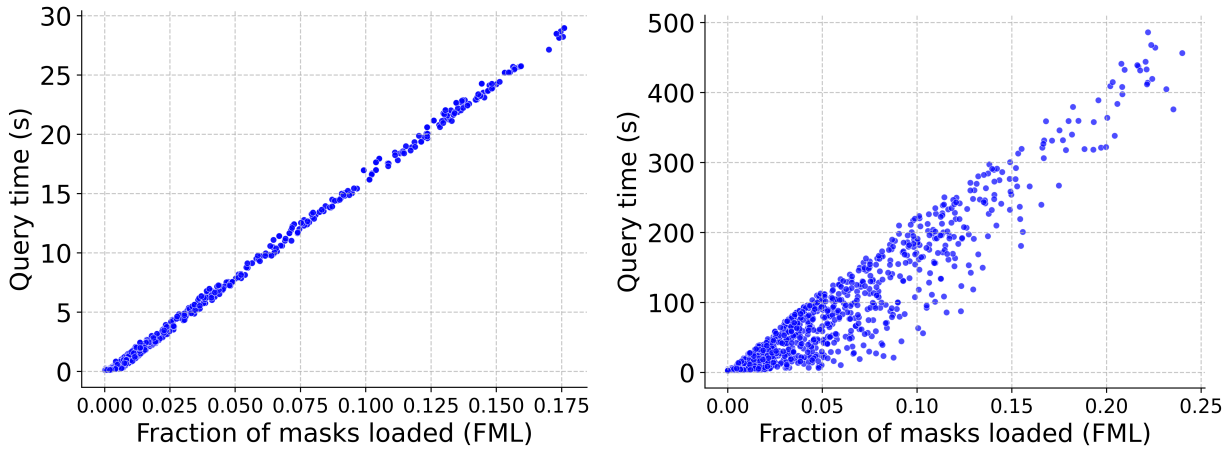
(a) *WILDS* (b) *ImageNet*

Figure 4.7: Query time of MaskSearch for different query types. Index size for MaskSearch: $\sim 5\%$ of dataset size.

4.3.3 Performance on Different Query Types

In this experiment, we evaluate the performance of MaskSearch on three types of queries with varying parameters. We only show the execution times of MaskSearch because, for each query type, baseline methods have similar execution times as the queries of the same type in Section 4.3.2, regardless of specific query parameters. For each dataset and query type, we generate 500 queries with randomized parameters and execute them using MaskSearch:

- **Filter**: this query type contains mask selection queries with a filter predicate $\texttt{CP}(mask, roi, (lv, uv)) >$



(a) *WILDS*, Pearson's $r = 0.99$ (b) *ImageNet*, Pearson's $r = 0.96$

Figure 4.8: Relationship between end-to-end query time and the fraction of masks loaded (FML) for a query.

T. For every query, roi is set as the foreground object bounding box in a mask generated by YOLOv5 [136]. lv and uv are randomly selected from $[0.1, ..., 0.9]$ and uv is always greater than lv. The count threshold T is randomly chosen from $[0, 1, ..., \text{total \# pixels}]$.

- **Top-K**: this query type returns masks ranked by $\texttt{CP}(mask, roi, (lv, uv))$. For each query, roi is randomly generated as any rectangle within the masks. This roi is generated once for each query and remains constant across all masks. k is set to 25. The order of query result, i.e., `ORDER BY ... DESC` or `ASC`, is randomly selected for each query.
- **Aggregation**: this type of query returns images ranked by $\texttt{mean}(\texttt{CP}(mask, roi, (lv, uv)))$ of multiple masks associated with each image. Two masks are associated with each image and they are generated by GradCAM based on different models. k is set to 25. roi, lv, uv, and the order of the query result is randomly selected for each query.

Figure 4.7 shows the distribution of query execution times for each query type on both *WILDS* and *ImageNet*. The figure displays the median, minimum, maximum, and interquartile range (IQR) of these times. The whiskers represent outliers, which are defined as values that

are more than 1.5 times the IQR away from the median.

MaskSearch demonstrates its superior query execution performance across all query types with varying parameters. Even when considering the worst-case execution time (i.e., the outliers), MaskSearch would still outperform the baselines by a considerable margin, because the baselines would still load all masks from disk and process them, regardless of the query parameters.

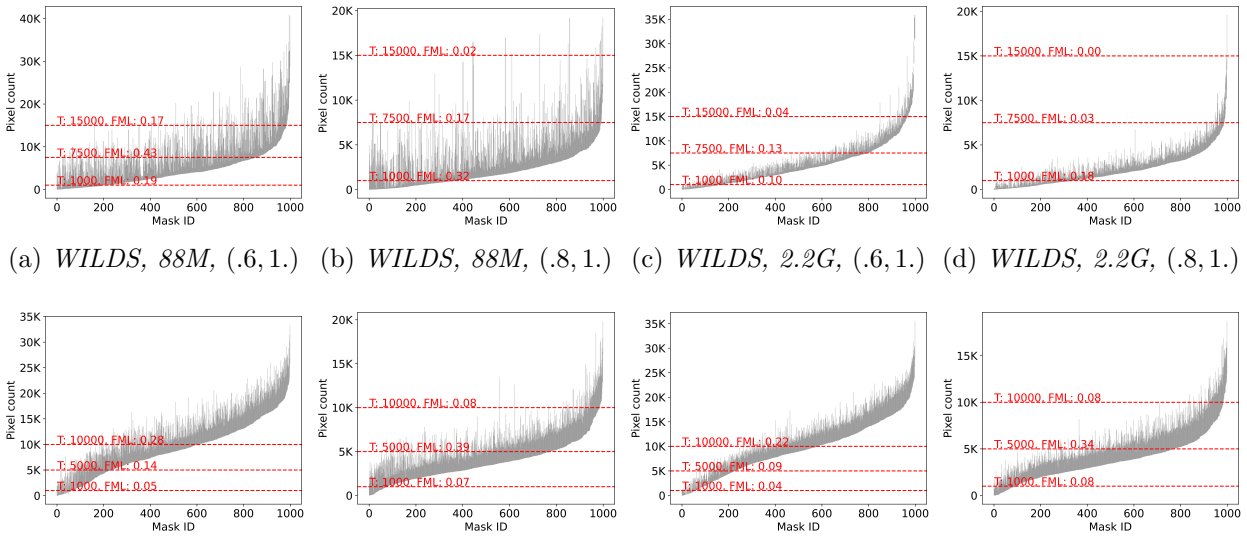
Moreover, we find that the query execution times of MaskSearch do not exhibit a strong correlation across different query types. We note that the 75th percentile of the *Filter* query type has a longer execution time than that of the other two query types. This is because, for the other query types, MaskSearch compares the bounds (of `CP`) with the `CP` values of the current top-k set (k=25). This process generally allows for more efficient mask filtering than comparing the bounds with a fixed count threshold T in the *Filter* query type. For example, on *WILDS*, at the 75th percentile in query time, the number of masks pruned in MaskSearch's filter stage during query execution is 21,184 for the *Filter* query type, 22,106 for *Top-K*, 21,677 for *Aggregation*.

Instead, we observe that the execution times tend to differ more significantly among queries with different parameters within the same query type. In fact, as we discuss further in Section 4.3.4, for a given dataset, the query execution time of MaskSearch is primarily determined by the fraction of masks loaded (FML), i.e., masks that are loaded from disk and used to compute its `CP` value during query execution. The difference in execution times within the same query type is mainly due to the difference in the FML for each query. For example, for the *Filter* query type on *WILDS*, the FML at the 25th, 50th, and 75th percentiles are 0.002, 0.012, and 0.049, respectively.

4.3.4 MaskSearch*'s Query Time Analysis*

In this section, we explore factors affecting MaskSearch's query execution time by analyzing 1500 *Filter* queries, defined in Section 4.3.3, executed by MaskSearch on each dataset.

With Figure 4.8, we first establish that, given a dataset, MaskSearch's query execution



(a) *WILDS, 88M,* $(.6, 1.)$ (b) *WILDS, 88M,* $(.8, 1.)$ (c) *WILDS, 2.2G,* $(.6, 1.)$ (d) *WILDS, 2.2G,* $(.8, 1.)$

(e) *ImageNet, 6.5G,* $(.6, 1.)$(f) *ImageNet, 6.5G,* $(.8, 1.)$(g) *ImageNet, 23G,* $(.6, 1.)$(h) *ImageNet, 23G,* $(.8, 1.)$

Figure 4.9: Distributions of bounds of $\texttt{CP}(mask, roi, (lv, uv))$ computed by MaskSearch. Each subfigure represents the distribution for a combination of $(\text{dataset}, \text{index size}, (lv, uv))$, shown as the title of each. Each vertical segment represents the lower and upper bounds of $\texttt{CP}(mask, roi, (lv, uv))$ for a single mask. For each mask, roi is the foreground object bounding box. We show the distribution of bounds for 1000 randomly sampled masks in each subplot. The x-axes represent the masks sorted by their lower bounds. The horizontal dashed lines represent examples of the count threshold T. FML is the fraction of masks loaded by MaskSearch given a predicate $\texttt{CP}(mask, roi, (lv, uv)) > T$. For each count threshold T, FML is equal to the fraction of the vertical segments that intersect with the horizontal dashed line defined by T. Note the different scales of the y-axes.

time is proportional to the fraction of masks loaded (FML) for each query. The FML for a query is defined as the ratio of masks loaded from disk and used to compute their actual CP values to the total number of masks the query targets. The Pearson's correlation coefficient between query time and FML is 0.99 for *WILDS* and 0.96 for *ImageNet*. It again corroborates that query execution time is dominated by loading masks from disk and computing their CP values, with a higher FML indicating more masks being loaded from disk.

Now that we have established the relationship between query execution time and FML, we investigate the factors that affect FML, including the query parameters (region of interest roi,

pixel value range (lv, uv), count threshold T), data in the masks ($mask$), and index granularity (index size). For MaskSearch, FML is the fraction of masks that are neither pruned nor added directly to the result set by the filter stage in the filter-verification framework. FML corresponds to *Case 3* in Step 2 of the filter stage; for each mask belonging to this case, its lower bound $\underline{\theta}$ for `CP` computed by MaskSearch is not greater than the count threshold T and its upper bound $\bar{\theta}$ for `CP` is greater than T, i.e., $\underline{\theta} \leq T < \bar{\theta}$.

Figure 4.9 shows the distribution of bounds computed by MaskSearch for both datasets and queries with varying parameters from the 1500 *Filter* queries analyzed. Each subfigure shows the distribution of bounds for a different (dataset, index size, (lv, uv)) combination. The roi for all subfigures is the foreground object bounding box. The (vertical) segments in each subfigure represent the bounds computed by MaskSearch for 1000 masks randomly sampled from the dataset. Each red horizontal dashed line represents an example count threshold T. In this way, each subfigure visualizes the relationship between the bounds and FML: for each count threshold T, FML equals the fraction of the segments intersecting with the red dashed line defined by T.

In each subfigure, different count thresholds T lead to varying FMLs for the same dataset, index size, and query parameters, as the fraction of segments intersecting with the red dashed line changes.

Comparing subfigures with the same roi and (lv, uv) but on different datasets reveals that different sets of masks can result in different FMLs for the same query parameters because of different pixel value distributions in the roi of the masks. Similarly, changing roi essentially alters the set of masks targeted by the query, leading to different FMLs. Subfigures with the same dataset and roi but different (lv, uv) configurations also exhibit different bound distributions and FMLs for the same count threshold T.

Moreover, subfigures sharing the same dataset and (lv, uv) but with varying index sizes display different bound distributions and FMLs. Larger index sizes offer more granular indexes, tighter bounds (shorter vertical segments), and lower FMLs for the same query parameters. For example, comparing Figure 4.9 (a) and (c), the bounds computed by MaskSearch for

WILDS with $(lv, uv) = (0.6, 1.0)$ are tighter for the larger index size. Therefore, the FML for the same count threshold T is lower for the larger index size.

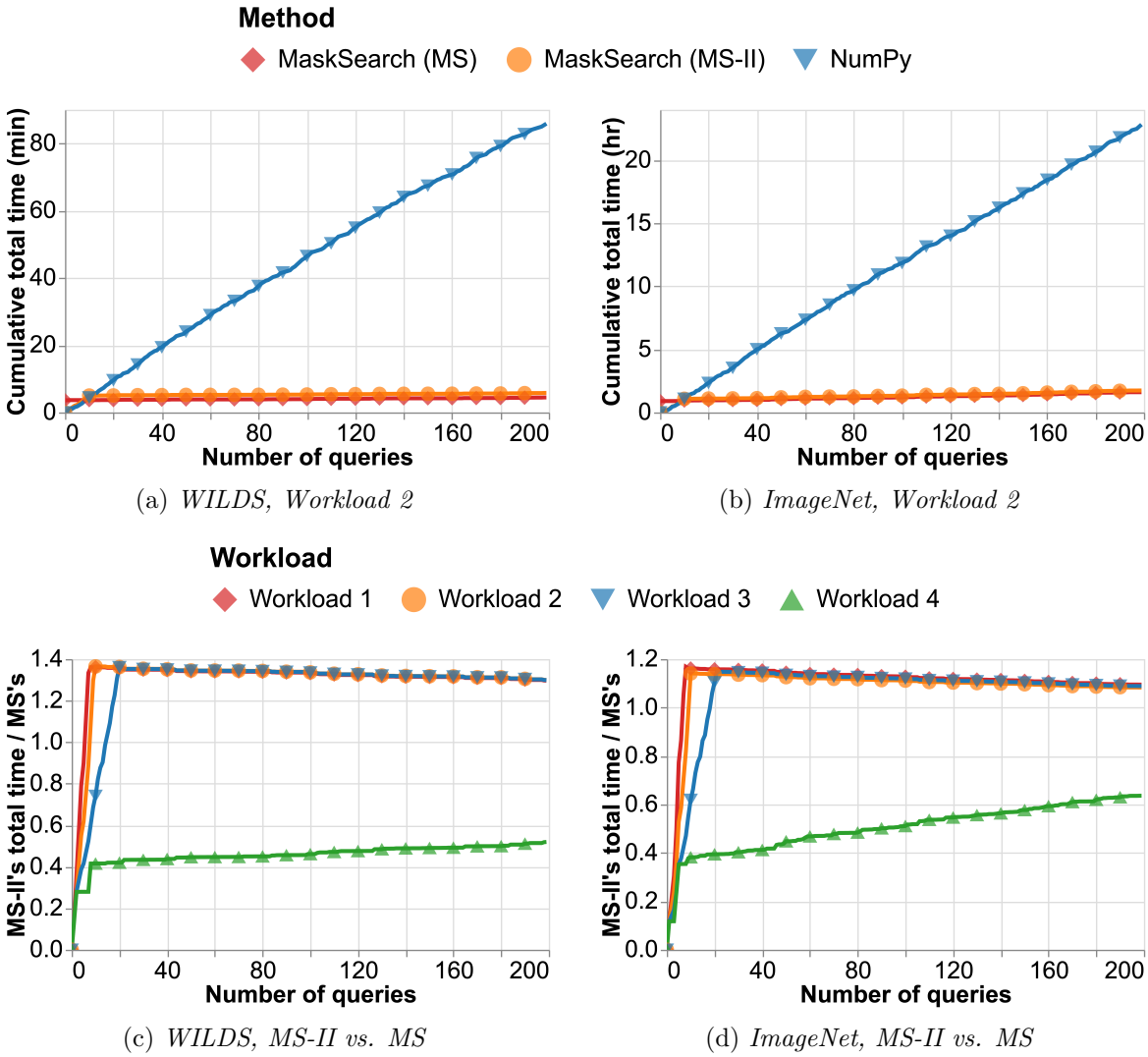
In summary, the data in the masks, region of interest roi, pixel value range (lv, uv), and index size determine the distribution of bounds computed by MaskSearch. The count threshold T defines the FML given the distribution of bounds, and the FML dictates the query execution time of MaskSearch. The granularity of the index represents a tradeoff between index size and query time, depending on user application requirements and available resources.

4.3.5 Multi-Query Workload Performance

In this section, we evaluate MaskSearch on multi-query workloads with and without the incremental indexing technique (Section 4.2.6) which mitigates MaskSearch's potential start-up overheads. We generate workloads to simulate the exploration and analysis processes of users who seek to identify sets of masks satisfying a given predicate.

We simulate workloads where a user begins with a query targeting masks of image subsets belonging to certain classes and then progressively explores masks associated with other classes. For example, to identify images with spurious correlations (Scenario 2 from the beginning of this chapter), the user may first look at the confusion matrix and identify classes with high false positive rates. Then, the user may issue queries to retrieve images predicted as those classes to identify possible spurious correlations. Several queries may be issued targeting those masks, as different query parameters (e.g., roi, lv, uv, T) may be used to retrieve and rank masks with different properties, e.g., masks focusing on the foreground object and masks focusing on the background. After analyzing the returned masks, the user may continue to explore masks of other classes and repeat the process.

To account for this behavior, we generate four different workloads for each dataset, each of which comprises 200 *Filter* queries, with query parameters randomly generated following the approach described in Section 4.3.3. A parameter p_{seen} is associated with each workload, representing the likelihood of querying previously targeted masks within the same workload.



(a) *WILDS, Workload 2*

(b) *ImageNet, Workload 2*

(c) *WILDS, MS-II vs. MS*

(d) *ImageNet, MS-II vs. MS*

Figure 4.10: Cumulative total time, incl. index building time and query time, for multi-query workloads. MS-II and MS refer to MaskSearch w/ and w/o incremental indexing, respectively. (a) and (b) show the total time for MS, MS-II, and NumPy for Workload 2; (c) and (d) show the ratio of the cumulative total time of MS-II to that of MS for all workloads. The index size for MS is $\sim 5\%$ of the corresponding dataset. MS-II builds the index incrementally using the same index configuration as MS.

Randomized query parameters and p_{seen} are intended to simulate the user's behavior of issuing multiple queries targeting the same set of masks with different parameters to retrieve masks

having different properties. Additionally, each query within a workload targets a specific subset of masks (e.g., masks of images predicted as certain classes) from the corresponding full dataset. Let N denote the total number of masks within a dataset. The number of masks targeted by each query, n, is randomly chosen from $[0.1 \cdot N, 0.2 \cdot N, 0.3 \cdot N]$. Then, the set of targeted masks is generated as follows, we sample without replacement n masks consisting of $p_{seen}\%$ targeted masks and $(1-p_{seen})\%$ unseen ones. Note that when the number of remaining unseen masks is less than $n \cdot (1 - p_{seen})$, we include all the unseen masks in the current query and switch to only sampling seen masks for the remaining queries in the workload.

The workloads are labeled as Workload 1, 2, 3, and 4, with their respective p_{seen} values set to 0.2, 0.5, 0.8, and 1.0. These probabilities signify varying levels of dataset exploration, with Workload 1 exhibiting the highest degree of exploration and Workload 4 exhibiting the lowest. By evaluating MaskSearch's performance across these diverse workloads, we aim to assess its effectiveness under a range of dataset exploration scenarios.

