Project Report: Visual Similarity Search in Haystack

Team Members

Sunkari Sai Satvik - 2021101117
Rushil Kaul - 2021101063

Table of Contents:

---

1. Statement of the Problem

The project introduces a visual similarity search feature to Haystack, a large-scale image storage system. This feature will arrange and allow users to find images based on visual content, leveraging similarity measures, rather than relying solely on metadata based searching. By implementing this, we aim to provide a more intuitive and content-driven search functionality for image storage systems.

2. Scope of the Project

1. **Haystack-Style Storage**

    - A basic implementation of Haystack's separation of metadata and raw image data for optimized retrieval. Includes implementation of clients, a webserver, directory, cache, stores, etc.
2. **Visual Similarity Search**
    - Arranges images in logical volumes on the basis of similarity scores.
    - Develop a feature that retrieves a ranked list of visually similar images based on a query photo.

3. Haystack - File System Implementation

Haystack, developed by Facebook, is an image storage system designed to optimize the storage and retrieval of large-scale image datasets. It separates metadata from raw image data, allowing faster access and reduced overhead. Key concepts include:

- Needle Storage: Images (needles) are stored in large aggregated files (haystacks).
- Metadata Management: Metadata like file names, offsets, and sizes are stored separately to streamline retrieval.
- Read and Write Efficiency: Logical and physical volumes ensure scalable storage.

For this project, we mimicked Haystack's structure by implementing:

1. Separation of Metadata and Raw Data:
    - Metadata stored in directory for quick lookup.
    - Raw image data stored in separate physical volumes.
2. Logical-Physical Mapping:
    - Logical volumes mapped to physical volumes.
    - Logical volumes marked as read-only once full.
3. Needle Structure:

    - Each photo is stored as a “needle” with metadata fields (e.g., ID, flags) and data (raw image).
4. Caching System:
    - A non-dynamic, hash-based caching mechanism for frequently accessed images.
5. Web Server:
    - The web server acts as the backbone of the system for request handling, managing communication and ensuring seamless interaction between the Client and various components of the Haystack-inspired architecture.

The core operations in Haystack’s standard implementation—**read, write, and delete**—are designed to handle large-scale image datasets efficiently by leveraging logical-physical volume mappings, caching mechanisms, and the separation of metadata from raw image data. The web server coordinates requests between the client and the underlying components, ensuring seamless operation.

1. Write Operation

The write operation stores an image by separating metadata and raw image content while adhering to volume capacity limits.

Steps:

1. Client to Web Server Request:
    - The client sends a write request to the web server, including the image data.
2. Web Server to Directory Communication:
    - The web server forwards the request to the directory, which identifies a suitable logical volume (marked as read-write and not full).
3. Web Server to Store Communication:
    - The raw image data is sent to the appropriate physical volume mapped to the logical volume. The store writes the data to a free slot, and the metadata (photo ID, offset, size) is updated in the directory.

Key Features:

- Logical volumes ensure efficient storage and prevent fragmentation.
- Metadata updates enable quick future retrievals.

2. Read Operation

The read operation retrieves an image using its photo ID, leveraging metadata lookups and caching for efficiency.

Steps:

1. Client to Web Server Request:
    - The client sends a read request with the photo ID to the web server.
2. Web Server to Directory Query:
    - The web server queries the directory for metadata associated with the photo ID, including logical and physical volume mappings, machine, cache.
3. Cache Check:
    - The web server checks if the image exists in the cache. If found, the image is directly returned to the client rather than reading from store.
4. Web Server to Store Retrieval:
    - If the image is not cached, the cache forwards the request to the store. The store retrieves the image using the metadata (physical volume and offset) and sends it back to the client.
    - The retrieved image is added to the cache for faster future access.

Key Features:

- Metadata-driven lookups minimize access times.
- The cache improves performance for frequently requested images.

3. Delete Operation

The delete operation flags an image as deleted.

