HashFile : An Efficient Index Structure For Multimedia Data

Dongxiang Zhang [#1], Divyakant Agrawal [†2], Gang Chen [*3], Anthony K. H. Tung [#4]

[#]*School of Computing, National University of Singapore,* {[1]zhangdo,[4]atung}@comp.nus.edu.sg

[†]*Department of Computer Science, University of California, Santa Barbara,* [2]agrawal@cs.ucsb.edu

[*]*College of Computer Science, Zhejiang University,* [3]cg@cs.zju.edu.cn

***Abstract*—Nearest neighbor (NN) search in high dimensional space is an essential query in many multimedia retrieval applications. Due to the curse of dimensionality, existing index structures might perform even worse than a simple sequential scan of data when answering exact NN query. To improve the efficiency of NN search, locality sensitive hashing (LSH) and its variants have been proposed to find approximate NN. They adopt hash functions that can preserve the Euclidean distance so that similar objects have a high probability of colliding in the same bucket. Given a query object, candidate for the query result is obtained by accessing the points that are located in the same bucket. To improve the precision, each hash table is associated with m hash functions to recursively hash the data points into smaller buckets and remove the false positives. On the other hand, multiple hash tables are required to guarantee a high retrieval recall. Thus, tuning a good tradeoff between precision and recall becomes the main challenge for LSH. Recently, locality sensitive B-tree(LSB-tree) has been proposed to ensure both quality and efficiency. However, the index uses random I/O access. When the multimedia database is large, it requires considerable disk I/O cost to obtain an approximate ratio that works in practice.**

In this paper, we propose a novel index structure, named HashFile, for efficient retrieval of multimedia objects. It combines the advantages of random projection and linear scan. Unlike the LSH family in which each bucket is associated with a concatenation of m hash values, we only recursively partition the dense buckets and organize them as a tree structure. Given a query point q, the search algorithm explores the buckets near the query object in a top-down manner. The candidate buckets in each node are stored sequentially in increasing order of the hash value and can be efficiently loaded into memory for linear scan. HashFile can support both exact and approximate NN queries. Experimental results show that HashFile performs better than existing indexes both in answering both types of NN queries.

I. Introduction

Due to the proliferation of Web 2.0 social and community systems, a large number of multimedia objects are publicly available. Efficient access to such multimedia objects needs to be supported in order for the Web users to benefit from such data. In this paper, we study the problem of nearest neighbor (NN) search, which is an essential query in many multimedia retrieval applications. We investigate processing strategies for both exact and approximate NN queries. The former has wide applications in similarity search, pattern recognition, clustering and classification. The latter is particularly suitable for efficient retrieval in a large scale database at the risk of certain loss in quality.

The most common and straightforward method for solving exact NN problem is based on hierarchical space partitioning, resulting in various kinds of tree structure indexes [23], [14], [5], [20]. The multi-dimensional feature space is split into smaller partitions and organized as a tree structure. Data close to each other are grouped in the node so that they can be pruned together without accessing each individual point inside. However, the pruning power of these indexes decreases as dimensionality grows and most of the tree nodes will be accessed, taking considerable CPU and I/O cost. In this case, the performance of existing index structures degrades rapidly and even becomes worse than a simple sequential scan of the data [28]. Due to this curse of dimensionality, it is difficult to build indexing support to efficiently answer exact NN queries.

To provide efficient similarity search, the research community has focused on approximate NN search in recent years. Among various efforts, locality sensitive hashing (LSH) [11], [10] and its variants have received considerable attention. The LSH family adopts hash functions that preserve the distance in the Euclidean space so that similar objects have a high probability of colliding in the same bucket. If there are l hash tables and each table is associated with m hash functions, an object o will be hashed to $H(o) = [h_1, h_2, ..., h_l]$, where $h_i = G_{i_1}(o)G_{i_2}(o)...G_{i_m}(o)$. Given a query object q, the search space includes the buckets in the l hash tables where q is located. All the objects in these buckets are scanned to return the approximate NN result. As m increases, the bucket size becomes smaller and more false positives are removed. Precision increases but recall degrades. Similarly, as l increases, more buckets are examined. Recall is improved but precision may become worse. Thus, the main challenge of LSH is to tune a good tradeoff between precision and recall. To achieve a high search accuracy, hundreds of hash tables are normally used [10] and require a large amount of memory space. In [22], multi-probe LSH was proposed to reduce the number of hash tables and obtain the same search quality. Since multi-probe LSH is adhoc and without theoretical guarantee, Tao et al. have recently proposed the locality sensitive B-tree (LSB-tree) [27] to ensure both quality and efficiency. The drawback of LSB tree is that it uses random IO access, which requires a considerable number of disk accesses when the database is large.

To support efficient NN query processing, we propose a

novel index structure, named *HashFile*, based on the following three observations:

1) In LSH, the hash function is likely to place most of data objects into the buckets near the mean hash value, resulting in a skewed distribution of bucket size. When m increases, the dense buckets can be recursively partitioned to reduce the number of false positives. However, other buckets containing fewer points will be partitioned as well. Since the data points inside these buckets are well separated from others, further partitioning will generate a large number of very small buckets and the quality of the approximate results may degrade.
2) Expanding the search space by inspecting neighboring buckets can significantly improve the result quality [22], [18] because some missing nearest neighbors can be retrieved in this way without building a new hash table. Although such a method is adhoc and without theoretical guarantee, we argue that the theoretical bound obtained by previous works[11], [27] is too loose to be applied in practice. For example, LSB-tree only guarantees 4-approximate NN with at least constant probability.
3) Disk page access method plays an important role in the cost of the index look-up [3], especially for high dimensional data. The time for random access is higher than that of sequential access by many times.

In our implementation, the entire data set is first hashed into a set of buckets like LSH. The dense buckets are fetched and re-hashed so as to further separate the objects inside to remove the false positives. The remaining buckets only contain a small number of points and will not be further partitioned. The process will be repeated until there is no dense bucket. In this manner, the distribution of the bucket size is much more balanced than that generated by LSH. We organize HashFile as a tree structure and each node is associated with a unique hash function as well as a data file to store the buckets that can not be further partitioned. These buckets are stored in increasing order of the hash value for linear scanning. We propose a dynamic bucket allocation strategy to guarantee a minimum storage utilization of 50%. Given a query point q, the search algorithm explores the tree in a top-down manner and only retrieves those pages around the query object. The candidate pages in each file are retrieved using sequential scan, with a single disk seek operation followed by the data transfer.

In this paper, we choose random projection as our hash function. This is because it is simple, database-friendly [1] and yields comparable results to conventional dimensionality reduction, such as PCA, for both image and text data [6]. Furthermore, using random projection, we can extend the search algorithm to support exact NN search in L_1 norm, which is found to be effective for multimedia data[2]. Experiment results on image data sets show that performance is improved as random projection is useful to filter away the data points that are far away and the remaining candidates are processed efficiently using a linear scan. In summary, in this paper we propose HashFile, a novel index structure for processing nearest neighbor queries efficiently over multimedia databases. HashFile has the following desirable features:

1) It supports approximate NN search in the Euclidean space as well as exact NN search in L_1 norm.
2) It has a linear space complexity of $O(2N+N/B)$ where N is the data size and B is the page size.
3) Experiment results show that HashFile outperforms state-of-the-art techniques for processing both types of NN queries.

