Time Efficient and Quality Effective K Nearest Neighbor Search in High Dimension Space

by

Renwei Yu

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2011 by the
Graduate Supervisory Committee:

Kasim Selçuk Candan, Chair
Maria Luisa Sapino
Yi Chen
Hari Sundaram

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

K-Nearest-Neighbors (KNN) search is a fundamental problem in many application domains such as database and data mining, information retrieval, machine learning, pattern recognition and plagiarism detection. Locality sensitive hash (LSH) is so far the most practical approximate KNN search algorithm for high dimensional data. Algorithms such as Multi-Probe LSH and LSH-Forest improve upon the basic LSH algorithm by varying hash bucket size dynamically at query time, so these two algorithms can answer different KNN queries adaptively. However, these two algorithms need a data access post-processing step after candidates' collection in order to get the final answer to the KNN query. In this thesis, Multi-Probe LSH with data access post-processing (Multi-Probe LSH with DAPP) algorithm and LSH-Forest with data access post-processing (LSH-Forest with DAPP) algorithm are improved by replacing the costly data access post-processing (DAPP) step with a much faster histogram-based post-processing (HBPP). Two HBPP algorithms: LSH-Forest with HBPP and Multi-Probe LSH with HBPP are presented in this thesis, both of them achieve the three goals for KNN search in large scale high dimensional data set: high search quality, high time efficiency, high space efficiency. None of the previous KNN algorithms can achieve all three goals. More specifically, it is shown that HBPP algorithms can always achieve high search quality (as good as LSH-Forest with DAPP and Multi-Probe LSH with DAPP) with much less time cost (one to several orders of magnitude speedup) and same memory usage. It is also shown that with almost same time cost and memory usage, HBPP algorithms can always achieve better search quality than LSH-Forest with random pick (LSH-Forest with RP) and Multi-Probe LSH with random pick (Multi-Probe LSH with RP). Moreover, to achieve a very high search quality, Multi-Probe with HBPP is always a better choice than LSH-Forest with HBPP, regardless of the distribution, size and dimension number of the data set.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all the important people who help me to finish this thesis.

I am grateful to my advisor Dr. Kasim Selçuk Candan for his great help in both academia and research in my three years graduate study at Arizona State University. I would like to thank Dr. Maria Luisa Sapino, Dr. Yi Chen, Dr. Hari Sundaram, for joining my graduate committee and giving me invaluable guidance on my thesis research.

I would like to thank Xinxin, Wei, Shruti, Mijung, Mahsa, Parth, Mithila, Jung Hyun Kim, Jong Kim, Yan, my friends and colleagues in Emitlab. They help me so much during my research work. I would also like thank my friend Shanshan Liang, for her great patient and passion in the proofreading and editing of my thesis draft.

Finally, I would like to thank my parents, my sister, my brother who always support and encourage me in pursuing my graduate degree.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Finding the K-Nearest-Neighbors (KNN) [33] of a query point is a fundamental problem in many application domains. Some examples are data mining [20], machine learning [9], information retrieval [11, 14, 32], multimedia database [13, 31], bioinformatics [25, 16], and duplicate document detection [34, 8]. In the vector model, the object of interest is usually represented as a point in the R^d space, where d is the number of features of the object we are interested in [17]. The number of features can range anywhere from a few hundreds to several thousands. A distance metric of these points can be used to quantify the dissimilarity of objects, so the similarity of objects is inversely related to the distance.

The KNN (K-Nearest-Neighbors) [12] problem has been studied for decades. It can be defined as follows: given a collection of data points M, a query point Q, return the result set, R, of data points from the collection which satisfy the following condition: $|R| = K$ and if $a \in R, b \in M$ and $b \notin R$, then $F(Q,a) \leq F(Q,b)$. Here $F()$ is the distance function selected by the user. For example, $F()$ can be the Euclidean

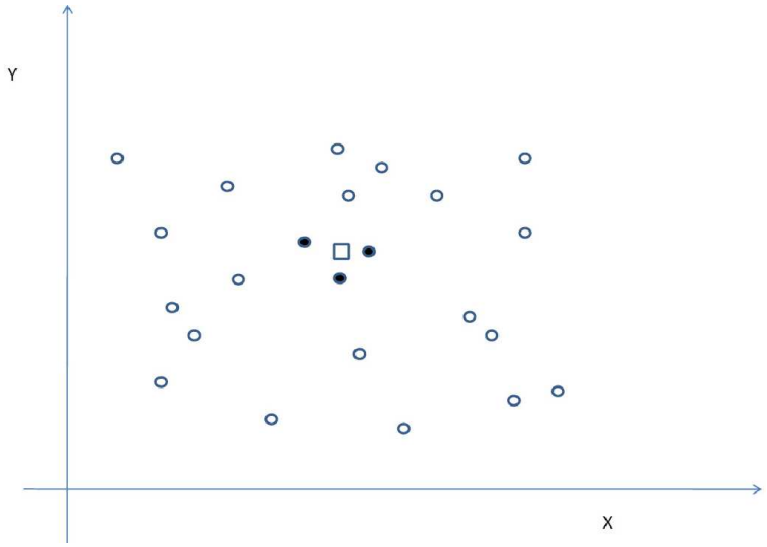

Figure 1.1: 3 black circle points are 3-nearest neighbors of query point based on Euclidean distance (circle points represent data set and rectangle point is the query point)

distance, as shown in Figure 1.1.

1.1 Challenges

Large Data Size Challenge

One naive way to execute a KNN query would be to simply compute the distances from the query point to all the n points in the data set (n is the size of the data set) and pick the closest K points. Obviously, this naive algorithm will be very costly, and it is only feasible when the data size is small enough to fit into main memory (when the size is large, the data set can only be stored on disk, so both the distance computation cost and the disk I/O cost will be huge). However, the size of the data collections nowadays is increasing vastly. Consequently, the data size we need to deal with is becoming larger and larger; from several GBs to TBs (it means millions to billions of high dimensional data vectors/points). Given such a large data size, efficient index structures are needed in order to prune unpromising data points as early as possible and return the KNN within a reasonable amount of time.

High Dimension Challenge

Pruning of unpromising data points often requires the support of index structures. There already exist many index structures, such as linear indexing (space filling curve [21]), high-dimension space partition (KD-tree [5], R-tree [19], SR-tree [23]) and locality sensitive hashing (LSH [22]). However, as the dimension of the data objects increases, algorithms based on many of the existing index structures are not much better than a naive algorithm that enumerates all $O(n)$ data points and returns the K closest points as the result. The reason is that in the high dimensional case, a large proportion of the data points/objects in the space needs to be considered as candidates of KNN. As shown in [36], when the number of dimensions exceeds 10, existing space partition index structure are slower than the brute-force sequential scan algorithm. This phenomenon is called the "*curse of dimensionality*" [4]. We will discuss these issues in detail in Chapter 2.

1.2 Thesis Contribution

As mentioned in [22], all known techniques for solving the exact KNN problem in high dimensional space fail due to the "*curse of dimensionality*", so the authors of [22] tried to solve approximate KNN instead. The definition of the approximate KNN is as follows: assume the maximum distance from the query object to the objects of the exact KNN result is r ($r = max\{Dis(q,p)\}$, $Dis()$ is the distance function, q is the query object, $p \in$(exact KNN result)), so the exact KNN objects are in the sphere with radius r in the high dimensional space, we want to return K objects within distance $(1+\varepsilon) \times r$ from the query point as the answer of approximate KNN query. In the definition, $\varepsilon \geq 0$ and the smaller ε is, the better the search quality. When $\varepsilon = 0$, the approximate KNN is the same as exact KNN. In [22], the authors proposed the well known algorithm called locality sensitive hashing (LSH, we will explain it in more detail in Chapters 2 and 3) to solve the approximate KNN problem efficiently.

The key idea of LSH is that objects that are close to each other are more likely to be hashed to the same bucket in the hash table than objects that are far apart [22]. So far, LSH is the most practical algorithm for high dimensional space KNN search problem, even though it is an approximate algorithm. However, one weakness of this basic LSH algorithm is that many parameters need to be tuned carefully in order to answer a domain specific KNN query. In the KNN search problem, we can assume there is a hyper-ball around query object within which there will be K objects as its KNN. For different KNN queries, the radius of the hyper-ball will vary for two reasons: (1) the data distribution around query object, (2) K, the size of the query result. The change of the radius may render the previous setup parameters useless, and subsequently render LSH to be unfeasible in terms of real world application where there exists a large variety of queries. In order to eliminate the parameter tuning requirement of the basic LSH algorithm, in [3] Bawa *et al.* proposed the LSH-Forest algorithm, in which the most

important improvement is the use of the prefix tree data structure to index the LSH signature of each data object. In essence, a leaf tree node is like a hash bucket in the basic LSH, and each level of the tree represents a different hash table (with a different bucket size). To answer a KNN query q, this algorithm descends each LSH tree to find the leaf having the largest prefix match with q's LSH signature, and then ascends synchronously in each LSH tree to check the possible siblings until it collects enough candidate nearest neighbor objects [3]. Inspired by the idea of LSH-Forest, we find that Multi-Probe LSH [27] can also be used to answer the KNN queries in a self-tuning manner. The reason is as follows: the main idea of LSH-Forest is using the LSH tree to vary bucket size in a query adaptive way; similarly, Multi-Probe LSH is also able to vary bucket size dynamically at query time (they achieve this goal via probing multiple relevant buckets, the union of which can be viewed as a single bigger bucket).

An ideal KNN search algorithm should be able to achieve 3 goals: high search quality, high time efficiency, high space efficiency. Although some above mentioned algorithms can achieve high search quality and high space efficiency, they failed to achieve high time efficiency because of a main drawback: they all require a costly post-processing step. For algorithms such as LSH-Forest and Multi-Probe LSH, after a sufficient number of candidate objects are collected, they need to be ranked by the distance to the query object, then the K objects that are most similar (thus closest) to the query point are returned as the KNN result to the query. In order to get the distance of each candidate object to the query object, we need to fetch the full vector of each candidate object from some data storage device (e.g., database), and compute the distance according to the distance function (for example, l_1 distance or l_2 distance). The vector fetching step can be costly. For high dimension data, there is a possibility that the data size is too large to be stored in main memory. For example, one of our experiment data sets with only 1 million vectors has the size of 8.1 GB (960 dimension GIST data stored in text format) [18]. In order to fetch a data object vector from a disk-

based index (e.g., BerkeleyDB [29] database in our case), we have to pay $O(\log N)$ disk I/O price (N is the number of data objects in the data set). When the candidate set size is not small, this disk read cost is not negligible. Moreover, the distance computation step can also be costly, especially when the number of dimensions is high or the distance function is costly. Furthermore, in some cases the vectors of data objects may not be visible at all for security/privacy reasons, rendering the data access post-processing step impossible.

The weaknesses of all previous KNN related algorithms is the costly post-processing step, hence we think it is necessary to find a more time efficient post-processing solution without visiting candidate objects vectors in data storage device at all. Here we propose our histogram-based post-processing (HBPP) method. The logic behind HBPP is that in each hash table, objects that are close to the query object are more likely to appear in the corresponding bucket than objects that are far apart from the query object. In the candidate set, which is the combination (not union in this case) of all the fetched buckets from each hash table, objects that appear more times are very likely to be closer to the query object than objects that appear fewer times. The work flow of our HBPP method is as follows: after the candidate objects are collected, we order them according to the occurrence count of each object (in a sense, it is very similar to building an occurrence histogram, as shown in Figure 1.2). Then we collect objects from the highest occurrence slot, then the next highest occurrence slot and so on, until we get K nearest neighbors. If the last slot from which we collect KNN objects has more objects than required, we will just randomly pick the required number of objects from that slot. For example, in Figure 1.2, if we want to pick 100 objects that are closest to the query object, we will collect all the objects with occurrence count 10 and 9, furthermore, we will randomly pick 34 objects from objects with occurrence count 8, thus we get 100 objects in total as the KNN result. Finally, we return these K objects as an answer to the KNN query. We will explain our algorithm in more detail in Chap-

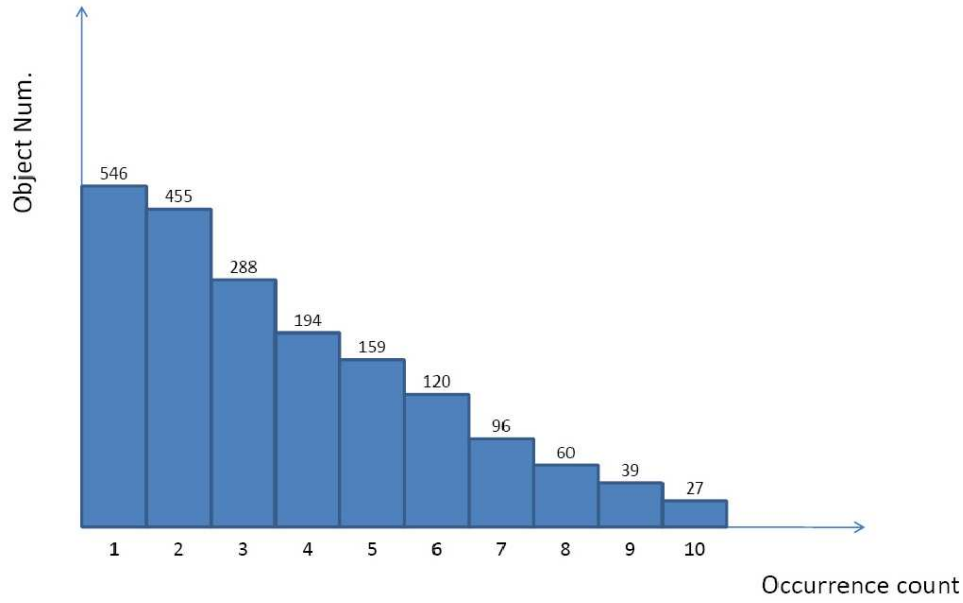

Figure 1.2: Object occurrence histogram in which the numbers at the bottom of each bar denote occurrence count; numbers on the top of each bar denote number of objects with the corresponding occurrence count in the candidate set (e.g., the first bar in the chart has the number 1 in bottom and number 546 on the top, it means there are 546 objects that occur once in the candidate set)

ter 4. In the experimental results chapter, we compared our algorithms (Multi-Probe LSH with Histogram-based post-processing (HBPP) and LSH-Forest with HBPP) with several other LSH based KNN algorithms, including Multi-Probe LSH with data access post-processing (DAPP), LSH-Forest with DAPP, Multi-Probe LSH with random pick(RP), LSH-Forest with RP. We have shown that our algorithms can always achieve high search quality (as good as LSH-Forest with DAPP and Multi-Probe LSH with DAPP) with much less time cost (one to several orders of magnitude speedup). We have also shown that with almost same time cost and memory usage, our algorithms can always achieve better search quality than the LSH-Forest with RP and Multi-Probe LSH with RP.

1.3 Organization of the Thesis

The organization of this thesis is as follows: we discuss related work in Chapter 2. Chapter 3 presents the overview of locality sensitive hashing (LSH) for nearest neighbors search, and other improved techniques based on LSH such as Multi-Probe LSH

and LSH-Forest. We will discuss our proposed algorithms in Chapter 4 and present the mathematical model that explains HBPP theoretically. Chapter 5 describes the experimental setups which evaluate different algorithms for multiple data sets. Chapter 6 discusses the experimental results. We conclude this thesis in Chapter 7.

Chapter 2

RELATED WORK

The most important problem for multi-dimensional query processing is how to index the high-dimensional objects. One straightforward and intuitive way would be mapping the objects from a multidimensional space into one dimension, and indexing them with a data structure like B-tree.

2.1 Linear Indexing

Space-filling curves such as Hilbert Curve [21] (introduced by Hilbert in 1891) and Z-order Curve [28] will be appropriate methods. Although Hilbert curve fills the space effectively, it is not computationally efficient. So instead, a Z-order curve that has a very efficient mapping implementation, is used in practice for high dimensional space.

The z-value of a point is obtained by interleaving the binary representations of its coordinate values [38]. For example, given a point (3, 5) in a 2-d space, its binary representation is (011,101). Hence, its z-value is 011011=27. The Z-order curve for a data set S is generated by connecting the points in S in the descending order of their z-values [38].

In [38], the authors utilize the z-values to map points in a multidimensional space into one dimension, and then translate the KNN search for a query point q into one dimensional range search on the z-values around the q's z-value. To avoid the information loss in the mapping, they produce some independent, randomly shifted copies of the original data set. Then they do the same operation on the shifted data set (mapping and range query). By doing this, the approach theoretically guarantees to provide a constant factor approximate (in terms of the radius of the K nearest neighbor ball) solution, and it could be extended to find the exact KNN efficiently in any fixed dimensions.

However, the weakness of space-filling curves is that when the multi-dimensional data objects are mapped to one dimensional space and stored using traditional index structures, some amount of information is lost (for example, the jumps in Z-order curve), which may result in misses or false positives during query processing. Another way to index the high-dimensional data points would be to apply a multi-dimensional indexing method to index the original space directly.

KD-tree, which was introduced by Jon Bentley [5] in 1975, is the first high-dimensional indexing structure used to solve the KNN problem. KD-tree always splits the space along one single dimension every time. To ensure that every dimension gets the chance to divide the space, the dividing dimension is selected at each level of the KD-tree in a round-robin manner. A KNN query can be processed efficiently in a KD-tree by using a branch and bound method to quickly eliminate large portions of the search space. For a KD-tree and its variants, they build the indexing structure in a top-down manner. But this top-down method results in the dead space problem [19], because the entire space needs to be covered at each level of the tree even if some portion of the space is totally empty [7]. To solve the dead space problem, [19] introduced R-tree in 1984, which uses a bottom-up data-partition strategy instead of space partition method. At the leaf level, minimum bounding region (MBR) [19] is used to cluster nearby data points together. Then in the higher level node, larger MBR is formed by clustering nearby lower level MBRs. The highest level is the root node, the only level which has to cover the entire space [7]. Actually, R-tree is a tree data structure very similar to the well known B-tree, but is used in the spatial access case, i.e., for indexing multi-dimensional data objects. In [19], the algorithm uses depth-first (DF) traversal paradigm, starts from the root of R-tree, and visits recursively the node with the smallest mindist to q (query point) until the leaf level where a potential NN (Nearest

Neighbor) exists is reached. Subsequently, the algorithm conducts backtrackings. In particular, during backtracking to the higher levels, DF only visits those nodes whose minimum distance to q is smaller than the distance between the NN candidate retrieved so far and the query point. In other words, R-tree is used to prune a large portion of unpromising data points quickly.

2.3 Locality Sensitive Hashing

In many modern applications, the number of features (dimensions) can be very large [13, 15]. As the number of dimensions increases, all the algorithms mentioned above become less effective. The reason is that a large proportion of the objects in the space needs to be compared to the query point in this case, so these algorithms are not much better than the sequential scan algorithm that compares all the objects in the space to the query point. This phenomenon is called the "*curse of dimensionality*" [4]. To deal with this issue, Locality Sensitive Hashing (LSH) [22] is introduced.

Locality sensitive hashing (LSH) [22] is a technique for grouping points in space into "*buckets*" based on some distance function. We will give more details about LSH in Chapter 3. Points that are close to each other under the chosen metric are mapped to the same bucket with high probability. The key idea is to hash the points using several hash functions to ensure that for each function the probability of collision is much higher for objects that are close to each other than for those that are far away, in [17], the authors provide theoretical proof. So nearest neighbors can be obtained by hashing the query point and fetching points stored in the corresponding buckets. LSH is an approximate algorithm.

Multi-Probe LSH

One disadvantage of the basic LSH algorithm is that it requires a huge amount of hash tables in order to achieve good search quality (or low miss rate). In [27], the authors propose a new indexing scheme called Multi-Probe LSH to overcome this disadvantage. Multi-Probe LSH is based on the well-known basic LSH algorithm, but it intelligently probes multiple buckets (within the same hash table) that are likely to contain nearest neighbors of the query object. Thus Multi-Probe LSH requires much fewer hash tables to achieve the same search quality (recall rate). We will give more details about Multi-Probe LSH in Chapter 3.

LSH-Forest

Another significant drawback of the basic LSH algorithm is that it requires hand-tuning for many of its data dependent parameters in order to answer specific KNN queries (e.g., specific data domain, specific result size). So Bawa *et al.* [3] proposed a self-tuning indexing scheme called LSH-Forest which is applicable for KNN search in high-dimensional space. LSH-Forest is also based on the well-known technique of locality-sensitive hashing (LSH), but improves upon it in several perspectives: (a) it gets rid of the need for hand-tuning of many data-dependent parameters in basic LSH algorithm, (b) it has better performance compared to basic LSH for skewed data distribution, with same space and time complexity [3]. LSH-Forest algorithm achieves these improvements by indexing hash signatures of objects (of each LSH function) with prefix-tree, instead of a hash table. As a result, the signature length of each object is able to vary at different levels of the prefix tree. In other words, one prefix tree can represent multiple hash tables with each node corresponding to one bucket. So the LSH-Forest algorithm can answer a specific KNN query adaptively by traversing from the leaf node toward the root node in each prefix tree simultaneously until enough candidate objects are collected to answer the KNN query.

2.4 Alternative Methods

Another algorithm to beat the "*curse of dimensionality*" [4] is called Vector Approximation Files [37]. Observing that most existing algorithms are not better than sequential scan method in the high dimension case, it uses a memory stored reduced space granularity version of the data set to speed up the scan in the first run. Then in the second run, it fetches the complete vectors of the candidates from disk for post-processing. Vector Approximation Files is also an approximate algorithm.

Chapter 3

OVERVIEW OF LSH BASED KNN SEARCH ALGORITHMS

This chapter reviews the existing LSH-based indexing schemes. We first explain the key idea of the basic LSH algorithm, then we introduce two improved versions of the LSH scheme: one is called Multi-Probe LSH, which reduces the space complexity compared to the basic LSH algorithm; another is called LSH Forest, which is able to eliminate the data dependent parameter tuning step of the basic LSH algorithm.

3.1 Basic Idea of LSH

Locality sensitive hashing (LSH) functions map "*close*" objects into the same bucket with higher probability than "*far apart*" objects. Given a KNN query, steps to answer the query using LSH index are: (1) use LSH functions to hash the query object into a corresponding bucket in each hash table, (2) collect objects from the corresponding bucket in each hash table, then merge all these objects to form the candidate set, (3) rank all the candidates according to their distances to the query object, and return the K closest objects as the answer [27].

