EMBEDDINGS LAKE: A COST-EFFICIENT AND CLOUD-BASED VECTOR DATABASE

EMBEDDINGS LAKE: A COST-EFFICIENT AND CLOUD-BASED VECTOR DATABASE

A project report submitted in partial
fulfillment of the requirements for the degree of
Master of Science Computer Science

By

William Archbold
US Military Academy, West Point 2010
Bachelor of Science. In Systems Engineering

May 2025
University of Colorado Denver

ABSTRACT

For decades, databases have empowered scientists and engineers to efficiently store and query large volumes of data. However, the most rapidly growing data category in recent years has been unstructured data. Vector Database Management Systems (VDBMSs) are a relatively recent innovative solution to manage unstructured data represented as high-dimensional vectors. Despite significant growth in both open-source and commercial VDBMS solutions over the past decade, academic research focused on benchmarking and analyzing their performance is limited.

This master project first evaluates the strengths and limitations of several leading VDBMS platforms. Based on the knowledge and insights gained from this analysis, we have developed Embeddings Lake, a novel VDBMS designed for cloud-native environments.

Embeddings Lake is a cloud-native, cost-effective vector storage system designed to overcome local memory limitations without introducing the complexity of physically distributed architectures. The system enables users to define the number of shards used for data storage, thereby optimizing memory consumption and enhancing scalability. Embeddings Lake also supports parallel shard querying to maintain both high accuracy and query performance. It is built to run 100% on the cloud and be entirely serverless so that the end-user does not have to worry about maintaining patches or updates. Furthermore, it allows users to select from three different distance metrics to better tune their query results.

This work includes a comprehensive performance analysis of Embeddings Lake. Experiments examine the impact of vector dimensionality and radii on accuracy, speed,

and the distribution of sharding. Embeddings Lake's accuracy and cost estimates were compared to a handful of other industry-leading VDBMSs. Finally, this work outlines potential directions for future enhancements, including improving the shard methods, finer parameter tuning, and integrating the latest research in graph algorithms.

This Project Report is approved for recommendation to the Graduate Committee.

Project Advisor:

____

Ilkyeun Ra, PhD

MS Project Committee:

____

Farnoush Banaei-Kashani, PhD

____

Zhengxiong Li, PhD

TABLE OF CONTENTS

1. Introduction 1

1.1 Problem 1

1.2 Project Objective 1

1.3 Approach 1

1.4 Organization of this Project Report 2

2. Related work 3

2.1 Background 3

2.2 Vector Databases 3

2.3 Related Research 5

2.4 Preliminary Study 6

2.4.1 Selected Databases 6

2.4.2 Hardware and Software 7

2.4.3 Methodology 8

2.4.4 Accuracy 9

2.4.5 Comparison Analysis 11

3. Motivation 14

3.1 Local Machine Limitations 14

3.2 Commercial Cloud VDBMS 14

3.3 Hybrid Cloud Approach 16

3.3.1 Vector Lake Shortcomings 17

4. PROPOSED SOLUTION: Embeddings lake 18

4.1 Query Accuracy 18
4.1.1 Distance Metrics 18
4.1.2 Result Sorting 20
4.1.3 HNSW Insertion Point 21
4.2 Migration to the Cloud 22
4.3 Serverless Cloud Solution 25
4.4 Improve Query Efficiency 26
5. Performance Evaluation and Discussion 28
5.1 Test Environment 28
5.2 Distance Metric Comparison 30
5.2.1 Animals Dataset 30
5.2.2 Butterflies Dataset 33
5.3 Dimensionality 33
5.4 Efficiency 35
5.5 Embeddings Lake vs Other VDBMS 36
5.6 Cost 37
5.7 Effect of Hashing 39
6. Conclusions 43
6.1 Summary 43
6.2 Future Work 43
6.2.1 Improve Accuracy 43
6.2.2 Improve Efficiency 44
6.2.3 Improve Utility 46

References **47**

Appendix A: Data Lake Sort Implementation **52**

Appendix B: Butterfly dataset data **53**

B-1. Distance Metric Comparison with 512 Dimensions 53

B-2. 512 Dimensions Cosine Query 54

B-3. 512 Dimensions Manhattan (L1) Metric query 54

B-4. 512 Dimensions Euclidean (L2) Metric query 54

B-5. Accuracy % Increases from 512 to 768 Dimensions 54

Appendix C: Opensearch **58**

C-1. Deployment Configuration 58

C-2. Index and Query Configuration 58

Appendix D: Cost Calculations **59**

D-1. Embeddings Lake 59

D-2. SQS Source Pricing 59

D-3. API Gateway Source Pricing 60

D-4. S3 Source Pricing 60

D-5. Weaviate Pricing with 512 dimensions, 1M vectors, Performance storage 61

D-6. Weaviate Pricing with 512 dimensions, 5K vectors, Performance storage 61

D-7. Weaviate Pricing with 512 dimensions, 1M vectors, Compression storage 62

D-8. Weaviate Pricing with 512 dimensions, 5K vectors, Compression storage 62

D-9. Pinecone Pricing 63

D-10. AWS Opensearch Serverless Pricing for 1M embeddings 63

D-11. AWS Opensearch Serverless Pricing for Animals Dataset 64

D-12. AWS Opensearch Pricing for 1M embeddings 65

D-13. AWS Opensearch Pricing for Animals Dataset... 65

LIST OF FIGURES

Figure 1: Accuracy results for Fruits & Vegetables Dataset 10

Figure 2: Accuracy results for Animal Dataset.. 10

Figure 3: Incorrectly predicted "caterpillar" image for Weaviate 11

Figure 4: Incorrectly predicted "caterpillar" image for Chroma 11

Figure 5: Comparison of Chroma, Milvus, Qdrant, Weaviate requests per second (RPS)12

Figure 6: Comparison of Chroma, Milvus, Qdrant, Weaviate total time (seconds) 12

Figure 7: Weaviate Cost Pricing as of March 1st 2025 15

Figure 8: Vector Lake Cosine Distance Implementation... 19

Figure 9: Embeddings Lake AWS Cloud Architecture 23

Figure 10: Embeddings Lake Config File Location and Contents 24

Figure 11: Radius search scenarios... 26

Figure 12: Butterflies Dataset sample images 29

Figure 13: Distance Metric Comparison for Animals Dataset with 512 Dimensions 31

Figure 14: Highest Accuracy Scores Animals Dataset w/ 512 Dimensions..................... 33

Figure 15: Distance Metric Comparison for Butterflies Dataset with 768 Dimensions... 34

Figure 16: Sampling of time measurements for Animals Dataset, 512 Dimensions,
Manhattan query 36

Figure 17: Accuracy Results for Open-Source VDBMS 36

Figure 18: LSH distribution of gorilla images with 512 dimensions and 725 shards....... 39

Figure 19: LSH distribution of eagle images with 512 dimensions and 725 shards......... 40

Figure 20: LSH distribution of gorilla images with 512 dimensions and 23171 shards... 40

Figure 21: LSH distribution of eagle images with 512 dimensions and 23171 shards..... 41

Figure 22: LSH distribution of all animal categories in the Animal Dataset with 512 dimensions and 725 shards 42

Figure 23: LSH distribution of all animal categories in the Animal Dataset with 512 dimensions and 23171 shards 42

LIST OF TABLES

Table 1: DB-Engines' 10 most popular vector databases 7

Table 2: Selected databases' capabilities 7

Table 3: VDMBS Experiment Conditions 8

Table 4: Incorrectly quired images 11

Table 5: Varying Accuracies with Unseeded Insertion Point 22

Table 6: Embeddings Lake Experimental Set-Up 28

Table 7: Experiment Shard Values for Reference 29

Table 8: Test Query Image Example Embedding Values 30

Table 9: Best and Worst Accuracies by Shard for Animals Dataset, 512 dimensions and Cosine query 32

Table 10: Best and Worst Accuracies by Shard for Animals Dataset with 512 dimensions and Manhattan (L1) query 32

Table 11: Best and Worst Accuracies by Shard for Animals Dataset with 512 dimensions and Euclidean (L2) query 32

Table 12: Summary of Accuracy Percentage Change for Dimensionality Increase 35

Table 13: AWS Opensearch Comparison by dataset 37

Table 14: Highest accuracy achieved for each VDBMS and dataset with 512 dimensions 37

Table 15: Cost Estimate for Embeddings Lake vs Commercial VDBMS 38

1. INTRODUCTION

1.1 Problem

The need for data storage and faster retrieval is growing exponentially. Vector databases enable effective storage of unstructured data and querying for information based on nothing more than a sample picture. However, vector databases are either constrained by local hardware limitations or are cost-prohibitive when operating at scale in a distributed environment.

1.2 Project Objective

In this paper, we present, Embeddings Lake, a vector database alternative that maintains high accuracy while offering low-cost, cloud-based distribution and scale.

1.3 Approach

Embeddings Lake achieves this goal by building on the intent of the Vector Lake project. Vector Lake was designed to disperse the stored vectors across a user-controlled quantity of shards on a local machine, but in operation the system achieves a query accuracy of zero. Embeddings Lake forks from Vector Lake and fixes the critical flaws of the original system to fulfill the project's objective while adding new features to maintain industry competitive performance. The new features include parallel querying of tangential shards, additional query distance metrics, and 100% migration into the cloud to alleviate any concerns over local device limitations.

1.4 Organization of this Project Report

Chapter 2 first covers the background and history of vector databases and addresses the lack of comprehensive studies comparing the performance of various commercial vector databases. Lastly, Chapter 2 summarizes a comparison test we conducted to understand the strengths and weaknesses of the industry’s current state. Chapter 3 describes how the testing results in Chapter 2 motivated the search for a distributed vector database management system (VDBMS), what the current state of the cloud VDBMS environment offers, and how those offerings inspired Embeddings Lake. Chapter 4 describes what Embeddings Lake intended to accomplish and how Embeddings Lake’s goals were implemented. Chapter 5 provides testing results of how well Embeddings Lake met those goals. Lastly, Chapter 6 describes how future research and augmentations can improve Embeddings Lake's performance.

2. RELATED WORK

2.1 Background

Databases have enabled scientists and engineers to store and query vast amounts of data for decades, but the type of data that has increased the most rapidly is unstructured data [1]. When a typical smartphone user takes a picture on the phone, the camera application will create a set of structured metadata about the photo, but the bulk of the data the application creates is unstructured data composing the photograph. Though the information that pictures and videos present is perceivable to humans, it's not as easily categorizable for computers, so it's considered unstructured.

Content-based image retrieval (CBIR) has been used since the 1990s to help index images and videos based on visual cues such as color and texture [2]. More recently, large vision models (LVMs) such as Vision Transformer have shown excellent performance for computational image recognition [3], but efficient databases are needed to recognize unstructured content at scale. Conventional databases, such as SQL-based databases, are not optimal storage mechanisms for querying unstructured data due to their need for exact data matches [4] .

2.2 Vector Databases

Vector database management systems (VDBMSs) are suitable repositories for unstructured data. "Vectors are n-dimensional objects consisting of natural, real, or complex numbers where each number represents a feature or a part of a feature [5]." A vector's complexity can range from a single dimension representing the population of a country to multiple dimensions with each component's value representing the color shade

of a pixel in a picture. A VDBMS is a storage system that indexes vectors in an organized manner and allows searching for vectors that are similar to a given query vector.

VDBMSs can use similar storage techniques that conventional databases use to include sharding, caching, and partitioning, but there are several algorithms that can be used to index and query the database. The techniques are typically broken up by k-nearest neighbor (KNN) and approximate nearest neighbor (ANN). KNN includes brute force and various tree-based methods such as KD, Ball, R, and M trees. ANN is broken down into hashing, tree-based, and graph-based approaches. However, “traditional indexing methods, such as B-trees or hash tables, are not suitable for high-dimensional vectors because they suffer from dimensionality catastrophe” [4].

KNN search is expensive with time complexity of $O(dN + N \log k)$, but ANN provides faster search speed while maintaining sufficient accuracy. Hierarchical Navigable Small World (HNSW) is a graph-based, ANN algorithm that provides VDBMS with “state of the art performance” [6] .

Vectors can be stored in either stand-alone vector libraries or vector databases. Examples of libraries include HNSWLIB, Facebook’s FAISS, Spotify Annoy, Google’s ScANN, and NMSlib. Vector libraries are typically used for interacting with static snapshots of data, while vector databases are more suitable for create, read, update, and delete (CRUD) operations [7].

VDBMSs provide some unique advantages over traditional indexing databases. Traditional database can only collect exact matches for a query while VDBMSs offer semantic nuance. For example, if an end-user wanted to search a database for all animals with brown fur, the traditional database’s entries would all have to have an explicit value

defining either a Boolean declaring the animal had brown fur. Additionaly, the definition of what constitutes the color brown can become subjective. The end-user may have a broader or more narrow definition of the color brown than the builder of the database. Vector databases can help account for that nuance without ever needing a defined Boolean value.

Vector databases are offered as either storage systems exclusively for vectors or multi-modal. Multi-modal databases are VDBMS systems that aren't designed to store only vectors. Oftentimes, they were initially designed for the storage of other types of information but later modified to handle vectors. The most popular multi-modal VDBMS as of this writing is Elasticsearch [8] .

2.3 Related Research

VDBMSs that are exclusively for vectors are a relatively new type of database. The majority of VDMS research articles provide basic overviews of how VDBMS's operate in the abstract. Researchers at Carnegie Mellon, the University of Michigan, and the Harbin Institute of Technology listed the number of dimensions that twelve VDBMSs offered as of 2021 [6]. Toni Taipalus from the University of Jyväskylä provided a similar overview of how VDBMSs generally operate and provided a table of six VDBMSs, their licenses, and query support [5]. Though benchmarking surveys exist for indexing algorithms such as HNSW [9], few to no scientific studies have been conducted to compare VDBMSs performance while operating.

2.4 Preliminary Study

Though there are a handful of journal articles providing overviews of various VDBMSs, few to none of them offer performance comparisons. We conducted a precursor analysis to compare the behavior of the prevalent commercial VDBMSs to better understand their shortcomings and opportunities for improvement.

Effectively testing VDBMSs required synchronization of four components. First, available databases to compare amongst one another. Second, immutable sample data. Third, hardware/infrastructure. Lastly, quantifiable and consistent performance metrics.

2.4.1 Selected Databases

The databases selected needed to be open-source and available in container-format. Open source is preferred to proprietary databases because having the ability to inspect code could explain extraordinary behavior while conducting experiments. Trying to understand behavior with proprietary databases could require reverse engineering and can only lead to speculative explanations. Containerized databases are preferred over locally installed or locally running databases because containers can help isolate software and minimize the effects of outstanding variables, such as competing processes on a machine.

The databases must also have subject relevancy in the community. Comparing a commercially popular, regularly updated database to an obscure, outdated, and seldom-used database is unlikely to be revelatory. A website named DB-Engines tracks the popularity of various types of databases to include vector databases [8]. The most popular vector databases from DB-Engines are in *Table 1*.

DB-Engines Most Popular VDBMSs									
1.	Elasticsearch (Multi-modal)	3.	Aerospike (Multi-modal)	5.	Milvus (Vector)	7.	Chroma (Vector)	9.	Weaviate (Vector)
2.	Kdb(Multi-modal)	4.	Pinecone (Vector)	6.	DolphinsDB	8.	Qdrant (Vector)	10.	ObjectBox (Multi-modal)

Table 1: DB-Engines' 10 most popular vector databases

Table 1 has a mix of Multi-modal and Vector databases. Multi-modal databases are databases that can store more than vectors, while vector databases are built exclusively for storing vectors. Though comparing Multi-modal to Vector-exclusive databases would be interesting, this research focused on vector-only databases.

