Using VDMS to Index and Search 100M Images

Luis Remis
ApertureData
luis@aperturedata.io

Chaunté W. Lacewell
Intel Labs
chaunte.w.lacewell@intel.com

ABSTRACT

Data scientists spend most of their time dealing with data preparation, rather than doing what they know best: build machine learning models and algorithms to solve previously unsolvable problems. In this paper, we describe the Visual Data Management System (VDMS), and demonstrate how it can be used to simplify the data preparation process and consequently gain in efficiency simply because we are using a system designed for the job. To demonstrate this, we use one of the largest available public datasets (YFCC100M), with 100 million images and videos, plus additional data including machine-generated tags, for a total of about ~12TB of data. VDMS differs from existing data management systems due to its focus on supporting machine learning and data analytics pipelines that rely on images, videos, and feature vectors, treating these as first class citizens. We demonstrate how VDMS outperforms well-known and widely used systems for data management by up to ~364x, with an average improvement of about 85x for our use-cases, and particularly at scale, for a *image search* engine implementation. At the same time, VDMS simplifies the process of data preparation and data access, and provides functionalities non-existent in alternative options.

PVLDB Reference Format:
Luis Remis and Chaunté W. Lacewell. Using VDMS to Index and Search 100M Images. PVLDB, 14(12): 3240-3252, 2021.
doi:10.14778/3476311.3476381

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/luisremis/visual_storm/tree/master/yfcc100m.

1 INTRODUCTION

Visual computing workloads performing analytics on images and/or videos have become prolific across a wide range of application domains. This is in part due to the growing ability of machine learning (ML) techniques to extract information from the visual data which can subsequently be used for informed decision making [28]. The insights this information can provide depend on the application: retail vendors might be interested in knowing which are the most visited areas of their stores using security video feeds as input, or a doctor might want know the effect of a specific treatment by looking at the changes in size of a tumor from a brain scan.

Despite the increasing use of visual data processing, there has been very little research on the *management* of visual data. Most of the current storage solutions for visual data are an ad-hoc collection of tools and systems, that are re-purposed and adapted to work with visual data. The approach of re-purposing and integrating solutions not designed for a task results in resource utilization inefficiencies[2]. The combination of systems is required to manage both *metadata* and *visual data*. We use *visual data* to refer to any pixel-data (images, videos, frames in a video, etc.), and feature vectors (a.k.a. descriptors, or embeddings), which are representations of the pixel data. We use *metadata* to refer to any data that is important within the context of specific applications. Information about a patient (name, last name, unique identifiers), or about a clinic (name, location, etc.) are examples of what we refer to as *metadata* in the context of a health-care IT system. One can think of *metadata* as the data an application would store in rows and tables following the relational model.

To illustrate the point on the need for multiple systems to build the data infrastructure for applications, consider a ML developer constructing a pipeline for extracting brain tumor information from existing brain images in a classic medical imaging use case. This requires assigning consistent identifiers for the scans and adding their *metadata* in a relational or key-value database[17]. If the queries require a search over patient information, then patients are associated with their brain scans. Finally, if the ML pipeline needs images with a different resolution than the original, there is additional compute diverted towards pre-processing the original images. All these steps require understanding different software solutions that provide various functionalities that can then be stitched together with many scripts for this specific use case. Moreover, if the pipeline identifies new metadata to be added for the tumor images, most databases make it hard to change the schema on the fly. As another example, many applications can be studied through the use of large and publicly available datasets. Applications include basic image search functionality (through the use of human-generated tags), advanced image search through the use of machine-generated tags and feature vectors[8, 27] for each image, and video summarization. For these use-cases, the usual first step consists on selecting a subset of the data before running any processing, and a large effort is devoted to filtering, curating, and pre-processing the data. Selecting subsets of data is by itself a time consuming task, as it involves loading all metadata into a solution that enables searching based on tags (relational database, graph database, csv files, etc.), and building the necessary pipelines for querying and retrieving the right data.

More generally, data scientists and machine learning developers usually end up building an ad-hoc solution that results in a combination of databases and file servers to store metadata and visual data (images, videos), respectively [30]. This is integrated with a set of custom scripts that tie multiple systems together, unique

*Luis Remis was a Research Scientist at Intel Labs until late 2019. Most of this work was done while at Intel Labs.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476381

not only to a specific application/discipline but often to individual researchers or development teams [22, 30]. These ad-hoc solutions make replicating experiments difficult, and more importantly, they do not scale well when deployed in the real-world. The reason behind such complexity is *the lack of a system that can be used to store and access all the data the application needs, including metadata, images, videos, and feature vectors.*

In this paper, we describe VDMS and show how it provides a comprehensive solution to the data management for applications that heavily rely on visual data. VDMS is an Open Source project designed to enable efficient access of visual data. To the best of our knowledge, a rich set of functionalities designed for visual data management, provided behind an integrated API, is unique to VDMS and we were unable to find a system with similar functionality. While there are a number of big-data frameworks [32, 33], systems that can be used to store metadata [19, 25, 31, 34], and systems that manipulate a specific category of visual data [2, 5], VDMS can be distinguished from them on the following aspects:

- *Design for analytics and machine learning*: By targeting visual data for use cases that require manipulation of visual information and associated metadata,
- *Ease-of-use*: By defining a common API that allows applications to combine their complex metadata searches with operations on resulting visual data, and together with full support for feature vectors. VDMS goes beyond the traditional SQL or OpenCV level interfaces that do one or the other.
- *Performance*: We show how a unified system such as VDMS can outperform an ad-hoc system constructed with well-known discrete components. Because of the capabilities we have built into VDMS, it handles complex queries significantly better than the ad-hoc system without compromising the performance of simple queries.

In order to evaluate VDMS in a realistic use case, we use the YFCC100M dataset[35]. The YFCC100M dataset is the largest public multimedia collection. It contains the metadata of around 99.2 million photos and 0.8 million videos from Flickr, plus expansion packs [1] that include a variety of multidimensional data, all of which were shared under one of the various Creative Commons licenses. We have used this dataset for multiple proof of concepts and applications within our research lab.

Overall, this paper makes the following contributions:

- We present the design and implementation of the Visual Data Management System. To the best of our knowledge, there are no other system capable of managing visual data and its metadata behind a unified API and that, at the same time, enable applications to operate over visual data (images, videos, feature vectors).
- We introduce the VDMS API in detail, designed for Machine Learning and visual data analytics at scale.
- We use our internal use-cases and implementations of visual data processing pipeline to make a comprehensive evaluation between common ad-hoc solutions that rely on multiple systems and VDMS.
- We use the largest public available image dataset, YFCC100m, together with machine generated tags for our real-world use case, providing a performance evaluation both from the point of view of high concurrency as well as scale (the database size, including images and metadata is 12TB in size).

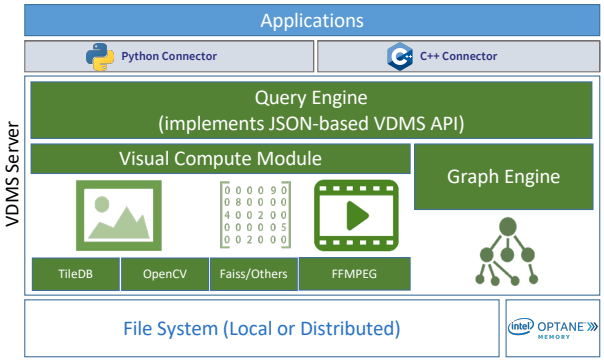

Figure 1: VDMS Architecture

2 RELATED WORK

There are several databases and data management systems that focus on enabling analytics or combining transactional and analytic workloads over large scale data, such as SciDB [5], BigTable [6], Shark [38], and Vertica [18]. While these systems do not focus on visual processing as a primary entity, they are valuable resources in distributing and analyzing very large scale data. A more concrete application involving visual data include the Facebook architecture for photos that combine Tao [37], Haystack [3], and f4 [24] for metadata, hot/recent data, and warm data respectively. While the social aspect of Facebook is logically suited to a graph, the bulk of their data is already stored in MySQL [25], so they chose to develop a graph-aware cache. A similar approach was taken by Grail [7], which argued that a syntactic layer written on top of a relational database could answer graph queries. Diamond [12, 29] exploited active disks to perform discard-based searching order to shrink the amount of data returned to a reasonable size. We accomplish the same goal by exploiting persistent memory to store a graph database for metadata.

