Using VDMS to Index and Search 100M Images

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 99+ million images and videos, plus add-ons that include machine-generated tags and 4K dimensional feature vectors, for a total of ∼13TB 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 ∼35x, with an average improvement of about 15x for our use-cases, and particularly at scale. At the same time, VDMS simplifies the process of data preparation and data access, and provides functionalities non-existent in alternative options.

1 Introduction

Visual computing workloads performing analytics on videos and/or images 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 [22]. 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]. For example, 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. 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 which are typically larger. All these steps require understanding different software solutions that provide various functionalities that can then be stitched together with a script 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 [7] 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 cleaning 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 systems to store metadata and visual data (images, videos), respectively. This is integrated with a set of custom scripts that tie multiple systems together, unique not only to a specific application/discipline but often to individual researchers or development teams. 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 a completely 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 [24, 25], systems that can be used to store metadata [15, 18], 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 [26]. The YFCC100M 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 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.

We show how VDMS can be used as a centralize point for data management and data access even when having multiple modalities of data: Metadata, Image, Videos, and Feature Vectors.

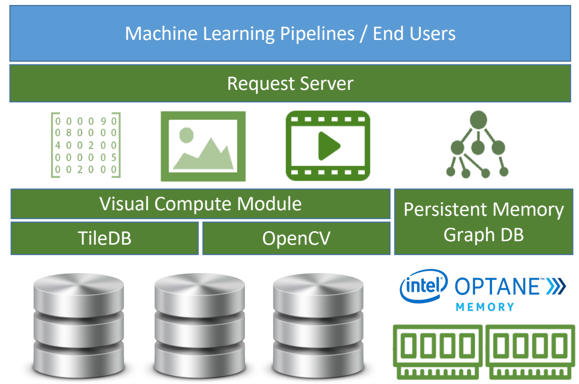


Figure 1: VDMS Architecture

2 VDMS Design & Implementation

In this section, we describe VDMS design principles and implementation, which was briefly introduced in previous work [22]. Figure 1 depicts the high-level architecture of VDMS. VDMS implements a client-server architecture that handles client requests concurrently, and coordinates query execution across the metadata and visual data components in order to return a unified response. Users interact with both metadata and visual data (i.e., images, videos, feature vectors) using a unified API, in a transactional manner.

The metadata component is the *Persistent Memory Graph Database* (PMGD) and the (visual) data component is our Visual Compute Module. The Visual Compute Module enables machine-friendly enhancements to visual data, exposing high-level abstractions to the *Request Server* for dealing with a variety of images and video formats (through OpenCV and ffmpeg), and different methods for indexing for feature vectors (including Facebook's Faiss [13], TileDB [21]). Finally, a main component of VDMS is in charge of implementing the API and orchestrating between the PMGD and the Visual Compute Module to serve client's requests. This component is the *Request Server*.

VDMS and its components are fully available as open source projects [1]. We briefly describe each of the main components as follows:

2.1 Persistent Memory Graph Database

We use the Persistent Memory Graph Database (PMGD) to provide an efficient storage solution addressing the increasing popularity of connected data and applications that benefit from graph-like processing. PMGD implements an in-persistent-memory graph database, PMGD, optimized to run on a platform equipped with persistent memory. PMGD provides a property graph model of data storage with the traditional atomicity, consistency, isolation, and durability (ACID) properties

[1] https://github.com/{Not shown during submission}

expected from databases. Graph represents an easier abstraction to model complex problems [28]. Moreover, the graph model makes it very suitable for the data and access patterns shown by visual metadata, which can be easily mapped into application-level abstractions by developers. For instance, abstractions like *BoundingBoxes* associated to images or videos can be easily represented using nodes and edges. With its natural ability to extend the schema very easily (due to the use of a property graph model), we can support new developments in machine learning that can lead to enhancements to existing metadata over time. These are the reasons the team chose a graph database over a relational database as the metadata management for the implementation of VDMS. PMGD is designed and optimized for persistent memory technologies like Intel Optane [11], which promise storage providing nearly the speed of DRAM and the durability of block-oriented storage.

2.2 Visual Compute Module

The Visual Compute Module was designed and implemented to provide an internal abstraction layer for interacting with visual data. It enables simple visual data handling and processing (i.e., basic building block operations like crop, resize, etc). 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 TileDB [21] and is well suited for images that are used in visual analytics.

