LuSH: A Generic High-Dimensional Index Framework

Zhou Yu, Jian Shao, and Fei Wu

College of Computer Science, Zhejiang University,
Hangzhou, China, 310027
{yuz,jshao,wufei}@zju.edu.cn

Abstract. Fast similarity retrieval for high-dimensional unstructured data is becoming significantly important. In high-dimensional space, traditional tree-based index is incompetent comparing with hashing methods. As a state-of-the-art hashing approach, Spectral Hashing (SH) aims at designing compact binary codes for high-dimensional vectors so that the similarity structure of original vector space can be preserved in the code space. We propose a generic high-dimensional index framework named LuSH in this paper, which means Lucene based SH. It uses SH as high-dimensional index and Lucene, the well-known open source inverted index, as underlying index file. To speedup retrieval efficiency, two improvement strategies are proposed. Experiments on large scale datasets containing up to 10 million data show significant performance of our LuSH framework.

Keywords: High-dimensional Index, Spectral Hashing, Lucene.

1 Introduction

With the rapid development of Internet, increasing amounts of unstructured data (e.g. image, video, music) are produced at every moment. How to well organize these large scale data and execute efficient retrieval is an urgent research issue. Unlike structured data, it's hard to compare similarity between two unstructured data. Taking image data as an example, the common method is extracting typical visual features (e.g. color histogram, Tamura texture[1], CEDD[2], SIFT[3], etc.) from data and then similarity of original data is converted to distance comparing in feature space (e.g., Euclidean distance). Unfortunately, to maintain the characteristics of original data, the extracted features are usually high-dimensional vectors. Hence, building an index structure to provide efficient retrieval in high-dimensional feature space is of crucial importance.

The trending method of high-dimensional index is Spectral Hashing (SH)[13]. It's an improvement of Semantic Hashing[4], which aims at designing compact binary codes for high-dimensional vectors so that the similarity structure of original vectors space can be preserved in the code space, then the similarity retrieval is executed based on these codes. In this way, the feature-extracted unstructured data object can be represented as a "small" code. Assume all data are hashed to binary codes (hamming code), given a query code of same metric, the Near Neighbors (NN) retrieval scenario is as follows: linear scan all codes in the database, sort hamming distance to query code and return the codes within a small distance *r*. Since the codes are all in low-dimensional hamming space, the retrieval is relatively fast.

Although the SH algorithm is a state-of-the-art index in high-dimension, there are still some problems to apply it in a system-level application. Firstly, linear scan based NN retrieval is infeasible when the scale of database reaches tens of millions and the concurrent access user number reaches hundreds or more. Secondly, how to fuse SH with underlying index storage should be taken into account, which involves cache, disk access, and index storage optimization.

In this paper, we propose an index framework based on SH. Since the underlying index storage is complex, the open source inverted index, Lucene[5] is applied to fuse with SH, taking over the management of index storage. To speedup retrieval, multi-probe retrieval and ID mapping strategies are proposed. We name our framework "LuSH", which means a Lucene based SH index.

The rest of the paper is organized as follows. In Section 2, we review the related work on semantic hashing related algorithm and Lucene index. In Section 3, we present our LuSH index framework and describe our improved algorithm in retrieval. In Section 4, we demonstrate the performance of our framework on large scale datasets. In Section 5, we conclude the paper and end up with future work.

2 Related Work

2.1 Semantic Hashing

Traditional tree-based multi-dimensional index such as R-Tree[6] and KD-Tree[7] are not made for high-dimensional data. They are best fit for the data less than ten dimensions. As the dimension increases, the efficiency of these algorithms decreases rapidly. This is called the "curse of dimensionality"[8]. It has been proved that if the scale of database N and dimension d don't satisfy $N \gg 2^d$, the efficiency of the index algorithm is worse than linear scan in the entire database.

Semantic Hashing[1] is proposed in this context. It aims at embedding high-dimensional data into a low-dimensional space, while remaining the semantic similarity structure of data in the original space. Semantic hashing is a concept, many algorithms are proposed according to this.