Figure 4.10 shows the performance of MaskSearch on these four workloads for both *WILDS* and *ImageNet*. MaskSearch is evaluated with and without incremental indexing against NumPy which represents existing methods that must load and process all masks from disk for each query. In the figure, MS-II refers to MaskSearch with incremental indexing and MS refers to MaskSearch without incremental indexing. We measure the cumulative total time, i.e., the time elapsed for index building plus the time elapsed for query execution, for each method. Note that the time to initially build the indexes without incremental indexing is included with the 0-th query for MS in all subfigures.

Figure 4.10 (a) and (b) show the cumulative total times for Workload 2. The results for other workloads are not shown because MS and NumPy have similar performance trends across all workloads. MS exhibits a slow growth in cumulative total time because it executes all queries efficiently with the filter-verification query processing framework. However, it incurs a start-up overhead due to the need to build indexes for all masks in the dataset ahead of time. In contrast, NumPy has no start-up overhead but suffers from rapid growth in its cumulative time because it does not reduce the required work for each query. Nevertheless,

the cost of building the indexes for MS is quickly amortized across the queries thanks to the filter-verification query processing framework and the CHI technique. On both datasets, MS outperforms NumPy after approximately 10 queries. MS-II strikes a good balance between MS and NumPy, eliminating the start-up overhead while achieving comparable query execution times to MS.

Figure 4.10 (c) and (d) show the ratio of cumulative total time between MS-II and MS for all workloads on both datasets. We first discuss the results for Workload 1, 2, and 3. For both datasets, we observe that this ratio grows rapidly at the beginning for Workload 1, 2, and 3, and then peaks at around 10 to 20 queries before decreasing gradually. The initial fast growth is due to the fact that for the first few queries, MS-II needs to answer them without the help of indexes for the unseen masks targeted, which is similar to the behavior of NumPy, and to build indexes for these masks. Among workloads, Workload 1 has the highest growth rate in this ratio because it has the lowest p_{seen} value, resulting in more unseen masks being targeted during the first few queries and therefore forcing MS-II to build indexes for more masks. Then, the ratio peaks at around 10 to 20 queries because, at this point, MS-II has built indexes for all the masks in the dataset, and subsequent queries can be executed using the filter-verification framework without index building. The peak ratio exceeds 1.0 because MS-II must load the masks from disk and compute their `CP` values during query execution the first time they are targeted. In contrast, MS utilizes pre-built indexes for all targeted masks in all queries, which results in a lower cumulative total time. Then, after the peak, the ratio decreases gradually because the cumulative total time for MS-II grows at a similar rate to MS's cumulative total time.

For Workload 4, on both datasets, MS-II never completes building the indexes for all masks, as only 30% of the masks in the dataset (6683 for *WILDS* and 399,351 for *ImageNet*) are eventually targeted by all the queries in this workload. As a result, after the rapid initial growth, the ratio of cumulative total time plateaus. This ratio never reaches 1.0 because the time spent by MS to build the indexes for the never-targeted masks is not amortized across queries.

Lastly, we note that users typically pause between queries to examine results. Hence, MaskSearch can leverage this interval to compute indexes, yielding better user-perceived latencies.

4.3.6 Real-World Use Cases

This section shows MaskSearch's real-world utility.

Improving Model Performance with MaskSearch on *WILDS*. *WILDS* is a benchmark designed to evaluate the robustness of ML models to distribution shifts [154]. We used MaskSearch to help improve the performance of an image classification model for the iWildCam dataset in *WILDS* by identifying images with spurious correlations and retraining the model with these images added to the training set after augmentation. The model we started from was a ResNet-50 model trained via empirical risk minimization (ERM) downloaded from the *WILDS* repository. We first issued a query to MaskSearch to retrieve the top-50 masks (and their corresponding images) that have the fewest salient pixels (i.e., pixel value > 0.8) in their object bounding boxes generated by YOLO [136]. The reason for this query is that images with spurious correlations often contain salient pixels in the background that the model may have learned to rely on; we would like the model to focus on the foreground object instead. Without MaskSearch, this query would take more than 2 minutes; with MaskSearch, it took less than a second. We then augmented these images by randomizing the pixels outside the bounding boxes of the objects and keeping the pixels inside the bounding boxes unchanged [260]. We added the augmented images to the original training set of iWildCam with their original labels and retrained the model for 2 epochs. After retraining, we found that the model's accuracy improved from 60% to 70% on the held-out OOD test set of iWildCam, which is a 16.7% relative improvement.

Understanding Vision Foundation Models with MaskSearch for Ophthalmology and Histopathology. We worked with a team of computational biologists who developed vision foundation models for ophthalmology and histopathology [277]. They generated gradient-based saliency maps [175] to study the features that the ophthalmology model

learned to predict diseases in 3D optical coherence tomography (OCT) images. To understand which images (2D OCT slices) contain the most signal and whether the signal learned by the model aligns with domain expertise, they issued declarative queries to MaskSearch to efficiently identify slices with the most (or fewest) salient pixels. They reported this process would have been more tedious without MaskSearch, commenting "MaskSearch makes our work much more efficient and allows us to focus on the analysis of the results rather than waiting for the results to be computed." They also noted that MaskSearch can be used for histopathology where the entire whole-slide images have large digital resolutions (e.g., 10K-100K $\times$ 10K-100K pixels) and models take patches of these images as input. These patches are usually 256 $\times$ 256 in size, so the number of patches for a single image can be up to 10K. MaskSearch can help them quickly identify the patches where there are likely diseased regions (e.g., patches with a large number of salient pixels).

4.4 MaskSearch *User Interface*

To further demonstrate the usability of MaskSearch, we develop a graphical user interface (GUI) for MaskSearch and designed a demonstration [272] that showcases MaskSearch's capabilities within real-world ML workflows, including a use case for improving model performance on *WILDS* described in Section 4.3.6, a scenario for searching for images attacked by adversarial perturbations described earlier in the chapter, and a scenario for investigating the discrepancies between human attention and model saliency for fine-grained image classification. In this section, we describe the MaskSearch GUI, as shown in Figure 4.11.

MaskSearch allows users to load and specify their models, datasets, and masks. This process is followed by the automatic calculation and display of the model's accuracy and a confusion matrix where each clickable cell represents the images whose ground truth label and predicted label are the corresponding row and column of the cell, respectively. For example, cell (146, 17) represents images of class 146 that were classified as class 17. The GUI shows the top-100 cells in terms of the number of misclassifications. As illustrated in Step 1 in Figure 4.11, this functionality allows for detailed visualization of the images from

Figure 4.11: An example workflow of using MaskSearch's GUI. In Step 1, 146 → 17 means that the image with a ground truth label 146: Meleagris Ocellata was misclassified as class 17: Panthera Onca. The images are from WILDS [154].

the selected cell (146, 17) with their corresponding masks. The initial data loading and the confusion matrix cells are not presented in Figure 4.11.

Input Section. The Input Section is demonstrated on the left of Steps 2 and 3 in Figure 4.11. It simplifies the creation and manipulation of search queries by providing a form that guides users through specifying their query, including defining an optional ROI (full mask by default), upper and lower bounds of the pixel value range, and choosing between different queries such as Top-K Query, Filter Query, and Aggregation Query. The ROIs are provided by the user, such as object bounding boxes generated by an off-the-shelf model. Based on the aforementioned user-specified parameters, the GUI generates an SQL query shown in the "Query Command" window, which allows users to inspect the formalized query and, if necessary, directly modify the SQL query for their search. Clicking "Execution Detail" (which needs to happen after clicking "Start Query") triggers the GUI to show the number of masks loaded from disk during query execution vs. the number of total masks.

Query Result Section. The Query Result Section, presented on the right of Steps 2 and 3 in Figure 4.11, displays the query results as a combination of images and their corresponding

masks, dependent on the specific scenario. For example, in Step 2 of Figure 4.11, the returned images are overlaid with their saliency maps and the object bounding boxes. The GUI also offers users the ability to click and zoom in on the query results in a popup window.

Dataset Augmentation. The interface also incorporates a dataset augmentation feature, as shown in Step 3 of Figure 4.11. This feature allows users to augment the returned images by randomizing the pixels outside the ROI with the original labels. Such an approach is known to guide the model to classify the images without relying on the randomized background pixels and thus improve the model's performance [260].

With the GUI, users can interactively explore image databases through mask properties and efficiently issue queries to retrieve images (and their masks) of interest.

4.5 Summary

In this chapter, we introduced MaskSearch, a system that accelerates queries that retrieve examples based on mask properties. By leveraging a novel indexing technique and an efficient filter-verification execution framework, MaskSearch significantly reduces the masks that must be loaded from disk during query execution. With an index 5% of the original dataset in size, MaskSearch accelerates individual queries by two orders of magnitude and consistently outperforms existing methods on various multi-query workloads.

Chapter 5

TQP: QUERY PROCESSING ON TENSOR COMPUTATION RUNTIMES

Database management system vendors have delivered constant performance improvement for decades by evolving software to keep up with Moore's law while influencing hardware development through close relationships with manufacturers. While data volumes and demand for analytics are growing faster than ever [250], the query performance improvement on CPUs has slowed down [261]. However, the count of processor transistors has continued to grow over the last decade, as hardware manufacturers adopted first multi-core CPU architectures and then augmented their computing platforms with specialized components such as GPUs, FPGAs, compression and encryption chips, digital signal processors (DSPs), and neural network (NN) accelerators. Although data management system builders have taken advantage of multi-core and SIMD instructions effectively [297, 216, 149], the explosion in the number of specialized hardware components, each with different characteristics and programming abstractions, makes it challenging to support all the exciting capabilities that these new powerful devices can offer.

On the other hand, the tremendous demand for memory and computation in AI [102], combined with the market fever for AI, is driving unparalleled investments in new hardware and software for AI. Hardware makers (e.g., Intel [111], Apple [44], Xilinx [276], AMD [40]), cloud vendors (e.g., Amazon [49], Microsoft [77], Google [138], Meta [96]), startups (e.g., Graphcore [6], Sambanova [12], Cerebras [4]), and car companies like Tesla [259] are investing heavily in this space. Venture capitals alone are pouring nearly $2B a quarter on special hardware for AI, aiming for a market expected to exceed $200B a year by 2025 [251]. On the software side, companies and open source communities are rallying behind a small number of

big efforts (e.g., PyTorch [10], TensorFlow [36], TVM [75]). The combination of investments in specialized hardware and large software communities focusing on performance allows these efforts to thrive. Our realization is that the machine learning (ML) community has made hardware accelerators accessible to nonspecialists (e.g., data scientists). The fact that the most popular ML frameworks are open-source, creates a virtuous cycle whereby any hardware vendor interested in the ML space must support these frameworks well to get adoption. At the same time, these large open source communities successfully tackle the labor-intensive problem of providing specialized kernels for various hardware, e.g., a month after Apple M1 was announced, TVM outperformed Apple's CoreML by 2× [258]. Hardware vendors can directly improve the kernels' performance or the hardware itself [23, 24, 27]. This further helps adoption since the performance improves at each new software and hardware release.

We argue that the best path forward for analytical database management systems is to embrace this tectonic shift and take advantage of the groundswell of new hardware and software targeting AI workloads. To demonstrate the viability of this idea, we propose and prototype a new query processor that runs SQL queries atop tensor computation runtimes (TCRs) such as PyTorch, TVM, and ONNX Runtime [25]. We name our prototype *Tensor Query Processor* (TQP). TQP transforms a SQL query into a tensor program and executes it on TCRs. To our knowledge, TQP is the first query processor built atop TCRs. Careful architectural and algorithmic design enables TQP to: (1) deliver significant *performance* improvements over popular CPU-based data systems, and match or outperform custom-built solutions for GPUs; (2) demonstrate *portability* across a wide range of target hardware and software platforms; and (3) achieve all the above with *parsimonious* and sustainable *engineering effort*.

The above might appear surprising as specialized hardware accelerators are notoriously hard to program, requiring much customization to extract the best performance. Furthermore, their programming abstractions differ sufficiently to make our goals of *performance* (G1), *portability* (G2), and *parsimonious engineering effort* (G3) seemingly hard to reconcile. However, the key is a compilation layer and a set of novel algorithms, which can map the

classical database abstraction to the prevalent one in ML, i.e., *mapping relational algebra to tensor computations*. This allows us to free-ride on existing labor-intensive efforts from the ML community to port and optimize TCRs across all the new specialized hardware platforms. The initial performance results are encouraging. On GPU, TQP is able to outperform open-source GPU databases in terms of query execution time. On CPU, TQP outperforms Spark [282], and it is comparable to a state-of-the-art vectorized engine, DuckDB [224], for several queries. Furthermore, when ML model prediction and SQL queries are used in concert, TQP is able to provide end-to-end acceleration for a 9× speedup over CPU baselines.

Pursuing our goals of *portability* and *parsimonious engineering effort*, we make a deliberate decision to target existing tensor APIs rather than customize lower-level operators. This decision potentially leaves some performance on the table but leads to a very sustainable long-term play, as TQP benefits from any performance enhancement and optimization added to the underlying software and hardware (e.g., [23]). To validate this proposition, we run TQP on different hardware settings: from CPUs, to discrete GPUs, to integrated GPUs (Intel and AMD), to NN accelerators (e.g., TPUs [138]), and web browsers. Furthermore, TQP is able to run the full TPC-H benchmark on both CPU and GPU with just around 8,000 lines of code—this is quite an achievement considering that until 2021 no GPU database was able to run all the 22 TPC-H queries [165].

Contributions. This chapter makes the following core contributions:

- We propose Tensor Query Processor (TQP) that comprises a collection of algorithms and a compiler stack for transforming relational operators into tensor computations.
- With TQP, we demonstrate that the tensor interface of TCRs is expressive enough to support all common relational operators.
- We evaluate the TQP approach extensively against state-of-the-art baselines on the TPC-H benchmark.

Organization. The remainder of the chapter is organized as follows. Section 5.1 introduces some background on TCRs. Section 5.2 summarizes the challenges and the design choices we

make. Section 5.3 introduces TQP, and Section 5.4 describes the algorithms used to implement several key relational operators with tensor programs. Experiments are in Section 5.5. Section 5.6 discusses the limitations of the current TQP implementation. The chapter is summarized by Section 5.7.

5.1 Background

In this section, we summarize the system support for tensor computation (Section 5.1.1), and provide a taxonomy of the tensor operations used throughout the chapter (Section 5.1.2).

5.1.1 Tensor Computation Runtimes (TCRs)

The last years have witnessed an increase in the popularity of ML models based on NNs [105]. While in the heydays, these models were implemented manually in C++, data scientists now can take advantage of several open-source ML frameworks simplifying the authoring and deployment of NN models. TensorFlow [1] and PyTorch [207] are considered the most popular of such frameworks.

ML frameworks follow a common architecture: at the top, they have a *high-level* Python API[1] where data is commonly represented as multi-dimensional arrays called *tensors*, while computation is expressed as a composition of *tensor operations* embedded into the Python language. At the lower level, they have a *runtime* and a *dispatcher/compiler* allowing to run the tensor operations over different hardware backends such as CPU, GPU, and custom ASICs, and using single node execution, distributed [167], or mobile/edge [107].

Modern ML frameworks allow running computation in an *interpreted mode* (often referred to as *eager execution*), or in a *compiled mode* (*graph execution*), enabling code optimizations such as common sub-expression elimination, operator fusion, code generation [20], and removing Python dependency [264, 263]. Interpreted vs. compiled execution is a popular dichotomy in query processing system implementations [148]. ML frameworks allow both

[1]Note that TCRs allow implementation in other languages too (e.g., Java [220], Rust [181], C# [98]). Python is however the default language of choice by data scientists.

modalities, and we explore the tradeoffs involved when using one versus the other. The current limits of tensor compilers are discussed in Section 5.5.

We will refer to ML frameworks, runtimes [25, 2], and compilers as tensor computation runtimes (TCRs) in the rest of the chapter.

5.1.2 Tensor Operations

TCRs provide hundreds of tensor operations. We provide a brief summary of the operators used in TQP, organized by category. Since TQP is currently built on top of PyTorch, from now on we will use the PyTorch naming convention. Note that similar tensor operations can be found in other TCRs. Additionally, here we take the freedom to provide a different taxonomy than the one found in the PyTorch documentation [222] and in previous work [157].

Creation. This category contains all operations used to create tensors, e.g., `from_numpy`, fill a tensor with specific elements (`zeros`, `ones`, `empty`, `fill`, `arange`) or create a tensor using the same shape of another tensor (`zeros_like`, `ones_like`).

Indexing and slicing. This category involves operations for selecting one or more elements of a tensor using the square bracket notation, or using indexing (`index_select`), a mask (`masked_select`), or a range (`narrow`).

Reorganization. This category includes `reshape`, `view`, and `squeeze` that reorganize the shape of a tensor (eventually by changing only its metadata). `gather`, `scatter` reorganize the elements of a tensor using an index, while `sort` sorts its elements.

Comparison. `eq`, `lt`, `gt`, `le`, `ge`, `isnan` are operators in this category. Other operations are `where` that implements conditional statements, and `bucketize` that implements binary search.

Arithmetic. `add`, `mul`, `div`, `sub`, `fmod`, `remainder` are in this category. We also include logical operators such as `logical_and`, `logical_or`, `negative`, and shift operations.

Join. This category allows to `concat` or `stack` multiple tensors.

Reduction. This category contains operations for calculating simple aggregates (`sum`, `max`, `min`, `mean`), aggregates over groups (`scatter_add`, `scatter_min`, `scatter_max`, `scatter_mean`),

logical reductions (`all`, `any`), as well as operations to build histograms (`bincount`, `histc`), `nonzero` (returning the indexes of non-zero elements), `unique` and `unique_consecutive`.

5.2 Query Processing on TCRs

In this section, we summarize the challenges (Section 5.2.2) and the design principles we commit to (Section 5.2.3) when building TQP. First, we show how relational operators can be implemented using tensor programs with an example (Section 5.2.1).

5.2.1 Relational Operators as Tensor Programs

TCRs operate over data represented as tensors. Tensors are arrays of arbitrary dimensions containing elements of the same data type. 0d-tensors are referred to as *scalars*, 1d-tensors as *vectors*, and 2d-tensors as *matrices*. For a tensor of n dimensions, its *shape* is an n-tuple where each element $i \in \{0, 1, \ldots, n\}$ specifies the size of the i-dimension. For example, a matrix with 10 rows and 5 columns is a 2d-tensor of shape (10, 5). This work only considers dense tensors where each element is explicitly stored in memory.

ML practitioners implement programs (NNs) as a composition of operations over tensors. While relational operations are commonly expressed as queries in a standalone language (e.g., SQL), tensor operations are embedded in a host language (e.g., Python), which is used to implement control flows and etc. Next, we introduce examples of implementing a filter using tensors.

Let us assume that we want to implement a simple filter condition over the L_QUANTITY column of the LINEITEM table: WHERE L_QUANTITY < 24. First, we can represent L_QUANTITY as a 1d-tensor of floating point numbers. We can then use the `lt` (`l`ess `t`han) operator to implement the filter condition (line 1 of Listing 5.1). `lt` generates a boolean mask which is then used as a parameter of the `masked_select` operator to generate the filtered version of the L_QUANTITY column vector (line 2 of Listing 5.1). The program can be easily extended over multiple conditions by intersecting the masks using `logical_and`.

