MaskSearch: Querying Image Masks at Scale

Dong He, Jieyu Zhang, Maureen Daum, Alexander Ratner, Magdalena Balazinska
Paul G. Allen School of Computer Science & Engineering, University of Washington
{donghe, jieyuz2, mdaum, ajratner, magda}@cs.washington.edu

***Abstract*—Machine learning tasks over image databases often generate masks that annotate image content (e.g., saliency maps, segmentation maps, depth maps) and enable a variety of applications (e.g., determine whether a model is learning spurious correlations or if an image was maliciously modified to mislead a model). While queries that retrieve examples based on mask properties are valuable to practitioners, existing systems do not support them efficiently. In this paper, we formalize the problem and propose MaskSearch, a system that focuses on accelerating queries over databases of image masks while guaranteeing the query result accuracy. MaskSearch leverages a novel indexing technique and an efficient filter-verification query execution framework. Experiments with our prototype 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 dataset exploration and analysis processes.**

I. Introduction

Machine learning (ML) tasks over image databases commonly generate masks that annotate individual pixels in images. For instance, model explanation techniques [1], [2], [3], [4], [5] generate saliency maps to highlight the significance of individual pixels to a model's output. In image segmentation tasks [6], [7], [8], masks denote the probability of pixels being associated with a specific class or an instance, while in depth estimation tasks [9], [10], masks reflect the depth of each pixel. Figure 1 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 [11], [12], [13], [14], [15], [16]. Other applications based on the properties of masks include identifying maliciously attacked examples using saliency maps [17], [18], [19], monitoring model errors [20], [21], [22] using segmentation masks, traffic monitoring and retail analytics using segmentation masks [23], [24].

The wide-ranging applications underscore an emerging necessity for ML practitioners: the capability to efficiently query and retrieve examples from image databases together with their masks, based on the properties of the latter [12], [25], [7]. Today, ML 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 [28]): Bob is an engineer developing a wild animal image classification model using the WILDS dataset [27]. He notices that the model's accuracy drops significantly when the background conditions change. So he computes saliency maps [3] for the misclassified images, examples of which are shown in Figure 2, where red (or blue) pixels indicate higher (or lower) importance for the model's prediction. He finds that the model focuses on the background pixels instead of the animals to make predictions. To correct the model's focus, Bob augments the dataset and retrains the model to ensure that it relies on relevant features. He first finds images where the model focuses on the areas outside the foreground objects (i.e., the animals, which can be detected by an object detector). Then, he randomizes the pixels outside the foreground objects in these images while leaving the original labels unchanged. These images are added to the dataset and the model is retrained. Such an approach is known to help improve model performance [28].*

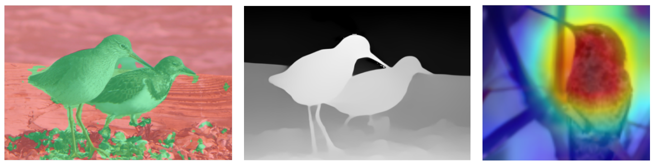

(a) Segmentation mask (b) Depth estimation mask (c) Saliency map

Fig. 1: Examples of image masks that annotate image content for ImageNet [26] images produced by ML tasks.

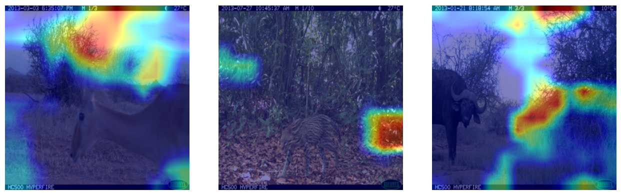

(a) (b) (c)

Fig. 2: Example images from *WILDS* [27] with saliency maps overlaid. Red pixels indicate high saliency and blue pixels indicate low saliency.

Scenario 2 *(inspired by [15]): Alice is developing a model to detect COVID-19 from chest X-rays. Although the model achieves high accuracy on training and validation sets, its predictions in hospitals often contradict PCR test results. Examining saliency maps of a few randomly selected images, Alice notices that high-value pixels seem to cluster around peripheral markers rather than lung regions, indicating the model is possibly learning confounding features (e.g., lateral markers) instead of medical pathology. To verify her hypothesis, Alice analyzes the saliency maps of a larger subset of images considering different-sized regions of interests corresponding* *to the main parts of the lungs.*

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 [29]. According to [12], 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. An example query for Scenario 2 retrieves images for which the model focuses its attention outside the lung region, specified as a bounding box. At the core of these queries are predicates on image masks that apply filters and aggregations on the values of pixels within regions of interest (ROIs), e.g., filters on the fraction of salient pixels in the lung region, which indicates model attention. 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 across masks, e.g., studying the difference between model saliency maps and human attention maps [25]. Top-k computations are also widely used. For example, Alice from Scenario 2 might be interested in finding the top-K X-rays whose saliency maps have the fewest salient pixels in the lung region. Formal definitions of our target queries are in §II-A.

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. Existing methods do not support these queries efficiently. Using NumPy 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 6). Existing multi-dimensional indexing techniques also do not provide better execution times because masks are dense arrays. Array databases such as SciDB [30] and TileDB [31], 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 6). While masks can be flattened as vectors and stored in vector databases, the latter support fundamentally different type of queries as we discuss further in §II-B, and are not designed for the types of queries in this paper.

MASKSEARCH accelerates the aforementioned queries without any loss in query result 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. 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 same result as if all masks had been individually scanned (i.e., MASKSEARCH produces exact results, not approximate ones).

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 [32], [33] and queries that contain ML models [34], [35], [36], [37] 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.

In summary, the contributions of this paper 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 (§II).
- We develop a novel indexing technique and an efficient filter-verification query execution framework (§III).
- 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 (§IV).

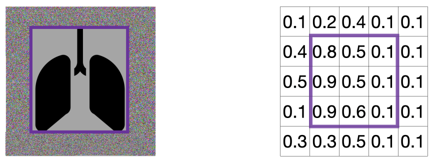
Overall, MASKSEARCH is an important next step toward the seamless and rapid exploration of a dataset based on masks generated by ML models. It is an important component in a toolbox of methods for ML model explainability and debugging.

II. QUERIES OVER MASKS

This section formalizes queries over masks and discusses the challenges associated with their efficient execution.