One of the most well-known hashing methods is Locality Sensitive Hashing (LSH)[9,10]. Hash code is generated by multiple projections to random hyperplanes. The projection is locality sensitive that if two high-dimensional points is similar, they have a large probability to share the same code. Due to random projection, LSH suffers from redundancy of hash code length and hash tables.

Some machine learning algorithm is proposed to make up the drawback of LSH. Restrict Boltzman Machine (RBM)[11], as a deep learning model algorithm, provides a new thought of hashing, and gives better performance compared with LSH. A simpler machine learning method Boosting SSC[12], uses adaBoost to train a set of learners and gets final hash code from these composite weak learners. These machine learning hashing algorithms achieve a more precise result than the random projected LSH, however, high complexity restricts their practical applicability.

Spectral Hashing[13] is proposed by Weiss in 2009, which has been demonstrated to outperform other hashing methods above.

2.2 Spectral Hashing

Spectral Hashing can be seen as an extension of Spectral Cluster[19]. The concrete formulation is as follows:

$$\min \sum_{i,j} W_{ij} \left|\left|y_i - y_j\right|\right|^2$$

$$\textbf{s.t.} \quad y_i \in \{-1,1\}^K, \quad \sum_i y_i = 0, \quad \frac{1}{n}\sum_i yiyi^T = \mathbf{I} \tag{1}$$

Where W_{ij} is the similarity matrix of sample i and j. y_i denotes the binary hashing code of sample i and K the number of code length. This formulation carry through the motivation of semantic hashing, similarity relationship of samples in the original high-dimensional space is preserved in low-dimensional hamming space.

Solution of equation 1 is NP hard. Spectral relaxing by removing the constraint $y_i \in \{-1, 1\}^K$, equation 1 is transformed to a spectral analysis problem, whose solutions are simply extracting k eigenvectors from the Laplacian matrix of W with minimal eigenvalues.

After the equation is solved, the following procedure is to learn a hash function with the solved eigenvectors and eigenvalues for novel input queries (out of sample problem), whose solutions are eigenfunctions of the weighted Laplace-Beltrami operators defined on manifolds[14,15,16]. Specifically for a case of uniform distribution on $[a, b]$, the eigenfunction of 1D-Laplacian is as follows.

$$\Phi_k(x) = \sin\left(\frac{\pi}{2} + \frac{k\pi}{b-a}x\right) \tag{2}$$

The final hash function $f(x) = sgn(\Phi_1(x)\Phi_2(x), \dots \Phi_K(x))$

2.3 Lucene Index

Lucene is an open source inverted index supported by the Apache Software Foundation. Due to its high efficiency, it's widely used in full text search engines, database management systems, file system, etc.

The purpose of inverted index is to allow efficient full text retrieval. Traditional index used in database system, e.g. B-Tree[17], does not support the keywords based full text retrieval, thus, inverted index is proposed. It not only stores the document itself, but also the inverted table mapping from keyword to document. In Lucene, for storage concern, the document and inverted table are associated using an internal id (see Fig.1).

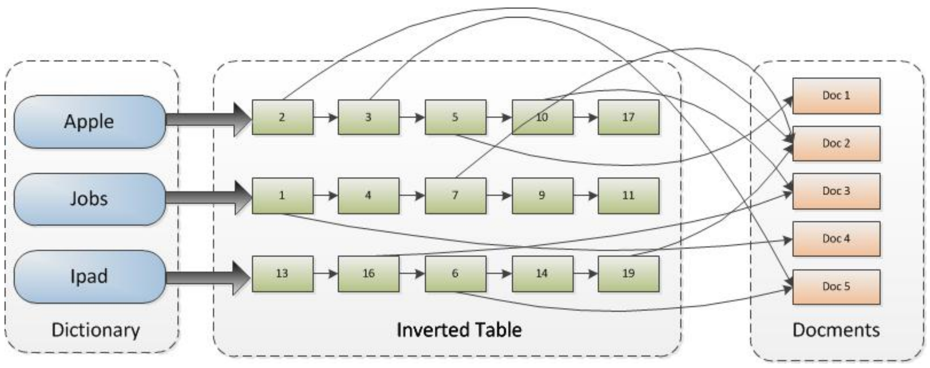

Fig. 1. Lucene index file

3 LuSH Framework

In this section, we describe the overall architecture of our LuSH. For better explanation, we use Content based Image Retrieval System (CBIR) as a typical application (see Fig.3). The framework can be partition into three modules: Training, Index and Retrieval (shown in Fig.3 with brown dot lines), we detail the three modules individually in the following subsections.