Listing 5.1: Filter implementation using bitmaps.

```
1 mask = torch.lt(l_quantity, 24)
2 output = torch.masked_select(l_quantity, mask)
```

This implementation is almost identical to the Bitmap-based representation [199] of filters in vectorized query processors [217, 225]. On CPU, TCRs have SIMD implementations for several condition and intersection operators. An alternative is to use indexes rather than masks. This is commonly referred to as Selection Vector representation [199, 232], and can be similarly implemented using tensor operators `lt`, `nonzero`, and `index_select`. Listing 5.2 shows how a selection vector can be extracted from the mask using the `nonzero` operation, returning the indexes of the valid rows (line 2). The output of the filter can then be built by fetching the values of the rows whose indexes are in the selection vector. This is implemented by `index_select` on line 3 (where `dim=0` is used to specify the indexes over the row dimension will be used). Again, this implementation is quite close to the one used in vectorized query processors [232], where, for example, the nonzero operation is implemented using the `COMPRESSSTORE` instruction on AVX-512 CPUs.

Listing 5.2: Filter implementation using selection vectors.

```
1 mask = torch.lt(l_quantity, 24)
2 idx = torch.nonzero(mask)
3 output = torch.index_select(l_quantity, dim=0, mask)
```

Listing 5.3 shows another implementation of the filter using Python control flow. Here, we iterate over all the elements of the input tensor and use a Python conditional statement. This implementation does not take advantage of any tensor operation beyond creating the output tensor.

Listing 5.3: Filter implementation using Python data-dependent control flow.

```
1 output = torch.zeros_like(l_quantity), j = 0
```

```
for i in range(l_quantity.shape[0]):
    datum = l_quantity[i]
    if datum < 24:
        output[j] = datum
        j = j + 1
output = output[:j, :]
```

Table 5.1 shows the performance of the three implementations. Selection Vector is faster than Bitmap in this case because the filter has a high selectivity. The implementation using Python control flow is considerably slower, and GPU execution of Python control flow is slower than CPU execution. This result highlights one of the design choices (Section 5.2.3) we make in TQP: avoid the use of data-dependent code in Python.

5.2.2 Challenges

Implementing a query processor on TCRs requires overcoming several challenges. After all, TCRs are built for authoring and executing NN models, not relational queries.

C1: *Expressivity.* Relational queries can contain filters with fairly complex expressions (e.g., LIKE, IN), sub-queries, group-by aggregates, joins (e.g., natural, anti, semi, outer), etc. It is not clear whether the tensor operations currently available in TCRs are enough to support all these relational operators.

Table 5.1: Execution times of filter over ∼6M elements in interpreted (Torch) and compiled (TorchScript) modes. The machine used was an Azure NC6 v2.

Implementation	CPU		GPU	
	Torch	TorchScript	Torch	TorchScript
Bitmap	36.6 ms	36.6 ms	2.9 ms	2.9 ms
Selection Vector	19.2 ms	18.2 ms	2.8 ms	2.8 ms
Python	23 s	22.7 s	200.3 s	200 s

C2: *Performance.* Even if a relational operator is implementable using tensors, this does not automatically lead to good performance, as the example in Listing 2 suggests. In fact, it is not clear whether tensor programs can achieve good performance, beyond NNs.

C3: *Data Representation.* To use TCRs as execution engines, relational tables must be transformed into a tensor representation. Previous approaches have explored this challenge (e.g., [126]), but their cost of translation is not negligible. Furthermore, TCRs commonly do not support strings or date data types.

C4: *Extensibility.* Running relational queries over TCRs makes running a query seamlessly over different hardware (CPU, GPU, ASICs, etc.) and backends (single node, distributed, edge, web browser, etc.) possible. A single monolithic compiler architecture does not work in all situations, therefore TQP's design must be flexible enough to address all these use cases.

5.2.3 Design Choices

When building TQP, we embrace the following design choices.

DC1: *Avoid implementing data-dependent control flow in Python.* As Table 5.1 suggests, computation in TQP must use tensor operations as much as possible. Note that for loops and conditionals over schema elements are acceptable (e.g., loops over the columns of a table). This design choice allows us to address **C2** and achieve **G1**.

DC2: *Tensor-based columnar format for input tabular data.* Relational data must be transformed into the tensor format. To do this, TQP adopts a columnar representation of tables, and considers each column in a table as a tensor. We provide more details on our data representation in Section 5.3.1. This design choice addresses **C3**.

DC3: *Adherence to TCRs' API.* This design choice is required for achieving **G2** and **G3**. In fact, if we start extending TCRs with new features and operators, eventually the system will hinter portability and increase the engineering effort because we will have to support them on any hardware. Hence, we take advantage of existing TCRs' API rather than try to extend them. With this design choice, we are also able to address **C1**.

DC4: *Extensible infrastructure allowing easy integration with relational and ML frame-*

works. Having a flexible infrastructure is of paramount importance since we desire to ride the wave of investments in ML. Therefore, we embrace an extensible architecture that allows different output target formats (e.g., PyTorch, ONNX), composed of a core compiler, pluggable frontends (e.g., query parser and optimizer). This design choice addresses **C4**.

Note that some of the design choices were made specifically to address some of the previously mentioned challenges. For instance, we aimed to address **C4** with **DC4**, while we can address the relational-to-tensor abstraction mismatch (**C3**) thanks to **DC2**. Finally, we can avoid disastrous performance by embracing **DC1**. Next we introduce TQP.

5.3 Tensor Query Processor (TQP)

In TQP, relational operators and ML models are compiled into tensor programs using a unified infrastructure, extended from Hummingbird [195, 189]. Here, we focus on the relational operator part, as the ML part was described in [195].

TQP Overview. TQP's workflow has two phases: (1) *compilation*: an input query is transformed into an executable tensor program; (2) *execution*: input data is first transformed into tensors, and then fed into the compiled program to generate the query result. Currently, TQP uses vanilla PyTorch in the compilation phase as the implementation target for the tensor programs. If necessary, PyTorch programs are lowered into different target formats for portability or performance goals. The selection of the hardware device to target is generally made in the compilation phase. Next, we first describe how TQP represents relational data using tensors (Section 5.3.1), and then describe each phase in detail (Section 5.3.2 and Section 5.3.3).

5.3.1 Data Representation

Before executing the query, TQP must convert the input (tabular) data to tensors. Databases often manage and convert data into their own proprietary format, and TQP is no different. TQP internally represents tabular data in a columnar format with virtual IDs [34], as

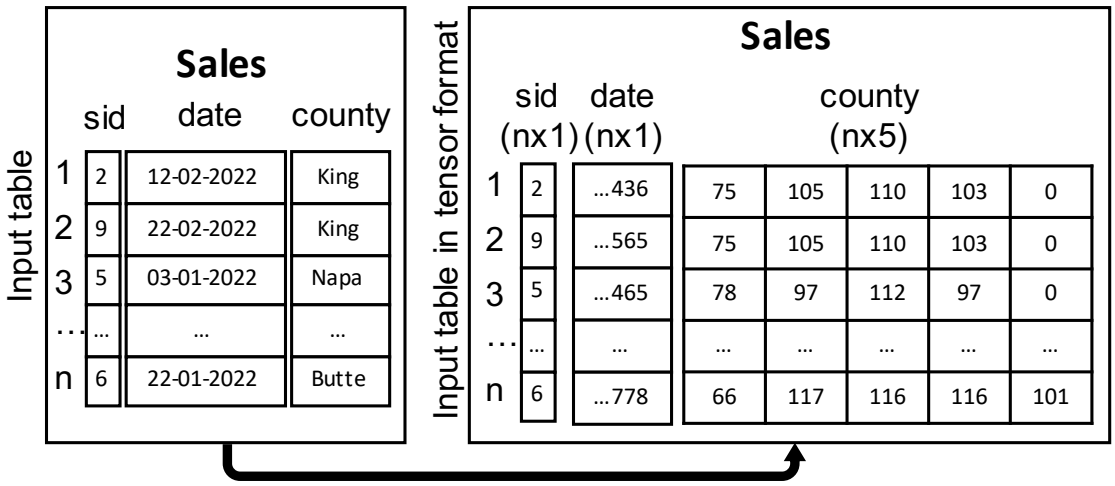


Figure 5.1: TQP represents input tables in a columnar format with a 2d-tensor per column.

illustrated in Figure 5.1. Data for each column is stored as a $(n \times m)$ tensor, where n is the input number of rows, and m is the length required to store the values. The translation logic is different depending on the column data type. For example, *numerical* columns (sid in Figure 5.1) can be directly represented as $(n \times 1)$ tensors. The conversion of numerical columns to tensors is often zero-copy. TQP represents *date* data in $(n \times 1)$ numeric tensors as the number of nanoseconds since some pre-defined epoch. In this case, (de)serialization may be required depending on the source/target *date* representation. Finally, TQP represents string columns using $(n \times m)$ numeric tensors, where m is the maximum character length of any string for that column. Given a string, TQP stores a character per tensor column and right-pads it with 0s if its length is smaller than m. We are actively working on adding support for encoded data (e.g., bit packing, run-length encoding, dictionary encoding) and more compact string representations [18].

5.3.2 Query Compilation

TQP's compilation phase is composed of four main layers, as shown in Figure 5.2: (1) The Parsing Layer (Section 5.3.2) converts an input SQL statement into an internal *intermediate*

representation (IR) graph depicting the query's physical plan, which is generated by an external *frontend* database system. The architecture decouples the physical plan specification from the other layers, therefore allowing for different frontends to be plugged in. (2) The Canonicalization and Optimization Layer (Section 5.3.2) performs IR-to-IR transformations. (3) The Planning Layer (Section 5.3.2) translates the IR graph generated in the previous layer into an *operator plan* in which each operator is mapped into a tensor program implementation. (4) The Execution Layer (Section 5.3.2), using the operator plan, generates an *executor* which is the program that runs on the target TCR and hardware. Next, before describing each layer in more detail, we give a quick overview of TQP's intermediate representation (IR).

Intermediate Representation (IR)

The IR is a graph-based data structure. It consists of a list of *operators* and *variables*. Each *operator* corresponds to a node in the graph, and it contains: (1) a list of input variables; (2) a list of output variables; (3) an *alias* identifying the operator type (e.g., SparkSQLFilter, SparkSQLSort and following the format ¡frontend name¿¡operator name¿) and (4) a *reference* to the corresponding operator instance in the original physical plan. The latter is used to instantiate the tensor program implementing the operator. For example, to create a filter, TQP needs to access the expressions contained in the original physical operator.

Edges represent data (tensors) flowing between operators. In particular, an edge connects an output variable from an operator to an input variable of another operator. A *variable* contains: (1) a unique identifier, and (2) the corresponding frontend column name in the original plan, which is used to translate expressions. When a variable is created, a unique identifier is generated deterministically based on information available in the graph. Variables in the IR are generated as follows. First, TQP generates a variable for each column in the input table. Then, these variables can be used as input to many operators; however, a new variable will always be created for the output of an operator. Thanks to this design: (1) properties (e.g., sorting information) can be immutably attached to columns; (2) the IR is easier to debug because variables, once defined, are never changed; and (3) TQP can detect

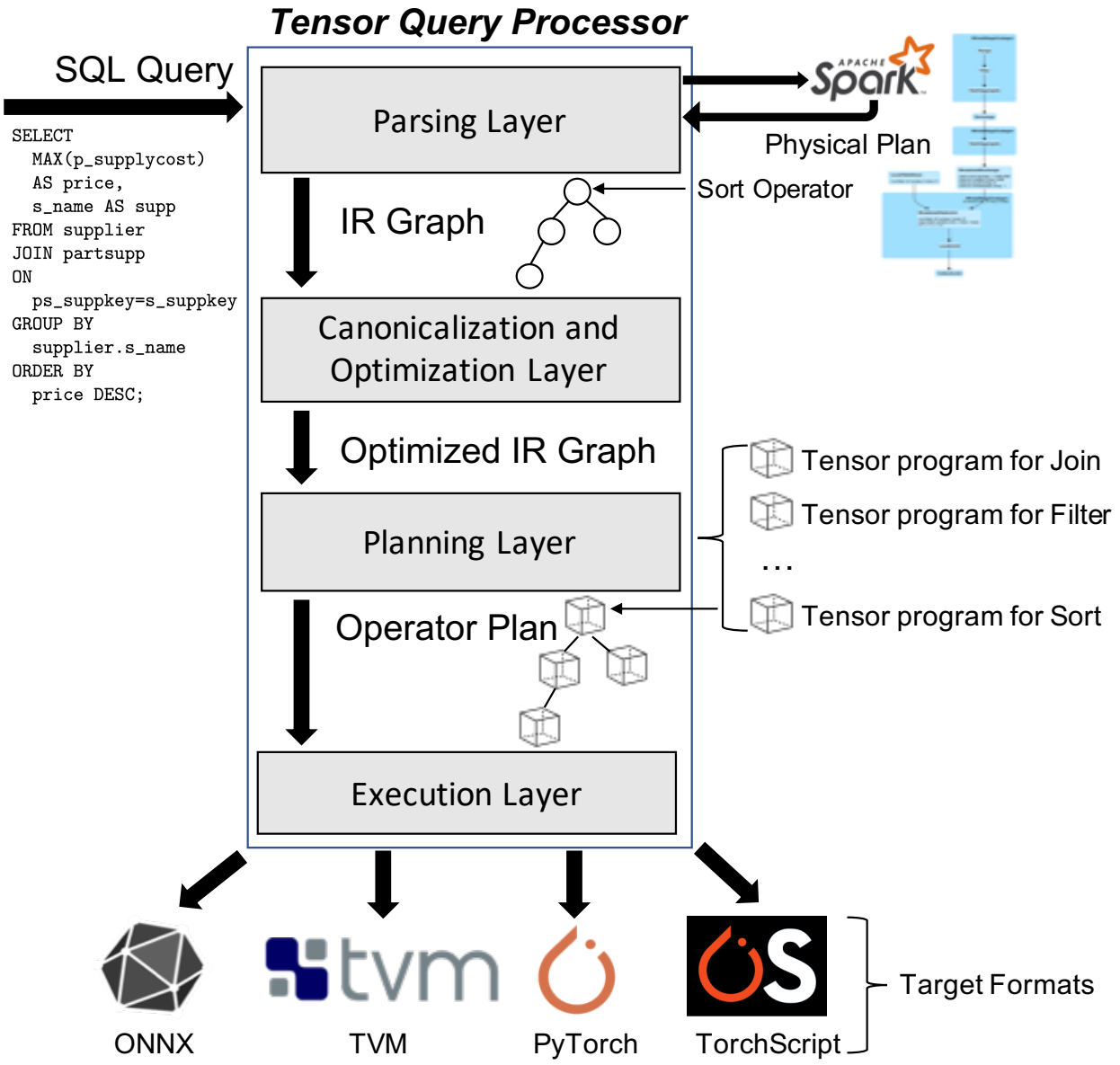


Figure 5.2: TQP's compilation phase.

at runtime when a column is not used anymore and safely garbage-collect it.

Parsing Layer

The goal of the Parsing Layer is to translate input queries into TQP's internal IR. This goal is accomplished in two steps: (1) input queries are parsed, optimized, and exposed as frontend-specific physical query plans; and (2) a frontend-specific parsing logic translates the physical plan into an IR plan.

In its current version, TQP supports queries expressed as Spark SQL statements, and it uses the PySpark API to parse, optimize, and return the physical plan in a JSON format. We plan to add support for Calcite [56], DuckDB [224], and Substrait [28][2]. Then the Spark parser constructs the internal IR version of the physical plan using a DFS post-order traversal. If an unsupported operator is found in the plan, this phase will fail with an exception. The list of operators supported by the IR is extensible (**DC4**).

Canonicalization and Optimization Layer

This layer implements IR graph transformations similarly to a classical rule-based optimizer. Rules are applied to the IR graph in two stages. In the first stage, *canonicalization*, the rules are used to eliminate any of the idiosyncrasies in the frontend system. For example, Apache Spark returns a projection operator with no inputs for COUNT * statements. In the second stage, *optimization*, rules rewrite the IR graph to improve the performance of the query. Examples of rewrite rules include pushing projection operators into filters or merging adjacent aggregate and projection operators. While we did not explore in depth the optimization space enabled by TQP's design, we show that hand-optimized tensor programs are more efficient than the one currently generated by TQP in Section 5.5.6.

[2]Note that we currently only support Apache Spark for relational frontends, but this is not a general limitation. TQP, in fact, supports all the ML frontends available in Hummingbird [189].

Planning Layer

In this layer, TQP transforms the optimized IR graph into an operator plan composed of PyTorch tensor programs implementing each operator in the IR graph. In Section 5.4, we describe some operator implementations in detail. The implementation of the Planning Layer is straightforward. For each operator in the IR graph, TQP fetches the corresponding implementation containing the tensor program from a dictionary, which is then instantiated with the IR operator's reference to the frontend physical operator instance.

Execution Layer

Here the operator plan is wrapped around a PyTorch *executor* object. This object is responsible for: (1) calling the tensor programs in the operator plan following a topological order; (2) wiring the output tensors generated by each program into the successive one; and (3) keeping track of tensor references to garbage collect them if not used anymore. Once the executor program is generated, TQP provides options to compile it into different *target formats* in addition to PyTorch interpreted execution. Currently, TQP allows lowering the executor into the TorchScript and ONNX formats, as well as to use TVM to compile it directly into machine-level code. Note that not all queries can be compiled into all formats since not all tensor operations are supported by all the target formats.

5.3.3 Query Execution

Once the executor program is generated, it can be executed over the input data. The program automatically manages (1) converting data into the tensor format; (2) data movements to/from device memory; and (3) scheduling of the operators in the selected device. Once the data is in the proper format and on the desired device, all the operators are executed sequentially. Regarding parallelization, TQP exploits the tensor-level intra-operator parallelism provided by the TCRs. However, given the poor scalability performance (Section 5.5.3), we are exploring support for inter-operator parallelism and data-parallel strategies. Once the

executor completes, TQP returns the query result in tensor, NumPy, or Pandas formats.

5.4 Operator Implementation in TQP

We described how TQP uses the Planning Layer to translate relational operators in the IR graph into tensor programs. Here we provide an overview of a few program implementations. TQP provides tensor-based implementations for the following relational operators: selection, projection, sort, group-by aggregation (sort-based), natural join (hash-based and sort-based), non-equi, left-outer, left-semi, and left-anti joins. TQP supports expressions including comparison and arithmetic operations, functions on *date* data type, IN, CASE, LIKE statements, as well as aggregate expressions using SUM, AVG, MIN, MAX, and COUNT aggregates (with and without DISTINCT). Finally, TQP supports *nulls*, and sub-queries (scalar, nested, and correlated), and PREDICT UDF[3] [187, 188]. With all the above, TQP is able to compile and execute all 22 queries of the TPC-H benchmark (**C1**). Interestingly, to support the full TPC-H benchmark, only the tensor operations listed in Section 5.1.2 are required, and we did not have to introduce any additional custom tensor operators (**DC3**). Among all the above operators, we describe how TQP implements relational expressions with tensor operations (Section 5.4.1), and implementations for two representative operators: join (sort- and hash-based, in Section 5.4.2 and Section 5.4.3, respectively), and group-by aggregation (Section 5.4.4). Finally, note that the filter implementation in TQP is close to the Bitmap representation described in Section 5.2.1.