Steps:

1. Client to Web Server Request:
    - The client sends a delete request with the photo ID to the web server.
2. Web Server to Directory Update:
    - The web server communicates with the directory to locate the logical and physical volumes associated with the photo ID.
    - The directory updates the metadata, flagging the photo ID as “deleted.”
3. Web Server to Store Update:
    - The web server relays the deletion request to the store. The store updates the corresponding needle’s metadata, marking it as deleted.
4. Cache Cleanup:
    - If the image exists in the cache, it is removed to ensure consistency.

Key Features:

- Logical deletion avoids immediate performance overhead.
- Consistent updates across the directory, cache, and store ensure system integrity.

4. Extension: Visual Similarity Search in Haystack

The extension introduced to Haystack enables visual similarity search, allowing users to retrieve images based on their visual content rather than just metadata. This is achieved by extracting visual features from images, indexing them, and performing similarity searches using Faiss. The system efficiently stores the features and retrieves visually similar images based on a query's feature vector.

Haystack was designed for efficient metadata-based image retrieval. However, it lacked a mechanism for visual similarity search, which allows users to search for images that are visually similar to a given query. To solve this, we extended Haystack with a deep learning-based feature extraction pipeline and integrated Faiss for fast similarity searches. The solution maintains Haystack's scalability by leveraging pre-trained models and efficient indexing techniques.

Implementation Steps

1. Feature Extraction using Pre-Trained Models

- Problem: For visual similarity search, images need to be represented as feature vectors that capture their visual content.
- Solution:
    - We used a pretrained MobileNetV2 model, for feature extraction. This model was chosen due to its efficiency and lower computational overhead.
    - We applied transfer learning to reduce the dimensionality of the model's output, generating compact 64-dimensional feature vectors.
    - These vectors represent the images' visual characteristics and are stored as metadata in the Haystack directory along with their photo IDs.
- Outcome: The compact feature embeddings allow us to store and index visual information efficiently, avoiding significant storage overhead.

2. Faiss for Efficient Similarity Search

- Problem: Finding similar images requires an efficient algorithm to compute the distances between feature vectors across a large dataset.
- Solution:
    - The Faiss library was chosen to handle the indexing and searching of the visual feature vectors.
    - Faiss uses the L2 (Euclidean) distance metric to find the closest vectors to a given query feature.

        - We implemented a search mechanism where the query image's feature vector is compared to those stored in Faiss, retrieving the most similar images based on distance.
- Outcome: Faiss significantly accelerates the similarity search process, providing fast and scalable retrieval, even with a large dataset of image features.

3. Load Balancing

In a large-scale system, search requests can be unevenly distributed, leading to bottlenecks where some servers or nodes bear more load than others. To address this, load balancing can be implemented across multiple nodes or processes that handle the Faiss index.

By distributing the similarity search load evenly, we can prevent any one server from being overwhelmed, ensuring faster query responses and better overall system performance. This can be done by partitioning the index into smaller, manageable chunks, allowing each node to process its share of the queries independently. This approach is particularly effective in distributed systems where multiple servers host parts of the Faiss index.

4. Query Pipeline

- Implementation:
    - When a user submits a query image, the system extracts its feature vector using the same pre-trained model.
    - The query feature is passed to the Faiss index, which returns the photo ID of the most similar image based on the feature vector distance.
    - The corresponding image is then retrieved from storage using the Haystack read operation, ensuring that metadata and image content are correctly mapped.

