

Reproduce of I-LSH: I/O efficient c-Approximate Nearest Neighbor Search in High-dimensional Space

肖江枫

摘要

最近邻 (Nearest Neighbor) 查询的问题在低维空间得到了很好的解决, 但由于维度灾难其在高维空间具有挑战性。作为效率和结果准确性之间的权衡, 各种 c -近似最近邻 (c -ANN: c -Approximate NN) 算法已经被提出, 以返回一个可信度至少为 δ 的 c -近似最近邻。原作者团队观察到, 现有的 c -ANN 搜索算法在其索引驻留在外部存储器上时, 对 I/O 效率有一些限制, 这对处理大规模的高维数据至关重要。

在原文中, 原作者介绍了一种基于增量搜索的 c -ANN 查询算法, 名为 I-LSH。之前的 LSH 方法指数扩展桶的宽度, 与其不同的是 I-LSH 采用了一种更自然的搜索策略来按增量访问对象的哈希值。原文提供了严格的理论分析来支持他们的增量搜索策略, 且他们的综合实验结果表明, 与当时最先进的 I/O 高效 c -ANN 技术相比, 在相同的理论保证下, I-LSH 可以实现更好的 I/O 效率。

关键词: c -ANN 查询算法; 局部敏感哈希; 增量搜索

1 引言

给定一组 d 维的对象和一个查询对象, 最近邻查询要找到与查询对象距离最小的对象。由于“维度灾难”的问题, 若要找到精确的距离最小的对象, 可能需要花费巨大的开销, 因此 c -ANN 的查询问题被提出, 并被应用于许多领域, 如数据库、计算机视觉、多媒体、机器学习和推荐系统。

局部敏感哈希 (LSH: Locality-Sensitive Hashing) 是一种广泛采用的支持 c -ANN 搜索的方法。除了理论上的保证外, 由于其出色的性能和易于实现, 它在实践中也获得了巨大的成功。大多数 LSH 算法是为外部存储器设计的, 如 SRS 和 QALSH, 但它们都没有很好的 I/O 效率。基于这个问题, 原作者旨在开发一种 I/O 高效的 c -ANN 查询新算法, 也即是原文的 I-LSH。

I-LSH 与其他 LSH 算法的不同之处在于它适用于高维数据, 并在投影维度上使用自然的增量搜索策略。原作者提供了严格的分析来证明 I-LSH 的正确性和效率。他们还对两个当时最先进的 I/O 高效 c -ANN 算法进行了关于 I/O 开销和结果准确性的广泛性能评估, 结果表明在相同的理论保证下, I-LSH 可以实现最佳的 I/O 性能表现。

2 相关工作

2.1 c -ANN 的定义

用 \mathcal{R}^d 表示在 d 维空间中给定的一个含有 n 个数据对象的数据集 D , 每个对象 o 在第 i 维上的坐标值被表示为 $o[i]$ 。则原文中给出 c -ANN 的定义如下: 对于一个给定的查询对象 q 和一个 d 维数据集 D , 假设 o^* 是 q 的最近邻, 距离为 R^* , 则 q 的 c -ANN 是一个数据对象 $o \in D$, 使得 $\|o, q\| \leq cR^*$, 其中 c 是近似比率。

2.2 局部敏感哈希

局部敏感哈希由 Indyk 等人在 1998 年首次提出^[1]，并由 Datar 等人扩展到欧几里德空间^[2]。LSH 方法将原始数据空间中的两个相邻数据点，通过相同的映射或投影变换后，这两个数据点在新的数据空间中仍然相邻的概率很大，而不相邻的数据点被映射到同一个桶的概率很小。LSH 函数可以保持数据对象之间的距离关系，这意味着对于两个点 o_1 和 o_2 ，如果 $\|o_1, o_2\|$ 是一个小值，那么 $\|h(o_1), h(o_2)\|$ 也可能是一个小值。

2.3 当时已有的 LSH 方法

2.3.1 QALSH

QALSH^[3]首先提出了一个查询感知的 LSH 函数，它可以正式表示为 $H_{\vec{a}}(o) = \vec{a} \cdot \vec{o}$ 。在我们的算法中，我们使用相同的哈希函数来推算 d 维数据集。该方法的实现是基于一组 B 树的，具有一定 I/O 效率。但 QALSH 是一个 c^2 -近似算法，这意味着它需要大量的哈希函数。