5.4.1 Expressions

Relational expressions such as SUM(L_EXTENDEDPRICE $*$ (1 - L_DISCOUNT)) can be found in projection operators, filter conditions, etc. In an expression tree, each leaf node represents a column or a constant value (e.g., L_EXTENDEDPRICE) and each branch node represents an operator (e.g., $*$). TQP keeps an internal dictionary that maps operators to their

[3]While generic UDFs are hard to support in TQP because of data conversion and data representation mismatches, Spark vectorized UDFs [19] can be supported on CPU.

corresponding tensor operations, e.g., $*$ to `torch.mul`. To implement an expression with tensor operations, TQP then performs a post-order DFS traversal on the expression tree. For each node it encounters in the expression tree, TQP evaluates its subtrees recursively. For each leaf node, TQP fetches (or generates) the proper column-tensor (constant value). For each internal operator, TQP retrieves the corresponding tensor operation (or a series of tensor operations) from the internal dictionary. In this way (and with the help of Python lambda functions), TQP generates a chain of tensor operations representing the evaluation of the expressions. As an example, from Q21 in TPC-H, the expressions O_ORDERSTATUS = 'F' AND RECEIPTDATE > L_COMMITDATE is implemented as `torch.logical_and(torch.eq(o_orderstatus,[70]), torch.gt(l_receiptdate,l_commitdate))`, where `[70]` is a 1x1 tensor storing the ASCII value for the constant '`F`'.

5.4.2 Sort-Based Join

TQP adopts a late materialization strategy for joins, similar to the one commonly used in columnar databases [35, 169]. TQP takes only the columns in the join predicate as input to the join, and the output is a set of pairs of indexes identifying the records for which the join predicate succeeds. The sort-based equi-join algorithm is shown in Algorithm 5.1, where, to simplify the description, we describe the case in which two integer columns are joined. With a few modifications, the algorithm is also able to support non-equi joins, left-semi joins, and outer joins. In the pseudocode, we use the typewriter font (e.g., `bucketize`) to denote tensor operations, and the capital font (e.g., CREATEOUTPUT) to denote class methods. Figure 5.3 further illustrates the algorithm.

First, TQP sorts the join-key columns from each table (lines 1 to 3 in Algorithm 5.1, ❶ in Figure 5.3). When multiple columns are used as join keys, TQP uses radix sort. If the join keys are not integers, e.g., floating point numbers, string, and etc, TQP assigns a unique integer to every unique join key (from both *left* and *right*) so that the corresponding tuples should join if and only if the assigned integers match for two join keys. The details are omitted in the pseudocode for brevity. Then, ❷, TQP builds two histograms for the join keys

Algorithm 5.1 Sort-Based Join Implementation in TQP.

Input: *data*: input columns passed as an array of tensors.
Output: an array of tensors representing the join output.

1: $left, right \leftarrow$ GETJOINKEYCOLUMNS($data$)
▷ Sort join keys
2: $left, leftIdx \leftarrow$ `sort`($left$)
3: $right, rightIdx \leftarrow$ `sort`($right$)
▷ Build histograms for the left and right key columns
4: $leftHist, rightHist \leftarrow$ `bincount`($left$), `bincount`($right$)
▷ Compute the number of rows for each pair of matching keys
5: $histMul \leftarrow$ `mul`($leftHist, rightHist$)
▷ Compute the prefix sums of histograms
6: $cumLeftHist \leftarrow$ `cumsum`($leftHist, dim = 0$)
7: $cumRightHist \leftarrow$ `cumsum`($rightHist, dim = 0$)
8: $cumHistMul \leftarrow$ `cumsum`($histMul, dim = 0$)
▷ Initialize the output size and output offsets
9: $outSize \leftarrow cumHistMul[-1]$
10: $offset \leftarrow$ `arange`($outSize$)
▷ Find the bucket of matching keys to which each output belongs
11: $outBucket \leftarrow$ `bucketize`($offset, cumHistMul$)
▷ Compute the indexes from left and right in the join output
12: $offset$.`sub_`($cumHisMul[outBucket] - histMul[outBucket]$)
13: $leftOutIdx \leftarrow leftIdx[cumLeftHist[outBucket] - leftHist[outBucket] + \texttt{div}(offset, rightHist[outBucket], rounding = \text{“}floor\text{”})]$
14: $rightOutIdx \leftarrow rightIdx[cumRightHist[outBucket] - rightHist[outBucket] + \texttt{remainder}(offset, rightHist[outBucket])]$
15: **return** CREATEOUTPUT($data$, $leftOutIdx$, $rightOutIdx$)

from *left* and *right*, respectively, i.e., TQP counts the number of occurrences for each unique join key (line 4). Then, ❸ by multiplying the values (element-wise) of the histograms (line 5), TQP computes the bucket sizes: the number of output rows for each matching join key from *left* and *right*. Afterward, TQP computes the prefix sums for the *left* and *right* histograms (❹), as well as their element-wise multiplication (❺) (lines 6 to 8). The prefix sums will be used later to retrieve, from each join output, the position in *left* and *right*. The total size of the output of the join is then computed as the last element of the prefix sum containing the bucket sizes (line 9), and ❻ TQP generates an index array (*offset*) of the same size

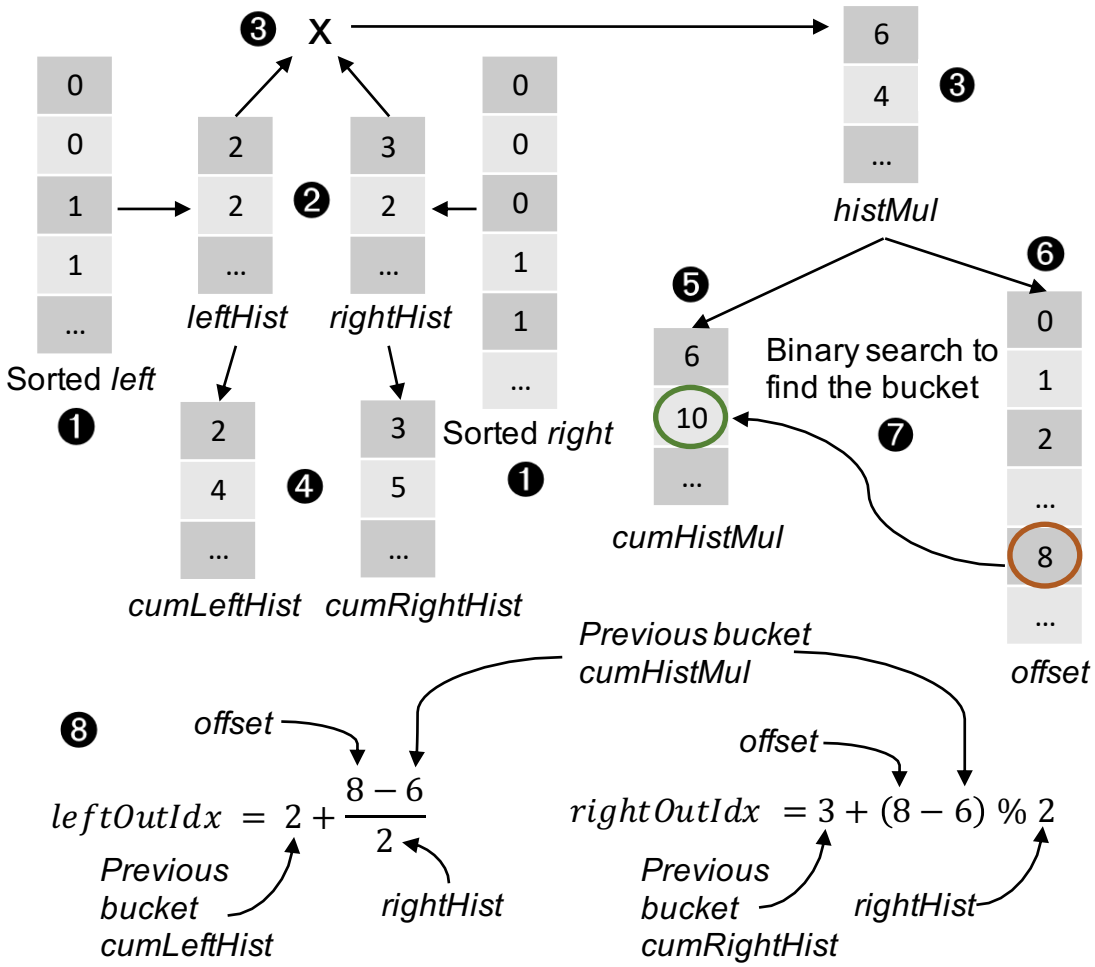


Figure 5.3: An example of the sort-based join implementation in TQP.

(line 10). Then, ❼ TQP performs a parallel binary search on the prefix sum containing the bucket sizes to find the matching join key (bucket) to which each row in the output of the join belongs (line 11). Next, ❽ TQP computes the indexes from *left* and *right* that generate each row in the output of the join. Figure 5.3 shows the computation process for row 8 in the join output of the example. To compute the indexes from *left* and *right* that are part of a given offset in the output of the join, TQP first subtracts *offset* by the prefix sum of bucket sizes prior to the current bucket (line 12). Now *offset* becomes the offset in each bucket of the matching join keys. TQP then adds to the *offset* the previous bucket from the

respective prefix sum histogram (*cumLeftHist* and *cumRightHist*, respectively), and adds the result (quotient for *leftOutIdx*, remainder for *rightOutIdx*) of *offset* divided by the number of join keys from *right* in the current bucket of matching join keys (lines 13 to 14). Finally, for each row in the join output, TQP knows which rows from *left* and *right* contributed to it. It then generates the join output (line 15, not depicted in Figure 5.3). It is important to note that all computations in this join implementation are achieved using tensor operations, with only minimal usage of Python code.

5.4.3 Hash-Based Join

The hash equi-join algorithm is shown in Algorithm 5.2. The definition of the input and output here is the same as in Section 5.4.2. The algorithm is similar to the classical hash join algorithm, except that the build and probe phases are interleaved and repeated as many times as the maximum number of elements that share a hash value (line 6). The algorithm is as follows: TQP first generates the indexes (line 2) and the hash values (line 3) for the *left* and *right* tables. Afterward, TQP computes a histogram over the table on which the hash table will be built (*left* in this case, line 4) and checks the maximum number of elements in a hash bucket (line 5). Then, TQP repeatedly builds a hash table (lines 7 and 8) and probes it (lines 11 to 14) to find matching keys (lines 15 to 17). Matching keys are accumulated across iterations (lines 18 and 19). In each iteration, TQP also keeps track of the indexes that are stored in the hash table such that they will not appear in subsequent iterations (lines 9 and 10). To achieve this, let m be the hash table size; TQP appends an additional $(m + 1)$-th bucket to the hash table and uses it to redirect the already scattered indexes. Note that when there are no hash collisions, TQP skips the logic of lines 9 to 10 and 18 to 19. This path is therefore close to the optimal.

Compared to the sort-based join, when there are no hash collisions, this implementation is around 30% to 50% faster on CPU and 2× faster on GPU. When there are hash collisions, it is faster than the sort-based join for cases in which at most around 15 elements share a hash value; when there are more than 15 elements sharing a hash value, the sort-based join is

Algorithm 5.2 Hash-Based Join Implementation in TQP.

Input: *data*: input columns passed as an array of tensors.
Output: an array of tensors representing the join output.

1: $left, right \leftarrow$ GETJOINKEYCOLUMNS($data$)
2: $leftIdx, rightIdx \leftarrow$ `arange`($left.shape[0]$), `arange`($right.shape[0]$)
▷ Compute the hash values for join keys (m is the max hash table size)
3: $leftHash, rightHash \leftarrow$ `remainder`($left, m$), `remainder`($right, m$)
▷ Build the histogram of hash values for the left join keys
4: $hashBincount \leftarrow$ `bincount`($leftHash$)
5: $maxHashBucketSize \leftarrow$ `max`($hashBincount$)
▷ Build and probe the hash table in an interleaved way
6: **for** $i \in range(maxHashBucketSize)$ **do**
7: $hashTable \leftarrow$ `full`($(m+1,), -1$)
8: $hashTable$.`scatter_`$(0, leftHash, leftIdx)$
▷ Skip those scattered for future iterations by setting their hashes to m
9: $leftIdxSct \leftarrow$ `masked_select`($hashTable, hashTable \geq 0$)
10: $leftHash[leftIdxSct] \leftarrow m$
▷ Probe the current hash table and get the left and right indexes
11: $leftCandIdx \leftarrow hashTable[rightHash]$
12: $validKeyMask \leftarrow leftCandIdx \geq 0$
13: $validLeftIdx \leftarrow$ `masked_select`($leftCandIdx, validKeyMask$)
14: $validRightIdx \leftarrow$ `masked_select`($rightIdx, validKeyMask$)
▷ Find the indexes that have matching join keys
15: $matchMask \leftarrow left[validLeftIdx] == right[validRightIdx]$
16: $leftMatchIdx \leftarrow$ `masked_select`($validleftIdx, matchMask$)
17: $rightMatchIdx \leftarrow$ `masked_select`($validrightIdx, matchMask$)
▷ Append the indexes to the global results
18: $leftOutIdx \leftarrow$ `cat`($(leftOutIdx, leftMatchIdx)$)
19: $rightOutIdx \leftarrow$ `cat`($(rightOutIdx, rightMatchIdx)$)
20: **return** CREATEOUTPUT($data, leftOutIdx, rightOutIdx$)

faster. We are currently working on a partitioned hash-join implementation.

5.4.4 Aggregation

Algorithm 5.3 shows the pseudocode of the aggregation implementation. First, TQP horizontally concatenates the values of the group-by columns (lines 1 and 2). TQP then sorts the values of the concatenated columns using radix sort and permutes all the input data columns

Algorithm 5.3 Aggregation Implementation in TQP.

Input: *data*: input columns passed as an array of tensors.
Output: the aggregation output as an array of tensors.

1: $grpByCols \leftarrow$ GETGROUPBYCOLUMNS($data$)
▷ Generate unique groups
2: $grps \leftarrow$ `cat`($grpByCols, dim = 1$)
3: $grps, grpsInvIdx \leftarrow$ `sort`($grps$)
4: $data \leftarrow [col[grpsInvIdx]\ for\ col\ in\ data]$
5: $grpsUnique, invIdxs \leftarrow$ `uniqueConsecutive`($grps, inverse = True$)
▷ Evaluate the aggregation expression
6: **return** [EVALUATE($data, grpsUnique, invIdxs$)]

according to this sorted order (lines 3 and 4). Using `uniqueConsecutive`, TQP eliminates all but the first key from every consecutive group of equivalent keys. Concurrently, TQP computes the inverted indexes that indicate in which bucket (identified by a unique key) each row in the sorted list ends up (line 5). Finally, with the unique key list and inverted indexes, TQP evaluates the aggregate expression for all groups. This last operation makes use of the expression generated (at compile time) as described in Section 5.4.1.

5.5 Evaluation

Key questions. The evaluation aims to answer the following key questions:

- On CPU, is TQP's performance comparable to other data processing systems on a single core (Section 5.5.1)?
- On GPU, is TQP's performance comparable to other GPU databases (Section 5.5.2)?
- How well does TQP scale with the increase in the number of CPU cores and dataset sizes (Section 5.5.3)?
- What is the cost/performance tradeoff of TQP on GPU (Section 5.5.4)?
- Which operation takes the most time in query execution (Section 5.5.5)?
- Can hand-optimized query plans improve TQP's query time (Section 5.5.6)?
- Can TQP accelerate workloads mixing ML and relational queries (Section 5.5.7)?

- What are the overheads (Section 5.5.8)?
- Can TQP run over different hardware and software backends while minimizing the engineering effort (Section 5.5.9 and Section 5.5.10)?

Baseline systems. Our goal is to compare TQP with state-of-the-art query processing systems for different hardware settings. Specifically, for CPU execution, we compare TQP with Apache Spark [282] (recall that Spark and TQP share the same query plans) and DuckDB [224]: a state-of-the-art vectorized engine. For GPU execution, we compare TQP with two well-known open-source GPU databases: BlazingSQL [3] and OmnisciDB [7].

Hardware and software setup. For all the experiments (except when noted otherwise), we use an Azure NC6 v2 machine with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA P100 GPU (with 16 GB of memory). The machine runs Ubuntu 18.04 with PyTorch 1.11, torch-scatter 2.0.9, BlazingSQL 21.8.1, PySpark 3.1.1, OmnisciDB 5.9.0, DuckDB 0.4.0, RAPIDS 21.08, CUDA 10.2, TVM 0.8 and scikit-learn 0.21.3.

Experimental setup. We use the TPC-H benchmark [79] which consists of 22 queries. We use the parameters specified in the query validation sections in [79]. We generate data at different scale factors from 1 to 10 where 1 means 1 GB of input data in total using the dbgen tool from the TPC-H benchmark. Note that some queries can run on scale factors larger than 10 in GPUs, thanks to TQP's ability to push projections into data conversion. We are working on supporting out-of-memory computation by leveraging PyTorch's DataLoader [21]. We load the generated data from disk into Pandas dataframes. All dataframes use the data types as specified in the benchmark, except for decimals: we use doubles for all systems since TQP does not support decimals yet. Subsequently, we register/convert each dataframe into each system's internal format, e.g., Spark dataframes for Spark[4], PyTorch tensors for TQP, CUDA dataframes for BlazingSQL, etc., and move the data to the GPU, when applicable. We measure the total query execution time, including the time for generating the output.

[4]For Spark, we additionally load the working datasets in memory using `cache`.

For each experiment, we do 10 runs where the first 5 are for warm-up. The reported numbers are median values of the last 5 runs.

Key takeaways. The key evaluation results are as follows:

- TQP's query execution time on CPU using a single core is better than Spark's over the same physical plans.
- TQP's scalability on CPU is poor because of PyTorch lacking parallelization in some operators' implementation and its intra-operator parallelism model.
- TQP is, in general, slower than DuckDB on CPU, but for several queries, TQP is comparable or even better.
- Hand-optimized plans can improve TQP's performance, which suggests that a TCR-aware query optimizer is required to achieve the best performance.
- TQP's query execution time on GPU is usually better than both BlazingSQL's and OmnisciDB's, and TQP supports more queries in TPC-H than they do.
- When ML model prediction and SQL queries are mixed together, TQP is able to provide end-to-end acceleration which delivers up to 9× performance improvement over CPU baselines.
- TQP on GPU performs favorably, and the query time speedup justifies the dollar cost increase compared to CPU-only systems.
- TQP can run queries on different hardware and software backends (including integrated GPUs and web browsers), with orders of magnitude fewer lines of code required, compared with the baseline systems.