3 VDMS DESIGN & IMPLEMENTATION

In this section, we further describe VDMS design principles and implementation, which was briefly introduced in previous work [28]. VDMS implementation is fully open-sourced [1].

VDMS implements the typical client-server architecture that handles client queries transactionally and concurrently, similar to what most common relational and non-relational database systems [6, 25, 34] use. The main difference between other data management systems and VDMS is that it goes beyond the typical supported data types (string, integers, floats, blobs, JSON-documents, etc.), recognizing visual entities (image, videos, feature vectors, etc.) as first class citizens. VDMS API enable users to insert, index, process, and query visual data, as well as provides full support for inserting, indexing and querying user-defined metadata. Users interact with

[1] https://github.com/IntelLabs/vdms

both metadata and visual data using a unified API, in a transactional manner.

VDMS provides a *graph model* abstraction with the traditional atomicity, consistency, isolation, and durability (ACID) properties expected from databases. This is, users interact with their objects (metadata, images, videos, etc.), as if these objects were in a connected graph. Graph represents an easier abstraction to model complex problems, making it very suitable for the data and access patterns shown by visual metadata, which can be easily mapped into application-level abstractions by developers [37]. For instance, abstractions like *BoundingBoxes* associated to images or videos can be easily represented using nodes and edges in a graph. This is the main reason why the team chose a graph model over a relational one for the implementation of VDMS API.

VDMS API provides a mechanism to insert and connect Images, Videos, BoundingBoxes, Frames, and Descriptors (feature vectors), together with any metadata associated with the objects. Each object is a node in the graph. The information associated to each visual object (image, video, etc.) is modeled as "properties" of the node in the graph. Users can query, filter, and retrieve these objects based on its properties. VDMS does not simply treat these objects as binary blobs of data, but rather understands them and the type of processing that is common for them, providing the ability to run processing on-the-fly, both at insertion and retrieval time. This is one of the main differentiating aspects when compared to other database systems, including relational databases. In a relational database, for instance, one can query *and compute* over values in a column only for basic data types (strings, integers, floats), and over the abstractions the relational model supports (tables, columns). This is, a SQL query can retrieve the *computed average* "salary" of all the employees in a company (stored using the table/columns abstraction), but cannot perform any computation over data stored as a blob or binary object. VDMS, because it recognized the nature of visual objects by design, provides the ability to *compute* on these visual objects (image, videos, etc.).

VDMS API also allows users to insert application-defined "Entities", that enable applications to model any use-case specific metadata. An "Entity" object (and its properties) is a node in the graph. For example, a user can define an Entity object of a class "Person", and *connect* this person to one or multiple image objects. Later, the user can retrieve the Entity object corresponding to a person, together with all the images *connected* to it. In some cases, users may need to apply different processing operations to the visual data for their application. We chose to support specific operations due to their frequent use in ML applications using visual data. Common operations supported for both images and videos are thresholding, cropping, and resizing. Additional operations such as flipping and rotation are supported for images and extracting frames by interval is supported for videos. In most ML applications, a subset of these operations are included in the preprocessing stage of model training and/or inferencing which supports majority of the research communities who analyze visual data and may benefit from replacing customized solutions with VDMS.

By providing the ability to store visual data objects together with application-defined entities and its properties, VDMS can manage all the data the application need behind a single, unified API. This means users can access all data (metadata, images, videos, etc.) from the VDMS API and even implement their own API/front-end on top of VDMS without adding any additional abstractions. This is in contrast with current applications that rely on a combination of multiple data management systems and APIs to access different portions of the data they need [22, 30, 30, 37].

Figure 1 depicts the high-level architecture of VDMS, which is composed by several sub-components, and a Query Engine that implements the unified API and hides all the complexity underneath. We first describe the VDMS API, and then describe its subcomponents and design decisions.

3.1 VDMS API

One of the most important differentiating aspects of VDMS is its API. VDMS is unique in recognizing visual entities (i.e., images, videos, etc.) as first class citizens. Thus, VDMS' API revolves around visual data operations and retrieval, but at the same time, enables applications to store any other application-specific metadata. VDMS API is easy to use and explicitly pre-defines certain primitives associated with metadata, images, videos, and feature vectors. Authors have paid particular attention to hide the complexities of our internal implementation and up-level the API to a JSON-based API, which is very popular across various application domains. By defining a new JSON-based API, there is a trade-off between expressiveness (compared to well-established query languages like SPARQL, Gremlim, or even SQL) and the ability to natively support visual data operations. However, we believe it is possible for our API to achieve similar levels of expressiveness compared to more mature query languages over time.

Listing 1 shows a sample query for inserting an image, properties associated with the image, and metadata to VDMS. The metadata, in this case, is the information about the "autotag", which is provided by the dataset in this specific *image-search* application. In this example, the transaction inserts an application-defined "Entity" of the class "autotag", with the "name" property being *alligator*, inserts an Image with its "latitude" and "longitude", stores the image as a JPG (VDMS will transcode if needed), and creates a connection between the Image and the Entity, with a "prob" property (which indicates the probability of that image containing an object of type *alligator*) with a specific value. This query is performed *transactionally*: either all the image, metadata and connection are inserted, or none of them are and an error is returned. Note that no schema needs to be defined in advance. The Entity of class "autotag", with its properties, are declared and added at insertion time, without any need to define a schema of objects (and its properties) before hand. This is a benefit over relational databases which require the user to identify how data is divided into different tables and determine the relationship between tables.

Listing 2 shows another sample query, in this case, for retrieval. In this particular example, the transaction retrieves all the images of *alligators* with probability higher than 0.66, filter by latitude and longitude within 1 degree, apply a resize operation to make the images 224x224, rotate the images 45.34 degrees, and return the images as "png" files. It is important to note how the API natively supports basic building blocks of visual data processing, like resize, rotation, or transcoding (changing output formats and encodings). Moreover, because VDMS is in control of the metadata and the

```
"AddEntity"{
    "_ref" : 1,
    "class": "autotag",
    "properties": {
        "name": "alligator"
    }
},
"AddImage":{
    "_ref": 2,
    "properties": {
        "latitude":    36.23433,
        "longitude": -116.80666
    },
    "format": "jpg"
},
"AddConnection": {
    "ref1": 1,
    "ref2": 2,
    "properties": {
        "prob": 0.7653
    }
}
```

Listing 1: Sample Query for Image Insertion - The query expresses the following: Insert an Entity of the class "autotag", with the "name" property being *alligator*, insert an Image with its "latitude" and "longitude", store the image as a JPG, and create a connection between the image and the "autotag", with a property "prob" (which indicates the probability of that image containing an object of type *alligator*).

```
"FindEntity"{
    "_ref" : 1,
    "class": "autotag",
    "constraints": {
        "name": ["==", "alligator"]
    }
},
"FindImage":{
    "format": "png",
    "constraints": {
        "latitude": [">=", 36.23433,
                     "<=", 38.23433]
        "longitude":[">=", -114.80666,
                     "<=", -116.80666]
    },
    "operations": [{
            "type": "resize",
            "height": 224,
            "width":  224,
        }, {
            "type": "rotate",
            "angle": 45.34
    }],
    "link": {
        "ref":1,
        "constraints": {
            "prob": [">=", 0.66]
        }
    }
}
```

Listing 2: Sample Query for Image Retrieval - The query expresses the following: Find all the images connected to the autotag *alligator* with probability higher than 0.66, filter the images by latitude and longitude within 1 degree, apply a resize operation to make the images 224x224, rotate the image 45.34 degrees, and return the images as "png" files.

image data, re-ordering or pre-processing operations can be done transparently, increasing the performance of the retrieval process. The API allows interaction with metadata, images, videos, bounding boxes, frames, feature vectors, and more in a similar fashion, and it is fully documented on the project's Github wiki [2]. The visual data pre-processing operations supported in VDMS are generic and application independent. By design, we only include pre-processing operations that are not specific to one domain, but rather general for visual analytics.