2.3.2 SRS

SRS^[4]使用一个多维索引 R+ 树来解决 QALSH 存在的这个问题。它能以极小的索引尺寸保持理论上的保证。然而，由于 R+ 树存储结构的特性，当 $m > 5$ 时，它的效果并不好。

2.3.3 其他 LSH 方法

原文中还列举了一些 I/O 高效但没有理论保证的 ANN 查询算法。例如，刘英帆等人提出针对 ANN 问题的 SK-LSH^[5]，作为 LSB-tree 的改进，它使用线性顺序而不是 Z-顺序来提高 I/O 效率。在刘英帆的另一篇论文^[6]中，他们提出了基于 PQ 方法^[7]的 I/O 高效算法。

2.4 指数扩展桶宽度

QALSH 和 C2LSH^[8]均采用了桶的指数扩展桶宽度的策略，桶的宽度必须是 c 的幂（即桶的宽度以指数方式增长），这可能会导致一些反直觉的情况，如图 1(a) 和 (b) 所示。

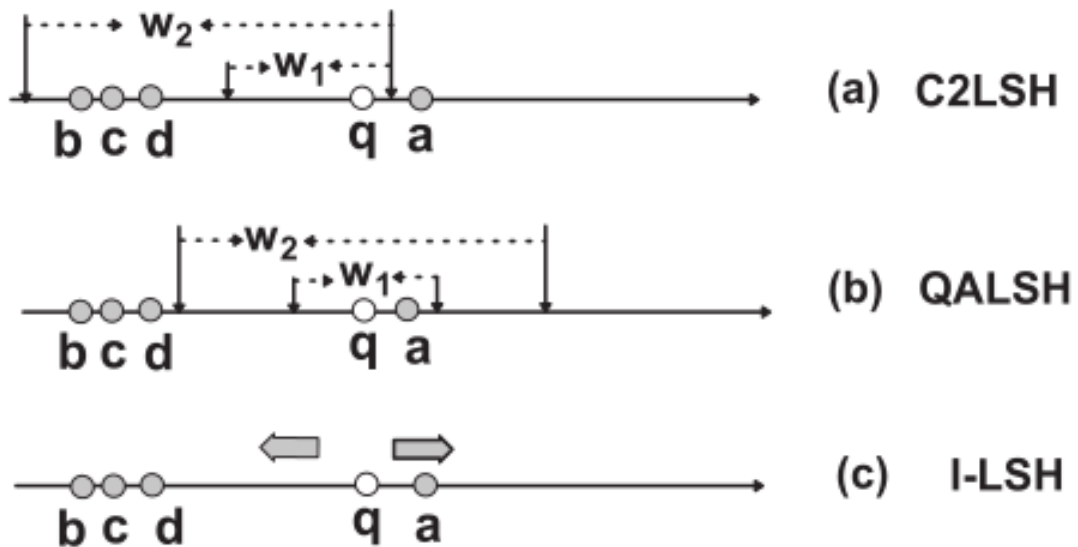


图 1: 指数增长桶宽度和增量搜索

3 本文方法

3.1 本文方法概述

原文提出递增的 LSH 方法，即 I-LSH(incremental LSH)，与前文诸多方法最大的区别是它没有采用指数扩展桶宽度的策略，而是采用了自然增量搜索策略来找到与查询对象在某个维度的 LSH 函数映射值最近的对象，如上页图 1(c) 所示。他们表明，I-LSH 的这一策略可以通过在搜索算法中使用近似比率 c ，而不是 C2LSH 和 QALSH 中使用的 \sqrt{c} ，来显著提高性能。存储结构上则遵循 QALSH 的框架来享受 B+ 树带来高效的顺序 I/O。I-LSH 算法由两部分组成：建立索引和查询处理。

3.2 建立索引

建立索引的过程与 QALSH 类似。对于一个给定具有 n 个 d 维数据对象的数据集 D ，我们使用 m 个 2-stable 的哈希函数将每个对象 o 随机投影成 m 个维度的哈希值，用 $h_i(o) = \vec{a}_i \cdot \vec{o}$ 表示第 i 个哈希函数。对于第 i 个哈希投影维度，我们使用 B+ 树为每个对象保存一对信息 $(ID(o), h_i(o))$ ，其中 $ID(o)$ 是对象 o 的 ID，哈希值 $h_i(o)$ 是搜索关键字。

