QALSH User Manual

Qiang HUANG

Jianlin FENG

March 18, 2016

Contents

Chapter 1

Introduction

1.1 What's QALSH?

QALSH is a package written in the C++ programming language. It provides a randomized access method for the c-Approximate Nearest Neighbor (or simply c-ANN) search in the high dimensional Euclidean space, where c is an approximation ratio. QALSH is based on the Query-Aware Locality-Sensitive Hashing scheme described in [6].

1.2 Detail Description

Locality-Sensitive Hashing (LSH) [7, 2] and its variants [1, 8, 9, 4] are the well-known indexing schemes for c-ANN search in high-dimensional space. The LSH scheme was first introduced by Indyk and Motwani [7] for use in Hamming space $\{0,1\}^d$, and later was extended for use in the Euclidean space $\mathcal{R}^d$ by Datar et al.[2]. The key idea of LSH schemes is to make use of a set of "distance preserving" hash functions so that the collision probability of the "close" objects is much higher than that of the "far-apart" objects.

Traditional Locality-Sensitive Hashing schemes [7, 2, 1, 8] are the randomized methods for directly solving the (R, c)-Near Neighbor (or simply (R, c)-NN) problem, which is a decision version of the c-ANN search problem. It is known that the c-ANN search problem can be reduced to a series of (R, c)-NN problem with complexity $O(\log(n/\epsilon))$ [7, 5], where $c = 1+\epsilon$. Let $\|o_1 - o_2\|$ denote the Euclidean distance between two objects o_1 and o_2. Given a database D of objects in d-dimensional Euclidean space $\mathcal{R}^d$, the c-ANN search problem and (R, c)-NN problem can be defined as follows.

Definition 1 (c-ANN search) *Given an approximation ratio $c > 1$, the c-ANN search problem is to construct a data structure which for any query $q \in \mathcal{R}^d$ finds an object $o \in D$ such that $\|o - q\| \leq c\|o^* - q\|$, where o^* is the NN of q in D.*

Definition 2 ((R, c)-NN search) *Given an approximation ratio $c > 1$ and a search radius $R > 0$, the (R, c)-NN problem is to construct a data structure which for any query $q \in \mathcal{R}^d$ does the following:*

- *If there exists an object $o \in D$ s.t. $\|o - q\| \leq R$, then returns an arbitrary object $o' \in D$ s.t. $\|o' - q\| \leq cR$;*
- *If $\|o - q\| > cR$ for all $o \in D$, then return nothing;*
- *Otherwise, the result is undefined.*

Compared to traditional LSH schemes, QALSH solves the c-ANN search problem without the factor $O(\log(n/\epsilon))$, by using the virtual rehashing scheme. QALSH also solves the (R, c)-NN search problem.

Using the query-aware bucket partition, QALSH works with any approximation ratio $c > 1$, while the state-of-the-art LSH schemes for external memory, namely C2LSH [4] and LSB-Forest [9], only work with approximation ratio of integer $c \geq 2$. QALSH is shown to outperform C2LSH and LSB-Forest, especially in high-dimensional space [6]. Specifically, by using an approximation ratio $c < 2$, QALSH can achieve much better query quality.

The rest of the manual is organized as follows. Section 2 describes how to use the package and the format of the input/output files. In Section 3, we introduce the QALSH scheme, and then discuss the parameter settings of QALSH, as well as implementation details. We discuss the structure of the code of QALSH in Section 4. Finally, Chapter 5 contains FAQ.

Chapter 2

QALSH Usage

In this chapter, we first show how to compile the package. Then, we describe how to use the package of QALSH. Next, we discuss the memory usage of QALSH. Finally, we present the format of input and output files of QALSH.

2.1 Compilation

In the package of QALSH, we have already written the `Makefile`. Therefore, in order to compile QALSH, you can simply type `make` from the root directory of the package.

Notice that the package was default compiled with `g++ -std=c++11` with `-w -O3`, running under Linux. For the Microsoft Windows users, you first need to create a new project for QALSH on Microsoft Visual Studio and copy all files of `*.h` and `*.cpp` into the project. Then, you can compile QALSH under Visual C++ running with Microsoft Visual Studio.

2.2 Main Usage

We first describe the parameters of the package. There are ten parameters in the package of QALSH. The description of these parameters is displayed in Table 2.1.

Table 2.1: Description of the Parameters of QALSH

Parameters	Data Type	Description
-alg	integer	options of algorithms (0 - 3)
-d	integer	dimensionality of the dataset
-n	integer	cardinality of the dataset
-qn	integer	number of queries
-B	integer	page size
-c	real	approximation ratio ($c > 1$)
-ds	string	file path of the dataset
-qs	string	file path of the query set
-ts	string	file path of the ground truth set
-of	string	output folder to store output files of QALSH

Now, we describe the main algorithms of the package. There are four kinds of options of algorithms (i.e. -alg) in the package of QALSH:

Option 0: Ground-Truth. The usage of this option is described as follows:

```
-alg 0 -n -qn -d -ds -qs -ts
```

Under this option, we find the ground truth results (i.e., the exact Nearest Neighbors) of queries. The program would read the cardinality (i.e., `-n`) of dataset, the number of queries (i.e., `-qn`) and the dimension (i.e., `-d`) of data objects. The data objects and the queries will be loaded into memory from the file path of dataset (i.e., `-ds`) and the file path of query set (i.e., `-qs`). The ground truth results will be written to the ground truth file (i.e., `-ts`). Under this option, we assume the memory is free and large enough and all data objects are stored in the memory, so that we can find the exact NN in a quick manner.

Option 1: Indexing. The usage of this option is described as follows:

```
-alg 1 -n -d -B -c -ds -of
```

Under this option, we construct the hash tables of QALSH for c-ANN search. Except reading the cardinality (i.e., `-n`) and the dimension (i.e., `-d`) of the dataset, the program also reads the page size (i.e., `-B`) and the approximation ratio (i.e., `-c`). Under this option, we also assume the memory is large enough so that all the data objects and the hash tables could be stored in the memory. The program will write four kinds of files into the output folder (i.e., `-of`): the files of hash tables, a parameter file of QALSH, a new format of dataset, and an output file to record the wall-clock time of indexing.