A. 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 3 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,



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

(a) Example x-ray (b) Mask

Fig. 3: A toy image motivated by [15] 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.

```
MasksDatabaseView (
  mask_id INT PRIMARY KEY,
  image_id INT, --Image that generated the mask
  model_id INT, --Model that generated the mask
  mask_type INT, --Type of mask
  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 $(6, 4, \text{ResNet-50}, \text{SaliencyMap}, [[0.9, 0.5, \ldots], \ldots])$, referring to a saliency map (mask #6) computed for image #4 using ResNet-50 [38]. Note that an image can have multiple or no masks and there cannot be multiple `image_id`s for the same `mask_id` as `mask_id` is the primary key.

An ROI is a bounding box. Figure 3 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`. In our current implementation, ROIs is stored as an array of coordinate values representing the bounding box in a separate table that is joined with `MasksDatabaseView` during query processing.

Basic Queries. MaskSearch supports queries that specify: (1) regions of interest within 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), and aggregates over those pixels that satisfy the predicates (i.e., count of pixels). A query over masks 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] ...
ORDER BY val [ASC | DESC] [LIMIT K]
```

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 (e.g., the center of the masks for standardized images such as X-rays) or obtained from another table (e.g., a table containing bounding boxes) joined with `MasksDatabaseView`.

`CP` function. At the core of a query is the `CP` function, which stands for "Count of 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,

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

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 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 §I. 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 that corresponds to the lungs. Hence, she writes a query that computes the number of salient (e.g., with value > 0.85) pixels within the lung region, which she specifies manually as a bounding box, `roi`. She retrieves all the images where the number of salient pixels is less than $10{,}000$:*

```
SELECT image_id FROM MasksDatabaseView
WHERE CP(mask, roi, (.85, 1.)) < 10000;
```

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 images with the lowest ratios:

```
SELECT image_id,
  CP(mask,roi,(.85,1.))/CP(mask,-,(.85,1.)) 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] ...
GROUP BY [image_id | model_id | mask_type]
HAVING <filter on aggregate> [AND | OR] ...
ORDER BY aggregate [ASC | DESC] [LIMIT K]
```