The vector databases that met the criteria in *Table 1* are Milvus, Chroma, Weaviate, and Qdrant. *Table 2* summarizes each of their capabilities. Pinecone was not selected because it's a proprietary VDBMS.

VDBMS	License	First Release	Cloud Mgmt	Index Algorithm	Multi-node Scaling
Milvus	Apache 2.0	2019	Yes	HNSW, IVF, PQ, etc..	Yes
Chroma	Apache 2.0	2023	No	HNSW	No
Weaviate	BSD 3-Clause	2019	Yes	HNSW, Flat, Dynamic	Yes
Qdrant	Apache 2.0	2022	Yes	HNSW	No

Table 2: Selected databases' capabilities

2.4.2 Hardware and Software

Table 3 lists the hardware and software versions used for the experiment.

Component	Version
Computer	November 2023 Macbook Pro
CPU	Apple M3 Pro
Memory	36 GB
Operating System	Sonoma 14.5
Python	3.11.10
Docker	27.2.0
Docker Image - Weaviate	1.27.5
Docker Image - Qdrant	1.12.4
Docker Image - Milvus	2.4.17
Docker Image - Chroma	0.5.20

Table 3: VDMBS Experiment Conditions

2.4.3 Methodology

Two sets of data were used for the experiment. The first dataset is the Fruits and Vegetables Dataset from Kaggle, which contains 100 images of 36 different fruits and vegetables for a total of 3,600 training images and 10 images for each fruit in a testing directory [10]. The total Fruits Dataset size is approximately 2 gigabytes. The second dataset is the Animal Dataset from Kaggle, which contains 5,400 images of 90 different animals [11]. The Animal Dataset size is 579 megabytes.

In order to store images into the four vector databases selected in Section 2.4.1, the images first need to be embedded. Vision Transformer “Base” transformer with a patch size of 32 pixels (ViT-B/32) was selected to vectorize the sample data. ViT-B/32 is part of OpenAI’s Contrastive Language-Image Pre-Training (CLIP) network of image and text encoders [12]. Every image in the datasets has to be opened, preprocessed, and encoded into a vector using the Vit-B/32 model before the picture can be loaded into a vector database. The original size of the Animals Dataset was 579 megabytes. Once embedded into vectors with 512 dimensions, the size of the Animals Dataset is 10.32 megabytes as calculated in Appendix D-1.

The first test was to see how well each of the four vector databases could accurately query all the images that were in the same category as a random sample image. No memory limitations were introduced in this test. For the Fruits and Vegetables dataset, a testing set already came with the dataset, but no testing set came with the Animals dataset, so one random picture of every animal was moved to a testing set.

The second test was to see how well each database performed when limiting the amount of memory in the container. The specific steps of this experiment involved embedding the Animals Dataset images and uploading the embeddings to a collection for each vector database running in its own exclusive container. Once the embeddings were uploaded, a Python script executed a thousand queries looking for similar images. The same thousand images were queried across all four databases in the same order. The Python script collected the total time for all queries, the average time per query, and the 90th and 95th percentile for the average queries. It also collected the average number of queries per second the database can handle.

The test executed for each database six times at memory limits of 48, 64, 80, 96, 128, 256, and 512 megabytes.

2.4.4 Accuracy

All four databases were able to accurately recall all images from the Fruits and Vegetable dataset, as shown in *Figure 1*.

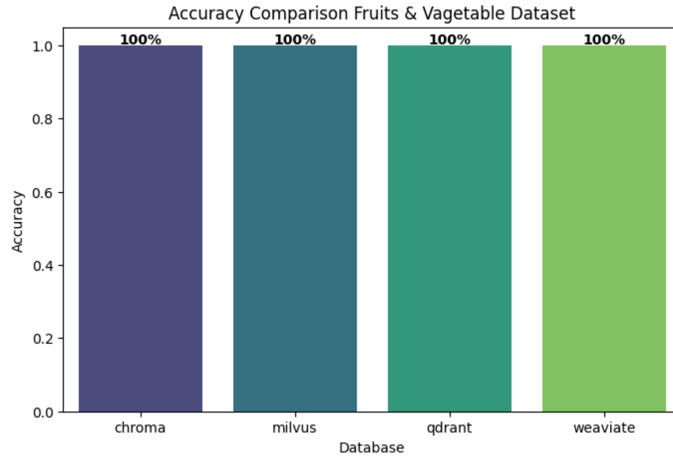


Figure 1: Accuracy results for Fruits & Vegetables Dataset

Accuracy predictions for the Animals dataset were slightly lower and had some variance among the four vector databases, as shown in *Figure 2*.

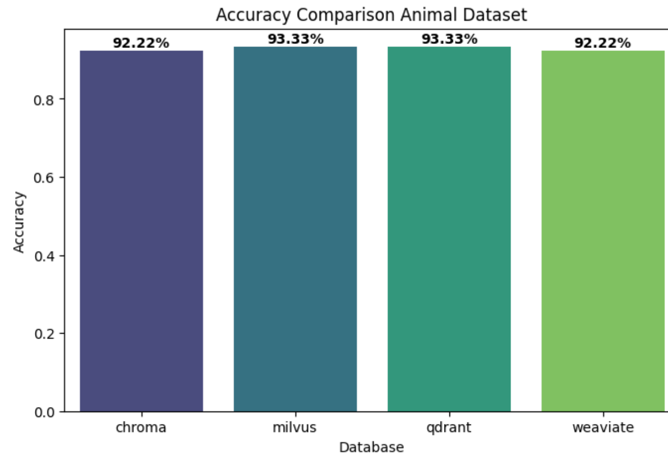


Figure 2: Accuracy results for Animal Dataset

The animals that each vector database incorrectly predicted are listed in *Table 4*. There were five queried test animals, and all four databases had the same incorrect recalled image. All four databases got the query for an eagle wrong, but Chroma was the only one to retrieve 'duck' while the other three retrieved 'pigeon'. Chroma and Weaviate had lower accuracies because they both queried for 'caterpillar' but got 'deer' and 'hare', respectively. The incorrect 'caterpillar' images for Chroma and Weaviate are

shown in *Figure 3 and Figure 4*. Even though the incorrectly tested ‘caterpillar’ is a picture of a bulldozer, Milus and Qdrant still got it right.

	Chroma	Milvus	Qdrant	Weaviate
		(test image, retrieved image)		
1	('mosquito', 'fly')	('mosquito', 'fly')	('mosquito', 'fly')	('mosquito', 'fly')
2	('duck', 'goose')	('duck', 'goose')	('duck', 'goose')	('duck', 'goose')
3	('eagle', 'duck')	('eagle', 'pigeon')	('eagle', ' pigeon ')	('eagle', ' pigeon ')
4	('hornbill', 'ox')	('hornbill', 'ox')	('hornbill', 'ox')	('hornbill', 'ox')
5	('pelecaniformes', 'swan')	('pelecaniformes', 'swan')	('pelecaniformes', 'swan')	('pelecaniformes', 'swan')
6	('koala', 'bear')	('koala', 'bear')	('koala', 'bear')	('koala', 'bear')
7	('caterpillar', 'deer')			('caterpillar', 'hare')

Table 4: Incorrectly quired images

Figure 3: Incorrectly predicted “caterpillar” image for Weaviate

Figure 4: Incorrectly predicted “caterpillar” image for Chroma

2.4.5 Comparison Analysis

Figure 5 shows each of the databases’ average requests per second (RPS) for the varying memory levels.

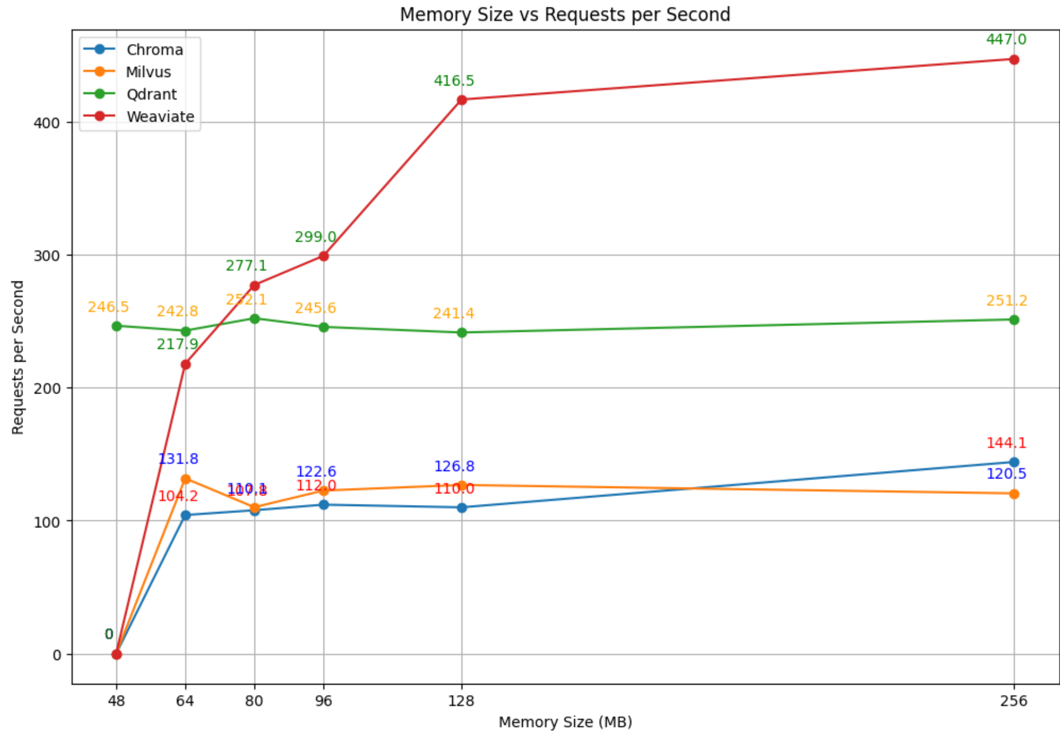


Figure 5: Comparison of Chroma, Milvus, Qdrant, Weaviate requests per second (RPS)

Figure 6 shows each of the databases' total time for the varying memory levels.

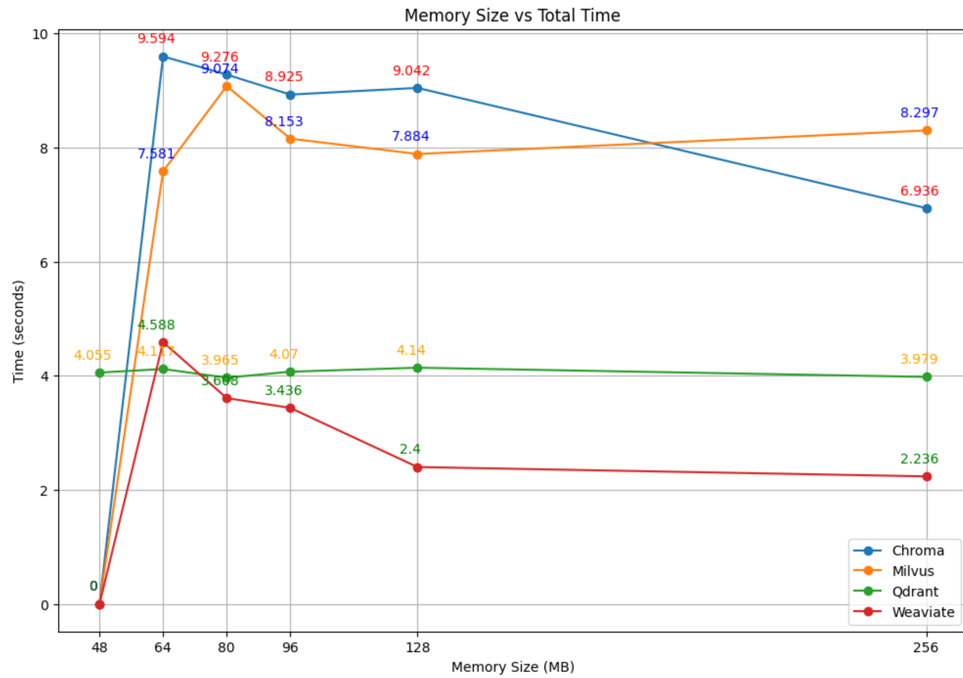


Figure 6: Comparison of Chroma, Milvus, Qdrant, Weaviate total time (seconds)

The graphs in the two figures show similar results. Databases saw either little to no improvement or significant improvement. Milvus and Qdrant saw little to no improvement with more memory. The databases that saw the most improvement with more memory were Weaviate, followed by Chroma. Weaviate's website says that though "memory determines the maximum supported dataset size, [memory] does not directly influence query speed". However, Weaviate also advertises that it can take advantage of memory mapped files, which it says is "efficient, but much slower than in-memory storage" [13]. This experiment did not explicitly set any memory mapping parameters for Weaviate.

All of the databases failed to load with 48 megabytes of memory except for Qdrant. Qdrant didn't fail until its container was limited to 32 megabytes. This may be due to Qdrant's support of memory mapped storage, which "creates a virtual address space associated with the file on disk. Mmapped files are not directly loaded into RAM. Instead, they use a page cache to access the contents of the file. This scheme allows flexible use of available memory. With sufficient RAM, it is almost as fast as in-memory storage" [14]. Though there is a way to use memory mapping, this feature was not explicitly set in this experiment. Milvus v2.4 also offers memory mapping [15], but it wasn't explicitly enabled.

ChromaDB is the only database that doesn't offer memory mapping but does offer a least recently used (LRU) caching feature, but LRU is only provided at the collection, not the vector level, which means it wouldn't have helped in this experiment.

3. MOTIVATION

3.1 Local Machine Limitations

At some point in the performance testing of the research outlined in Section 2.4, every database failed when memory was limited enough. We can assume that similar failures can occur when the available memory is significantly larger, but still only a fraction in size of the dataset. That assumption is applicable to real world scenarios, which can have significantly larger dataset collections potentially in the gigabyte, terabyte and even petabytes of size. Not only do commercial/industrial collections have significantly higher data requirements, but they also have high availability needs, and cannot afford system crashes, as was seen in the Section 2.4 experiment. Migrating the VDBMS to a distributed system could help mitigate this shortcoming. Weaviate [16], Milvus [17], and Qdrant [18] offers multiple node support. ChromaDB does not offer clusters/sharding. It provides only a single-node client/server model [19]. However, distributed systems require coordination and oftentimes multiple servers and internetworking fixed and variable costs.

3.2 Commercial Cloud VDBMS

To maintain a scaled and distributed system without requiring the overhead of physical systems, VDBMS providers have begun to offer cloud-based VDBMSs. Many of the VDBMSs are now offering cloud-based solutions to offload users' dependency on local hardware, including Weaviate [20] and Qdrant [21]. However, the organizations' cloud offerings can still be expensive. As of the writing of this article, Weaviate's

Serverless Cloud starts at $25 per month for approximately two million stored objects and goes up to $608/month for 50 million objects [22], as per *Figure 7*.