The rest of the paper is organized as follows. First, in Section II, we review the necessary background to provide the underlying ideas and the intuition behind HashFile for a better understanding. The detailed algorithm for inserting, updating and deleting data objects into and from HashFile is presented in Section III. The query processing strategies for exact NN and approximate NN are proposed in Sections IV and V, respectively. In Section VI, we analyze the complexity of data operations and query processing. In Section VII, we review related work in the area of high-dimensional indexing. Extensive experiments on the real image data sets are conducted in Section VIII to establish the superiority of the proposed methods over current state-of-the art NN processing techniques. Section IX concludes the paper.

II. The Preliminaries

In this section, we briefly review the theoretical background on random projection and motivate the intuition for the notion of HashFile.

A. Random Projection

Random projection is a powerful method for dimensionality reduction. It is computationally efficient and sufficiently accurate. Given a d dimensional data set P with n points and a random matrix $R_{d\times k}$, the projection is computed as follows:

$$P'_{n\times k} = P_{n\times d} \times R_{d\times k},$$

which results in a k dimensional data set P' with n points. Random projection can preserve the Euclidean distance in the lower dimensional space provided the constraints specified by the Johnson-Lindenstrauss Lemma [13] hold.

Lemma 2.1 (Johnson-Lindenstrauss Lemma): Given $\epsilon > 0$ and an integer n, let k be a positive integer such that $k \geq k_0 = O(\epsilon^{-2} logn)$. For every set P of n points in $\mathbb{R}^d$, there exists $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$ such that for all $u, v \in P$, we have

$$(1-\epsilon) \parallel u-v \parallel^2 \leq \parallel f(u)-f(v) \parallel^2 \leq (1+\epsilon) \parallel u-v \parallel^2$$

The choice of the random matrix is the key step and it is often Gaussian distributed with mean 0 and variance 1 to solve the ϵ-approximate nearest neighbor problem [16], [19], [11], [22], [27]. In [1], Achlioptas showed that the computational efficiency can be further improved by replacing the Gaussian distribution with much simpler random distributions:

Lemma 2.2: Given $\epsilon, \beta > 0$, let $k_0 = \frac{4+2\beta}{\epsilon^2/2 - \epsilon^3/3} \log n$. For integer $k \geq k_0$, let R be a $d \times k$ random matrix where the elements are independent random variables from either one of the following probability distributions:

$$r_{ij} = \begin{cases} +1 & with\ probability\ 1/2 \\ -1 & with\ probability\ 1/2 \end{cases}$$

or

$$r_{ij} = \sqrt{3} \begin{cases} +1 & with\ probability\ 1/6 \\ 0 & with\ probability\ 2/3 \\ -1 & with\ probability\ 1/6 \end{cases}$$

With probability at least $1 - n^{-\beta}$, for all $u, v \in P$,

$$(1-\epsilon) \parallel u - v \parallel^2 \leq \parallel f(u) - f(v) \parallel^2 \leq (1+\epsilon) \parallel u - v \parallel^2$$

In practice, we can select either of the two projection functions for dimensionality reduction. No matter which one is used, we always have the element r_{ij} in the random matrix such that $r_{ij} \in \{-1, 0, 1\}$.

B. Distance constraint for exact NN query using L_1

Suppose $\mathcal{H}$ is a hash function derived from any row of $R_{d \times k}$ such that it hashes an object o with d dimensions into one dimensional value:

$$\mathcal{H}(o) = \lfloor \sum_{i=1}^{d} h_i \cdot o_i \rfloor$$

Since each element h_i in $\mathcal{H}$ is from $\{-1, 0, 1\}$, we can easily achieve a lower bound for the exact L_1 distance.

Lemma 2.3: Given two d dimensional data points x and y and a hash function $\mathcal{H} : \mathbb{R}^d \rightarrow \mathbb{R}^1$ with $h_i \in \{-1, 0, 1\}$, we have

$$\parallel x - y \parallel_{L_1} \geq |\mathcal{H}(x) - \mathcal{H}(y)| - 1$$

Proof:

$$\begin{aligned} \parallel x - y \parallel_{L_1} &= \sum_{i=1}^{d} |x_i - y_i| \\ &\geq \sum_{i=1}^{d} |h_i(x_i - y_i)| \\ &\geq |\sum_{i=1}^{d} h_i(x_i - y_i)| \end{aligned}$$

Since for two real values a and b, $|a - b| \geq ||\lfloor a \rfloor - \lfloor b \rfloor| - 1$, we have

$$\parallel x - y \parallel_{L_1} \geq |\mathcal{H}(x) - \mathcal{H}(y)| - 1$$

■

Example. Figure 1 shows an illustrative example. Given three 4-dimensional data points x, y and z, a random hash function is generated to hash each point into a 1-dimensional value. It is obvious that the distance of the hashed value is the lower bound of their real distance. Since the elements are integers, we have $|a - b| = ||\lfloor a \rfloor - \lfloor b \rfloor|$ for two integers a and b. The factor "-1" can be dropped from the RHS and therefore we get: $\parallel x - y \parallel_{L_1} \geq |\mathcal{H}(x) - \mathcal{H}(y)|$.

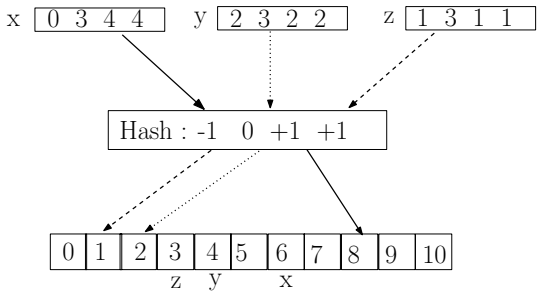


Fig. 1. A random projection example

$$\begin{aligned} \parallel x - y \parallel_{L_1} &= 6 \geq 5 = |\mathcal{H}(x) - \mathcal{H}(y)| \\ \parallel x - z \parallel_{L_1} &= 7 \geq 6 = |\mathcal{H}(x) - \mathcal{H}(z)| \\ \parallel y - z \parallel_{L_1} &= 3 \geq 1 = |\mathcal{H}(y) - \mathcal{H}(z)| \end{aligned}$$

Inspired by this, we adopt random projection to partition the large volume of high dimensional data points into buckets in 1-dimensional space. Each bucket is associated with a hash value and occupies one disk page. The pages are stored sequentially in increasing order of the hash value. Given a query point q, suppose it is hashed into the bucket h_q. We denote δ as the distance to the nearest neighbor ever found. According to Lemma 2.3, only the pages with hash value in $[q_h - \delta, q_h + \delta]$ need to be accessed. The other data points outside the range can be safely pruned. Figure 2 illustrates an example of frequency distribution of each hash value derived from a color histogram dataset with $200,000$ points corresponding to 200K images, which looks similar to a normal distribution. Most of the data points are hashed into the buckets around the mean hash value μ. For buckets far away from μ, the number of points inside the bucket drops dramatically. Therefore, if q_h is lucky to be far from μ, e.g., q_2 in the figure, the search space only includes a set of buckets with relatively few points inside the buckets. Since the pages with hash value in $[q_h - \delta, q_h + \delta]$ are stored sequentially, we can use linear scan to process the data without resorting to random I/O accesses. However, if $q_h = q_1$ as shown in the figure, we need to access the majority of the data points to answer exact NN query.