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 in §I, 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>.7),-,(.7,1.))AS s
FROM MasksDatabaseView WHERE mask_type IN (1, 2)
GROUP BY image_id ORDER BY s DESC LIMIT 10;
```

Usefulness for segmentation masks. In image segmentation, each pixel is often assigned a discrete label or a probability value indicating the likelihood of belonging to a desired class. In such cases, the query predicates can be set to retrieve images/masks where a specific label dominates or where the probability exceeds a threshold in an ROI.

B. Challenges

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. 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 [26] takes more than 30 minutes to complete (Figure 6). While parallel processing can reduce wall clock times, it does not reduce the total amount of work done. As we discuss later, MaskSearch reduces the total amount of work required to process a query, and could benefit from compression and parallelization, which are orthogonal techniques.

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 (e.g., 65 billion pixels for ImageNet).

Vector DBs, both functionally and practically do not support our target queries. They are designed for similarity searches to find matches to a query vector. 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 metadata in vector DBs is impractical because the ROIs and pixel value ranges are specified at query time and can be arbitrary. Additionally, vector DBs often have dimensionality limits (e.g., Milvus: 32,768 [39] and Pinecone: 20,000 [40]), which are insufficient for the number of pixels in masks.

Array databases [30], [31] 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.

III. MaskSearch

MaskSearch efficiently executes queries over a database of image masks while guaranteeing query result accuracy. The key challenge is that the database of masks is too large to fit in memory, and loading and processing all masks is slow.

To accelerate such queries, MaskSearch introduces a novel type of index, called the Cumulative Histogram Index (CHI) (§III-A), and an efficient filter-verification query execution framework (§III-B). 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 query result accuracy and overcomes the bottleneck of query execution by significantly reducing the number of masks that must be loaded from disk.

A. 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.

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 possible pixel values.

Instead, CHI is a data structure that efficiently provides upper and lower bounds on a query predicate, 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 to verify the predicate.

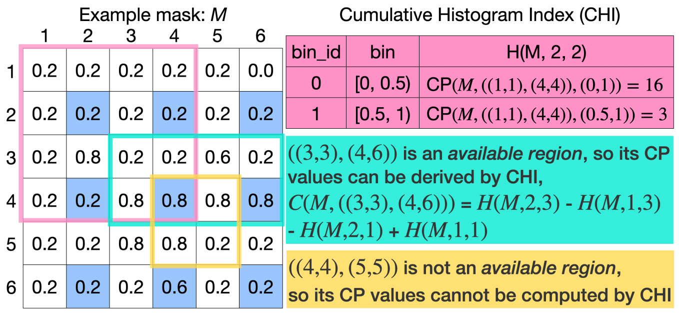
CHI Details. CHI leverages two key ideas: discretization and cumulative counts. Discretization reduces the total amount of information in the index, while cumulative counts yield highly efficient lookups. We explain both here.

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, it 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 identifying all the cells that intersect with roi and all the bins that intersect with (lv, uv) and perform our query execution (discussed in §III-B) 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 reverse sum enables us to quickly compute bounds on the number of pixels exceeding a given threshold, which is especially useful in model explanation workloads. 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 $mask$, then formally,

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

Example: *Figure 4 shows an example with $w_c = 2$, $h_c = 2$, and $b = 2$. Each blue mask cell, (x_c, y_c), marks the corner of a discretized region. With $b = 2$, the pixel value range becomes 2 bins, $[0, 0.5)$ and $[0.5, 1)$. We build $H(M, x_c/w_c, y_c/h_c)$ for each blue mask cell (x_c, y_c), e.g., the blue mask cell $(4, 4)$, corresponds to index entry $(x_c = 2, y_c = 2)$, and $H(M, 2, 2)[0] = \texttt{CP}(M, ((1, 1), (4, 4)), (0, 1)) = 16$ (all 16 pixels are in $[p_{min}, p_{max})$) and $H(M, 2, 2)[1] =$*



	1	2	3	4	5	6
1	0.2	0.2	0.2	0.2	0.2	0.0
2	0.2	0.2	0.2	0.2	0.2	0.2
3	0.2	0.8	0.2	0.2	0.6	0.2
4	0.2	0.2	0.8	0.8	0.8	0.8
5	0.2	0.2	0.8	0.8	0.2	0.2
6	0.2	0.2	0.2	0.6	0.2	0.2

bin_id	bin	H(M, 2, 2)
0	[0, 0.5)	CP$(M, ((1,1), (4,4)), (0,1)) = 16$
1	[0.5, 1)	CP$(M, ((1,1), (4,4)), (0.5,1)) = 3$

$((3,3), (4,6))$ is an *available region*, so its CP values can be derived by CHI, $C(M, ((3,3), (4,6))) = H(M,2,3) - H(M,1,3) - H(M,2,1) + H(M,1,1)$

$((4,4), (5,5))$ is not an *available region*, so its CP values cannot be computed by CHI

Fig. 4: An example of CHI, CP, *available region*, and C.

$\texttt{CP}(M, ((1, 1), (4, 4)), (.5, 1)) = 3$ (3 pixels are in $[0.5, p_{max})$).

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 3.1: 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 are bounding boxes that start from the bottom-right corner of a blue cell ($(0, 0)$ included) 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 (§III-B). 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 $mask\_id$,

$$\begin{aligned} C(M, ((x_1, y_1), (x_2, y_2))) &= H(M, x_2/w_c, y_2/h_c) \\ &- H(M, (x_1 - 1)/w_c, y_2/h_c) - H(M, x_2/w_c, (y_1 - 1)/h_c) \\ &+ H(M, (x_1 - 1)/w_c, (y_1 - 1)/h_c) \end{aligned} \quad (2)$$

where $-$ and $+$ are element-wise subtraction and addition, respectively. Equation (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 (finitely)-additive. For any $mask\_id$ and r, $C(mask\_id, r)[\lceil p_{max}/\Delta \rceil]$ is always 0 for notation simplicity. ***Example:*** *Figure 4 shows an example of computing* $C(M, ((3,3),(4,6)))$.

After MASKSEARCH computes the reverse cumulative sums of pixel counts, C, for a region r, the pixel counts between any two bin boundaries 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 (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 §III-B.

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 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 technique described in §III-F. The number of CHI 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.

B. 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 §III-C, 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 §III-D and §III-E, 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:

- **Filter stage**: prune the masks that *are guaranteed to fail* the predicate P, and add the masks that *are guaranteed to satisfy* P 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 query result accuracy because it only prunes the masks that are guaranteed to fail P and adds the masks that are guaranteed to satisfy P directly to the result set; it subsequently verifies any uncertain masks to ensure the query result accuracy.

1) 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, it reduces the number of masks loaded from disk during the verification stage (§III-B2) 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 CHI to derive the bounds for arbitrary roi and (lv, uv).
Notation. P denotes $\texttt{CP}(mask, roi, (lv, uv)) > T$. 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 $C(mask\_id, r)$.

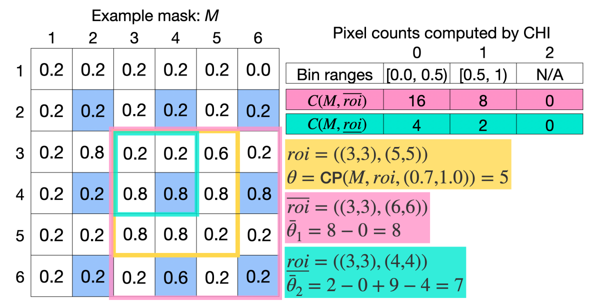