VDMS 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* [17], or both. 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 TileDB. In addition, the Visual Compute Library provides a wrapper for another high-dimensional index implementation, Facebook's Faiss [13].

2.3 Request Server

Developers and users of machine learning frameworks and data science applications favor simpler interfaces to access and process data. They cannot be expected to deal with two different ways of interacting with information (metadata and visual data) instead of focusing on the algorithmic parts of their pipelines. VDMS takes care of coordinating client requests across the metadata and the visual data, and managing multiple clients through the *Request Server*. The *Request Server* is a key component of the system, as it implements VDMS' API and coordinates request and responses from the PMGD and Visual Compute Module subsystems. It decomposes the user queries into metadata and visual data requests, invokes the relevant calls behind the scene, and returns a coherent and unified response that is easy for the user to parse and interpret.

2.4 VDMS API

One of the most important differentiating factor of VDMS is its interface. VDMS is unique in recognizing visual entities (i.e., images, videos, etc) as first class citizens. Thus, VDMS provides an API that revolves around visual data operations and retrieval. VDMS API is easy to use and explicitly predefines 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. 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). The API allows interaction with metadata, images, videos, and feature vectors in a similar fashion, and it is fully documented on the project's Github wiki [2].

2.5 Client Library

The client library implements TCP/IP based connectors to the VDMS Server, similar to most database [18, 20]. Users can connect to VDMS and implement queries using VDMS'

[2]https://github.com/{Not shown during submission}

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

Listing 1: 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.

API by defining JSON commands conforming to the query protocol we have defined. 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 blob using Google Protobufs [27] and sends it to the VDMS server. It also receives a similarly formed response from VDMS and returns it to the client. The responses require JSON parsing on the client side for the metadata string that indicates how to interpret the blobs field. 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.

3 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 a baseline following the methodology used in the industry, and following what we have done in the past in order to solve the data search problem, which we describe in Section 3.3. We have also included performance evaluation on the Video and Feature Vectors functionality in the appendix for further reference, together with a description of the key aspects of those functionalities.

3.1 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 [26]. 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. We have also used feature vectors generated for every image and first frame of every video [1] to implement a similarity search. 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.

3.2 Experimental Setup

Given that there are no other open-source systems that provides similar functionality and interfaces as VDMS, we have

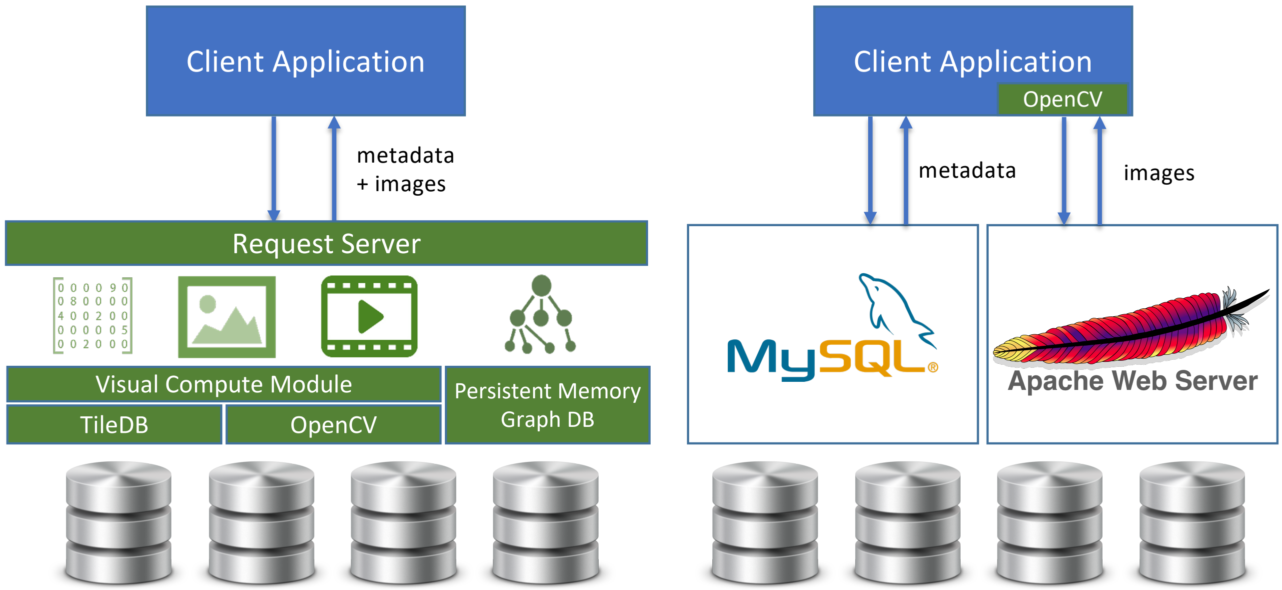


Figure 2: Comparison Systems: Logical view of the interaction between the client application with VDMS (left) and the baseline system (right).