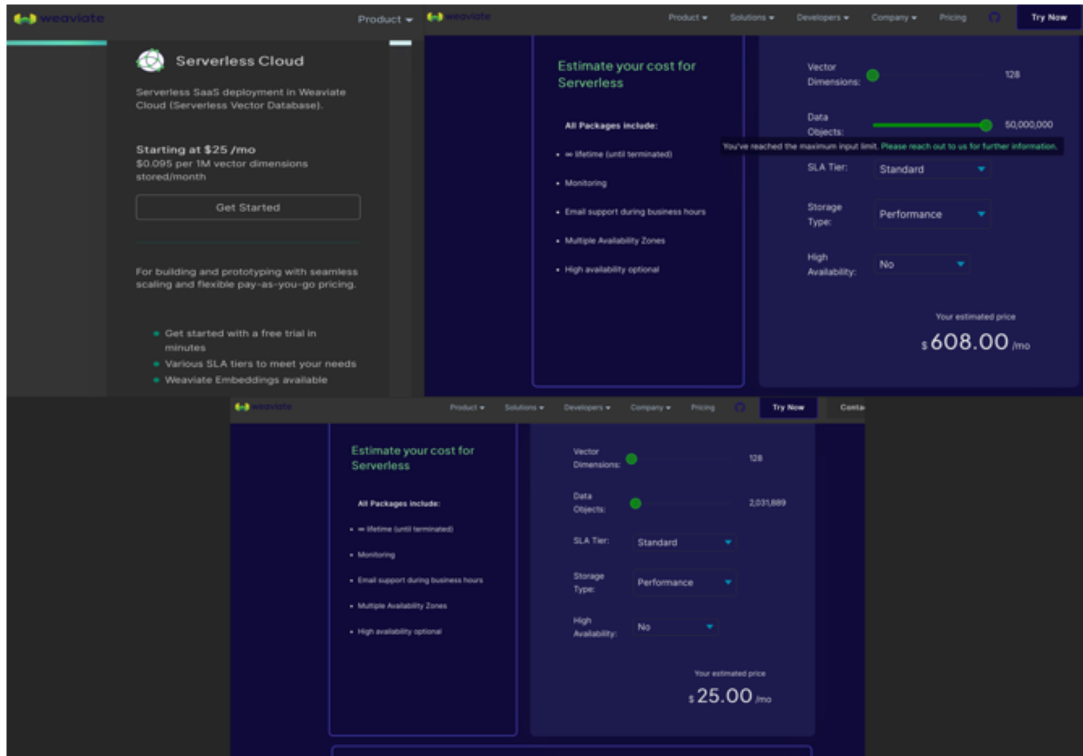


Figure 7: Weaviate Cost Pricing as of March 1st 2025

Manu is an open source, cloud-focused VDBMS based on Milvus that breaks up storage and querying into multiple layers to increase and customize performance based on the user's needs [23]. However, Milvus's Manu seems to be the only open-sourced cloud offering. Manu's code hasn't been updated since 2022. Amazon Opensearch, which is a fork of Elasticsearch, offers vector storage and search but is limited to supporting "the HNSW algorithm with Faiss" [24] and is not a native VDBMS [25, p. 1611] . ChromaDB does not have a cloud model but will release one in 2025 [26]. As mentioned in Section 2.4.1, Pinecone is a popular cloud-based VDBMS, but is a proprietary solution.

3.3 Hybrid Cloud Approach

Vector Lake is a VDBMS project that attempts to offer the best of the cloud advantages while minimizing cost and maintaining an open-source standard. It is advertised as a "robust, vector database designed for low maintenance, cost, efficient storage and ANN querying of any size vector data distributed across S3 files"" while utilizing the HNSW as its underlying storage algorithm [27]. Vector Lake can be installed locally as a Python package and called upon to instantiate, add to, and query a VDBMS. Vector Lake differentiates itself from the typical commercial VDBMS by offering custom sharding solutions where the shards can be stored in AWS Simple Storage Service (S3) for minimal cost that the user can pay without worrying about a middle tier VDBMS company's fee.

Vector Lake utilizes a three-step process to store and query vector embeddings. First, a lake is instantiated based on the desired number of shards and number of dimensions in the embeddings. Second, when an embedding is either added or queried, it is first hashed using Locality-Sensitive Hashing (LSH). LSH is a technique used to quickly find similar vectors by hashing them into "buckets" or shards where similar vectors are likely to fall into the same bucket. Instead of performing expensive pairwise comparisons, LSH reduces the search space by grouping similar vectors together using a set of random hyperplanes that are constructed at the time of the lake's instantiation. With the proper bucket identified, the third step executes, which involves an approximate nearest neighbor (ANN) search using the HNSW algorithm described in Section 2.2. The core idea of HNSW is to build a graph where each data point is connected to other similar points. This graph is hierarchical with different levels. The top levels provide a broad

overview and help the process to jump to relevant sections, while the lower levels contain more detailed connections for precise searches. This structure allows the algorithm to move through the graph efficiently, reducing the time it takes to find the nearest neighbors of a given insertion or query.

3.3.1 Vector Lake Shortcomings

The Vector Lake library has not been updated since August 2023 and has not only some minor defects and bugs, but also critical failures. First, Vector Lake's S3 interface only works with a locally running simulated S3 bucket and not an actual S3 bucket in the Amazon Web Services cloud. It fails to run due to some erroneous type declarations and JSON conversions. More importantly, even with those issues fixed, Vector Lake could not identify any of the images from the Animal Dataset with an accuracy score of zero.

4. PROPOSED SOLUTION: EMBEDDINGS LAKE

This project proposes Embeddings Lake – a cost-efficient, open-source VDBMS optimized for the cloud based on the Vector Lake VDBMS, but with a few critical improvements. First, it will have a significantly higher accuracy score comparable to the commercial VDBMS offerings. Second, it will migrate both processing and storage to the cloud to circumvent any limitations of local hardware. Third, it will be a fully serverless cloud application that doesn't require the user to maintain virtualized machinery. Fourth, it will improve query efficiency for rapid information retrieval.

The source code for Embeddings Lake is available at https://github.com/btreb-ht2/embeddings-lake/tree/main .

4.1 Query Accuracy

4.1.1 Distance Metrics

Embeddings Lake's core algorithm for organizing embeddings is the HNSW algorithm. HNSW does a series of vector comparisons to find similar vectors. One method to make a similarity comparison between two vectors is to use Cosine Distance; however, Vector Lake erroneously utilizes Cosine Similarity instead. Cosine similarity measures how similar two vectors are, ranging from -1 (opposite) to 1 (identical) as per *Formula (1)*.

$$Cosine\ Similarity(A, B) = \cos(\Theta) = \frac{A \cdot B}{||A||||B||} \quad (1)$$

Cosine distance is defined as *1 - cosine similarity*, meaning closer vectors have a smaller distance (closer to 0) and vice versa as per *Formula (2)*.

$$Cosine\ Distance(A, B) = 1 - \frac{A \cdot B}{||A||||B||} \quad (2)$$

Figure 8 shows Vector Lake's code implementation of the cosine distance formula. The author possibly made a logical error. The function name cosine_distance suggests that it calculates the distance between vectors. However, the implementation calculates cosine similarity, which measures how close two vectors are, not how far apart they are. Embeddings Lake corrected this error and implemented the proper cosine distance formula.

```
01    import numpy as np
…
02    def cosine_distance(a, b):
03      return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

Figure 8: Vector Lake Cosine Distance Implementation

Though Vector Lake was primarily using an erroneous version of Cosine Similarity, it did have a correct definition of the Euclidean Distance (L2) formula (*Formula (3)*) but wasn't in use. Embeddings Lake added the API capability so the end-user could select Euclidean Distance as an option.

$$Euclidean\ Distance(A, B) = ||A - B|| \quad (3)$$

There are other distance metrics available that could be added as options. Research has shown that certain metrics may be more appropriate in particular circumstances. For example, some researchers have found that Manhattan distance is "consistently more preferable than Euclidean distance…for high dimensional data mining

applications" [28]. Due to this research, Embeddings Lake added an implementation of the Manhattan Distance (L1) as shown in *Formula (4)*.

$$Manhattan\ Distance(A, B) = |A - B| \quad (4)$$

4.1.2 Result Sorting

Vector Lake was also erroneously retrieving and sorting queried results because the computed distances and their corresponding vectors and metadata information were handled separately leading to mismatches. More specifically, the code would retrieve a list of tuples containing the shard's row index number and cosine distance that were closest to the queried vector, as shown below in *Algorithm 1*, which summarizes Vector Lake's implemented code.

ALGORITHM 1: Data Lake – Sort Query Results

```
Initialize three empty lists: results, computed_distances, rows
For each shard in the shards to search:
    Extract the indices and distances from the search results of the closet vectors in the graph
    Get the shard's metadata rows
    Use the extracted indices to find their corresponding vectors in the shard's metadata rows
    For each index in the closest indices, but reversed:
        append the shard's metadata for the index to the rows list
    Put the vectors found earlier into the results list
    Put the distances found earlier into the computed_distances list
    Put the results, computed_distances, and rows into a list of tuples
    Sort the list by distance
    Separate the sorted tuples back into the three individual lists: results, computed_distances, rows
    Return the top vector from the results list and metadata from the rows list
```

The rest of Data Lake's query sorting code tried to combine the closest distances, row data, and respective vectors into a list and sort by the distance. However, the raw row

data was inverted before all three components were recombined causing a mismatch of the data. The actual code for the inversion can be seen in line 19 of Appendix A.

Embeddings Lake corrected and simplified the sorting of the best query results by keeping the cosine distances and row indexes together as per *Algorithm 2*. Embeddings Lake also made an explicit effort to delete any duplicated returned results.

ALGORITHM 2: Embeddings Lake – Sort Query Results