In [22], Indyk and Motwani proposed the idea of locality sensitive hashing (LSH) for the first time. By using LSH functions, two objects that are "*close*" to each other are more likely to collide than two "*far apart*" objects. The definition of LSH functions is as follows [27]:

Definition 1. A function family $H = \{h : S \to U\}$ is called (r_1, r_2, p_1, p_2)-sensitive for distance function $Dis()$, if for any $p, q \in S$ (here S is the data set, $r_1 < r_2$, $p_1 > p_2$)

If $Dis(p,q) \leq r_1$, then $Pr_H[h(p) = h(q)] \geq p_1$

If $Dis(p,q) \geq r_2$, then $Pr_H[h(p) = h(q)] \leq p_2$

By using such functions, we can ensure that "*close*" objects (within distance r_1)

are more likely (since $p_1 > p_2$) to have the same hash value than "*far apart*" objects (with distance more than r_2). More generally, the collision probability of two objects p and q decreases strictly with the distance between them.

The distance function $Dis(a,b)$ we mentioned above can vary, as long as it is l_p norm (e.g., Hamming distance, Manhattan distance, Euclidean distance), as shown below:

$Dis(a,b) = \{\sum_{i=1}^{d}(a_i - b_i)^p\}^{1/p}$, here a and b are two points in the d-dimensional space, and a_i is the coordinate value of point a in the $i-th$ dimension, b_i is defined similarly.

The LSH family that is applicable for l_p norm distance is based on p-stable distributions [10]. This point was first proposed by Datar *et al.* Some well know p-stable distributions are [1]:

$\diamond$ Cauchy distribution, defined by the density function $f(x) = \frac{1}{\pi}\frac{1}{1+x^2}$; it is 1-stable

$\diamond$ Gaussian (normal) distribution, defined by the density function $f(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$; it is 2-stable

Locality sensitive hash functions for l_p norm usually have the following format:

$H_{A,B}(\vec{v}) = \lfloor \frac{\vec{A}\cdot\vec{v}+B}{W} \rfloor$

Here, $\vec{v}$ is the corresponding vector of some data object/point in the high dimensional space, $\vec{A}$ is a d-dimensional vector whose entries are chosen randomly from a p-stable distribution, and B is a real number chosen randomly from the range $[0, W]$ [1]. In this thesis, the distance function is Euclidean distance (l_2 norm), so we will use a 2-stable distribution (normal distribution) to generate the random vector $\vec{A}$. More generally, the algorithms proposed in this thesis can work for any l_p norm distance function, as long as there is a corresponding p-stable distribution to generate the random vector

in the hash function.

The authors of [22] illustrate how to build an indexing data structure for nearest neighbors search using a locality sensitive hash function family H. In order to obtain the desired search quality, they need to amplify the gap between the collision probability of "*close*" objects (e.g., with a distance less than r_1) and the collision probability of "*far apart*" objects (e.g., with a distance greater than r_2, $r_2 > r_1$). To achieve this goal, the authors concatenate several functions $h \in H$. Specifically, given the integer c, the authors define a function family $G = \{g : S \rightarrow U^c\}$, and for each $g \in G$, $g(v) = \{h_1(v), \cdots, h_c(v)\}$ where $h_i \in H$. "*For an integer L, the algorithm chooses L such functions $g_1, \cdots, g_L$ from G, independently and uniformly at random*" [1]. Each of the L functions g_i $(1 \leq i \leq L)$ is used to construct one hash table; as a result, there are L hash tables.

After concatenating several hash functions, $h \in H$, the gap between the collision probability of "*nearby*" objects (e.g., within a distance less than r_1) and the collision probability of "*far apart*" objects (e.g., with a distance greater than r_2, $r_2 > r_1$) is amplified. For example, assume there are 4 objects a, b, e, f, and $Dis(a,b) = r_1$, $Pr_H[h(p) = h(q)] = p_1$, $Dis(e,f) = r_2$, $Pr_H[h(p) = h(q)] = p_2$. Since $r_1 < r_2$ and due to the monotonic decreasing property of the hash function, we have $p_1 > p_2$. At this point, the gap between the collision probability of (a,b) and the collision probability of (e,f) is $t = p_1/p_2$. After concatenating c hash functions $h \in H$ to form new hash functions $g \in G$, $Pr_H[g(p) = g(q)] = p_1^c$, $Pr_H[g(e) = g(f)] = p_2^c$. Now the gap becomes $T = \frac{p_1^c}{p_2^c} = (\frac{p_1}{p_2})^c = t^c$. Since $t > 1$, $c > 1$, we have $T > t$.

However, after concatenating several hash functions, $h \in H$, not only does the collision probability of "*far apart*" objects (e.g., objects e and f) becomes very small (e.g., p_2^c), but also the collision probability of "*nearby*" objects (e.g., objects a and b) is reduced (e.g., p_1^c). This means that the miss rate of "*nearby*" objects is increased. As a result, we need multiple hash tables in order to reduce the miss rate of "*nearby*"

objects. Given L hash tables, the miss rate of "*nearby*" objects is $1-(1-p_1^c)^L$ [1]. As we can see, when c is fixed and L increases, the miss rate decreases; on the other hand, when L is fixed and c increases, the miss rate also increases.

Then [22] starts to preprocess the data set and construct LSH-based indexing structure. There are 2 main steps [27]: (1) Given parameters c, L, W, it constructs L hash tables, the hash function of each hash table is composed of c randomly picked locality sensitive hash functions h ($h \in H$); (2) For each of the L hash tables, it puts each point $v \in P$ (P is the data set) into the corresponding bucket.

After constructing the LSH index structure, a nearest neighbors' query can be processed in three steps, which have been explained in the first paragraph of this section.

3.2 Multi-Probe LSH

As we mentioned in Section 3.1, one significant drawback of the basic LSH algorithm is that it may require a huge amount of hash tables to reduce the miss rate of nearest neighbors. For example, over 580 hash tables are used in [6]. The size of each hash table will scale linearly with the dataset size (when the number of dimension is fixed). When these hash tables are too large to be stored in the main memory, they have to be stored on disk. As a result, checking a hash bucket may require disk accesses, which are much slower than memory accesses and hence this causes the dramatic performance drop of basic LSH algorithm.

In [30], Panigrahy gives the theoretical study of basic LSH algorithm and proposes an entropy-based LSH algorithm. The key idea of entropy-based LSH algorithm is it generates many randomly "*perturbed*" objects that are close to the query object in the space. Then these objects are queried in addition to the query object, and returns the union of all the fetched objects as the candidate set [27]. In this way, Panigrahy argues that entropy-based LSH method can reduce the space complexity by increasing

the time cost of query processing, but this is still much better than the basic LSH algorithm in the situation where it has to store its hash tables on disk for its larger space complexity.

Entropy-based LSH algorithm has two main drawbacks [27]: (1) The nearest neighbor distance R_n needs to be known beforehand in order to run the perturbation step. However, R_n is really a data dependent parameter. (2) The perturbation step is not computationally efficient, since generating randomly "*perturbed*" points and computing their corresponding hash values are costly and slow.

To overcome the drawbacks of the basic and entropy-based LSH algorithms, the authors in [27] propose a new method called Multi-Probe LSH. It has two main improvements over the entropy-based LSH algorithm: (1) Multi-Probe LSH does not need the knowledge of the nearest neighbor distance (i.e., R_n, which is data dependent) to order the hash buckets. However, such knowledge is required in entropy-based LSH algorithm. (2) Multi-Probe LSH algorithm in a sense probes hash buckets in a more efficient manner and only checks buckets with the highest success/collision probabilities. In the paper, the authors give a simple scoring system which estimates the success probabilities of hash buckets quite well. By using this scoring system, they are able to order the hash buckets for exploration quickly.

The main idea of the Multi-Probe LSH algorithm is to use an intelligently generated probing sequence to explore multiple buckets (in each hash table) that are likely to contain the nearest neighbors of a query object. According to the property of locality sensitive hashing, if some nearest neighbors of the query object q are not hashed to the same bucket as q, they are very likely to be hashed to some "*slightly different*" buckets (i.e., buckets whose signatures only differ slightly to q's signature). So Multi-Probe LSH algorithm uses a probing vector with c entries corresponding to c hash functions (since the hash function $g(v)$ of each hash table is composed of c hash functions, $h \in H$) to explore these "*slightly different*" buckets, in addition to the bucket where the query

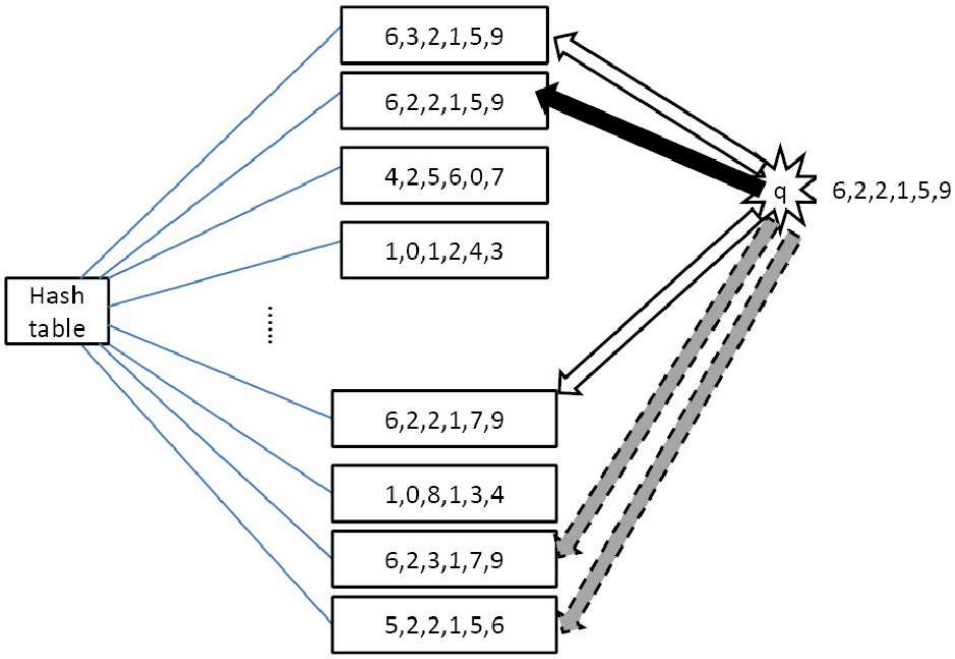

Figure 3.1: Given a query object, Multi-Probe LSH not only checks if the bucket has the same signature as query object's (indicated by black arrow), but also checks all 1-step buckets (indicated by solid white arrows) and 2-step buckets (indicated by dashed gray arrows)

objects is hashed, as shown in Figure 3.1. In [27], the authors have developed two probing sequences for the Multi-Probe LSH algorithm. One is the step-wise probing sequence; another is the query-directed probing sequence. The latter one is shown to be more efficient than the former.

The experimental results in [27] show that the Multi-Probe LSH algorithm can significantly reduce both time and space complexity as compared to the previous LSH-based algorithms (e.g., basic LSH algorithm, entropy-based LSH algorithm). In comparison to the basic LSH algorithm, Multi-Probe LSH algorithm can achieve the same search quality with similar time cost but much fewer hash tables. On the other hand, when compared to the entropy-based LSH algorithm, the Multi-Probe LSH algorithm has a lower query processing time and 5 to 8 times fewer hash tables to achieve the same search quality.

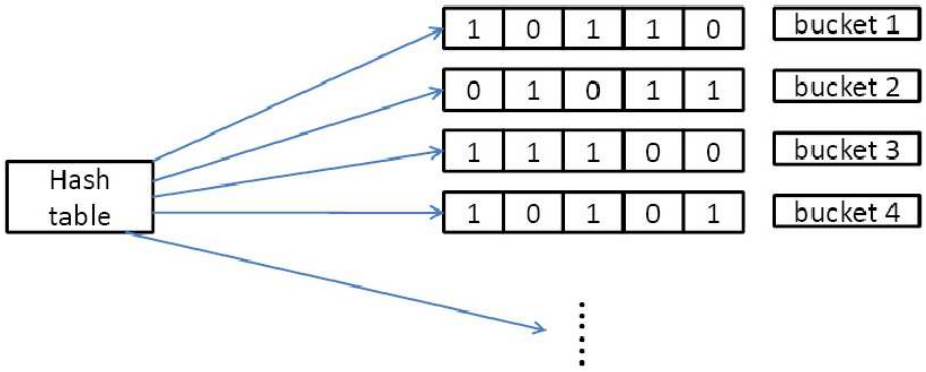

Figure 3.2: Each bucket is identified by the string of integers produced by each of the c LSH functions $h \in H$

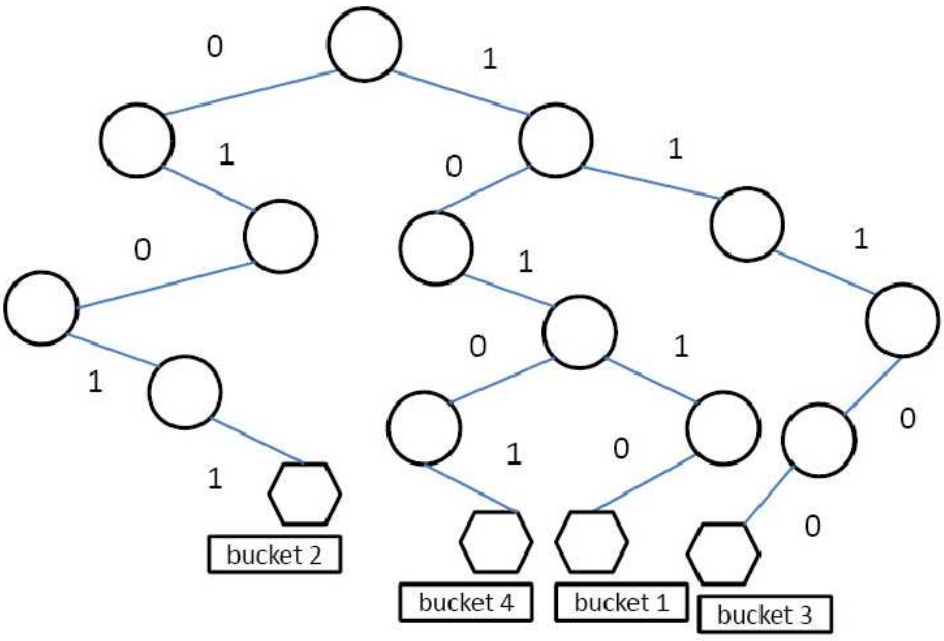

Figure 3.3: The bucket's signature is translated to the path of a prefix tree from the root node to leaf node

3.3 LSH-Forest

Another drawback of the basic LSH algorithm for nearest neighbors search is that it cannot tune the parameters intelligently to answer different nearest neighbor queries (e.g., KNN queries with different result sizes). Once the parameters of the basic LSH (e.g., W, L, c) are decided and the hash tables are built, the bucket size of each table cannot be changed any more. So for a given KNN query, the bucket size is very likely to be too large or too small to answer this query. If the bucket size is too small, then there may be not enough objects in the candidate set. On the other hand, if the bucket size is too large, then there are too many candidate objects, thus the post-processing cost will be high. In the worst case, it will not be better than the brute-force, sequential scan method.

In [3], Bawa *et al.* propose an indexing scheme called LSH-Forest which is applicable in high dimensional nearest neighbors searches. LSH-Forest algorithm is also inspired by the idea of the basic LSH algorithm, but it has several improvements compared to the basic LSH: (1) It does not require hand-tuning of several data dependent parameters (e.g., L, W, c) as in the basic LSH algorithm. (2) It can achieve better search qualities for skewed data distributions while keeping the same space and time complexity [3].

In LSH-Forest, each hash table is presented by a prefix tree, so the number of hash functions per table can be adapted for different approximation distances in order to process different KNN queries. By concatenating the bits on the path from the root node of the prefix tree to the leaf node we get the hash signature (of point p) $g(p) = \{h_1(p), \cdots, h_c(p)\}$, which is the hash label of p in the tree (Figure 3.2 and 3.3). "*Making c big enough would ensure that each object has a distinct hash label. A maximum label length c_m is used to limit the depth of one tree*" [35]. In fact, a leaf node of a tree can be viewed as a hash bucket in the basic LSH algorithm. Given a nearest

neighbors query, LSH-Forest descends each LSH prefix tree to find the leaf having the largest prefix match with the query object's label, and then ascends to check possible siblings simulaneously. In other words, it increases the bucket size of each LSH tree until it collects enough candidate objects.

3.4 Multi-Probe LSH vs. LSH-Forest

The same goal (i.e., answering KNN query in a self-tuning manner, so that the bucket size can vary according to different KNN queries) can also be achieved by using the Multi-Probe LSH method which is mentioned in Section 3.2. Although Multi-Probe LSH is originally proposed to reduce storage cost by probing more than one bucket in each hash table (so we need fewer hash tables to achieve the same search quality), it can also be used to enlarge the bucket size (all the probed buckets can be viewed as one single larger bucket) gradually by probing more and more buckets in each table synchronously until enough candidate objects are collected (instead of being used to save storage cost, in this thesis the Multi-Probe LSH method is being used to vary the bucket size to answer different KNN queries).

In this thesis, both Multi-Probe LSH and LSH-Forest are used as the indexing structures for our HBPP algorithm. These two algorithms have mainly two differences: (1) In the index building step, Multi-Probe LSH uses LSH tables to store the data points, while LSH-Forest uses LSH trees. (2) In the candidates collection step, Multi-Probe LSH increases the candidate set size by exploring more and more buckets in each LSH table, while LSH-Forest increases the candidate set size by traversing toward the higher level node of each LSH tree (from the leaf node) and collecting all the leaf nodes under that node. We will discuss the differences between these two algorithms in more detail in Section 4.2.

Chapter 4

PROPOSED ALGORITHMS

Having seen that all the state of the art LSH based algorithms suffer from the costly post-processing step, here we give a more efficient approximate post processing method to overcome this disadvantage. Our method also utilizes the property of the famous locality sensitive hashing function. We will discuss the innovation of our algorithm in the following paragraphs. In Section 4.2 and 4.3, we will give the algorithm details and the mathematical model formally.

4.1 What is the Innovation?

In the d-dimensional space, for a point p, there is a corresponding vector v_p which starts from the original point to p. As we mentioned in Chapter 3, data sets in high dimensional space with l_p norm distance metric can use the LSH scheme based on p-stable distribution function to solve the KNN search problem. The details are as follows [1]: (1) A d-dimensional vector $\vec{A}$ is generated, entries in $\vec{A}$ are drawn from a p-stable distribution. (2) A real number B is generated, B is a real number picked randomly from the range $[0, W]$, where W is a reasonably large real number. (3) The locality sensitive hash function is defined with the following format: $H_{A,B}(\vec{V}) = \lfloor \frac{\vec{A}\cdot\vec{V}+B}{W} \rfloor$. In other words, a locality sensitive hash function maps a d-dimensional vector $\vec{V}$ to an integer.

Now let us analyze the locality preserving property of the locality sensitive hash function. The dot product $\vec{A}\cdot\vec{V}$ in the hash function maps vector $\vec{V}$ to a real number and the corresponding real number distance between two vectors $\vec{V_1}$ and $\vec{V_2}$ is $(\vec{A}\cdot\vec{V_1} - \vec{A}\cdot\vec{V_2})$ [1]. Since vector $\vec{A}$ is drawn from a p-stable distribution, the real number distance is distributed as $\|\vec{V_1} - \vec{V_2}\|_p t$ where t is a value which follows the p-stable distribution [1]. In fact, one can imagine that the locality sensitive hash function first maps vector $\vec{V}$ to a real number line, then cuts the line into segments with length W, and assigns the corresponding segment number to the vector $\vec{V}$. So if two points are

close to each other in the space, the projection distance of their vectors also tends to be small, so their vectors are very likely to have the same segment number. Now we are clear about the locality preserving property of LSH function.

More formally, in [1], the authors give the formula to compute the probability that two vectors $\vec{V_i}$ and $\vec{V_j}$ collide under a hash function picked randomly from the hash family. Let $f_p(s)$ present the probability density function of the absolute value of the p-stable function. For two vectors $\vec{V_i}$ and $\vec{V_j}$, let $z = \|\vec{V_i} - \vec{V_j}\|_p$. Let A be a d-dimensional vector with entries picked randomly from a p-stable distribution, B be a real number chosen randomly from $[0, W]$. We have

$$p(z) = Pr_{A,B}[h_{A,B}(V_1) = h_{A,B}(V_2)] = \int_0^W \frac{1}{z} f_p(\frac{s}{z})(1 - \frac{s}{W})ds.$$

Hence it is easy to see that for a fixed parameter W the probability of collision $p(z)$ decreases monotonically with z (i.e., the distance between two points i and j).

This advantageous property of LSH is the key for our efficient histogram-based post-processing (HBPP) method. Given two objects X and Y, and one query object q, let us assume $Dis(X,q) = r_1$, $Dis(Y,q) = r_2$, $r_1 < r_2$, hence we have $p(r_1) > p(r_2)$ for any hash function $h_{A,B}()$. It means that object X is more likely to collide with q than Y. Given the concatenated hash function $g(p) = \{h_1(p), \cdots, h_c(p)\}$ for a hash table, the probability that object X collides with object q in the same bucket of the hash table is $(p(r_1))^c$, while the probability for object Y to collide with object q in the same bucket is $(p(r_2))^c$. So we have $(p(r_1))^c \gg (p(r_2))^c$. It means object X is more likely to be hashed into the same bucket as the query object q than object Y. In other words, in a locality sensitive hash table, objects that are close to the query object are much more likely to appear in the corresponding bucket than objects that are far apart from the query object. Given L such locality sensitive hash tables, we expect that objects closer to the query object will occur more times in the combined candidate set. As a result, we can use the occurrence count to rank the candidate objects as an approximation to

the real ordering. The real ordering of candidate objects can be obtained by computing the real distances of the candidate objects (to the query object) and ranking them by distance. As we mentioned earlier, this real ordering step can be very costly.

To our best knowledge, all previous LSH based algorithms for near neighbors search have not considered using the occurrence information for the pruning of false positive objects in the candidate set. So in this thesis, we propose to use this useful information to speed up the post-processing step. In Section 6.1 of Chapter 6, we will show that by using our occurrence histogram based pruning method (HBPP), the query processing can be improved by at least an order of magnitude, while still maintaining high search quality.

4.2 Algorithms

There are mainly 3 steps in our algorithms: (1) Index building, which is similar to the previously mentioned LSH-based nearest neighbors search algorithms (e.g., LSH-Forest, Multi-Probe LSH). (2) Candidate objects collection, this step is also similar to that of LSH-Forest algorithm or Multi-Probe LSH algorithm. We enlarge bucket size gradually to fetch enough objects from each LSH tree/table. (3) Histogram-based post-processing on candidate set. This is the most innovative step compared to existing algorithms. We use the occurrence count of candidate objects to estimate their distances (since occurrence count is a good approximation of distance) to the query object, which is a much cheaper operation compared to the real distance computation. In fact, step 2 and step 3 together can be called the query processing step, which will be described below.