implemented an equivalent visual data management system as a baseline, comprised of a combination of widely available, off-the-shelf components: 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 decided to use a relational database instead of a non-relational database because [12, 16]:

- Relational databases support atomicity, consistency, isolation, and durability (ACID) while non-relational may compromise some ACID properties.
- The YFCC100m data is clearly structured.
- We need to efficiently collate and return metadata records.

The baseline implementation only partially replicates the functionalities that VDMS offers when it comes to image and metadata handling, built for the purpose of an ad-hoc image search implementation. This implementation is based upon internal tools used for ML-based pipelines for media. 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 visual data workloads, building an approach like the one implemented as a baseline for this work is the common practice in the industry [3, 28].

For all our experiments, we use two servers, one hosting a VDMS server and another hosting the baseline implementation. 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 has 256GB of DDR4 DRAM, while the server hosting VDMS has 64GB of DDR4 DRAM. We decided to run VDMS server in the machine with less DRAM to make sure MySQL had no disadvantage, and because previous evaluation indicated smaller footprint in the case of VDMS when compared to similar baselines based on MySQL. Other than the difference in DRAM space, machines are identical. Both servers run Ubuntu 16.04. 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. Figure 2 shows a logical view of the difference between the interaction of the client application (retrieves metadata and images) with VDMS (left) and the baseline (right). The client application was implemented using Python 3 for both VDMS and the baseline.

It is worth noting that the images are stored in a shared repository (ext4 filesystem on a RAID 6 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. More information and evaluation on video-specific functionalities can be found in the appendix of this work. In the case of the baseline, metadata is stored in MySQL 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, which was 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, while still providing ACID-compliant transactions.

For the metadata, we built VDMS and MySQL databases using the YFCC100M dataset with incremental database sizes. For simplicity, we named the database based on the approximate number of images it contains, as follows: 1M, 5M, 10M, 50M, 100M. The VDMS and MySQL databases 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 failures in data preparation/loading because of incomplete/inconsistent formatting, which is common in large datasets [9]. In our set up, that difference is very small: less than 0.1% in terms of number of elements (images and/or metadata information).

3.2.1 Data Representation

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 property 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 3.1). 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/Baseline: 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

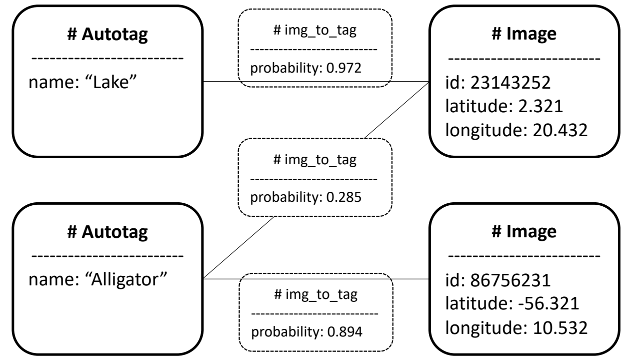


Figure 3: VDMS Data Representation Using a Property Graph: Example on two images and 2 *autotags* with their respective probabilities expressed in the *connection*. Image id 23143252 has two *autotags* assigned: *Alligator* with probability 0.285, and *Lake* with probability 0.872. Similarly, Image id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

Section 3.1); 2) *taglist* table: contains one row per autotag element (always 1,570 rows); 3) *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 [10]. 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 [10, 20]. However, the time to build a database increased. We later changed the buffer pool size to a multiple of the default value, i.e. 16x, which produced the best results for loading time.

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 [10, 20]. 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.