To solve this problem, we need to further partition the dense buckets. A new hash function is created for each dense bucket to further partition the data points inside. We expect that some amount of data far away from q can still be pruned away via the re-hashing. However, as each bucket is associated with only a value of $\mathcal{H}(\cdot)$, the bucket size is relatively small. As shown in Figure 2, even the densest bucket contains less than $1,000$ data points. The pruning power of re-hashing in such a small partition is very limited. Each time we access the new partition, only very few points can be pruned. This results in high I/O cost to perform sequential scan on these small files. Therefore, we adopt the popular hash function in LSH

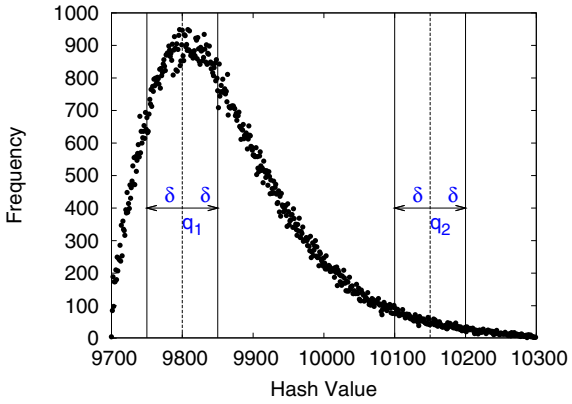


Fig. 2. Hash value frequency of a color histogram dataset

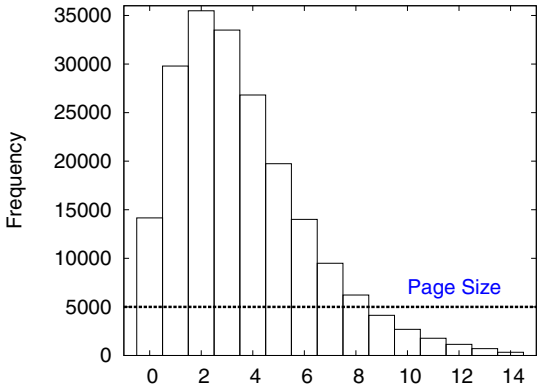


Fig. 3. New frequency distribution of window based hashing

to increase the number of points hashed to each bucket:

$$\mathcal{H}(o) = \lfloor \frac{\vec{a} \cdot \vec{o} + b}{W} \rfloor$$

In our case, a_i is selected the same with r_{ij}. b_i is not required and simply set as 0. W is the hash window size. The original hash space is split into intervals with length W and the data points hashed to the same interval will be stored in the same bucket. Obviously, the larger W is, the more data points will be hashed to the same bucket. Figure 3 shows the new frequency distribution in the larger buckets. W is set to 450 and there are now only 15 partitions in the hash space. The densest bucket contains around $35,000$ data points. Given a fixed page size, all the buckets whose size exceeds the page size will be re-partitioned to gain additional pruning power. When q is located in the dense area, we do not need to access all the data points in the nearby buckets as in Figure 2. Instead, points far away from q can still be pruned to save the CPU cost. Finally, we prove that we can still achieve a lower bound of distance constraint using the new hash function.

Lemma 2.4: Given two d dimensional data points x and y and a hash function $\mathcal{H}_w(o) = \lfloor \frac{h_i \cdot o_i}{W} \rfloor$, we have

$$\| x - y \|_{L_1} \geq W(|\mathcal{H}_w(x) - \mathcal{H}_w(y)| - 1)$$

Proof:

$$\begin{aligned} \| x - y \|_{L_1} &\geq |\sum_{i=1}^{d} h_i(x_i - y_i)| \\ &= W|\sum_{i=1}^{d} \frac{h_i x_i}{W} - \sum_{i=1}^{d} \frac{h_i y_i}{W}| \\ &\geq W(|\mathcal{H}_w(x) - \mathcal{H}_w(y)| - 1) \end{aligned}$$

■

In the extreme case, when W is set to $+\infty$, HashFile degrades to linear scan without any pruning power. In our experiment setup, we will study how the performance varies with this parameter.

III. HashFile Index Structure

In the section, we first present the overall design of the HashFile index structure then describe the algorithms for data insertion, update and deletion in HashFile. The notations used in the presentation are summarized in Table I.

TABLE I
NOTATION TABLE

d	the data dimension
o	an object with d dimensions
W	the hash window size
$\mathcal{H}_w$	$\mathcal{H}_w(o) = \lfloor \frac{\sum_{i=1}^{d} h_i \cdot o_i}{W} \rfloor$
$[l_i, h_i]$	the hash interval of a page
B	the page size to host B objects
δ	the best NN distance ever found
μ	the storage utilization rate
κ	the tree height of HashFile

A. HashFile Overview

We build HashFile in a top-down manner based on the distribution of the data set. In the beginning, we randomly generate a hash function in the root node and create a disk file to store the data. Since we are unable to predict the hash range, we cannot allocate a collection of fixed-size pages in advance. Moreover, such a static allocation strategy wastes a lot of storage resource because the buckets far from the mean hash value usually contain a small number of data points, resulting in a low storage utilization. Hence, we propose a dynamic allocation mechanism which guarantees that the page utilization is at least 50%.

Initially, there is only one page allocated in the disk file with a hash interval $(-\infty, \infty)$. The size of the page is fixed as B, indicating that it can accommodate B data points. The first B objects can be successfully inserted into the page. When the $(B+1)$-th object arrives, the page is full and we select a hash value h_s and split the page into two new pages. Now, the object can be inserted into the new page based on its hash value. As new objects are inserted continuously, more and

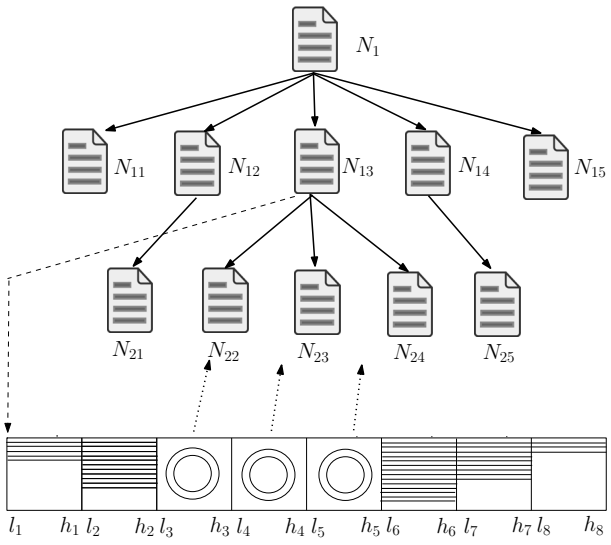


Fig. 4. The structure of HashFile and HashNode

more split operations occur and the hash intervals of the page become smaller and smaller. Finally, an object may be hashed to a full bucket in which all the data points inside this page are associated with the same hash value and can not be split any more using the node's hash function. To insert this new object, we need to create a child node with a new random projection function. The data points in that page are extracted and re-hashed into the child node's data file. At this time, we have vacant space in the child node to accommodate the new object.