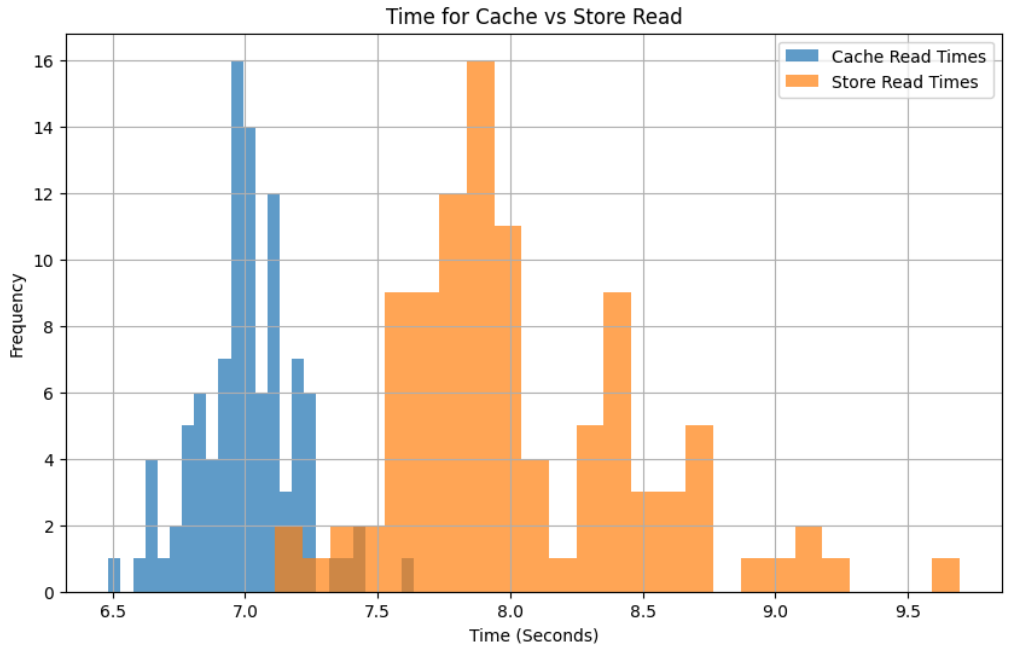
5. Assumptions

1. Feature Compactness: We assumed that 64-dimensional embeddings would be sufficient for distinguishing visual content and enabling effective similarity search without consuming excessive storage space.
2. Relatively Compact Dataset Size and Number of Classes: The initial dataset used for testing (Tiny ImageNet) was assumed to be moderate in size due to computational constraints, allowing for relatively fast feature extraction and indexings.