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

Table 2: MySQL Database - 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

3.2.2 Database Building Time

One of the first things we noticed is the difference in the time needed to build each database, where VDMS outperforms MySQL by a large margin. This analysis includes only the metadata, as the images are stored in a shared filesystem. Figure 4 illustrates how VDMS can build databases faster than MySQL, and how the speedup is sustained as the database size grows. Key difference in the build times are attributed to

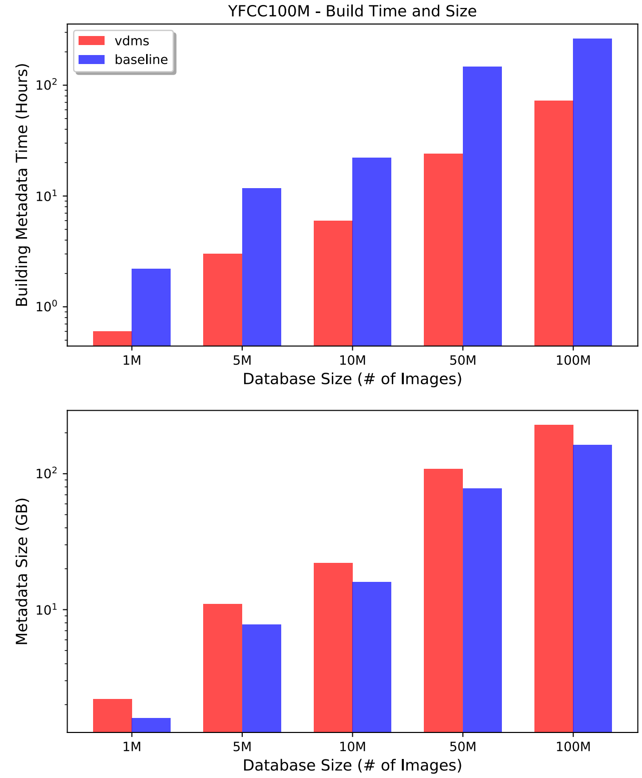


Figure 4: Time to build and size (in GB) of MySQL and VDMS databases.

the low-level implementation of how MySQL reads and stores data from the files, the optimizations (increased InnoDB pool size, etc.) needed to handle large datasets such as YFCC100M, and the efficiency of PMGD. On average, it took MySQL around 3.72x longer to build each database than VDMS.

3.2.3 Database Storage Footprint

Another important aspect to note is that VDMS requires more storage for metadata, shown in Figure 4. This is space used to store information about each node/connection. The Graph Database internal to VDMS (called 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 our results. VDMS required 30-41% more storage than MySQL for storing the same amount of metadata. This may become a factor if storage is a limitation, but it should also be noted that even if we have a 41% increase in metadata size, metadata accounts for less than 2% of the overall database 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 foot print 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 [11].

3.3 Images Search

In order to evaluate VDMS and the baseline on our use-case queries, we implemented 5 queries that filter and retrieve a specific set of images. We chose these queries because they represent typical use-cases where a cohort of images is to be retrieved and processed from a large corpus of data. As we mentioned before, we took this approach due to the lack of standard benchmarks that are oriented towards visual data retrieval. We use the metadata associated with the images to filter said images.

We use the *autotags* (as they contain information about the content of the image), and geo-location information (latitude/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 [14]. 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 [8].

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

- *q1* - ***1tag***: Find metadata/images with one specific autotag (i.e. alligator, lake, etc).
- *q2* - ***1tag_resize***: Find metadata/images with one specific autotag and resize to 224x224.
- *q3* - ***1tag_resize_geo***: Find metadata/images with one specific autotag, resize to 224x224, and in a particular geo-location (with a 20 degrees radius in latitude and longitude).
- *q4* - ***2tag_resize_geo_and***: Find metadata/images with two specific autotags (i.e. alligator AND lake), resize to 224x224, and in a particular geo-location (with a 20 degrees radius in latitude and longitude).
- *q5* - ***2tag_resize_geo_or***: Find metadata/images with either of two specific autotags (i.e. alligator OR lake), resize to 224x224, and in a particular geo-location (with a 20 degrees radius in latitude and longitude).

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 (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 implementation, the query involves JOIN operations between the 3 tables. The implementation of this evaluation, as well as all the queries, are available under the benchmarks branch of the VDMS project [3] and some examples of the queries can be found in the appendix of this paper.