Index Building

There are mainly two choices about the indexing structures, one is original LSH table (as in Multi-Probe LSH), and the other is LSH tree (as in LSH-Forest). (1) For the first choice, we can achieve the goal of dynamically expanding buckets in each table

by probing more and more hash buckets. As in [3], each hash bucket ideally should contain only one object to achieve the best search quality. However, in this ideal case, the number of buckets will be equal to the number of data objects and more buckets mean extra memory cost. As a result, we will impose a largest bucket number B_m in each table. In the basic design of LSH, the bucket number is decided by c (the number of concatenated hash functions $h \in H$) and the number of segments in each h. If we fix c, then the only parameter affecting the bucket number is the segment number of each hash function h, which is affected by W and the space range of the data set. So according to the specific data domain, the specific value of W is required in order to get the appropriate segment number of hash functions. So in that sense, Multi-Probe LSH has at least one data-dependent parameter. (2) For the second choice, the goal of dynamically expanding buckets in each LSH tree can be achieved by moving from leaf node of the tree towards the root node. As in [3], each leaf node can be viewed as a bucket in the hash table and should contain only one object in the ideal case. However, doing so may require the depth to be extremely large, which results in large number of tree nodes and huge memory cost. Similarly as in the first choice, we also give a largest leaf number F_m in each tree. As we know, the number of leaves in a binary tree is only decided by the depth of the tree. So given the value F_m, we can get the depth of the LSH tree in a data independent manner. In that sense, LSH-Forest is more data independent and self-tuning.

Candidate Objects Collection

a. Candidate Objects Collection for LSH-Forest

Given an LSH-Forest consisting of L LSH trees built on a set of objects, the candidate objects for the KNN query of object q can be collected by traversing the LSH trees in two steps. In the first top-down step, we descend each LSH tree T_i to find the leaf node having the largest prefix match with q's hash signature as shown in Figure 4.1 [3]. In the second bottom-up step, M data objects are collected from each tree while we are

```
Algorithm: DESCEND(q,y_i, f)

  Input: query q at level y_i on node f
  Output:a leaf node matches q's signature with max length

  if f is a leaf then
      return f;
  else
      z=y_i+1;
      h_z(q) =Evaluate g_i(q,z);
      n=child node of f from branch with label h_z(q);
      p=DESCEND(q,z,n);
      return p;
  end if
```

Figure 4.1: The top-down step in candidates collection on LSH tree T_i initiated with DESCEND(q,0,$root_i$), adapted from [3]

```
Algorithm: FORESTCOLLECT(s[0,···,l−1])

  Input: s_i is the corresponding leaf node for LSH tree T_i
  Output: a hash table P which contains candidate objects and their occurrence count

  Budget[0,···,l−1] = {0,···,0}; /*Budget is an array with values initialized with 0 */
  P = ∅; /*P is an empty hash table*/

  for i = 0 to l-1 do
      while Budget[i] < M do
          A=Descendants[s[i]];
          for j = 0 to A.length-1 do
              if P.containsKey(A[j]) then
                  count=P.getValue(A[j]);
                  count ← count + 1;
                  P.update(A[j],count);
              else
                  P.put(A[j],1);
              end if
              Budget[i] ← Budget[i] + 1;
              if Budget[i] ≥ M then
                  break;
              end if
              j ← j + 1;
          end for
      end while
      i ← i + 1;
  end for
  return P;
```

Figure 4.2: The bottom-up step in candidates collection initiated with arguments returned by DESCEND

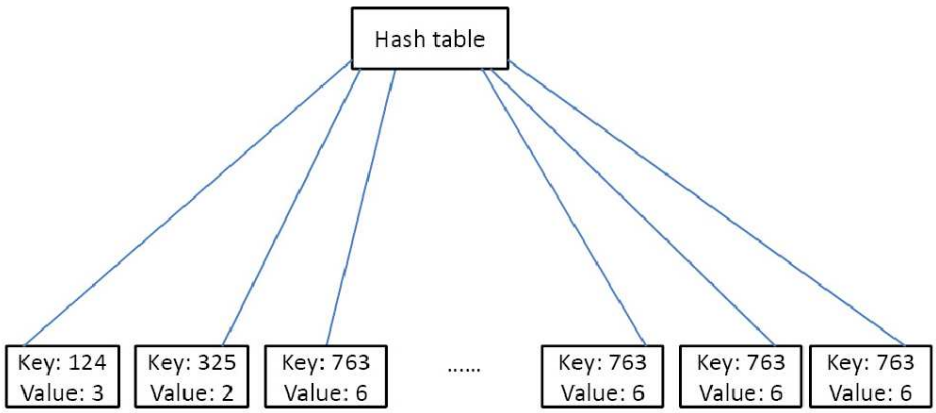

Figure 4.3: The result hash table of the candidate collection step, the key is the object ID, the value is the occurrence count of the object in the candidate set

ascending from the corresponding leaf node (that is found in the first step) to the root node [3]. While we are collecting candidate objects synchronously from each LSH prefix tree, we remember the occurrence count of each distinct object (there will be M' distinct candidate objects, where $M' < L \times M$). The second step is shown in Figure 4.2 and an example of the candidate objects collection result is shown in Figure 4.3. These two steps work together as the candidate objects collection step.

b. Candidate Objects Collection for Multi-Probe LSH

Given L hash tables built on a set of objects, the candidate objects for the KNN query of object q can be collected as follows: we generate q's signature $g_i(q)(0 \leq i \leq L)$ by using the hash function for each hash table. Then we start collecting M objects in each hash table. First we collect objects from the bucket with the highest success/collision probability, given the parameters $(q, g_i(q), g_i())$. If there are not enough objects, we collect objects from the bucket with the next highest success/collision probability and so on, until M objects are collected. When we are collecting candidate objects synchronously from each hash table, we remember the occurrence count of each distinct object (there will be M' distinct candidate objects, where $M' < L \times M$). The full process of candidate objects collection for Multi-Probe LSH is shown in Figure 4.4.

Histogram-based Post-processing

```
Algorithm: MULCOLLECT(q)

Input: query object q
Output: a hash table P which contains candidate objects and their occurrence count

Budget[0,···,l−1] = {0,···,0}; /*Budget is an array with values initialized with 0 */
P = ∅; /*P is an empty hash table*/

for i = 0 to l-1 do
    while Budget[i] < M do
        A=NextBestBucket(q, g_i(q), g_i());
        for j = 0 to A.length-1 do
            if P.containsKey(A[j]) then
                count=P.getValue(A[j]);
                count ← count + 1;
                P.update(A[j],count);
            else
                P.put(A[j],1);
            end if
            Budget[i] ← Budget[i] + 1;
            if Budget[i] ≥ M then
                break;
            end if
            j ← j + 1;
        end for
    end while
    i ← i + 1;
end for
return P;
```

Figure 4.4: The candidate objects collection step of Multi-Probe LSH

```
Algorithm: POST-PROCESS(P,K)

Input: candidate set hash table P, K is the size of query result
Output: a set of K objects as the KNN query result

Budget[0,···,l−1] = Bucketsort(P); /*Budget is an array with values initialized with 0 */
R = ∅;

for i = l − 1 to 0 do
    if R.size ≥ K then
        break;
    end if
    for j = 0 to B[i].length − 1 do
        R.put(B[i][j]);
        if R.size ≥ K then
            break;
        end if
        j ← j + 1;
    end for
    i ← i − 1;
end for
return R;
```

Figure 4.5: The post processing step initiated with arguments returned by FOREST-COLLECT/MULCOLLECT

Now we will introduce the histogram-based post-processing (HBPP) step. When we get the candidate set, we have the occurrence count of each candidate object. Thus, we rank the candidate objects according to their occurrence counts (occurrence count can be used as an approximation of distance of each candidate object to the query object). Since the maximum possible occurrence count of one object is L (L is the total number of hash tables/LSH trees), we can use bucket sort to order these candidate objects according to their occurrence counts quickly. Then we collect objects from the highest occurrence count slot, then the next highest occurrence count slot and so on, until we get enough number of nearest neighbors, in this case, it is K (we give an example of KNN objects collection in the end of Section 1.2). The post-processing step is shown in Figure 4.5 and an example of the sorted candidate objects is shown in Figure 1.2.

4.3 Mathematical Model

Here we give a mathematic model to explain why our histogram-based post-processing algorithm works. We build our model under the assumption that the indexing structure is LSH-Forest.

In the high dimensional space, the whole space can be viewed as a hyper-ball. As we know, the hyper-ring around the out-most of the hyper-ball will become a significant volume relative to that of the hyper-ball, because of the "*curse of dimensionality*" [8]. If the data set distribution is approximate to uniform distribution, then we can say that most of the data points will be on the surface of the hyper-ball.

For a given query object q, there exists a hyper-ball with q as the center, and r as the radius. Within this hyper-ball there are K objects, not including the query object. These K objects are the real KNN of q. As we explained in the above paragraph, we can assume that all these K objects are on the hyper-surface of this hyper-ball in the high dimensional space.

Meanwhile, we can view the whole data space as a hyper-ball with radius $R(R > r)$, and we can assume that all the false positive objects are on the hyper-surface of this larger hyper-ball.

We assume that p_r is the probability of the real KNN objects colliding with the query object in any locality sensitive hash function $h \in H$, which is drawn uniformly from the p-stable distribution, and p_R for all other false positive objects. Given the depth t (Let us assume M is the candidates budget size of each tree, N is the size of the false positives, and K is the size of the real KNN result. Since LSH tree is a binary and balanced tree when the data distribution is approximate to uniform, we have $t = \lfloor \log_2(\frac{N+K}{M})) \rfloor$) of one LSH tree, we know that the probability of the real KNN objects colliding with the query object in the same bucket is $(p_r)^t$, and $(p_R)^t$ for false positive objects. We set $p_1 = (p_r)^t$, and $p_2 = (p_R)^t$, for simplicity purposes. So in each tree, the probability of a real KNN objects appearing in the bucket of query object is p_1 and the probability of missing the bucket is $(1 - p_1)$. As a result, the probability of a real KNN object occurring x times in L LSH trees is $C_L^x (p_1)^x (1 - p_1)^{L-x}$, and for simplicity, we will denote it as $BinomDist[L, x, p_1]$. Similarly, for a false positive object, the probability that it appears x times in the candidate set is $BinomDist[L, x, p_2]$. Then we define the accumulated probability of an object appearing at least i times. For a real KNN object, it is $\sum_{x=i}^{L} BinomDist[L, x, p_1]$, and for simplicity, we denote it as $CDF[BinomDist[L, p_1], i]$. Similarly, the accumulated probability for a false positive object is $CDF[BinomDist[L, p_2], i]$.

So the expected number of objects occurring at least x times is:
$CDF[BinomDist[L, p_1], x] \times K + N \times CDF[BinomDist[L, p_2], x]$. Let us denote it as $ResultNum$. In order to fetch the best K objects from the candidate set, we need to decrease x (starting from L) until $ResultNum \geq K$. Let us assume that the largest x value satisfying the condition $ResultNum \geq K$ is x_s. Then we collect objects from the highest occurrence slot, then the next highest occurrence slot and so on, until we

get K nearest neighbors. If the last slot (slot x_s) from which we collect KNN objects has more objects than required, we will just randomly pick the required number of objects from that slot. These K objects form the KNN of the query object. To evaluate the recall of the result, we need to know the number of real KNN object in these K retrieved objects. The expected number of real KNN objects in the slots with number equal or larger than x_s+1 is $K \times CDF[BinomDis[L,p_1],x_s+1]$, the expected number of false positives in the slots with number equal or larger than x_s+1 is $N \times CDF[BinomDis[L,p_2],x_s+1]$, the expected number of real KNN objects in the slot with number x_s is $K \times BinomDist[L,p_1,x_s]$, the expected number of false positives in the slot with number x_s is $N \times BinomDis[L,p_2,x_s]$. So the expected number of real KNN objects in the K retrieved objects is

$K \times CDF[BinomDis[L,p_1],x_s+1] + (K - K \times CDF[BinomDis[L,p_1],x_s+1] - N \times CDF[BinomDis[L,p_2],x_s+1]) \times \frac{K \times BinomDist[L,p_1,x_s]}{K \times BinomDist[L,p_1,x_s] + N \times BinomDis[L,p_2,x_s]}$, and the recall is

$CDF[BinomDis[L,p_1],x_s+1] + (K - K \times CDF[BinomDis[L,p_1],x_s+1] - N \times CDF[BinomDis[L,p_2],x_s+1]) \times \frac{BinomDis[L,p_1,x_s]}{K \times BinomDist[L,p_1,x_s] + N \times BinomDis[L,p_2,x_s]}$.

To verify the correctness of the model, we tested it with following scenarios:

1. Given R, r, N, M, L, what is the relation between recall and K?

2. Given R, r, N, K, L, what is the relation between recall and M?

3. Given R, r, N, M, K, what is the relation between recall and L?

The patterns we find in the model from the above scenarios match well with real data case. We will give more details in Section 6.4 of Chapter 6.

Discussion

With the help of the above model, it is possible to predict the recall value for a KNN query, for given L, K, N, M, and local distribution and global distribution of the dataset.

The local distribution is used to estimate r, while the global distribution is used to set W. The reason is as follows: for a given query object and K, we can estimate r (r is the maximum distance of KNN objects from the query object) if we know the distribution information around the query object. Given r and W, we can obtain the value of p_r, thus we know p_1 ($p_1 = (p_r)^t$, and the value of t can be estimated by using local distribution, global distribution, N, K and M). Then the expected number of real KNN objects we collect from each LSH tree is $p_1 \times K$. So the expected number of false positives we collected from each LSH tree is M'=$M - p_1 \times K$, thus p_2 can be obtained by $\frac{M'}{N}$ (p_2 means the collision probability of any false positive object with the query object in each LSH tree, so it also means the percent of false positives being collected in each LSH tree). Now we have obtained all the required parameters in the recall formula presented above (x_s is determined by p_1, p_2, K and N). Thus we can use the recall formula to predict the recall value.

Chapter 5

EXPERIMENTAL SETUP

In this section, we describe the setup of our experiments, including the experimental data sets, algorithms to be evaluated in the experiments, evaluation metrics, and implementation details.

5.1 Experimental Data Sets

There are four different data sets used in our experiments. We want to show under different data distributions, how our histogram-based post-processing (HBPP) methods help reduce time cost while there remains high search quality and high space efficiency in the KNN query processing.

SIFT Data

This data set is created by extracting scale invariant feature transform (SIFT) [26] vectors from an online image data set [24]. SIFT vectors are local descriptors of the interesting points of an image that are invariant to image scaling, translation, rotation, and also partially invariant to illumination and projections. By default, SIFT vectors have 128 dimensions, and the value in each dimension is integer. In our experiments, we extract one million SIFT vectors from the image data set.

GIST Data

We download this data set from [18]. There are totally 1 million vectors in the data set, and each vector has 960 dimensions. The data type of value in each dimension is double. These vectors are generated by projecting objects in an image to a high dimensional (960 dimensions) space. By doing so, scenes sharing membership in semantic categories (e.g., streets, highways) are projected close together[2]. GIST data is mainly used in our experiments to show the scalability of our histogram-based post-processing (HBPP) algorithms to high dimensions.

Uniform Data

This data set is created synthetically. Each vector is generated as follows: for each dimension in the vector, we assign a double value, which is randomly picked from the range $[0, 50]$, to it. For the purpose of comparing with results of SIFT data, we also set the number of dimensions for this data set as 128. It is shown in [39] that for vectors/points generated uniformly in high dimensional space as above, the distribution of their distances will obey power-law (the distances between most point pairs are small, while very few points pairs have large distances). We also generated 1 million vectors for this data set.

Normal Data

This data set has 20 clusters, and data points in each cluster obey normal distribution. This data set is created by first randomly generating 20 points in the high dimensional space (128 dimensions, each dimension has the value range $[0, 50]$), then for each point, we use it as the cluster center to generate the corresponding cluster. For each cluster, we randomly pick variance from the value range $[0, 20]$. We set the number of data points to be equal among all clusters (50000 if the total data size is 1 million), thus the densities of clusters (in the space) vary. By doing so, we make sure the distribution is skewed in this data set. Each vector/point in this data set also has 128 dimensions and we generated 1 million vectors for this data set.

5.2 Experimental Algorithms

In the experiment, we implemented six KNN search algorithms: (1) LSH-Forest with random pick (LSH-Forest with RP); (2) Multi-Probe LSH with random pick (Multi-Probe LSH with RP); (3) LSH-Forest with data access post-processing (LSH-Forest with DAPP); (4) Multi-Probe LSH with data access post-processing (Multi-Probe LSH with DAPP); (5) LSH-Forest with histogram-based post-processing (LSH-Forest with HBPP) and (6) Multi-Probe LSH with histogram-based post-processing (Multi-Probe

LSH with HBPP). These six algorithms can be categorized into two classes: LSH-Forest based algorithms and Multi-Probe LSH based algorithms.

LSH-Forest based Algorithms

Three algorithms belong to this class: LSH-Forest with RP, LSH-Forest with DAPP and LSH-Forest with HBPP. All these three algorithms use LSH-Forest as the indexing structure and collect candidate objects as mentioned in the Chapter 4. After enough candidate objects are collected from each LSH tree, we union all of them to form the candidate set. The main difference among these three algorithms is the post-processing step after candidates collection. For LSH-Forest with RP algorithm, we just randomly pick K objects from the candidate set as the answer to the KNN query. For LSH-Forest with DAPP algorithm, we compute the distances of all candidate objects to the query object and pick K objects with smallest distances, so this post-processing step needs to access the database to retrieve the data vectors. For LSH-Forest with HBPP algorithm, we apply our histogram-based post-processing method (as mentioned in Chapter 4).

Multi-Probe LSH based algorithms

There are also three algorithms in this class: Multi-Probe LSH with RP, Multi-Probe LSH with DAPP, and Multi-Probe LSH with HBPP. All these three algorithms use LSH hash table as the indexing structure and collect enough candidate objects from each table by using the Multi-Probe LSH algorithm as mentioned in Chapter 4. Then they union all candidate objects to get the candidate set. The main difference among these three algorithms is also the post-processing step after candidate collection, as mentioned in the above paragraph.

5.3 Evaluation Metrics

The performance of a KNN search algorithm can be measured in three aspects: search quality, time cost, and space complexity. An ideal KNN algorithm should achieve all three goals: high search quality, high time efficiency, and high space efficiency.

The search quality can be evaluated by the fraction of the real KNN objects we are able to retrieve in the query result (i.e., recall).

Moreover, the search quality can also be evaluated by comparing the sum of distances of K retrieved objects (to the query object q) to the sum of distances of real K nearest neighbors (to q), which can be represented formally as follows:

error ratio$= \varepsilon = \frac{\sum_{i=1}^{d} Dis(q,p'_i)}{\sum_{i=1}^{d} Dis(q,p_i)}$-1, $p_i \in$ (Real KNN),$p'_i \in$ (K retrieved objects)

As we can see, the smaller ε is, the better the search quality is. When ε is 0, then our algorithm performs perfectly, in which case the K objects retrieved by our algorithm are the real KNN. We will evaluate the search quality by using both these two metrics in the experiments.

Time efficiency can be measured by the query time, which is time cost per query. Space efficiency is measured by the total memory usage.

5.4 Implementation Details

We have implemented the six algorithms described above. We store all the experiment data sets in a popular database called BerkeleyDB [29]. We will read these data sets from BerkeleyDB to build the hash index structure in memory. Furthermore, vectors will be fetched from BerkeleyDB to compute distances in the post processing step for the DAPP algorithms.

The evaluations are done on a server machine with one four-processor Intel 3.00 GHz CPU with 6144 KB cache. The server has 8 GB of DRAM and a 320 GB and 7200 RPM SATA disk. The operation system of this server machine is Ubuntu 10.2.

Chapter 6

EXPERIMENTAL RESULTS

In this chapter, we report the evaluation results of the six algorithms using the four data sets (SIFT Data, GIST Data, Uniform Data, Normal Data). We are interested in answering questions including how the search quality (evaluated by both error ratio and recall) and time cost trade-offs for different algorithms, given same amount of memory usage. The experimental results are clustered into 4 sections: efficiency and effectiveness results, scalability results, sensitivity results and model comparison results.

6.1 Efficiency and Effectiveness Results

In this section, three LSH-Forest based algorithms (namely LSH-Forest with RP, LSH-Forest with DAPP and LSH-Forest with HBPP) and three Multi-Probe LSH based algorithms (namely Multi-Probe LSH with RP, Multi-Probe LSH with DAPP and Multi-Probe with HBPP) will be evaluated by using four different data sets. For each data set, 0.1 million data objects are randomly picked from the original 1 million data set as the experimental data, then 100 objects are randomly picked from these 0.1 million objects as query objects and the time cost in the experiment is the average query processing time of these 100 queries. In these experiments, we set the number of LSH trees/tables to be 160 ($L = 160$). To ensure the memory usages of the six algorithms for the same data set are equal, firstly we define the depth of the LSH trees for LSH-Forest based algorithms to be 8 and the number of hash functions $h \in H$ of each hash table for Multi-Probe LSH based algorithms to be 6, then we record the memory costs of LSH-Forest based algorithms for the experiment data set. Next we will pick the appropriate value for W (segment width of LSH functions) for Multi-Probe based algorithms for the same data set such that the memory cost is the same. We also set the KNN percent (percent of KNN result size compared to the full data size, e.g., KNN percent=5% means the size of KNN result is 5000 if the data set size is 0.1 million) to be 5%. We will show the scalability results for these algorithms in section 6.2.