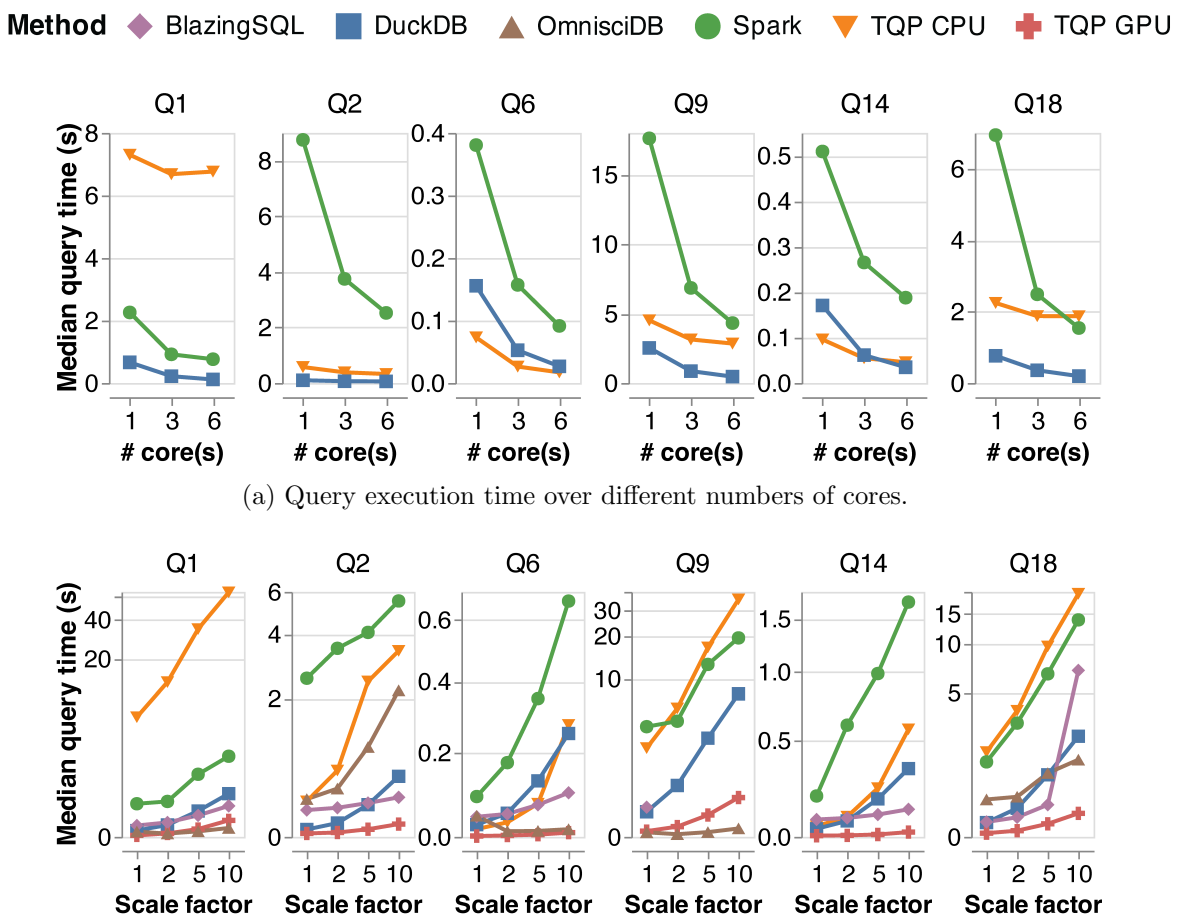
5.5.1 Single Core Execution on CPU

In this first experiment, we use a single CPU core and TPC-H at scale factor 1. The results are shown in Table 5.2 (under CPU). We compare Spark and DuckDB vs. TQP, using both interpreted (TQP) and compiled execution with TorchScript (TQPJ). Spark, DuckDB, and TQP can support all 22 queries.

Table 5.2: Query execution time (in seconds) on the TPC-H benchmark (scale factor 1). Bold numbers highlight the best performance for the specific setup (CPU or GPU). We evaluate TQP in two modalities: interpreted (TQP) and compiled using TorchScript (TQPJ). N/A means the query execution did not finish because of an error. TQPJ currently does not support materialized views.

Query	CPU (1 core)				GPU			
	Spark	DuckDB	TQP	TQPJ	Blazing	Omnisci	TQP	TQPJ
Q1	2.261	**0.664**	7.535	7.301	0.216	0.095	0.027	**0.026**
Q2	8.751	**0.101**	0.629	0.577	0.238	0.351	0.039	**0.028**
Q3	3.669	**0.273**	1.154	1.165	0.128	0.293	0.027	**0.024**
Q4	4.719	**0.216**	1.050	1.087	0.093	0.292	0.020	**0.018**
Q5	6.963	**0.302**	2.459	2.963	0.164	0.064	0.048	**0.042**
Q6	0.381	0.156	0.143	**0.073**	0.045	0.047	0.003	**0.002**
Q7	5.569	**0.430**	2.236	1.931	0.244	0.067	0.042	**0.035**
Q8	4.034	**0.278**	2.460	2.503	0.215	0.079	0.050	**0.039**
Q9	17.61	**2.533**	4.518	4.616	0.569	**0.072**	0.105	0.092
Q10	15.98	**0.430**	1.168	1.184	0.173	0.740	0.057	**0.052**
Q11	1.047	**0.034**	0.476	0.324	N/A	0.084	0.016	**0.009**
Q12	4.063	**0.309**	0.976	0.966	0.069	0.062	0.025	**0.021**
Q13	6.081	**0.181**	9.379	9.197	0.303	**0.069**	0.153	0.136
Q14	0.509	0.171	0.124	**0.096**	0.076	N/A	0.007	**0.005**
Q15	2.640	0.291	**0.133**	N/A	N/A	**0.086**	0.129	N/A
Q16	16.94	**0.093**	3.664	3.699	N/A	3.689	0.320	**0.301**
Q17	3.165	**0.381**	2.303	2.466	0.121	0.132	0.061	**0.051**
Q18	6.942	**0.765**	2.245	2.406	0.204	0.593	0.053	**0.048**
Q19	2.300	**0.419**	1.577	1.316	0.188	0.058	0.042	**0.036**
Q20	4.232	**0.276**	2.032	1.975	0.149	N/A	0.048	**0.041**
Q21	12.39	**0.932**	25.49	24.25	N/A	N/A	0.158	**0.151**
Q22	3.919	**0.069**	0.315	0.296	N/A	N/A	0.011	**0.010**

In terms of query time, TQPJ is either comparable to TQP or better. This is because TorchScript removes Python code dependency and provides optimizations not offered by vanilla PyTorch [91]. TQP outperforms Spark for most queries, sometimes by an order of magnitude (e.g., Q10, Q15, and Q22). Given that TQP uses the same physical plans



(a) Query execution time over different numbers of cores.

(b) Query execution time over different scale factors.

Figure 5.4: Scalability on selected queries from TPC-H. For TQP, we report the best time of the interpreted (PyTorch) and compiled (TorchScript) versions. In (a), the scale factor is 1. In (b), all CPU methods use 6 cores. BlazingSQL throws errors for Q9 at scale factors 2, 5, and 10. OmnisciDB does not support Q14. The y-axes in (b) are in (symmetric) log scale.

as Spark, this suggests that the tensor abstraction is indeed good for executing relational queries. The practical reasons are: (1) TQP is column-oriented, while Spark is row-oriented. This makes the former better suited for analytical queries; (2) some tensor operations use SIMD instructions, while Spark does not exploit vectorization; (3) in TQP, tensor operations are implemented in C++, while Spark is Java-based; (4) Spark is designed as a scale-out

system. For queries (i.e., Q1, Q13, and Q21) where TQP is slower than Spark, the reasons are: (1) TQP's left anti-join and left outer-join implementations are not optimized; (2) the performance of the `uniqueConsecutive` operator in PyTorch is not optimal. Finally, we compare TQP against DuckDB, a state-of-the-art vectorized execution engine. TQP has better performance than DuckDB only for 3 queries. For the other queries, DuckDB clearly outperforms TQP. If we exclude Q1, Q13, and Q21 (discussed above), TQP's query times are within the same order of magnitude as DuckDB's. To evaluate whether this poor performance compared with DuckDB is due to bad query plans or the tensor abstraction, we hand-code better query plans and tensor programs in Section 5.5.6 and show that TQP can match and even outperform DuckDB on CPU.

5.5.2 Execution on GPU

In this experiment, we evaluate the performance of TQP on GPU. The results are shown in Table 5.2 (under GPU). Starting from TQP vs. TQPJ, as in the CPU case, TQPJ outperforms TQP. Compared with the baselines, TQP (interpreted or compiled) outperforms BlazingSQL (Blazing in the table) for all the queries, and it outperforms OmnisciDB (Omnisci in the table) on 15 queries out of the 18 queries supported by OmnisciDB. For the remaining 3 queries, TQP achieves query times within a factor of 2 from OmnisciDB. Note that TQP supports all 22 TPC-H queries, while BlazingSQL and OmnisciDB only support 17 and 18 queries, respectively.

Finally, if we compare the best CPU performance versus the best GPU ones, in general, we see that the query times on GPU are 1.5× to 48× better than the CPU ones (single core), except for Q16 where DuckDB is about 3× faster than the best-performing GPU system. This somehow counter-intuitive result is due to the fact that, at scale factor 1, GPU resources are not completely saturated. Therefore, it makes sense to explore how these systems scale with more data and more available core. This is what we explore next.

5.5.3 Scalability

For this and the following experiments, we select a representative set of queries: complex aggregation (Q1), joins and filters (Q2), simple filters (Q6), complex joins (Q9), simple join and aggregation (Q14), a complex mix of join, aggregation, and sub-queries (Q18).

Scaling the Number of Cores

In this experiment, we scale the number of available CPU cores from 1 to 6 over TPC-H at scale factor 1. Figure 5.4a compares the scaling performance of Spark, DuckDB, and TQP. Spark has the best scalability trend lines almost for all queries. DuckDB also scales well. TQP's scaling performance is, however sub-optimal, and for some queries increasing the number of cores provides no benefits. There are two reasons: (1) PyTorch uses intra-operator parallelism , which is not as efficient as the shuffle [282] or morsel-based [166] approaches in Spark and DuckDB, respectively; (2) some PyTorch operators run on a single core (e.g., `torch.unique()` and `torch.unique_consecutive()` [223] used in aggregation). We are investigating how to overcome this limitation by adding data-parallel support to TQP leveraging PyTorch Distributed Data Parallel [167, 26] or by adding parallel operator implementations.

Scaling the Data

In this experiment, we scale the dataset from 1 GB to 10 GB. In Figure 5.4b, we compare the scalability performance of CPU implementations running over 6 cores (Spark, DuckDB), as well as GPU systems (BlazingSQL and OmnisciDB). In general, we see that TQP CPU scales the worst for almost all queries (only Spark is worst for Q6 and Q14), while GPU systems scale better than the CPU ones. For Q1, OmnisciDB provides the best performance, followed by TQP GPU. For Q2, Q14, and Q18, TQP GPU has the best performance, while for Q6, TQP GPU is comparable to OmnisciDB. Finally, for Q9, OmnisciDB has the best performance. Q9 has six joins, and OmnisciDB is able to better use the GPU resources. This

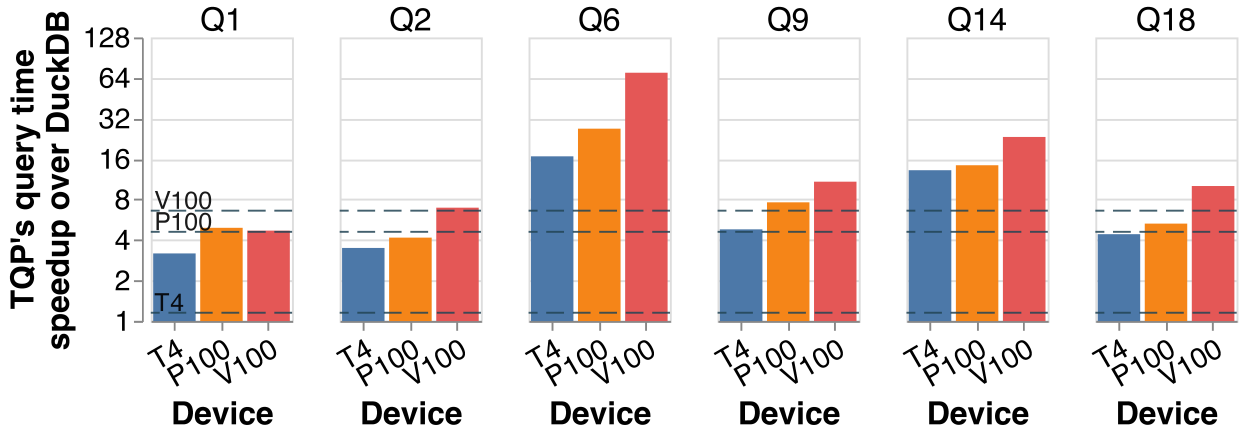
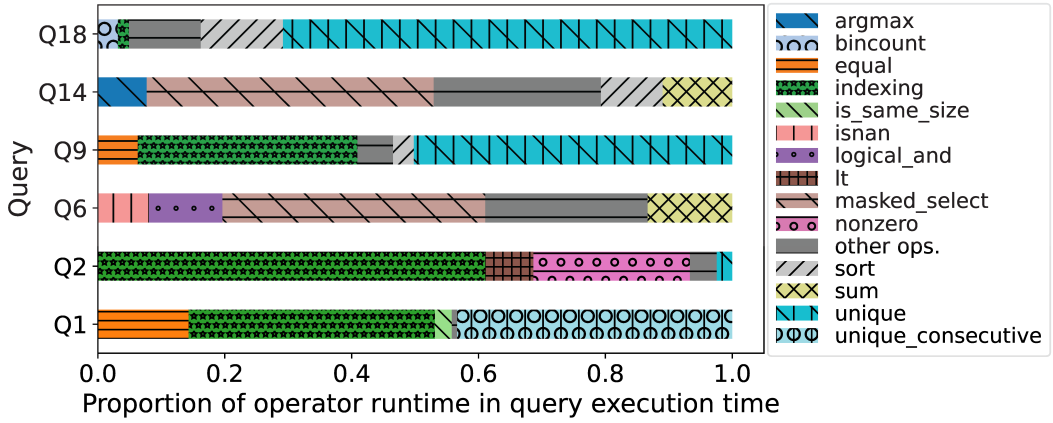


Figure 5.5: Cost/performance tradeoff for TQP on selected queries at scale factor 10. We plot the speedups of TQP on various GPUs (NVIDIA T4, P100 and V100) over DuckDB on a baseline CPU-only machine. The dashed lines represent the query time speedups required by the GPU executions to be more cost-effective compared to the DuckDB CPU baseline.

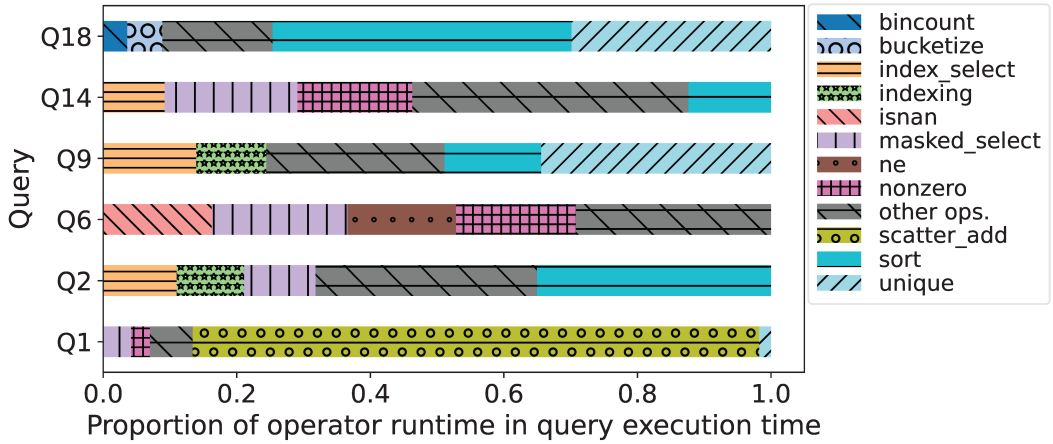
query is memory-bound, and the memory bandwidth of the P100 makes it much faster on GPU than on CPU.

5.5.4 Cost/Performance Tradeoff

We now provide a cost/performance analysis of TQP on GPU compared to a CPU-only baseline. Specifically, we select a general-purpose (CPU-only) VM in Azure with a dollar cost similar to the cheapest VM equipped with GPU (NC4as_T4_v3), and with a similar main memory size. Following these constraints, we select a D2ds_v5 with 8 CPU cores and 32 GB of memory. Then we compare the performance of DuckDB on the D2ds_v5 with TQP on (1) NC4as_T4_v3 (with an NVIDIA T4 GPU, about 15% more expensive than the CPU-only machine), (2) NC6s_v2 (with an NVIDIA P100, around 4.6× more expensive than the CPU-only VM), and (3) NC6s_v3 (with an NVIDIA V100, around 6.6× more expensive than the CPU-only VM). For each GPU VM type, we show the query time speedup required to be more cost-effective than the DuckDB baseline. That is, for the T4, the speedup provided by TQP has to be more than 15% to justify the cost increase of the T4 VM compared to the



(a) Query time breakdown for tensor operators on CPU



(b) Query time breakdown for tensor operators on GPU

Figure 5.6: Query time breakdown for tensor operators for selected TPC-H queries at scale factor 10.

DuckDB CPU baseline, 4.6× for the P100, 6.6× for the V100. The results for scale factor 10 are shown in Figure 5.5 for a few representative TPC-H queries. As shown, TQP on GPU is more cost-effective compared to DuckDB on the CPU-only machine: for 6 of the 6 selected queries (17 of the 21 supported queries[5] in the full TPC-H) for the T4; 5 of 6 (10 of 21 in the full TPC-H) for the P100; and 5 of 6 (9 of 21 in the full TPC-H) for the V100.

[5]OOM errors occurred when TQP ran Q21 at scale factor 10 on these GPUs.

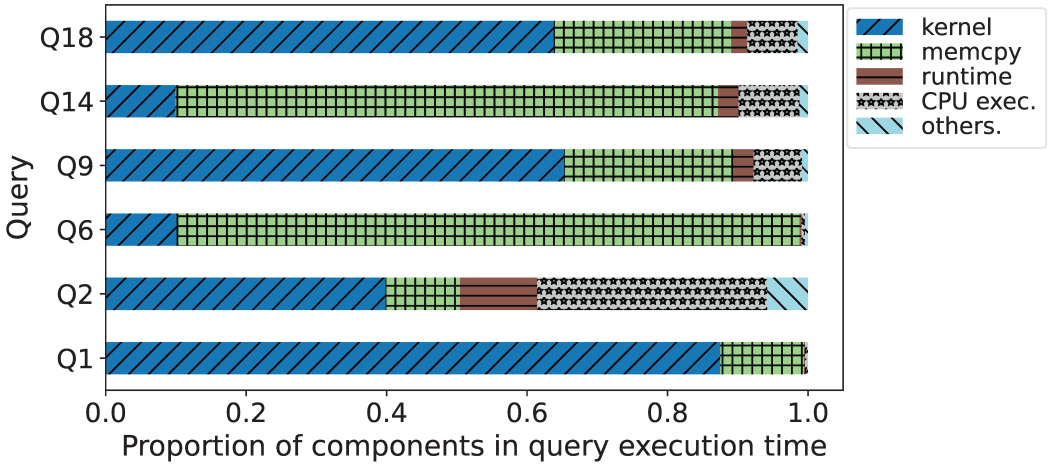


Figure 5.7: GPU utilization breakdown for selected TPC-H queries at scale factor 10. Utilization varies by query. Runtime is the time spent in scheduling the kernels.