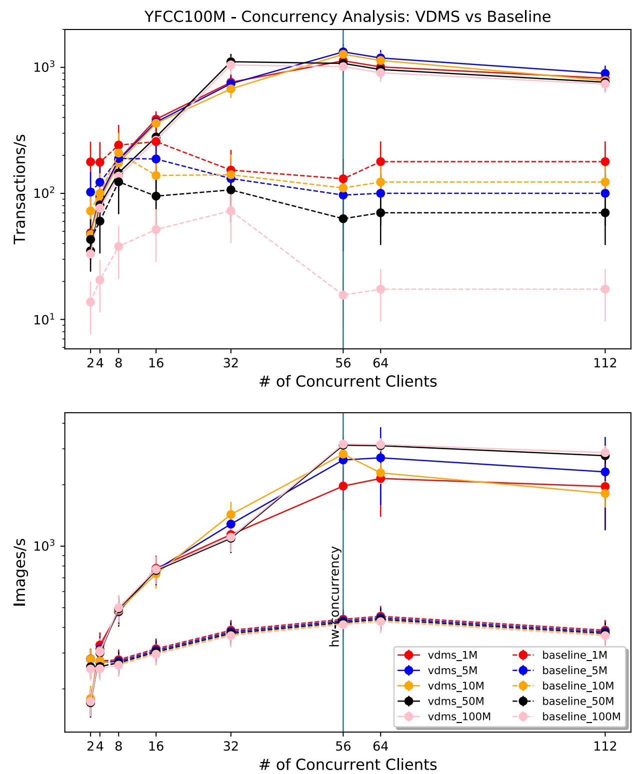


Figure 5: Concurrency Analysis on *q2* (*1tag_resize*). Hardware concurrency (number of physical cores in each system) is shown with a blue vertical line (hw-concurrency = 56). Top figure shows aggregated throughput (transactions per second) when retrieving only metadata associated with the images, as the number of concurrent clients increase. Bottom figure shows aggregated throughput (images per second) when retrieving resized versions of the images, as the number of concurrent clients increases.

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. Be-

[3]https://github.com/{Not shown during submission}

cause 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 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 baseline, of course. This way, we remove bottleneck introduced by network bandwidth that would otherwise over-complicate the understanding of the results.

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 [25] or Spark [24]. 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.

3.3.1 Concurrency Analysis

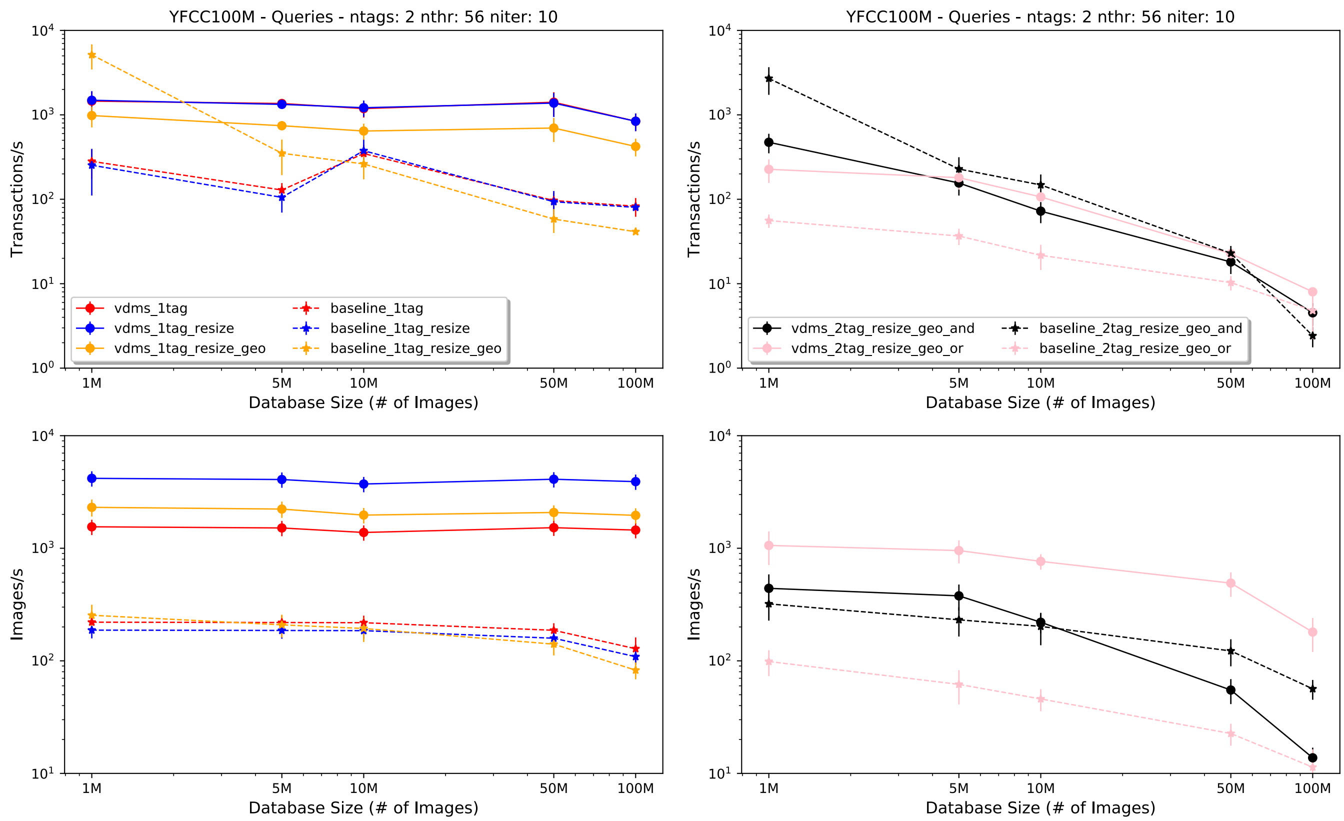
Figure 5 illustrates a concurrency analysis for *q2*(*1tag_resize*), described above, using both VDMS and the baseline. Here we evaluate the scalability of both systems, as the number of concurrent clients grows (x-axis) and as the size of the databases grows (each full-line represents a database size for VDMS and each dotted-line represents a database size for the baseline).