Option 2: QALSH. The usage of this option is described as follows:

```
-alg 2 -qn -d -ds -ts -of
```

Under this option, the program answers the c-ANN queries by using the QALSH scheme. All the data objects are stored in disk. We output the c-ANN results of queries to the output folder (i.e., `-of`).

Option 3: Linear Scan. The usage of this option is described as follows:

```
-alg 3 -qn -d -ds -ts -of
```

Under this option, the program answers the exact NN queries by using brute force linear scan method. Both data objects and queries are stored in disk. We limit the available memory at most two page sizes (`2B`): one is used for the data objects, the other for the queries. Under this option, we can answer the exact NN queries from enormous datasets, at the expense of a large amount of wall-clock time.

Notice that the format of major Input/Output files will be described in Section 2.4. Since the program takes a large amount of time for indexing in Option 1, especially for the large datasets, the program will not reconstruct the indices of QALSH (i.e., the hash tables) if the indices have already been constructed.

Since Option 2 requires the hash tables of QALSH for the c-ANN search, please ensure that *Option 1 is performed in front of Option 2* when you run the package. Since Option 2 also requires the ground truth results to evaluate the accuracy of the results of QALSH, please ensure that *Option 0 is performed in front of Option 2* either. A running example will be given in Section 5.

2.3 Memory

There are two kinds of options which may cost a large amount of memory: Option 0 and Option 1. For example, all the data objects and the hash tables of QALSH are stored in the memory under Option 1, so that we can build the hash tables of QALSH in a quick manner. For the large datasets, QALSH may require a large amount of memory which is more than the available physical memory. Under this situation, QALSH will start to swap the pages in disk, and the performance may be decreased by two or three orders of magnitude. Therefore, we should limit the maximum available memory of QALSH.

In the package of QALSH, the maximum available memory is limited to be 1 GB. If the program requires much more available memory, it will ask the users whether or not it permits to get more available memory. The users can determine the permission based on the maximum available physical memory of the running machine. Users can also specify the default maximum available memory for the package, where the setting is defined in the header file `def.h`.

2.4 File Formats

2.4.1 Dataset and Query Set

In the package of QALSH, the input files are the dataset and query set. Both files are text files, where the format of the two files is described as follows:

$$\begin{array}{l}
1\ \mathtt{object}_{1,1}\ \mathtt{object}_{1,2}\ \ldots\ \mathtt{object}_{1,d} \\
2\ \mathtt{object}_{2,1}\ \mathtt{object}_{2,2}\ \ldots\ \mathtt{object}_{2,d} \\
\ldots \\
N\ \mathtt{object}_{N,1}\ \mathtt{object}_{N,2}\ \ldots\ \mathtt{object}_{N,d}
\end{array}$$

Notice that each $\mathtt{object}_{i,j}$ is a real number, where $i \in \{1, 2, ..., N\}$ and $j \in \{1, 2, ..., d\}$. For the dataset, `N` is the number of data objects, while for the query set, `N` stands for the number of queries. If the users are interested in other kinds of input format, i.e. binary file, you can modify the corresponding input functions in the files `util.h` and `util.cpp`.

2.4.2 Ground Truth File

The ground truth file is a text file with the following format:

$$\begin{array}{l}
\mathtt{qn}\ \mathtt{maxk} \\
\mathtt{dist}_{1,1}\ \mathtt{dist}_{1,2}\ \ldots\ \mathtt{dist}_{1,maxk} \\
\mathtt{dist}_{2,1}\ \mathtt{dist}_{2,2}\ \ldots\ \mathtt{dist}_{2,maxk} \\
\ldots \\
\mathtt{dist}_{qn,1}\ \mathtt{dist}_{qn,2}\ \ldots\ \mathtt{dist}_{qn,maxk}
\end{array}$$

Here `qn` is the number of queries in the query set. `maxk` is the value of top-k nearest distances of queries. For example, if `maxk = 100`, it means that we record the top-100 nearest distances of queries. $\mathtt{dist}_{i,j}$ represents the j^{th} nearest distance of the i^{th} query, where $i \in \{1, 2, \cdots, qn\}$ and $j \in \{1, 2, \cdots, maxk\}$. Notice that the ground truth file is an output file in Option 0, while it is an input file in Option 2.

2.4.3 File with the c-ANN Results

The file with the c-ANN results is an output file in Option 2. This file is also a text file, which consists of four columns: the first column is the top-k value, the second is the number of page I/Os, the third is the actual approximation ratio, and the forth column is the wall-clock time (ms) of QALSH. Here is an example of the c-ANN result of QALSH on the dataset Mnist[1], where `maxk = 100`.

```
  1 1293 1.020495 14.134780
 10 1642 1.012048 18.482380
```

[1]http://yann.lecun.com/exdb/mnist/

```
 20 1750 1.008802 19.708120
 30 1795 1.009858 20.228491
 40 1843 1.012149 20.816339
 50 1881 1.012314 21.273911
 60 1913 1.013563 21.638359
 70 1935 1.014951 21.903069
 80 1961 1.015623 22.293859
 90 1980 1.016903 22.446239
100 2003 1.016988 22.753740
```

2.4.4 File with the Parameters of QALSH

The file with the parameters of QALSH is an output file in Option 1, which is also a text file. It stores the parameters and the hash functions of QALSH for a specific dataset. Below is an example where the dataset is Mnist. We display the parameters and the first hash function of QALSH in this example.

```
  n = 60000
  d = 50
  B = 4096
  ratio = 2.000000
  w = 2.719112
  p1 = 0.826171
  p2 = 0.503496
  alpha = 0.738074
  beta = 0.001667
  delta = 0.367879
  m = 65
  l = 48
  -1.151670 -0.569114 -0.026606 0.537871 1.186576 1.316770 0.759021 -0.302207
0.443488 1.022791 -1.628174 -2.088325 -2.651624 -0.459455 -1.632034 -1.096159
0.359487 -0.156942 0.688126 -0.303920 -0.174608 -0.626074 0.875614 -0.120494
-0.812363 0.237206 -0.109623 1.664516 -0.669561 0.081490 1.882015 -0.959778
-0.035968 -1.737308 1.283968 0.029587 -0.393256 1.313041 -0.836016 0.898663
-0.254134 0.774990 0.050716 -0.659192 -0.345206 1.808912 -1.512095 -1.074031
-0.019611 -1.351239
  ...
```

Here, the parameter `w` is the bucket width w for an query-aware LSH function. `p1` and `p2` stand for the collision probability p_1 and p_2 of QALSH, respectively. `alpha` represents the parameter α of QALSH, which is the collision threshold in percentage. `beta` is the percentage of false positive β in QALSH. `delta` represents the error probability δ of QALSH. `m` and `l` stand for the number of hash tables m and the collision threshold l of QALSH, respectively. All these parameters will be defined and described in Section 3.2, and the setting of these parameters would be discussed in Subsection 3.2.4.