```
Initialize one empty list: results
For each shard in the shards to search:
    Extract the indices and distances from the search results of the closet vectors in the graph
    For each index and distance returned:
        Get the raw row data from the shard that corresponds to the index value
        As a tuple, append the distance and the full raw row data to the results list
    Remove duplicates from results
    Sort results based on distance values
    Return the top vector from the results list and metadata from the rows list
```

4.1.3 HNSW Insertion Point

Vector Lake was also inconsistent with its query accuracy when the same lake was made with the same data repeatedly and given the exact same queries. An example of this is shown in *Table 5*, where a lake was instantiated with the Animals Dataset, a shard size of six, and quried with a radius value of zero repeated ten times.

Iteration	Accuracy (%)
1	65
2	60
3	74
4	64
5	66
6	52
7	66
8	66
9	67
10	63

Table 5: Varying Accuracies with Unseeded Insertion Point

Table 5 shows that Embeddings Lake can create a wide variance in query accuracy when nothing else changes. In this example, the same query test could result in accuracy scores ranging from 52% to 74%. This was due to Data Lake using a random, but not consistently, deterministic random seed value for its insertion point into the HNSW graph. Using an inconsistent random seed can produce the same query results so long as the graph is preserved in memory. However, as soon as it would have been deleted and needed to be rebuilt for later use, the value would have changed. Embeddings Lake added a default random seed value of 42 to ensure the entry point followed a deterministic pattern and would create consistent results. Though the results would be consistent, the results could be consistenly a lower accuracy score than what could be achieved with a different seed value. To account for this, the end user can change the seed if they want to use a different value and potentially return a different accuracy score.

4.2 Migration to the Cloud

The second goal of Embeddings Lake was to offload all processing and storage to the cloud. Embeddings Lake was built specifically to deploy in the Amazon Web

Services cloud. The general architecture can be seen in *Figure 9*, which includes numbered icons to orient the reader.

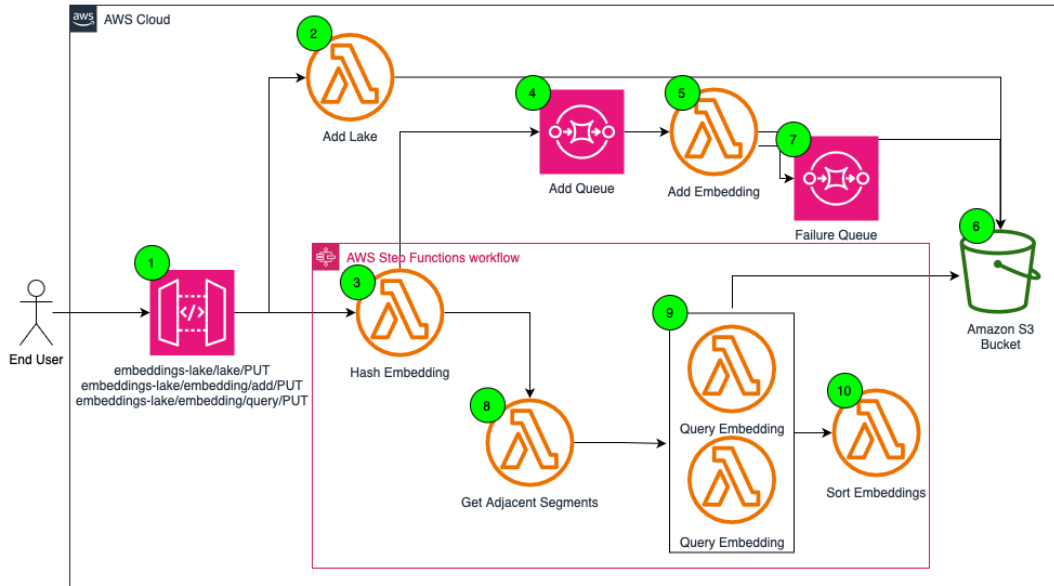


Figure 9: Embeddings Lake AWS Cloud Architecture

The first icon in *Figure 9* is a Representational State Transfer (REST) Application Programming Interface (API), which includes three endpoint Uniform Resource Locators (URLs) that the end user can make HTTP calls on to interact with Embeddings Lake. The first API call is embeddings-lake/lake/PUT, which transfers control to Figure 9 *Icon 2*, which represents a serverless function that will create a JavaScript Object Notation (JSON) configuration file as an object in Simple Storage Service (S3). An example of the lake configuration object and its content is shown in *Figure 10*.

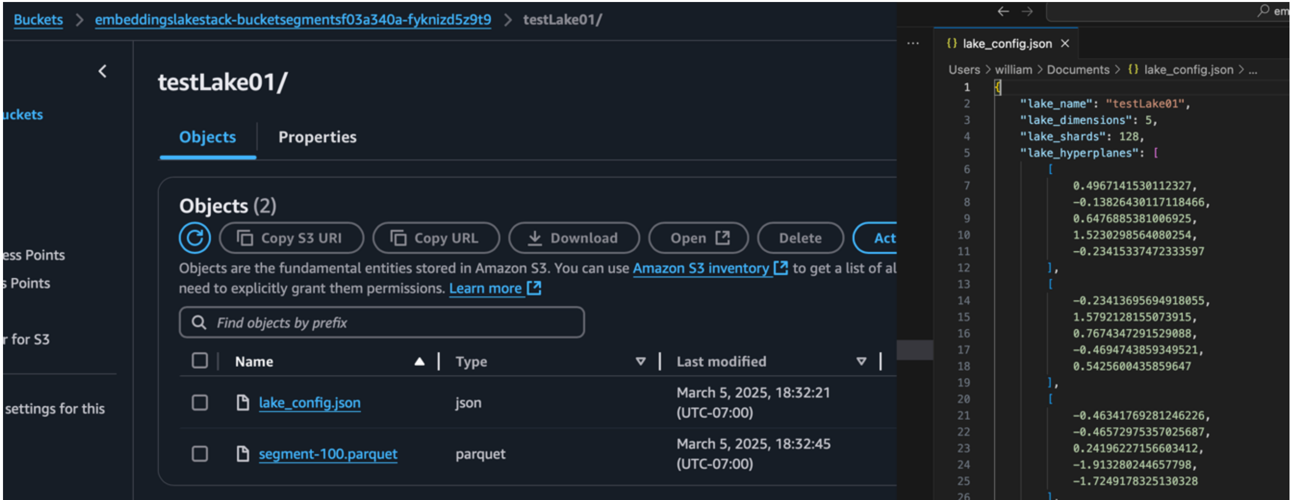


Figure 10: Embeddings Lake Config File Location and Contents

The second HTTP call in the API is embeddings-lake/embedding/add/PUT, which allows the user to add an embedding to an already instantiated lake. The API transfers control to the Lambda function in *Figure 9 Icon 3*, which hashes the embedding to identify the appropriate shard to add the embedding to. The function appends the name of the shard to a JSON message that is stored in a queue shown in *Figure 9 Icon 4* to wait for processing. The lambda function signified by *Figure 9 Icon 5* will retrieve the embedding messages from the queue to add the embedding to the lake. The queue can have multiple messages for all possible lakes at the same time. The lambda processor adds every embedding to the appropriate lake. Should the lambda fail to add an embedding to its respective lake, the embedding is added to a queue signified in *Figure 9 Icon 7*.

The third HTTP call in the API is embeddings-lake/embedding/query/PUT, which allows the user to query an embedding for the most similar embedding in a particular lake. Again, the API transfers control to the Lambda function in *Figure 9 Icon 3* to find the appropriate shard. The lambda in *Figure 9* Icon 8 will find all adjacent shards should

the user want to search those as well. The lambdas in *Figure 9 Icon 9* search the lake for similar embeddings, and all results are assembled and sorted in *Figure 9 Icon 10*'s lambda, which returns the most similar embedding in the HTTP return message's content body.

4.3 Serverless Cloud Solution

While Vector Lake was stateful, Embeddings Lake is stateless promoting a lightweight, cost-efficient architecture. Vector Lake introduced a "just in time" loading concept of a shard called a Lazy Bucket, which loaded a shard into memory only if it was needed to either add or query a vector. However, once the shard was loaded, the shard stayed loaded for the duration of the instantiated lake's operational use, even if shard would never have another vector added to it or be needed for a query. Embeddings Lake uses a serverless cloud model where all the components of Embeddings Lake are segmented into AWS Lambda functions (*Figure 9*), which are ideal for quick (less than 15 minutes) bursts of code usage. Embeddings Lake takes advantage of Lambda functions by ensuring that once a shard is used, the memory holding that shard is released at the completion of the API call using that respective shard. The state of the lake is preserved in S3 storage and only partially loaded into a lambda function when an API call is made to ensure memory usage is minimized to only what is needed and reduces cost. This also fixes a critical flaw of Vector Lake, which lacked the ability to reload a shard.

4.4 Improve Query Efficiency

Vector Lake's query method can search only one shard for related vectors. Embeddings Lake adds a radial search of tangential shards to ensure the best possible result is returned. Embeddings Lake runs the tangential search on each shard in its own lambda function in parallel, as signified in *Figure 9 Icon 9*. Running in parallel helps prevent a sacrifice of speed for accuracy. The radial search is a user-controlled value that can be any integer equal to or greater than zero. *Figure 11* helps explain how the radial search operates.

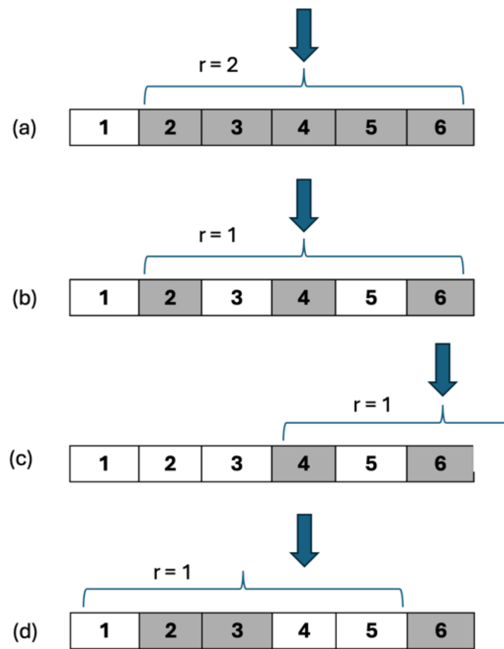


Figure 11: Radius search scenarios

The example lake in *Figure 11 (a)* has a shard capacity of six, with buckets two through six occupied with data. The queried embedding has an entry point at bucket four and a radius of two. Because every bucket within the query's radius has data, all five occupied buckets will be searched. In the *example shown in Figure 11 (b),* the entry point is the same, but the radius is equal to one. Though the radius is half of the value in the

first example, the distance of the buckets searched is the same because the second example has unoccupied buckets, which Embeddings Lake doesn't consider when calculating the radius. If the calculated entry point is at the edge of a lake, as in the example in *Figure 11 (c),* then the radius does not wrap around to the opposite end of the lake. In the *Figure 11 (d)* example, the queried hash points to a shard that is not occupied with any embeddings. In this circumstance, the entry shard is the nearest non-empty shard. If there are two nearest shards, then the shard with the lower index is selected.

5. PERFORMANCE EVALUATION AND DISCUSSION

A series of tests were conducted to validate how well Embeddings Lake met the goals of the project outlined in Section 4. The below sections provide details of each of those tests and their results.

5.1 Test Environment

To measure the effectiveness of Embeddings Lake, a similar testing environment was created to what was used in Section 2.4 to include the same Animals Dataset and CLIP Vit-B/32 embedding model. *Table 6* provides an updated list of relevant hardware and software.

Component	Version
Computer	November 2023 Macbook Pro
CPU	Apple M3 Pro
Memory	36 GB
Operating System	Sequoia 15.3
Python	3.10.16
AWS Code Development Kit	2.177.0

Table 6: Embeddings Lake Experimental Set-Up

The same Animals Dataset from Section 2.4 was used to test Embeddings Lake, but the Fruit Dataset was not used due to its inability to differentiate the commercial VDBMSs' performance. The second dataset used in the following experiments is the Butterflies Dataset, which consists of 4,379 images of fifty different types of butterflies inputted into the data lakes and two test images per butterfly species. *Figure 12*: Butterflies Dataset sample images displays eight sample images of the Butterflies Dataset.

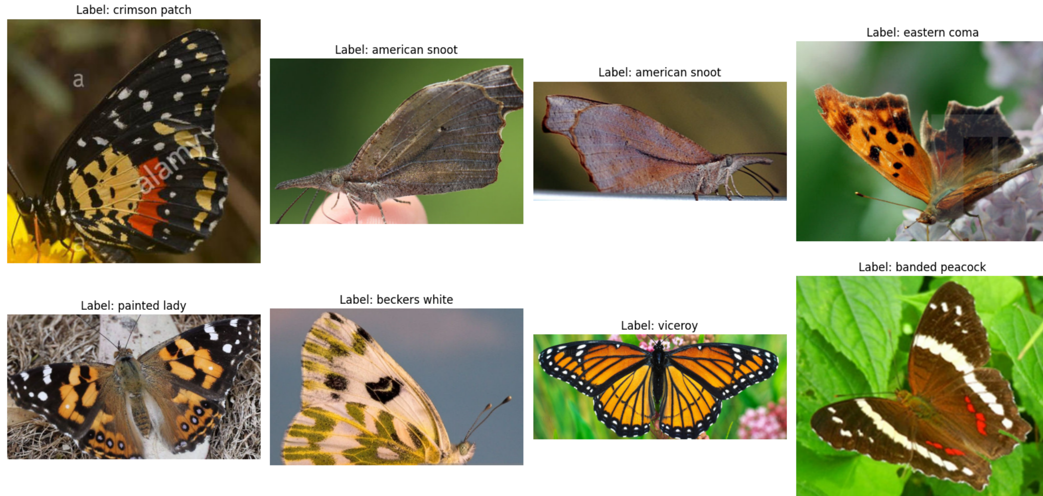


Figure 12: Butterflies Dataset sample images

When instantiating a new lake in Embeddings Lake, the user provides a shard count, which is only an approximation of the actual number of shards. The actual number of shards must be a power of two because the LSH function's hyperplanes split the space into two regions of a binary zero or one, and every additional hyperplane doubles the number of partitions. In the experiments, every shard value used was the first possible approximate shard value that would result in the next power of two. *Table 7* shows the user-provided shard values and their corresponding actual shard values, which were used throughout the experiment. The actual shard values in the experiments are powers of two ranging from 2^1 (2) to 2^{15} (32768). *Table 7* can be used as a reference in the experiments to get the actual shard values.

User-Inputted Shards and Actual Shards Values															
User-Inputted:	2	3	6	12	23	46	91	182	363	725	1449	2897	5793	11586	23171
Actual:	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768

Table 7: Experiment Shard Values for Reference

Rerunning or reusing the CLIP embedding model does not change any image embedding's vector values iteration from iteration. For example, the embedding values

for the Animals Dataset test query images shown in *Table 8* will be the same for every test. This ensures that the test query embeddings are consistent across all experiments.

Test image	CLIP Vit-B/32 512-dim embedding value
'Animals-Test/hippopotamus/4ba6ebbc29.jpg'	[0.19532963633537292, …. 0.09531612694263458]
'Animals-Test/sparrow/68a66379c8.jpg'	[0.34455978870391846, …. -0.22413086891174316]

Table 8: Test Query Image Example Embedding Values

5.2 Distance Metric Comparison

5.2.1 Animals Dataset

Figure 13 displays line graphs summarizing the accuracy performance of the three different distance metrics Embeddings Lake implemented in Section 4.1.1. The figure displays the graphs of the same Animals Dataset query test executed against a different version of the Animals Dataset Lake that is sharded in accordance with *Table 7*. The general trend in every graph is accuracy improves as the radius improves. In the figure for the graph labeled “Shard 2,” the three metrics’ accuracy plateaus when the radius value is equal to one. This is a reasonable result because when the lake is broken up into two shards, a query with a radius of one can search up to three buckets but will only search the existing two buckets. Increasing the radius beyond that value searches no additional buckets, and consequently, the accuracy stays the same. In the graphs with higher shard values, the accuracy scores become less and less linear, and the three models regularly leapfrog each other as the radial value increases, leading to no clear superior metric. In some graphs, however, specific metrics dominate at all radial values. For example, in the “Shard 46” graph, the Euclidean metric dominates at all radii, but in the