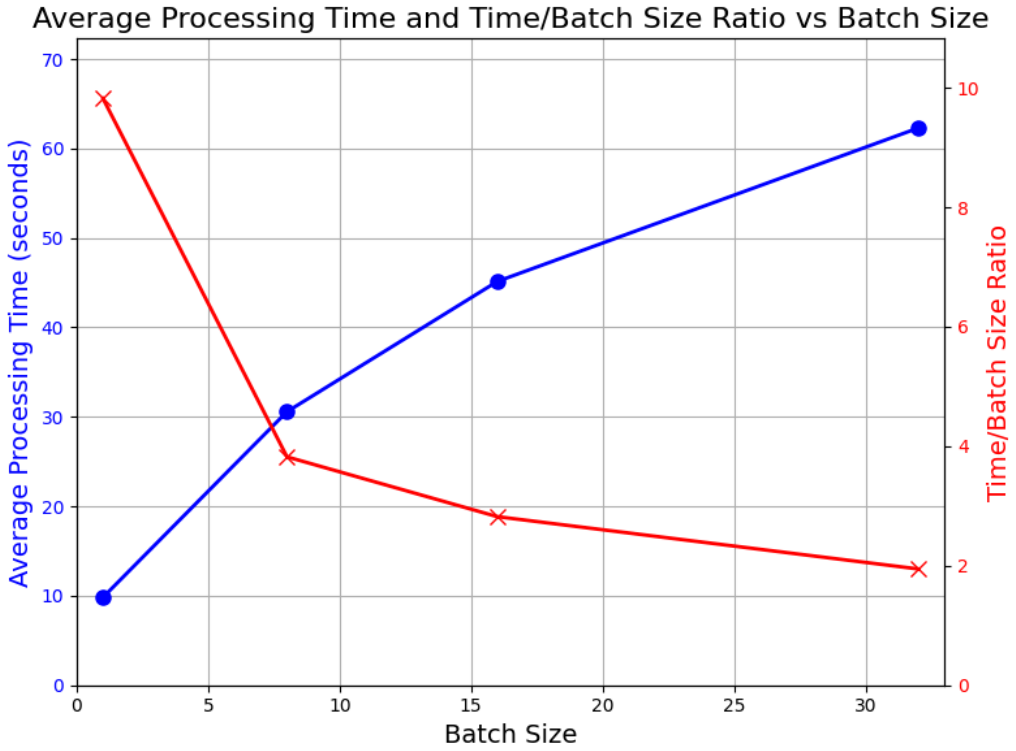
6. Challenges, Ideas Attempted

1. Choice of MobileNetV2 Due to Computational Constraints:
    - Idea: Initially, we considered using a larger or more powerful model like ResNet or EfficientNet for feature extraction, as they are known for their high accuracy in capturing fine-grained visual features.
    - Issue: Both ResNet and EfficientNet require significantly more computational resources, including memory and processing power, which proved to be a bottleneck. Running these models for feature extraction on a large dataset resulted in slow processing times and high memory consumption, making real-time feature extraction more difficult for the scale of our application.
    - Resolution: To address this, we switched to MobileNetV2, which is a more lightweight model designed specifically for efficiency, especially in resource-constrained environments. It provides a good balance between speed and accuracy, making it more suitable for the project's computational constraints. While it sacrifices some fine-grained accuracy compared to ResNet, the trade-off was necessary for the performance and scalability required in the system.
2. Dynamic Logical Volume Assignment in Haystack Directory:
    - Idea: We explored dynamically assigning logical volumes in the Haystack directory, where the logical volume could adjust its size and properties depending on the volume of incoming image data. This would allow better load balancing and efficient storage allocation based on actual usage patterns.
    - Issue: Implementing dynamic logical volume resizing caused complications in data consistency and management. For instance, images assigned to a dynamic volume would need to be frequently reallocated if the volume size changed, potentially leading to data migration overhead and performance degradation. Additionally, the need to constantly monitor and adjust the volume allocation led to increased system complexity and maintenance challenges.
    - Resolution: Instead of dynamic resizing, we reverted to using fixed logical volumes with predefined sizes. The fixed allocation allowed for simpler management and more predictable performance. We also implemented a function to mark volumes as read-only when full, which prevented unnecessary reallocations and kept data consistency intact.

5. Benchmarking and Graphs

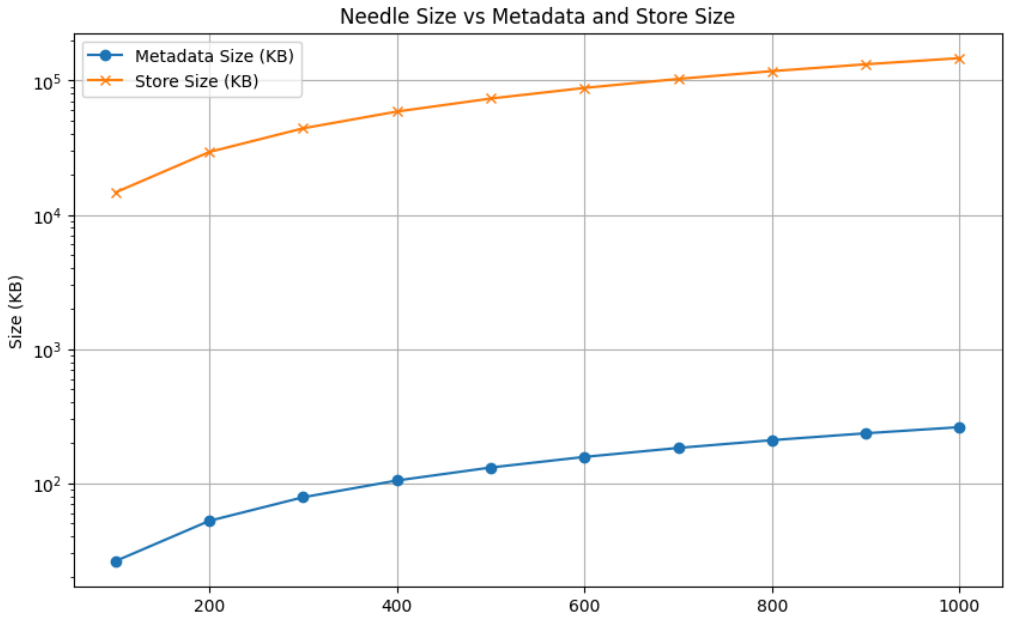


Time taken for each cache vs store read, expressed as histogram plot. On average, cache reads saved about 15% time in comparison to store reads on lookup, for the first 100 reads.