Chapter 3

Algorithm Description

In this chapter, we first introduce the Query-Aware LSH function family used in QALSH. Then, we describe the Query-Aware LSH (QALSH) scheme and simply discuss the parameter settings. Next, we show the space and query time complexity of QALSH for c-ANN search. Finally, we describe some important implementation details in the package of QALSH.

Since this is a user manual for engineering, we omit the proof here. If the users are interested in the proof of lemmas or propositions, please refer to the full paper [6].

3.1 Query-Aware LSH Family

In this section, we first review the basic definition of LSH family. Then, we introduce the concept of query-aware LSH functions, and show that query-aware LSH family is able to support *virtual rehashing* in a simple and quick manner.

3.1.1 Basic Definition of LSH Family

A family of LSH functions is able to partition "closer" objects into the same bucket with an accordingly higher probability. If two objects o and q are partitioned into the same bucket by a hash function h, we say o and q collide under h. Formally, an LSH function family (or simply an LSH family) in Euclidean space is defined as:

Definition 3 (LSH family) *Given a search radius r and approximation ratio c, an LSH function family $H = \{h : \mathcal{R}^d \to U\}$ is said to be (R, cR, p_1, p_2)-sensitive, if, for any $o, q \in \mathcal{R}^d$, we have*

- *if $\|o - q\| \leq R$, then $Pr_{h\in H}$[o and q collide under h] $\geq p_1$;*
- *if $\|o - q\| > cR$, then $Pr_{h\in H}$[o and q collide under h] $\leq p_2$.*

An LSH family is useful if $c > 1$ and $p_1 > p_2$. For ease of reference, p_1 and p_2 are called positively-colliding probability and negatively-colliding probability, respectively.

3.1.2 $(1, c, p_1, p_2)$-sensitive Query-Aware LSH Family

Constructing LSH functions in a *query-aware* manner consists of two steps: random projection and query-aware bucket partition. Formally, a query-aware hash function $h_{\vec{a}}(o) : \mathcal{R}^d \to \mathcal{R}$ maps a d-dimensional object $\vec{o}$ to a number along the real line identified by a random vector $\vec{a}$, whose entries are drawn independently from $\mathcal{N}(0, 1)$. For a fixed $\vec{a}$, the corresponding hash function $h_{\vec{a}}(o)$ is defined as follows:

$$h_{\vec{a}}(o) = \vec{a} \cdot \vec{o} \tag{3.1}$$

For all the data objects, their projections along the random line $\vec{a}$ are computed in the pre-processing step. When a query object q arrives, we obtain the query projection by computing $h_{\vec{a}}(q)$. Then, we use the query as the "anchor" to locate the *anchor bucket* with width w (defined by $h_{\vec{a}}(\cdot)$), i.e., the interval $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$. If the projection of an object o (i.e., $h_{\vec{a}}(o)$), falls in the anchor bucket with width w, i.e., $|h_{\vec{a}}(o) - h_{\vec{a}}(q)| \leq \frac{w}{2}$, we say o collides with q under $h_{\vec{a}}$.

We now show that the family of hash functions $h_{\vec{a}}(o)$ coupled with query-aware bucket partition is locality-sensitive. In this sense, each $h_{\vec{a}}(o)$ in the family is said to be a query-aware LSH function. For objects o and q, let $s = \|o - q\|$. Due to the stability of standard normal distribution $\mathcal{N}(0,1)$, we have that $(\vec{a} \cdot \vec{o} - \vec{a} \cdot \vec{q})$ is distributed as sX, where X is a random variable drawn from $\mathcal{N}(0,1)$ [2]. Let $\varphi(x)$ be the probability density function (PDF) of $\mathcal{N}(0,1)$, i.e., $\varphi(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$. The collision probability between o and q under $h_{\vec{a}}$ is computed as follows:

$$\begin{aligned} p(s) &= Pr_{\vec{a}}[|h_{\vec{a}}(o) - h_{\vec{a}}(q)| \leq \tfrac{w}{2}] \\ &= Pr[|sX| \leq \tfrac{w}{2}] \\ &= Pr[-\tfrac{w}{2s} \leq X \leq \tfrac{w}{2s}] \\ &= \int_{-\frac{w}{2s}}^{\frac{w}{2s}} \varphi(x)\, dx \end{aligned} \tag{3.2}$$

Accordingly, we have Lemma 1 as follows:

Lemma 1 *The query-aware hash family of all the hash functions $h_{\vec{a}}(o)$ that are identified by Equation 3.1 and coupled with query-aware bucket partition is $(1, c, p_1, p_2)$-sensitive, where $p_1 = p(1)$ and $p_2 = p(c)$.*

3.1.3 Virtual Rehashing

LSH schemes such as C2LSH do not solve the c-ANN search problem directly. Because an (R, cR, p_1, p_2)-sensitive LSH family requires R to be pre-specified so as to compute p_1 and p_2. The c-ANN search can be reduced to a series of (R, c)-NN search with properly increasing search radius $R \in \{1, c, c^2, c^3, ...\}$. Therefore, for each R, we need an (R, cR, p_1, p_2)-sensitive LSH family. For each $R \in \{c, c^2, ...\}$, by deriving an (R, cR, p_1, p_2)-sensitive hash family from the $(1, c, p_1, p_2)$-sensitive hash family for $R = 1$, hash tables for all the subsequent radii can be virtually imposed on the physical hash tables for $R = 1$. This is the underlying idea of *virtual rehashing* of C2LSH.