选取 m 个 2-stable 的哈希函数的过程，其实就是从 2-stable 分布（即正态分布）中独立地为 m 个 d 维向量的所有分量取值的过程，这 m 个向量即是 \vec{a}_i 。再如上所说用 d 维的数据对象 o 与这 m 个向量分别作内积，得到 m 个值即是数据对象在对应投影维度上的哈希值 $h_i(o)$ 。把一个数据对象的 m 个哈希值组成一个 m 维向量，就是这个数据对象经过选取的 LSH 函数得到的映射。

3.3 查询处理

同样的， d 维空间中的查询对象 q 也将被映射到 m 个投影维度中。然后数据集的数据对象及它们在 m 个投影维度的哈希值，将根据与查询对象在对应维度的投影间的距离被递增访问。如果一个对象的 m 个哈希值中有 αm 个被访问过，那么该对象就变成一个候选对象。这一访问过程有两个终止条件。条件 1：如果 I-LSH 找到了 βn 个候选对象，算法将停止；条件 2：如果当前最近的对象 o_{min} 满足 $\|o_{min}, q\| \leq cR$ ，其中 R 是当前搜索的范围，算法将停止。请注意， α 、 β 和 c 是预先定义的参数。其中 $0 < \alpha < 1$ ，用于控制数据对象成为候选对象所要求各个维度的哈希值最少被访问的次数； $0 < \beta < 1$ ，用于控制候选对象的数量达到终止条件时的阈值； $c > 1$ ，即是 c -ANN 查询的参数 c 。

3.4 I-LSH 算法伪代码

下页的 Procedure 1 展示了 I-LSH 的伪代码。在每次迭代中，算法选择与 q 的某个投影距离最小的数据点 o 作为当前点（第 4 行），并在第 7 行计算相应的搜索范围 R 。如果 o 已经被访问过 αm 次，算法将计算其与 q 的欧氏距离，并将其加入候选点集（第 9 行至第 11 行）。如果两个终止条件中的任何一个得到满足，该算法将返回当前最近的点作为查询对象 q 的 c -ANN。

4 复现细节

4.1 与已有开源代码对比

I-LSH 这一科研成果的复现工作没有任何相关源代码可供参考，复现过程均由本人研读原文理解算法后独立完成。

Procedure 1 Incremental LSH search(\mathcal{B}, q)

Input: \mathcal{B} : m B+ tree indices for object IDs and hash values;
 q : the query object;
 ω : the initial bucket width;

Output: o : the c -ANN object

```
1  $n_{can} := 0$ ;  $d_{min} := 0$ ;  
2 Apply  $m$  hash function on  $q$ ;  
3 while  $n_{can} < \beta n$  do  
    4  $o \leftarrow$  next object with smallest projected distance;  
    5  $i \leftarrow$  the projection dimension  $o$  comes from;  
    6  $r \leftarrow |h_i(o) - h_i(q)|$ ;  
    7  $R \leftarrow \frac{2r}{\omega}$ ;  
    8  $cn(o) = cn(o) + 1$ ;  
    9 if  $cn(o) == \alpha m$  then  
        10 compute  $\|o, q\|$  and update  $o_{min}$ ;  
        11  $n_{can} = n_{can} + 1$ ;  
    end  
    12 if  $R_{min} \leq cR$  then  
        13 break;  
    end  
14 return  $o_{min}$ ;  
end
```

4.2 实验环境搭建

安装 17.4.0 版本的 Microsoft Visual Studio Community 2022 (64 位) - Current 开发工具，内含产品 Microsoft Visual C++ 2022，提供使用 C++ 语言开发项目的实验环境。

4.3 界面分析与使用说明

运行项目后会随机初始化 LSH 函数和数据向量，完成数据向量的 LSH 映射，然后分别将数据向量及其在各个投影维度的哈希值按照哈希值大小排序保存在对应投影维度的存储结构中。之后会提示输入想要查询 c -ANN 的向量，依照向量的分量连续输入并敲击回车即可。