“Shard 725” graph, the Manhattan metric has the highest accuracy scores throughout the test. At differing shard values, every metric reaches accuracy levels in the >90% range.

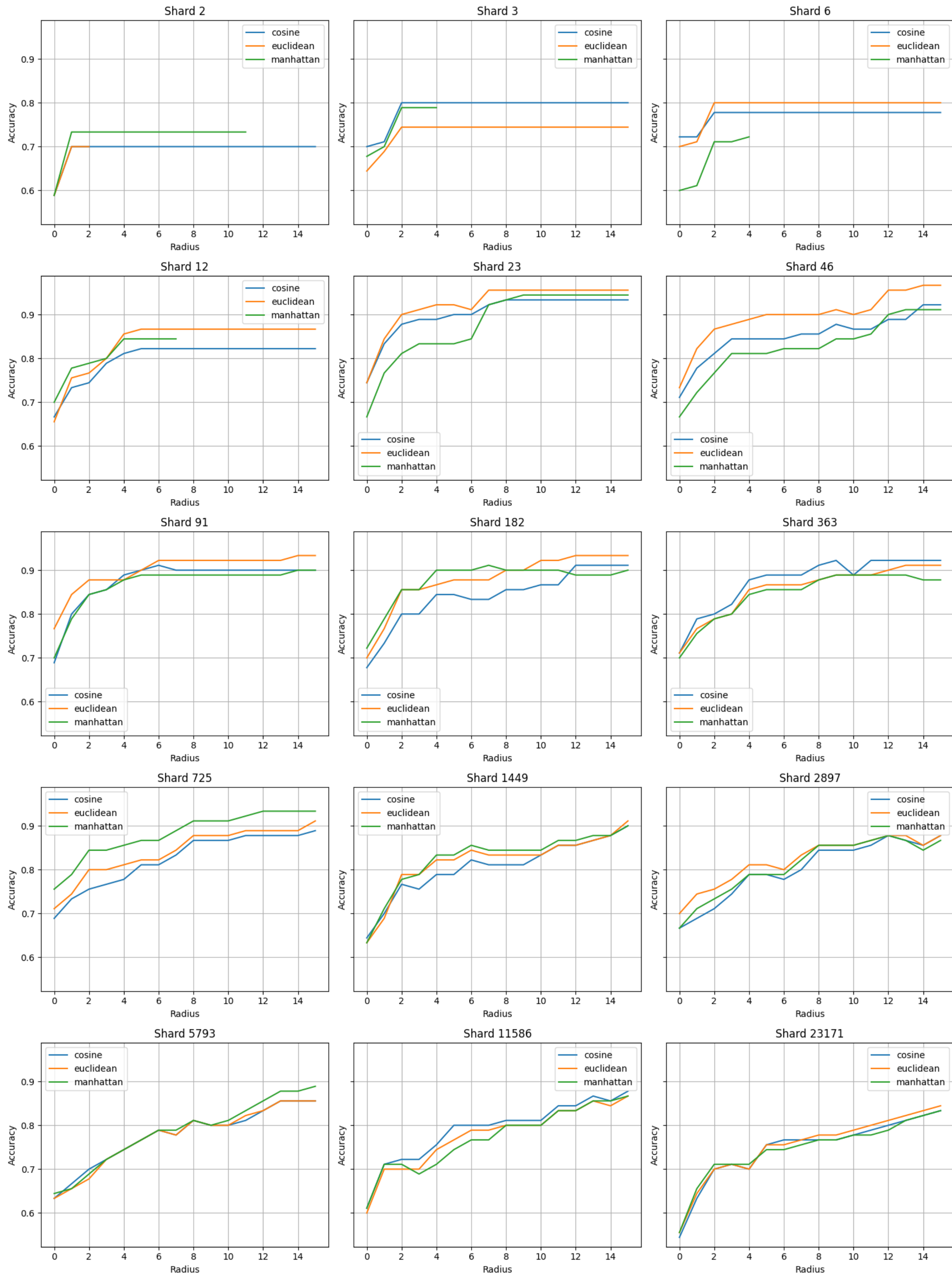


Figure 13: Distance Metric Comparison for Animals Dataset with 512 Dimensions

Table 9, Table 10, and *Table 11* summarize the highest and lowest accuracy values of the Cosine, Euclidean, and Manhattan distance metric queries. In all three query types, the lowest accuracies were achieved with a radius of zero, suggesting that adding a radial factor to Embeddings Lake had a significant impact on maximizing accuracy. Cosine and Manhattan each had six shard versions that achieved accuracy scores of over 90%, while Euclidean had seven shard versions that achieved the same threshold. Euclidean achieved the highest score for any radii and shard count with an accuracy of 96%.

	2	3	6	12	23	46	91	182	363	725	1449	2897	5793	11586	23171	Avg
Highest	70	80	77	82	93	92	91	91	92	88	90	87	85	87	83	84.5
Radius	*1*	*2*	*2*	*5*	*8*	*14*	*6*	*12*	*9*	*15*	*15*	*12*	*13*	*15*	*15*	*9.6*
Lowest	58	70	72	66	74	71	68	67	71	68	64	66	63	61	54	68.9
Radius	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0.0*

Table 9: Best and Worst Accuracies by Shard for Animals Dataset, 512 dimensions and Cosine query

	2	3	6	12	23	46	91	182	363	725	1449	2897	5793	11586	23171	Avg
Highest	73	78	72	84	94	91	90	91	88	93	90	87	88	86	83	85.7
Radius	*1*	*2*	*4*	*4*	*9*	*13*	*14*	*7*	*9*	*12*	*15*	*12*	*15*	*15*	*15*	*10.1*
Lowest	58	67	60	70	66	66	70	72	70	75	63	66	64	61	55	68.2
Radius	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0.0*

Table 10: Best and Worst Accuracies by Shard for Animals Dataset, 512 dimensions and Manhattan (L1) query

	2	3	6	12	23	46	91	182	363	725	1449	2897	5793	11586	23171	Avg
Highest	70	74	80	86	95	96	93	93	91	91	91	87	85	86	84	85.4
Radius	*1*	*2*	*2*	*5*	*7*	*14*	*14*	*12*	*13*	*15*	*15*	*12*	*13*	*15*	*15*	*10.3*
Lowest	58	64	70	65	74	73	76	70	71	71	63	70	63	60	55	68.9
Radius	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0.0*

Table 11: Best and Worst Accuracies by Shard for Animals Dataset, 512 dimensions and Euclidean (L2) query

Figure 14 summarizes the highest accuracy scores from *Table 9, Table 10,* and *Table 11* in the form of line graphs. The graphs reveal a general trend of accuracy starting at its lowest point with a small to nonexistent radius and then increasing and plateauing

around a shard size of 46 and then gradually decreasing again as the shard size continues to increase.

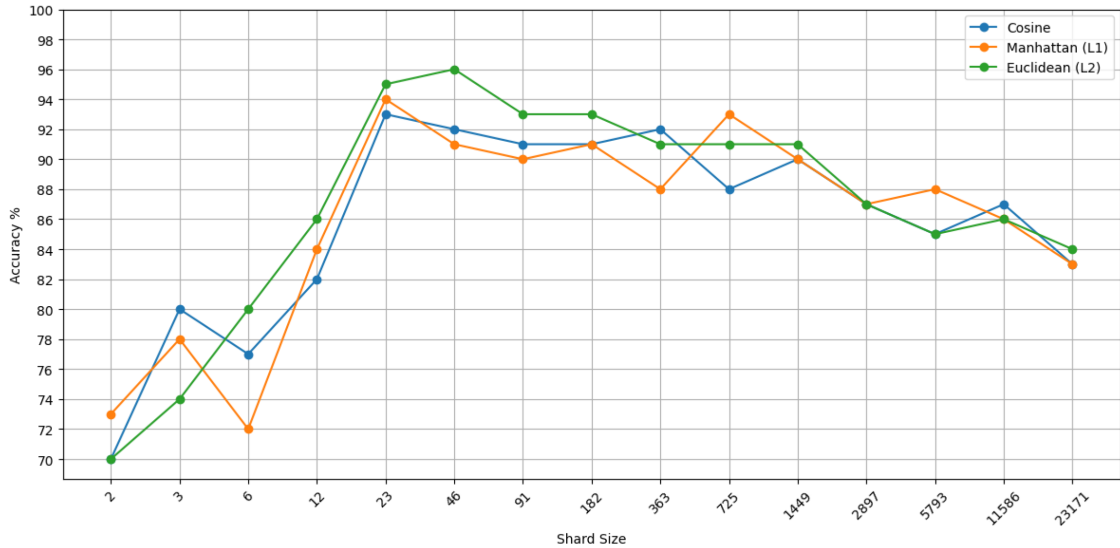


Figure 14: Highest Accuracy Scores Animals Dataset w/ 512 Dimensions

5.2.2 Butterflies Dataset

Appendix B-1 shows a similar trend to *Figure 13*, where accuracy increases as the radius increases in every shard size, but it shows no clear, dominant distance metric performer.

Appendix B-2, B-3, and B-4 display the highest and lowest accuracy scores for each distance metric. Unlike the Animals Dataset, which had all three metrics perform about the same on average, the Manhattan distance metric achieved a higher peak accuracy average than the other two metrics.

5.3 Dimensionality

Section 4.1.1 mentions how research inspired the addition of the Manhattan metric to Embeddings Lake because the Manhattan metric allegedly performs better at

higher dimensions. The Butterflies Dataset was rerun at a higher dimension to address the research's assertion and help determine whether adding the metric was worthwhile. To rerun the data at the the higher dimension, a different CLIP model had to be used. The test used the OpenAI Clip ViT-L-14 model with the "laion2b_s32b_b82k" pretrained dataset. The ViT-L-14 model was needed to provide 768 dimensions per image. Because the additional dimensions require more processing time, the test was run on shard sizes of 6, 23, 91, 363, 1449, 5793, and 23171, and the general trend results are shown in *Figure 15*. The results show overall higher accuracy values than the test's 512-dimension counterpart in Appendix B-1.

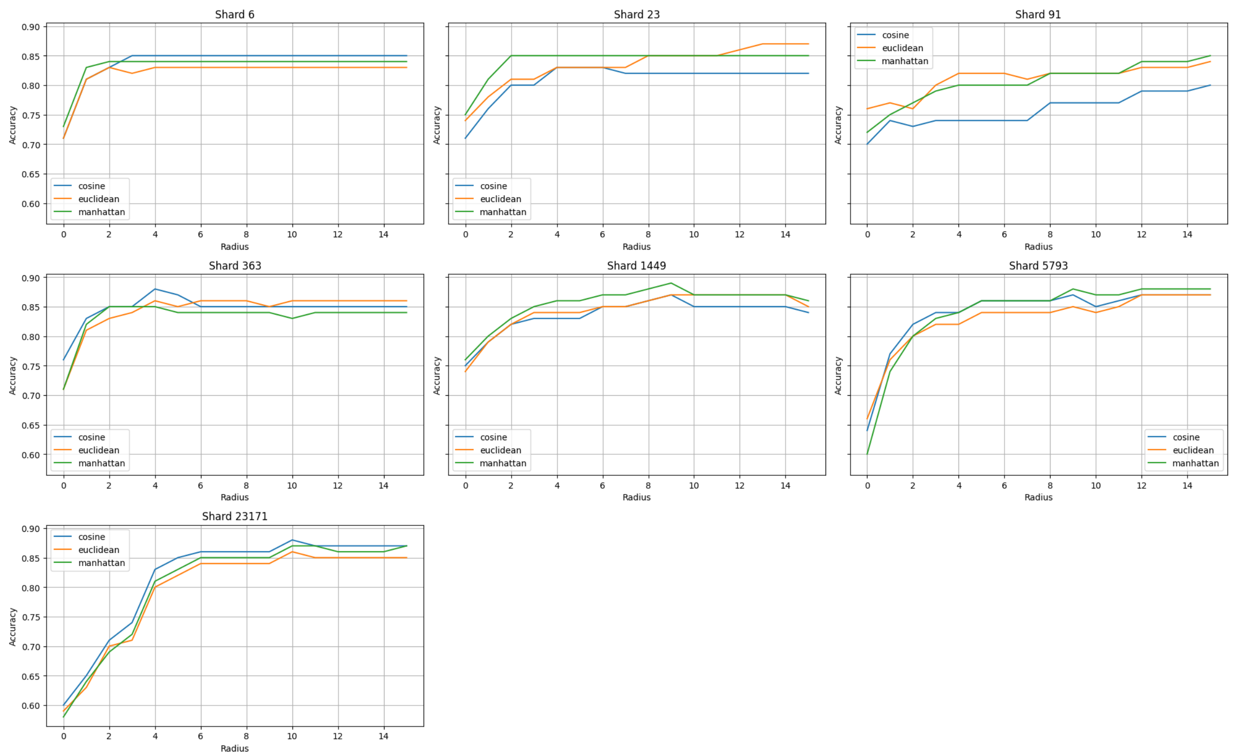


Figure 15: Distance Metric Comparison for Butterflies Dataset with 768 Dimensions

Table 12 displays the percentage increase in accuracy for each metric at every radius for every shard. The raw data is available in Appendix B-5. The last column in *Table 12* is a tally of each metric in the last column of Appendix B-5. The results show

that all three models had a similar increase in accuracy, but the Euclidean metric had the fewest best increases for any given combination of shard size and radius. The results also seem to suggest that the Cosine metric had the most improvement when varying dimensionality, not Manhattan distance.

Metric	Average Percent Increase	Tally Best Row Increases
Cosine	37.3	45
Manhattan (L1)	35.5	43
Euclidean (L2)	37.0	24

Table 12: Summary of Accuracy Percentage Change for Dimensionality Increase

5.4 Efficiency

One of the goals of Embeddings Lake was not to sacrifice search speed for accuracy by implementing a parallel search. To determine how well Embeddings Lake achieved this goal, a sampling of time measurements for the 512-dimension Animals Dataset with a Manhattan Metric were taken and are shown in *Figure 16*. Generally, the time to complete a query grows at a linear rate as a function of the radius. There were a few exceptions to the trend, such as when shard size was 12 and 23, where the time plateaus in the higher radial values.

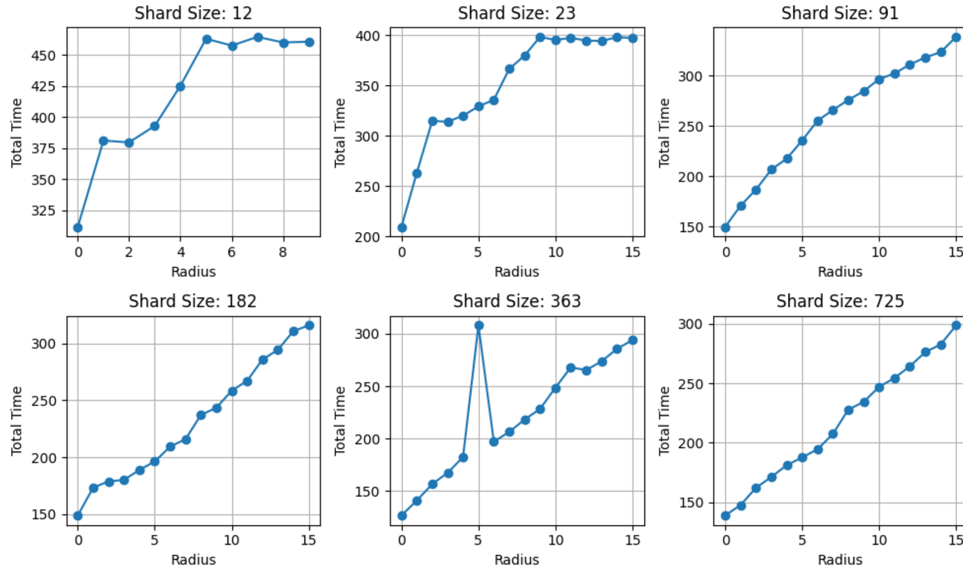


Figure 16: Sampling of time measurements for Animals Dataset, 512 Dimensions, Manhattan query

5.5 Embeddings Lake vs. Other VDBMS

A goal of Embeddings Lake was to migrate to the cloud while maintaining competitive results compared to other commercial offerings. Accuracy scores can already be compared to the containerized 3rd party solutions for the Animals Dataset used in Section 2.4. That experiment's test was rerun with the Butterflies Dataset, and the results are shown in *Figure 17*.

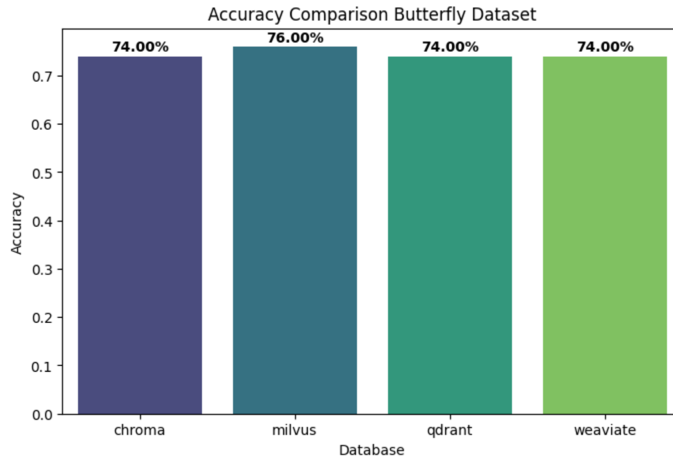


Figure 17: Accuracy Results for Open-Source VDBMS

Embeddings Lake is a 100% cloud-based VDBMS, while the comparative VDBMSs in *Figure 17* were designed to run exclusively on a local machine. *Table 13* displays the accuracy and timing results of another 100% cloud-based system, AWS Opensearch. Opensearch is a NoSQL database that added the Facebook AI Similarity Search (FAISS) k-nearest neighbors search capability in May 2024 [29], which was what was used in this comparison experiment. The Opensearch cluster configuration can be seen in Appendix C.

	Accuracy	Time (sec)
Animals	95%	15.41
Butterflies	76%	10.47

Table 13: AWS Opensearch Comparison by dataset

Table 14 displays a summarized result of Embeddings Lake and all other third party VDBMSs highest achieved query accuracy scores. Though Embeddings Lake achieved the lowest accuracy for the Butterflies Dataset, it did achieve the highest accuracy score for the Animals Dataset.

	Embeddings Lake	Opensearch	Chroma	Milvus	Qdrant	Weaviate
Animals	96%	95%	92%	93%	93%	92%
Butterflies	72%	76%	74%	76%	74%	74%

Table 14: Highest accuracy achieved for each VDBMS and dataset with 512 dimensions

5.6 Cost

A goal of Embeddings Lake was to provide a cloud-based system that was more cost efficient than the industry's existing solutions. *Table 15* provides a summary of how

well Embeddings Lake is estimated to cost compared to various other VDBMS estimated pricing. The detailed calculations and sources for the pricing information can be found in Appendix D.

Embeddings	Embeddings Lake	Weaviate	Pinecone	AWS Opensearch	AWS Opensearch Serverless
Animals Dataset	$3.62	$25.00	$25.00	$26.28	$350.42
1 Million	$18.62	$48.64	$25.00	$26.55	$350.45

Table 15: Cost Estimate for Embeddings Lake vs Commercial VDBMS

The most significant cost increase for Embeddings Lake moving from the Animals Dataset to the 1 million embeddings scenario is the Lambda function to add the embedding to the respective shard's parquet file in S3. Appendices D-5 through D-8 show Weaviate's cost calculator, which allows for a more direct dimension-to-dimension comparison with the 512-dimension version of the Animals Dataset used throughout this section. The most significant factor driving the cost of Weaviate's estimate is the "Storage Type" selection, which can be either "Compression" or "Performance." For this research's comparison, "Performance" was selected because "Compression" utilizes algorithms such as product quantization (PQ) and scalar quantization (SQ) [30], which aren't implemented in Embeddings Lake. Even if the comparison utilized "Compression" for the 1 million embeddings category, Embeddings Lake without compression implemented would still cost less than Weaviate's $25 estimate per Appendix D-8.

AWS Opensearch is displayed in two categories in *Table 15* – the conventional structure, which was what was used in *Table 13* and the "Serverless" version costing over three hundred dollars per month. The "Serverless" version was included in the comparison because its intent is more aligned with Embeddings Lake than the

conventional option because both Embeddings Lake and Opensearch Serverless abstract away the virtual machine to the point where the end user will never have to worry about ensuring the computer is properly secured or updated with proper patching and software.

5.7 Effect of Hashing

Locality-sensitive hashing plays a significant role in how Embeddings Lake distributes and shards data. *Figure 18* shows how the Animals Dataset Lake distributed all images labeled "Gorilla" with 512 dimensions and 725 shards.

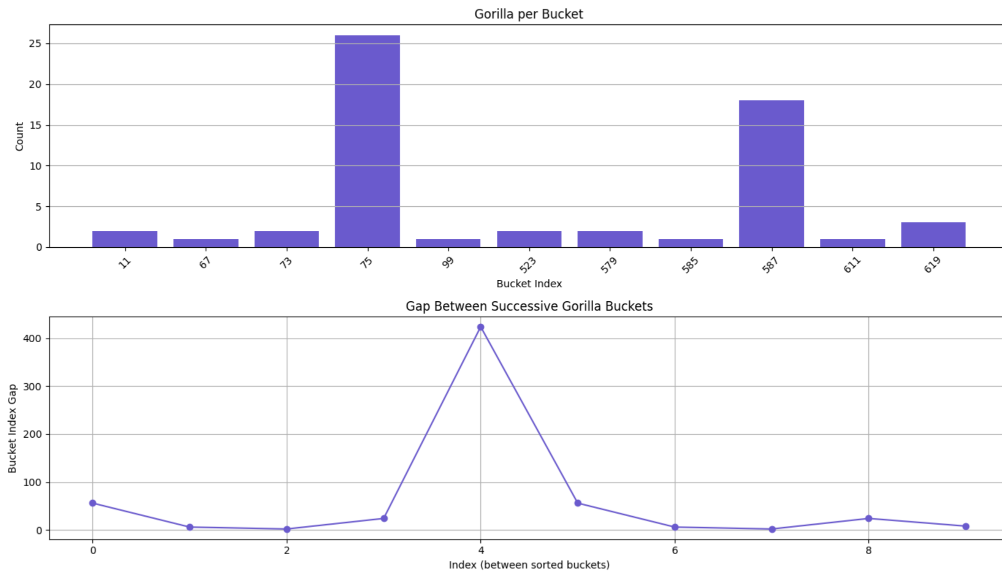


Figure 18: LSH distribution of gorilla images with 512 dimensions and 725 shards

Figure 19 shows a similar pair of graphs but for images labeled "Eagle". These examples highlight how LSH can influence the spread of similar images. The gorilla images are generally split apart into two main buckets, but those buckets are far apart from one another. The eagle images are also generally split into two main buckets, but those buckets are sequentially next to each other.

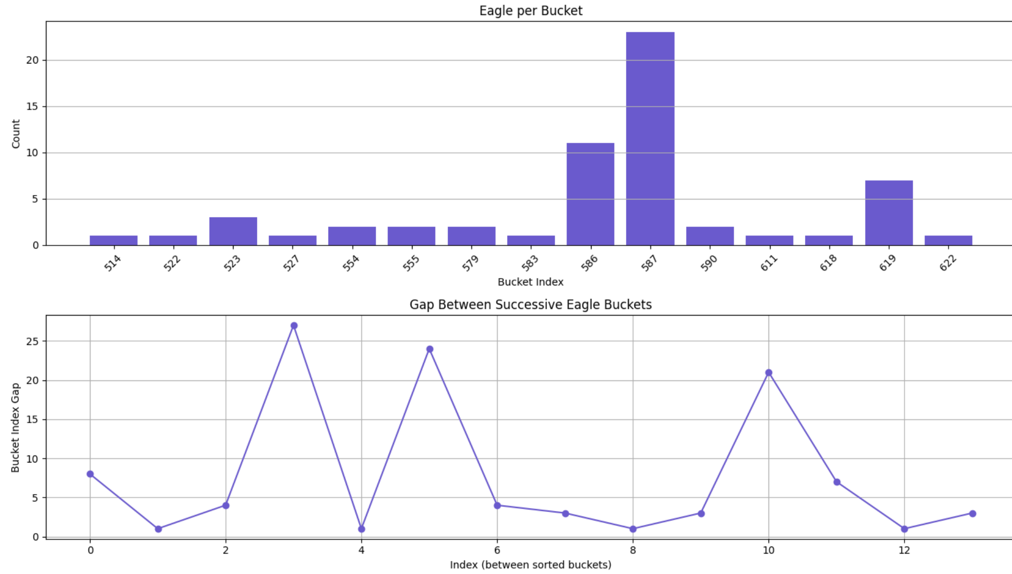


Figure 19: LSH distribution of eagle images with 512 dimensions and 725 shards

Figure 20 shows how the Animals Dataset Lake distributed all images labeled "Gorilla" with 512 dimensions and 23171 shards instead of the 725 shards in *Figure 18*. Increasing the lake's shard value had a flattening effect on the distribution of the gorilla images, leading to a higher number of buckets where a concentration of gorilla images resides.

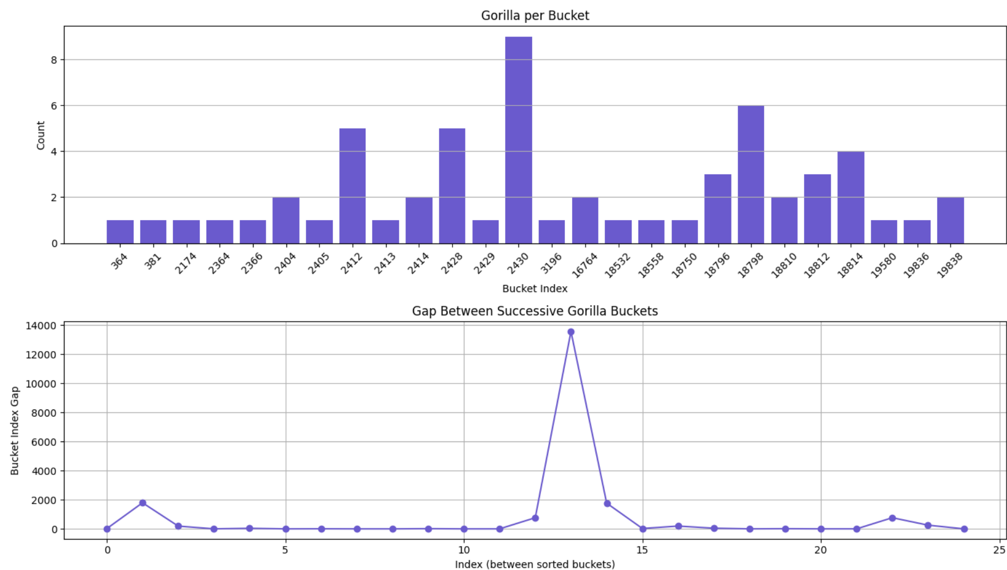


Figure 20: LSH distribution of gorilla images with 512 dimensions and 23171 shards

Figure 21 shows how the Animals Dataset Lake distributed all the images labeled "Eagle" with 512 dimensions and 23171 shards instead of the 725 shards in *Figure 19*. The change in the distribution of eagle images does not appear to be as dramatic as the change for gorilla images. On the contrary, the eagle image distribution is now more concentrated in one bucket than in two.

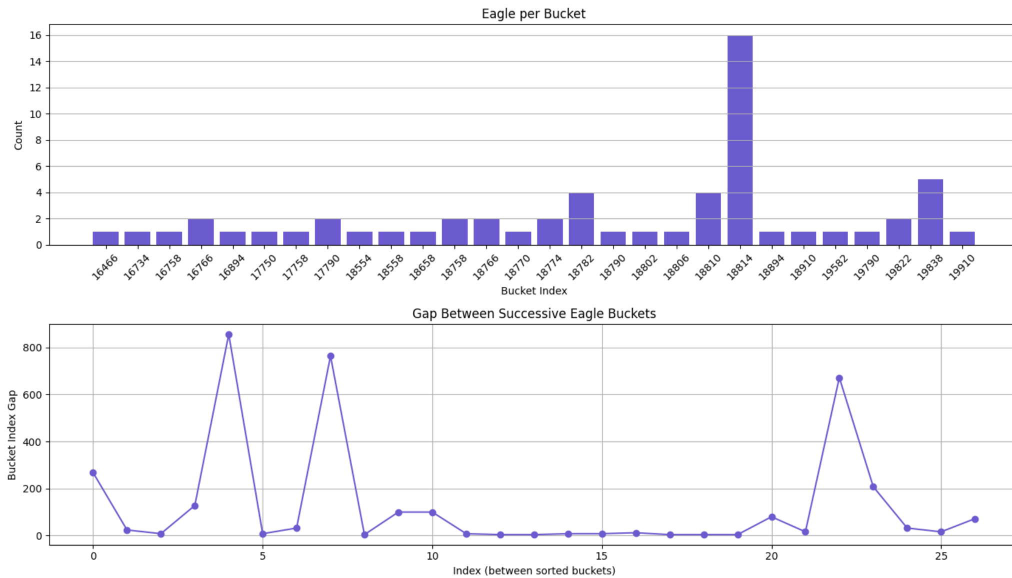


Figure 21: LSH distribution of eagle images with 512 dimensions and 23171 shards

Figure 22 and *Figure 23* show histograms of the overall distribution for the animal categories in the Animals Dataset, with the twenty highest spread values for lakes with shard sizes of 725 and 23171, respectively. Though the two histograms share a similar set of animal categories, the ranking of the spread does not stay consistent as shard size changes. For example, each graph animal with the highest spread, "panda" and "bear", aren't even listed on the opposing graph.