We now show that QALSH can do virtual rehashing by deriving (R, cR, p_1, p_2)-sensitive functions from Equation 3.1. Virtual rehashing of QALSH enables it to work with any $c > 1$, while both C2LSH and LSB-Forest only work with integer $c \geq 2$.

Proposition 1 *The* query-aware *hash family*

$$H_{\vec{a}}^R(o) = \frac{h_{\vec{a}}(o)}{R}$$

is (R, cR, p_1, p_2)-sensitive, where c, p_1, p_2 and $h_{\vec{a}}(\cdot)$ are the same as defined in Lemma 1, and R is a power of c (i.e., c^k for some integer $k \geq 1$).

Given a query q and a pre-specified bucket width w, for each $R \in \{1, c, c^2, ...\}$, we now define the round-R anchor bucket B^R as the anchor bucket with width w defined by $H_{\vec{a}}^R(\cdot)$, i.e., the interval $[H_{\vec{a}}^R(q) - \frac{w}{2}, H_{\vec{a}}^R(q) + \frac{w}{2}]$, which is centered at q's projection $H_{\vec{a}}^R(q)$ along the random line $\vec{a}$. In other words, the round-R anchor bucket can be located by query-aware bucket partition with bucket width w as before. Specifically, B^1 is simply the interval $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$, which is the anchor bucket with width w defined by $h_{\vec{a}}(\cdot)$, also known as $H_{\vec{a}}^1(\cdot)$.

To find the (R, c)-NN of a query q, we only require to check the round-R anchor bucket B^R for the specific R. To find the c-ANN of q, we check the round-R anchor buckets round by round for gradually increasing R ($R \in \{1, c, c^2, ...\}$). All the round-R anchor buckets defined by $H_{\vec{a}}^R(\cdot)$ are centered at the projection of q along the same $\vec{a}$, and can be located along $\vec{a}$ with properly adjusted bucket width. Therefore,

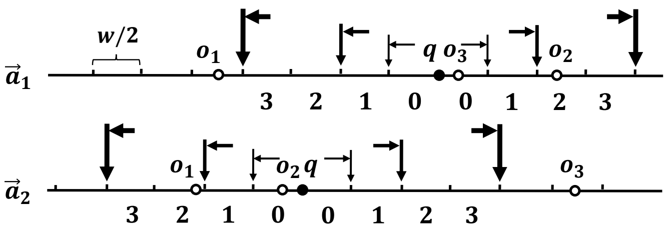

Figure 3.1: Virtual Rehashing of QALSH for $c = 2$

we only need to keep one physical copy of the data projections along $\vec{a}$. Using the results of the following Propositions 2 and 3, we can virtually impose B^R over B^1, and hence B^{cR} over B^R. This is the underlying idea of virtual rehashing of QALSH.

Proposition 2 *Given a query q and a bucket width w, B^R contains B^1, and the width of B^R is R times the width of B^1, i.e., wR.*

Proposition 3 *Given a query q and a bucket width w, for any radius R, B^{cR} contains B^R, and the width of B^{cR} is c times the width of B^R.*

Referring to Figure 3.1, on the random line $\vec{a_1}$, the interval of width w centered at q is B^1, which is indicated by "00". B^2 and B^4 are indicated by "1001" and "32100123", respectively. Virtual rehashing of QALSH is equal to symmetrically searching half-buckets of length $\frac{w}{2}$ one by one on both sides of q.

3.2 QALSH Scheme

3.2.1 QALSH for (R, c)-NN Search

QALSH directly solves the (R, c)-NN problem by exploiting a base $\mathcal{B}$ of m query-aware LSH functions $\{H^R_{\vec{a}_1}(\cdot), H^R_{\vec{a}_2}(\cdot), \ldots, H^R_{\vec{a}_m}(\cdot)\}$. Those LSH functions are mutually independent, and are uniformly selected from an (R, cR, p_1, p_2)-sensitive query-aware LSH family. For each $H^R_{\vec{a}_i}(\cdot)$, we build a hash table T_i which is indexed by a B^+-tree, as will be described in Subsection 3.4.2.

To find the (R, c)-NN of a query q, we first compute the hash values $H^R_{\vec{a}_i}(q)$ for $i \in \{1, 2, \ldots, m\}$, and then use the B^+-trees over T_i's to locate the m round-R anchor buckets B^R. For each object o that appears in the m anchor buckets, we collect its *collision number* $\#Col(o)$, which is formally defined as follows:

$$\#Col(o) = \left|\left\{H^R_{\vec{a}_i} \mid H^R_{\vec{a}_i} \in \mathcal{B} \wedge |H^R_{\vec{a}_i}(o) - H^R_{\vec{a}_i}(q)| \le \frac{w}{2}\right\}\right|. \tag{3.3}$$

Given a pre-specified collision threshold l, object o is called *frequent* (with respect to q, w and $\mathcal{B}$) if $\#Col(o) \ge l$. We prefer to collecting collision numbers first for objects whose projections are closer to the query projection. We only need to find the "first" βn frequent objects (where β is clarified later and n is the cardinality of D) and compute the Euclidean distances to q for them. If there is some frequent object whose distance to q is less than or equal to cR, we return YES and the object; Otherwise, we return NO.

The base cardinality m is one of the key parameters for QALSH, which need to be properly chosen so as to ensure that the following two properties hold at the same time with a constant probability:

- $\mathcal{P}_1$: If there exists an object o whose Euclidean distance to q is within R, then o is a frequent object.
- $\mathcal{P}_2$: The total number of *false positives* is less than βn, where each false positive is a frequent object whose Euclidean distance to q is larger than cR.

The above assertion is assured by Lemma 2 as follows, which guarantees correctness of QALSH for the (R, c)-NN search. Let l be the collision threshold, α be the collision threshold in percentage, we have $l = \alpha m$. Let δ be the error probability, β be the percentage of false positives.