3.1 Training Module

The purpose of the module is to learn a hash function using a training set. We prepare a portion of samples (assume n samples) which represent the entire database. Generally, random sampling strategy is used. Then, we extract feature of these samples to get a matrix $X \in R^{n \times d}$ and solve the objective function in section 2.2, and get K minimal eigenvalues and their corresponding eigenvectors. The hash function can be seen as the projections of an input data to the K eigenvectors and binarize the each bit at a threshold.

In consideration of storage optimization, the code length K we choose is the multiple of 8 (size of a byte). In practice, 32 bits (size of an integer) or 64 bits (size of a long integer) is preferred.

3.2 Indexing Module

When the hash function $f(x)$ is obtained, the next step is to use this function to encode all the data objects in the entire database. The same visual feature as we used in the training step is extracted for each item in the dataset. Assume the database has N data objects, every object has its unique identifier (UID), and we gain N hamming codes of which the code length is K. We then add the N hamming codes and their corresponding UID into index.

We design a generalized index interface for our index structure, which is for future extension. Lucene is an implementation of the interface just like a plug-in. The generic index interface we design contains functions as follows:

- **Initialize:** load the index file and trained hash function
- **Add:** input the document containing hash code and corresponding UID, return a boolean value of whether it's successfully added into index
- **Delete:** input the document containing hash code and corresponding UID, return a boolean value of whether it's successfully deleted from index
- **Update:** input the old and new documents, execute delete operation for the old document at first, then add the new document
- **Commit:** commit the operation up to now and makes the updates to index retrievable

In Lucene, document is the basic index unit. We design a document consist of an hash code and its UID. The original data and feature vectors are optionally stored depending on application requirements. When executing an add (delete or update) operation to the index file, each hash code is a keyword in the dictionary and internal ids are appended to it if these documents share the same hash code. The other information is stored in the document structure with the internal id as its entry.

3.3 Retrieval Module

After the indexing procedure for the whole database is completed, we can execute efficient NN retrieval using the index structure. Similar with the index building procedure, for any query object, we first extract features and then use the hash function again to obtain a hash code. To achieve efficient NN retrieval using the hash code, two strategies are proposed.

3.3.1 Multi-probe Retrieval

When we try to execute NN retrieval for a given hash code, linear scanning in the database is used in traditional SH. It's not efficient enough when database is large scale. Since in most applications, the NN we expect to fetch is a small portion of the entire database (e.g. hamming distance within 2), it's easy to enumerate all the possible NN we need, which we called a probing sequence. Then we use the probing sequence to execute exact query (find the same hash code) in the database. Each exact code retrieval problem can be simply solved in $O(logn)$ time complexity with a good index structure (binary search). Furthermore, each probe is independent from each other, so retrieval for each probe can be highly parallel computed. This is a remarkable improvement compared with the $O(n)$ linear scan method when n is of very large scale.

The multi-probe sequence Δ for a query code is generated using Algorithm 1. For example, given a probe radius R, the sequence Δ has $\sum_{r=0}^{R} C(K,r)$ elements as shown in Fig.2, where $C(K,r) = \frac{(K(K-1)\ldots(K-r+1))}{r!}$. Since every probe in Δ is sorted with hamming distance from the original query code, the fetched results are also in ascending ordered by the distance from query code.