5.5.5 Performance Breakdown

In this experiment, we show the major contributing factors to the query execution time. TQP is integrated with TensorBoard [14], which provides performance breakdowns and makes it easy to spot bottlenecks [48]. We start by looking into which tensor operators are responsible for the majority of the execution time. Figures 5.6a and 5.6b show the breakdown for a few selected queries on CPU and GPU, respectively. Interestingly, even if TQP uses the same algorithms on both CPU and GPU, the same query can show different operator contributions. For example, for Q1 on CPU, most of the time is spent on computing the unique elements, while on GPU, most is spent on scatter_add. This is because the quality of the operator implementations is different for CPU and GPU. Across queries, on CPU and GPU, the majority of time is also spent on different operators. On CPU, most queries are bounded by unique operators, masked_select, and indexing; on GPU, most of the time is spent on sorting, unique and nonzero. These observations suggest that: (1) the quality of kernels differs between CPU and GPU, e.g., after further investigation, we find that the GPU implementation of scatter_add is not optimal, and nonzero requires host/device synchronization [29] (however, we believe that over time the community will fix such performance issues); and (2) it might

be worth investigating backend-aware tensor algorithms.

Finally, we report the GPU utilization for the same set of queries in Figure 5.7. As we can see, each query has different utilization characteristics. For instance, Q1 contains complex aggregation, and it spends 87% of the time on kernel execution; conversely, Q6 and Q14 are simple queries, and most of the time is spent allocating GPU memory. Finally, Q2 spends a considerable amount of time in generating the output on CPU.

5.5.6 Hand-Optimized Plans

Table 5.3: Query execution time (in seconds) on selected TPC-H queries (scale factor 10). TQP Hand-Opt. uses hand-optimized tensor programs. We use Torch, JIT, and TVM to refer to execution using PyTorch (interpreted), TorchScript (compiled), and TVM, respectively. Bold numbers highlight the best performance for the specific setup: CPU (1 core), CPU (6 cores), or GPU.

TPC-H Query	CPU (1 core)				CPU (6 cores)				GPU			
	Best Baseline	TQP Hand-Opt.			Best Baseline	TQP Hand-Opt.			Best Baseline	TQP Hand-Opt.		
		Torch	JIT	TVM		Torch	JIT	TVM		Torch	JIT	TVM
Q1	6.54 (DuckDB)	**5.97**	6.89	N/A	**1.1** (DuckDB)	4.68	5.17	N/A	0.17 (OmnisciDB)	**0.13**	0.13	N/A
Q6	1.5 (DuckDB)	0.87	1.18	**0.24**	0.25 (DuckDB)	0.66	0.71	**0.12**	0.02 (OmnisciDB)	**0.01**	0.01	0.06
Q9	45.11 (DuckDB)	19.34	**18.66**	N/A	**7.75** (DuckDB)	14.59	13.83	N/A	**0.14** (OmnisciDB)	0.45	0.44	N/A
Q14	1.7 (DuckDB)	0.52	0.49	**0.47**	0.33 (DuckDB)	0.12	**0.10**	0.16	0.12 (BlazingSQL)	0.01	**0.01**	0.30

Next, we study whether TQP's performance can be improved with a better optimizer that can generate better tensor programs. To understand this, we hand-optimize the tensor programs for a few selected queries similarly to what a reasonable optimizer with knowledge about cardinalities and tensor characteristics would do, e.g., avoid sorting (or computing unique values) over already sorted (or made unique) columns, and select better join implementations. The results are shown in Table 5.3, where we report the best baseline for each setting (CPU 1 and 6 cores, and GPU), and over three execution modes: interpreted PyTorch (Torch), compiled TorchScript (JIT), and compiled using TVM. TVM only supports Q6 and Q14.

If we focus on the CPU numbers first, TQP's performance is comparable to or even better than that of DuckDB's, while TQP was much slower compared to DuckDB both on single- and multi-core execution when not using the hand-optimized plans. TQP is now faster than DuckDB for all queries over 1 CPU core, and two queries over 6 CPU cores. For some queries, TQP is faster than DuckDB by a large margin, e.g., for Q6, 1-core TVM execution is 6× faster. This is because TVM uses code generation and operator fusion to minimize intermediate data materialization across operators. When scaling to 6 cores, TQP scales well only for Q14, while DuckDB scales linearly. For the other queries, TQP's query times improve by at most 2×. This again shows the limitations of PyTorch's scalability on CPU, which cannot be improved by only using better tensor programs.

Finally, on GPU, we see that OmnisciDB has still better performance for Q9, although TQP's query time for Q9 on GPU improves by 4×, when using the hand-optimized plans. This is because TQP's aggregate implementation heavily uses sorting, while OmnisciDB uses hash-based implementations.

5.5.7 Prediction Queries

We now investigate the performance benefits of using a unified runtime for queries mixing relational and ML operators. We use *prediction queries* as a use case, i.e., queries embedding a trained ML model performing predictions over some input data [188]. Recall that TQP natively supports predictions of any PyTorch model (e.g., NNs), and traditional ML models through its integration with Hummingbird [195]. Here, we join the CUSTOMER and ORDERS tables in TPC-H (scale factor 10), and train a gradient boosting tree model (with 128 trees with max depths of 8) over a mix of categorical (C_ORDERSTATUS) and numerical features (C_CUSTKEY, C_NATIONKEY, C_ACCTBAL, SUM(O_TOTALPRICE)) after we apply one-hot encoding and feature scaling, respectively. We run a prediction query using the trained model over the query with two filter predicates added (C_MKTSEGMENT = 'BUILDING' AND O_ORDERDATE >= DATE '1993-10-01'). Note that this prediction query mixes ML operators (tree ensemble, one-hot encoding, scaling, and concatenation) with relational ones

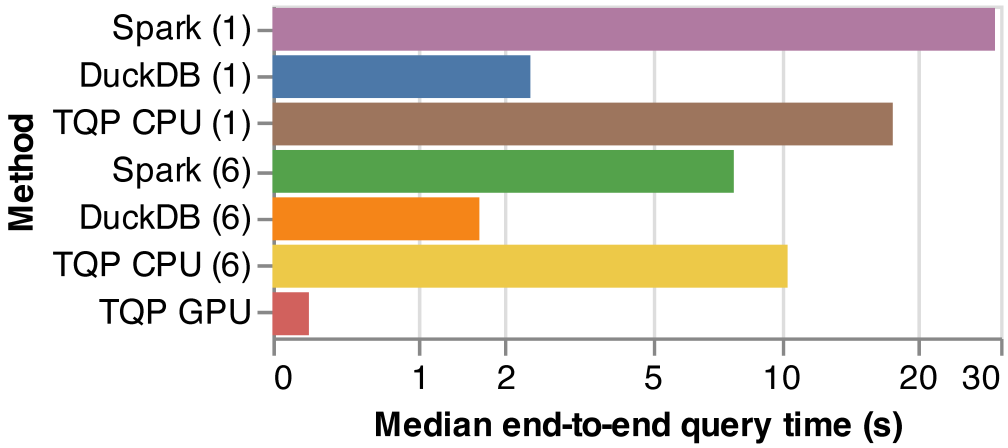


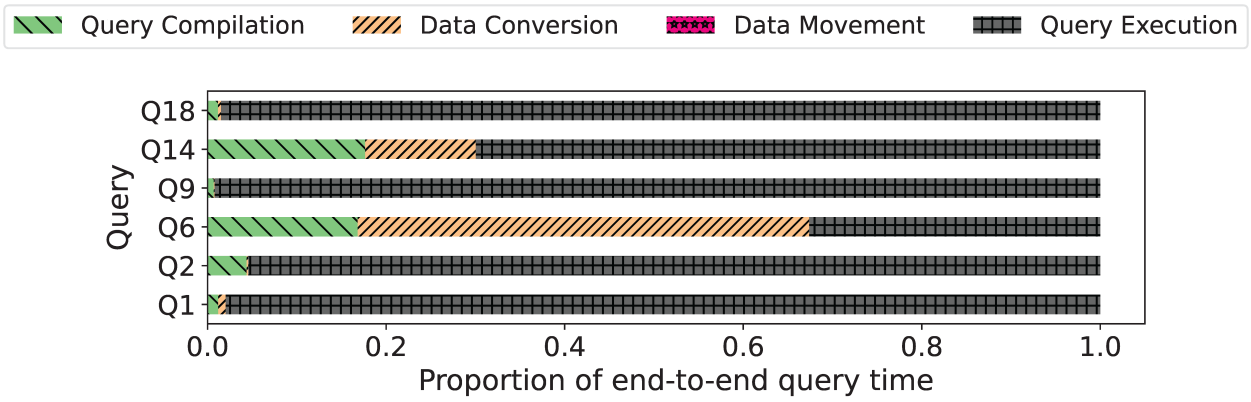
Figure 5.8: Query time on a query mixing ML model prediction and relational operators. In parenthesis shows the number of CPU cores. The x-axis is in (symmetric) log scale.

(join, aggregation and filtering). We compare TQP with two baselines: one where the prediction query is executed over Spark (MLlib [184] is used to build the model), and one where we use DuckDB for the relational part and scikit-learn [212] for the ML part[6]. Since TQP subsumes Hummingbird, it is able to compile both the ML and the relational operators of the query into a unified plan executable on TCRs. Figure 5.8 shows the result. For CPU single core, TQP is about 40% faster than Spark, while DuckDB with scikit-learn is about 7× faster than TQP. When enabling all cores, Spark and DuckDB scale much better than TQP, for the reasons described in Section 5.5.3. Finally, TQP is able to exploit GPU acceleration end-to-end, which brings a 9× improvement of query time compared to the best CPU baseline.

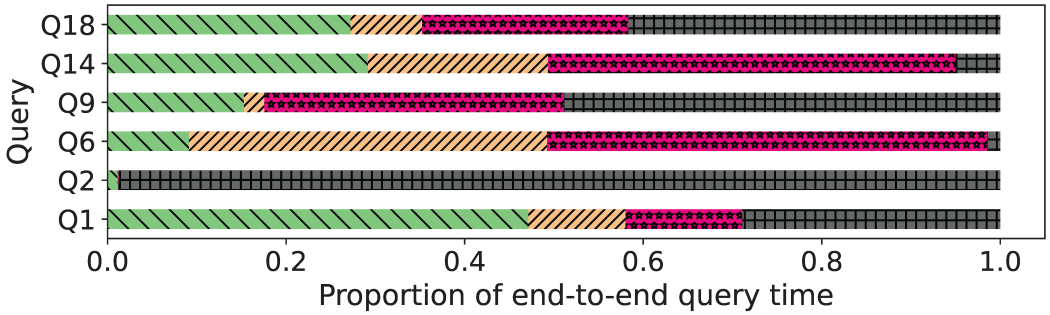
5.5.8 Overheads

Next, we evaluate the overheads of TQP for both CPU and GPU. The breakdown of the end-to-end execution with all overheads is shown in Figure 5.9. Note that: (1) data conversion

[6]Note that moving data from DuckDB to scikit-learn is zero-copy since DuckDB can directly return data in Pandas dataframe format [22].



(a) End-to-end execution breakdown on CPU



(b) End-to-end execution breakdown on GPU

Figure 5.9: End-to-end execution time breakdown (incl. all overheads, and w/o pipelining and caching) for selected queries at scale factor 10.

is done once and many databases (e.g., BlazingSQL, OmnisciDB, Spark, SQL Server, etc.) requires it; (2) TQP pipelines data movement (to the GPU) with query execution (non-blocking IO), while for this experiment we explicitly make data movement blocking; (3) the machine in this experiment uses PCIe 3 which is 4× slower than the latest version, PCIe 5; (4) query compilation can be cached, but here we report the full query compilation time as the sum of the time for the frontend database to generate the physical plan, and the time for TQP to generate the final executable tensor program.

If we focus first on the CPU side (Figure 5.9a), compilation and data conversion take the majority of the time only for simple queries (e.g., Q6), while for the other queries, the

majority of the time is spent on the query execution. However, in the GPU case (Figure 5.9b), except for Q2 and Q9, the majority of the time is spent on data operations (conversion and movement) and compilation. However, in practice, as described above, these overheads are hidden (e.g., data movement using pipelining) or are one-time overheads (data conversion and query compilation). Regarding query compilation, 90% of the time is spent initializing the PyTorch models from the Spark plans, and we are currently investigating how to speed up this process. Finally, using TorchScript adds substantial compilation overheads since queries are traced using input samples.

Table 5.4: Query time (in milliseconds) of TPC-H Query 6 (scale factor 1) using the hand-optimized plan over different hardware and software backends. In parenthesis is the TCR used as well as the compilation stack (when applicable).

Intel UHD Graphics 630 (TVM on Metal)	AMD Radeon Pro 5300M (TVM on Metal)	NVIDIA K80 (PyTorch)	NVIDIA V100 (PyTorch)	TPU (PyTorch on XLA)	Chrome (ORT on WASM)
62	17	5	1	25	1900

5.5.9 Portability

To evaluate whether TQP can run on different hardware and software backends, we run TPC-H Query 6 with the hand-optimized plan on: (1) two integrated graphic cards, one from Intel, and one from AMD; (2) two discrete GPUs from NVIDIA (K80 and V100: the former a generation before the P100 GPU used for the experiments in the previous sections; the latter one, one generation after); (3) a custom ASIC used for NN training and inference (TPU); and (4) a web browser. We use a scale factor of 1. The results are shown in Table 5.4. This experiment proves the versatility of TQP. For the integrated GPUs, we use TVM to code-generate the query using Metal [45]. For the two discrete GPUs, we use vanilla PyTorch,

while for the TPU, we use the XLA backend for PyTorch[7] [221]. Finally, we are able to run the query in the browser by exporting it into the ONNX format and running it in Chrome using ONNX Runtime (ORT) for WebAssembly (WASM) [190].

5.5.10 Engineering Effort

To demonstrate the minimal engineering effort required by TQP to run queries over different hardware, we compare the lines of code for a few relational operators (hash and sort-based joins, aggregation) across all evaluated systems. For each relational operator and each system, we use cloc [81] to count the lines of source code (excluding comment and blank lines) from the files containing the algorithmic functionality of the operator. This is admittedly a subjective process, but we believe the numbers of lines of code can roughly reflect the engineering effort required to implement relational operators in each system. Table 5.5 shows the results. Compared with the baselines, TQP requires significantly lower engineering effort: up to 10× less compared to CPU implementations, and 50× less compared to GPU ones. It is worth noting that TQP is able to target different hardware with the same implementation, so the engineering effort required for TQP to scale over different hardware is constant. The other baseline systems do not share this property. For instance, to run Spark on GPU (e.g., using RAPIDS [13], the same backend of BlazingSQL), we would have to add the lines of code for the GPU implementation.

5.6 Limitations and Discussion

TQP is the first query processor on TCRs, and as such, it currently suffers many limitations. For example, support for NULLs is limited, and data types such as decimals or non-UTF-8 strings are not available. We plan to add better support for NULLs by attaching a validity bitmap to each tensor, as in Apache Arrow [46]. The choice of supporting strings through padding is mostly driven by the lack of support for sparse and jagged tensors in TCRs. There

[7]Note that PyTorch/XLA does not support all the necessary tensor operations and the execution fallback to regular CPU for part of the query is not available.

Table 5.5: Lines of source code for implementing relational operators, excluding blank lines and comments.

System	Relational Operator		
	Hash Join	Sort-Based Join	Aggregation
TQP (Various HW)	148	182	104 (sort-based)
Spark(CPU)	706	1439	637 (sort-based)
DuckDB (CPU)	1415	877	1466 (hash-based)
BlazingSQL (GPU)	1628	N/A	1389 (hash-based)
OmnisciDB (GPU)	10141	N/A	2416 (hash-based)

is research in this space [94], and we plan to support non-dense tensors. We also plan to add the possibility to encode strings.

As we see in Section 5.5, TQP's scalability is currently limited by PyTorch's implementation. Along the same line, PyTorch does not natively support out-of-core computation, and hence TQP is limited to datasets fitting in (device) memory. We are actively working on enabling TQP over batched tensors, which will allow us to scale beyond memory, and enable better parallelization opportunities. Furthermore, from Section 5.5.6, we realize that there is a large gap between the tensor programs currently generated by TQP and the optimal ones. We are extending TQP's optimizer to fill this gap.

Finally, adding support for recursive queries [168, 244, 283] is interesting future work, since iterative programs can be efficiently executed on GPUs [157] as long as they fit in the device memory.

5.7 Summary

In this chapter, we presented TQP, the first system able to run relational queries on TCRs. TQP is able to take advantage of all the innovation poured into TCRs, as well as to run efficiently on any hardware devices supported by TCRs. Our experiments showed not only that TQP is capable of running the full TPC-H benchmark on TCRs, but also that TQP's

performance is comparable and often superior to that of specialized CPU and GPU query processing systems.

Chapter 6

CONCLUSION

The emerging era of AI is bringing significant challenges and opportunities to data management systems. This dissertation considered two core challenges.

First, modern data management systems must efficiently support emerging AI workloads that not only consume vast amounts of data but also generate substantial data artifacts. Traditional data management systems, however, struggle to meet the demands of these new workloads, which include storing, indexing, and querying large-scale input datasets and model artifacts.

Second, due to the tremendous computational and memory demands of AI workloads, the advent of specialized AI infrastructure presents new opportunities to improve the performance of query processing in data management systems at a time when improvements on CPUs have slowed down. Such infrastructure, which runs in data centers, is being increasingly deployed in the cloud and on-premise and is thus readily available for data management systems to use. However, the diversity in hardware characteristics and programming abstractions of AI infrastructure make it difficult for system builders to fully leverage this powerful infrastructure.

This dissertation presented two lines of work intended to bridge the gap between data management systems and AI. First, we introduced two data systems, DeepEverest and MaskSearch, designed to efficiently support AI model explanation and dataset exploration workloads. In Chapter 3, we introduced DeepEverest, a system designed to accelerate neural network explanation queries that return input examples in the dataset with certain neuron activation patterns, i.e., *interpretation by example* queries. These queries facilitate understanding of the functionality of groups of neurons in the neural network by tying

that functionality to input examples in the dataset. DeepEverest consists of an efficient indexing technique and a query execution algorithm with various optimizations to accelerate *interpretation by example* queries. Our evaluation shows that DeepEverest, using less than 20% of the storage of full materialization, significantly accelerates individual queries by up to 63× and consistently outperforms existing systems and approaches on multi-query workloads that simulate neural network interpretation processes. DeepEverest appeared in VLDB 2022 [115] and is available at `https://github.com/uwdb/DeepEverest`.