3.2 Graph Engine

The Graph Engine used in VDMS is the *Persistent Memory Graph Database* (PMGD). PMGD was developed at Intel Labs, and was designed and optimized for persistent memory technologies like Intel Optane [13], which promise storage providing nearly the speed of DRAM and the durability of block-oriented storage. PMGD is also fully open-sourced [3]. PMGD implements an in-persistent-memory graph database optimized to run on a platform equipped with persistent memory, but also provides the option to run on SSD and ext4 filesystem using OS support to provide transactional guarantees (through *msync*). PMGD provides a property graph model of data storage with the traditional atomicity, consistency, isolation, and durability (ACID) properties expected from databases. Specifically, the database remains consistent, each transaction either completes entirely or not at all, and the effects of a completed transaction survives any failure. In cases where the system may crash or fail, PMGD has the ability to recover to a consistent state without loading its content into memory. This is done through an undo log, chosen because it is optimistic and it handles reads after writes naturally. Upon failures, PMGD consults the log to restore the database to its state just past the last committed transaction. PMGD comes as a C++ library that VDMS uses for implementing its query engine. Because PMGD already supports most of the graph abstractions we wanted to see on VDMS API, it was our preferred option. Also, our internal evaluation shows that PMGD is faster than other graph databases, including Neo4j[23]. In this paper, the evaluation uses SSDs instead of persistent memory. Hence, PMGD performance evaluation will be published separately to highlight the advantages of using persistent memory for visual data.

3.3 Visual Compute Module

The Visual Compute Module was designed and implemented to provide an internal abstraction layer for interacting with visual data. It enables the query engines to coordinate and perform visual data handling and processing (i.e., basic building block operations like crop, resize, etc. for images/videos and k-nearest neighbor search for feature vectors), shown in Figure 1. For traditional formats (jpg, png, tiff, mp4, etc.), the interface is an abstraction layer over OpenCV. However, it also provides a way to use novel formats that are better suited for visual analytics: a novel, array-based lossless image format. This format is built on the array data manager

[2] https://github.com/IntelLabs/vdms/wiki/API-Description
[3] https://github.com/IntelLabs/pmgd

TileDB [26] and is well suited for images that are used in visual analytics. Note that, even if VDMS currently provides support for array-based lossless image format, we do not use it as part of this evaluation. This work focuses on a more direct comparison with a combination of alternatives, and thus we use the traditional format (jpg and png). The performance comparison between the array-based lossless image format we developed and other similar formats (like png or tiff) are outside of the scope of this evaluation.

VDMS also provides full support for video storage and operations, in a similar way it does for images. This includes support for encoding, decoding, and transcoding of *mp4*, *avi*, and *mov* containers, as well as support for *xvid*, *H.263* and *H.264* encoders. This is supported through the use of either OpenCV [4] or *libffmpeg*[21], or both, plus additional implementation to support fast random access to video frames. All operations supported for images in VDMS are also supported at the video and frame level of the API. On top of that, there are a number of video-specific operations that are supported, such as the interval operations, enabling users to retrieve clips at different frames-per-second (fps) versions of the video.

Another key differentiating factor of VDMS is that it allows the creation of indexes for high-dimensional feature vectors and the insertion of these feature vectors associated with entities, images, and/or videos. Feature vectors are intermediate results of various machine learning or computer vision algorithms when run on visual data. Feature vectors are also known as *descriptors* or *visual descriptors*. We use these terms interchangeably. These descriptors can be classified, labeled, and used to build search indexes. Feature Vectors support is provided through our implementation based on high-dimensional sparse arrays, also using array-based approaches. In addition, the Visual Compute Library provides a wrapper for another high-dimensional index implementation, Facebook's Faiss [15]. Users can, through our API, use different indexing techniques for feature vectors, depending on their application's need.

3.4 Client Library

The client library implements TCP/IP based connectors to the VDMS Server, similar to most databases[25, 31, 34]. Users can connect to VDMS and run queries using VDMS' API by defining a transaction using JSON objects. The client library provides a simple method that accepts a JSON string and an array or vector of blobs. Internally, the library wraps the query string and blobs using Google Protobufs [36] and sends it to the VDMS server. It also receives a similarly formed response from VDMS and returns it to the client. Currently, client libraries are implemented for Python and C++ client. The client libraries are lightweight, as they simply implement the communication protocol between the client and the server. This makes it easier for developers to implement similar client libraries using any other programming language of their choice.

4 EVALUATION

We have used the YFCC100M dataset to evaluate different aspects of our system. We use the images in the dataset and its associated metadata to implement an image-search engine based on properties associated with those images. This is a very common use-case we have encountered when building applications such as smart-retail, sports applications, and video summarization. For these type of applications, the starting point is usually a large set of data that must be curated before proceeding with the data processing (such as neural network training). In order to evaluate the different aspects of the performance on the image search, we have built different baselines following the methodology used in the industry, and following what we have done in the past in order to build an image search engine, which we describe in Section 4.3.

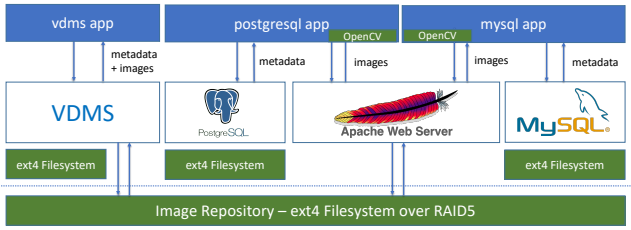

Figure 2: Comparison Systems: Logical view of the interaction between the client application with VDMS (left) and the baseline systems based on Apache Web Server and PostgreSQL (middle), or MySQL (right). The image repository is shared.

4.1 YFCC100M Dataset

The Yahoo! Flickr Creative Commons 100m (YFCC100M) dataset is a large collection of 100 million public Flickr media objects created to provide free, shareable multimedia data for research. This dataset contains approximately 99.2 million images and 0.8 million videos with metadata characterized by 25 fields such as the unique identifier, userid, date the media was taken/uploaded, location in longitude/latitude coordinates, device type the media was captured, URL to download the media object, and the Creative Commons license type information. The YFCC100M dataset also contains *autotags* provided as a set of comma-separated concepts such as people, scenery, objects, and animals from 1,570 trained machine learning classifiers [35]. Together with each *autotags*, there is a probability associated with each tag to indicate certainty of the classification. This is, an image can have the *autotags* "people", "person", "party", "outdoor", and each *autotag* assigned will be accompanied by a probability of that *autotags* being present in that image/frame. Given that there is no standard benchmark oriented towards visual data queries, we have built a series of queries to filter this dataset that is modeled after our internal use cases for many of the mentioned applications we have worked with.

4.2 Experimental Setup

Given that there are no other open-source systems that provide similar functionality and interfaces as VDMS (i.e., transactionally dealing with images and metadata behind a single interface), we have implemented two equivalent visual data management system as baselines, design specifically for the image search use case, and comprised of a combination of widely available, off-the-shelf components. The first baseline uses MySQL Server 5.7 (for storing

metadata), Apache Web Server 2.4.18 (as interface for image access), and OpenCV 3.3 (to provide pre-processing operations on images). We chose Apache Web Server to work as a file server (serving the image files), because it provides the lowest possible overhead (when compared to other object store system), at the expenses of providing less functionality (authentication, data integrity checking, etc.). The other baseline system replaces the MySQL Server 5.7 with a most advanced open-source relational database, PostgreSQL 9.5 [34]. We decided to use relational databases in the baseline systems instead of non-relational databases because of their maturity, and also because it is the most common tools used for storing and querying metadata [14, 20]:

- Relational databases support atomicity, consistency, isolation, and durability (ACID) while non-relational may compromise some ACID properties.
- The YFCC100m data is clearly structured, and can be modeled with relatively ease with a relational database.
- We need to efficiently collate and return metadata records, and SQL engines are very mature and optimized for that task.

The baseline implementations only partially replicate the functionalities that VDMS offers when it comes to image and metadata handling, and it was built for the purpose of an ad-hoc image search implementation. The baseline implementations are based upon internal tools used for ML-based pipelines for visual data, which is common practice in the industry [3, 37]. We have implemented a set of client-side applications that take care of retrieving the components from the different systems, and applies pre-processing operations when needed.