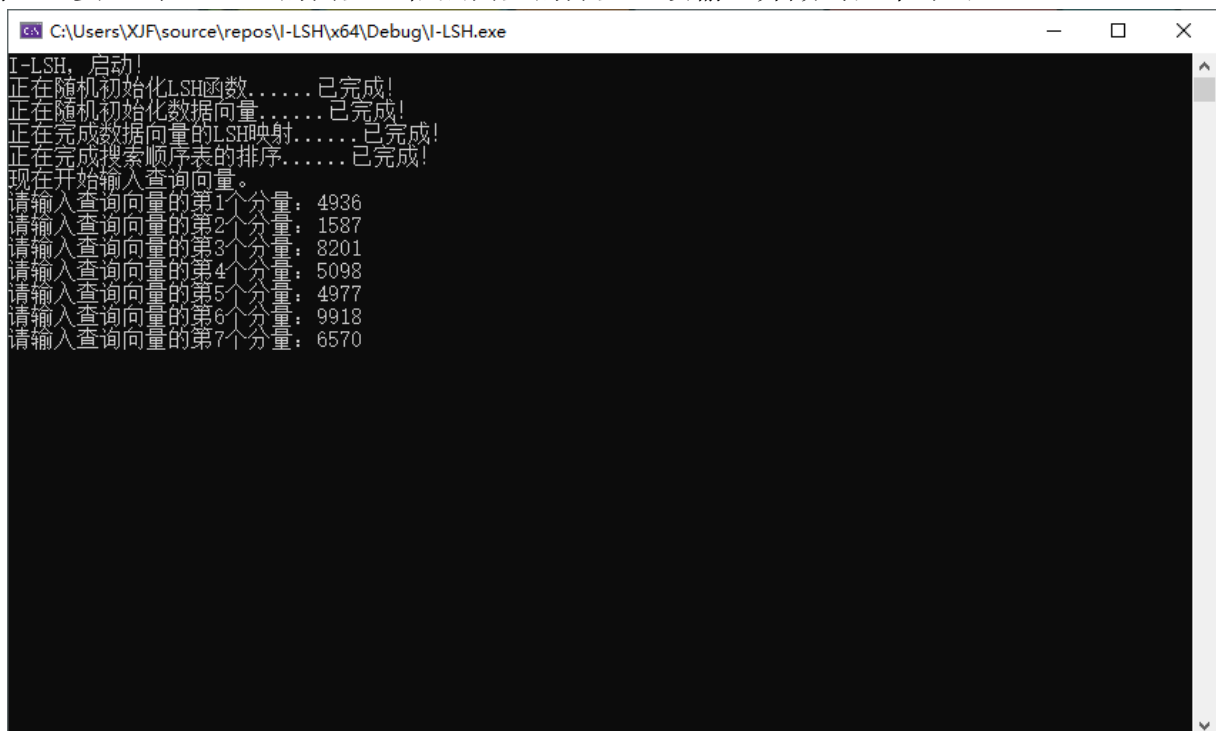
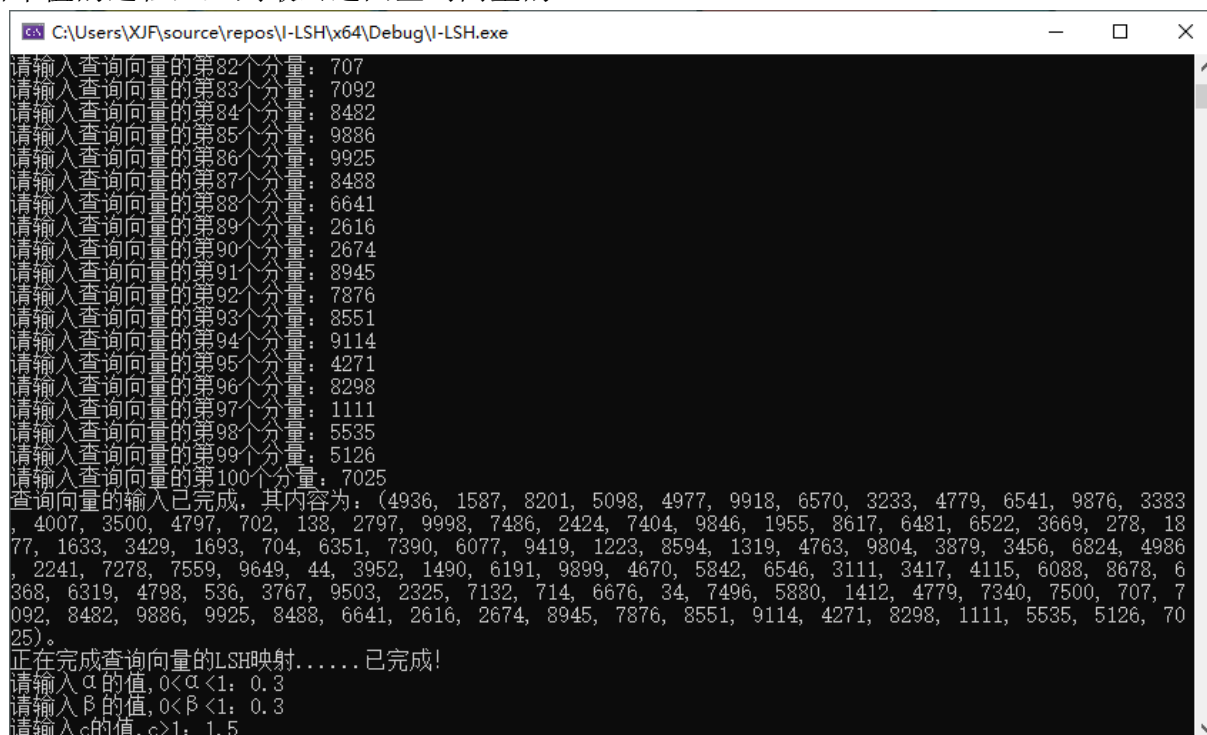