When a session of MASKSEARCH starts, the CHI of each mask is loaded from disk to memory and will be held in memory for the duration of the system run time. Note that the size of the CHI of a mask is much smaller than the size of the mask itself, and therefore, even if the CHI of a mask is on disk, computing 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, for each $mask$ targeted by P, 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.

Approach 1 first identifies the smallest *available region* (definition 3.1) 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 $\overline{roi}$. Then, $C(\overline{roi})$ (i.e., $C(mask\_id, \overline{roi})$) can be computed by CHI using Equation (2). Finally, $\bar{\theta}_1$ is computed as,

$$\bar{\theta}_1 = C(\overline{roi})[\lfloor lv/\Delta \rfloor] - C(\overline{roi})[\lceil uv/\Delta \rceil] \tag{3}$$

where $\lfloor \cdot \rfloor$ (and $\lceil \cdot \rceil$) denotes the floor (and ceiling) of $\cdot$.



	0	1	2
Bin ranges	[0.0, 0.5)	[0.5, 1)	N/A
$C(M, \overline{roi})$	16	8	0
$C(M, \underline{roi})$	4	2	0

Fig. 5: An example of MaskSearch computing the upper bounds, $\bar{\theta}_1$ and $\bar{\theta}_2$, given a mask, roi, and (lv, uv).

Approach 2 first identifies the largest *available region* (definition 3.1) 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 (2). Finally, $\bar{\theta}_2$ is computed as,

$$\bar{\theta}_2 = C(\underline{roi})[\lfloor lv/\Delta \rfloor] - C(\underline{roi})[\lceil uv/\Delta \rceil] + |roi| - |\underline{roi}| \quad (4)$$

where $|\cdot|$ denotes the area of a region. $\underline{roi}$ does not always exist where w_c and h_c are large. In such cases, $\bar{\theta}_2$ 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,

$$C(roi)[\lfloor lv/\Delta \rfloor] - C(roi)[\lceil uv/\Delta \rceil] \geq \theta \quad (5)$$

We now show the following theorem.

Theorem 3.1: $\bar{\theta}$ 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 3.1: We first show $\bar{\theta}_1 \geq \theta$.

$$\begin{aligned}
\bar{\theta}_1 &= C(\overline{roi})[\lfloor lv/\Delta \rfloor] - C(\overline{roi})[\lceil uv/\Delta \rceil] && (6)\\
&\geq \texttt{CP}(\overline{roi}, (lv, uv)) && (7)\\
&= \texttt{CP}(roi, (lv, uv)) + \texttt{CP}(\overline{roi} \setminus roi, (lv, uv)) && (8)\\
&\geq \texttt{CP}(roi, (lv, uv)) = \theta && (9)
\end{aligned}$$

where Inequality (7) follows from Equation (5).

Let L denote $(\lfloor lv/\Delta \rfloor * \Delta, \lceil uv/\Delta \rceil * \Delta)$, then,

$$\begin{aligned}
\theta &= \texttt{CP}(roi, (lv, uv)) && (10)\\
&\leq \texttt{CP}(roi, L) && (11)\\
&= \texttt{CP}(\underline{roi}, L) + \texttt{CP}(roi \setminus \underline{roi}, L) && (12)\\
&\leq \texttt{CP}(\underline{roi}, L) + |roi| - |\underline{roi}| && (13)\\
&= C(\underline{roi})[\lfloor lv/\Delta \rfloor] - C(\underline{roi})[\lceil uv/\Delta \rceil] + |roi| - |\underline{roi}| && (14)\\
&= \bar{\theta}_2 && (15)
\end{aligned}$$

where Inequality (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: *Figure 5 shows an example, where the mask is the same as in Figure 4. The first approach identifies $\overline{roi}$, which is $((3,3),(6,6))$, and $C(M, \overline{roi})$ is computed by Equation (2). Then, $\bar{\theta}_1$ is computed using Equation (3), i.e., $C(M, \overline{roi})[1] - C(M, \overline{roi})[2] = 8 - 0 = 8$. The second approach identifies $\underline{roi}$, which is $((3,3),(4,4))$, and $C(M, \underline{roi})$ is computed by Equation (2). Then, $\bar{\theta}_2$ is computed using Equation (4), i.e., $C(M, \underline{roi})[1] - C(M, \underline{roi})[2] + |roi| - |\underline{roi}| = 2 - 0 + 9 - 4 = 7$. Finally, $\bar{\theta} = \min(\bar{\theta}_1, \bar{\theta}_2) = 7$.*

The two approaches are effective in yielding bounds in different scenarios. The first approach is more effective when roi and $\overline{roi}$ 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.

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 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.

Example: *Following the example in Figure 5, suppose the predicate is $\texttt{CP}(M, ((3,3),(5,5)), (0.7, 1.0)) > 9$. We know that $\bar{\theta} = 7 \leq T = 9$, so the mask is pruned in Case 1 and not loaded from disk to memory.*

2) Verification Stage: This stage verifies 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.

C. Generic Predicates

MaskSearch 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 $F = \texttt{CP}_1(...)\,\text{op}_1\,\texttt{CP}_2(...) \cdots \text{op}_{\text{n-1}}\,\texttt{CP}_n(...)$. MaskSearch uses the 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 §III-B1, as long as F is monotonic with respect to each $\texttt{CP}_i$ function. Common operators that make F monotonic include $+, -, \times$.

D. Aggregation

MaskSearch supports queries that contain scalar aggregates on CP functions or on the CP function over mask aggregations, as described in §II. 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 §III-C 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 §III-B to process the query. The index for the aggregated masks is either built ahead of time or incrementally built (§III-F), 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 using indexes built for the individual masks.

E. Top-K

To answer top-k queries, MASKSEARCH follows a similar idea as described in §III-B, 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 $\texttt{CP}$ 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,

$$\texttt{CP}(mask, roi, (lv, uv)) > \min_{mask' \in R} \texttt{CP}(mask', roi, (lv, uv)) \quad (16)$$

MASKSEARCH sequentially processes the masks. For each mask, it computes the upper bound $\bar{\theta}$ and compares $\bar{\theta}$ with the $\texttt{CP}$ 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 16.

F. Incremental Indexing

As we show in §IV-B and §IV-C, 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 §III-B, it checks if the CHI of the mask is already built. If so, MASKSEARCH directly proceeds as described in §III-B. If not, MASKSEARCH executes the query by loading the masks from disk and evaluating them against the predicate. 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.

IV. EVALUATION

A. Experimental Setup

Implementation. MASKSEARCH is written in Python as a library and can work seamlessly with existing databases that store and index 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 [27]. *WILDS* contains 22,275 images from the in-distribution and out-of-distribution validation sets of the iWildCam dataset [27]. For each image, we use GradCAM [3] to generate two saliency maps for two different ResNet-50 [38] models obtained from [27]. Each saliency map is 448×448 pixels. The second, called *ImageNet*, contains 1,331,167 images from the ImageNet dataset [26]. We also use GradCAM [3] to generate saliency maps for ResNet-50 [38] models. 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 specified otherwise, we set $b = 16$ for pixel value discretization for both datasets; we use $w_c = h_c = 64$ for *WILDS* and $w_c = h_c = 28$ for *ImageNet*, so that the uncompressed index sizes are about 5% of the compressed sizes (6.5 GB for *ImageNet* and 88 MB for *WILDS*). The index building times are 3 minutes for *WILDS* and 50 minutes for *ImageNet*. More granular indexes are discussed in §IV-D.