We start by analyzing Figure 5 (top), which shows aggregated throughput (transactions per second) when retrieving only metadata associated with the images, as the number of concurrent clients increase. The first thing to notice is that at low concurrency (2 to 8 concurrent clients), both systems show similar performance, except in the 100M case of the baseline. Note that, for this particular experiment, baseline translates directly to MySQL performance, as the metadata-only queries only involve running a query to MySQL. For the baseline system, 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, and it is actually slightly detrimental (112 concurrent clients case). On the other hand, the baseline seems to deliver less aggregated throughput after 16 threads, with an increase for 64 threads, but these effects are hard to interpret fully as the standard deviation in the measurements is high. We noticed throughout many experiments that MySQL results showed higher standard deviation, meaning less consistent and noise measurements, when compared to VDMS. We tried increasing the number of measurements and discarding outliers but were not able to get less noisy results.

We continue by looking at Figure 5 (bottom), which shows aggregated images per second delivered by each system. Again, we evaluate the scalability of both systems, as the number of concurrent clients grows (x-axis) and as the size of the databases grows (each full-line represents a database size for VDMS and each dotted-line represents a database size for the baseline). Note that most of the baseline dotted-lines 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 baseline. Here, the baseline is the full architecture described in Figure 2 (right). Figure 5 (bottom) shows a similar trend as the top figure when it comes to low concurrency. The baseline does as good and even better than VDMS with 2 or 4 concurrent clients. However, as concurrency increases beyond 4 concurrent clients, the difference in throughput becomes clear, with VDMS reaching its peak performance at 56 concurrent clients. This query (as well as *q3*, *q4* and *q5*) runs 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.

There are many reasons why we see this 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 for other use cases 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 bottom 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 baseline). This is in contrast with the baseline, where 2 rounds of blocking back-and-forth communication with the server is needed, as depicted in Figure 2. Note that on the point 1), one could argue that the opposite will happen when the resize operation retrieves a 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



(a) *q1*(red), *q2*(blue),and *q3*(orange).

(b) *q4*(black), and *q5*(pink).

Figure 6: Performance Analysis using 5 queries from our use-case described in the Experimental Setup Section. We show queries in different figures for readability reasons. Top figures show the throughput of the queries for just retrieving metadata associated with the images. Bottom figures show the throughput when retrieving both metadata and images, plus operations applied to images when applicable (queries 2, 3, 4 and 5). The experiments show the performance of both systems (VDMS and baseline) as the database size increases. These queries were run using 56 simultaneous clients (nthr = 56), and averaged out of 10 runs (niter = 10), each client running 2 transactions (ntags = 2).

image is much more common and is the common practice when it comes to image processing through CNNs [8, 14].