Lemma 2 *Given $p_1 = p(1)$ and $p_2 = p(c)$, where $p(\cdot)$ is defined by Equation 3.2. Let α, β and δ be defined as above. For $p_2 < \alpha < p_1$, $0 < \beta < 1$ and $0 < \delta < \frac{1}{2}$, $\mathcal{P}_1$ and $\mathcal{P}_2$ hold at the same time with probability at least $\frac{1}{2} - \delta$, provided the base cardinality m is given as below:*

$$m = \left\lceil \max\left(\frac{1}{2(p_1-\alpha)^2} \ln\frac{1}{\delta}, \frac{1}{2(\alpha-p_2)^2} \ln\frac{2}{\beta} \right) \right\rceil . \tag{3.4}$$

3.2.2 QALSH for c-ANN Search

Given a query q and a pre-specified bucket width w, in order to find the c-ANN of q, QALSH first collects frequent objects from round-1 anchor buckets using $R = 1$; if frequent objects collected so far are not enough, QALSH *automatically* updates R, and hence collects more frequent objects from the round-R anchor buckets via virtual rehashing, and etc., until finally enough frequent objects have been found or a *good enough* frequent object has been identified. The c-ANN of q must be one of the frequent objects.

QALSH is quite straightforward, as shown in Algorithm 1. A candidate set C is used to store the frequent objects found so far, and is empty at the beginning.

Algorithm 1 QALSH

Input:

c is the approximation ratio, β is the percentage of false positives, δ is the error probability. m is the number of hash tables, l is the collision threshold.

Output:

the nearest object o_{min} in the set C of frequent objects.

```
1:  $R = 1; C = \emptyset$;
2:  while $|C| < \beta n$ do
3:      for each $i = 1$ to $m$ do
4:          increase $\#Col(o)$ by 1 if $o$ is found in the round-$R$ anchor bucket, i.e., $|H^R_{\vec{a}_i}(o) - H^R_{\vec{a}_i}(q)| \le \frac{w}{2}$;
5:          if $\#Col(o) \ge l$ then
6:              $C = C \cup o$;
7:          end if
8:      end for
9:      if $|\{o \mid o \in C \wedge \|o, q\| \le c \times R\}| \ge 1$ then
10:         break;
11:     end if
12:     update radius $R$;
13: end while
14: return the nearest object $o_{min} \in C$;
```

Terminating Conditions. QALSH terminates in one of the two following cases which are supported by the two properties $\mathcal{P}_1$ and $\mathcal{P}_2$ of Lemma 2, respectively:

- $\mathcal{T}_1$: At round-R, there exists at least 1 frequent object whose Euclidean distance to q is less than or equal to cR (referring to Lines 9 – 11 in Algorithm 1).
- $\mathcal{T}_2$: At round-R, at least βn frequent objects have been found (referring to Line 2 and Line 13 in Algorithm 1).

Update of Search Radius R. It can be checked that, in Algorithm 1, if the terminating condition $\mathcal{T}_1$ is still not satisfied at the moment, i.e., we have not found a good enough frequent object, then we need to update R in Line 12. For ease of reference, let R and R' denote current and next search radius, respectively.

QALSH automatically update the search radius R by leveraging the projections of data objects. Recall that each of the m hash tables of QALSH is simply a B^+-tree, we can easily find the object o which is closest to q and exists outside of the current round-R anchor bucket. Thus we have m such objects in total. Suppose their distances to q (in terms of projections) are sorted in ascending order and denoted as d_1, d_2, . . ., and d_m, i.e., d_1 is the smallest and d_m is the biggest. Let d_{med} denote the median of $\{d_1, d_2, \ldots, d_m\}$. QALSH automatically set R' to be $R' = c^k$ such that $\frac{wR'}{2} \geq d_{med}$, where integer $k = \lceil \log_c(2d_{med}/w) \rceil$. Therefore, there are at least $\frac{m}{2}$ objects for collecting collision number in the next round of search.

The underlying intuition is as follows. If we set R' according to d_1, the round-R' anchor buckets may contain too few data objects for collecting collision number and hence we waste the scan of the round-R' anchor buckets. On the other hand, if we set R' according to d_m, since d_m may be too large, round-R' anchor buckets may contain too many data objects, and hence we may do unnecessary collision number collection and Euclidean distance computation. In addition, R' has to be $R' = c^k$ for integer k, so that the theoretical framework of QALSH is still assured.

3.2.3 QALSH for c-k-ANN Search

To support the c-k-ANN search, QALSH only needs to change its terminating conditions of c-ANN:

- $\mathcal{T}_1'$: At round-R, there exist at least k frequent objects whose Euclidean distance to q are within cR.
- $\mathcal{T}_2'$: At round-R, there are at least $(\beta n + k - 1)$ frequent objects that have been found.

3.2.4 Parameter Settings

The accuracy of QALSH is controlled by error probability δ, approximation ratio c and false positive percentage β. δ controls the success rate of the LSH-based methods for c-ANN search. A smaller c means a higher accuracy. Intuitively, a bigger β allows QALSH to check more frequent objects, and hence achieves a better search accuracy, with higher costs in terms of random I/Os. In the package of QALSH, we set $\delta = \frac{1}{e}$ to make a fair comparison to LSB-Forest and C2LSH with the same success probability. Similar to C2LSH, we set $\beta = 100/n$ to restrict the number of random I/Os. We let c be an input parameter of QALSH specified by users.

Then, we consider the base cardinality m, collision threshold percentage α and collision threshold l. m is computed by the following expression:

$$m = \left\lceil \frac{\left(\sqrt{\ln \frac{2}{\beta}} + \sqrt{\ln \frac{1}{\delta}}\right)^2}{2(p_1 - p_2)^2} \right\rceil. \tag{3.5}$$

Then, α could be determined by the following expression:

$$\alpha = \frac{\eta \cdot p_1 + p_2}{1 + \eta}, \tag{3.6}$$

where $\eta = \sqrt{\frac{\ln \frac{2}{\beta}}{\ln \frac{1}{\delta}}}$. After setting the values of m and α, we compute the integer collision threshold l as follows:

$$l = \lceil \alpha m \rceil. \tag{3.7}$$

Finally, we consider the setting of bucket width w and the collision probability p_1 and p_2. As we know, the base cardinality m is simply the number of hash tables in QALSH. A small m leads to small time and space overhead in QALSH. It follows from Equation 3.5 that m decreases monotonically with the difference $(p_1 - p_2)$ for fixed δ and β. In order to maximize $(p_1 - p_2)$ so that to minimize the value of m, for any approximation ratio $c > 1$, w is automatically decided by the following equation:

$$w = \sqrt{\frac{8c^2 \ln c}{c^2 - 1}} \tag{3.8}$$

After setting the value of w, p_1 and p_2 can then be computed by Equation 3.2, where $p_1 = p(1)$ and $p_2 = p(c)$.