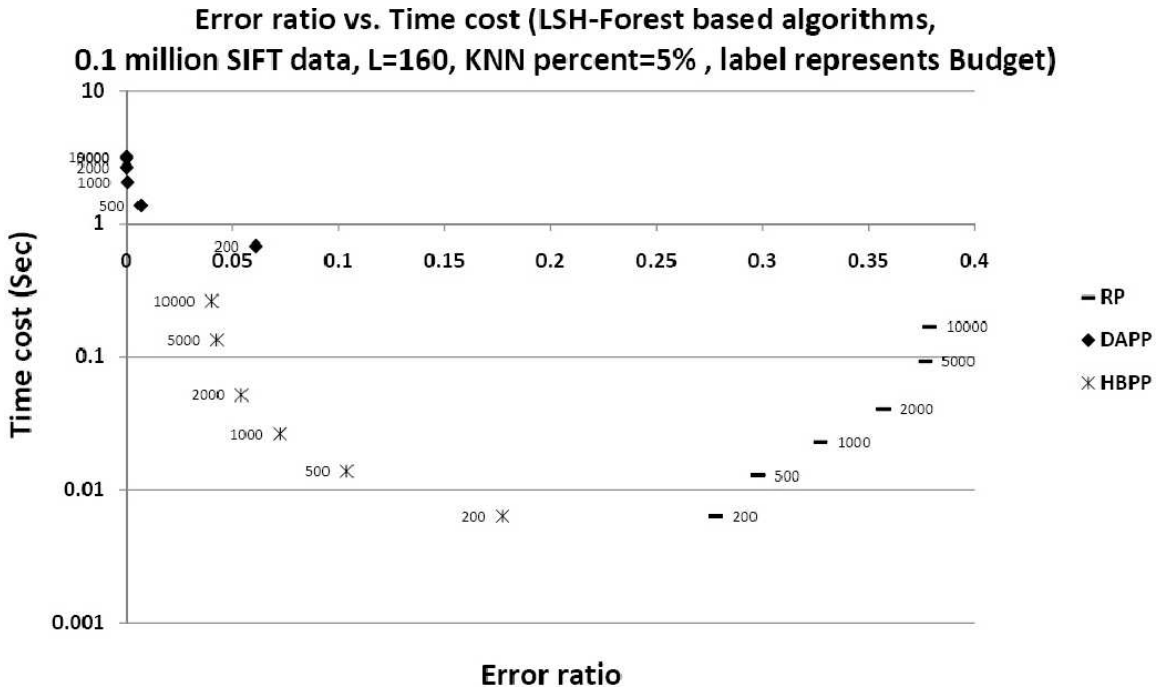

Figure 6.1: Error ratio vs. Time cost results of LSH-Forest based algorithms with 0.1 million SIFT Data

LSH-Forest based Algorithms Evaluated based on Error Ratio

In Figures 6.1, 6.2, 6.3 and 6.4, we show the performances of three LSH-Forest based algorithms (LSH-Forest with RP, LSH-Forest with DAPP and LSH-Forest with HBPP) by using the four different data sets.

In Figures 6.1, 6.2, 6.3 and 6.4, every curve represents an "*Error ratio vs. Time cost*" trend of one of the three LSH-Forest based algorithms. We vary the time cost of each algorithm by changing the budget size. Budget size means the number of candidate objects collected from each LSH tree, and larger budget size means higher time cost. On the other hand, large budget size also decreases the error ratio (except for LSH-Forest with RP algorithm, in which case the error ratio may increase). Thus, we get the "*Error ratio vs. Time cost*" curves in each of the four figures.

As shown in Figures 6.1, 6.2, 6.3 and 6.4, for LSH-Forest with DAPP and LSH-Forest with HBPP, we can achieve better search quality by increasing the budget size

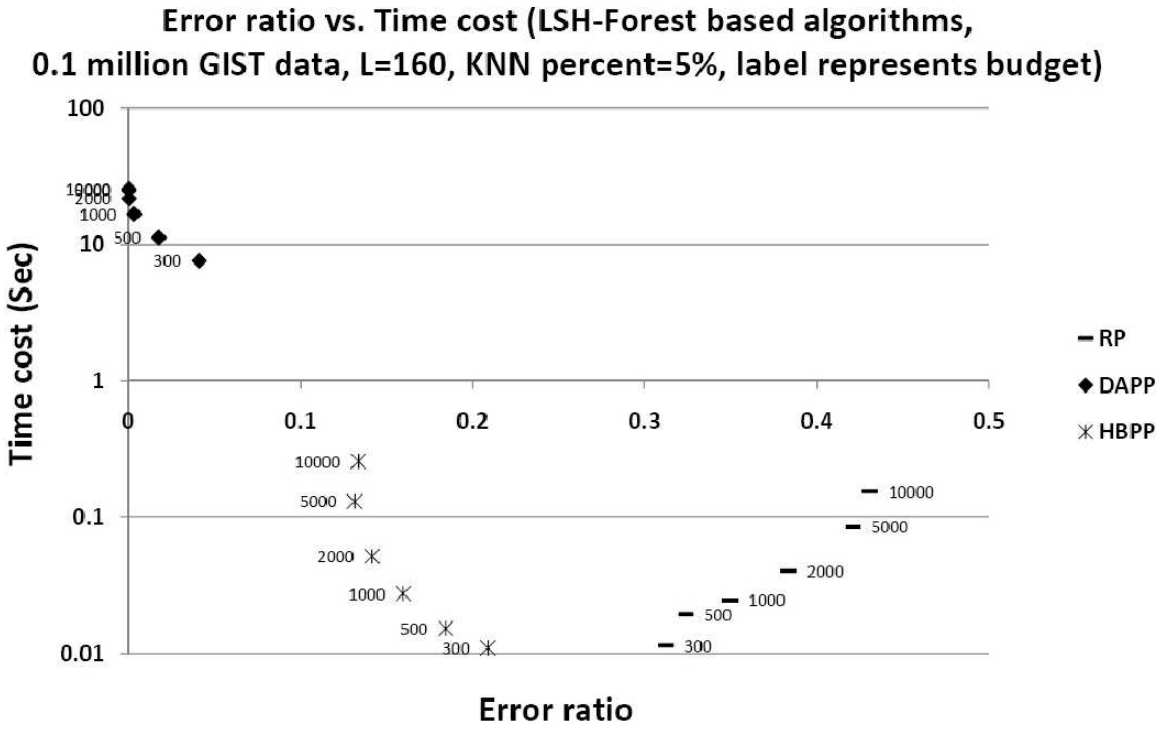

Figure 6.2: Error ratio vs. Time cost results of LSH-Forest based algorithms with 0.1 million GIST Data

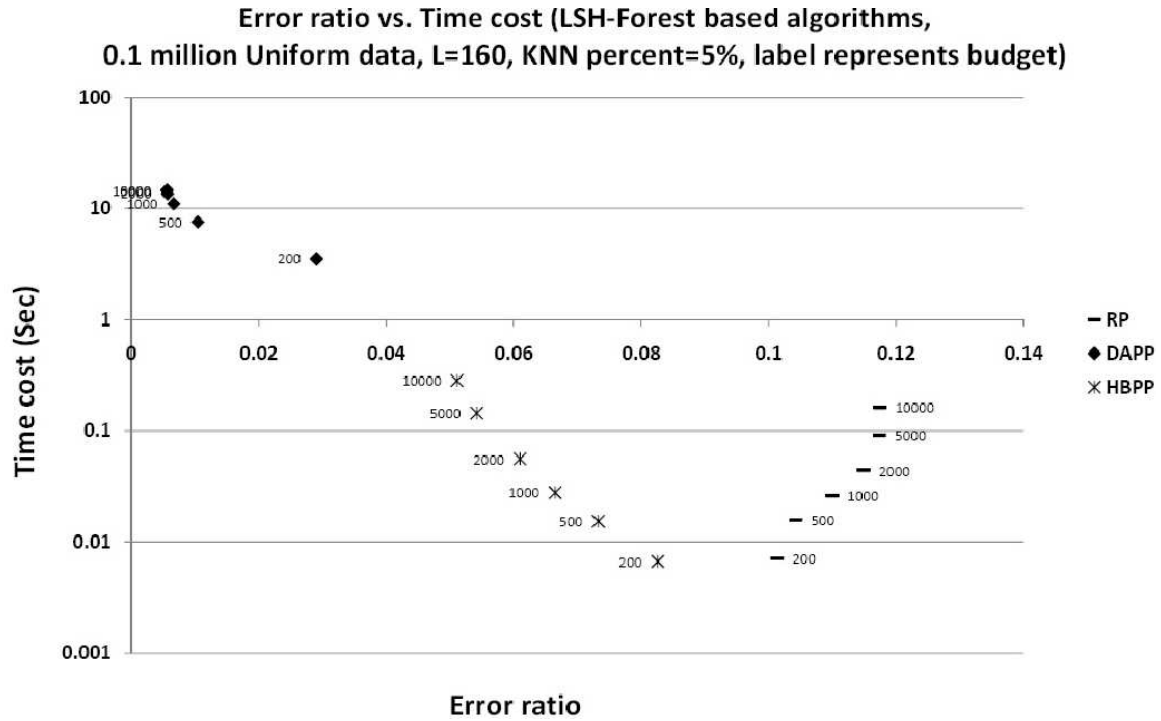

Figure 6.3: Error ratio vs. Time cost results of LSH-Forest based algorithms with 0.1 million Uniform Data

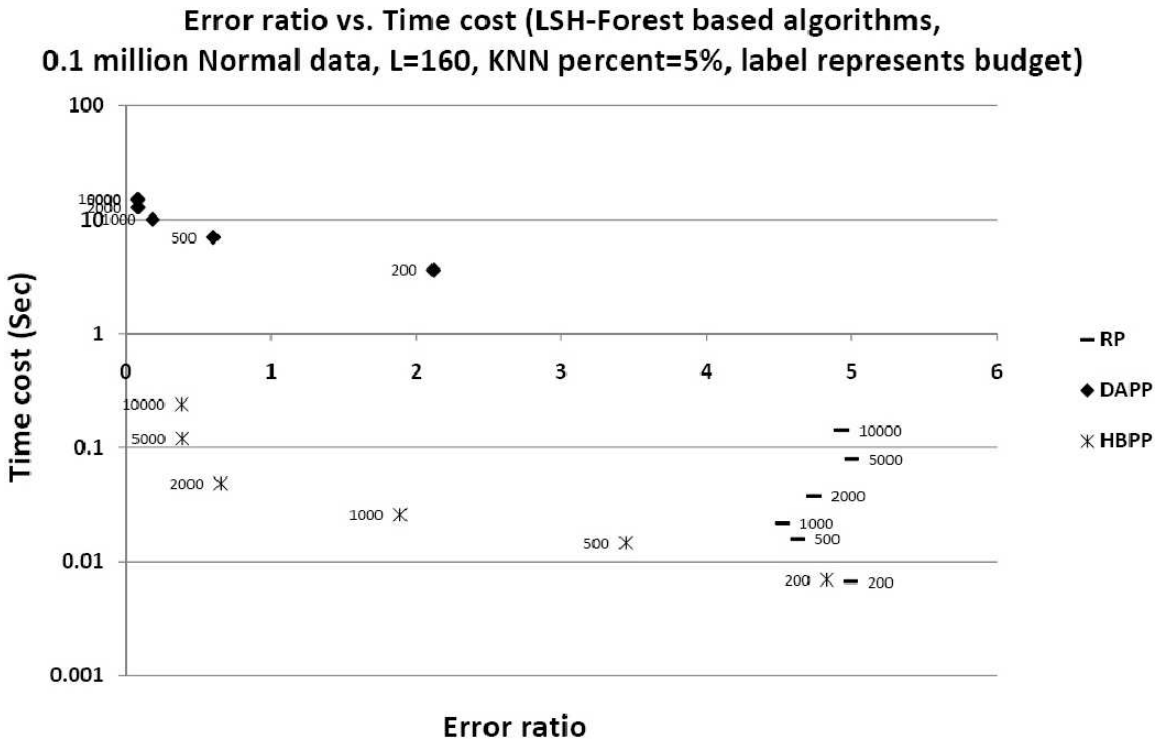

Figure 6.4: Error ratio vs. Time cost results of LSH-Forest based algorithms with 0.1 million Normal Data

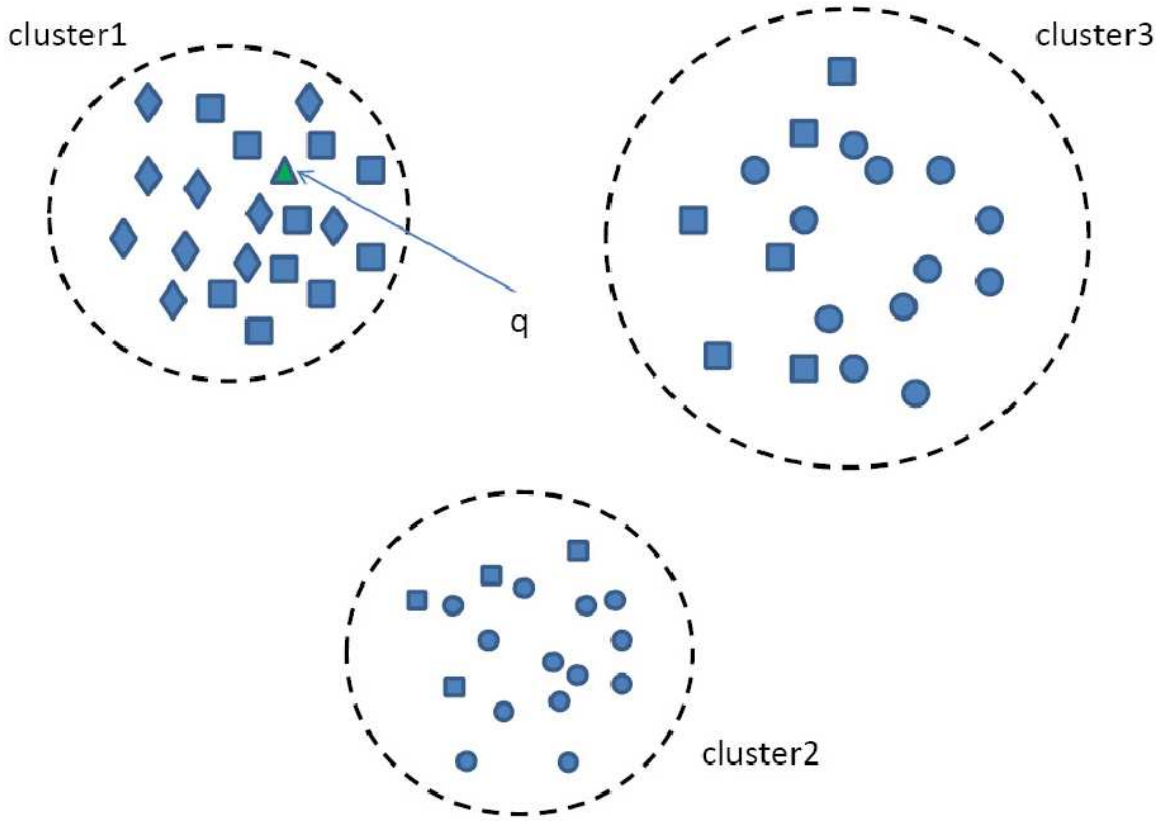

Figure 6.5: 20 nearest neighbors example of Normal Data

(thus increasing time cost). However, for LSH-Forest with RP, error ratio increases (i.e., search quality becomes worse) as the budget size increases. The reason is that more candidate objects also mean higher percentage of false positive objects in the final result if there is no effective pruning in the post-processing step. We also notice two interesting patterns: (1) given the same budget size, LSH-Forest with HBPP can achieve much lower error ratio (better search quality) than LSH-Forest with RP with a little more time cost (this also shows LSH-Forest with HBPP is very time efficient). (2) To achieve the same error ratio (search quality), LSH-Forest with HBPP is much faster than LSH-Forest with DAPP.

As we can see from these four figures, to achieve the same search quality (error ratio), LSH-Forest with DAPP requires at least an order of magnitude more time cost than LSH-Forest with HBPP. Especially for GIST Data in Figure 6.2, the time gain is even more. The reason is that GIST Data has 960 dimensions, so the data access post-processing (DAPP) will be even more expensive compared to other data sets (with 128 dimensions). In fact, the time cost of the histogram-based post-processing (HBPP) algorithm will not increase as dimension increases. Because in histogram-based post-processing, only the occurrence counts of candidate objects (keys) will be considered, it has nothing to do with the high dimensional vectors of the data objects which are stored in the disk database. This means algorithms with HBPP scale quite well with number of dimensions. However, for algorithms with DAPP, given that the size of the candidate set is defined, as the dimensions of the objects increase, both the disk I/O cost and distance computation cost increase. Another interesting thing worth mentioning is that given the same budget size, the time cost of LSH-Forest with DAPP algorithm for SIFT Data is much less than that for Uniform Data and Normal Data (these three data sets all have 128 dimensions). That is because the value type of vectors in SIFT data is integer, but the value type of vectors in Uniform Data and Normal Data is double. Thus both the disk I/O cost and distance computation cost (of LSH-Forest with DAPP)

for SIFT Data are much less than those of Uniform Data and Normal Data.

We also note that Figure 6.4 is quite different from other three figures. In Figure 6.4, the error ratios of all the curves go beyond 1, they even go beyond 5 in the worst case. However, for all other three figures, the error ratios never go beyond 1. This is due to the relation between the KNN percent in the experiment and the cluster size of the Normal Data. In the experiment, we set KNN percent=5%, and the size of each cluster in the Normal Data is also 5% of the full data set size. Since these clusters are very likely to be far away from each other (as we described in Section 5.1 in Chapter 5), a perfect KNN answer should only contain all the objects in the cluster to which the query object belongs. However, since LSH-based algorithms only give an approximate answer, there must be some portions of false positive objects in the KNN result which belong to other far away clusters. In fact, given the same recall, the KNN results of other three data sets will have much lower error ratios than those of Normal Data, since the false positive objects in the results of other three data sets are not much further from the query point compared to the real KNN objects (but most of the false positive objects in the Normal Data case are much further from the query points compared to the real KNN objects). As we can see from Figure 6.1, 6.2 and 6.3, even for LSH-Forest with random pick (RP), the error ratio is still below 1. However, for skewed data set such as Normal Data, the difference between a real KNN object and a false positive object is huge from the perspective of the distance from the query object. This is also the reason that for LSH-Forest with random pick (RP) in Figure 6.4, the error ratio is very high. When we collect enough candidates, most of the very far away false positive objects will be eliminated from the final KNN result after post-processing (by DAPP or HBPP), so we can achieve small error ratio. Actually, Figure 6.4 is a good example to show that our LSH-Forest with HBPP performs quite well even for skewed data set. In Figure 6.5, an example is used to show what we described above. In the example, the triangle in cluster1 is the query object, and we ask for 20 nearest neighbors of the query

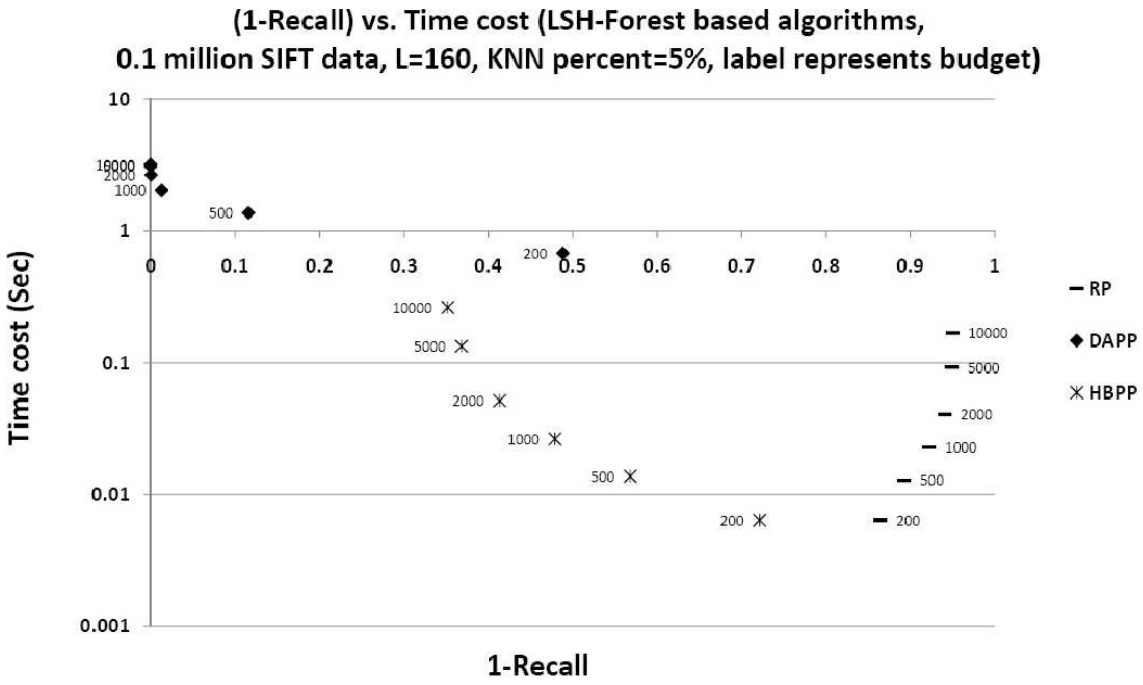

Figure 6.6: (1-Recall) vs. Time cost results of LSH-Forest based algorithms with 0.1 million SIFT Data

object. The real 20 nearest neighbors of the query object are all the objects in cluster1, and 20 rectangles are returned as the approximate answer. Since LSH and its variants are approximate algorithms, there might be some false positive objects from other clusters, and these far away false positives cause the high error ratio. The diamond objects in cluster1 are misses.

LSH-Forest based Algorithms Evaluated based on Recall

In this subsection, we evaluate the three LSH-Forest based algorithms with four different data sets. All the setups are the same as the previous subsection, except that the search quality is evaluated by recall instead of error ratio.