图 2: 输入查询向量

完成查询向量的输入后，项目同样会自动完成其 LSH 映射。此外还需要输入 I-LSH 算法中可设置的三个参数： α 、 β 和 c ，其功能已在 3.3 节中得到阐述。参数设置完成后，项目就会进行迭代递增搜索哈希值的过程，直到最终返回查询向量的 c -ANN。



```
C:\Users\XJF\source\repos\I-LSH\x64\Debug\I-LSH.exe
请输入查询向量的第82个分量: 707
请输入查询向量的第83个分量: 7092
请输入查询向量的第84个分量: 8482
请输入查询向量的第85个分量: 9886
请输入查询向量的第86个分量: 9925
请输入查询向量的第87个分量: 8488
请输入查询向量的第88个分量: 6641
请输入查询向量的第89个分量: 2616
请输入查询向量的第90个分量: 2674
请输入查询向量的第91个分量: 8945
请输入查询向量的第92个分量: 7876
请输入查询向量的第93个分量: 8551
请输入查询向量的第94个分量: 9114
请输入查询向量的第95个分量: 4271
请输入查询向量的第96个分量: 8298
请输入查询向量的第97个分量: 1111
请输入查询向量的第98个分量: 5535
请输入查询向量的第99个分量: 5126
请输入查询向量的第100个分量: 7025
查询向量的输入已完成, 其内容为: (4936, 1587, 8201, 5098, 4977, 9918, 6570, 3233, 4779, 6541, 9876, 3383, 4007, 3500, 4797, 702, 138, 2797, 9998, 7486, 2424, 7404, 9846, 1955, 8617, 6481, 6522, 3669, 278, 18, 77, 1633, 3429, 1693, 704, 6351, 7390, 6077, 9419, 1223, 8594, 1319, 4763, 9804, 3879, 3456, 6824, 4986, 2241, 7278, 7559, 9649, 44, 3952, 1490, 6191, 9899, 4670, 5842, 6546, 3111, 3417, 4115, 6088, 8678, 6, 368, 6319, 4798, 536, 3767, 9503, 2325, 7132, 714, 6676, 34, 7496, 5880, 1412, 4779, 7340, 7500, 707, 7, 092, 8482, 9886, 9925, 8488, 6641, 2616, 2674, 8945, 7876, 8551, 9114, 4271, 8298, 1111, 5535, 5126, 7025)。
正在完成查询向量的LSH映射.....已完成!
请输入α的值, 0<α<1: 0.3
请输入β的值, 0<β<1: 0.3
请输入c的值, c>1: 1.5
```