For all our experiments, we use two servers with Ubuntu 20.04, one hosting a VDMS server and another hosting the baseline implementations. Both servers have a dual-socket Intel® Xeon® Platinum 8180 CPU @ 2.50GHz (Skylake), each CPU with 28 physical cores with hyper-threading enabled, for a total of 112 logical cores per server. The server hosting MySQL and PostgreSQL has 256GB of DDR4 DRAM, while the server hosting VDMS has 194GB of DDR4 DRAM. We decided to run the VDMS server in the machine with less DRAM to make sure MySQL and PostgreSQL had no disadvantage. Previous evaluation indicated smaller footprint in the case of VDMS when compared to similar baselines based on MySQL. To build both MySQL and PostgreSQL on the same machine, we stored each database on separate SSD drives. Other than the difference in DRAM space and storage needed for two baselines, the machines are identical. The client application running the queries and measuring round-trip time is connected to the server through a 1GB wired link through a 10GB back-plane switch, same as both servers. The client application was implemented using Python 3 for both VDMS and the baselines. Figure 2 shows a logical view of the difference between the interaction of the client application (retrieves metadata and images) with VDMS (left), the PostgreSQL baseline (middle), and the MySQL baseline (right).

It is worth noting that the images are stored in a shared repository (ext4 filesystem on a RAID 5 configuration of 16TB) that both Apache WebServer and VDMS have direct access. In the case of videos, only the first frame is used for the image search. In the case of the baselines, metadata is stored in MySQL and PostgreSQL using an attached SSD disk. Even if VDMS has native support for Optane Persistent Memory, we do not use it in this experiment because of fairness of comparison with respect to MySQL and PostgreSQL, which were not designed for Persistent Memory type of storage. The benefits of Persistent Memory for metadata and a full evaluation of the PMGD subsystem is material for another paper, and outside the scope of this evaluation. For this experiment, in the case of VDMS we simply use a similar attached SSD disk to store metadata. Even if PMGD, the graph database used by VDMS, is designed for persistent memory, it can deliver good performance when using SSDs directly.

For the metadata, we built VDMS, MySQL, and PostgreSQL databases using the YFCC100M dataset with incremental database sizes to study the effects at scale. For simplicity, we named the database based on the approximate number of images it contains, as follows: 1M, 5M, 10M, 50M, 100M. The baseline systems have comparable number of elements. The exact number of images/elements in each database are shown in Table 1 and 2. The differences can be attributed to minor failures in data preparation/loading because of incomplete/inconsistent formatting, which is common in large datasets [10]. In our set up, that difference is very small: less than 0.253% in terms of number of elements (images and/or metadata information).

4.2.1 Data Representation. We detail the data representation for each of the systems we implemented:

VDMS: For each database size, we created an instance of VDMS using the image/video metadata, the machine-generated *autotags* associated with each image/video identifier, and the list of 1,570 *autotags*. Internally, that information is represented as a graph, where we have one node for each image, one node for each tag (always 1,570 tags), and a connection between each image and its respective tag(s). For instance, if an image has four *autotags* assigned, there will be four connections between that image and the different nodes for those *autotags*. The probability the *autotag* is present in an image is expressed as a property in the *connection* between the two nodes. Figure 3 shows an example on two images, two *autotags*, and the *connections* between those *autotags* and the images. Image id 23143252 has two *autotags* assigned: *Alligator* with probability 0.285, and *Lake* with probability 0.872. Image id 86756231, on the other hand, has a single *autotags* assigned: *Alligator* with probability 0.894. On average there are 8 tags assigned to each image so there will be around 8 times more connections than images, as shown in Table 1. Also, each image node will contain multiple properties associated with it (some of which are listed in Section 4.1). The number of nodes (representing images and *autotags*) are dependent on the database size and the *connections* are responsible for 90% of the elements in each database instance, as shown in Table 1. It is also important to note that we create indexes for the image identifier, *autotags* properties, and longitude/latitude coordinates to enable faster retrieval.

MySQL-Based System (*mysql*): Each MySQL database is created in a similar manner as VDMS but the data is represented as three tables, following the relational model: 1) *images* table: contains one row per image, and a column for each property associated with the images (some of which are listed in Section 4.1); 2) *taglist* table: contains one row per autotag element (always 1,570 rows); 3)

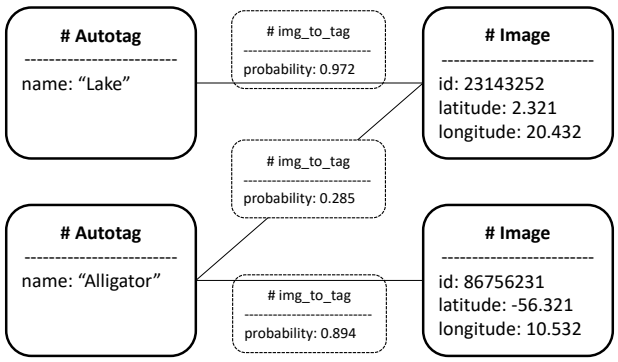

Figure 3: VDMS Data Representation Using a Property Graph: Example on two images and 2 *autotags* with their respective probabilities expressed in the *connection*. Image id 23143252 has two *autotags* assigned: *Alligator* with probability 0.285, and *Lake* with probability 0.872. Similarly, Image id 86756231 has a single *autotags* assigned: *Alligator* with probability 0.894.

Table 1: VDMS Database - Number of Elements

DB Name	# Images	# Connections	# TagList
1M	1,000,000	8,503,045	1,570
5M	5,000,000	42,505,478	1,570
10M	10,000,000	85,040,404	1,570
50M	50,000,000	425,162,070	1,570
100M	99,205,984	895,572,430	1,570

autotags table: contains one row per autotag assigned to an image. Each row contains a foreign key to the image, a foreign key to the tag, and the probability assigned to that tag belonging to that image. Given that there are 8 autotags, on average, per image, the *autotags* table has around 8 times the number of rows present in the *metadata* table, as can be seen in Table 2. Using a Python client and simple queries, the *taglist* table is read from the list of tags with an auto-incremented *tagid* as a primary key, and the metadata table is read from the YFCC100M metadata using the identifier as a primary key. The *autotags* table contains the generated autotags and probabilities for entries of the *images* table. To generate the table, we split the *autotags* data for each database by the image identifier and autotag into new files. The new files are read into the *autotags* table with the image identifier and *tagid* as foreign keys.

In an attempt to have the best MySQL configuration possible for this use case, we explore several parameters to increase the performance of both loading the data, as well as executing the queries. In particular, MySQL optimizes threads and transactions out-of-box, but it cannot handle the entire YFCC100M dataset without configuring specific parameters. When creating large databases, a data lock may occur to protect the data from concurrent updates [11]. To avoid this mechanism, we increased the buffer pool size to increase the amount of memory allocated to internal data structures. It is recommended to set the buffer pool size to 60-80% of the physical memory size [11, 25]. However, the time to build a database increased. We later changed the buffer pool size to be 16x the default value, where we saw the best performance.

By default, MySQL uses the available operating system threads to execute n requests in parallel where n is the number of background read/write I/O threads. Setting the respective parameters in the MySQL configuration file can limit the number of concurrent threads and the number of background threads. When a limit is set on the number of threads, and no threads are available, requests will go into a FIFO queue until threads are available to execute the request [11, 25]. We ran a few experiments investigating the effects of setting a limitation on the number of concurrent and background threads. We concluded that the default settings perform better for large databases instead of setting a limit. Therefore, we let MySQL automatically handle the concurrency.

PostgreSQL-Based System (*postgresql*): Each PostgreSQL database is created in the exact same manner as MySQL where data is represented as three tables: *images*, *taglist*, and *autotags*. PostgreSQL works well out-of-box and optimizations were not necessary, as in MySQL, to load or execute queries. When executing queries for larger databases, it is pertinent to have enough disk space for both the database and tablespace. The tablespace is associated with a database and stores the temporary files created within the database object [34]. When processing the larger databases, additional storage may be needed for the tablespace or a diskfull error may occur. To avoid this, we allocated a separate SSD disk specifically for the tablespace of all databases. This was required, specially for 100M database, in order to make the queries complete execution without failures, and represented a big disadvantage in terms of the disk space footprint during query execution.

By default, PostgreSQL can manage concurrent access to data well using Multiversion Concurrency Control (MVCC) and a Serializable Snapshot Isolation (SSI) level of transaction isolation. MVCC provides better performance by using SSI to guarantee querying data never blocks writing data, and vice versa [34]. The PostgreSQL configuration file initially limits the maximum number of concurrent connections to the database server to 100. We increased this limit to 200 to complete the concurrency analysis in Section 4.3.1.