In Figures 6.6, 6.7, 6.8 and 6.9, every curve represents a "*(1-Recall) vs. Time cost*" trend of one of the three LSH-Forest based algorithms. The time cost of each algorithm is varied by changing the budget size. The value $(1-Recall)$ can be decreased by increasing the budget size (except for LSH-Forest with RP algorithm, in which case the value $(1-Recall)$ may increase). Thus, we get the "*(1-Recall) vs. Time*

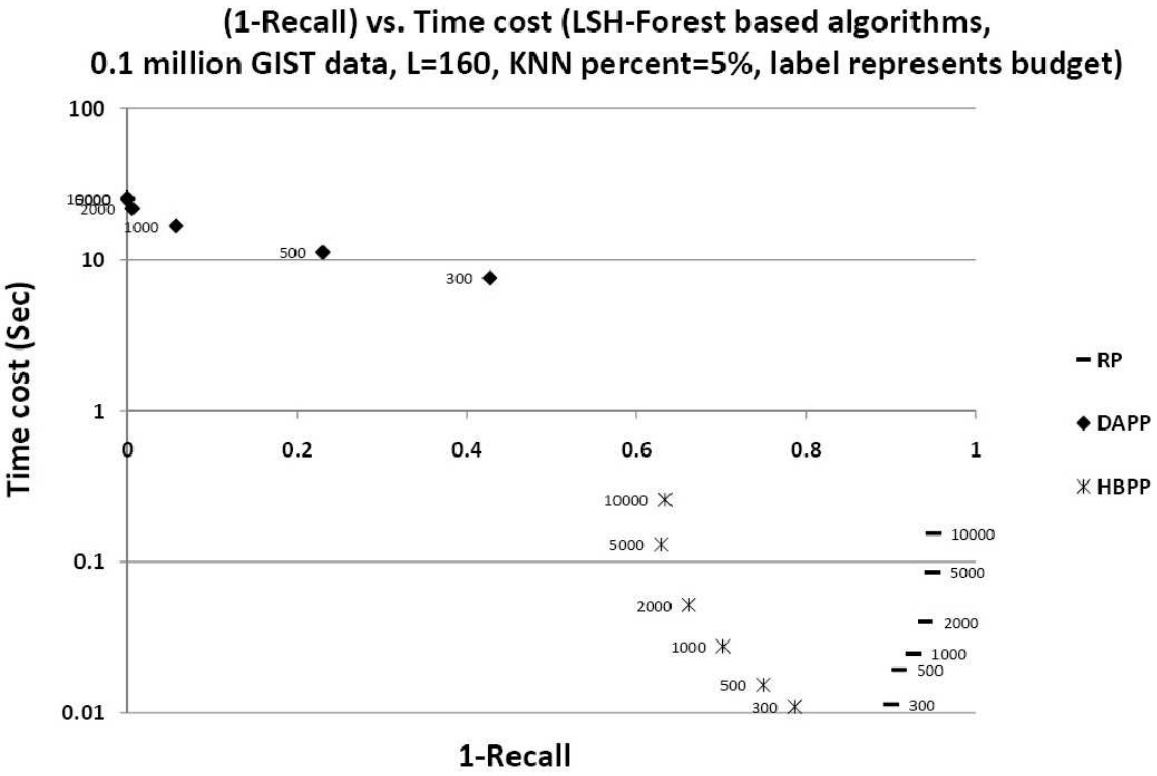

Figure 6.7: (1-Recall) vs. Time cost results of LSH-Forest based algorithms with 0.1 million GIST Data

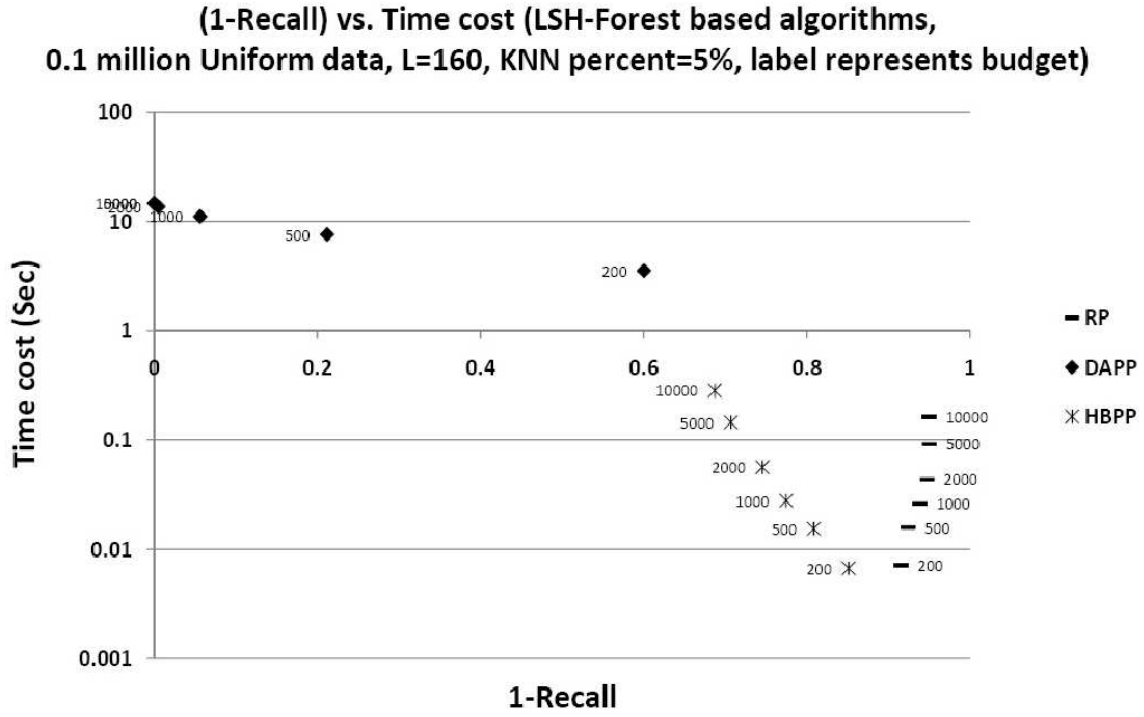

Figure 6.8: (1-Recall) vs. Time cost results of LSH-Forest based algorithms with 0.1 million Uniform Data

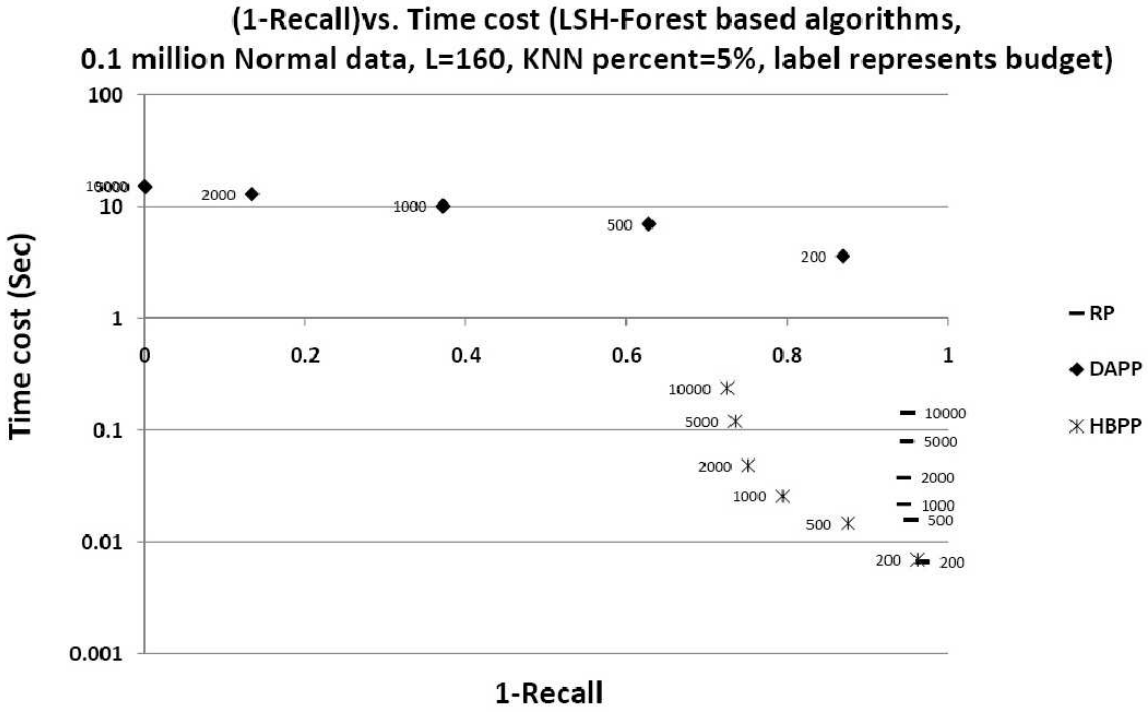

Figure 6.9: (1-Recall) vs. Time cost results of LSH-Forest based algorithms with 0.1 million Normal Data

cost" curves in each of the four figures.

As shown in Figures 6.6, 6.7, 6.8 and 6.9, there are several similar patterns as previous subsections: (1) For LSH-Forest with DAPP and LSH-Forest with HBPP, we can achieve higher recall (better search quality) by increasing budget size (thus increasing time cost). However, for LSH-Forest with RP, recall decreases as the budget size increases. (2) Given the same budget size, LSH-Forest with HBPP can achieve much higher recall (better search quality) than LSH-Forest with RP, with a little more time cost. (3) To achieve the same recall, LSH-Forest with DAPP requires at least an order of magnitude more time cost than LSH-Forest with HBPP. Especially for GIST Data in Figure 6.7, the time gain is even more, and the reason is explained in the previous subsection.

We also note that for LSH-Forest with HBPP, the best recall it can achieve for the two real data sets (SIFT Data, GIST Data) is even higher than the two synthetic data sets (Uniform Data, Normal Data).

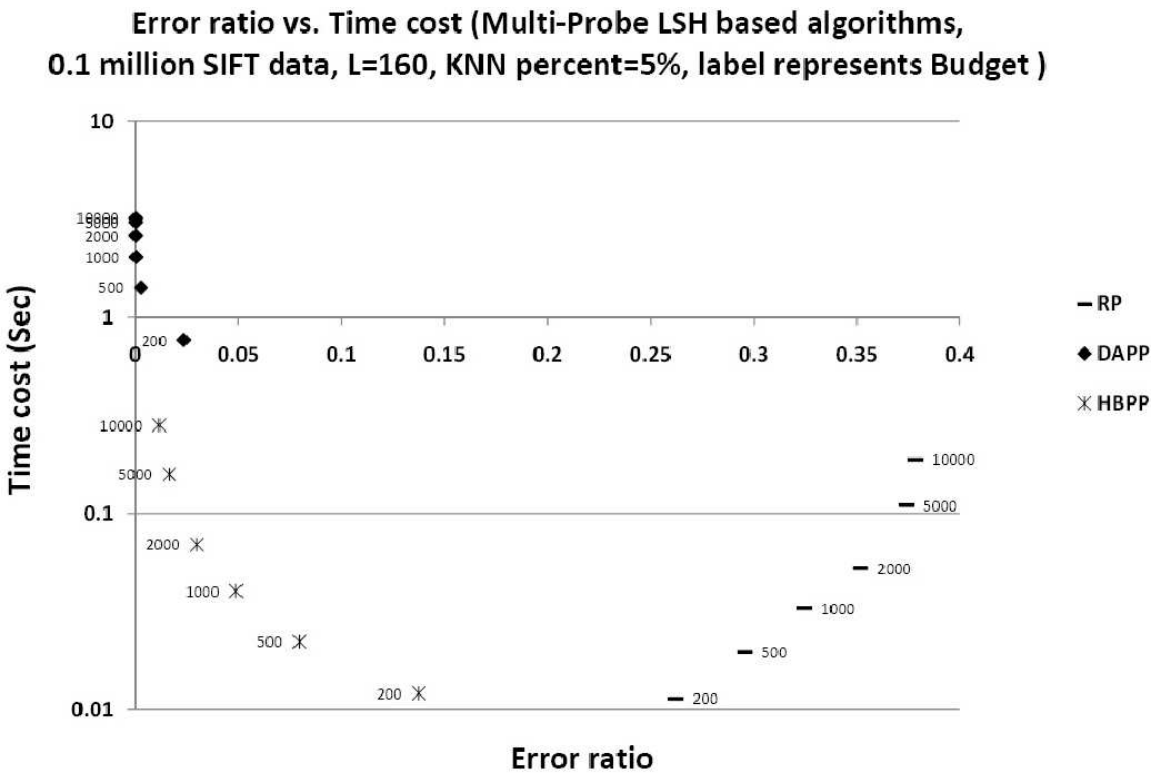

Figure 6.10: Error ratio vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million SIFT Data

Multi-Probe LSH based Algorithms Evaluated based on Error Ratio

In Figure 6.10, 6.11, 6.12 and 6.13, we show the performances of three Multi-Probe LSH based algorithms (Multi-Probe LSH with RP, Multi-Probe LSH with DAPP and Multi-Probe LSH with HBPP) by using four different data sets.

In Figures 6.10, 6.11, 6.12 and 6.13, every curve represents an "*Error ratio vs. Time cost*" trend of one of the three Multi-Probe LSH based algorithms. We vary the time cost of each algorithm by budget size, and a larger budget size means more time cost. One the other hand, a larger budget size also decreases the error ratio (except for Multi-Probe LSH with RP algorithm, in which case the error ratio may increase). Thus, we get the "*Error ratio vs. Time cost*" curves in each of the four figures.

As shown in Figures 6.10, 6.11, 6.12 and 6.13, for Multi-Probe LSH with DAPP and Multi-Probe LSH with HBPP, we can achieve better search quality by increasing

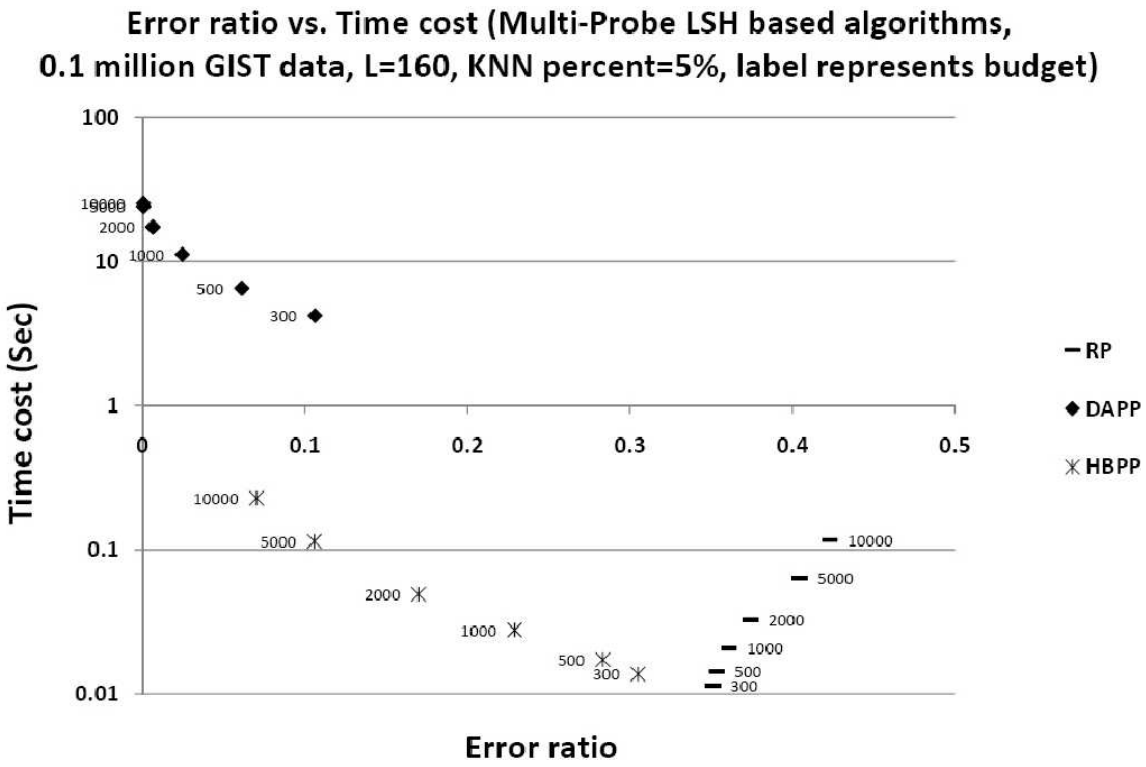

Figure 6.11: Error ratio vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million GIST Data

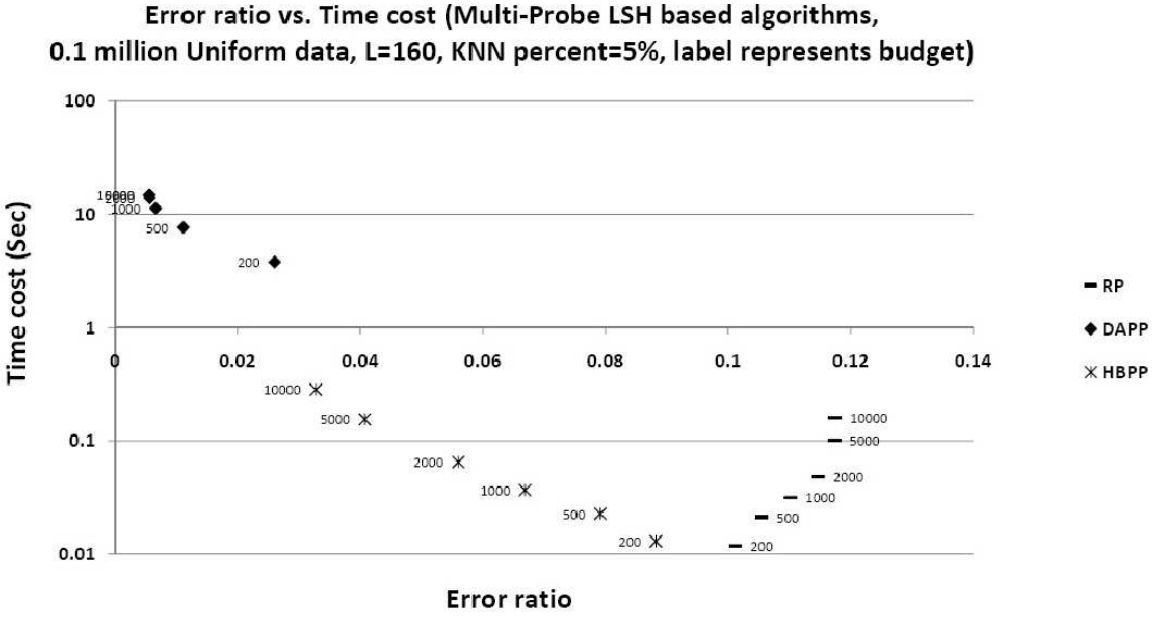

Figure 6.12: Error ratio vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million Uniform Data

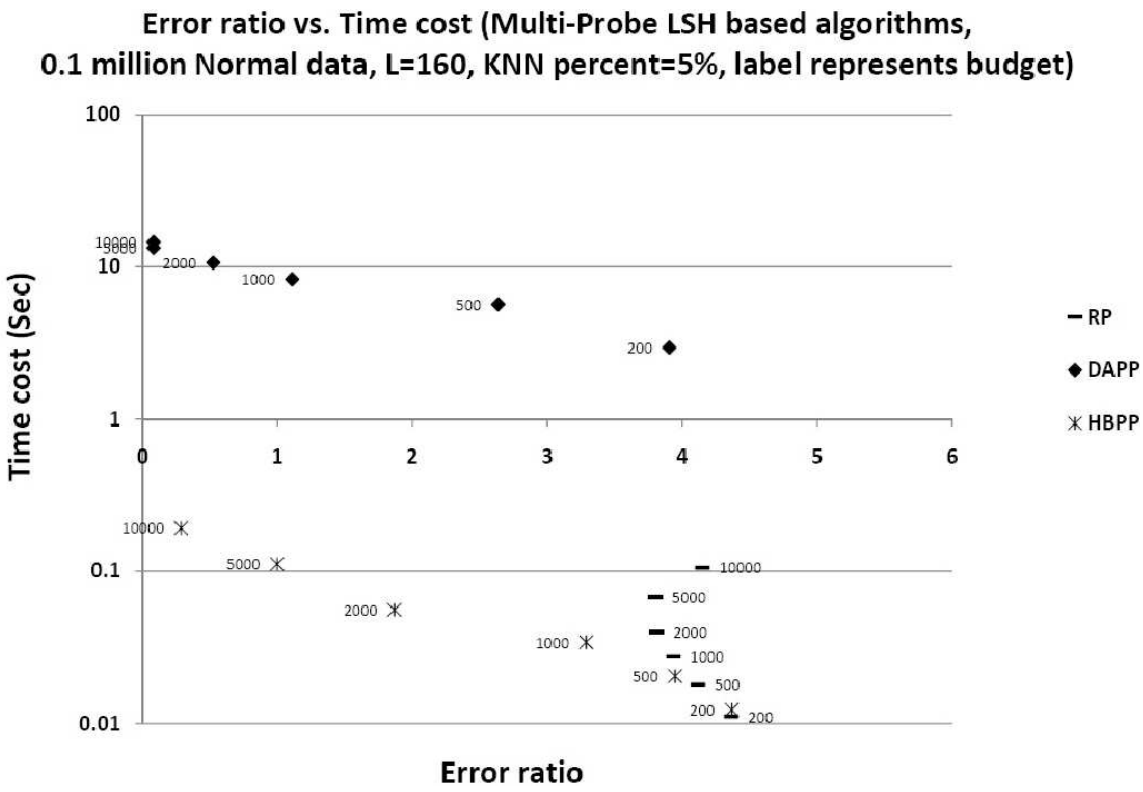

Figure 6.13: Error ratio vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million Normal Data

the budget size (thus increasing the time cost). However, for Multi-Probe LSH with RP, search quality becomes worse as the budget size increases. The reason is that more candidate objects also means higher percentage of false positive objects in the final result if there is no effective pruning in the post-processing step. We also notice two interesting patterns: (1) Given the same budget size, Multi-Probe LSH with HBPP can achieve much lower error ratio (better search quality) than Multi-Probe LSH with RP with a little more time cost (this also shows Multi-Probe LSH with HBPP is very time efficient). (2) To achieve the same error ratio (search quality), Multi-Probe LSH with HBPP is much faster than Multi-Probe LSH with DAPP.

As we can see from these four figures, to achieve the same search quality (error ratio), Multi-Probe LSH with DAPP requires at least an order of magnitude more time cost than Multi-Probe LSH with HBPP. Especially for GIST Data in Figure 6.11, the time gain is even more. The reason is explained in subsection "*LSH-Forest based Algo-*

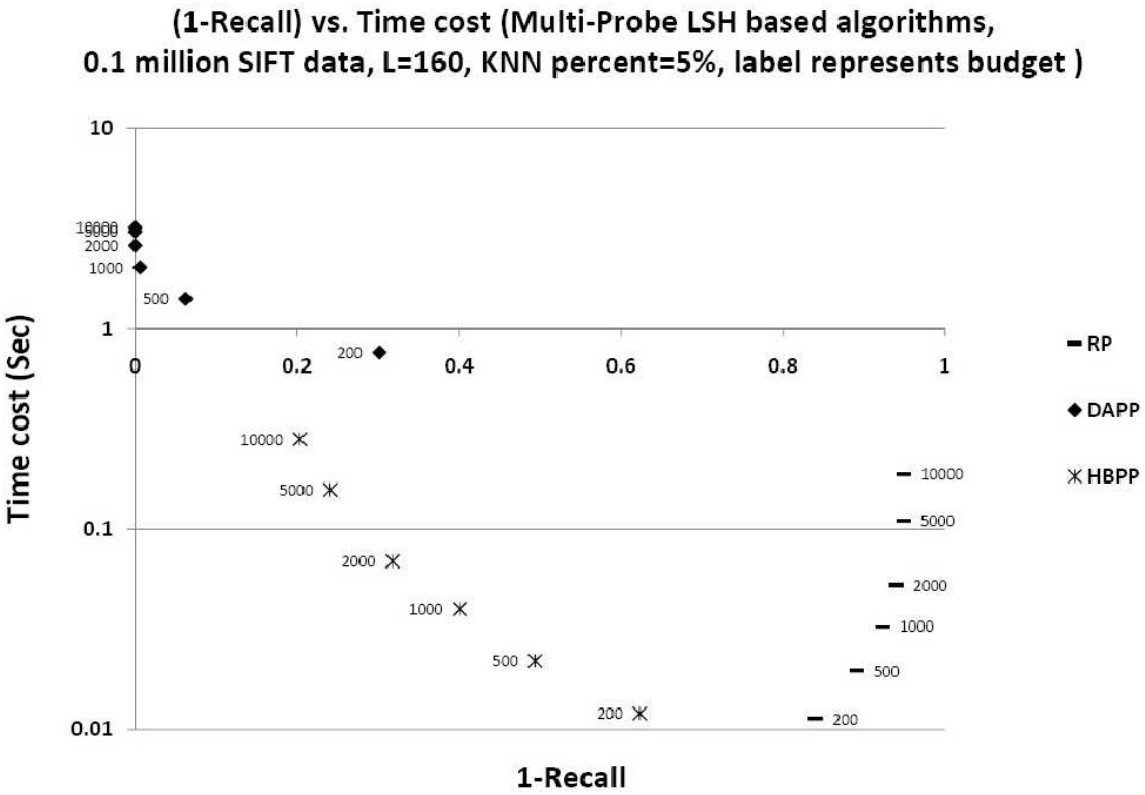

Figure 6.14: (1-Recall) vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million SIFT Data

rithms Evaluated based on Error Ratio". It is another evidence of the nice dimension scalability of our HBPP methods. We also observe that given the same budget size, the time cost of Multi-Probe LSH with DAPP algorithm for SIFT Data is much less than those for Uniform Data and Normal Data. The reason is also described in subsection "*LSH-Forest based Algorithms Evaluated based on Error Ratio*".