Baselines. To our knowledge, no existing system reduces the work of loading masks and computing the $\texttt{CP}$ function. We compare MASKSEARCH to three baselines: (1) PostgreSQL 10, storing masks as 2D arrays and implementing $\texttt{CP}$ as a UDF in C; (2) TileDB 2.17.1 [31] with TileDB-Py 0.23.1, where masks are stored as a 3D array (mask ID, height, width) and tile sizes are set to 448×448 for *WILDS* and 224×224 for *ImageNet* for optimal performance; and (3) NumPy 1.21.6, storing masks as NumPy arrays on disk with a Python-implemented, vectorized $\texttt{CP}$ function.

Machine configuration. Experiments were run on an AWS EC2 p3.2xlarge instance with an Intel Xeon E5-2686 v4 processor (8 vCPUs, 61 GiB memory), an NVIDIA Tesla V100 GPU (16 GiB memory), and EBS gp3 volumes (3000 IOPS, 125 MiB/s). The GPU is used only for mask computation. We evaluate MASKSEARCH in a single-node setup, aligning with typical data scientist workflows [41].

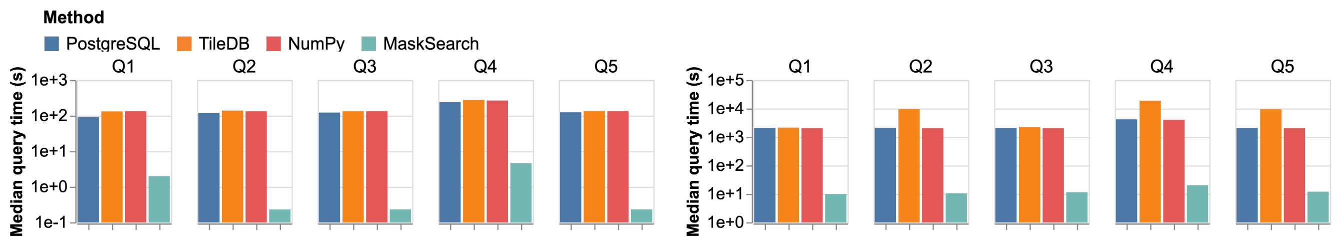
B. Individual Query Performance

We first evaluate the performance of MASKSEARCH on 5 individual queries motivated by the use cases in §I and §II. The specific parameters for each query are shown in Table I. Q1 and Q2 are motivated by Scenario 2 in §I, Q3 is a variant of Example 1 in §II, Q4 is a variant of Example 2 in §II, and Q5 is Example 2 in §II.

k is set to 25 for top-k 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 [42]. We build the CHI

TABLE I: 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 $\texttt{mean}(\texttt{CP}(mask, roi, (lv, uv)))$ for $mask$s associated with two models, $roi =$ object, $(lv, uv) = (0.8, 1.0)$
Q5	Returns top-25 images with largest $\texttt{CP}(\texttt{intersect}(mask), roi, (lv, uv))$ for $mask$s associated with two models, $roi =$ object, $(lv, uv) = (0.8, 1.0)$



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

Fig. 6: End-to-end individual query execution time on *WILDS* and *ImageNet*. 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.

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 6. In addition, Table II displays the number of masks loaded from disk by each system during query execution. Note that all baseline methods load the same number of masks; thus, the reported value represents their common performance.

As Figure 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. The standard deviations of the runs are small, typically within one second. 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 others. 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 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.

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. Table II highlights the key advantage of MASKSEARCH: it significantly reduces the number of masks

TABLE II: Number of masks loaded during query execution. MS = MASKSEARCH. 1.3M = 1,331,167, 2.6M = 2,662,334.

Dataset	Method	Q1	Q2	Q3	Q4	Q5
WILDS	MS	407	40	32	874	48
	Any baseline	22,275	22,275	22,275	44,550	22,275
ImageNet	MS	2696	3849	2943	1494	2768
	Any baseline	1.3M	1.3M	1.3M	2.6M	1.3M

loaded from disk during query execution. Unlike the baselines, which load all masks targeted by a query, MASKSEARCH leverages its filter-verification framework to avoid unnecessary disk operations by skipping masks guaranteed to satisfy or fail the query predicate. This reduction directly correlates with the improved query execution times. 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 ($2\times$ bound computation than other queries), as it contains an aggregation.

Using faster disks or parallelizing computation can reduce wall-clock time for both MASKSEARCH and the baselines, but they do not decrease the total resources consumed by the baselines since the same number of masks must still be loaded and processed. Our experiments show that doubling the disk read bandwidth reduces query execution time by approximately 50% for both MASKSEARCH and the baselines, and parallelizing computation can further reduce query time. However, these approaches only reduce wall-clock time and not the total work performed. In contrast, MASKSEARCH reduces the total work required, as evidenced by the fewer masks loaded and processed in Table II. When all masks reside in memory, the query processing bottleneck shifts from disk I/O to evaluating the CP function for every mask. Our execution framework can quickly compute bounds and would still accelerate query execution by avoiding unnecessary CP function evaluation.

We further evaluate MASKSEARCH on larger-scale datasets by executing Q1, Q2, and Q3 on duplicates of *ImageNet* and find that MaskSearch's query time grows linearly with the dataset size. With more than 10 million masks queried, MaskSearch executes each query in 90 seconds, e.g., for Q2, querying 1.33 M masks takes around 11 seconds and querying 10.65 M masks takes around 85 seconds.

C. Performance on Different Query Types

We evaluate MaskSearch's performance on three query types with varying parameters, using 500 randomized queries per dataset and query type. Execution times of MaskSearch are shown, as baseline methods perform similarly to the queries analyzed in §IV-B, regardless of the specific query parameters.

- **Filter**: mask selection queries with a filter predicate $\texttt{CP}(mask, roi, (lv, uv)) > T$. For every query, roi is set as the foreground object bounding box in a mask generated by YOLOv5 [42]. lv and uv are randomly selected from $[0.1, ..., 0.9]$ and $uv > lv$. The count threshold T is randomly chosen from $[0, 1, ..., \text{total \# pixels}]$.
- **Top-K**: returns masks ranked by $\texttt{CP}(mask, roi, (lv, uv))$. 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 result order (`DESC` or `ASC`), is randomly selected for each query.
- **Aggregation**: returns images by $\texttt{mean}(\texttt{CP}(mask, roi, (lv, uv)))$ across two GradCAM [3] masks from different models. Parameters roi, lv, uv, and result order are randomized and k is set to 25.

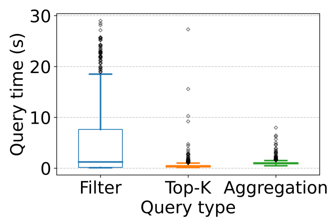
Figure 7 displays query execution times, showing median, range, IQR, and outliers. MaskSearch consistently outperforms baselines across all query types and parameters, even in the worst-case scenarios, as baselines load and process all masks regardless of query parameters.

Query times vary across and within query types. For example, *Filter* queries tend to have higher 75th percentile times due to less efficient filtering compared to *Top-K* and *Aggregation*, which leverage comparisons with dynamic top-k bounds. On *WILDS*, the number of masks pruned at the 75th percentile is 21,184 for *Filter*, 22,106 for *Top-K*, and 21,677 for *Aggregation*.

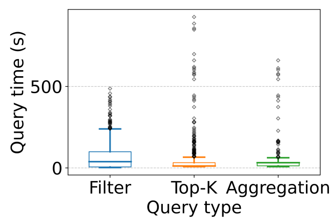
We observe that the query times tend to differ more significantly among queries with different parameters than within the same query type. Execution time differences within a query type stem primarily from variations in the fraction of masks loaded (FML). For *Filter* queries on *WILDS*, FML values at the 25th, 50th, and 75th percentiles are 0.002, 0.012, and 0.049, respectively. As discussed in §IV-D, FML determines MaskSearch's query time for a given dataset.