3.3 Space and Query Time Complexity

Now, we analyse the space and query time complexity of QALSH. Since QALSH is designed for external memory, we consider the disk overhead (i.e., page I/Os) for the space and query time complexity.

Since we set $\beta = 100/n$, βn is constant. From Equation 3.5 and 3.7, we have $m = O(\log n)$ and $l = O(\log n)$. Since our implementation of each B^+-tree is similar to a string B-tree introduced in [3], the height E of the B^+-tree can be bound by $O(\log_B(n/B))$.

3.3.1 Space Overhead

The space overhead of QALSH consists of three parts:

- The dataset costs $O(\frac{nd}{B})$ page I/Os;
- The index of QALSH costs $mn/B = O(\frac{n \log n}{B})$ page I/Os, where m is the number of hash tables which store n data objects' id and projection;
- The hash functions cost $md/B = O(\frac{d \log n}{B})$ page I/Os.

Therefore, the total number of page I/Os of QALSH is $O(\frac{nd}{B} + \frac{n \log n}{B} + \frac{d \log n}{B}) = O(\frac{nd + n \log n}{B})$.

3.3.2 Query Time complexity

The query time complexity of QALSH also consists of three parts:

- Locating the m round-1 anchor buckets in B-tree costs $mE = O(\log n \cdot \log_B(n/B))$ page I/Os;
- In the worst case, finding the frequent objects as candidates needs to do collision counting for all the n objects over each hash table, so that we need to scan at most nl object id's. Therefore, the complexity of this part is $nl/B = \frac{n \log n}{B}$;
- We calculate the Euclidean distance for the limited $\beta n = 100$ candidates, then this part costs 100 page I/Os.

Therefore, the query time complexity of QALSH is $O(\log n \cdot \log_B(n/B) + \frac{n \log n}{B} + 100) = O(\frac{n \log n}{B})$ page I/Os.

3.4 Implementation Details

3.4.1 A New Organization of Dataset

When answering a c-ANN query, QALSH requires to compute the Euclidean distance for a set of candidate objects. Since the candidates are randomly appeared in different queries, we require to compute Euclidean distance for a set of random objects for each query. Therefore, computing Euclidean distance for the candidate objects may lead to a large amount of random page I/Os, especially when answering a large number of queries at the same time. In this package, we adopt two strategies to alleviate this problem.

First, we limit the number of false positives βn to be a constant number (i.e., 100, as described in Section 3.2.4), so that we only need to check at most a constant number of candidates. Therefore, the number of random page I/Os for each query is limited.

Second, we divide the dataset into a set of small files. This is because the main overhead of the program for candidate computation is the step to load the candidate objects from the large dataset, which generates a large number of random page I/Os. In order to reduce the number of random I/Os, we divide the large dataset into a set of small files. When you require to read a candidate object, you can simply load its corresponding small file with one disk I/O, but without any random access. Therefore, the wall-clock time could be largely reduced, even though the program still use the same number of page I/Os.

Each small file is a binary file with a fixed size `B`. Let k be the number of data objects stored in each small file, then k can be computed as follows:

$$k = \left\lfloor \frac{dn \cdot SIZE}{\mathtt{B}} \right\rfloor, \tag{3.9}$$

where $SIZE$ is the size of a real number. In this package, we use the data type `float` to represent a real number, where $SIZE = 4$ Bytes. Let n_s be the number of small data files , n_s is computed as follows:

$$n_s = \left\lceil \frac{n}{k} \right\rceil. \tag{3.10}$$

The organization of the small files is described as follows: We store the first k objects (object id between 1 and k) in the 1^{st} small file; Then, we store the second k objects (object id between $k+1$ and $2k$) in the 2^{nd} small file; We keep storing the objects in the same manner; Finally, we store the last (at most) k objects in the n_s^{th} small file.

We should note that this strategy may generate a large number of small files in disk. But since the physical memory is quite large in current machines, the users can reduce the number of small files by setting a relatively large value of page size `B`. In the future, we will also consider to cache a number of data objects in the memory to reduce the number of page I/Os.

3.4.2 Data Structure of Hash Tables

In QALSH, the hash table can be viewed as a list T of pairs $(h(o), ID_o)$ for each data object o, where ID_o is the object id referring to o and $h(o)$ is the hash value of o. In order to quickly lookup a series of anchor buckets to find the candidate objects, the list is sorted in ascending order of $h(o)$, and is then indexed by a B^+-tree. Here, we introduce the structure of B^+-tree of QALSH.

At the beginning, we simply store the pair $(h(o), ID_o)$ for each data object o in the leaf node. Then, we use the hash value $h(o)$ of the first object o as the key of the leaf node. For the index node, we use the key and the address of a leaf node as one entry of index node. It is easy to see that the structure of index node is optimal. Without the key or the address, we can't search or update on the B^+-tree any more.

Now, we focus on the structure of leaf node. As described in Section 3.3.2, the major query time of QALSH happens when we scan the B^+-tree to find the candidates. Strictly speaking, it happens when counting the frequencies of the objects by scanning the object ids in the leaf nodes of B^+-tree. However, this structure of leaf node spends a half of space to store the hash value $h(o)$, which is useless for the search.

Then, we consider to modify the structure of leaf node. At first, in order to store large enough object ids (i.e., ID_o) at the leaf node, we only store the ID_o in the leaf node. When constructing the B^+-tree, we continue to use the hash value $h(o)$ of the first object o as the key of the leaf node, even though we do not store $h(o)$ in the leaf node. By using this structure, the time efficiency is significantly improved. For example, the number of page I/Os of QALSH are reduced about 50%.