Algorithm 1. *Generate Multi-Probe Sequence*

```
Input: query points hash code $q$, code length of hash code $K$, probe radius $R$
Output: Multi-probe sequence $\Delta$

$\Delta \leftarrow \emptyset$
for $r = 0$ to $R$
    Set of probes $p$ with distance $r$ from $q$ are defined as: $\Omega_r = \{ p \mid dist(q,p) = r \}$
    $\Delta \leftarrow \Delta \cup \Omega_r$
end for
```

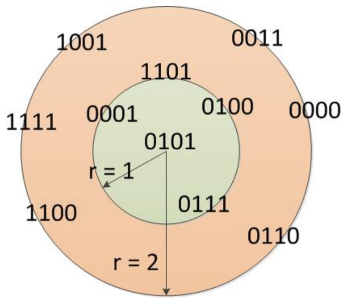

Fig. 2. An example of generating a multi-probe sequence with probing radius within 2. The query code is "0101" in the center of the circle. The sequence has 1+4+6 = 11 elements.

3.3.2 ID Mapping

As demonstrated in section 2.3, in the index structure of Lucene, internal ids are stored in inverted table. The query hash code is first located in the dictionary, then, internal ids linked to the hash code are fetched. After this, a second fetch step using the internal ids is executed to get the UID in the documents.

The second fetch step is redundant as the whole document is taken out. However, in our LuSH framework, we only need the UID. To speed up retrieval, we add an extra structure to store the mapping from internal id to UID in memory. Consider that the mapping maybe to too large to store in memory, cache strategy is used.

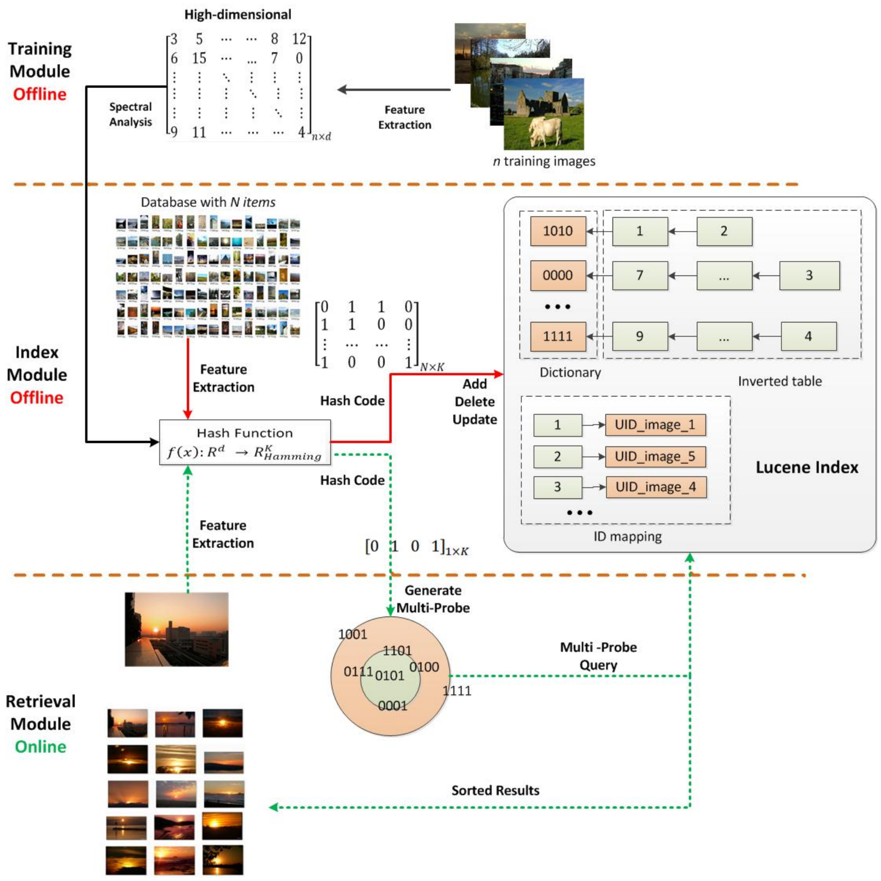

Fig. 3. System architecture of LuSH

4 Experimental Results

In this section, we evaluate the performance of our LuSH framework on a typical application of CBIR system and test the accuracy and efficiency on large datasets (up to 10 million).