In the case of VDMS, we did not attempt to tune any parameter to avoid unfairness in the comparison against the baselines. Unlike the baselines, VDMS can handle the entire YFCC100m dataset using the default parameters provided by the implementation.

4.2.2 Indexes: For all systems, we created indexes over the properties we used for search, such as name of *autotag*, and geo-location values. Building indexes for the right properties and objects is basic operation that would be present in any real-world deployment, and measuring performance without them would lead to useless analysis in our real-world applications and use cases.

4.2.3 Database Storage Footprint. The Graph Database used by VDMS (PMGD) was designed for performance, especially in environments where persistent memory is present. This design decision comes as a trade-off for storage footprint, which is noticeable in other PMGD evaluation we have run in the past. Because PMGD was not optimized for low storage footprint, authors feared that any improvement would come at the expenses of a significant increase on disk occupancy. We measured metadata disk footprint for that

Table 2: MySQL and PostgreSQL Databases - Number of Rows in each Table

	Table		
DB Name	images	autotags	taglist
1M	1,000,000	8,508,380	1,570
5M	4,987,379	42,425,905	1,570
10M	10,000,000	85,095,265	1,570
50M	50,000,000	425,446,208	1,570
100M	99,206,564	896,002,496	1,570

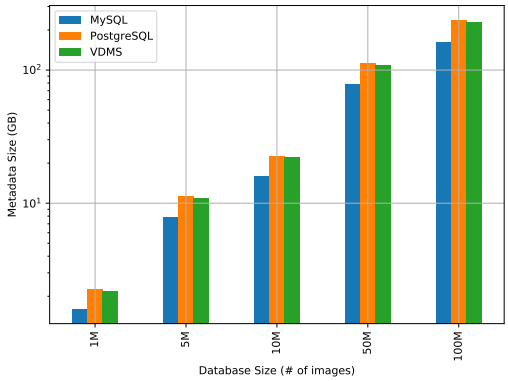

Figure 4: Data footprint (in GB) of MySQL, PostgreSQL, and VDMS databases.

reason, and show the results in Figure 4. PostgreSQL and VDMS require comparable amount of storage for metadata which is more than required by MySQL, shown in Figure 4. In the case of VDMS, this is space used to store information about each node/connection. VDMS required 37-41% more storage than MySQL for storing the same amount of metadata. On the other hand, PostgreSQL required 1-4% more storage than VDMS. The storage footprint may become a factor if storage is a limitation, but it should also be noted that metadata accounts for less than 2% of the overall database size even if we have a 41% increase in metadata size. For example, the largest database (both metadata and images) we built (100M) has around 230GB of metadata and 12TB of images. In systems where persistent memory is a scarce resource, the increased storage footprint of PMGD may represent a challenge. On the other hand, persistent memory is expected to be available in the order of TBs per server, which should fit the metadata of intensive use-cases[13].

4.3 Image Search

In order to evaluate VDMS and the baseline systems for our use-case queries, we implemented 6 queries that filter and retrieve a specific set of images. We chose these queries because they are the same that we use when filtering a cohort of images to be retrieved and processed from a large corpus of data. As we mentioned before, we took this approach mainly because we wanted to replicate systems we have built internally for our use cases, and also because of the lack of standard benchmarks that are oriented towards visual data retrieval.

The implementation of this evaluation, as well as all the queries, are available open-source for reproducibility [4], together with all the results of our evaluation [5].

We use the metadata associated with the images to filter the images. We use the *autotags* (as they contain information about the content of the image), and geo-location information (latitude and longitude) of the images for search and filtering. Note that, even if we use geo-location for our study, any other property assigned to the images can be used to refine the search in both VDMS and baseline implementations. On top of that, and for our use cases, we would like to extract more information about the content of the image through the use of ML, such as Convolutional Neural Networks [16]. For this, we resize the images to 224x224, which is the input layer size for popular variations of neural networks for object detection on images [9]. We used both ResNet and Yolov3 for object detection, both of which have the requirement of a downsized, lower resolution image as input.

To evaluate the access to metadata and images, we use the following queries, based on our internal use cases:

- *q1* - ***1tag_resize***: Retrieve images with one specific autotag and resize to 224x224.
- *q2* - ***1tag_geo_resize***: Retrieve images with one specific autotag, in a particular geo-location, and resize to 224x224,.
- *q3* - ***2tag_resize_and***: Retrieve images with two specific autotags (i.e. alligator AND lake), and resize to 224x224.
- *q4* - ***2tag_resize_or***: Retrieve images with either of two specific autotags (i.e. alligator OR lake), and resize to 224x224.
- *q5* - ***2tag_geo_resize_and***: Retrieve images with two specific autotags (i.e. alligator AND lake), in a particular geo-location, and resize to 224x224.
- *q6* - ***2tag_geo_resize_or***: Retrieve images with either of two specific autotags (i.e. alligator OR lake), in a particular geo-location, and resize to 224x224.

It is important to note that when querying for images with certain *autotags*, we also apply a filter using the probability. For instance, we only retrieve images with an autotag *alligator* and a probability higher than 92%. These probabilities are both present in VDMS (in the form of a property of the *connection* between the image and that *autotag*), as well as in *mysql* and *postgresql* (in the form of a column in the *autotags* table that links images with tags). In the case of VDMS, the query involves a graph traversal query that starts from the *autotag* node and ends in the images node, following *connections* between the image and that *autotag*). In the case of the baseline implementations, the query involves JOIN operations between the 3 tables.

Also, note that the size of the result (number of images retrieved) is linear with the size of the database. This is, if a query returns 100 images for the 1M database, it will return around 1000 images for the 10M database. This poses a problem when evaluating performance as the size of the database increase, and clearly understanding the measurements. Because of this reason, we control the number of returned images for all the databases using the probability of the *autotags* (higher probabilities returns less images), so that the queries in this experiment return a similar number of images

[4] https://github.com/luisremis/visual_storm/tree/master/yfcc100m
[5] https://github.com/luisremis/visual_storm/tree/master/yfcc100m/python/eval/results

for all database sizes. In other words, as the size of the database increase, we increase the probability threshold for the queries. We do this for both VDMS and the baselines, of course. This way, we remove bottleneck introduced by network bandwidth that would otherwise over-complicate the understanding of the results, and play in disadvantage for the baseline implementations.

Image search based on metadata is very expensive in large databases. Because of the large volume of data, the processing of the retrieved images is performed in parallel, using multi-core and/or distributed systems. For instance, a common implementation of an image processing pipeline would involve the use of distributed processing frameworks like Hadoop [33] or Spark [32]. Consequently, it is key that the data management system used supports concurrency, providing multiple workers with data in parallel. The ability to scale with the number of simultaneous clients is key for the applicability of visual data management systems like VDMS. Because of this, *we put emphasis on the analysis of* ***concurrency and throughput****, rather than latency.*

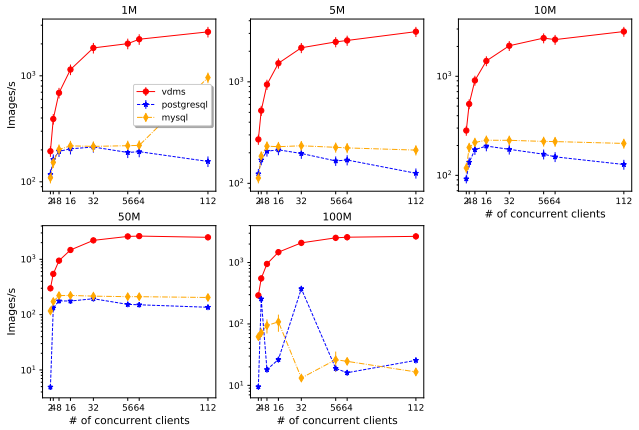

Figure 5: Concurrency Analysis on *q2* (*1tag_geo_resize*) from our use-case described in the Experimental Setup Section. The figures show the throughput (images per second) when retrieving resized versions of the images, as the number of concurrent clients increases. Hardware concurrency (number of physical cores in each system) is 56.