Figure 4 illustrates an example of HashFile with the tree structure and the logical view of one internal tree node. The tree is built by continuous insertion of the data points. When a bucket in the parent node can not be split any more, child nodes are created to further partition the dense bucket. Therefore, each internal node contains a list of child nodes and pages as shown in the figure. Each cell represents one page with a hash interval $[l_i, h_i]$. The pages are stored in increasing order of the hash value, i.e., $l_i \leq h_i \leq l_{i+1}$ and $\bigcup [l_i, h_i] = (-\infty, \infty)$. The pages with circles inside are buckets with only one hash value and the data inside have been extracted and re-hashed into the child node. In this example, we have $l_3 = h_3$, $l_4 = h_4$ and $l_5 = h_5$. Therefore, node N_{13} has three child nodes N_{22}, N_{23} and N_{24}. Note that in the physical file, these three buckets do not occupy any page as their content has been extracted and stored in the child nodes' disk files. In this way, we ensure that the remaining disk pages are still sequentially stored.

B. Data Insertion

Given a data object o to be inserted into HashFile, we need to find the disk page in the tree node where o is hashed into. Since the hash value range of each tree node is $(-\infty, +\infty)$, we can ensure that the hash value of o must be located either in a disk page or in one of the child nodes. Thus, we start from the root node and compare the hash value of the intervals of each page. If we find a page p_i with interval $[l_i, h_i]$ such that $h_o \in [l_i, h_i]$, o will be inserted into this page. Otherwise, o must be hashed into one of the child nodes. We compare the hash value with the list of child nodes until we find a match. The child node is visited in a similar way with the root node. Finally, we must be able to find a disk page p_i in the tree node to host the object o.

If the found page p_i has vacant space, we directly append o into the page. Otherwise, we need to check the hash range of p_i to determine whether we split the page or create a new child node. If objects inside p_i are associated with multiple hash values, we can find the median hash value h_s so that we can split p_i into two new pages p_{i_1} and p_{i_2}. h_s is selected to make sure the page is split into two even parts to guarantee the storage utilization is no less than 50%. The hash intervals of the p_{i_1} and p_{i_2} become $[l_i, h_s]$ and $[h'_s, h_i]$ respectively. Note that the objects hashed to h_s could appear in both the

Algorithm 1 `InsertData(`o`)` **: Insert an object into HashFile**

Input: Data object o

```
1.  p_i = FindPage(o)
2.  if p_i is not full then
3.      append o in p_i
4.  else
5.      if h_i > l_i then
6.          select a hash value h_s
7.          split the pages into two pages p_{i_1} and p_{i_2} with
            intervals [l_i, h_s] and [h'_s, h_i] respectively
8.          leave data points of p_{i_1} in the page p_i
9.          create a new page for p_{i_2} and insert p_{i_2} next to p_{i_1}
10.         if h_o ≤ h_s then
11.             append o in p_{i_1}
12.         else
13.             append o in p_{i_2}
14.     else
15.         create a new hash function and a new disk file as the
            child node
16.         extract p_i and hash the points to the new file
17.         move p_j to p_{j-1} for j > i in the parent node
18.         InsertData(o)
```

Algorithm 2 `FindPage(`o`)` **: Find the disk page o is hashed into**

Input: Data object o

Output: The disk page that o is hashed into

```
1.  init node as the root node
2.  while 1 do
3.      h_o = node.hash(o)
4.      if h_o is the hash value of a child node child then
5.          node = child
6.      else
7.          find disk page p_i so that h_o ∈ [l_i, h_i]
8.          return page p_i
```

new pages. In this case, $h_s = h'_s$. Otherwise, $h'_s = h_s + 1$. We leave the objects hashed into $[l_i, h_s]$ in the original page p_i and create a new physical page for p_{i_2}. To ensure that the hash intervals are in increasing order for sequential scan, we insert the page p_{i_2} right next to p_i in the file.

If p_i is full and all the objects inside p_i are hashed to the same value, we can not split the page any more. Thus, we create a new hash function as the child node for further partition of p_i. All the contents in p_i are extracted and hashed into the child node. p_i is deleted in the parent node and we move the block of pages $\{p_j | j > i\}$ ahead to fill the gap. The object o can now be inserted into the child node with the new hash function. The detailed algorithm of data insertion is shown in Algorithm 1.

C. Data Deletion

The delete operation is simpler than the insert operation. Given an object o to delete, we first find the disk page where o is located using Algorithm 2. Then, we sequentially scan the data inside and delete the object. A simpler implementation is to maintain a bitmap with B bits for each page. Each bit indicates whether the corresponding slot is free or in use. To delete an object, we only need to set the status flag as free. In this case, to insert an object, we only need to find any free slot for the data.

If the page becomes empty after the delete operation, we do not delete it. The reason is that the overhead caused by the empty page in the query processing stage is negligible. The pages are loaded into the memory in blocks and do not require additional CPU cost when scanning the data. Moreover, in the Web 2.0 applications, insert operations are much more frequent than delete operations. There would be new objects inserted into the page in the future. Thus, we can save the overhead of removing the page cost by simply leaving the page there and reusing it later when new objects are inserted.

D. Data Update

Given an object o to update, since its hash value could be updated as well, we can not simply update it in the data file. Instead, we treat an update operation as a deletion followed by an insertion.

IV. Exact NN Query Processing

In this section, we present the exact NN query processing algorithm using the proposed HashFile. We adopt the lower bound distance constraint in Lemma 2.4 to prune the search space. Since we have organized the disk pages in the file in increasing order of the hash value, the candidates can be retrieved efficiently.

Algorithm 3 `ExactNN` **: Exact NN Search in HashFile**

Input: Query object q
Output: The distance δ from q to its nearest neighbor

1. init δ
2. δ = `ExactNNInNode` $(q, root, \delta)$
3. return δ

Algorithm 3 and Algorithm 4 show how exact NN search is performed in HashFile. Given a query point q, we start from the root node and recursively search the child nodes in the depth-first order. Suppose q is hashed to q_h in the current tree node, the candidate hash space is $[h_q - \frac{\delta}{W} - 1, h_q + \frac{\delta}{W} + 1]$. We check the list of child nodes whether their hash values are located in this range. The candidate child nodes are put in the heap in increasing order of their distance to h_q so that the most promising child node can be accessed first in the best first search manner. Meanwhile, we check the the disk pages to find the start offset and end offset whose hash intervals intersect with our candidate search space. The block of pages are loaded into memory using random access and scanned sequentially. If a better result is found, we update δ accordingly. Finally, the algorithm terminates until all the candidate nodes are explored.