D. Query Time Analysis

This section analyzes factors affecting MaskSearch's query execution time using 1500 randomized *Filter* queries. Figure 8 shows that query execution time is proportional to the fraction of masks loaded (FML), defined as the ratio of masks loaded from disk to the total masks targeted by a query. Pearson's correlation coefficient between query time and FML

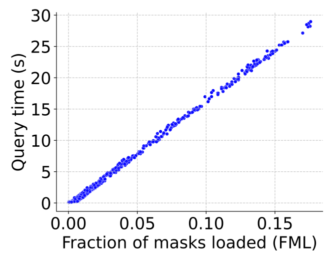


(a) *WILDS*

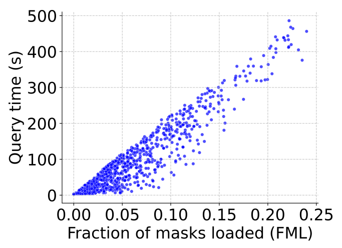


(b) *ImageNet*

Fig. 7: Query time of MaskSearch for different query types. Index size for MaskSearch: $\sim 5\%$ of dataset size.



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



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

Fig. 8: Relationship between end-to-end query time and the fraction of masks loaded (FML) for a query.

is 0.99 for *WILDS* and 0.96 for *ImageNet*, confirming that loading masks and computing `CP` values dominate query time.

FML depends on query parameters (region of interest roi, pixel value range (lv, uv), count threshold T), mask data, and index granularity. Specifically, FML represents masks not pruned or directly added to the result set during filtering, corresponding to *Case 3* in Step 2 of the filter stage.

Figure 9 illustrates how FML varies with index size, pixel value range (lv, uv), and count threshold T. Larger index sizes produce tighter bounds, reducing FML and query time. Variations in (lv, uv), roi, and datasets also impact FML due to differences in pixel value distributions. A higher number of buckets and smaller cell sizes lead to more precise bounds, albeit at the cost of increased index sizes. Optimal settings for these parameters vary across deployment scenarios.

In summary, query time is dictated by FML, which is influenced by query parameters, mask data, and index size. Larger indexes reduce query time but require more resources, reflecting a trade-off between index granularity and performance based on application needs.

E. Multi-Query Workload Performance

In this section, we evaluate MaskSearch on multi-query workloads with and without the incremental indexing technique (§III-F), which mitigates start-up overheads. As shown in §IV-F, querying masks is an iterative process where users issue multiple queries with varying parameters to explore and analyze the dataset. To simulate this, we generate workloads that reflect the user's exploration of masks with specific properties. We assume a user initially queries masks from certain classes and progressively explores masks from other classes. For example, when identifying spurious correlations (see §I), a user might first target classes with high false positive rates, then query

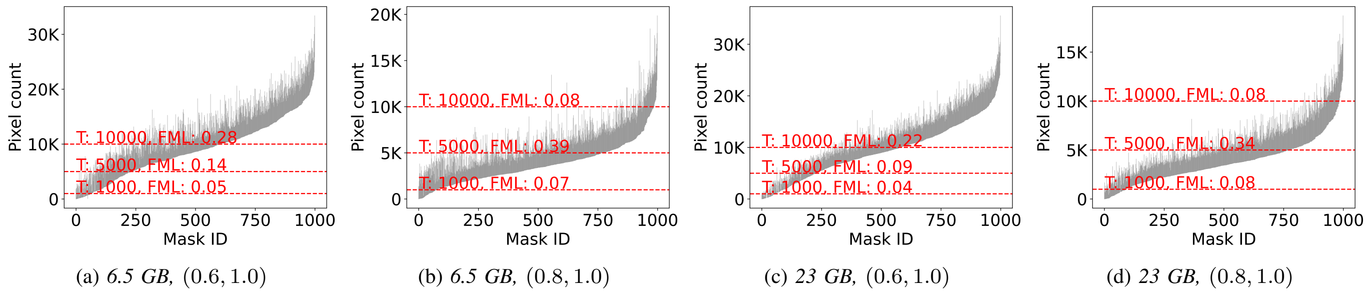


Fig. 9: Distribution of bounds for *ImageNet* masks computed by MASKSEARCH. Each subfigure represents a combination of (index size, (lv, uv)). Each vertical segment represents the lower and upper bounds of $\texttt{CP}(mask, roi, (lv, uv))$ for a single mask. FML is the fraction of masks loaded by MASKSEARCH given a predicate $\texttt{CP}(mask, roi, (lv, uv)) > T$, which is equal to the fraction of the vertical segments that intersect with the horizontal dashed line defined by T.

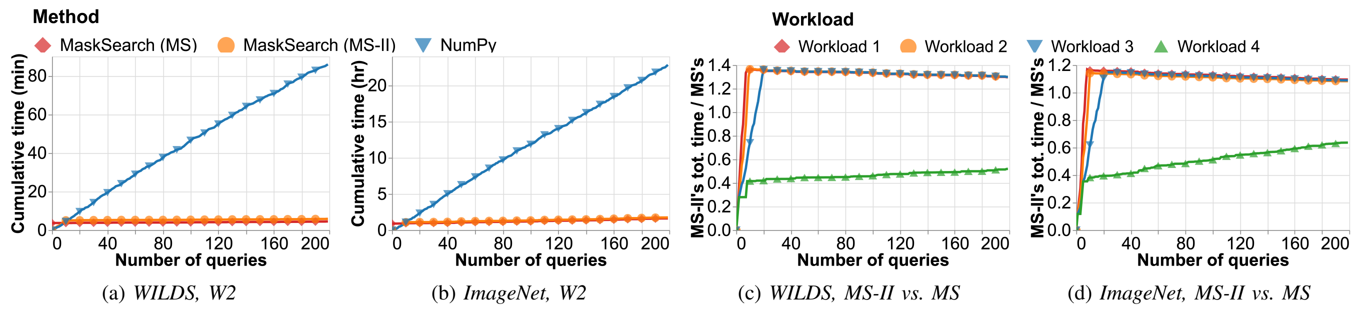


Fig. 10: Cumulative total time, incl. index building time and query time, for multi-query workloads. MS-II and MS refer to MASKSEARCH w/ and w/o incremental indexing, respectively. (a) and (b) show the total time for MS, MS-II, and NumPy for Workload 2; (c) and (d) show the ratio of the cumulative total 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.

images predicted as those classes using varying parameters (e.g., focusing on foreground or background objects).

We generate four workloads per dataset, each consisting of 200 *Filter* queries as in §IV-C. Each workload is characterized by a parameter p_{seen}, the likelihood of targeting previously queried masks. The number of masks targeted, n, is randomly chosen from $[0.1 \cdot N, 0.2 \cdot N, 0.3 \cdot N]$, where N is the total number of masks. To form the mask set, we sample n masks without replacement, comprising $p_{seen}\%$ previously queried masks and $(1 - p_{seen})\%$ new ones; if there are insufficient unseen masks, all remaining unseen masks are included and subsequent queries sample only seen masks. The workloads, labeled W1, W2, W3, and W4, have p_{seen} values of 0.2, 0.5, 0.8, and 1.0, respectively, where W1 demonstrates the most exploration and W4 focuses solely on previously queried masks.

Figure 10 presents MASKSEARCH's performance on the workloads for both datasets. MS-II refers to MASKSEARCH with incremental indexing, MS refers to MASKSEARCH without it, and NumPy represents methods that load and process all masks for each query. We measure the cumulative total time, which includes index building and query execution.

Figure 10 (a) and (b) present cumulative total times for W2; other workloads are omitted because MS and NumPy show similar trends. MS exhibits slow cumulative growth thanks to efficient query processing via the filter-verification framework. However, it incurs a start-up overhead due to the need for index building for all masks ahead of time, which is included with the 0-th query in the figure: 3 minutes for *WILDS* and 50 minutes for *ImageNet*. With incremental indexing, the first query takes 29 seconds for *WILDS* and 6 minutes for *ImageNet*. Each subsequent query takes on average 1.6 seconds for *WILDS* and 30 seconds for *ImageNet*.

In contrast, NumPy has no start-up overhead but suffers from rapidly increasing cumulative time. The indexing cost for MS is quickly amortized over queries, leading MS to outperform NumPy after roughly 10 queries, while MS-II eliminates start-up overhead and achieves comparable query times to MS.

Figure 10 (c) and (d) show the ratio of cumulative times between MS-II and MS. Initially, the ratio is 0 since MS's time includes full index building. The ratio grows rapidly because (1) MS-II must process unseen masks and build indexes, while (2) MS can answer queries efficiently using pre-built indexes. MS-II's slowest queries are only 20% to 40% slower than those of MS (with pre-indexing time amortized): on *WILDS*, 15 seconds for MS-II versus 11 seconds for MS; on *ImageNet*, 197 seconds versus 173 seconds. The ratio varies with workload—W1 shows the highest initial ratio due to its lower p_{seen} value, which requires more index building.

The ratio peaks once MS-II finishes indexing and accelerates subsequent queries. For W1, W2, and W3, the peak exceeds 1.0 because MS-II must load and process all masks on first query, and MS's batch indexing benefits from vectorization.

After peaking, the ratio decreases as MS-II's cumulative time grows similarly to MS's. In W4, the ratio remains below 1.0 since only 30% of masks are queried (6683 for *WILDS* and 399,351 for *ImageNet*), meaning MS-II avoids unnecessary indexing; once indexed, the ratio plateaus.

F. Real-World Use Cases

This section shows MASKSEARCH's real-world utility.