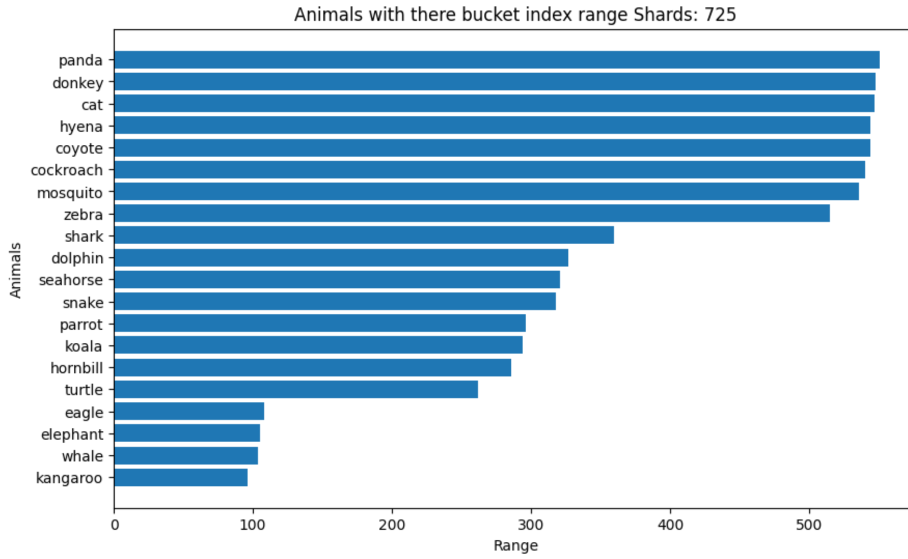


Figure 22: LSH distribution of all animal categories with 512 dimensions and 725 shards

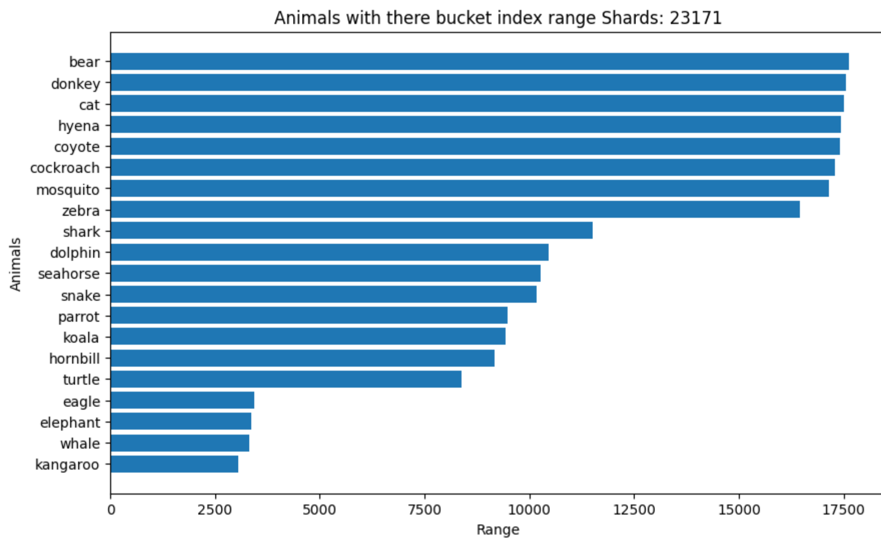


Figure 23: LSH distribution of all animal categories with 512 dimensions and 23171 shards

6. CONCLUSIONS

6.1 Summary

The goal of Embeddings Lake was to provide a 100% cloud-based, serverless VDBMS that maintains high accuracy while providing better cost efficiency than the commercial alternatives. The testing results of Section 5 show that Embeddings Lake can produce both similar and sometimes more accurate querying results than commercial offerings at a lower price. Embeddings Lake can significantly impact the VDBMS community because the pay-to-use alternatives are not as open-source as Embeddings Lake, allowing end-users unlimited opportunity to customize a competent solution without relying on 3rd party intermediary approvers.

6.2 Future Work

Though Embeddings Lake is a competitive solution in the VDBMS community, there are opportunities to improve performance. Some of those opportunities are highlighted in the below sections.

6.2.1 Improve Accuracy

Based on the results in Section 5.2, accuracy scores were at their lowest values when the radius was zero and improved as the radius increased. A possible explanation for this is when the radius is a low value, query matching is significantly more dependent upon how Embeddings Lake hashes the query image than at higher radii values. For example, in a hypothetical scenario, if all embeddings of an ant animal are distributed to a far end of the lake's buckets, but a particular query for a different ant image placed the

entry point at the opposite end of the lake's buckets, getting a match would require a high enough radii value to reach the ant images in the distant buckets. This isn't a preposterous scenario because, as Section 5.7 shows, similar items can land on opposite sides of the lake due to the binary nature of LSH. The LSH hash result is based on dot products with each hyperplane, so small changes in the vector can flip the outcome from the bucket with an index of zero to the bucket with an index of negative one. Therefore, semantically similar items may reside on opposite sides of the lake, even though they may visually appear to be similar items. Future work could look at how sharding can be improved to reduce the burden of having a large radius but also include an easy, intermediate change of having the radius wrap around to the lake's opposite end of buckets.

Embeddings Lake does not allow for customization of its HNSW algorithm parameters. Granting end-users access to tune these parameters could allow for higher accuracy scores. Two such parameters are the "M" or maximum number of edges a node can have in a graph and the "EF Construction" or entry factor, which is the number of candidate neighbors considered for connection during index construction. Adding these features can, however, introduce trade-offs because increasing both values generally leads to higher accuracy but slows performance speed [31].

6.2.2 Improve Efficiency

Though timing measurements weren't conducted for adding vectors to Embeddings Lake, future research could establish a baseline and further improve performance by adding locks and lock management so that the Lambda function handling embedding additions executes in parallel.

Section 4 outlined how a goal of Embeddings Lake was to search nearby buckets in parallel instead of sequentially to prevent a sacrifice of query speed for query accuracy. However, despite some exceptions, the results in Section 5.4 show that query time generally continues to grow at a linear rate. Some growth in query time should be expected because every shard that is searched in parallel represents one more set of results that need to be sorted in *Figure 9 Icon 10's* lambda function. *Figure 9 Icon 10*'s lambda function organizes all the results from the parallel shard searches by computed distance and returns the closest result to the end-user. Future work could temporarily rebuild Embeddings Lake to run the shard queries in sequence to measure whether the query time grows at a significantly faster linear rate than it is now. Future work could also allow the end-user to customize how many of the top results each parallel query feeds *Icon 10*'s lambda function. Embeddings Lake currently has each query return the top four closest embeddings. Lowering the value could result in a faster query but risk decreasing accuracy.

In Section 5.6, one of the methods used to estimate cost savings for Weaviate was product (PQ) and scalar (SQ) quantization for vector compression. PQ could be evaluated as an alternative or supplement to Emeddings Lake's existing organizational algorithms. PQ can help break up high-dimensional embeddings into smaller, more manageable chunks stored in tables for searches without losing their integrity. Research has shown that when combined with an inverted file system, PQ's accuracy to efficiency trade-offs can outperform state-of-the-art alternatives [32].

6.2.3 Improve Utility

Embeddings Lake does not have an option to delete data in the lakes. Though adding this feature would likely be straightforward, maintaining query accuracy levels after consistent updates may be more difficult because researchers at the University of Electronic Science and Technology in Chengdu, China, observed through the study "that the HNSW can create unreachable points in [a] graph after deletion and insertion operations, leading to adverse affects." Adding their proposed algorithms, HNSW replace-update (HNSW-RU) and Mutual Neighbor replace-update (MN-RU) could significantly reduce the number of unreachable points and improve HNSW update speeds [32].

REFERENCES

[1] N. Khan, I. Yaqoob, I. A. T. Hashem, Z. Inayat, W. Kamaleldin , M. Ali, M. Alam, M. Shiraz and A. Gani, "Big Data: Survey, Technologies, Opportunities, and Challenges," *The Scientific World Journal,* no. 712826, p. 18, 2014.

[2] Z. Liang, Y. Yang and Q. Tian, "SIFT meets CNN: A decade survey of instance retrieval," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 40, no. 5, pp. 1224-1244, 2017.

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ArXiv,* vol. 11929, 2020.

[4] Y. Han, C. Liu and P. Wang, "A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge," *arXiv,* no. 2310.11703, 2023.

[5] T. Taipalus, "Vector database management systems: Fundamental concepts, use-cases, and current challenges," *Cognitive Systems Research,* vol. 85, no. 101216, 2024.

[6] Z. Jing, Y. Su and Y. Han, "When Large Language Models Meet Vector Databases: A Survey," *arXiv,* vol. 2402.01763, 2024.

[7] E. Cardenas, "Vector Library versus Vector Database," Weaviate, 1 December 2022. [Online]. Available: https://weaviate.io/blog/vector-library-vs-vector-database#example-use-cases. [Accessed 1 December 2024].

[8] "DB-Engines Ranking of Vector DBMS," November 2024. [Online]. Available: https://db-engines.com/en/ranking/vector+dbms. [Accessed 28 November 2024].

[9] M. Aumüller, E. Bernhardsson and A. Fait, "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems,* vol. 87, no. 101374, 2020.

[10] K. Seth, "kaggle," kaggle, 2021. [Online]. Available: https://www.kaggle.com/datasets/kritikseth/fruit-and-vegetable-image-recognition. [Accessed 15 November 2024].

[11] S. Banerjee, "Animal Image Dataset," Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/iamsouravbanerjee/animal-image-dataset-90-different-animals. [Accessed 15 November 2024].

[12] A. Radford, J. . W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, . G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger and I. Sutskever, "Learning Transferable Visual Models From Natural Language Supervision," *arXiv,* vol. 2103.00020, 2021.

[13] Weaviate, "Resource Planning," Weaviate, 2024. [Online]. Available: https://weaviate.io/developers/weaviate/concepts/resources#the-role-of-memory. [Accessed 30 11 2024].

[14] Qdrant , "Qdrant Documentation Concepts - Storage," Qdrant, 2024. [Online]. Available: https://qdrant.tech/documentation/concepts/storage/#vector-storage. [Accessed 30 November 2024].

[15] Milvus, "Milvus Memory Mapping," Milvus, 2024. [Online]. Available: https://milvus.io/docs/mmap.md#Configure-memory-mapping. [Accessed 30 November 2024].

[16] Weaviate, "Horizontal Scaling," Weaviate, 2024. [Online]. Available: https://weaviate.io/developers/weaviate/concepts/cluster#motivation-1-maximum-dataset-size. [Accessed 1 December 2024].

[17] Milvus, "Milvus Glossary," Zilliz, [Online]. Available: https://milvus.io/docs/v2.2.x/glossary.md#Milvus-cluster. [Accessed 4 December 2024].

[18] Qdrant, "Qdrant Distributed Deployment," Qdrant, [Online]. Available: https://qdrant.tech/documentation/guides/distributed_deployment/#distributed-deployment. [Accessed 4 December 2024].

[19] Chroma, "Chroma Deployment," Chroma, [Online]. Available: https://docs.trychroma.com/deployment. [Accessed 4 December 2024].

[20] Weaviate, "Weaviate Cloud," Weaviate, [Online]. Available: https://weaviate.io/developers/wcs. [Accessed 4 December 2024].

[21] Qdrant, "About Qdrant Managed Cloud," Qdrant, [Online]. Available: https://qdrant.tech/documentation/cloud/#. [Accessed 4 December 2024].

[22] Weaviate, "Weaviate Cloud Serverless Price Model," 24 03 2025. [Online]. Available: https://weaviate.io/pricing/serverless. [Accessed 24 03 2025].

[23] R. Guo , X. Luan , L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, Z. Cao, Y. Qiao , T. Wang, B. Tang and C. Xie, "Manu: A Cloud Native Vector Database Management System," *arXiv,* vol. 2206.13843, p. 14, 2022.

[24] Amazon Web Services, "Working with vector search collections," Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/opensearch-service/latest/developerguide/serverless-vector-search.html#serverless-vector-limitations. [Accessed 4 December 2024].

[25] J. J. Pan, J. Wang and G. Li, "Vector Database Management Techniques and Systems," in *In Companion of the 2024 International Conference on Management of Data (SIGMOD/PODS '24)*, New York, Association for Computing Machinery, 2024, p. 597–604.

[26] Chroma, "Chroma AWS Deployment," Chroma, [Online]. Available: https://docs.trychroma.com/deployment/aws. [Accessed 4 December 2024].

[27] A. Miasoiedov, "Vector Lake," 3 August 2023. [Online]. Available: https://github.com/msoedov/vector_lake. [Accessed 20 12 2024].

[28] C. C. Aggarwal, A. Hinneburg and D. A. Keim, "On the Surprising Behavior of Distance Metrics in High Dimensional Space," *Lecture Notes in Computer Science, vol 1973. Springer, Berlin, Heidelberg.,* p. 420–434, 12 October 2001.

[29] AWS, "Amazon OpenSearch Service now supports OpenSearch version 2.13," Amazon Web Services, 21 May 2024. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2024/05/amazon-opensearch-

supports-opensearch-version-2-13/?utm_source=chatgpt.com. [Accessed 17 04 2025].

[30] Weaviate, "Compression," Weaviate, [Online]. Available: https://weaviate.io/developers/weaviate/starter-guides/managing-resources/compression#compression-algorithms. [Accessed 18 April 2025].

[31] Milvus, "HNSW," Zilliz, [Online]. Available: https://milvus.io/docs/hnsw.md. [Accessed 17 04 2025].

[32] H. Jégou, M. Douze and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 33, no. 1, pp. 117-128, 2011.

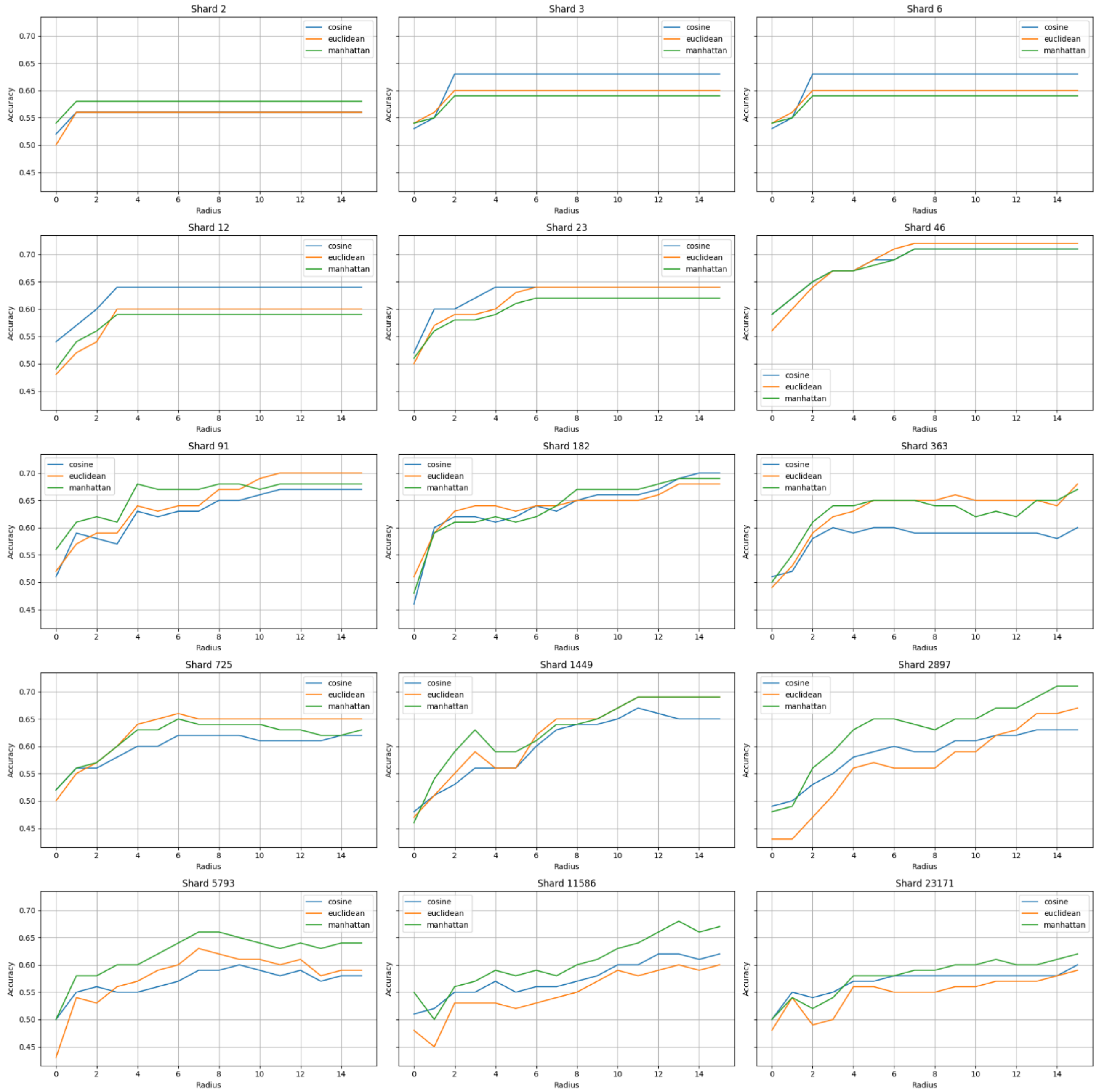
[33] W. Xiao, Y. Zhan, R. Xi , M. Hou and J. Liao, "Enhancing HNSW Index for Real-Time Updates: Addressing Unreachable Points and Performance Degradation," *arXiv,* vol. 2407.07871, 2024.

APPENDIX A: DATA LAKE SORT IMPLEMENTATION