However, by setting a large page size `B` of the dataset P53[1] during the experiments, i.e. `B = 65,536` bsytes, we found that the accuracy of QALSH is also largely reduced. The main reason is that too much object ids are stored in one leaf node. For example, for the dataset P53, the number of object ids in a leaf node is about $16,800$, while the cardinality `n` of P53 is only $31,159$. Therefore, the search keys in the index node are not enough to separate the anchor bucket in a fine granularity.

In order to solve the above problem, we propose a new data structure for the leaf node. In the leaf node, we not only store the object ids, but also store some hash values. However, instead of storing hash value for each object, we only store one hash value for every $1,024$ objects, where the hash value is corresponding to the first of $1,024$ objects. In this data structure, the leaf node consists of two parts: a large number of object ids and a small number of hash values. We use the first hash value as the key of the leaf node. Using the small number of hash values, we are able to locate the anchor bucket in a fine granularity. Therefore, the accuracy of QALSH could be guaranteed and won't be restricted by the size of page `B`. In addition, since the leaf node still contains a large number of object ids, the time efficiency is still improved significantly.

We should note that the value of $1,024$ is an empirical value. Our experimental results show that, by storing one hash value for every $1,024$ objects, both accuracy and efficiency of QALSH perform very well. In the package of QALSH, we define this value as a constant `INDEX_SIZE_LEAF_NODE` in the header file `def.h`. Users can modify this value in the actual application.

[1]`http://archive.ics.uci.edu/ml/datasets/p53+Mutants`

Chapter 4

The QALSH Code

In this chapter, we first give an overview of the code of QALSH. Then, we introduce the data structures and several important interfaces of QALSH.

4.1 Code Overview

The core of the QALSH code consists of three main components:

- `ann.cpp, ann.h –` contain the brute-force linear scan method and the interfaces of indexing and c-ANN search of QALSH. 4.3.
- `qalsh.cpp, qalsh.h –` contain the implementation of the data structure of QALSH. The main functionality is for initializing the parameters of QALSH, indexing the hash tables and answering the c-ANN of queries.
- `b_tree.cpp, b_tree.h –` contain the implementations of the data structure of B^+tree, which is used to store the hash tables in disk.

The remaining files in the package of QALSH are described as follows:

- `b_node.cpp, b_node.h –` contain the implementations of the data structures of leaf node and index node of B^+tree, which is used to provide some assistant functions to construct a B^+tree.
- `block_file.cpp, block_file.h –` contain the implementations of a data structure to read/write files from/to disk. This data structure also provides some assistant functions to construct a B^+tree.
- `random.cpp, random.h –` contain some mathematical functions to generate random variables from some specific distributions (i.e., Gaussian distribution) and to compute the collision probability.
- `util.cpp, util.h –` contain some general purpose functions, such as calculating Euclidean distance, input/output operations for the datasets and query sets.
- `def.h –` contains the definitions of general purpose data types, macros and constants.
- `main.cpp –` contains the function to describe the usage of QALSH, and the `main()` function which reads the input parameters and calls the corresponding functions for c-ANN search.

4.2 Data Structures

There are four important data structures in the package of QALSH.

- `QALSH` – the fundamental data structure for the c-ANN search. This data structure contains the parameters used to construct the hash tables, the hash functions for the dataset, and the pointers to the B^+tree. This data structure is defined in `qalsh.h` and implemented in `qalsh.cpp`.
- `BTree` – the basic data structure to index the hash tables of QALSH. The detail descriptions of `BTree` is given in Section 3.4.2. This data structure is defined in `b_tree.h` and implemented in `b_tree.cpp`.
- `BNode` – the basic structure of node of `BTree`. This structure mainly contains the number of entries, the left sibling address and right sibling address, an array of entries and the level (depth in B^+-tree). Based on this basic structure, we also derive the structures of `BIndexNode` and `BLeafNode` for our specific index node and leaf node, respectively. All these three data structures are defined in `b_node.h` and implemented in `b_node.cpp`.
- `BlockFile` – the basic structure of reading and writing files for `BTree`. This structure mainly contains a file name and its corresponding file pointer for input/output operations, the length of one block (a.k.a. page), and the number of blocks. The data structure is defined in `block_file.h` and implemented in `block_file.cpp`.

The package of QALSH also contain the following data structures.

- `ResultItem` – the structure which is used to store the c-ANN results of the queries. The structure contains an object id and the Euclidean distance between the object and the query. This structure is defined in `qalsh.h`.
- `HashValue` – the structure of hash table in memory to store the hash value. This structure contains an object id and the hash value of the object. This structure is defined in `qalsh.h`.
- `PageBuffer` – the structure which is used to store the buffer of a page when answering the c-ANN search. This structure contain a `BLeafNode` pointer, the position of this `BLeafNode` in its corresponding index node, the current position of the entries in this `BLeafNode` and the size of entries for one scan. This structure is also defined in `qalsh.h`.

4.3 QALSH Interface

Now, we introduce three important functions at the interface of the package of QALSH. The first two functions are sufficient for answering a c-ANN query. The last two function answers the nearest neighbor search by using brute-force linear scan method. All the functions are declared in the file `ann.h` and implemented in `ann.cpp`.

1. To construct the hash tables of QALSH, user can use the function:

```
int indexing(
    int   n,
    int   d,
    int   B,
    float ratio,
    char* data_set,
    char* output_folder);
```

The input parameters of this function have the following meaning:

- `n` – the cardinality of dataset;
- `d` – the dimensionality of dataset;
- `B` – the page size;
- `ratio` – the approximation ratio of the c-ANN results;
- `data_set` – the file of data objects;
- `output_folder` – the output folder which is used to store the hash tables;

The function `indexing()` will first call the function `init()` to calculate the parameters of QALSH and to generate the hash function. The function `init()` belongs to class `QALSH` in the file `qalsh.h`. Then, the function `indexing()` will call the function `bulkload()`. The function `bulkload()` will compute the hash values of data objects, sort the hash values in ascending order, index the hash tables by B^+-tree and then write them to disk (i.e., to the folder `output_folder/indices/`). The function `bulkload()` also belongs to the class `QALSH`.