We also note that Figure 6.13 is quite different from other three figures. In Figure 6.13, the error ratios of all the curves go beyond 1, they even go beyond 5 in the worst case. However, for all other three figures, the error ratios never go beyond 1. The reason is the same as that is mentioned in subsection "*LSH-Forest based Algorithms Evaluated based on Error Ratio*". Figure 6.13 is another good example to show that our HBPP methods perform quite well even for skewed data set.

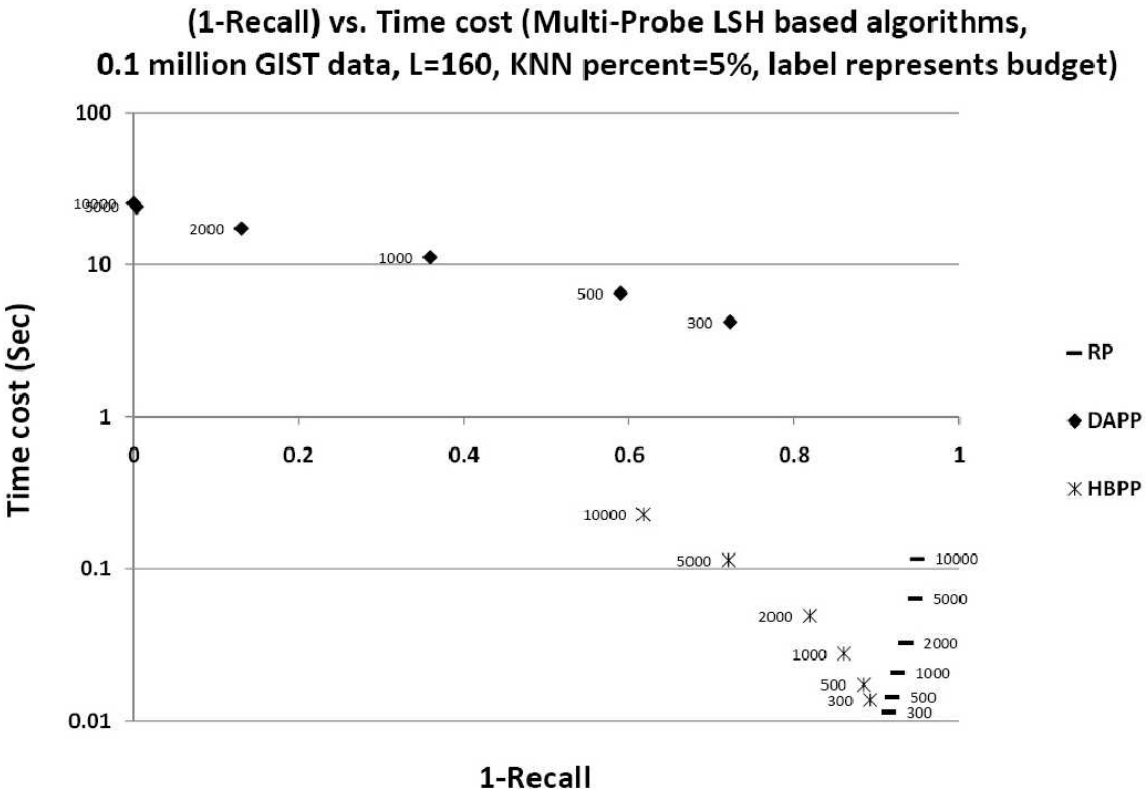

Figure 6.15: (1-Recall) vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million GIST Data

Multi-Probe LSH based Algorithms Evaluated based on Recall

In this subsection, we evaluated the three Multi-Probe LSH based algorithms with four different data sets. All the setups are the same as the previous subsection, except that the search quality is evaluated by recall instead of error ratio.

In Figures 6.14, 6.15, 6.16 and 6.17, every curve represents a "*(1-Recall) vs. Time cost*" trend of one of the three Multi-Probe LSH based algorithms. The time cost of each algorithms is varied by changing the budget size. The value $(1-Recall)$ can be decreased by increasing the budget size (except for Multi-Probe LSH with RP algorithm, in which case the value $(1-Recall)$ may increase). Thus, we get the "*(1-Recall) vs. Time cost*" curves in each of the four figures.

As shown in Figures 6.14, 6.15, 6.16 and 6.17, there are several similar patterns as previous subsections: (1) For Multi-Probe LSH with DAPP and Multi-Probe LSH

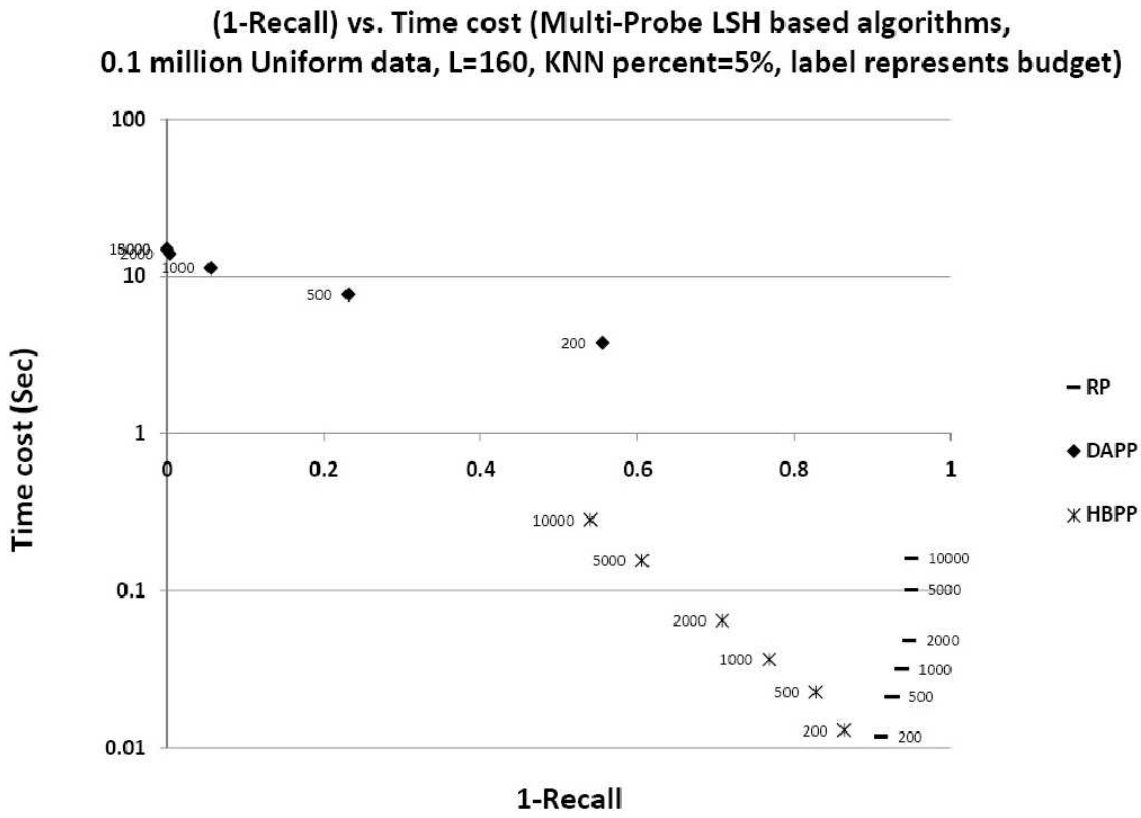

Figure 6.16: (1-Recall) vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million Uniform Data

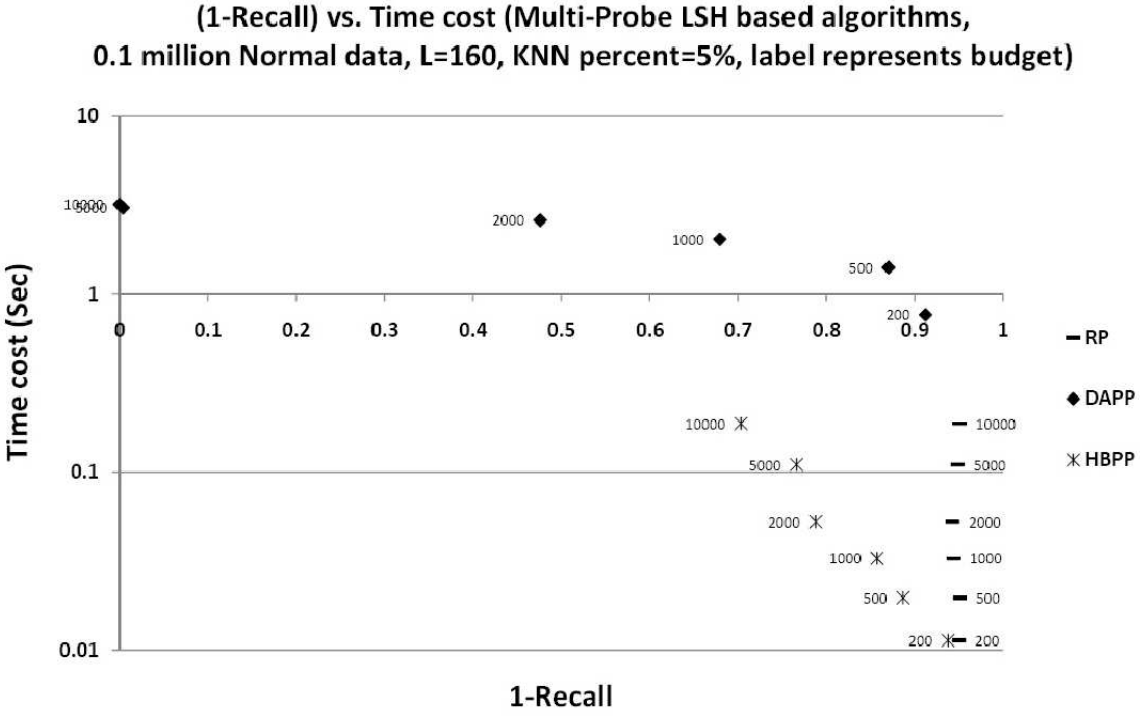

Figure 6.17: (1-Recall) vs. Time cost results of Multi-Probe LSH based algorithms with 0.1 million Normal Data

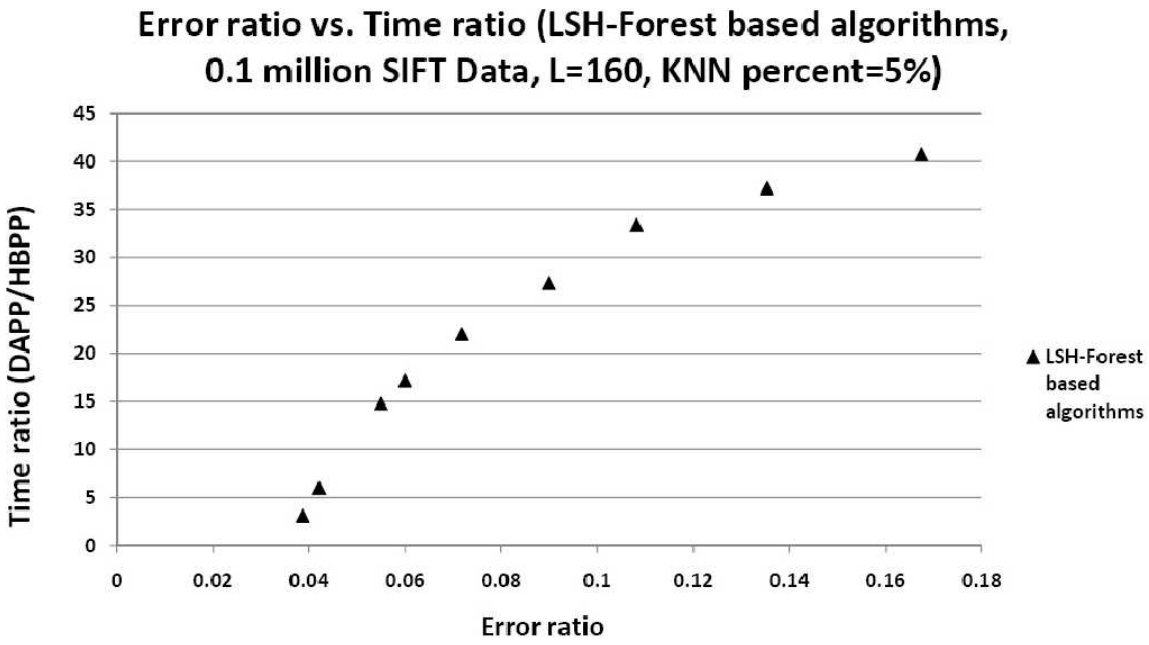

Figure 6.18: Error ratio vs. Time ratio (DAPP/HBPP) results of LSH-Forest based algorithms with 0.1 million SIFT Data

with HBPP, we can achieve higher recall (better search quality) by increasing the budget size (thus increasing time cost). However, for Multi-Probe LSH with RP, recall decreases as the budget size increases. (2) Given the same budget size, Multi-Probe LSH with HBPP can achieve much higher recall (better search quality) than Multi-Probe LSH with RP, with a little more time cost. (3) To achieve the same recall, Multi-Probe LSH with DAPP requires at least an order of magnitude more time cost than Multi-Probe LSH with HBPP. Especially for GIST Data in Figure 6.15, the time gain is even more. The reason is explained in subsection "*LSH-Forest based Algorithms Evaluated based on Error Ratio*".

Summary of Efficiency and Effectiveness Results

After checking all the figures in Section 6.1, we observe that for the same data set and memory usage, Multi-Probe LSH with HBPP always outperforms LSH-Forest with HBPP (e.g. Figure 6.1 and 6.10). The reason is as follows: given the same amount of memory usage and number of LSH trees/tables, the initial size of buckets in Multi-Probe LSH based algorithms is smaller than that in LSH-Forest based algorithms (LSH

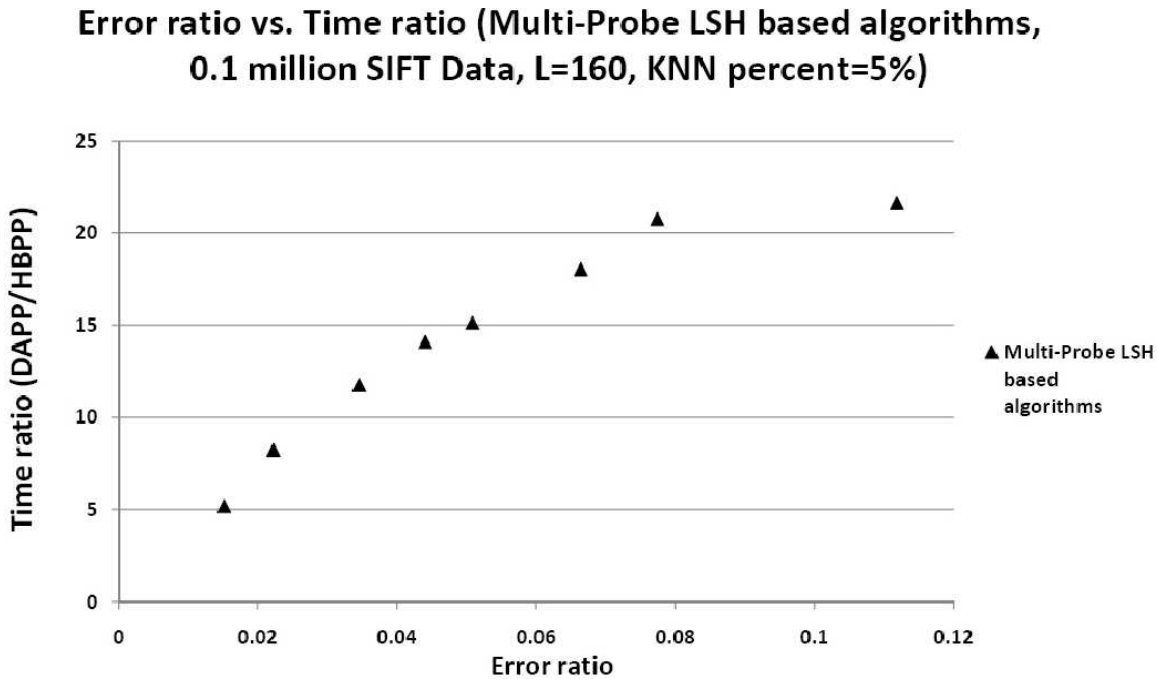

Figure 6.19: Error ratio vs. Time ratio (DAPP/HBPP) results of Multi-Probe LSH based algorithms with 0.1 million SIFT Data

tree has many internal nodes which cost non-negligible amount of memory, so we can not set the depth of the LSH tree big enough to make the initial size of buckets as small as in Multi-Probe LSH based algorithms), so Multi-Probe LSH based algorithms are able to collect candidates more precisely. As a result, Multi-Probe LSH wins when we want to achieve very high search quality.

Next, we are interested in answering following questions: to achieve the same error ratio, how much time gain HBPP has compared to DAPP? How does the error ratio affect the time gain? We answer these questions in Figure 6.18 and 6.19. In Figure 6.18, we show the "*Error ratio vs. Time ratio*" pattern for LSH-Forest based algorithms (DAPP and HBPP). As shown in the figure, to achieve a reasonable low error ratio, HBPP has at least one order of magnitude time gain. As the error ratio increases, the time gain also increases. In Figure 6.19, we show the "*Error ratio vs. Time ratio*" pattern for Multi-Probe LSH based algorithms (DAPP and HBPP), and the pattern is very similar as in Figure 6.18.

In a word, the two algorithms using our histogram-based post-processing method (LSH-Forest with HBPP, Multi-Probe LSH with HBPP) perform much better than other four algorithms. With the same time cost and memory usage, LSH-Forest with HBPP and Multi-Probe LSH with HBPP can achieve much higher search quality than LSH-Forest with RP and Multi-Probe LSH with RP. To achieve the same search quality with same amount of memory usage, LSH-Forest with HBPP and Multi-Probe LSH with HBPP need much less time cost than LSH-Forest with DAPP and Multi-Probe LSH with DAPP. And the experiment results show that if we want to achieve very high search quality, Multi-Probe LSH with HBPP algorithm is a better choice than LSH-Forest with HBPP algorithm.

6.2 Scalability Results

In this section, the scalability of three LSH-Forest based algorithms and three Multi-Probe LSH based algorithms will be evaluated. In the experiment, we test SIFT Data of different sizes (10000, 100000, 1000000). The two smaller data sets are generated by picking randomly from the original 1 million SIFT Data, so that the data distribution information is kept the same for these three data sets. We define all other parameters: L is set to be 160, budget percent (budget percent means the ratio of budget size comparing to the full data set size. for example, if budget size is 5000 and full data set size is 100000, then budget percent is 5%) is set to be 5%, KNN percent is set to be 5%, the memory usage is same for all algorithms. The main purpose of this section is to show that as the data size increases, how the time cost of each algorithm scales.