In Chapter 4, we introduced MaskSearch, a system designed to enable efficient querying over databases of image masks generated by AI tasks; it supports the retrieval of masks with properties that are important for applications, such as identifying spurious correlations, detecting adversarial examples, and monitoring model errors in AI applications. MaskSearch applies a novel indexing technique and an efficient filter-verification query execution framework to accelerate mask search queries. Our experiments show that MaskSearch, using indexes approximately 5% of the compressed data size, accelerates individual queries by up to two orders of magnitude and consistently outperforms existing methods on various multi-query workloads that simulate mask search processes in AI applications. MaskSearch is under submission [117] and is available at `https://github.com/uwdb/MaskSearch`. A demonstration of the system is accepted at VLDB 2024 [272].

We then presented the second line of work in this dissertation, i.e., incorporating AI infrastructure to accelerate relational query processing in data management systems. Chapter 5 introduced the Tensor Query Processor (TQP), the industry's first query processor that compiles SQL queries into tensor programs and executes them on any hardware backend supported by the tensor computation runtime, including CPUs, GPUs, and TPUs. TQP comprises a collection of tensor algorithms that implement relational operators and a compiler stack for transforming SQL queries into tensor programs. We demonstrated TQP's potential to use AI infrastructure to accelerate relational query processing in data management systems, showing its support for the full TPC-H benchmark and its ability to outperform state-of-the-art systems. TQP appeared in VLDB 2022 [116] and its demonstration [47] won

the Best Demo Award at VLDB 2022.

6.1 Future Work Related to DeepEverest *and* MaskSearch

One important direction for future work is to extend DeepEverest and MaskSearch to support more complex and diverse AI model explanation and dataset exploration workloads. For example, MaskSearch now supports mask search queries based only on the count of pixels in specified regions and pixel value distributions. Future work could extend MaskSearch to support more complex queries, such as queries that involve the connectivity of pixels in certain value ranges in the masks (e.g., find saliency maps that have the largest number of clusters of connected pixels with values in a certain range, which could be useful for identifying diffused attention by the model). Future work could also extend DeepEverest to support more complex queries, such as queries involving groups of neurons that span multiple layers of the neural network. It would be interesting to explore how to adapt or extend the indexing and query processing techniques in DeepEverest and MaskSearch to support these more complex queries.

Another potentially valuable direction for future work would be extending DeepEverest and MaskSearch to support explanation workloads for foundation models [68, 37, 257], which have become increasingly prevalent in recent years. It remains to be examined how foundation models could become more transparent and explainable and how database indexing and query processing techniques could be leveraged to support the interpretation of these models, which are orders of magnitude larger than traditional neural networks.

6.2 Future Work Related to TQP

Although TQP has shown promising results in accelerating relational query processing using AI infrastructure, it has limitations that we plan to address in future work, as detailed in Section 5.6. One limitation is its lack of support for certain SQL features, such as NULLs, encoded strings, and recursive queries. Future work could better support these features.

We could further enable TQP to work over batched tensors, which would let it process

larger-than-memory-sized datasets and provide more parallelization opportunities. In addition, we described in Section 5.5.6 the large gap between the tensor programs currently generated by TQP and the optimal tensor programs that could be generated by a human expert. Future work could explore how to bridge this gap, potentially by designing a query optimizer that is aware of the tensor algorithms and hardware backends and thus able to generate more efficient tensor programs.

Another interesting direction for future work is exploring how to efficiently support the full loop of AI model training and inference within a data management system. With TQP, relational operators and AI operators (e.g., ML model prediction operators) could be compiled into a unified abstraction, tensor programs, and executed on the same hardware backend. This opens the possibility of training modern AI models within a data management system that directly ingests data from the database and stores the trained models and artifacts back into the database, with no need to move data between systems. It remains to be investigated how "user-friendly" AI model training could become after being integrated into a data management system and how efficiently data management systems could support multi-modal data processing (e.g., relational data, image/video data, text data, graph data) and multi-stage model training and inference pipelines. Despite some attempts in this direction [100], there remains large potential for future work in this area.

6.3 Final Remarks

This dissertation makes an important contribution towards improving modern data management systems to better support AI workloads and towards leveraging modern AI infrastructure to accelerate relational query processing. The emerging era of AI will bring further challenges and opportunities to data management systems, and there is fertile ground for future exploration and innovation in this changing landscape.

BIBLIOGRAPHY

[1] TensorFlow. https://www.tensorflow.org, 2018.

[2] Tensor-RT. https://developer.nvidia.com/tensorrt, 2019.

[3] BlazingSQL. https://blazingsql.com/, 2020.

[4] Cerebras. https://cerebras.net/, 2020.

[5] Cerebras Software. https://cerebras.net/product/#software, 2020.

[6] GraphCore. https://www.graphcore.ai/, 2020.

[7] OmnisciDB. https://www.omnisci.com/, 2020.

[8] ONNX. https://github.com/onnx/onnx/blob/master/docs/Operators.md, 2020.

[9] Opencl. https://www.khronos.org/opencl/, 2020.

[10] Pytorch Ecosystem. https://pytorch.org/ecosystem/, 2020.

[11] PyTorch Release for IPU. https://medium.com/pytorch/graphcore-announces-production-release-of-pytorch-for-ipu-f1a846de1a2f, 2020.

[12] Sambanova: Massive Models for Everyone. https://sambanova.ai/, 2020.

[13] Spark-RAPIDS. https://nvidia.github.io/spark-rapids/, 2020.

[14] TensorBoard. https://github.com/tensorflow/tensorboard, 2020.

[15] Tensorflow XLA. https://www.tensorflow.org/xla, 2020.

[16] Tesla's data engine and what we should learn from it. https://www.braincreators.com/insights/teslas-data-engine-and-what-we-should-all-learn-from-it, 2020. Accessed: 2023-04-20.

[17] Tidypredict. https://tidypredict.netlify.com/, 2020.

[18] Gpu-accelerated string processing with rapids. https://www.nvidia.com/en-us/on-demand/session/gtcfall20-a21131/, 2021.

[19] Introducing pandas udf for pyspark. https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html, 2021.

[20] Code with eager execution, run with graphs: Optimizing your code with revnet as an example. https://blog.tensorflow.org/2018/08/code-with-eager-execution-run-with-graphs.html, 2022.

[21] Datasets & DataLoaders. https://pytorch.org/tutorials/beginner/basics/data_tutorial.html, 2022.

[22] Efficient sql on pandas with duckdb. https://duckdb.org/2021/05/14/sql-on-pandas.html, 2022.

[23] Intel Extension for PyTorch. https://pytorch.org/tutorials/recipes/recipes/intel_extension_for_pytorch.html, 2022.

[24] Introducing Accelerated PyTorch Training on Mac. https://pytorch.org/blog/introducing-accelerated-pytorch-training-on-mac/, 2022.

[25] ONNX Runtime. https://github.com/microsoft/onnxruntime, 2022.

[26] PyTorch Distributed Overview. https://pytorch.org/tutorials/beginner/dist_overview.html, 2022.

[27] PyTorch for AMD ROCm Platform now available as Python package. https://pytorch.org/blog/pytorch-for-amd-rocm-platform-now-available-as-python-package/, 2022.

[28] Substrait. https://github.com/substrait-io, 2022.

[29] torch.nonzero. https://pytorch.org/docs/stable/generated/torch.nonzero.html, 2022.

[30] TorchScript Documentation. https://pytorch.org/docs/stable/jit.html, 2022.

[31] Meerkat and the path to foundation models as a reliable software abstraction. https://hazyresearch.stanford.edu/blog/2023-03-01-meerkat, 2023. Accessed: 2023-04-20.

[32] Limits - pinecone. https://docs.pinecone.io/docs/limits, 2024. Accessed: 2024-01-22.

[33] Product faq - milvus. https://milvus.io/docs/product_faq.md, 2024. Accessed: 2024-01-22.

[34] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., Hanover, MA, USA, 2013.

[35] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. *The Design and Implementation of Modern Column-Oriented Database Systems*. 2013.

[36] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

[37] AI@Meta. Llama 3 model card. https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md, 2024.

[38] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for top-k queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 495–506. VLDB Endowment, 2007.

[39] Amazon.com. Redshift ml. https://aws.amazon.com/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift.-using-sql-with-amazon-redshift-ml, 2021.

[40] AMD. Rocm. https://rocmdocs.amd.com/en/latest/, 2022.

[41] Saleema Amershi, Max Chickering, Steven M Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. Modeltracker: Redesigning performance analysis tools for machine learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 337–346. ACM, 2015.

[42] Michael R. Anderson, Michael J. Cafarella, Germán Ros, and Thomas F. Wenisch. Physical representation-based predicate optimization for a visual analytics database. *CoRR*, abs/1806.04226, 2018.

[43] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 459–468. IEEE, 2006.

[44] Apple. Apple neural engine. `https://en.wikipedia.org/wiki/Apple_A11#Neural_Engine`, 2022.

[45] Apple. Metal. `https://developer.apple.com/metal/`, 2022.

[46] Apache Arrow. Apache arrow. `https://arrow.apache.org/docs/index.html`, 2022.

[47] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. Share the tensor tea: How databases can leverage the machine learning ecosystem. *Proc. VLDB Endow.*, 15(12):3598–3601, aug 2022.

[48] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. Share the tensor tea: How databases can leverage the machine learning ecosystem. *Proc. VLDB Endow.*, 15(12), 2022.

[49] AWS. Inferentia. `https://aws.amazon.com/machine-learning/inferentia/`, 2022.

[50] Maximilian Bandle and Jana Giceva. Database technology for the masses: Sub-operators as first-class entities. *Proc. VLDB Endow.*, 14(11):2483–2490, 2021.

[51] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 475–486. VLDB Endowment, 2006.

[52] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. Miris: Fast object track queries in video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1907–1921, 2020.

[53] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 6541–6549. IEEE, 2017.

[54] David Bau, Jun-Yan Zhu, Hendrik Strobelt, Agata Lapedriza, Bolei Zhou, and Antonio Torralba. Understanding the Role of Individual Units in a Deep Neural Network. *Proceedings of the National Academy of Sciences*, 117(48):30071–30078, 2020.

[55] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[56] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, pages 221–230, New York, NY, USA, 2018. ACM.

[57] Alexander Behm. Photon: A high-performance query engine for the lakehouse. In *CIDR*. www.cidrdb.org, 2022.

[58] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[59] Deblina Bhattacharjee, Martin Everaert, Mathieu Salzmann, and Sabine Süsstrunk. Estimating image depth in the comics domain. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 2070–2079, January 2022.

[60] Avinash N Bhute and BB Meshram. Content based image indexing and retrieval. *arXiv preprint arXiv:1401.1742*, 2014.

[61] Alceu Bissoto, Michel Fornaciali, Eduardo Valle, and Sandra Avila. (de) constructing bias on skin lesion datasets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[62] Matthias Boehm, Iulian Antonov, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, and Benjamin Rath. Systemds: A declarative machine learning system for the end-to-end data science lifecycle. *CoRR*, abs/1909.02976, 2019.

[63] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, sep 2016.

[64] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[65] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237. www.cidrdb.org, 2005.

[66] Sebastian Breβ, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6):797–822, 2018.

[67] Paul G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. Association for Computing Machinery.

[68] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[69] Francesco Del Buono, Matteo Paganelli, Paolo Sottovia, Matteo Interlandi, and Francesco Guerra. Transforming ML predictive pipelines into SQL with MASQ. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2696–2700. ACM, 2021.

[70] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: Visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 745–747. ACM, 2006.

[71] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[72] Rich Caruana, Hooshang Kangarloo, John David Dionisio, Usha Sinha, and David Johnson. Case-based explanation of non-case-based learning methods. In *Proceedings of the AMIA Symposium*, page 212. AMIA, 1999.

[73] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *J. ACM*, 37(2):200–212, apr 1990.

[74] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *J. ACM*, 37(2):200–212, apr 1990.

[75] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 579–594, Berkeley, CA, USA, 2018. USENIX Association.

[76] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. *Proc. VLDB Endow.*, 12(5):544–556, 2019.

[77] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38:8–20, 2018.

[78] Sam Cohan. Delivering ml products efficiently: The single-node machine learning workflow. https://medium.com/udemy-engineering/delivering-ai-ml-products-efficiently-the-single-node-machine-learning-workflow-bad1389410af, 2021.

[79] Transaction Processing Performance Council. Tpc benchmark h. `http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`, 2018.

[80] Patrick Damme, Marius Birkenbach, Constatinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaž Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pinar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz Wrosz, Aleš Zamuda, Ce Zhang, and Xiao Xiang Zhu. Daphne: An open and extensible system infrastructure for integrated data analysis pipelines. In *12th Annual Conference on Innovative Data Systems Research (CIDR '22)*, 2022.

[81] Albert Danial. cloc: v1.92. https://doi.org/10.5281/zenodo.5760077, 2021.

[82] Abhishek Das, Harsh Agrawal, Larry Zitnick, Devi Parikh, and Dhruv Batra. Human attention in visual question answering: Do humans and deep networks look at the same regions? *Computer Vision and Image Understanding*, 163:90–100, 2017.

[83] DataFromSky. Retail - datafromsky. `https://datafromsky.com/retail/`, 2023.

[84] DataFromSky. Traffic monitoring - datafromsky. `https://datafromsky.com/traffic-monitoring/`, 2023.

[85] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pages 253–262. ACM, 2004.

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

[87] Maureen Daum, Enhao Zhang, Dong He, Magdalena Balazinska, Brandon Haynes, Ranjay Krishna, Apryle Craig, and Aaron Wirsing. Vocal: video organization and interactive compositional analytics. In *12th Annual Conference on Innovative Data Systems Research (CIDR'22)*, 2022.

[88] Maureen Daum, Enhao Zhang, Dong He, Stephen Mussmann, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. Vocalexplore: Pay-as-you-go video data exploration and model building. *Proceedings of the VLDB Endowment*, 16(13):4188–4201, 2023.

[89] Alex J DeGrave, Joseph D Janizek, and Su-In Lee. Ai for radiographic covid-19 detection selects shortcuts over signal. *Nature Machine Intelligence*, 3(7):610–619, 2021.

[90] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[91] Zachary DeVito. Torchscript: Optimized execution of pytorch programs. `https://program-transformations.github.io/slides/pytorch_neurips.pdf`, 2019.

[92] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[93] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. In *Proceedings of Machine Learning and Systems*, volume 3, pages 38–54, 2021.

[94] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding. *CoRR*, abs/2110.10221, 2021.

[95] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336, New York, NY, USA, 2012. ACM.

[96] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, et al. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[97] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Qian Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the qbic system. *Computer*, 28(9):23–32, 1995.

[98] .NET Foundation. Torchsharp - pytorch .net bindings. `https://github.com/dotnet/TorchSharp`, 2020.

[99] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1603–1618, New York, NY, USA, 2018. ACM.

[100] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. The tensor data platform: Towards an ai-centric database system. *arXiv preprint arXiv:2211.02753*, 2022.

[101] Floris Geerts, Thomas Muñoz, Cristian Riveros, Jan Van den Bussche, and Domagoj Vrgoč. Matrix query languages. *SIGMOD Rec.*, 50(3):6–19, 2021.

[102] A. Gholami, Z. Yao, S. Kim, M. W. Mahoney, and K. Keutzer. Ai and memory wall. `https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8`, 2021.

[103] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587. IEEE, 2014.

[104] Abel Gonzalez-Garcia, Davide Modolo, and Vittorio Ferrari. Do semantic parts emerge in convolutional neural networks? *International Journal of Computer Vision*, 126(5):476–494, 2018.

[105] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[106] Ian J. Goodfellow, Quoc V. Le, Andrew M. Saxe, Honglak Lee, and Andrew Y. Ng. Measuring invariances in deep networks. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, pages 646–654. Curran Associates Inc., 2009.

[107] Google. Improved On-Device ML on Pixel 6, with Neural Architecture Search. `https://ai.googleblog.com/2021/11/improved-on-device-ml-on-pixel-6-with.html`, 2021.

[108] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[109] Rıza Alp Güler, Natalia Neverova, and Iasonas Kokkinos. Densepose: Dense human pose estimation in the wild. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7297–7306, 2018.

[110] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[111] Habana. Habana. `https://habana.ai/`, 2022.

[112] Sasun Hambardzumyan, Abhinav Tuli, Levon Ghukasyan, Fariz Rahman, Hrant Topchyan, David Isayan, Mark McQuade, Mikayel Harutyunyan, Tatevik Hakobyan, Ivo Stranic, et al. Deep lake: A lakehouse for deep learning. *arXiv preprint arXiv:2209.10785*, 2022.

[113] Xixian Han, Jianzhong Li, and Hong Gao. Efficient top-k retrieval on massive data. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2687–2699, 2015.

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

[115] Dong He, Maureen Daum, Walter Cai, and Magdalena Balazinska. Deepeverest: Accelerating declarative top-k queries for deep neural network interpretation. *Proc. VLDB Endow.*, 15(1):98–111, 2021.

[116] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. Query processing on tensor computation runtimes. *Proc. VLDB Endow.*, 15(11):2811–2825, sep 2022.

[117] Dong He, Jieyu Zhang, Maureen Daum, Alexander Ratner, and Magdalena Balazinska. Masksearch: Querying image masks at scale. *arXiv preprint arXiv:2305.02375*, 2023.

[118] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[119] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[120] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, 2013.

[121] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and et al. The madlib analytics library: Or mad skills, the sql. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[122] Pedro Holanda and Hannes Mühleisen. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, New York, NY, USA, 2019. ACM.

[123] Sungsoo Ray Hong, Jessica Hullman, and Enrico Bertini. Human factors in model interpretability: Industry practices, challenges, and needs. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW1):1–26, 2020.

[124] Julia Hornauer and Vasileios Belagiannis. Heatmap-based out-of-distribution detection. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2603–2612, 2023.

[125] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, October 2018. USENIX Association.

[126] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. TCUDB: accelerating database with tensor processors. *CoRR*, abs/2112.07552, 2021.

[127] Dylan Hutchison, Bill Howe, and Dan Suciu. Laradb. *Proceedings of the 4th Algorithms and Systems on MapReduce and Beyond - BeyondMR'17*, 2017.

[128] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. Joining ranked inputs in practice. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 950–961. Elsevier, 2002.

[129] Matteo Interlandi, Konstantinos Karanasos, Dong He, Dalitso Hansini Banda, Jesus CAMACHO RODRIGUEZ, Rathijit Sen, and Supun Chathurang Nakandala. Method and system for query processing over tensor runtimes, August 3 2023. US Patent App. 17/587,952.

[130] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. Declarative recursive computation on an rdbms: Or, why you should use a database for distributed machine learning. *Proc. VLDB Endow.*, 12(7):822–835, 2019.

[131] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2010.

[132] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dongjin Shin, and Byung-Gon Chun. JANUS: fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 453–468. USENIX Association, 2019.