4.1 DataSet and Evaluation Criterion

To evaluate the performance of LuSH, we conduct experiments on two image datasets as follows.

- Flickr Images: an image dataset crawled by ourselves, which contains 100,000 images of varies of species.
- Sogou Images[18]: a huge scale image dataset provided by Sogou Lab. It contains up to 10 million images with categories of people, animals, building, scenery, etc.

To evaluate the performance of LuSH, four criterions are concerned.

- precision of the retrieval result
- Index storage size
- Index building time
- query response time

4.2 Experimental Setup

To apply LuSH in a CBIR system, the first thing is to extract visual features from images. In our experiment, Color and Edge Directivity Descriptor (CEDD)[2] is used.

The two datasets have different emphases in our experiments. The Flickr Images dataset is relatively tiny; it's suitable to evaluate precision of the retrieval result since the computation for ground truth for a very large dataset is time costing. The Sogou Images dataset is used to test the performance of our index structure, i.e. index size, build time and query response time.

For the Flickr Images dataset, we evaluate the precision of our LuSH. 10,000 images are random selected from the database as the training set and another 100 images as query set. The ground truth for each query is the top 1% NN in Euclidean distance and the NN we retrieval using our LuSH are within hamming distance 2. To show the correlation of precision and code length *K, K* ranges from 8 to 64.

For the Sogou Images dataset, we generate four subsets in different size with random sampling, i.e. sizes of the subsets are 10^4, 10^5, 10^6, 10^7. We build index for the four subsets using a multi-thread strategy (20 threads). When index building procedure is completed, we analyze the size of index file and the index building time, then use 100 random samples to execute query in these indexes and evaluate average query response time. Since the scale of dataset is large, code length K is set 64 and radius for NN is within 2 to achieve high precision.

4.3 Experimental Results Analysis

The precision result of LuSH on Flickr Image dataset is shown in Fig.4. As the code length increase, the precision increase simultaneously, accompanying with the time cost for computing. For the scale of 100,000 images, $K = 40$ is a relatively ideal trade-off.

The size of index files and index build time on Sogou Images dataset are demonstrated in Table 1, the size of index file, index build time are linear increase with the size of dataset, which is in our expectation.

Focus on the largest subset with $N = 10$ million, the size of index file is about 1GB, which is 10^{-5} of the original images and 10^{-2} of the extracted feature vectors. This is an impressive compression ratio. The corresponding indexing time is less than one hour, since it's an offline step, it satisfies most of the system-level application's requirement.

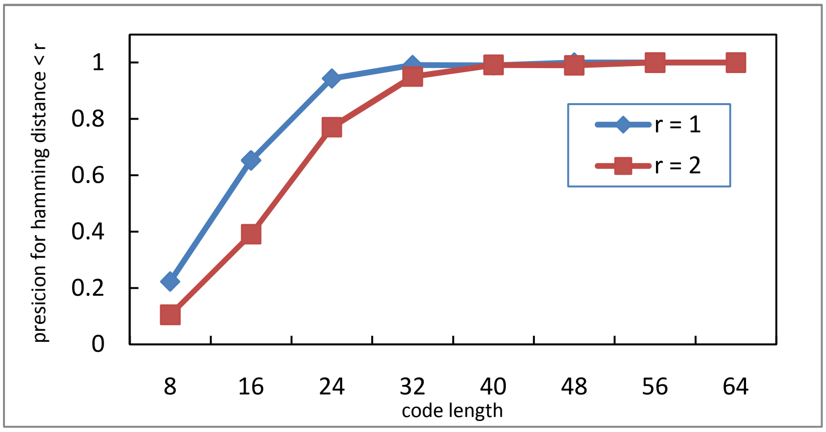

Fig. 4. Correlation of precision and hash code length

Table 1. Size of index file and index build time

Size of dataset (*N*)	Size of Index File(*MB*)	Index Building Time(*s*)
10,000	1.5	15
100,000	13	63
1,000,000	124	446
10,000,000	1151	3353

We then evaluate the average query response time. Since linear scan is not applicable to this large scale datasets. We don't compare our multi-probe strategy with it. We compare the standard Lucene's "quadric" fetch method with our ID mapping strategy and show result in Table.2. It is obvious that our ID mapping strategy gets better results than the standard method in all of the four different sizes of subsets.