Improving Model Performance with MASKSEARCH on *WILDS*. This use case corresponds to Scenario 1 in §I. *WILDS* is a benchmark designed to evaluate the robustness of ML models to distribution shifts [27]. 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 downloaded from the *WILDS* repository. We first issued a query to MASKSEARCH to retrieve the top-50 masks (and their corresponding images) which have the fewest salient pixels (i.e., pixel value > 0.8) in their object bounding boxes (generated by YOLO [42]). 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 [28]. 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. In contrast, simply augmenting a random set of 50 images and retraining only increased the model's accuracy from 60% to 63%.

Understanding Vision Foundation Models with MASKSEARCH for Ophthalmology and Histopathology. We worked with a team of computational biologists who develop vision foundation models for ophthalmology [43] and histopathology [44]. They generated gradient-based saliency maps [45] 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 series of 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 256×256 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).

V. RELATED WORK

Image masks in ML tasks. Masks are widely used in ML to annotate image content, e.g., saliency maps [1], [2], [3], [4] and segmentation maps [6], [7], [8]. Practitioners use them for a variety of applications, including identifying maliciously attacked examples [17], [18], [19], detecting out-of-distribution examples [46], monitoring model errors [20], [21], [22], and performing traffic and retail analytics [23], [24]. These applications motivate the design of MASKSEARCH and could utilize MASKSEARCH's efficient query execution to quickly retrieve examples that satisfy the desired properties.

Data systems for ML workloads and queries. Numerous systems have been proposed to better support ML workloads and queries [47], [48], [49], [50], [51], [52], [53], [54], [55]. MASKSEARCH is related to systems that support the explanation and debugging of ML models [56], [57], [58], [59], [60]. Among these, DeepEverest [60] is the closest to MASKSEARCH. It is designed to support the efficient retrieval of examples based on neural representations, helping users better understand neural network behavior. While MASKSEARCH also focuses on efficiently retrieving examples, it targets queries based on mask properties rather than neural representations.

Image databases and querying. Many systems and techniques support efficient queries over image databases [61], [62], [63], [64], [65]. However, these methods are not optimized for our target queries. For example, VDMS [62] focuses on retrieving images based on metadata, while DeepLake [66] supports content-based queries but lacks support for querying based on aggregations over pixels. Array databases [30], [31] are designed for handling multi-dimensional dense arrays but 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 our target queries.

VI. CONCLUSION

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 around 5% of the size of the dataset, MASKSEARCH accelerates individual queries by two orders of magnitude and consistently outperforms existing methods on various multi-query workloads.

References

[1] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International conference on machine learning*. PMLR, 2017, pp. 3319–3328.

[2] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: removing noise by adding noise," *arXiv preprint arXiv:1706.03825*, 2017.

[3] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 618–626.

[4] B. Zhou, A. Khosla, L. A., A. Oliva, and A. Torralba, "Learning Deep Features for Discriminative Localization." *CVPR*, 2016.

[5] S. Singla, E. Wallace, S. Feng, and S. Feizi, "Understanding impacts of high-order loss approximations and features in deep learning interpretation," 2019.

[6] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," 2018.

[7] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick, "Segment anything," 2023.

[8] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015.

[9] D. Bhattacharjee, M. Everaert, M. Salzmann, and S. Süsstrunk, "Estimating image depth in the comics domain," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, January 2022, pp. 2070–2079.

[10] V. Patil, C. Sakaridis, A. Liniger, and L. 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.

[11] L. Oakden-Rayner, J. Dunnmon, G. Carneiro, and C. Ré, "Hidden stratification causes clinically meaningful failures in machine learning for medical imaging," in *Proceedings of the ACM conference on health, inference, and learning*, 2020, pp. 151–159.

[12] G. Plumb, M. T. Ribeiro, and A. Talwalkar, "Finding and fixing spurious patterns with explanations," 2022.

[13] A. Bissoto, M. Fornaciali, E. Valle, and S. Avila, "(de) constructing bias on skin lesion datasets," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2019, pp. 0–0.

[14] J. K. Winkler, C. Fink, F. Toberer, A. Enk, T. Deinlein, R. Hofmann-Wellenhof, L. Thomas, A. Lallas, A. Blum, W. 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*, vol. 155, no. 10, pp. 1135–1141, 2019.

[15] A. J. DeGrave, J. D. Janizek, and S.-I. Lee, "Ai for radiographic covid-19 detection selects shortcuts over signal," *Nature Machine Intelligence*, vol. 3, no. 7, pp. 610–619, 2021.

[16] Y. Ming, H. Yin, and Y. Li, "On the impact of spurious correlation for out-of-distribution detection," 2021.

[17] D. Ye, C. Chen, C. Liu, H. Wang, and S. Jiang, "Detection defense against adversarial attacks with saliency map," 2020.

[18] S. Wang and Y. Gong, "Adversarial example detection based on saliency map features," *Applied Intelligence*, pp. 1–14, 2022.

[19] C. Zhang, Z. Yang, and Z. Ye, "Detecting adversarial perturbations with salieny," in *Proceedings of the 6th International Conference on Information Technology: IoT and Smart City*, 2018, pp. 25–30.

[20] "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.

[21] D. KANG, J. GUIBAS, P. BAILIS, T. HASHIMOTO, Y. SUN, and M. ZAHARIA, "Data management for ml-based analytics and beyond."

[22] "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.

[23] DataFromSky, "Traffic monitoring - datafromsky," 2023. [Online]. Available: https://datafromsky.com/traffic-monitoring/

[24] ——, "Retail - datafromsky," 2023. [Online]. Available: https://datafromsky.com/retail/

[25] A. Das, H. Agrawal, C. L. Zitnick, D. Parikh, and D. Batra, "Human attention in visual question answering: Do humans and deep networks look at the same regions?" 2016.

[26] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[27] P. W. Koh, S. Sagawa, H. Marklund, S. M. Xie, M. Zhang, A. Balsubramani, W. Hu, M. Yasunaga, R. L. Phillips, S. Beery, J. Leskovec, A. Kundaje, E. Pierson, S. Levine, C. Finn, and P. Liang, "WILDS: A benchmark of in-the-wild distribution shifts," *CoRR*, vol. abs/2012.07421, 2020. [Online]. Available: https://arxiv.org/abs/2012.07421

[28] S. Teso, Ö. Alkan, W. Stammer, and E. Daly, "Leveraging explanations in interactive machine learning: An overview," *Frontiers in Artificial Intelligence*, vol. 6, p. 1066049, 2023.

[29] S. R. Hong, J. Hullman, and E. Bertini, "Human factors in model interpretability: Industry practices, challenges, and needs," *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, no. CSCW1, pp. 1–26, 2020.