4.3.1 Concurrency Analysis. To analyze these results, one needs to compare the full-line (VDMS) versus the dotted-lines (two baselines). We start with a simple query that involves a simple metadata filtering over 1 tag and also geo-location, plus pre-processing operations. Figure 5 illustrates a concurrency analysis for *q2* described above (*1tag_geo_resize*), using VDMS and the baselines. Here we evaluate the concurrency of all systems, as the number of concurrent clients grows (x-axis) and as the size of the databases grow. Figure 5 shows throughput (images per second) when retrieving resized versions of the images, as the number of concurrent clients increase. The first thing to notice is that VDMS outperforms both baselines for all databases for this query by a large margin. For the baseline systems, in the case of 100M, the increase in the size of data seems to have a larger impact in performance. This result can be attributed to the increase in the complexity of the JOIN operation as the number of rows in the tables increases.

Another thing to notice is that, as the number of concurrent clients increases, VDMS throughput continues to increase up to 56 threads, which is the hardware concurrency of the system. Also, more parallelism after 56 threads does not increase the delivered throughput for the larger databases (50M and 100M) but there is a slight improvement in the other databases (1M, 5M, and 10M). On the other hand, both baselines seem to deliver less aggregated throughput after 16 threads except for the 100M case. In this case, *postgresql* has a performance spike at 32 clients and the performance for both baselines is less stable when compared to VDMS. Note that most of the throughput for the baseline systems are very close to each other. This is mostly an effect of the log-scale used, which is needed to clearly depict the difference between VDMS and the baselines. Here, the baselines are the full architectures described in Figure 2 (middle and right).

All queries run a resize operation on the image, an operation that requires decoding, resizing, and encoding the image before sending it back to the client. These operations are mainly compute bound, and that is the reason for the system to stop scaling beyond the number of physical cores. In contrast, the baseline does not scale nearly as well as VDMS, and we see that even after increasing concurrency, the increase in throughput is just about 2x. When comparing the case of 56 or 64 concurrent clients, VDMS delivers between 8x and 10X the throughput.

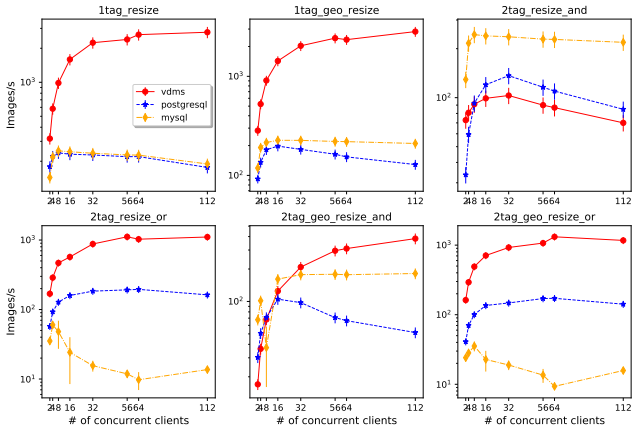

Figure 6: Concurrency Analysis for all the queries using the 10M database. Each figure shows the throughput (images per second) when retrieving resized versions of the images as the number of concurrent clients increases. Hardware concurrency (number of physical cores in each system) is 56.

We continue by looking at Figure 6 which shows the images per second delivered by each system for all queries using the 10M database, as the number of concurrent clients grows (x-axis). This figure takes a closer look at the concurrency for each of the six queries. VDMS consistently outperforms each of the baseline systems for queries *q1*, *q2*, *q4*, and *q6* as concurrency grows. This figure shows a similar trend as Figure 5 when it comes to concurrency. The VDMS throughput continues to significantly increase up to 56

concurrent clients, which is the hardware concurrency of the system, for all queries except *q3*. In this query, the throughput seems to decrease after 32 clients and both baselines outperform VDMS after 8 concurrent clients. Prior to 8 clients, VDMS has a slight advantage over *postgresql* which has a major performance improvement at 16 concurrent clients. On the other hand, *mysql* maintains its improvement in throughput over *postgresql* and VDMS over all concurrent clients. In the case of *q5*, the throughput of the baselines are less aggregated than those of VDMS up to 16 concurrent clients. When it comes to low concurrency, the baselines do as good and even better than VDMS with 2 or 4 concurrent clients. At 16 concurrent client, the performance of the *mysql* baseline begin to stabilize while the throughput of *postgresql* begins to degrade. However, as concurrency increases beyond 16 concurrent clients, the difference in throughput becomes clear, with VDMS reaching its peak performance at 112 concurrent clients.

In the case of VDMS, we see the performance in the case of *q3* and *q5* suffers significantly in comparison to the other queries. The reason for that lack of scalability lies on the query implementation: given that VDMS does not yet support operators that enable querying images that have both connections to a *tagA* and a *tagB*; we have to implement this transaction by doing 2 retrievals. This involves retrieving partial information in the first retrieval, applying an INTERSECTION operation in the client, and doing a second retrieval to bring the right metadata and/or images. The reason for this is a lack of operations that would enable this query to be run entirely on the server is not an inherent limitation to VDMS but rather just a missing implementation. In the case of *q4* and *q6*, it is worth noting that for the OR operation, there is no need for 2 retrievals (2 transactions). Rather, a single transaction is performed and the result filtered on the client. Future VDMS releases will add more of such operators (AND, OR, etc.) in order to prevent unnecessary retrievals and extra filtering on the client side.

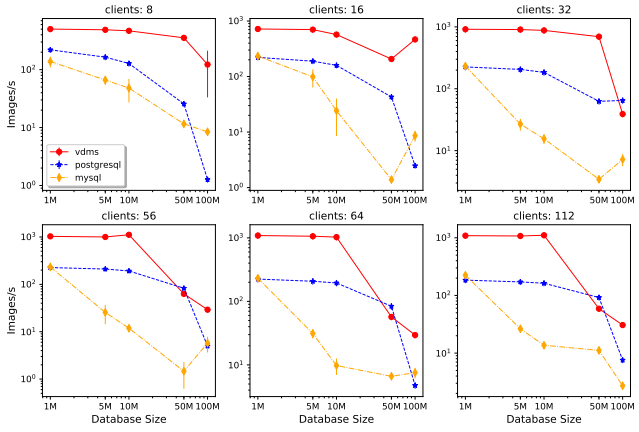

Figure 7: Concurrency Analysis on *q4* (*2tag_resize_or*) from our use-case described in the Experimental Setup Section. The experiments show the throughput as images per second for all systems (VDMS and two baselines) as the database size increases.

In the case of *q4* and *q6*, we see the performance for *mysql* degrades up to 64 concurrent clients, then there's an improvement with 112 concurrent clients. To further evaluate the concurrency of these queries with an OR operation, we continue by looking at Figure 7. This figure illustrates a concurrency analysis for *q4* (*2tag_resize_or*) which shows the images per second delivered by each system as the database size grows (x-axis). We show different number of concurrent clients in different figures for readability reasons. For this query, VDMS delivers higher throughput for databases 1M through 10M for all concurrent clients. As the number of clients and the size of database grows, the performance for this query drops and the performance of *postgresql* becomes more comparable to VDMS. Lets look at the case with 32 concurrent clients. In this case, the throughput of VDMS drops drastically with the 100M database and *postgresql* outperforms by a small margin. As the clients increase, we see this trade-off occurs with the 50M database instead of 100M while in the 100M database, VDMS outperforms both baselines. In this case, the throughput of *postgresql* drastically degrades for the 100M database and the performance of the baseline systems are more comparable.

When considering Figure 5 through Figure 7, there are many reasons why we generally see performance improvement, the main being that the entire operation (metadata query, image fetching and resizing) happens on the server side in the case of VDMS, within a single message exchange between the client and the server. Many of the inefficiencies that come with combining tools that were designed without visual data in mind simply disappear when building a tool that treats visual entities as first class citizens, as it is the case of VDMS. Another reason, which is quantifiable in the figures, is that VDMS sends resized (smaller) versions to the client instead of the full image to be resized on the client side (as is the case in the baselines). This is in contrast with the baseline implementations, where 2 rounds of blocking back-and-forth communication with the server is needed, as depicted in Figure 2. Note that one could argue that the opposite will happen when the resize operation retrieves an up-sampled (larger) version of the image instead of a down-sampled (smaller) one. In practice, retrieving an up-sampled version is not a common use case, given that up-sampling the image does not add any extra information that can help, for instance, improve the accuracy of a ML model. The case of down-sampling the original image is much more common and is the common practice when it comes to image processing through CNNs [9, 16].