```
01    def _query(self, vector, k: int = 4) -> list:
02        results = []
03        computed_distances = []
04        rows = []
05        vector = np.array(vector)
06        ts = time.time()
07        rows_append = rows.append
08        for shard in self.adjacent_routing(vector):
09            te = time.time() - ts
10            logger.debug(f"adjacent routing took vps{1/te:.1f}")
11            shard._lazy_load()
12            shard_np = np.array(shard.vectors)
13
14            closest_indices_d = shard.search(vector, k=k)
15            closest_indices = [idx for idx, _ in closest_indices_d]
16            shard_rows = shard.dirty_rows
17            closest_vectors = shard_np[closest_indices]
18
19            for idx in closest_indices[::-1]:
20                rows_append(shard_rows[idx])
21            shard_rows = []
22
23            results.extend(list(closest_vectors))
24            computed_distances.extend([d for _, d in closest_indices_d])
25            if len(results) >= k:
26                break
27            logger.debug("next shard")
28
29        result_vectors = np.array([vector for vector in results])
30        if not result_vectors.any():
31            return [], []
32
33        combined = list(zip(computed_distances, result_vectors, rows))
34        # Sort the combined list by distances
35        sorted_combined = sorted(combined, key=lambda x: x[0])
36        # Unzip the sorted list
37        sorted_distances, sorted_vectors, sorted_rows = zip(*sorted_combined)
38        return sorted_vectors[:k], sorted_rows[:k]
```

APPENDIX B: BUTTERFLY DATASET DATA

B-1. Distance Metric Comparison with 512 Dimensions



B-2. 512 Dimensions Cosine Query

	2	3	6	12	23	46	91	182	363	725	1449	2897	5793	11586	23171	Avg.
Highest	56	63	63	64	64	71	67	70	60	62	67	63	60	62	60	63.4
Radius	*1*	*2*	*2*	*3*	*4*	*7*	*11*	*14*	*3*	*6*	*11*	*13*	*9*	*12*	*15*	*7.5*
Lowest	52	53	53	54	52	59	51	46	51	52	48	49	50	51	54	52.7
Radius	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*2*	*0.1*

B-3. 512 Dimensions Manhattan (L1) Metric query

	2	3	6	12	23	46	91	182	363	725	1449	2897	5793	11586	23171	Avg.
Highest	58	59	59	59	62	71	68	69	67	65	69	71	66	68	62	64.9
Radius	*1*	*2*	*2*	*3*	*6*	*7*	*4*	*13*	*15*	*6*	*11*	*14*	*7*	*13*	*15*	*11.3*
Lowest	54	54	54	49	51	59	56	48	50	52	46	48	50	50	50	51.4
Radius	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0.1*

B-4. 512 Dimensions Euclidean (L2) Metric query

	2	3	6	12	23	46	91	182	363	725	1449	2897	5793	11586	23171	Avg.
Highest	56	60	60	60	64	72	70	68	68	65	69	67	63	60	59	64.0
Radius	*1*	*2*	*2*	*3*	*6*	*7*	*11*	*13*	*15*	*5*	*11*	*15*	*7*	*13*	*15*	*8.4*
Lowest	50	54	54	48	50	56	52	51	49	50	47	43	43	45	48	49.3
Radius	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0.1*

B-5. Accuracy % Increases from 512 to 768 Dimensions

Shard Size	Radius	Cosine	Manhattan	Euclidean	Best
6	0	33.96	35.19	31.48	Manhattan
6	1	47.27	50.91	44.64	Manhattan
6	2	31.75	42.37	38.33	Manhattan
6	3	34.92	42.37	36.67	Manhattan
6	4	34.92	42.37	38.33	Manhattan
6	5	34.92	42.37	38.33	Manhattan
6	6	34.92	42.37	38.33	Manhattan
6	7	34.92	42.37	38.33	Manhattan
6	8	34.92	42.37	38.33	Manhattan

Shard Size	Radius	Cosine	Manhattan	Euclidean	Best
6	9	34.92	42.37	38.33	Manhattan
6	10	34.92	42.37	38.33	Manhattan
6	11	34.92	42.37	38.33	Manhattan
6	12	34.92	42.37	38.33	Manhattan
6	13	34.92	42.37	38.33	Manhattan
6	14	34.92	42.37	38.33	Manhattan
6	15	34.92	42.37	38.33	Manhattan
23	0	36.54	47.06	48	Euclidean
23	1	26.67	44.64	36.84	Manhattan
23	2	33.33	46.55	37.29	Manhattan
23	3	29.03	46.55	37.29	Manhattan
23	4	29.69	44.07	38.33	Manhattan
23	5	29.69	39.34	31.75	Manhattan
23	6	29.69	37.1	29.69	Manhattan
23	7	28.12	37.1	29.69	Manhattan
23	8	28.12	37.1	32.81	Manhattan
23	9	28.12	37.1	32.81	Manhattan
23	10	28.12	37.1	32.81	Manhattan
23	11	28.12	37.1	32.81	Manhattan
23	12	28.12	37.1	34.37	Manhattan
23	13	28.12	37.1	35.94	Manhattan
23	14	28.12	37.1	35.94	Manhattan
23	15	28.12	37.1	35.94	Manhattan
91	0	37.25	28.57	46.15	Euclidean
91	1	25.42	22.95	35.09	Euclidean
91	2	25.86	24.19	28.81	Euclidean
91	3	29.82	29.51	35.59	Euclidean
91	4	17.46	17.65	28.12	Euclidean
91	5	19.35	19.4	30.16	Euclidean
91	6	17.46	19.4	28.12	Euclidean
91	7	17.46	19.4	26.56	Euclidean
91	8	18.46	20.59	22.39	Euclidean
91	9	18.46	20.59	22.39	Euclidean
91	10	16.67	22.39	18.84	Manhattan
91	11	14.93	20.59	17.14	Manhattan
91	12	17.91	23.53	18.57	Manhattan
91	13	17.91	23.53	18.57	Manhattan
91	14	17.91	23.53	18.57	Manhattan
91	15	19.4	25	20	Manhattan

Shard Size	Radius	Cosine	Manhattan	Euclidean	Best
363	0	49.02	42	44.9	Cosine
363	1	59.62	49.09	52.83	Cosine
363	2	46.55	39.34	40.68	Cosine
363	3	41.67	32.81	35.48	Cosine
363	4	49.15	32.81	36.51	Cosine
363	5	45	29.23	30.77	Cosine
363	6	41.67	29.23	32.31	Cosine
363	7	44.07	29.23	32.31	Cosine
363	8	44.07	31.25	32.31	Cosine
363	9	44.07	31.25	28.79	Cosine
363	10	44.07	33.87	32.31	Cosine
363	11	44.07	33.33	32.31	Cosine
363	12	44.07	35.48	32.31	Cosine
363	13	44.07	29.23	32.31	Cosine
363	14	46.55	29.23	34.37	Cosine
363	15	41.67	25.37	26.47	Cosine
1449	0	56.25	65.22	57.45	Manhattan
1449	1	54.9	48.15	54.9	Cosine
1449	2	54.72	40.68	49.09	Cosine
1449	3	48.21	34.92	42.37	Cosine
1449	4	48.21	45.76	50	Euclidean
1449	5	48.21	45.76	50	Euclidean
1449	6	41.67	42.62	37.1	Manhattan
1449	7	34.92	35.94	30.77	Manhattan
1449	8	34.37	37.5	32.31	Manhattan
1449	9	35.94	36.92	33.85	Manhattan
1449	10	30.77	29.85	29.85	Cosine
1449	11	26.87	26.09	26.09	Cosine
1449	12	28.79	26.09	26.09	Cosine
1449	13	30.77	26.09	26.09	Cosine
1449	14	30.77	26.09	26.09	Cosine
1449	15	29.23	24.64	23.19	Cosine
5793	0	28	20	53.49	Euclidean
5793	1	40	27.59	40.74	Euclidean
5793	2	46.43	37.93	50.94	Euclidean
5793	3	52.73	38.33	46.43	Cosine
5793	4	52.73	40	43.86	Cosine
5793	5	53.57	38.71	42.37	Cosine
5793	6	50.88	34.37	40	Cosine

Shard Size	Radius	Cosine	Manhattan	Euclidean	Best
5793	7	45.76	30.3	33.33	Cosine
5793	8	45.76	30.3	35.48	Cosine
5793	9	45	35.38	39.34	Cosine
5793	10	44.07	35.94	37.7	Cosine
5793	11	48.28	38.1	41.67	Cosine
5793	12	47.46	37.5	42.62	Cosine
5793	13	52.63	39.68	50	Cosine
5793	14	50	37.5	47.46	Cosine
5793	15	50	37.5	47.46	Cosine
23171	0	20	16	22.92	Euclidean
23171	1	18.18	18.52	16.67	Manhattan
23171	2	31.48	32.69	42.86	Euclidean
23171	3	34.55	33.33	42	Euclidean
23171	4	45.61	39.66	42.86	Cosine
23171	5	49.12	43.1	46.43	Cosine
23171	6	48.28	46.55	52.73	Euclidean
23171	7	48.28	44.07	52.73	Euclidean
23171	8	48.28	44.07	52.73	Euclidean
23171	9	48.28	41.67	50	Euclidean
23171	10	51.72	45	53.57	Euclidean
23171	11	50	42.62	49.12	Cosine
23171	12	50	43.33	49.12	Cosine
23171	13	50	43.33	49.12	Cosine
23171	14	50	40.98	46.55	Cosine
23171	15	45	40.32	44.07	Cosine

APPENDIX C: OPENSEARCH

C-1. Deployment Configuration

Opensearch Configuration	
Opensearch version	2.17
Nodes	1
Node compute type	t3.small.search
Availability Zones	1

C-2. Index and Query Configuration