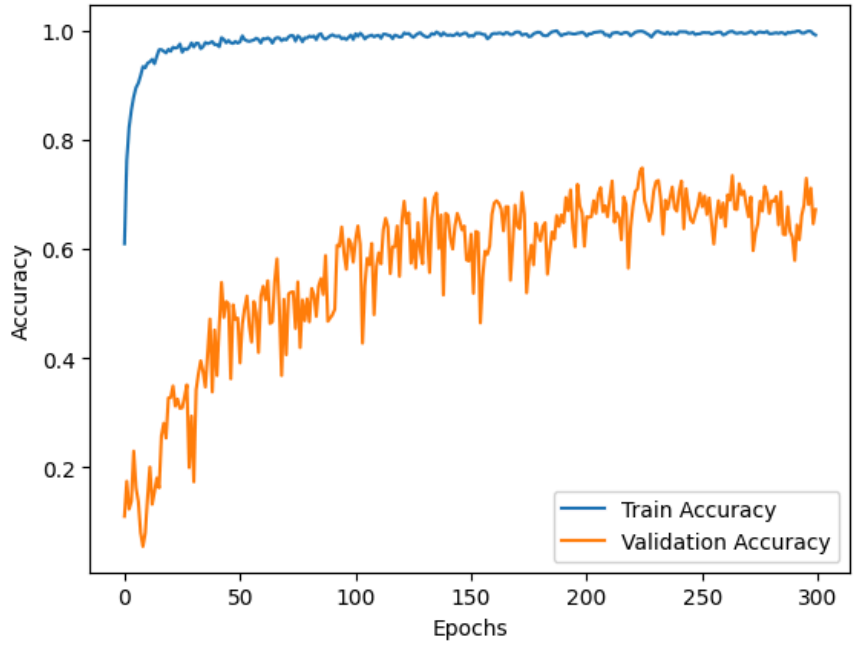


We applied an optimization from the primary Haystack source paper (in references) that lowers per-image write time by uploading images in batches, similar to how

photos are generally uploaded in albums. This approach reduces per-image write time by minimizing messaging overhead and increases the potential for distributed computing.



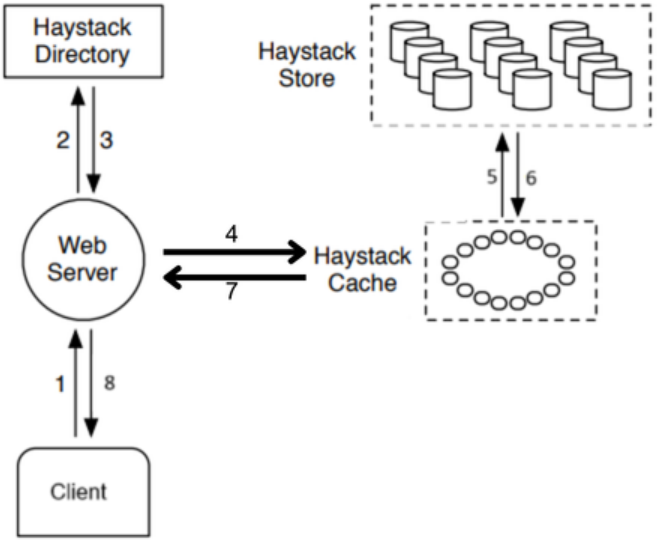
Metadata stored in a directory offers significant storage space reduction compared to storing feature embeddings in the store. Even with the inclusion of feature embeddings in the directory, the reduction in storage size for metadata remains substantial, with a difference on the order of 1000 (log scale), as the number of images increases (Y-axis).