Table 2. Average query response time

Size of dataset (*N*)	Average query response time(*ms*)	
	Standard	ID mapping
10,000	8.97	**7.15**
100,000	30.08	**25.2**
1,000,000	201	**180.7**
10,000,000	1960	**1840**

5 Conclusion and Future Work

In this paper, we propose a generic high-dimensional index framework: LuSH, which fuses the Spectral Hashing with Lucene index. To speedup retrieval efficiency, two strategies are proposed:

- Multi-probe retrieval strategy is used instead of linear search in SH.
- An extra mapping from Lucene's internal id to UID is added in Lucene index structure to avoid redundant fetch operation.

We use CBIR system as a typical application to demonstrate the architecture of LuSH and then do experiment on large scale image datasets (up to 10 million) to show the performance of LuSH framework.

In the future, we consider extending LuSH to distributed index environment.

Acknowledgment. This work is supported by the HEGAOJI Project under grant No.2010ZX01042-002-003-001, the National Natural Science Foundation of China No. 61105074, 61103099, the China Postdoctoral Science Foundation, and the Fundamental Research Funds for the Central Universities.

References

1. Tamura, H., Mori, S., Yamawaki, T.: Textural features corresponding to visual perception. IEEE Transactions on Systems, Man, and Cybernetics 8(6), 460–472 (1978)
2. Chatzichristofis, S.A., Boutalis, Y.S.: CEDD: Color and Edge Directivity Descriptor: A Compact Descriptor for Image Indexing and Retrieval. In: Gasteratos, A., Vincze, M., Tsotsos, J.K. (eds.) ICVS 2008. LNCS, vol. 5008, pp. 312–322. Springer, Heidelberg (2008)
3. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision (IJCV) 60(2), 91–110 (2004)
4. Salakhutdinov, R., Hinton, G.: Semantic hashing. International Journal of Approximate Reasoning 50(7), 969–978 (2009)
5. Apache Lucene Project, `http://lucene.apache.org/`
6. Guattman, A.: R-Tree: A Dynamic Index Structure for Spatial Searching. In: ACM SIGMOD Int. Conf. on Management of Data, Boston, pp. 47–57 (1984)
7. Bentley, J.L.: Multidimensional binary search Trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)
8. Bellman, R.: Adaptive control processes: a guided tour. Princeton University Press (1961)
9. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 604–613. ACM, New York (1998)
10. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS, pp. 459–468 (2006)
11. Salakhutdinov, R., Hinton, G.: Learning a nonlinear embedding by preserving class neighbourhood structure. In: AI and Statistics, p. 5 (2007)
12. Shakhnarovich, G., Viola, P., Darrell, T.: Fast pose estimation with parameter sensitive hashing. In: ICCV, pp. 750–757 (2003)

13. Weiss, Y., Torralba, A., Fergus, R.: Spectral hashing. In: Advances in Neural Information Processing Systems, pp. 1753–1760 (2009)
14. Bengio, Y., Delalleau, O., Roux, N., et al.: Learning eigenfunctions links spectral embedding and kernel PCA. Neural Computation 16(10), 2197–2219 (2004)
15. Coifman, R., Lafon, S., Lee, A., et al.: Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps. Proc. of the National Academy of Sciences of the United States of America 102(21), 7426 (2005)
16. Belkin, M., Niyogi, P.: Towards a theoretical foundation for Laplacian-based manifold methods. Journal of Computer and System Sciences 74(8), 1289–1308 (2008)
17. Comer, D.: Ubiquitous B-tree. ACM Computing Surveys (CSUR) 11(2), 121–137 (1979)
18. SogouP2.0, http://www.sogou.com/labs/dl/p2.html
19. Ng, A.Y., Jordan, M.I., Weiss, Y.: On Spectral Clustering: Analysis and an algorithm. Journal of Advances in Neural Information Processing Systems 2, 849–856 (2002)