[133] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, New York, NY, USA, 2019. ACM.

[134] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of Machine Learning and Systems*, volume 1, pages 27–39, 2019.

[135] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: An experimental evaluation. *Proc. VLDB Endow.*, 7(8):625–636, apr 2014.

[136] Glenn Jocher. YOLOv5 by Ultralytics. `https://github.com/ultralytics/yolov5`, May 2020.

[137] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[138] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray

Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[139] Minsuk Kahng, Pierre Y Andrews, Aditya Kalro, and Duen Horng Polo Chau. Activis: Visual exploration of industry-scale deep neural network models. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):88–97, 2017.

[140] Minsuk Kahng, Dezhi Fang, and Duen Horng Chau. Visual exploration of machine learning results using data cube analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–6. ACM, 2016.

[141] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *arXiv preprint arXiv:1805.01046*, 2018.

[142] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, aug 2017.

[143] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. Data management for ml-based analytics and beyond.

[144] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. Semantic indexes for machine learning-based queries over unstructured data. *arXiv preprint arXiv:2009.04540*, 2020.

[145] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. Extending relational query processing with ML inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, misc Proceedings*. www.cidrdb.org, 2020.

[146] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[147] David Kernert, Frank Köhler, and Wolfgang Lehner. Slacid - sparse linear algebra in a column-oriented in-memory database system. In *Proceedings of the 26th International*

Conference on Scientific and Statistical Database Management, New York, NY, USA, 2014. ACM.

[148] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018.

[149] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009.

[150] Mijung Kim and K. Selçuk Candan. Tensordb: In-database tensor manipulation with tensor-relational query plans. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 2039–2041, New York, NY, USA, 2014. ACM.

[151] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026, 2023.

[152] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1:77:1–77:29, 2017.

[153] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International conference on machine learning*, pages 1885–1894. PMLR, 2017.

[154] Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Sara Beery, Jure Leskovec, Anshul Kundaje, Emma Pierson, Sergey Levine, Chelsea Finn, and Percy Liang. WILDS: A benchmark of in-the-wild distribution shifts. *CoRR*, abs/2012.07421, 2020.

[155] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. Technical report, USA, 1997.

[156] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, Ana Klimovic, and Gustavo Alonso. Modularis: Modular relational analytics over heterogeneous distributed platforms. *VLDB*, 14(13):3308–3321, 2021.

[157] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. Tensors: An abstraction for general data processing. *Proc. VLDB Endow.*, 14(10):1797–1804, 2021.

[158] Josua Krause, Adam Perer, and Kenney Ng. Interacting with predictions: Visual inspection of black-box machine learning models. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5686–5697. ACM, 2016.

[159] Sanjay Krishnan and Eugene Wu. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, pages 1–6. ACM, 2017.

[160] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[161] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. Principles of explanatory debugging to personalize interactive machine learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces*, pages 126–137. ACM, 2015.

[162] Arun Kumar, Matthias Boehm, and Jun Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722, New York, NY, USA, 2017. ACM.

[163] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. 2020.

[164] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[165] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. The art of balance: A rateupdb™ experience of building a cpu/gpu hybrid database product. *Proc. VLDB Endow.*, 14(12):2999–3013, 2021.

[166] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[167] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch

distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12), 2020.

[168] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. Mlog: Towards declarative in-database machine learning. *Proc. VLDB Endow.*, 10(12):1933–1936, 2017.

[169] Zhe Li and Kenneth A. Ross. Fast joins using join indices. *The VLDB Journal*, 8(1):1–24, apr 1999.

[170] Zachary C Lipton. The mythos of model interpretability. *Queue*, 16(3):31–57, 2018.

[171] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. A declarative system for optimizing ai workloads. *arXiv preprint arXiv:2405.14696*, 2024.

[172] Mengchen Liu, Jiaxin Shi, Zhen Li, Chongxuan Li, Jun Zhu, and Shixia Liu. Towards better analysis of deep convolutional neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):91–100, 2016.

[173] Shuying Liu and Weihong Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734. IEEE, 2015.

[174] Ting Liu, Andrew W Moore, and Alexander Gray. New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, 7:1135–1158, 2006.

[175] Zixuan Liu, Ehsan Adeli, Kilian M Pohl, and Qingyu Zhao. Going beyond saliency maps: Training deep models to interpret deep models. In *Information Processing in Medical Imaging: 27th International Conference*, pages 71–82. Springer, 2021.

[176] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1493–1508, New York, NY, USA, 2018. Association for Computing Machinery.

[177] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[178] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. *Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects*, pages 1633–1649. ACM, New York, NY, USA, 2020.

[179] Minghuang Ma, Haoqi Fan, and Kris M Kitani. Going deeper into first-person activity recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1894–1903. IEEE, 2016.

[180] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. *Proc. VLDB Endow.*, 9(9):636–647, may 2016.

[181] Laurent Mazare. Pytorch rust bindings. `https://github.com/LaurentMazare/tch-rs`, 2020.

[182] Parmita Mehta, Stephen Portillo, Magdalena Balazinska, and Andrew Connolly. Toward sampling for deep learning model diagnosis. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1910–1913. IEEE, 2020.

[183] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, 2016.

[184] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[185] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, 2017.

[186] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering*, pages 1393–1394. IEEE, 2017.

[187] Microsoft. Predict in t-sql. `https://docs.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-ver15`, 2021.

[188] Microsoft. Tutorial: Score machine learning models with predict in serverless apache spark pools. `https://docs.microsoft.com/en-us/azure/synapse-analytics/machine-learning/tutorial-score-model-predict-spark-pool`, 2021.

[189] Microsoft. Hummingbird. https://github.com/microsoft/hummingbird, 2022.

[190] Microsoft. Onnx runtime web—running your machine learning model in browser. https://cloudblogs.microsoft.com/opensource/2021/09/02/onnx-runtime-web-running-your-machine-learning-model-in-browser/, 2022.

[191] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, volume 26, pages 3111–3119. Curran Associates Inc., 2013.

[192] Yifei Ming, Hang Yin, and Yixuan Li. On the impact of spurious correlation for out-of-distribution detection. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pages 10051–10059, 2022.

[193] Christoph Molnar. *Interpretable Machine Learning*. 2019. https://christophm.github.io/interpretable-ml-book/.

[194] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, pages 807–814. Omnipress, 2010.

[195] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 899–917. USENIX Association, 2020.

[196] Amir Netz, Jeff Bernhardt, Usama Fayyad, and Surajit Chaudhuri. Integration of data mining and relational databases. In *Proceedings of the 26th International Conference on Very Large Databases*. VLDB Endowment, 2000.

[197] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[198] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, misc Proceedings*. www.cidrdb.org, 2020.

[199] Amadou Ngom, Prashanth Menon, Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, and Andrew Pavlo. *Filter Representation in Vectorized Query Execution*. ACM, New York, NY, USA, 2021.

[200] Anh Nguyen, Alexey Dosovitskiy, Jason Yosinski, Thomas Brox, and Jeff Clune. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 3387–3395. Curran Associates Inc., 2016.

[201] Luke Oakden-Rayner, Jared Dunnmon, Gustavo Carneiro, and Christopher Ré. Hidden stratification causes clinically meaningful failures in machine learning for medical imaging. In *Proceedings of the ACM conference on health, inference, and learning*, pages 151–159, 2020.

[202] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 3(3):e10, 2018.

[203] Stephen M Omohundro. *Five Balltree Construction Algorithms*. International Computer Science Institute Berkeley, 1989.

[204] HweeHwa Pang, Xuhua Ding, and Baihua Zheng. Efficient processing of exact top-k queries over disk-resident sorted lists. *The VLDB Journal*, 19(3):437–456, 2010.

[205] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.

[206] Nicolas Papernot and Patrick McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765*, 2018.

[207] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[208] Vaishakh Patil, Christos Sakaridis, Alex Liniger, and Luc Van Gool. P3depth: Monocular depth estimation with a piecewise planarity prior. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[209] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proc. VLDB Endow.*, 14(2):202–214, 2020.

[210] Johns Paul, Jiong He, and Bingsheng He. Gpl: A gpu-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950, New York, NY, USA, 2016. ACM.

[211] Johns Paul, Shengliang Lu, and Bingsheng He. Database systems on gpus. *Foundations and Trends® in Databases*, 11(1):1–108, 2021.

[212] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[213] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(12):2033–2046, 2020.

[214] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, 2016.

[215] Gregory Plumb, Marco Tulio Ribeiro, and Ameet Talwalkar. Finding and fixing spurious patterns with explanations. *arXiv preprint arXiv:2106.02112*, 2021.

[216] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, New York, NY, USA, 2015. ACM.

[217] Orestis Polychroniou and Kenneth A. Ross. Towards practical vectorized analytical query engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. ACM.

[218] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. Toward gpus being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, New York, NY, USA, 2015. ACM.

[219] Shreya Prasad, Arash Fard, Vishrut Gupta, Jorge Martinez, Jeff LeFevre, Vincent Xu, Meichun Hsu, and Indrajit Roy. Large-scale predictive analytics in vertica: Fast data transfer, distributed model creation, and in-database prediction. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1657–1668, New York, NY, USA, 2015. ACM.

[220] PyTorch. Pytorch java bindings. `https://github.com/pytorch/java-demo`, 2020.

[221] PyTorch. Pytorch on xla devices. https://pytorch.org/xla/release/1.9/index.html, 2022.

[222] PyTorch. Torch.tensor documentation. https://pytorch.org/docs/stable/tensors.html, 2022.

[223] PyTorch. Unique.cpp. https://github.com/pytorch/pytorch/blob/7ee0712642492ef221a69d3fdf13b607f406bd78/aten/src/ATen/native/Unique.cpp, 2022.

[224] Mark Raasveldt and Hannes Mühleisen. Data management for data science - towards embedded analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, misc Proceedings*. www.cidrdb.org, 2020.

[225] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, 2013.

[226] Luis Remis and Chaunté W. Lacewell. Using vdms to index and search 100m images. *Proc. VLDB Endow.*, 14(12):3240–3252, jul 2021.

[227] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM, 2016.

[228] J Rogers, R Simakov, E Soroush, P Velikhov, M Balazinska, D DeWitt, B Heath, D Maier, S Madden, J Patel, et al. Overview of scidb: Large scale array storage, processing and analysis. In *2010 International Conference on Management of Data, SIGMOD'10*, pages 963–968, 2010.

[229] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

[230] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Comput. Surv.*, 55(1), 2022.

[231] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 49–68. ACM, 2013.

[232] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242, New York, NY, USA, 2013. ACM.

[233] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[234] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`. Accessed: 2021-03-30.

[235] Sebastian Schelter, Shannon Quinn, Suneel Marthi, and Andrew Musselman. Samsara: Declarative machine learning on distributed dataflow systems. 2016.

[236] Raimondo Schettini, Gianluigi Ciocca, Silvia Zuffi, Istituto Tecnologie, Infomatiche Multimediali, and Consilio Ricerche. A survey of methods for colour image indexing and retrieval in image databases. 02 2001.

[237] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[238] Maximilian Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. ML2SQL - compiling a declarative machine learning language to SQL and python. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 562–565. OpenProceedings.org, 2019.

[239] Maximilian Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. In-database machine learning: Gradient descent and tensor algebra for main memory database systems. In *BTW 2019*, pages 247–266. Gesellschaft für Informatik, Bonn, 2019.

[240] Thibault Sellam, Kevin Lin, Ian Huang, Michelle Yang, Carl Vondrick, and Eugene Wu. Deepbase: Deep inspection of neural networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1117–1134, 2019.

[241] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 618–626, 2017.

[242] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1907–1922, New York, NY, USA, 2016. ACM.

[243] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1617–1632, New York, NY, USA, 2020. ACM.

[244] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149, New York, NY, USA, 2016. ACM.

[245] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[246] Sahil Singla, Eric Wallace, Shi Feng, and Soheil Feizi. Understanding impacts of high-order loss approximations and features in deep learning interpretation. In *International Conference on Machine Learning*, pages 5848–5856. PMLR, 2019.

[247] Phanwadee Sinthong and Michael J. Carey. Polyframe: A retargetable query-based approach to scaling dataframes. *Proc. VLDB Endow.*, 14(11):2296–2304, 2021.

[248] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.

[249] Rupesh Kumar Srivastava, Jonathan Masci, Sohrob Kazerounian, Faustino Gomez, and Jürgen Schmidhuber. Compete to compute. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, pages 2310–2318. Curran Associates Inc., 2013.

[250] Statista. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. January 2022.

[251] Statista. Worldwide ai hardware market revenues. https://www.statista.com/statistics/1003890/worldwide-artificial-intelligence-hardware-market-revenues/, January 2022.

[252] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In Judith Bayard Cushing, James French, and Shawn Bowers, editors, *Scientific and Statistical Database Management*, pages 1–16, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[253] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[254] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.

[255] Umar Syed and Sergei Vassilvitskii. Sqml: Large-scale in-database machine learning with pure sql. In *Proceedings of the 2017 Symposium on Cloud Computing*, page 659, New York, NY, USA, 2017. ACM.

[256] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[257] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[258] OctoML AI team. Tvm on m1 gpus performance. https://octoml.ai/blog/on-the-apple-m1-beating-apple-s-core-ml-4-with-50-model-performance-improvements/, February 2022.

[259] Tesla. Tesla unveils chip to train a.i. models inside its data centers. https://www.cnbc.com/2021/08/19/tesla-unveils-dojo-d1-chip-at-ai-day.html, 2022.

[260] Stefano Teso, Öznur Alkan, Wolfgang Stammer, and Elizabeth Daly. Leveraging explanations in interactive machine learning: An overview. *Frontiers in Artificial Intelligence*, 6:1066049, 2023.

[261] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[262] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 648–659. VLDB Endowment, 2004.

[263] TVM. Bring your own codegen to tvm. `https://tvm.apache.org/docs/dev/how_to/relay_bring_your_own_codegen.html`, 2022.

[264] TVM. Pass infrastructure. `https://tvm.apache.org/docs/arch/pass_infra.html`, 2022.

[265] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1285–1300, 2018.

[266] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–3, 2016.

[267] Tin Vu. Deep query optimization. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1856–1858, New York, NY, USA, 2019. ACM.

[268] Eric Wallace, Shi Feng, and Jordan Boyd-Graber. Interpreting neural networks with nearest neighbors. *arXiv preprint arXiv:1809.02847*, 2018.

[269] Dalin Wang, Feng Zhang, Weitao Wan, Hourun Li, and Xiaoyong Du. Finequery: Fine-grained query processing on cpu-gpu integrated architectures. In *2021 IEEE International Conference on Cluster Computing*, pages 355–365, 2021.

[270] Shen Wang and Yuxin Gong. Adversarial example detection based on saliency map features. *Applied Intelligence*, pages 1–14, 2022.

[271] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. Spores: Sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951*, 2020.

[272] Lindsey Linxi Wei, Chung Yik Edward Yeung, Hongjian Yu, Jingchuan Zhou, Dong He, and Magdalena Balazinska. Demonstration of masksearch: Efficiently querying image masks for machine learning workflows. *arXiv preprint arXiv:2404.06563*, 2024.

[273] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2009.

[274] Julia K Winkler, Christine Fink, Ferdinand Toberer, Alexander Enk, Teresa Deinlein, Rainer Hofmann-Wellenhof, Luc Thomas, Aimilios Lallas, Andreas Blum, Wilhelm Stolz, et al. Association between surgical skin markings in dermoscopic images and diagnostic performance of a deep learning convolutional neural network for melanoma recognition. *JAMA dermatology*, 155(10):1135–1141, 2019.

[275] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1317–1334, 2020.

[276] Xilinx. Xilinx ai engine technology. `https://www.xilinx.com/products/technology/ai-engine.html`, 2022.

[277] Hanwen Xu, Naoto Usuyama, Jaspreet Bagga, Sheng Zhang, Rajesh Rao, Tristan Naumann, Cliff Wong, Zelalem Gero, Javier González, Yu Gu, et al. A whole-slide foundation model for digital pathology from real-world data. *Nature*, pages 181–188, 2024.

[278] Dengpan Ye, Chuanxi Chen, Changrui Liu, Hao Wang, and Shunzhi Jiang. Detection defense against adversarial attacks with saliency map. *International Journal of Intelligent Systems*, 37(12):10193–10210, 2022.

[279] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[280] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. Tensor relational algebra for distributed machine learning system design. *Proc. VLDB Endow.*, 14(8):1338–1350, 2021.

[281] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proc. VLDB Endow.*, 6(10):817–828, 2013.

[282] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*, 2012.

[283] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory and Practice of Logic Programming*, 17(5-6):1048–1065, 2017.

[284] Matthew D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014*, pages 818–833. Springer, 2014.

[285] Chiliang Zhang, Zhimou Yang, and Zuochang Ye. Detecting adversarial perturbations with salieny. In *Proceedings of the 6th International Conference on Information Technology: IoT and Smart City*, pages 25–30, 2018.

[286] Enhao Zhang, Maureen Daum, Dong He, Manasi Ganti, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. Equi-vocal demonstration: Synthesizing video queries from user interactions. *Proceedings of the VLDB Endowment*, 16(12):3978–3981, 2023.

[287] Enhao Zhang, Maureen Daum, Dong He, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. Equi-vocal: Synthesizing queries for compositional video events from limited user interactions. *Proceedings of the VLDB Endowment*, 16(11):2714–2727, 2023.

[288] Huayi Zhang, Binwei Yan, Lei Cao, Samuel Madden, and Elke Rundensteiner. Metastore: Analyzing deep learning meta-data at scale. *Proc. VLDB Endow.*, 17(6):1446–1459, may 2024.

[289] Shile Zhang, Chao Sun, and Zhenying He. Listmerge: Accelerating top-k aggregation queries over large number of lists. In *International Conference on Database Systems for Advanced Applications*, pages 67–81. Springer, 2016.

[290] Yu Zhang, Wei Han, James Qin, Yongqiang Wang, Ankur Bapna, Zhehuai Chen, Nanxin Chen, Bo Li, Vera Axelrod, Gary Wang, et al. Google usm: Scaling automatic speech recognition beyond 100 languages. *arXiv preprint arXiv:2303.01037*, 2023.

[291] Yuhao Zhang and Arun Kumar. Panorama: a data system for unbounded vocabulary querying over video. *Proceedings of the VLDB Endowment*, 13(4):477–491, 2019.

[292] B. Zhou, A. Khosla, Lapedriza. A., A. Oliva, and A. Torralba. Learning Deep Features for Discriminative Localization. *CVPR*, 2016.

[293] Bolei Zhou, David Bau, Aude Oliva, and Antonio Torralba. Interpreting deep visual representations via network dissection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(9):2131–2145, 2018.

[294] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Object detectors emerge in deep scene cnns. *arXiv preprint arXiv:1412.6856*, 2014.

[295] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2921–2929. IEEE, 2016.

[296] Bolei Zhou, Yiyou Sun, David Bau, and Antonio Torralba. Revisiting the Importance of Individual Units in CNNs via Ablation. *arXiv preprint arXiv:1806.02891*, 2018.

[297] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 145–156, New York, NY, USA, 2002. ACM.