4.3.2 Scalability Analysis. The next step in our analysis involves looking more deeply at how the performance scales with the number of elements in the database. Figure 8 shows the evaluation of the queries we analysed for our use case. Each figure shows the throughput when retrieving images, plus operations applied to images when applicable. The experiments show the performance of all systems (VDMS and two baselines) as the database size increases in terms of number of images. These queries were run using 56 simultaneous clients, and averaged over 10 runs. To analyze these plots, one needs to compare the full-line (VDMS) versus the dotted-lines (two baselines), each plot representing a different query.

For *q1* and *q2*, we can appreciate higher performance being delivered by VDMS when compared to the baselines, and how this improvement is maintained as the size of the database increase

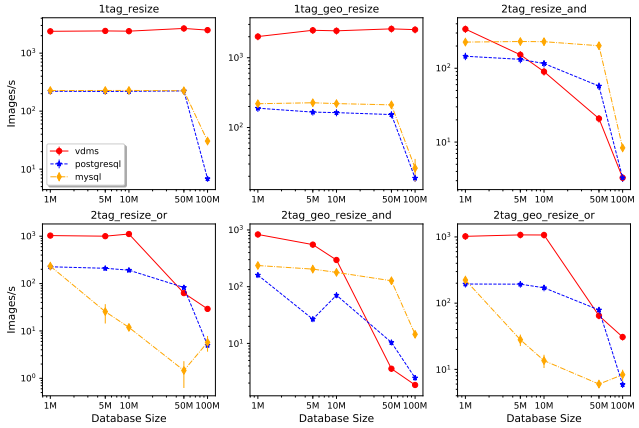

Figure 8: Throughput Analysis using all queries from our use-case described in the Experimental Setup Section. We show queries in different figures for readability reasons. The experiments show the performance of all systems (VDMS and the two baselines) as the database size increases. These queries were run using 56 simultaneous clients, and averaged out of 10 runs.

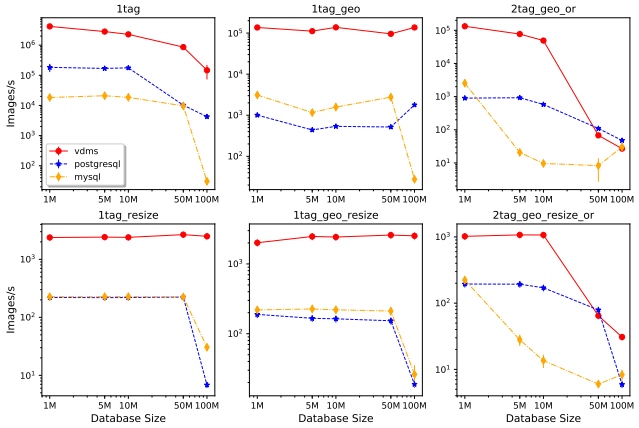

Figure 9: Throughput Analysis using a subset of the queries, with and without resize.

while, on the other hand, the baseline implementations have a drop in performance for the 100M database. For *q3*, we see that VDMS performs best when the database size is small (1M images), but as the database size increases, the performance degrades as well. For this query, *mysql* (and in some cases *postgresql*) outperforms VDMS for databases larger than 1M. This also occurs for *q5* but only for databases larger than 10M. This is entirely attributed to the 2-round process needed for this query, as we explained before which is more visible in the larger databases. It is interesting to note that adding filtering by geo-location (*q5*) decreases the performance as the database sizes scales. The behavior is different in the case of *q6*, which also filters by geo-location. This is attributed to the fact that OR queries involves processing larger results, and thus does not benefit from the filtering which is evident when compared to *q4*.

From the first 2 queries, as well as *q4* and *q6* (with exception to the 50M case), we clearly see that VDMS outperforms the baseline systems when retrieving visual data and applying operations. This is one of the most important finding, as it validates the design principles of VDMS, which aims to provide scalability and performance acceleration at the type of queries that require visual data access and transformations.

We designed VDMS with the idea in mind that pre-processing operations will be commonly performed, as visual data is generally pre-processed before analytics computation are performed on them (i.e., a resize plus normalization done on images before passing them through a neural network). Figure 9 shows how the query throughput varies when a pre-processing operations is applied (in this case, a resize operation). The first thing to notice is that a resize operation is a significant part of the overall operation (retrieval + preprocessing). Doing a resize operation as part of the retrieval process brings the throughput down by orders of magnitude. This is the case for both VDMS and the baselines. Nonetheless, VDMS is orders of magnitude faster when retrieving images both with and without resize operations. An important thing to notice is that, even if a resized image is smaller, and thus transfered faster between the server and the client, transferring the image as is (without resize) is faster in this case. This is a characteristic of the dataset, which has relatively small images (the average image is about 150KB in size). We have seen on other applications and datasets with higher resolution images that transferring resized (or cropped) images is significantly faster than transferring images as they are stored. An interesting aspect to notice is that, when removing the resize operations, differences between the 2 baseline systems become more noticeable. For instance, for an image retrieval based on a single tag, we see how *postgresql* outperforms *mysql*, but when adding geo-location filtering, we see *mysql* showing a small advantage. When the resize operation is performed, both baselines do very similar on 1tag based queries, and orders of magnitude worst than VDMS. Note that in the baseline systems, the resize happens on the client side, rather than on the server side. This is the common case in most analytics applications, where pre-processing operations happen on the client side (or wherever the analytics computation happens), rather than near where the data is located, as it is the case with VDMS. We kept both the server and client systems the same to avoid the computation capabilities difference complicating our understanding of the results.

Finally, Figure 10 summarizes the results comparing VDMS and *postgresql*. In comparison to *postgresql*, VDMS provides up to 364x speedup (for the case of *q1*), and an average improvement in throughput of about 85x. This is an impressive improvement over the two baseline systems. More importantly, we see that the speedup increases as the database size grows, showing that VDMS scales better than the baseline systems. We also see how *q3* and *q5* have poor performances and scalability when compared to the baselines as the database size grows, except in the case of the largest database (100M), where the baselines performed very poorly. This evaluation served the purpose of understanding the importance of VDMS server side operators that enable more complex queries for our use cases. The team will address the missing implementation

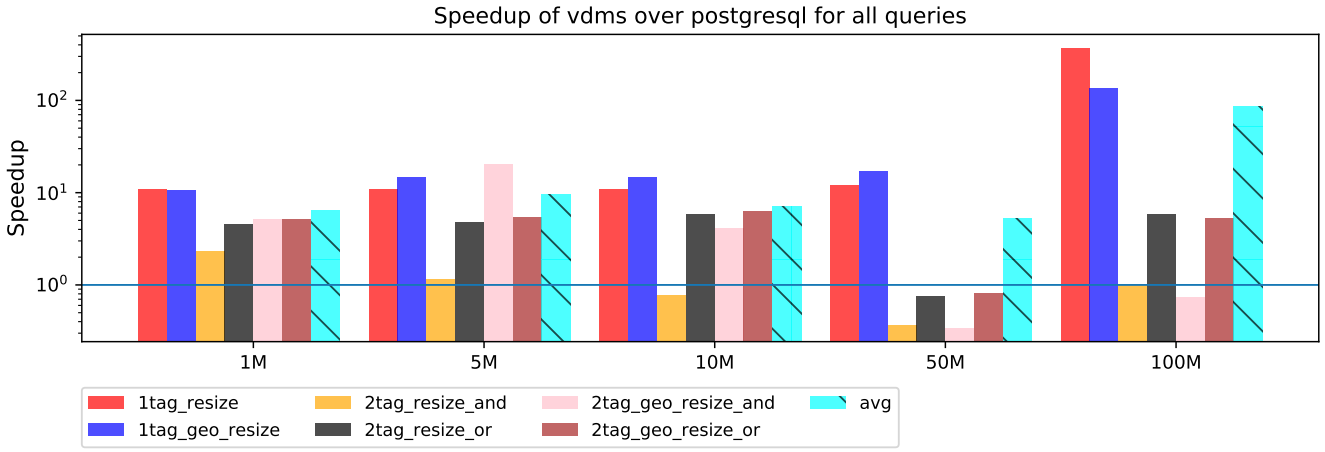

Figure 10: Summary of performance gains for all queries. We see up to 364x speedup (*q1*), and an average of about 85x. More importantly, we see that the speedup grow as the database size increases, showing that VDMS scales better than the PostgreSQL+Apache baseline.

as part of future work. Most of the performance improvements can be attributed to the design principles of VDMS, which aims to eliminate the need of combining and re-purposing systems that were designed to handle types of data other than visual. VDMS, by design, eliminates most of the inefficiencies that result from a forced integration of components designed for a different range of applications.