```
index_body = {
    "settings": {
        "index.knn": True
    },
    "mappings": {
        "properties": {
            "file_path": {"type": "text"},
            "label": {"type": "text"},
            "embedding": {
                "type": "knn_vector",
                "dimension": 512,
                "method": {
                    "engine": "faiss",
                    "name": "hnsw"
                }
            }
        }
    }
}
```

```
for test_image_path in tqdm(
    test_image_paths,
    total=len(test_image_paths),
    desc="Searching from OpenSearch Index"
):
    # search
    test_embedding = get_embeddings(test_image_path)
    start_time = time.time()
    search_body = {
        "query": {
            "knn": {
                "embedding": {
                    "vector": test_embedding,
                    "k": 1
                }
            }
        }
    }

    response = client.search(index=index_name, body=search_body)
    end_time = time.time()
```

APPENDIX D: COST CALCULATIONS

D-1. Embeddings Lake

Assumptions	Animals Dataset							
512 Dimensional Embedding	5310 images -> 5310 embeddings							
In memory using float32, each float is 4 bytes	5310 * 2KB = 10620 KBs							
512 * 4 bytes = 2048 bytes or 2 KB per embedding	10620 KBs = 10.38 MBs							
API Gateway								
$3.50 for first 333M requests/month								
Hash Lambda*								
average 120ms duration								
memory set to 256MB	Dataset	API Gateway	Hash Lambda	Fifo Queue	Add Lambda	DLQ	S3	total
	Animals Dataset	$ 3.50	$ 0.00	$ -	$ 0.09	$ -	$ 0.03	$ 3.62
Fifo Queue	(5310 embeddings)							
First 1M request/month are free								
	1 Million Embeddings	$ 3.50	$ 0.07	$ -	$ 10.00	$ -	5.05	$ 18.62
Add Lambda*								
average 2000ms duration								
memory set to 512 MB								
	*Lambda estimate calculation provided by https://dashbird.io/lambda-cost-calculator/							
S3 Storage								
$.023 / GB first 50TB								
$0.000005 / PUT								

D-2. SQS Source Pricing

aws

About AWS | Contact Us | Support | English | My Account | Sign In to the Co

Amazon Q | Products | Solutions | Pricing | Documentation | Learn | Partner Network | AWS Marketplace | Customer Enablement | Events | Explore More

Amazon SQS | Overview | Features | Pricing | Getting Started | Resources | FAQs

How are Amazon SQS requests priced?

The pricing for monthly requests is shown below:

Region:

US East (Ohio)

	Standard Queues (per Million requests)	FIFO Queues (per Million requests)
First 1 Million Requests/Month	Free	Free
From 1 Million to 100 Billion Requests/Month	$0.40	$0.50
From 100 Billion to 200 Billion Requests/Month	$0.30	$0.40
Over 200 Billion Requests/Month	$0.24	$0.35

D-3. API Gateway Source Pricing

aws

About AWS | Contact Us | Support | English | My Account | Sign In

Amazon Q | Products | Solutions | Pricing | Documentation | Learn | Partner Network | AWS Marketplace | Customer Enablement | Events | Explore More

Amazon API Gateway | Overview | Features | Pricing | Getting Started | Resources | FAQs | Partners

US East (Ohio)

PAGE CONTENT

- Free Tier
- HTTP APIs
- REST APIs
- WebSocket APIs
- Additional Charges
- Pricing Examples
- Pricing Resources

API Calls

Number of Requests (per month)	Price (per million)
First 333 million	$3.50
Next 667 million	$2.80
Next 19 billion	$2.38
Over 20 billion	$1.51

Caching

D-4. S3 Source Pricing

aws pricing calculator

Feedback | Language: English | Con

▼ Show calculations

Unit conversions

S3 Standard Average Object Size: 2 KB x 9.5367432e-7 GB in a KB = 0.00000190734864 GB

Pricing calculations

2 GB per month / 0.00000190734864 GB average item size = 1,048,575.996 unrounded number of objects
Round up by 1 (1048575.9960) = 1048576 number of objects
Tiered price for: 2 GB
2 GB x 0.023 USD = 0.05 USD
Total tier cost = 0.046 USD (S3 Standard storage cost)
1,000,000 PUT requests for S3 Standard Storage x 0.000005 USD per request = 5.00 USD (S3 Standard PUT requests cost)
0.046 USD + 5.00 USD = 5.05 USD (Total S3 Standard Storage, data requests, S3 select cost)
S3 Standard cost (monthly): 5.05 USD

1,048,576 number of objects x 0.000005 USD = 5.24288 USD (Cost for PUT, COPY, POST requests for initial data)
S3 Standard cost (upfront): 5.24 USD

D-5. Weaviate Pricing with 512 dimensions, 1M vectors, Performance storage

D-6. Weaviate Pricing with 512 dimensions, 5K vectors, Performance storage

D-7. Weaviate Pricing with 512 dimensions, 1M vectors, Compression storage

D-8. Weaviate Pricing with 512 dimensions, 5K vectors, Compression storage

D-9. Pinecone Pricing

Pinecone | Product | Docs | Customers | Resources | Pricing | Contact | Log in | Sign up

Pinecone runs on fully managed infrastructure that scales with you. Start building today with product and support plans tailored to your needs.

AWS | GCP | Azure

Starter
For trying out and for small applications.
Start for Free
Free
View included usage
- Pinecone Serverless
- Pinecone Inference
- Pinecone Assistant
- Console Metrics
- Community Support

Standard POPULAR
For production applications at any scale.
Get Started
from $25 / month
Includes $15 / mo usage credits
- Unlimited Serverless, Inference, and Assistant usage
- Choose your cloud and region
- Import from object storage
- Multiple projects and users
- User and API Key RBAC
- Backup and Restore
- Prometheus metrics
- Includes Free support
- Response SLAs available via Developer or Pro support add-on

Enterprise
For mission-critical production applications.
Get Started | Request Trial
from $500 / month
Includes $150 / mo usage credits
- Everything in Standard
- 99.95% Uptime SLA
- SAML SSO
- Private Networking
- Customer Managed Encryption Keys
- Audit Logs
- Service Accounts
- Admin APIs
- HIPAA Compliance
- Pro support included

Compare Plans

D-10. AWS Opensearch Serverless Pricing for 1M embeddings

aws pricing calculator | Feedback | Language: English | Contact Sales | Create an AWS Acc

Choose a location type Info: Region
Choose a Region: US East (Ohio)

Configure Amazon OpenSearch Service domain | Configure Amazon OpenSearch Serverless | Configure Amazon OpenSearch Ingestion

OpenSearch serverless

How many Indexing OCUs? 1

How many Search and Query OCUs? 1

How big is the index data? 2 | Unit: GB

One OCU comprises 6 GB of RAM, corresponding vCPU, GP3 storage (used to provide fast access to the most frequently accessed data), and data transfer to Amazon Simple Storage Service (S3).

Show calculations

1 OCUs x 730 hours in a month x 0.24 USD = 175.20 USD (Indexing cost)
1 OCUs x 730 hours in a month x 0.24 USD = 175.20 USD (Search and query cost)
2 GB x 0.024 USD = 0.05 USD (Managed storage cost)
175.20 USD + 175.20 USD + 0.05 USD = 350.45 USD (Amazon OpenSearch Serverless cost)
Amazon OpenSearch Serverless cost (monthly): 350.45 USD

Total Upfront cost: 0.00 USD
Total Monthly cost: 350.45 USD
Show Details | Cancel | Save and view summary | Save and add s

D-11. AWS Opensearch Serverless Pricing for Animals Dataset

aws pricing calculator

Feedback

Lang

Choose a location type Info

Region

Choose a Region

US East (Ohio)

Configure Amazon OpenSearch Service domain

Configure Amazon OpenSearch Serverless

OpenSearch serverless

How many Indexing OCUs?

1

How many Search and Query OCUs?

1

How big is the index data?

1

Unit

GB

One OCU comprises 6 GB of RAM, corresponding vCPU, GP3 storage (used to provide fast access to the most frequently accessed

Show calculations

1 OCUs x 730 hours in a month x 0.24 USD = 175.20 USD (Indexing cost)
1 OCUs x 730 hours in a month x 0.24 USD = 175.20 USD (Search and query cost)
1 GB x 0.024 USD = 0.02 USD (Managed storage cost)
175.20 USD + 175.20 USD + 0.02 USD = 350.42 USD (Amazon OpenSearch Serverless cost)
Amazon OpenSearch Serverless cost (monthly): 350.42 USD

Total Upfront cost: 0.00 USD
Total Monthly cost: 350.42 USD

Show Details

D-12. AWS Opensearch Pricing for 1M embeddings

Selected Instance:
t3.small.search
vCPU: 2
Memory (GiB): 2

Utilization (On-Demand only)
Data Instances will be always at 100% utilization dedicated for your Amazon Opensearch Service domain, to stop incurring charges you must Delete the domain.
Value: 100
Unit: %Utilized/Month

Instance Node Type: General purpose

Pricing model: OnDemand

Storage Type: EBS Only

▼ Show calculations
1 instance(s) x 0.036 USD hourly x (100 / 100 Utilized/Month) x 730 hours in a month = 26.2800 USD (Amazon OpenSearch Service data ins
Amazon OpenSearch Service data instance cost (monthly): 26.28 USD

Amazon OpenSearch Service data instance cost (upfront): 0.00 USD

Total Upfront cost: 0.00 USD
Total Monthly cost: 26.55 USD
Show Details ▼

Storage Info

Number of instances
Enter the total number of data instances using EBS
1

Storage for each Amazon OpenSearch Service instance: General Purpose SSD (gp2)

Storage amount: 2
Unit: GB

▼ Show calculations
2 GB x 0.135 USD x 1 instances = 0.27 USD (EBS Storage Cost)
Amazon OpenSearch Service storage cost (monthly): 0.27 USD

UltraWarm Instances (Optional)

Number of nodes
minimum of 2 nodes if enabling UltraWarm
0

Total Upfront cost: 0.00 USD
Total Monthly cost: 26.55 USD
Show Details ▼

D-13. AWS Opensearch Pricing for Animals Dataset

Selected Instance:
t2.small.search
vCPU: 1
Memory (GiB): 2

Utilization (On-Demand only)
Data Instances will be always at 100% utilization dedicated for your Amazon Opensearch Service domain, to stop incurring charges you must Delete the domain.
Value: 100
Unit: %Utilized/Month

Instance Node Type: General purpose

Pricing model: OnDemand

Storage Type: EBS Only

▼ Show calculations
1 instance(s) x 0.036 USD hourly x (100 / 100 Utilized/Month) x 730 hours in a month = 26.2800 USD (Amazon OpenSearch Service data ins
Amazon OpenSearch Service data instance cost (monthly): 26.28 USD

Amazon OpenSearch Service data instance cost (upfront): 0.00 USD

Total Upfront cost: 0.00 USD
Total Monthly cost: 26.28 USD
Show Details ▼

Storage Info

Number of instances
Enter the total number of data instances using EBS
1

Storage for each Amazon OpenSearch Service instance: General Purpose SSD (gp2)

Storage amount: 11
Unit: MB

▼ Show calculations
Unit conversions
Storage amount: 11 MB x 0.0009765625 GB in a MB = 0.0107421875 GB
Pricing calculations
0.0107421875 GB x 0.135 USD x 1 instances = 0.00 USD (EBS Storage Cost)
Amazon OpenSearch Service storage cost (monthly): 0.00 USD

UltraWarm Instances (Optional)

Number of nodes
minimum of 2 nodes if enabling UltraWarm
0

Total Upfront cost: 0.00 USD
Total Monthly cost: 26.28 USD
Show Details ▼