Algorithm 4 `ExactNNInNode` **: Exact NN Search in the HashNode**

Input: Query object q, a tree node $node$ and δ
Output: The new δ

1. $h_q = node.hash(q)$
2. **for** each child node $child$ **do**
3. $\quad cdist = W(|child.hashvalue - h_q| - 1)$
4. $\quad$ **if** $cdist < \delta$ **then**
5. $\quad\quad$ add $child$ to the heap ordered by $cdist$
6. **for** each node $candiddate$ in the heap **do**
7. $\quad \delta$ = `ExactNNInNode` $(q, candidate, \delta)$
8. **for** each disk page p_i in increasing order **do**
9. $\quad$ find the list of pages from p_{start} to p_{end} so that their intervals intersect with $[h_q - \frac{\delta}{W} - 1, h_q + \frac{\delta}{W} + 1]$
10. load the block of pages into memory
11. **for** each object o in the block **do**
12. $\quad dist = \| o - q \|_{L_1}$
13. $\quad$ **if** $dist < \delta$ **then**
14. $\quad\quad \delta = dist$
15. return δ

V. Approximate NN Query Processing

In LSH, each hash table is associated with m hash functions. Intuitively, we can consider all the buckets are organized in a height-balanced tree in which the path length from the root to the leaf nodes is always m. When m increases, the bucket size becomes smaller and it is more likely for the objects close to each other to be hashed into neighboring buckets and missed by the search algorithm. Thus, expanding the search space by inspecting neighboring buckets can significantly improve the result quality [22], [18].

Inspired by this, we propose a flexible and effective method to process approximate NN query. Users can specify a parameter λ to determine the search space. If q is hashed to q_h, we only explore the neighboring buckets with hash value interval intersecting with $[h_q - \lambda, h_q + \lambda]$. Note that λ specifies the query range in the hash space instead of the number of neighboring buckets. Figure 5 shows an example in which q is hashed to B_5 with hash value 9. When $\lambda = 1$, only three

buckets $\{B_4, B_5, B_6\}$ are in the search space. All the points in B_6 are scanned sequentially. Since B_4 and B_5 correspond to child nodes, they will be processed in a similar way with their parent node. When $\lambda = 3$, the search space expands to $\{B_2, B_3, B_4, B_5, B_6, B_7\}$. Since buckets $\{B_2, B_6, B_7\}$ are stored sequentially in the data file, we can load them into memory for linear scan.

Algorithm 5 and 6 illustrate how to answer an approximate NN query. The search process starts from the root node in a top-down manner and recursively explore the neighboring buckets within the search range. These candidate buckets are retrieved using sequential scan. When λ is 0, we only examine the points located within the same bucket with the query object. The is the same with the basic LSH.

Algorithm 5 `ApproximateNN` **: Approximate NN Search in HashFile**

Input: Query object q and query range λ
Output: The distance δ from q to its approximate nearest neighbor

1. init δ
2. δ = `ApproximateNNInNode` $(q, root, \delta, \lambda)$
3. return δ

Algorithm 6 `ApproximateNNInNode` **: Approximate NN Search in the HashNode**

Input: Query object q, a tree node $node$, distance δ, and query range λ
Output: A new δ

1. $h_q = node.hash(q)$
2. **for** each child node $child$ **do**
3. $\quad wdist = |child.hashvalue - h_q|$
4. $\quad$ **if** $wdist < \lambda$ **then**
5. $\quad\quad$ add $child$ to the heap ordered by $wdist$
6. **for** each disk page p_i in increasing order **do**
7. $\quad$ find the list of pages from p_{start} to p_{end} so that their intervals intersect with $[h_q - \lambda, h_q + \lambda]$
8. load the block of pages into memory
9. **for** each object o in the block **do**
10. $\quad dist = \| o - q \|_{L_2}$
11. $\quad$ **if** $dist < \delta$ **then**
12. $\quad\quad \delta = dist$
13. return δ

VI. Complexity and Cost Analysis

In this section, we analyze the storage utilization as well as the query time cost in answering both exact and approximate NN queries. Since the split operation partitions one page into two even parts, we have the first lemma on the storage utilization:

Lemma 6.1: Given a data set P with N points, where $N \gg B$, after inserting all the data points into HashFile, the storage utilization rate for each page $\mu \geq 50\%$.

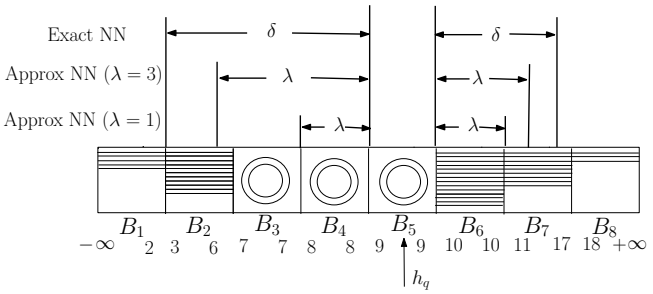


Fig. 5. Approximate search in the tree node

Proof: We prove that $\mu \geq 50\%$ is the loop invariant during the batch insertion:

Initialization : After the first $\frac{B}{2}$ data points are inserted, there is only one page and $\mu = 50\%$.

Maintenance : When a new data is inserted, there are three cases.

1) The page has free space to host the data. The number of points in that page will increase by 1. Thus, $\mu' > \mu \geq 50\%$.
2) If the page is full and a split occurs, since the objects in the page can be evenly partitioned, we have $\mu \geq 50\%$ for the two new generated pages[1].
3) If the page is full and can not be split, a new child node is created and this page is moved to the child node's data file. Similar with case 2), when the new object is inserted to this page, a split occurs at the child node and $\mu \geq 50\%$ still holds.

Termination : When the insertion terminates, $\mu \geq 50\%$ still holds. ■

A. Storage Cost

Since the storage utilization rate for each page is no less than 50%, we only need at most $2N$ space to store the data points. Also, there would be at most N/B tree nodes, the storage cost for HashFile is $O(2N + N/B)$.

B. Exact NN Query

In the worst case, the search space includes the entire HashFile when no points can be pruned. Since $N \gg B$, each node contains at least two disk pages using the even split approach, there are at most N/B tree nodes in HashFile. The exact query processing algorithm visits all the data points and the tree nodes. Hence, the cost is $O(N(1 + \frac{1}{B}))$.

C. Approximate NN Query

In the approximate NN query, we usually select a small W so that the densest bucket contains a small fraction of the data size, say ϵN. Since we only visit at most $2\lambda + 1$ buckets in the root node, the total number of data accessed would be $O((2\lambda+1)\epsilon N)$, spread in $O(\frac{2(2\lambda+1)\epsilon N}{B})$ buckets. As each node

[1]Counting in the new data to be inserted, we actually split $B + 1$ objects into two pages and $\frac{B+1}{2} \geq \lceil \frac{B}{2} \rceil$

contains at least two buckets, we need to access $O(\frac{(2\lambda+1)\epsilon N}{B})$ node. The cost of approximate NN query becomes $O((1+\frac{1}{B})(2\lambda+1)\epsilon N)$.

VII. Related Work

In this section, we briefly review the index structures that are widely used to process exact NN queries and approximate NN queries in multimedia databases.

A. High dimensional index for exact NN query

There exist mainly three types of approaches to answering exact NN query based on the idea of space partitioning [23], [5], [14], [20], data approximation [28], [24], [17], [3] or one dimensional transformation [4], [12], [29].

The most common index structures are based on the notion of space partitioning, resulting in various types of tree-based index structures such as k-d-b tree [23], X-tree [5], SR-tree [14] and TV-tree [20]. In these trees, the pruning power of these methods becomes very weak as the dimensionality of data becomes very high. This can be offset by maintaining trees with very large height, but in that case since the number of internal nodes grows exponentially with the tree height, tremendous storage overhead is incurred. The performance of such trees degrades to be worse than linear scan. Although HashFile is also organized as a tree structure, the hierarchy partitioning is based on the one-dimensional hash value.