Because of space constraints, we are only showing a subset of our evaluation. More details about our comprehensive evaluation are available [6].

5 CONCLUSION

In this paper, we described VDMS design and implementation and show a comprehensive evaluation on our Image Search Application. We use one of the largest publicly available datasets: The Yahoo Flickr Creative Commons 100M (YFCC100M), together with the expansions packs that include machine-generated labels. We show how VDMS compares against a combination of industry standard systems, all of which are needed to replicate only a portion of VDMS' functionality. We see improvements up to 364x in certain queries, and an average improvement of about 85x when compared to PostgreSQL+Apache. When compared to MySQL+Apache, we see up to 96x speedup and an average improvement of 31x. The design of VDMS, which was conceived as a data management system that treats visual entities as first class citizens, can remove inefficiencies that result from re-purposing and combining solutions that were not designed for the job while providing simpler and richer interfaces. VDMS' easy-to-use interfaces outperform industry standard systems with a set of functionalities which, to the best of our knowledge, are not available in any other single data management solution for visual data. VDMS was designed for analytics and it can efficiently handle complex queries which can simplify the design of future applications that rely on visual data.

[6] https://github.com/luisremis/visual_storm/tree/master/yfcc100m/python/eval/results

6 FUTURE WORK

Based on the results of this evaluation and our experience with VDMS for this and other internal use cases, we have identified certain features that are needed to fully exploit the potential of the unified back-end for visual data infrastructure. In particular, we have identified the urgent need for a set of UNION/INTERSECTION interfaces to enable richer queries and improve performance, given the limitations discussed on 4.3.1. Even if this evaluation focuses on image search using metadata properties, VDMS has the capability to also perform similarity search based on feature vectors. The team will work on a evaluation of this feature as future work, as more work is needed to arrive at a comparable and fair baseline. Last, but not least, we recognize the need for a distributed implementation of VDMS. Most of the subcomponents and abstractions were design with a distributed architecture in mind, but also most of the applications we encountered in our internal use cases would fit on a single server plus a distributed file system. With a validation of the benefits of the abstractions provided by VDMS, the natural next step is working on scaling the system out.

ACKNOWLEDGMENTS

We would like to thank the many people that made this project possible and helped us through the process, as this work is the results of many efforts. We want to specially thank our senior technologists Nilesh Jain and Ravi Iyer for their full support and input during the duration of the project. We want to specially acknowledge Philip Lantz and Vishakha Gupta for their help with PMGD, key to loading large datasets into VDMS. We want to thank Jim Blakley for his input during the various phases of our project, and for advocating and promoting VDMS. We want to acknowledge the Intel Labs VDMS team for their efforts open-sourcing and maintaining the system. We want to thank Jason Gardner for his helping in setting up many of the servers and infrastructure needed to conduct our experiments.

REFERENCES

[1] Giuseppe Amato, Fabrizio Falchi, Claudio Gennaro, and Fausto Rabitti. 2016. YFCC100M-HNfc6: A Large-Scale Deep Features Benchmark for Similarity Search. In *Similarity Search and Applications*. Springer International Publishing, 196–209. https://doi.org/10.1007/978-3-319-46759-7_15

[2] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. 1998. The Multidimensional Database System RasDaMan. In *Proc. of the 1998 ACM SIGMOD* (Seattle, Washington, USA) *(SIGMOD '98)*. ACM, 575–577. https://doi.org/10.1145/276304.276386

[3] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. 2010. Finding a Needle in Haystack: Facebook's Photo Storage.. In *9th USENIX Symposium on OSDI*, Vol. 10. 1–8.

[4] Gary Bradski and Adrian Kaehler. 2013. *Learning OpenCV: Computer Vision in C++ with the OpenCV Library* (2nd ed.). O'Reilly Media, Inc.

[5] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 963–968.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[7] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. 2015. The Case Against Specialized Graph Analytics Engines.. In *CIDR*.

[8] Robert Fergus, Li Fei-Fei, Pietro Perona, and Andrew Zisserman. 2005. Learning object categories from google's image search. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, Vol. 2. IEEE, 1816–1823.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[10] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. 2011. Modeling and tolerating heterogeneous failures in large parallel systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[11] Geir Hoydalsvik. 2019. MySQL Connection Handling and Scaling. Retrieved July 23, 2021 from https://mysqlserverteam.com/mysql-connection-handling-and-scaling/

[12] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, Mahadev Satyanarayanan, Gregory R Ganger, Erik Riedel, and Anastassia Ailamaki. 2004. Diamond: A Storage Architecture for Early Discard in Interactive Search.. In *FAST*, Vol. 4. 73–86.

[13] IntelPR. 2015. Intel and Micron Produce Breakthrough Memory Technology. Retrieved July 23, 2021 from http://goo.gl/MUWm0W

[14] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. 2012. A Survey and Comparison of Relational and Non-Relational Database. *International Journal of Engineering Research and Technology* 1 (2012). Issue 6.

[15] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *CoRR* abs/1702.08734 (2017). arXiv:1702.08734 http://arxiv.org/abs/1702.08734

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[17] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1717–1722.

[18] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173* (2012).

[19] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *Proc. of the VLDB Endowment* 5, 12 (2012), 1790–1801.

[20] Ziqi Li. 2019. NoSQL Databases. https://doi.org/10.22224/gistbok/2018.2.10

[21] Libffmpeg. [n.d.]. FFMPEG Library. Retrieved July 23, 2021 from http://source.ffmpeg.org

[22] Ruben Mayer and Hans-Arno Jacobsen. 2020. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–37.

[23] Justin J Miller. 2013. Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, Vol. 2324.

[24] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's Warm {BLOB} Storage System. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 383–398.

[25] Oracle Co. [n.d.]. The world's most popular open source database. Retrieved July 23, 2021 from https://www.mysql.com/

[26] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB Array Data Storage Manager. *Proc. VLDB Endowment* 10, 4 (Nov. 2016), 349–360. https://doi.org/10.14778/3025111.3025117

[27] Jianbin Qin, Wei Wang, Chuan Xiao, and Ying Zhang. 2020. Similarity query processing for high-dimensional data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3437–3440.

[28] Luis Remis, Vishakha Gupta-Cledat, Christina R. Strong, and Ragaad Altarawneh. 2018. VDMS: An Efficient Big-Visual-Data Access for Machine Learning Workloads. *Systems for Machine Learning Workshop (SysML) at NIPS, Montreal, Canada* abs/1810.11832 (2018). arXiv:1810.11832 http://arxiv.org/abs/1810.11832

[29] Mahadev Satyanarayanan, Rahul Sukthankar, Lily Mummert, Adam Goode, Jan Harkes, and Steve Schlosser. 2010. The unique strengths and storage access characteristics of discard-based search. *Journal of Internet Services and Applications* 1, 1 (2010), 31–44.

[30] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015), 2503–2511.

[31] SingleStore, Inc. [n.d.]. SingleStore: The Single Database for All Data-Intensive Applications. Retrieved July 23, 2021 from https://www.singlestore.com/

[32] The Apache Software Foundation. [n.d.]. Apache Spark: Lightning-fast unified analytics engine. Retrieved July 23, 2021 from https://spark.apache.org/

[33] The Apache Software Foundation. [n.d.]. What is Apache Hadoop? Retrieved July 23, 2021 from https://hadoop.apache.org/

[34] The PostgreSQL Global Development Group. [n.d.]. PostgreSQL: The World's Most Advanced Open Source Relational Database. Retrieved July 23, 2021 from https://www.postgresql.org/

[35] Bart Thomee, Benjamin Elizalde, David A. Shamma, Karl Ni, Gerald Friedland, Douglas Poland, Damian Borth, and Li-Jia Li. 2016. YFCC100M. *Commun. ACM* 59, 2 (Jan 2016), 64–73. https://doi.org/10.1145/2812802

[36] Kenton Varda. 2008. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul* 72 (2008).

[37] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, et al. 2012. Tao: how facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 791–792.

[38] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 13–24.