3.3.2 Query Execution Analysis

The next step in our analysis involved running a different set of queries (described above in this section), to better understand the performance of the systems under different query conditions. Figure 6 shows the evaluation of the queries we analysed for our use case. Top figures show the throughput of these queries retrieving metadata associated with the images. Bottom figures show the throughput when retrieving both metadata and images, plus operations applied to images when applicable (queries 2, 3, 4,and 5). The experiments show the performance of both systems (VDMS and baseline) as the database size increases in terms of number of images. These queries were run using 56 simultaneous clients (nthr = 56), and averaged over 10 runs (niter = 10), each client running the retrieval of the tag 2 times (ntags = 2). The last parameter (ntags) is to avoid having some queries to finish the work too fast before other clients can even send the query to the server. This ensures that there is enough work to do in a query so that all queries execute in parallel on the server side. To analyze these plots, one needs to compare the full-line (VDMS) versus the dotted-line (baseline), each color representing a different query. For example, to compare *q1* performance, one needs to look at the full-red line (VDMS) and the dotted-red line (baseline).

3.3.3 Metadata Retrieval

First, we analyze the top figures, showing metadata transactions per second. We evaluate the performance when retrieving only metadata associated with the images, and not the images themselves. In this particular case, the baseline translates directly into MySQL performance. Note that for these

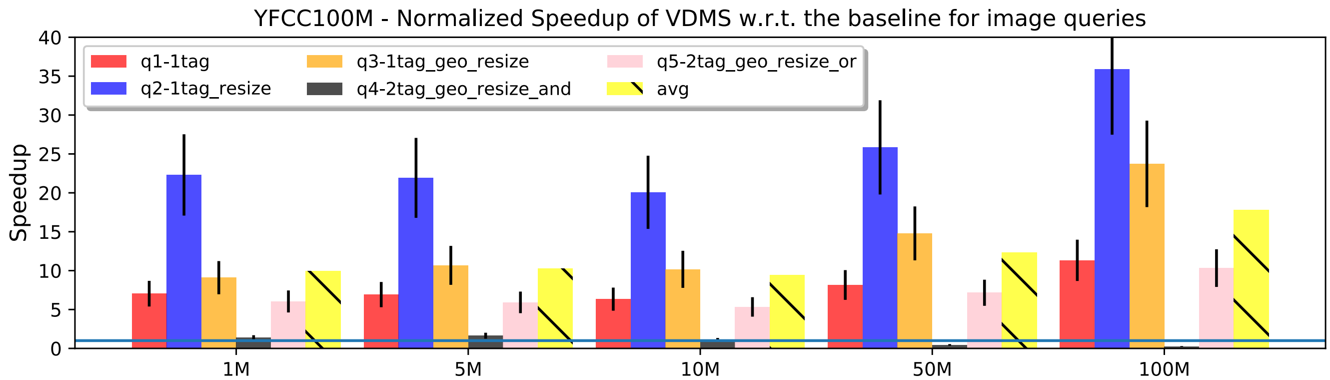


Figure 7: Summary of performance gains for all queries. We see up to 35x speedup (*q2*), and an average of about 15x. More importantly, we see that the speedup grow as the database size increases, showing that VDMS scales better than the baseline.

figures (top), *q1* and *q2* are essentially the same query. This is because the metadata retrieved for both queries does not change. The only difference between *q1* and *q2* is the presence of the resize operations that does not have any impact on analyzing performance of metadata retrieval. For *q1* and *q2*, we can appreciate higher performance being delivered by VDMS when compared to the baseline, and how this improvement is maintained as the size of the database increase. For *q3*, we see that MySQL performs best when the database size is small (1M images), with VDMS outperforming MySQL as the database increases in size. For *q4*, we see MySQL also outperforming VDMS on small size databases, and as the database size increase, the gap between the two narrows, similar to what we see for *q5*. It is interesting to note that adding filtering by geo-location (*q3*, and *q4*) slightly increases the performance of MySQL for small databases, and decreases it as the database sizes scales. The behaviour is different in the case of *q5*, 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 of the AND operation as *q4*. In the case of VDMS, we see *q3* performance is comparable to *q1* and *q2*, but *q4* suffers significantly when the scale of the database increases. 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 *q5*, we see a similar effect, although it is worth noting that for the OR operation there is no need for 2 retrievals. Rather, a single retrieval is performed and the result filtered on the client. Future release will add more of such operators (AND, OR, etc) in order to prevent unnecessary retrievals and extra filtering on the client side.