Scalability Results for LSH-Forest based Algorithms

In Figure 6.20 and 6.21 we show the scalability results of three LSH-Forest based algorithms. As shown in these two figures, the time cost of each of the three LSH-Forest algorithms scales almost linearly with the data size. As the data size increases, the time costs of RP and HBPP increase faster than that of DAPP. However, even when data size is large (e.g., 1 million), HBPP still has significant time gain compared to DAPP. In

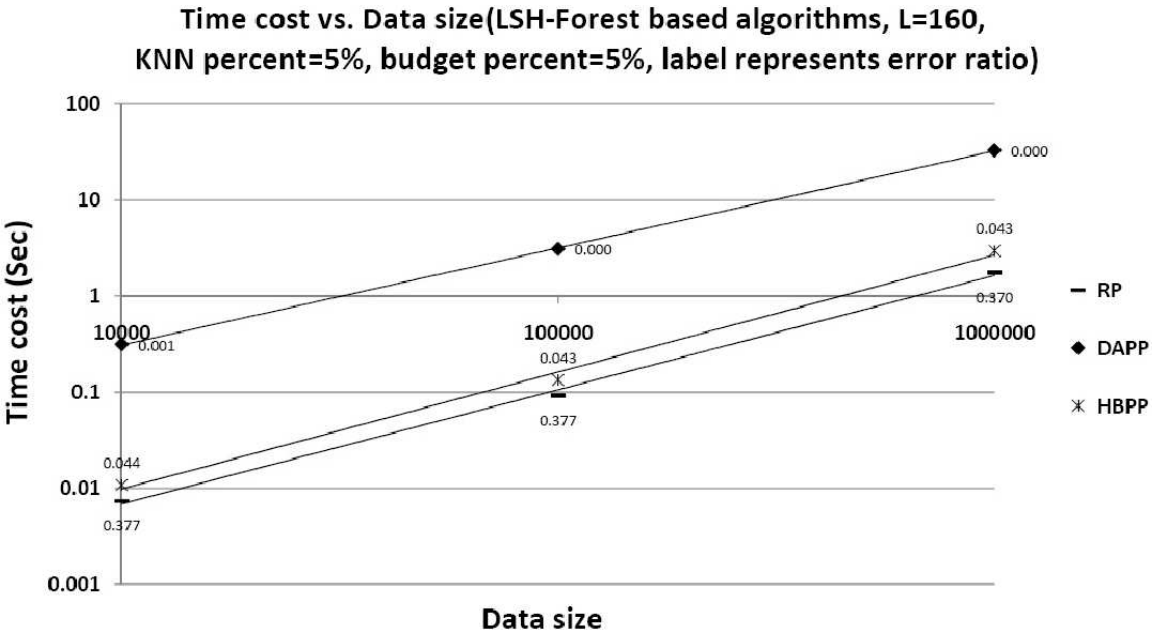

Figure 6.20: Scalability results of LSH-Forest based algorithms based on error ratio

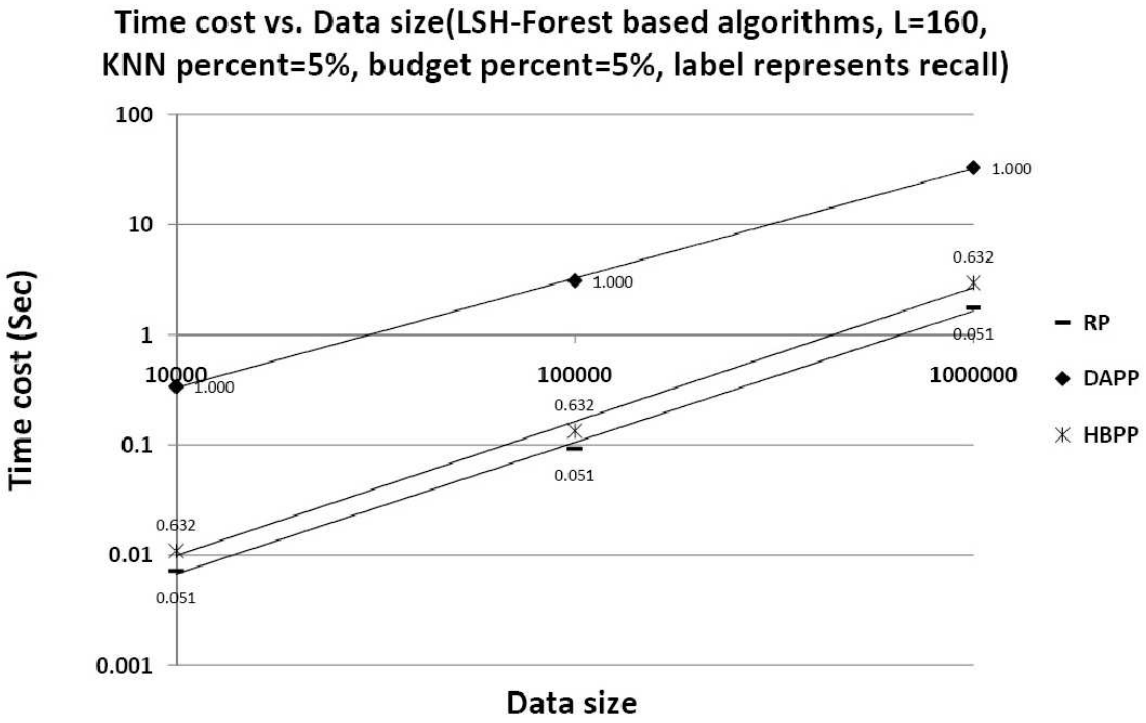

Figure 6.21: Scalability results of LSH-Forest based algorithms based on recall

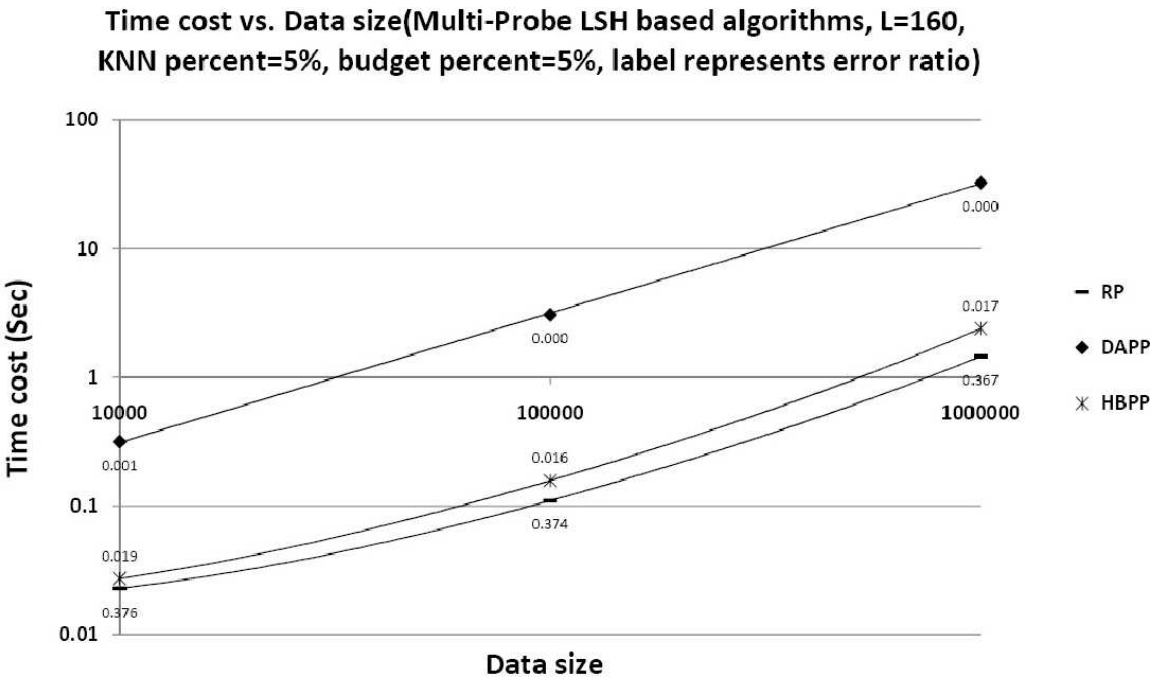

Figure 6.22: Scalability results of Multi-Probe LSH based algorithms based on error ratio

Figure 6.20, we also show the error ratio information on the label for all cases. As we can see, given the same budget percent, KNN percent, *L* and data distribution, the error ratio (search quality) is almost the same for each of the LSH-Forest based algorithms, no matter what the data size is. We also observe the same pattern in Figure 6.21, except that the search quality is represented by recall instead of error ratio.

Scalability Results for Multi-Probe LSH based Algorithms

In Figure 6.22 and 6.23 we show the scalability results of three Multi-Probe LSH based algorithms. As shown in these two figures, the time cost of DAPP scales almost linearly with the data size, and the time costs of RP and HBPP scale a little more than linearly with the data size. The reason is that as data size increases, the density of the space also increases, thus the number of buckets in every hash table increases (In LSH-Forest, the number of buckets in each LSH tree keeps the same as long as the depth is fixed, no matter the data size increases or not). For Multi-Probe LSH based algorithms, as the dataset size increases, more buckets need to be probed in order to fetch the same percent of candidates in each hash table. As a result, the time costs of HBPP and RP

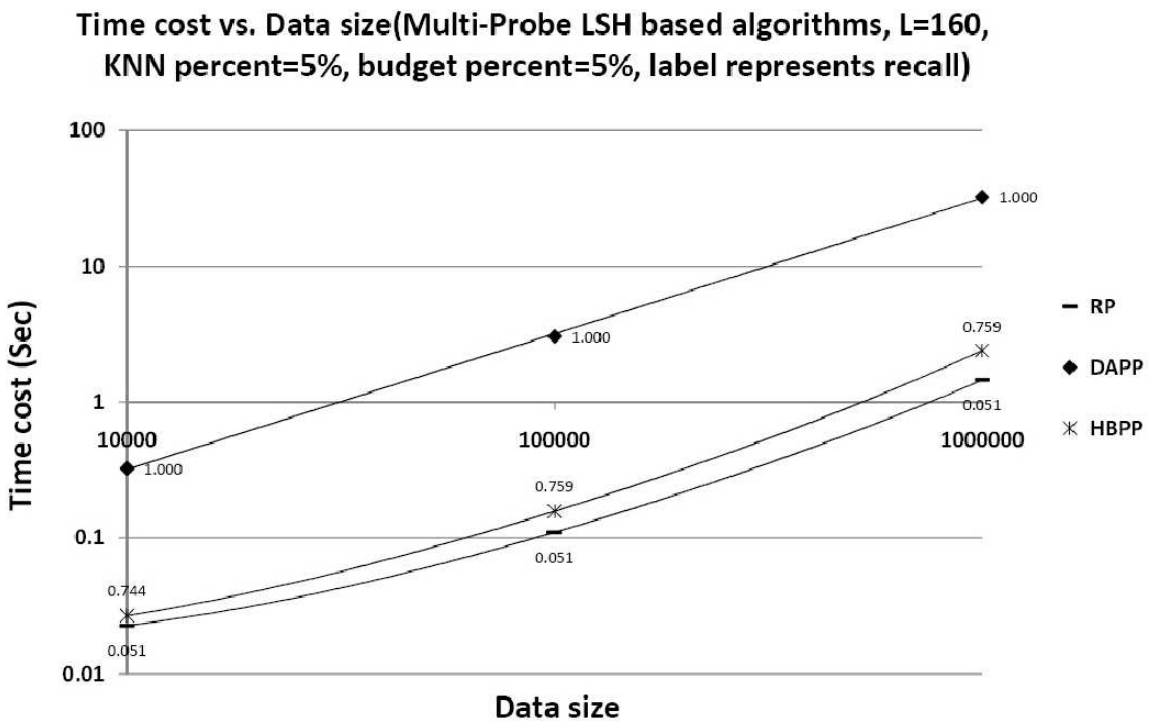

Figure 6.23: Scalability results of Multi-Probe LSH based algorithms based on recall

have more than linear scale with the data size. The reason why we do not observe the same pattern in DAPP is that the time cost of bucket fetching is only a very small portion of the total time cost of DAPP. We also observed that even when data size is large, HBPP still has significant time gain compared to DAPP. In Figure 6.20, we also show the error ratio information on the label in all cases. As we can see, given the same budget percent, KNN percent, L and data distribution, the error ratio (search quality) is almost the same for each of the Multi-Probe LSH based algorithms, no matter what the data size is. We also observe the same pattern in Figure 6.23, except that the search quality is represented by recall instead of error ratio.

6.3 Sensitivity Results

Now we have shown that our HBPP algorithms (LSH-Forest with HBPP and Multi-Probe LSH with HBPP) perform quite well compared to other algorithms. We want to find out what affects the search quality of our HBPP algorithms and how they affect the search quality. Since Multi-Probe LSH with HBPP always performs better (as shown in Section 6.1) than LSH-Forest with HBPP when we want to achieve very high

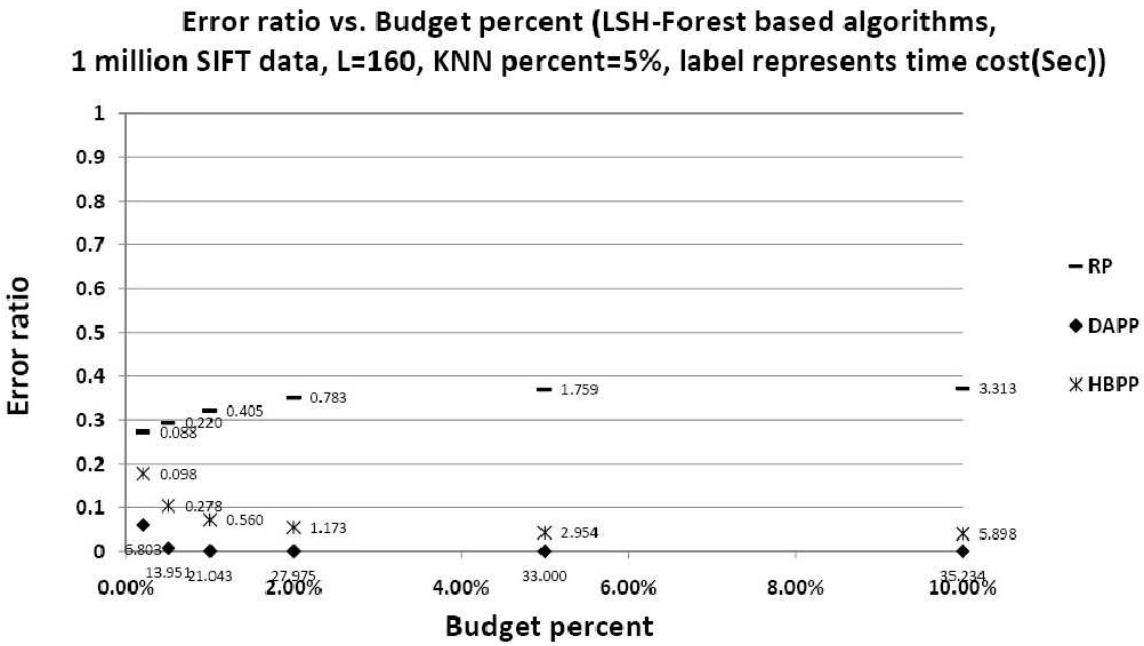

Figure 6.24: Error ratio vs. Budget percent of each LSH tree. $L = 160$, L means the number of LSH trees, KNN percent=5% means the result size of KNN query is 5%*1 million=50000

search quality (low error ratio or high recall), the latter one and its relative algorithms (three LSH-Forest based algorithms) will be used as a worst case example to show the sensitivity results.

Search Quality vs. Budget Percent

The first question is that for a given KNN query, if the budget percent (ratio of budget size from each LSH tree) increases (thus the combined candidate set size also increases), will the search quality be improved for each of the LSH-Forest based algorithms (LSH-Forest with RP, LSH-Forest with DAPP and LSH-Forest with HBPP)?

We give the answer in Figures 6.24 and 6.25. In Figure 6.24, the horizontal dimension of the figure represents percent of candidates collected from each LSH tree compared to the full data set size (i.e., budget percent, if we collect 20000 objects from each LSH tree, and total data size is 1 million, then budget percent=2%); the vertical dimension represents the error ratio, which is inversely related to the search quality. We use 1 million SIFT Data as experiment data and set $L = 160$ (L is the number of

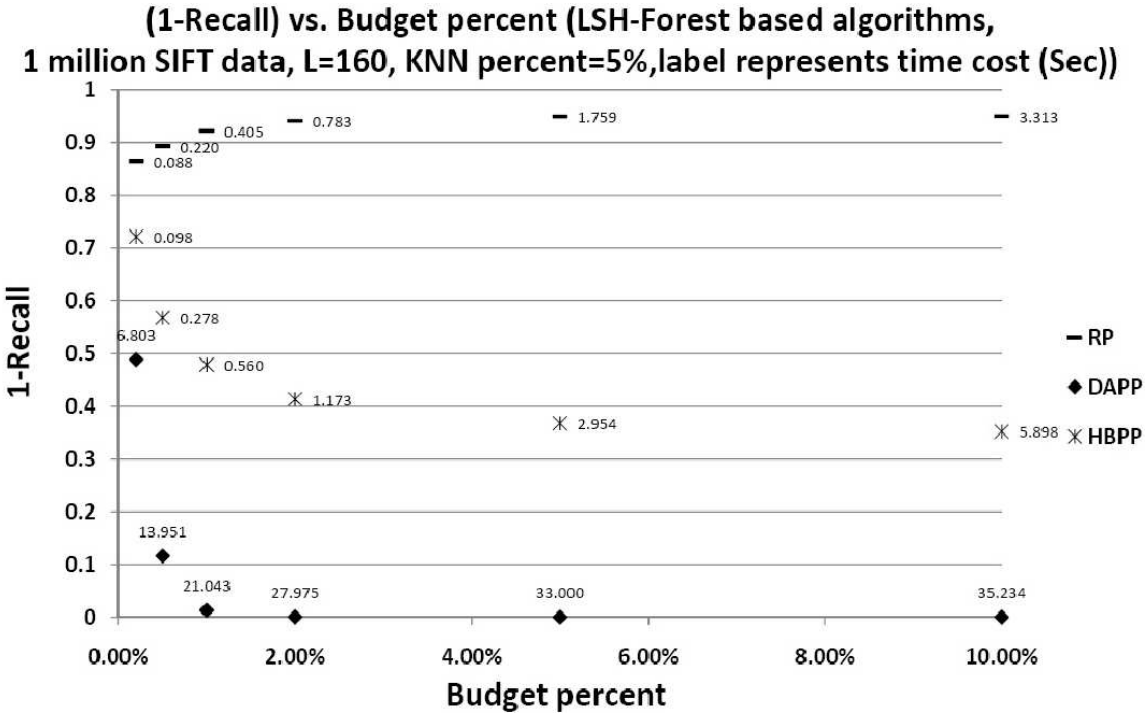

Figure 6.25: (1-Recall) vs. Budget percent of each LSH tree. $L = 160$, L means the number of LSH trees, KNN percent=5% means the result size of KNN query is 5%*1 million=50000

LSH trees), KNN percent=5%. As shown in Figure 6.24, for LSH-Forest with DAPP and HBPP, the search quality increases as the budget percent increases; however, for LSH-Forest with RP, the error ratio decreases as the budget percent increases. The reason is that for algorithms with effective post-processing steps (e.g., LSH-Forest with DAPP and LSH-Forest with HBPP) which are able to prune false positive objects, more candidates mean fewer misses in the final KNN result; however, for algorithms without effective post-processing steps (e.g., LSH-Forest with RP), more candidates means more false positives in the final KNN result. All the setups of Figure 6.25 are the same as in Figure 6.24, except that the search quality is represented by recall instead of error ratio. The "*Search quality vs. Budget percent*" pattern in Figure 6.25 is the same as we found for Figure 6.24.

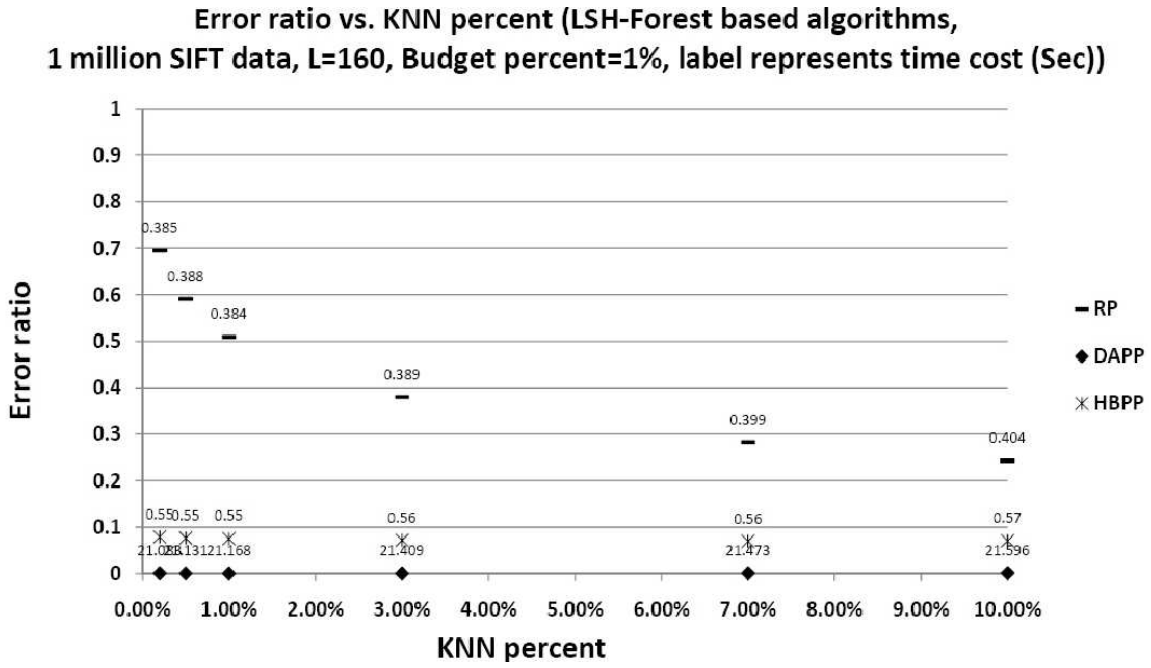

Figure 6.26: Error ratio vs. KNN percent. $L = 160$, L means the number of LSH trees

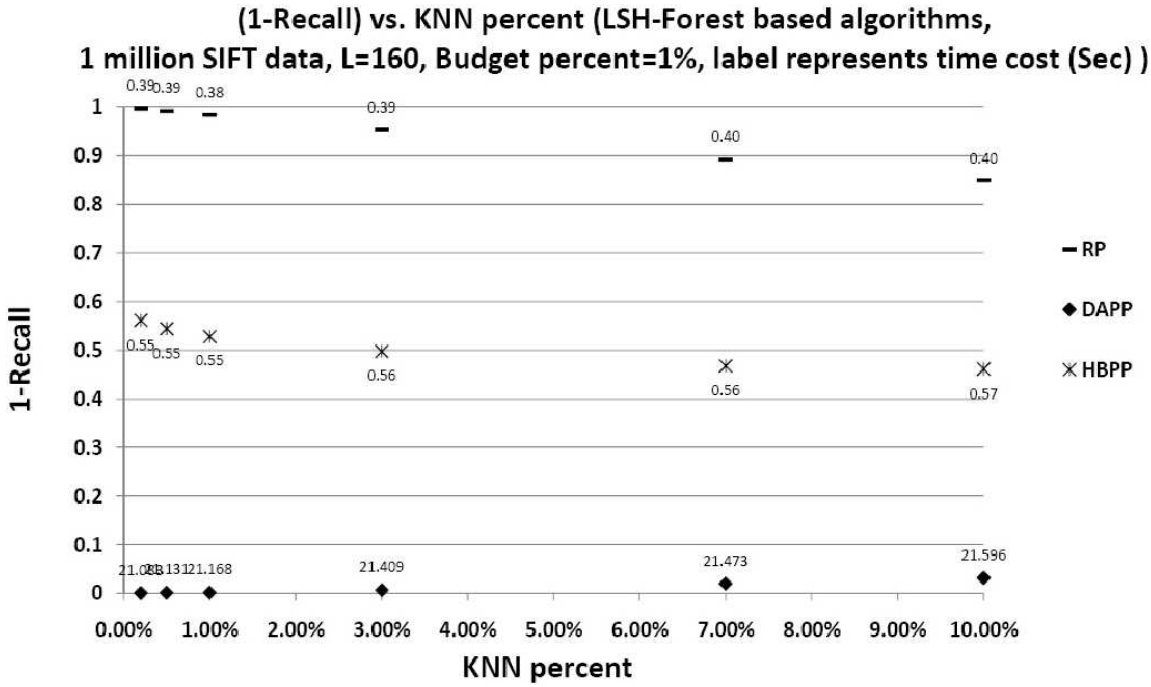

Figure 6.27: (1-Recall) vs. KNN percent. $L = 160$, L means the number of LSH trees

Search Quality vs. KNN Percent

Now our question is: if the result size of the KNN query is increased and all other parameters are fixed, will the search quality be improved for each of the LSH-Forest based algorithms?