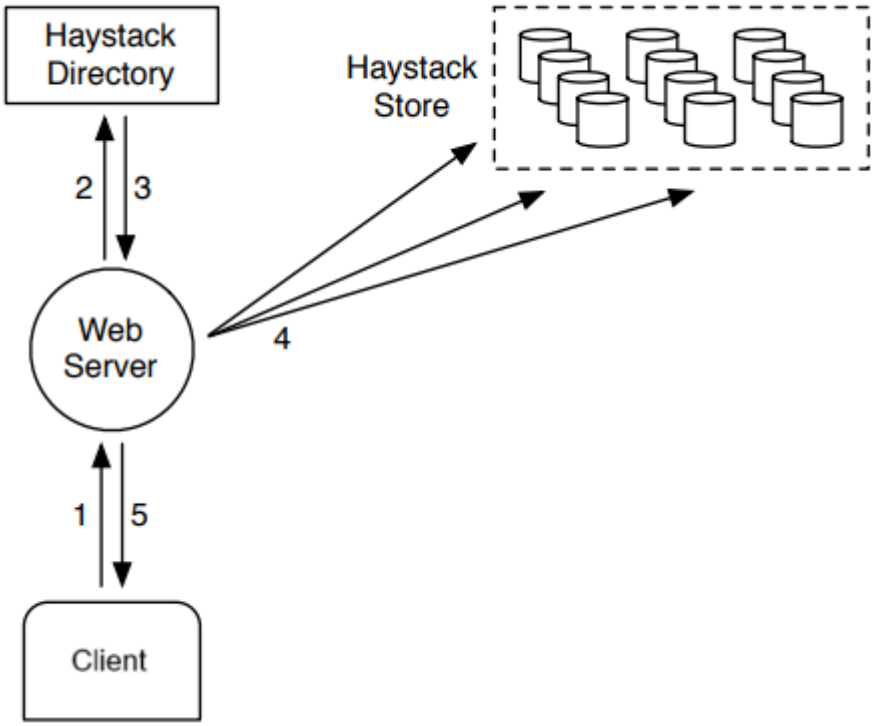
MobileNetV2 Train and Validation Accuracy for 20 classes - Model with best validation accuracy was stored.

6. Diagrams

Flowcharts for Operations:



Read, Delete Operations



Write Operation

7. Conclusion and Possible Future Work

The extension of Haystack with visual similarity search has significantly improved the system's ability to retrieve images based on visual content rather than relying on metadata alone. By leveraging pre-trained MobileNetV2 models for feature extraction and Faiss for indexing and search, we tried to achieve a balance between accuracy and performance. The system scales well with moderately sized datasets, but performance could be further optimized for larger datasets by exploring distributed indexing solutions or more advanced Faiss indexing methods.

Possible Future Work:

1. Scalability Improvements:
    - Test other approximate nearest neighbor search algorithms (e.g., IVF, HNSW) for faster search times at scale.
2. Advanced Feature Extraction:
    - Experiment with other pre-trained models like EfficientNet or ResNet to compare performance improvements in feature extraction accuracy.
    - Explore fine-tuning other pre-trained models to improve the quality of feature vectors.
3. Integration with Distributed Storage Systems:
    - Extend the system to distribute image storage across multiple cloud-based machines dynamically, allowing the use of a cloud-based infrastructure for large-scale data handling.
4. Mobile and Web Application:
    - Build a web or mobile interface to allow users to perform visual similarity searches on the fly. This could be useful for applications like image retrieval in digital libraries or media management systems.

8. Work Split

- Rushil Kaul:
    - Implemented the core functionality for feature extraction using MobileNetV2 and tested different models for performance.
    - Implemented Haystack Store functionality.
    - Worked on client terminal interface to run commands.
- Sunkari Sai Satvik:
    - Focused on the Haystack architecture and integrating the similarity search into the existing system.
    - Managed cache and directory structures, ensuring efficient data storage and retrieval.

- Set up the web server and communication between the client and server for smooth requests.

9. References and Links

1. **"Finding a Needle in Haystack: Facebook's Photo Storage"**
    - This paper outlines Haystack's efficient image storage and retrieval system. Understanding its core concepts is crucial for replicating a similar structure.
    - Finding a needle in Haystack: Facebook's photo storage
2. **"Faiss: A Library for Efficient Similarity Search"**
    - This library is pivotal for practical implementations of similarity search using pre-built indexing and search capabilities.
    - Source: GitHub
3. **Documentation and Tutorials**
    - We utilize readily available resources for pre-trained models (e.g., TensorFlow, PyTorch) and tutorials on Faiss integration.