In addition, the function `indexing()` also reorganises the format of dataset, as described in Subsection 3.4.1. The new format of dataset are stored in the folder `output_folder/data/`.

2. For the query operation, one can use the following function to answer the c-ANN results.

```
int lshknn(
    int   d,
    int   qn,
    char* query_set,
    char* truth_set,
    char* output_folder);
```

The input parameters of this function have the following meaning:

- `d` – the dimensionality of dataset;
- `n` – the cardinality of dataset;
- `query_set` – the file of query objects;
- `truth_set` – the file of the ground truth results;
- `output_folder` – the output folder which is used to store the c-ANN results;

In order to answer the c-ANN queries, the function `lshknn()` will first call the function `knn()` which belongs to the class `QALSH`. Then the function `lshknn()` will evaluate the c-ANN results of the queries. Finally, the results will be stored in the file `output_folder/qalsh.out`.

3. If one want to generate the ground truth of the queries, you can call the following function.

```
int ground_truth(
    int   n,
    int   qn,
    int   d,
    char* data_set,
    char* query_set,
    char* truth_set);
```

The input parameters of this function have the following meaning:

- `n` – the cardinality of dataset;
- `qn` – the number of query objects;
- `d` – the dimensionality of dataset;
- `data_set` – the file of data objects
- `query_set` – the file of query objects;
- `truth_set` – the file of the ground truth results;

The function `ground_truth()` will generate the ground truth by using the brute-force linear scan method, where the data objects are stored in memory. Notice that if the dataset is larger than maximum memory we set, the program will stop and ask the user for more memory.

4. The following function provide a base line to answer the nearest neighbor queries.

```
int linear_scan(
    int   n,
    int   qn,
    int   d,
    int   B,
    char* query_set,
    char* truth_set,
    char* output_folder);
```

The input parameters of this function have the following meaning:

- `n` – the cardinality of dataset;
- `qn` – the number of query objects;
- `d` – the dimensionality of dataset;
- `B` – the page size;
- `query_set` – the file of query objects;
- `truth_set` – the file of the ground truth results;
- `output_folder` – the output folder which is used to read the new format of dataset;

The function `linear_scan()` also answer the NN queries by using the brute-force linear scan method. However, the data objects are stored in disk. In this function, we limit the memory at most `2B` page size. Therefore, this function can find the exact NN for any large datasets, at the expense of longer wall-clock time.

Chapter 5

Frequent Anticipated Questions

In this chapter, we answer some questions that the users may ask when compiling and using the package.

1. How to compile the package?

   Answer: Since we have already written the `Makefile`, you can simply type **`make`** in the main directory of this package to compile the code.

2. The package has already been compiled, but I do not know how to run it?

   Answer: Before you run the package, please ensure that the format of dataset and query set are correct, where the format of these two files is described in Section 2.4.

   Now, we give an example to show how to run the package. Suppose that you have a dataset `Mnist.ds` and a query set `Mnist.q` which reside in the directory `./data/Mnist/`, and their formats are correct. The cardinality of the dataset `Mnist.ds` is `n = 60000` and the dimensionality is `d = 50`. For the query set `Mnist.q`, the number of queries is `qn = 100`. Suppose the page size is `B = 4096` and the approximation ratio is set to be `c = 2.0`.

   a. If you want to generate the ground truth of the queries in the query set, you can type the following command under the main directory:

   ```
   ./qalsh -alg 0 -n 60000 -qn 100 -d 50 -ds ./data/Mnist/Mnist.ds -qs
   ./data/Mnist/Mnist.q -ts ./data/Mnist/Mnist.gt2
   ```

   The program will generate the ground truth results, and the ground truth file `Mnist.gt2` will be written to the directory `./data/Mnist/`.

   b. If you want to build the index of QALSH, you can type the following command under the main directory:

   ```
   ./qalsh -alg 1 -n 60000 -d 50 -B 4096 -c 2.0 -ds ./data/Mnist/Mnist.ds
   -of ./results/Mnist/
   ```

   The program will build the hash tables of QALSH in the directory `./results/Mnist/indices/`. In addition, the program will generate a new organization of the dataset (i.e., a number of small files) in the directory `./results/Mnist/data/`. The indexing time will also be written to the file `./results/Mnist/L2_index.out`.

   c. If you want to answer the c-ANN queries by QALSH, you can type the following command under the main directory:

```
./qalsh -alg 2 -qn 100 -d 50 -qs ./data/Mnist/Mnist.q
-ts ./data/Mnist/Mnist.gt2 -of ./results/Mnist/
```

The program will answer the c-ANN queries by QALSH. The results of QALSH will be written to the file `./results/Mnist/L2_qalsh.out`.

d. If you want to answer the exact NN queries by linear scan, you can type the following command under the main directory:

```
./qalsh -alg 3 -n 60000 -qn 100 -d 50 -B 4096 -qs ./data/Mnist/Mnist.q
-ts ./data/Mnist/Mnist.gt2 -of ./results/Mnist/
```

The program will answer the exact NN queries by the brute-force linear scan method. The results will be written to the file `./results/Mnist/L2_linear.out`.

3. When I run the package, why does the program display the message "QALSH: hash tables exist." on the console and stop?

   Answer: If the console display this message, probably because the hash tables have already been constructed when you ask the program to build the hash tables. As discussed in Section 2.2, the hash tables of QALSH would be constructed only once a time. Therefore, the program told you that "hash tables exist". You can remove the hash tables, then the program will construct new hash tables for you.

Bibliography

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.

[2] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[3] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM (JACM)*, 46(2):236–280, 1999.

[4] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.

[5] S. Har-Peled. A replacement for voronoi diagrams of near linear size. In *FOCS*, pages 94–103, 2001.

[6] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1), 2015.

[7] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *ACM STOC*, pages 604–613, 1998.

[8] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *ACM-SIAM SODA*, pages 1186–1195, 2006.

[9] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM TODS*, 35(3):20, 2010.