We give the answer in Figures 6.26 and 6.27. In Figure 6.26, the horizontal dimension represents KNN percent (percent of KNN result size compared to the full data size, e.g., KNN percent=5% means the size of KNN result is 50000 if the data set size is 1 million); the vertical dimension represents the error ratio, which is inversely related to the search quality. We use 1 million SIFT Data as experiment data set and set set $L = 160$, budget percent=1%. In Figure 6.27, all the setups are the same as in Figure 6.26, except that the search quality is evaluated by recall instead of error ratio. There are several interesting patterns shown in Figure 6.26 and 6.27: (1) For LSH-Forest with RP, both the error ratio and (1-recall) decrease as KNN percent increases; the reason is that as the KNN percent increases and candidate set remains the same (since budget size keeps the same), the possibility that a randomly picked object is a real KNN objects also increases, so the recall increases and the error ratio drops. (2) For LSH-Forest with HBPP, recall increases as KNN percent increases; the reason is that as the KNN percent increases and candidate set keeps the same, the percent of false positives in the candidate sets decreases; it helps to decrease the false positive percent of the final KNN result and increase recall. On the other hand, since the error ratio is already very low when KNN percent is small, so the small increase of recall does not help much to decrease the error ratio. As a result, the error ratio keeps almost the same as KNN percent increases. (3) For LSH-Forest with DAPP, recall decreases as KNN percent increases; the reason is that as the KNN percent increases and candidate set keeps the same, the missing ratio of the real KNN objects also increases. So for an algorithm like LSH-Forest with DAPP which is able to find all the existing real KNN

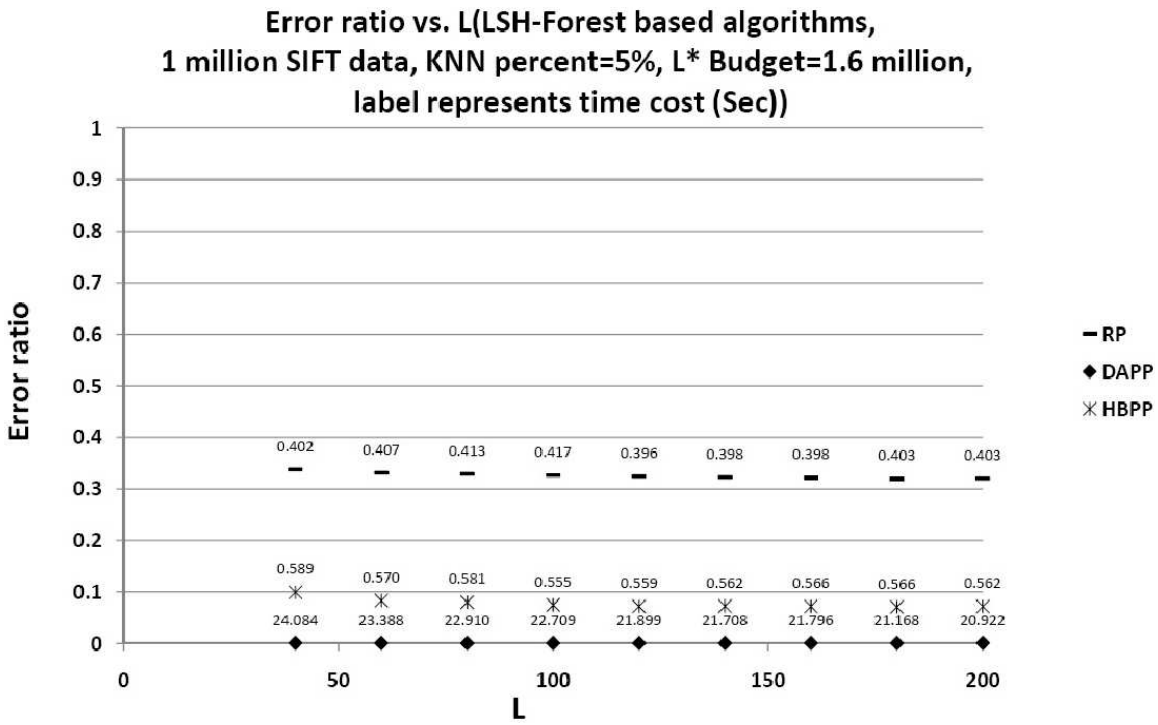

Figure 6.28: Error ratio vs. L (number of LSH trees). Budget means the number of candidate objects that are collected from each LSH tree, so L*Budget means the total number of candidate objects we collect (includes duplicate objects)

objects in the candidate set, increase of missing ratio means decrease of recall (it is not true for LSH-Forest with HBPP since it can not find all the real KNN objects in the candidate set). On the other hand, error ratio keeps the same as KNN percent increases; it is also because the error ratio is already very low when KNN percent is small, so the small decrease of recall does not affect error ratio much.

Search Quality vs. Number of LSH trees

Another question we are interested in answering is: given that all other parameters are fixed (e.g., KNN percent), will the increment of memory usage (i.e., space complexity, the number of LSH trees in our case) improve search quality for each of the LSH-Forest based algorithms?

We give our answer in Figure 6.28 and 6.29. In Figure 6.28, the horizontal dimension of the figure represents *L*, the number of LSH trees; the vertical dimension represents the error ratio. We use 1 million SIFT Data as experimental data set and set

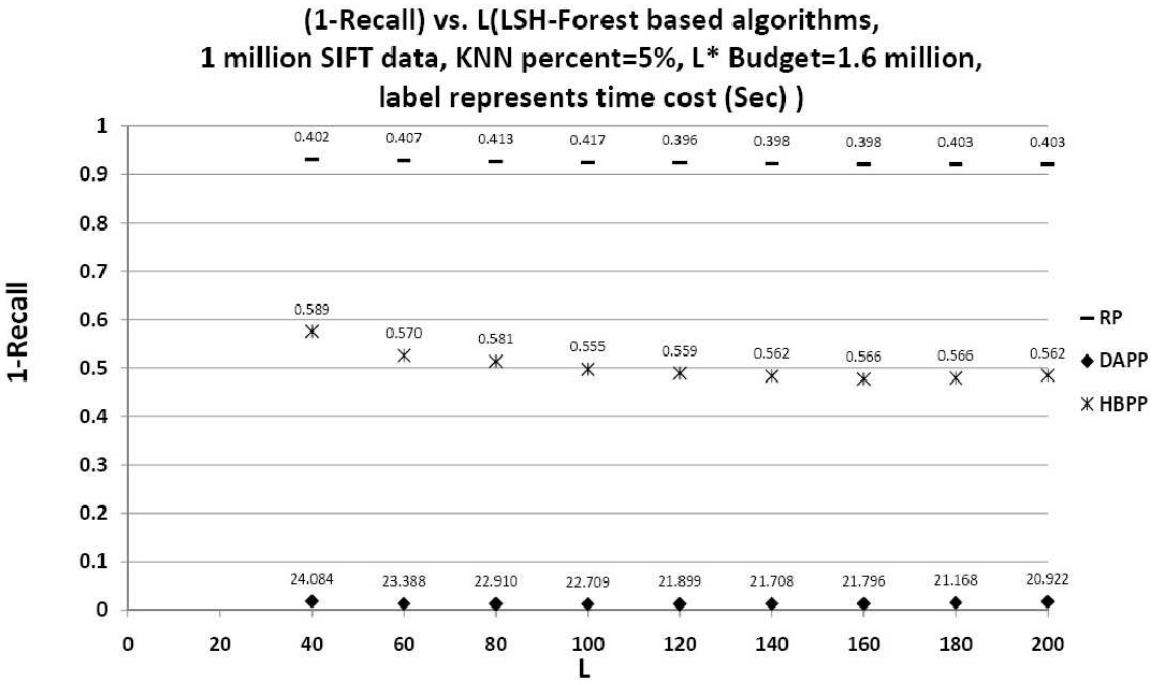

Figure 6.29: (1-Recall) vs. L (number of LSH trees). Budget means the number of candidate objects that are collected from each LSH tree, so L*Budget means the total number of candidate objects we collect (includes duplicate objects)

KNN percent=5%, and $L\times$ budget size=1.6 million. So as L increases, the budget size will decrease. There are several interesting patterns shown in Figure 6.28 and 6.29: (1) For LSH-Forest with RP, both error ratio and (1-recall) decrease as L increases; the reason is that M decreases as L increases, we collect candidate objects more precisely from more trees, so the search quality increases. As a result, error ratio and (1-recall) decrease. (2) For LSH-Forest with HBPP, recall increases as L increases; the reason is the same as for LSH-Forest with RP. However, error ratio does not change much as L increases; the reason is that error ratio is already very low when L is small, so the small increase of recall does not help much to decrease the error ratio. As a result, the error ratio remains almost the same as L increases. (3) For LSH-Forest with DAPP, error ratio and (1-recall) remain the same as L increases; that is because the search quality is already perfect when L is small, so increase of L does not help to increase search quality (reduce error ratio or (1-recall)) any more.

6.4 Model Comparison Results

In Section 4.3 of Chapter 4, we describe the mathematical model of LSH-Forest with HBPP algorithm and give 3 scenarios to verify the correctness of our model:

1. Given R, r, N, M, L, what is the relation between recall and K (KNN size)?

2. Given R, r, N, K, L, what is the relation between recall and M (budget size)?

3. Given R, r, N, M, K, what is the relation between recall and L (Number of LSH trees)?

Here r is the radius of the hyper-ball that has all the KNN objects (of the query) on its hyper-surface, R is the radius of the hyper-ball that has all the false positive points (of the query) on its hyper-surface, N is the size of the false positives, K is the size of the KNN result, L is the number of LSH trees, and M is the budget size of each LSH tree (i.e., the number of candidates collected from each LSH tree).

In the following 3 subsections, we will test the above mentioned 3 scenarios separately.

Recall vs. KNN Size

As shown in Figure 6.30, we show the "*Recall vs. KNN size*" patterns of LSH-Forst with HBPP algorithm for the model and SIFT Data. For both cases, we fix all other parameters except K (KNN size). In the experiment, we vary K from 2000 to 100000, and the recall changes accordingly. As we can see, the patterns of the model and real data are a little different, but the shape of these two lines are similar, and they both obey log scale.

The reason why the patterns of the model and SIFT Data are different is that in the model, we keep the parameter r the same as KNN size increases. However, in the real case, r changes as KNN size increases.

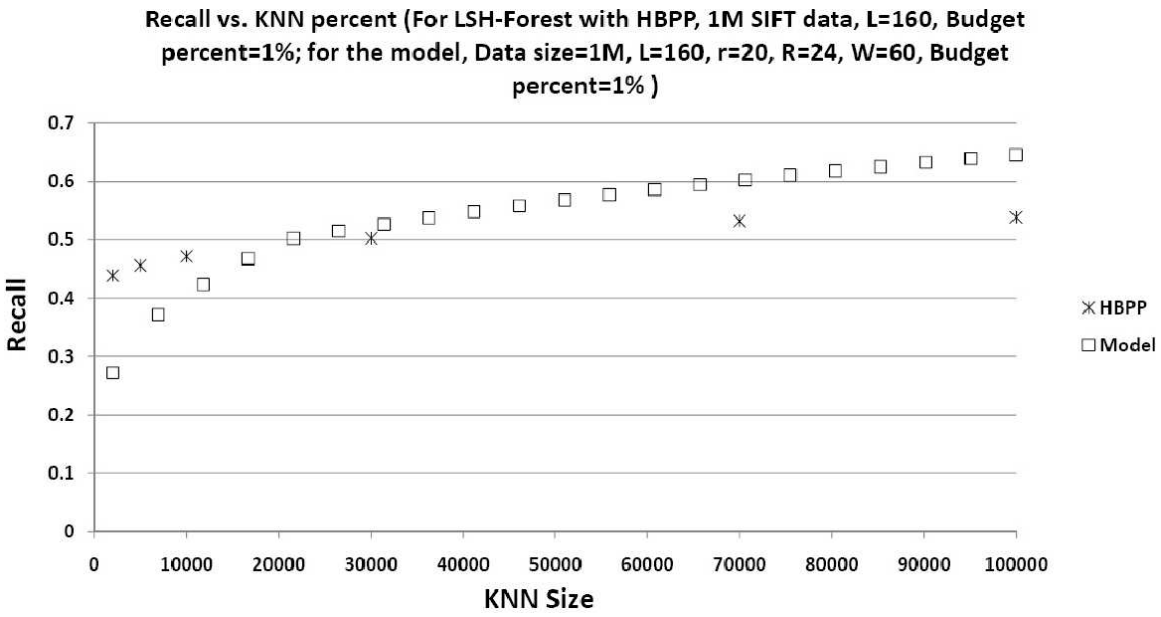

Figure 6.30: Recall vs. KNN size result (of the model and SIFT Data)

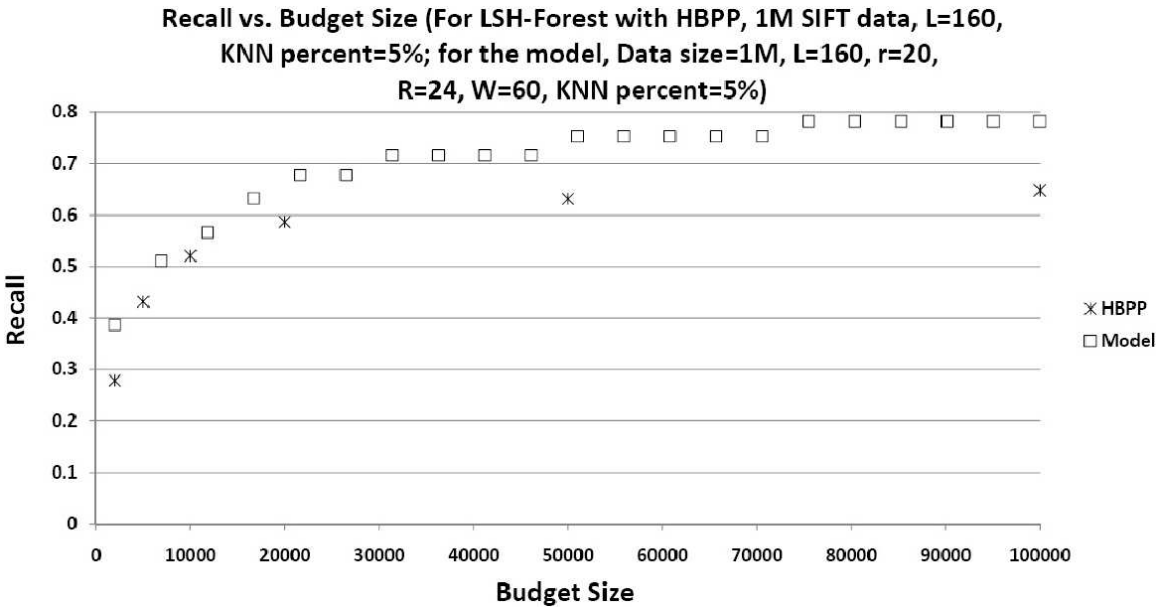

Figure 6.31: Recall vs. Budget size result (of the model and SIFT Data)

In the further comparison of other test scenarios, we will set the KNN size at 50000; the reason is that as long as the chosen KNN size does not cause big difference between the two patterns, it will not effect the comparison of other patterns much.

Recall vs. Budget Size

As shown in Figure 6.31, we show the "*Recall vs. Budget size*" patterns of LSH-Forst with HBPP algorithm for the model and SIFT Data. For both cases, we fix all other

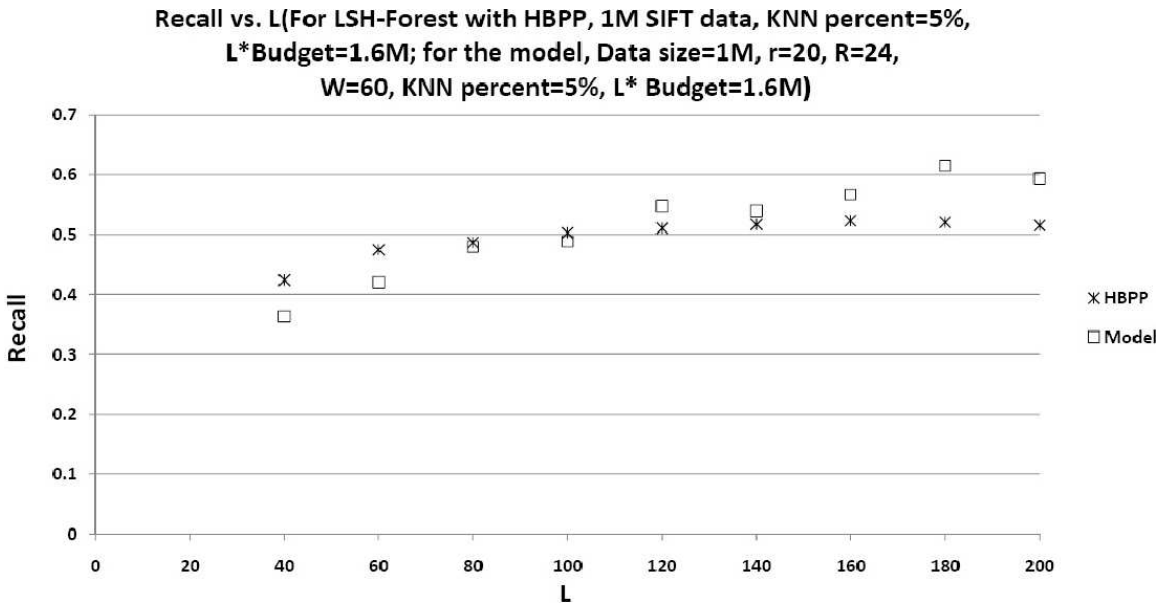

Figure 6.32: Recall vs. Number of LSH trees (of the model and SIFT Data)

parameters except M (budget size). In the experiment, we vary M from 2000 to 100000, and the recall changes accordingly. As we can see, pattern of the model matches that of SIFT Data quite well.

Recall vs. Number of LSH trees

As shown in Figure 6.32, we show the "*Recall vs. Number of LSH trees*" patterns of LSH-Forst with HBPP algorithm for the model and SIFT Data. For both cases, we fix all other parameters except L (number of LSH trees). In the experiment, we vary L from 40 to 200 (M also changes because we set $L \times M = 1600000$), and the recall changes accordingly. As we can see, the "*Recall vs. Number of LSH trees*" pattern of the model matches that of SIFT Data nicely.

Chapter 7

CONCLUSIONS

In this thesis, we improved the previous LSH-based KNN search algorithms (e.g. LSH-Forest with DAPP, Multi-Probe LSH with DAPP) which need a costly data access post-processing (DAPP) step. We achieve this by using a much faster histogram-based post-processing (HBPP) algorithm. We have shown the use of HBPP on two LSH algorithms: LSH-Forest and Multi-Probe LSH and showed that HBPP achieves the three goals for KNN search in large scale high dimensional data set: high search quality, high time efficiency, high space efficiency. As far as we know, none of the previous KNN algorithms can achieve all three goals. More specifically, relative to DAPP algorithms (LSH-Forest with DAPP and Multi-Probe LSH with DAPP), HBPP-based algorithms cost much less time to answer a KNN query with same search quality and memory usage; relative to random post selection (RP) algorithms (LSH-Forest with RP and Multi-Probe LSH with RP), HBPP-based algorithms achieve much higher search quality for a KNN query. Moreover, we observed that to achieve a high search quality, Multi-Probe LSH HBPP is better than LSH-Forest with HBPP, regardless of the distribution, size, and the number of dimensions of the data set.

References

[1] Alexandr Andoni and Piotr Indyk. Lsh user manual. http://www.mit.edu/andoni/LSH/manual.pdf.

[2] Antonio Torralba Aude Oliva. Modeling the shape of the scene: a holistic representation of the spatial envelope. http://people.csail.mit.edu/torralba/code/spatialenvelope/.

[3] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: Self-tuning indexes for similarity search. In *WWW*, pages 651–660. ACM Press, 2005.

[4] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[6] Jeremy Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[7] K. S. Candan and Maria Luisa Sapino. *Data Management for Multimedia Retrieval*. Cambridge University Press, Cambridge, UK, 2010.

[8] Jack G. Conrad, Xi S. Guo, and Cindy P. Schriber. Online duplicate document detection: signature reliability in a dynamic retrieval environment. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 443–452, New York, NY, USA, 2003. ACM.

[9] Scott Cost and Steven Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Mach. Learn.*, 10:57–78, January 1993.

[10] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[11] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. L, and Richard Harshman. Indexing by latent semantic analysis. *J. Am. Soc. Information Science*, pages 391–407, 1990.

[12] David Dobkin and Richard J.Lipton. Multidimensional searching problems. *SIAM Journal of Computing*, 2:181–186, 1976.

[13] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *J. Intell. Inf. Syst.*, 3:231–262, July 1994.

[14] Christos Faloutsos and Douglas W. Oard. A survey of information retrieval and filtering methods. Technical report, Department of Computer Science, University of Maryland, College Park, 1995.

[15] Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. Query by image and video content: The qbic system. *Computer*, 28:23–32, September 1995.

[16] Yanlan Gan, Jihong Guan, and Shuigeng Zhou. A pattern-based nearest neighbor search approach for promoter prediction using dna structural profiles. *Bioinformatics*, 25:2006–2012, 2009.

[17] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB Conference*, pages 518–529, 1997.

[18] Gist. Gist data. http://corpus-texmex.irisa.fr/.

[19] Ntonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[20] Trevor Hastie and Robert Tibshirani. Discriminant adaptive nearest neighbor classification. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18:607–616, June 1996.

[21] David Hilbert. Über die stetige abbildung einer linie auf ein flächenstck. *Mathematische Annalen*, 38:459–460, 1891.

[22] Piotr Indyk and Rejeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Symposium on Theory of Computing*, pages 604–613, 1998.

[23] Norio Katayama and Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *SIGMOD Conference*, pages 369–380, 1997.

[24] Caltech Computational Vision Lab. Caltech image data set. http://www.vision.caltech.edu/Image-Datasets/Caltech101/.

[25] Haiquan Li, Xinbin Dai, and Xuechun Zhao. A nearest neighbor approach for automated transporter prediction and categorization from protein sequences. *Bioinformatics*, 24:1129–1136, 2008.

[26] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–1157, Washington, DC, USA, 1999. IEEE Computer Society.

[27] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB Conference*, pages 950–961, 2007.

[28] Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada: IBM Ltd, 1966.

[29] Oracle. Berkeleydb. http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html.

[30] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 1186–1195, New York, NY, USA, 2006. ACM.

[31] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases, 1995.

[32] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[33] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publisher, 2006.

[34] Jenq-Haur Wang and Hung-Chi Chang. Exploiting sentence-level features for near-duplicate document detection. In *Proceedings of the 5th Asia Information Retrieval Symposium on Information Retrieval Technology*, AIRS '09, pages 205–217, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Shiyuan Wang and Kyle Chipman. Lsh-based indexing for similarity search on high-dimensional data. http://www.cs.ucsb.edu/ sywang/290D.htm.

[36] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB Conference*, pages 194–205, 1998.

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

[38] Bin Yao, Feifei Li, and Piyush Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *ICDE Conference*, pages 4–15, 2010.

[39] Qi Ye, Bin Wu, and Bai Wang. Distance distribution and average shortest path length estimation in real-world networks. In *Proceedings of the 6th international conference on Advanced data mining and applications: Part I*, ADMA'10, pages 322–333, Berlin, Heidelberg, 2010. Springer-Verlag.