[30] P. 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*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 963–968. [Online]. Available: https://doi.org/10.1145/1807167.1807271

[31] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The tiledb array data storage manager," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 349–360, 2016.

[32] W. Mann, N. Augsten, and P. Bouros, "An empirical evaluation of set similarity join techniques," *Proc. VLDB Endow.*, vol. 9, no. 9, p. 636–647, may 2016. [Online]. Available: https://doi.org/10.14778/2947618.2947620

[33] Y. Jiang, G. Li, J. Feng, and W.-S. Li, "String similarity joins: An experimental evaluation," *Proc. VLDB Endow.*, vol. 7, no. 8, p. 625–636, apr 2014. [Online]. Available: https://doi.org/10.14778/2732296.2732299

[34] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing neural network queries over video at scale," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1586–1597, aug 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137664

[35] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri, "Accelerating machine learning inference with probabilistic predicates," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1493–1508. [Online]. Available: https://doi.org/10.1145/3183713.3183751

[36] M. R. Anderson, M. J. Cafarella, G. Ros, and T. F. Wenisch, "Physical representation-based predicate optimization for a visual analytics database," *CoRR*, vol. abs/1806.04226, 2018. [Online]. Available: http://arxiv.org/abs/1806.04226

[37] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu, "Focus: Querying large video datasets with low latency and low cost," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 269–286. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/hsieh

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[39] "Product faq - milvus," https://milvus.io/docs/product_faq.md, 2024, accessed: 2024-01-22.

[40] "Limits - pinecone," https://docs.pinecone.io/docs/limits, 2024, accessed: 2024-01-22.

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

[42] G. Jocher, "YOLOv5 by Ultralytics," May 2020. [Online]. Available: https://github.com/ultralytics/yolov5

[43] Z. Liu, H. Xu, A. Woicik, L. G. Shapiro, M. Blazes, Y. Wu, C. S. Lee, A. Y. Lee, and S. Wang, "Octcube: A 3d foundation model for optical coherence tomography that improves cross-dataset, cross-disease, cross-device and cross-modality analysis," *arXiv preprint arXiv:2408.11227*, 2024.

[44] H. Xu, N. Usuyama, J. Bagga, S. Zhang, R. Rao, T. Naumann, C. Wong, Z. Gero, J. González, Y. Gu *et al.*, "A whole-slide foundation model for digital pathology from real-world data," *Nature*, pp. 181–188, 2024.

[45] Z. Liu, E. Adeli, K. M. Pohl, and Q. Zhao, "Going beyond saliency maps: Training deep models to interpret deep models," in *Information Processing in Medical Imaging: 27th International Conference*. Springer, 2021, pp. 71–82.

[46] J. Hornauer and V. Belagiannis, "Heatmap-based out-of-distribution detection," 2022.

[47] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. Lindstaedt, A. Phani, B. Rath *et al.*, "Systemds: A declarative machine learning system for the end-to-end data science lifecycle," *arXiv preprint arXiv:1909.02976*, 2019.

[48] Y. Asada, V. Fu, A. Gandhi, A. Gemawat, L. Zhang, D. He, V. Gupta, E. Nosakhare, D. Banda, R. Sen, and M. Interlandi, "Share the tensor tea: How databases can leverage the machine learning ecosystem," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3598–3601, aug 2022. [Online]. Available: https://doi.org/10.14778/3554821.3554853

[49] H. Miao, A. Li, L. S. Davis, and A. Deshpande, "Modelhub: Deep learning lifecycle management," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 1393–1394.

[50] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia, "Modeldb: a system for machine learning model management," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2016, pp. 1–3.

[51] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran, "Accelerating human-in-the-loop machine learning: Challenges and opportunities," in *Proceedings of the second workshop on data management for end-to-end machine learning*, 2018, pp. 1–4.

[52] D. He, S. C. Nakandala, D. Banda, R. Sen, K. Saur, K. Park, C. Curino, J. Camacho-Rodríguez, K. Karanasos, and M. Interlandi, "Query processing on tensor computation runtimes," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 2811–2825, sep 2022. [Online]. Available: https://doi.org/10.14778/3551793.3551833

[53] A. Phani, B. Rath, and M. Boehm, "Lima: Fine-grained lineage tracing and reuse in machine learning systems," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1426–1439.

[54] G. Gharibi, V. Walunj, R. Alanazi, S. Rella, and Y. Lee, "Automated management of deep learning experiments," in *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, 2019, pp. 1–4.

[55] F. Del Buono, M. Paganelli, P. Sottovia, M. Interlandi, and F. Guerra, "Transforming ml predictive pipelines into sql with masq," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2696–2700. [Online]. Available: https://doi.org/10.1145/3448016.3452771

[56] T. Sellam, K. Lin, I. Huang, M. Yang, C. Vondrick, and E. Wu, "Deepbase: Deep inspection of neural networks," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1117–1134.

[57] W. Wu, L. Flokas, E. Wu, and J. Wang, "Complaint-driven training data debugging for query 2.0," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1317–1334.

[58] P. Mehta, S. Portillo, M. Balazinska, and A. Connolly, "Toward sampling for deep learning model diagnosis," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1910–1913.

[59] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia, "Mistique: A system to store and query model intermediates for model diagnosis," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1285–1300.

[60] D. He, M. Daum, W. Cai, and M. Balazinska, "Deepeverest: Accelerating declarative top-k queries for deep neural network interpretation," *Proc. VLDB Endow.*, vol. 15, no. 1, p. 98–111, sep 2021. [Online]. Available: https://doi.org/10.14778/3485450.3485460

[61] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel *et al.*, "Finding a needle in haystack: Facebook's photo storage." in *OSDI*, vol. 10, no. 2010, 2010, pp. 1–8.

[62] L. Remis and C. W. Lacewell, "Using vdms to index and search 100m images," *Proc. VLDB Endow.*, vol. 14, no. 12, p. 3240–3252, jul 2021. [Online]. Available: https://doi.org/10.14778/3476311.3476381

[63] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. 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*, vol. 28, no. 9, pp. 23–32, 1995.

[64] A. N. Bhute and B. Meshram, "Content based image indexing and retrieval," *arXiv preprint arXiv:1401.1742*, 2014.

[65] R. Schettini, G. Ciocca, S. Zuffi, I. Tecnologie, I. Multimediali, and C. Ricerche, "A survey of methods for colour image indexing and retrieval in image databases," 02 2001.

[66] S. Hambardzumyan, A. Tuli, L. Ghukasyan, F. Rahman, H. Topchyan, D. Isayan, M. McQuade, M. Harutyunyan, T. Hakobyan, I. Stranic, and D. Buniatyan, "Deep lake: a lakehouse for deep learning," 2022.