VA-file and iDistance are the representative indexes for data approximation and one dimensional transformation respectively. Weber et al. proposed VA-file[28] which uses bit encoding for pruning and takes advantage of linear scan for query processing. A look-up on the real data file is triggered when a point cannot be pruned based on the compressed representation. A proper compression rate must be specified for the best performance. Otherwise, the performance would become CPU-bound or IO-bound when it is set too large or too small. The major drawback of VA-file is the lack of flexibility in a dynamic environment as the data in the two files are sequentially stored. iDistance [12], [29] attempts to solve the problem by building a light-weight index using B^+-tree. A collection of reference points, which can be dynamically or statically determined, are selected implicitly to partition the space in Voronoi cells. Instead of splitting the data space, iDistance indexes the distance to these reference points. The advantage is that the index size is relatively small and demonstrates satisfactory pruning power. However, its performance is sensitive to the selection of the reference points and too much random access of the disk pages is required as the selectivity is coarse for NN query in the high dimensional space.

In contrast, HashFile combines the advantage of random projection and linear scan. Random projection is useful to filter away the data points that are far away and the remaining candidates are processed efficiently using a linear scan. In our experiment, we compare HashFile with VA-file and iDistance.

B. LSH for approximate NN query

LSH [11], [10] has been widely applied to answer the approximate NN query and shown to be quite effective for similarity search in multimedia databases including text data [25], audio data [7], images [15] and videos [9]. The query cost grows sub-linearly with the data set size in the worst case. However, it is a trivial job to tune a good tradeoff between the precision and recall. In practive, hundreds of hash tables have to be built for a high search accuracy [10]. To reduce the number of hash tables, Lv et al. proposed multi-probe LSH[22]. It can obtain the same search quality with much less tables. Since multi-probe LSH is adhoc and without theoretical guarantee, Tao [27] recently has proposed LSB-tree to address both the quality and the efficiency of multimedia retrieval. The hash values are represented as 1-dimensional Z-order values and indexed in the B^+-tree. Multiple trees can be built to improve the result quality. Compared with existing LSH methods, HashFile only recursively hash the dense buckets to achieve more balanced data partitions. Each bucket hosts similar number of objects. On the other hand, HashFile takes advantage of linear scan, which is more efficient than random access used in LSB-tree.

VIII. Experiments

In this section, we study the performance of HashFile and compare it with state-of-the-art approaches using real image data sets. Both exact and approximate NN query processing algorithms are evaluated. All the experiments are conducted on a server with Quad-Core AMD Opteron(tm) Processor 8356, 128GB memory, running Centos 5.4.

A. Data Set and Query

We use **NUS-WIDE** [8] as the image data set, which contains $269,648$ web images from Flickr. Two types of image features widely used in the image retrieval applications are extracted:

1) **Color Histogram.** In image processing and photography, a color histogram summarizes the distribution of colors in an image. The color space is quantized into a set of bins and the value of each bin represents the number of projected pixels by color. In practice, LAB color space [26] is normally selected as the candidate because its space is linear and suitable for quantization. In our experiment, we extract 64-dimensional color histograms in LAB space from the image data set.
2) **SIFT.** The SIFT [21] descriptor has been widely used in image retrieval and object recognition due to its invariance with respect to translation, scaling, rotation and small distortions. Each feature is a 128-dimension vector extracted from $4*4$ subregions around the key point and each subregion is approximated by an 8-bin histogram of the image gradients.

We split the color histogram data set into two parts D_1 and Q_1 without intersection. 100 color histograms were randomly selected as the query. The leftover data are used as the underlying database. Similarly, we extract 10 million SIFT

features from the Flcikr photos for indexing and another 200 for query.

B. Performance Measurement

The goal of the experiments is to show that the performance of our index is better than state-of-the-art query processing methods in answering exact and approximate NN queries in the high dimensional vector space. Before we present the experiment results, we first address the performance measurement used in our experiments to judge the superiority of an index.

Due to the curse of dimensionality, the pruning power degrades in the high dimensional space, resulting in a large candidate set for the NN result. We need to load these candidates into memory and calculate their distance to the query object. Both disk I/O cost and CPU computation time play an important role in the query processing stage. Since HashFile takes advantage of sequential scan, it may be unfair to measure the I/O cost simply by the number of page access or disk access. To make a fair competition, we use average response time (ART) to measure the performance of an index in answering an exact NN query.

$$ART = (1/N_Q)\sum_{i=1}^{N_Q}(T_f - T_i)$$

where T_i is the time at which the query was issued, T_f is the time of query completion and N_Q is the total number of times exact NN query was issued. The ART is equivalent to the elapsed time including both I/O and CPU cost. Besides the ART, we also report the selectivity of different indexes in the query processing. If we assume the time cost of calculating the distance between a candidate to the query object is fixed as t, the CPU cost can be measured by the selectivity.

$$C_{cpu} = N * s * t$$

, where N is the data size and s is the average query selectivity.

In the approximate NN query, the quality of the result is usually represented by the distance ratio between the approximate NN and the exact NN. For top-k approximate NN query, we can adopt the same metric as in [27]:

$$\begin{aligned} R_i(q) &= \frac{\|o_i, q\|}{\|o_i^*, q\|}, 1 \le i \le k \\ R(q) &= \frac{\sum_{i=1}^{k} R_i}{k} \end{aligned}$$

where o_i is i-th approximate neighbor and o_i^* is the exact neighbor. Obviously, the smaller the value of $R(q)$, the better is the quality of results retrieved via approximate NN query. If $R(q)$ equals to 1, the query results are exact. However, the distance recall is not enough to measure the performance of an index as there is a trade-off between the access cost and the result quality. If more data are accessed in the buckets, a smaller $R(q)$ can be retrieved. Hence, we need to consider both factors in the performance measurement. Since LSB-tree uses random access and our index takes advantage of sequential scan, we still use ART to measure the access cost. We plot the curve of $R(q)$ as the y axis with respect to the ART as the x axis. The curve drops down as the ART increases, meaning that $R(q)$ is getting close to 1. We use the area under the curve as the measurement. If an index can retrieve an approximate result closer to the exact neighbors in less running time, we consider it to be superior.

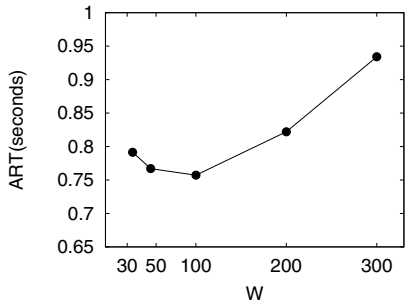


Fig. 6. Tune parameter W

C. Parameter Tuning

In our first experiment, we study how the performance varies with the only parameter W. The page size B is fixed as 100. We randomly select a subset with 1 million SIFT descriptors from D_2 as the data set and test the performance of the exact NN search algorithm.