3.3.4 Image Retrieval

We continue by analyzing the bottom figures, showing measured throughput, as images per second, delivered by each system. For the case of VDMS, *q1*(full-red-line) shows less throughput than *q2* (full-blue-line) (bottom left figure). This is expected as *q1* returns a full size version of the image, whereas *q2* returns a resized (smaller) version of the image, thus transferring less data over the network. In the case of the baseline, both *q1* and *q2* transfer the full size version of the image, and as part of *q2*, the resize is performed in the client. This is why, contrary to the VDMS case, *q1* performs better (even if slightly) when compared with *q2*. We can also see that, for VDMS, *q3* perform worst than *q2* because of the extra step needed for filtering based on geo-location. Moreover, we see a great performance degradation in the case of *q4* as the database size increases. This is entirely attributed to the 2-round process needed for this query, as we explained before. From the first 3 queries, as well as *q5*, we clearly see that VDMS outperforms the baseline 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.

Finally, Figure 7 summarizes the results. We see up to 35x speedup (for the case of *q2*), and an average improvement in throughput of about 15x. More importantly, we see that the speedup increases as the database size grows, showing that VDMS scales better than the baseline. We also see how *q4* shows poor performances and scalability when compared to the baseline, and 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 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 date other than visual. VDMS, by design, eliminates all the inefficiencies that result from a forced integration of components designed for a different range of applications.

4 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 and feature vectors. We show how VDMS compares against a combination of industry standard systems, all of which are needed to replicate a only portion of VDMS' functionality. We see improvements up to 35x in certain queries, and an average improvement of about 15x. 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.

5 Acknowledgments

Not shown during submission.

References

[1] Giuseppe Amato, Fabrizio Falchi, Claudio Gennaro, and Fausto Rabitti. Yfcc100m-hnfc6: A large-scale deep features benchmark for similarity search. In *Similarity Search and Applications*, pages 196–209. Springer International Publishing, 10 2016.

[2] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Proc. of the 1998 ACM SIGMOD*, SIGMOD '98, pages 575–577. ACM, 1998.

[3] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on OSDI*, volume 10, pages 1–8, 2010.

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

[5] Paul G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. In *Proc. of the 2010 ACM SIGMOD*, pages 963–968. ACM, 2010.

[6] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

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

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

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

[10] Geir Hoydalsvik. Mysql connection handling and scaling, March 2019. https://mysqlserverteam.com/mysql-connection-handling-and-scaling/.

[11] IntelPR. Intel and micron produce breakthrough memory technology, 2015.

[12] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research and Technology*, 1, 2012.

[13] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *CoRR*, abs/1702.08734, 2017.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

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

[16] Ziqi Li. Nosql databases, Jul 2019.

[17] Libffmpeg. Ffmpeg library. http://source.ffmpeg.org.

[18] MemSQL Inc. The real-time data warehouse you can run anywhere. https://www.memsql.com/.

[19] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEE Trans. on Pattern Analysis and Machine Intelligence*, 36, 2014.

[20] Oracle Co. The world's most popular open source database. https://www.mysql.com/.

[21] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proc. VLDB Endowment*, 10(4):349–360, November 2016.

[22] Not shown during submission. Not shown during submission. *Systems for Machine Learning Workshop (SysML) at NIPS, Montreal, Canada*, 2018.

[23] Josef Sivic and Andrew Zisserman. Video google: Efficient visual search of videos. In *Toward category-level object recognition*, pages 127–144. Springer, 2006.

[24] The Apache Software Foundation. Apache spark: Lightning-fast cluster computing. https://spark.apache.org/.

[25] The Apache Software Foundation. What is apache hadoop? https://hadoop.apache.org/.

[26] Bart Thomee, Benjamin Elizalde, David A. Shamma, Karl Ni, Gerald Friedland, Douglas Poland, Damian Borth, and Li-Jia Li. Yfcc100m. *Communications of the ACM*, 59(2):64–73, Jan 2016.

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

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