The result in Figure 6 shows that the performance degrades when W is set too large or too small. If W is chosen to be very large, a lot of data points will be hashed into the same bucket. It becomes easy for a bucket to be overloaded and generate new child nodes. Therefore, the tree becomes higher and more random accesses are needed to retrieve the tree nodes. Also, there are fewer disk pages in each node's file. The advantage of sequential scan is offset in this case. On the other hand, if W is set to be very small, the strategy of further partitioning the dense bucket becomes useless. The number of points in the buckets is small as well and the re-hashing does not take much effect.

D. Frequent Insertion

HashFile needs to maintain the sequential order of the disk pages based on the hash value. When a split operation occurs in a full page p_i, we need to move the pages $\{p_j | j > i\}$ backwards to make room for the new page. Similarly, when p_i contains objects with the same hash value and can not be split using the node's hash function, we need to delete p_i from the file and move forward the pages $\{p_j | j > i\}$ that are behind to fill the gap. We consider the page movement as a sequence of two disk operations: the block of pages $\{p_j | j > i\}$ is first read into the memory and then written to the target location in the disk file. The I/O cost is determined by the size of the block to be moved.

Figure 7 shows an example of how the number of disk pages in the root varies when data points are continuously inserted. In this example, we use the color histogram data set and set the

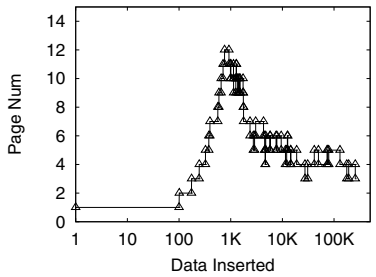


Fig. 7. The number of pages in the root node

page size to 100. From this curve, we can tell that the number of pages in the data file is always in a small scale. The first split occurred when the 101-st point arrived. After that, the pages in the root were split frequently and the number of pages in the file increased dramatically. As more data points were inserted, the buckets near the mean value could not be split anymore. These buckets were extracted from the file and re-hashed into the child nodes. Thus, the number of pages started to decrease. Finally, this number became relatively stable, varying between 3 to 6, because the buckets near the mean hash value have been deleted and new objects were more likely to be inserted into the child nodes. Therefore, the file size would not grow too large and cause too much I/O overhead in the page movement.

E. Exact NN Query

In this experiment, we compare HashFile with linear scan, VA-file, and iDistance. The parameter setting for each type of index is shown in Table II. We tune the bit number for each dimension in the VA-file in the range of $[4, 6]$ and select the best one to build the index. The number of reference points in iDistance is set to $2d$. The window size of W in HashFile is also set differently according to the data set. The exact NN query is executed on both the color histogram and the SIFT descriptor. We gradually increased the data set in D_1 from $180,000$ to $260,000$ and in D_2 from 2 million to 10 million.

TABLE II
PARAMETER SETTING

	VA-File	iDistance	HashFile
D_1	bit=5	ref=128	W=20
D_2	bit=4	ref=256	W=100

Table III shows the disk storage used for each of the three types of index. VA-file takes up the least storage cost. Besides the real data points, it only maintains the bitmap summary which can be stored efficiently. iDistance needs to build an additional B^+-tree and store the real data in the leaf entries. Since the page can not be fully utilized, it takes much more storage cost than VA-file. HashFile consumes slightly larger storage than iDistance. But it is still under the bound $O(2N + N/B)$.

The pruning power of the three types of index in answering exact top-50 NN queries is reported in Table IV. It counts the proportion of the real data points accessed in the query processing stage. VA-file has the best selectivity because the data space is split into exponential number of small cells and the real data points are tightly approximated by the cells. After comparing the query point to the bitmap file, only a very small number of real data need to be accessed. The pruning power of iDistance is mainly determined by the selection of the reference points. When we use the cluster center to partition the space into Voronoi cells, iDistance demonstrates a better pruning power than HashFile. The pruning of iDistance is based on real distance while HashFile is based on the lower bound distance constraint in the hash space. All the data points in the same page are sequentially scanned without any pruning. As d increases from 64 to 128, we can see that the pruning power of VA-file becomes even better. However, iDistance and HashFile need to access the majority of the data set.

TABLE III
INDEX STORAGE COST

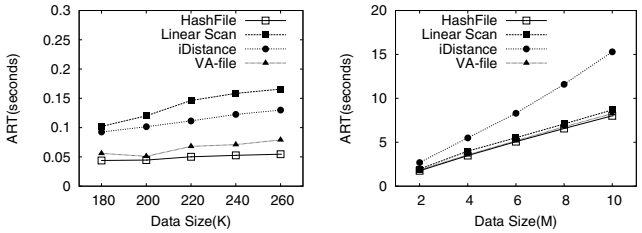
Color Histogram(MB)					
Data Size	180K	200K	220K	240K	260K
VA-file	49.4	54.9	60.4	65.9	71.4
iDistance	72	80	87	95	104
HashFile	74	82	89	96	104
SIFT Descriptor(GB)					
Data Size	2M	4M	6M	8M	10M
VA-file	1.125	2.25	3.375	4.5	5.625
iDistance	1.6	3.1	4.6	6.1	7.6
HashFile	1.8	3.5	5.3	7.0	9.2

TABLE IV
TOP-50 NN QUERY SELECTIVITY

Color Histogram					
Data Size	180K	200K	220K	240K	260K
VA-file	0.00281	0.00275	0.00269	0.00264	0.00260
iDistance	0.217	0.205	0.2	0.193	0.182
HashFile	0.301	0.291	0.285	0.278	0.268
SIFT Descriptor					
Data Size	2M	4M	6M	8M	10M
VA-file	0.00034	0.00024	0.00021	0.00019	0.00018
iDistance	0.841	0.769	0.696	0.645	0.599
HashFile	0.918	0.911	0.902	0.889	0.867

The average running time to answer the queries is shown in Figure 8. Since the color histogram data set is skewed, iDistance shows good pruning power and its performance is better than linear scan. But when it comes to the SIFT data set, which is more uniformly distributed, iDistance needs to access the majority of the pages and performs worst. Such a large amount of random access makes the index I/O bound. VA-File outperforms linear scan in both two data sets. It has extremely low selectivity in the query processing and the operations to calculate the minimum and maximum bound distance from the query point to the cells are optimized in the implementation to greatly save the CPU cost. HashFile adopts random projection to gain the pruning power and takes advantage of sequential scan to reduce the number of random access. It achieves significant superiority over the other index structures in the

color histogram data set, which is skewed to black and white colors. In the SIFT data set, the data is more uniformly distributed and HashFile needs to access a large population of the data. Comparing Figure 8(b) with Figure 8(b), we can see that the performance of high dimensional index start to degrade to linear scan as the dimensionality increases. This makes approximate NN search an appealing method to handle similarity search in a large scale multimedia database.



(a) Color Histogram (b) SIFT

Fig. 8. Performance of the exact top-50 NN search

F. Approximate NN Query

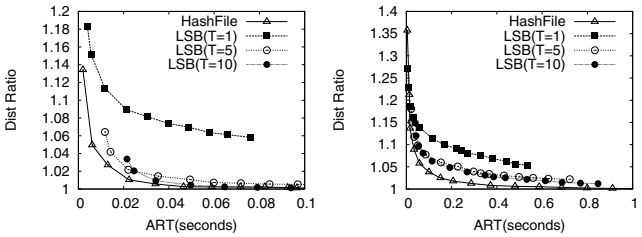
In this experiment, we test the performance of HashFile in answering the approximate NN query. We use the LSB trees [27] as the comparison method. In order to plot the curve of $R(q)$ with respect to the access cost for the LSB trees, we do not terminate the search algorithm based on the two conditions proposed in [27]. Instead, we set a parameter I as the number of iteration to execute. We gradually increase I so that more and more leaf entries will be accessed. With the increasing access cost, the algorithm returns the nearest neighbors with smaller $R(q)$. In this way, we can plot how the quality of the result varies with the running time. As to HashFile, we set a small window and gradually increase the parameter λ. Each time λ is increased, we can access more disk pages to find a better nearest neighbor. The experiment is conducted on D_1 with $260,000$ color histograms and D_2 with two million descriptors.

TABLE V
STORAGE COST OF HASHFILE AND LSB FOREST

	Color Histogram	SIFT Descriptor
HashFile	110MB	1.6GB
LSB(1 tree)	107MB	1.6GB
LSB(5 trees)	539MB	8GB
LSB(10 trees)	1094MB	16GB

We computed the overall disk storage costs for the LSB forest with varying number of trees. Table V depicts the storage cost for different variants. The size of HashFile is basically the same with that of one LSB tree. The size of the LSB-forest grows linearly with the number of trees. In particular, when 10 trees are used, the storage cost of LSB is almost 10 times that of HashFile.

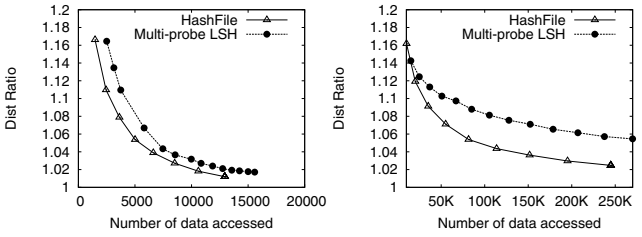
Figure 9 shows the tradeoff between $R(q)$ and ART on the top-20 nearest neighbors in D_1 and D_2 respectively. As more data are accessed, $R(q)$ declines to be near 1, indicating that better results are found. We can clearly tell from the figure that HashFile has significant superiority than LSB forests. Even though LSB forests consume much more storage cost than HashFile, when the number of trees increase to 10, its performance is still dominated by HashFile. Given a point in the curve of 10 LSB trees, we can always find another point in HashFile curve so that the query is processed with less running time but with better $R(Q)$. This validates the superiority of linear scan to random access.



(a) Color Histogram (b) SIFT

Fig. 9. Approximate NN query of LSB-tree and HashFile

Finally, we compare the search quality of HashFile with LSH. In HashFile, only the dense buckets will be further partitioned and the data points are associated with hash values of variable length. In contrast, the length is fixed as m when LSH is used. We use multi-probe LSH [2] as the comparison method because it is also adhoc and without theoretical guarantee. In this experiment, we set $m = \kappa$, i.e., the number of hash functions m is equal to the tree height of HashFile. We also tuned the parameter W in these two indexes to generate roughly the same number of buckets. Since multi-probe LSH is an in-memory index, the search quality is measured by the distance ratio with respect to the number of data accessed. We gradually increase the number of probe in multi-proble LSH and λ in HashFile to expand the search space. The experiment results on the color histogram and SIFT data sets are shown in Figure 10. Since the distribution of bucket size in LSH is much more skewed than HashFile, many false positives still exist in the dense buckets. Accessing these false positives will reduce the search quality. HashFile has more balanced bucket size and demonstrates better search quality.



(a) Color Histogram (b) SIFT

Fig. 10. Approximate NN query of Multi-probe LSH and HashFile

[2]http://lshkit.sourceforge.net/

IX. Conclusion

In this paper, we proposed a novel index structure, HashFile, to handle nearest neighbor queries in the high dimensional vector space which are needed to support object retrieval in multimedia databases. The index can support approximate NN search in the Euclidean space and exact NN search in L_1 norm. Users can select the query strategy based on their applications. HashFile is simple in structure and therefore is easy to be implemented into the real system and applications. It provides better efficiency in processing both the two types of NN queries. Experiments were conducted on real data sets to establish the superiority of Hashfile over other state-of-the-art index structures.

Acknowledgment

The research and system development reported in this paper was supported by the Singapore National Research Foundation Interactive Digital Media R&D Program, under research Grant NRF2008IDM-IDM004-047.

References

[1] D. Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.

[2] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 420–434, London, UK, 2001. Springer-Verlag.

[3] S. Berchtold, C. B02hm, H. V. Jagadish, H.-P. Kriegel, J. Sander, and J. S. Independent quantization: An index compression technique for high-dimensional data spaces. In *In ICDE*, pages 577–588, 2000.

[4] S. Berchtold, C. Böhm, and H.-P. Kriegal. The pyramid-technique: towards breaking the curse of dimensionality. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 142–153, New York, NY, USA, 1998. ACM.

[5] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 28–39, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[6] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, New York, NY, USA, 2001. ACM.

[7] R. Cai, C. Zhang, L. Zhang, and W.-Y. Ma. Scalable music recommendation by search. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 1065–1074, New York, NY, USA, 2007. ACM.

[8] T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y.-T. Zheng. Nus-wide: A real-world web image database from national university of singapore. In *Proc. of ACM Conf. on Image and Video Retrieval (CIVR'09)*, Santorini, Greece., July 8-10, 2009.

[9] W. Dong, Z. Wang, M. Charikar, and K. Li. Efficiently matching sets of features with random histograms. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 179–188, New York, NY, USA, 2008. ACM.

[10] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[11] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, New York, NY, USA, 1998. ACM.

[12] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[13] W. Johnson and J. Lindenstrauss. Extensions of lipschitz mapping into hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[14] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 369–380, New York, NY, USA, 1997. ACM.

[15] Y. Ke, R. Sukthankar, L. Huston, Y. Ke, and R. Sukthankar. Efficient near-duplicate detection and sub-image retrieval. In *In ACM Multimedia*, pages 869–876, 2004.

[16] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 599–608, New York, NY, USA, 1997. ACM.

[17] N. Koudas, B. C. Ooi, H. T. Shen, and A. K. H. Tung. Ldc: Enabling search by partial distance in a hyper-dimensional space, 2004.

[18] Y.-H. Kuo, K.-T. Chen, C.-H. Chiang, and W. H. Hsu. Query expansion for hash-based image object retrieval. In *MM '09: Proceedings of the seventeen ACM international conference on Multimedia*, pages 65–74, New York, NY, USA, 2009. ACM.

[19] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 614–623, New York, NY, USA, 1998. ACM.

[20] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: an index structure for high-dimensional data. Technical report, College Park, MD, USA, 1994.

[21] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[22] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[23] J. T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, New York, NY, USA, 1981. ACM.

[24] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 516–526, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[25] B. Stein. Principles of hash-based text retrieval. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 527–534, New York, NY, USA, 2007. ACM.

[26] G. Stockman and L. G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[27] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 563–576, New York, NY, USA, 2009. ACM.

[28] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[29] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.