图 3: 输入参数

4.4 创新点

本人的 I-LSH 复现，与原文的实现最大的区别在于存储结构的选取。原作者通过将所有数据对象在同个维度的投影哈希值存在一棵 B+ 树中，从而利用 B+ 树的有序性来快速搜寻到查询对象在某个维度的投影离哪个数据对象的投影点最近。然而最普通的顺序表已经能通过顺序查找来实现这一功能，只要先用二分查找法确定查询对象在某个维度的投影大致在哪个数据对象的投影旁边，之后根据投影值顺序向左向右找到该维度中未访问过的数据对象，从两侧各一个中选出离查询对象投影更近的那一个即可。

原作者为 m 个维度的数据对象投影设置 m 棵 B+ 树，本人复现时则改成为 m 个维度设置 m 张顺序表。相比原版的方式，这么设置存储结构使得在某个维度搜寻包夹查询对象投影的最近两个数据对象投影的时间开销会略微提高，但数量级都是对数级别，没有根本性区别；而随后找到该维度中未访问过的最近数据对象这一步，顺序查找的实现在两种存储结构中完全没有区别，B+ 树的叶子节点其实也是关键字的线性表。因此本人的复现虽然牺牲了一些可接受的时间开销，但是大幅简化了存储结构的实现工程。

5 实验结果分析

复现项目工作完毕后，即会如下页图 4 所示返回查询向量在随机生成的数据向量集中的 c -ANN。如果想要保持当前的数据向量集和 LSH 函数继续测试，可以输入 y 并敲击回车重新输入查询向量，进行新一轮的 c -ANN 查询。

图 4: 返回查询向量的 c -ANN

6 总结与展望

本次的复现项目没有算法性能的实机运行对照实验，这是因为没有原作者的源代码可供参考，原论文也只是一个算法理论推导为主的论文，重点并不在于具体的实现而是算法的设计方面。原论文中是在两个数据集 **Tiny** 和 **Million Song** 上，用 I-LSH 对照 QALSH 和 SRS 这两种更早的 LSH 算法来对比进行 c -ANN 查询所需要的 I/O 次数。其中 **Tiny** 这一数据集所在的网址无法访问，**Million Song** 数据集获取到了，但是内容是百万条歌曲信息，并不是单纯的欧氏空间向量。如果没能将他们映射成欧氏空间向量，也没有这一特殊空间距离的处理方式，我认为对照试验得到的结果没有参考意义，因此将工作重心放在 I-LSH 的算法复现中。

由于时间原因，有些早期预想的工作没能完成。例如递增查找的迭代过程，在所有维度中分别找到查询向量投影值与某个数据向量投影值的距离最小值后，还需要在各个维度间的最小值中选一个整体最小值，这个环节在我最初的计划中可以设计一个败者树的数据结构来加速。虽然相对直接逐个比较选出最小值来说，这么优化减小的时间开销也不是根本性的，但是确实算一种经典空间换时间的优化，有机会的话想往项目中加入这个环节。另外就是测试的数据向量集是随机生成的，没有考虑现实数据集的分布常态，有一定偏向性。这种数据集用于测试 c -ANN 查询的 I/O 性能结果会比较片面，但是作为复现项目的 I-LSH 算法本身没有影响。

从零开始的 I-LSH 复现，作为我跨到计算机类专业以来的第一个项目，体量虽然不大但是也花了不少心思。当然还有很多可以提升的空间，自己也有些早期的规划没能实现变成了遗憾。现在项目中没包含的 B+ 树、败者树等工作以后也会遇到，希望自己能尽快在未来的各种项目中提高计算机编程水平，掌握自主设计优美算法并将其于计算机之上优雅实现的能力。

参考文献

- [1] INDYK P, MOTWANI R. Approximate nearest neighbors: towards removing the curse of dimensionality [C]//Proceedings of the thirtieth annual ACM symposium on Theory of computing. 1998: 604-613.
- [2] DATAR M, IMMORLICA N, INDYK P, et al. Locality-sensitive hashing scheme based on p-stable distributions[C]//Proceedings of the twentieth annual symposium on Computational geometry. 2004: 253-262.
- [3] HUANG Q, FENG J, ZHANG Y, et al. Query-aware locality-sensitive hashing for approximate nearest neighbor search[J]. Proceedings of the VLDB Endowment, 2015, 9(1): 1-12.
- [4] SUN Y, WANG W, QIN J, et al. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index[J]. Proceedings of the VLDB Endowment, 2014.
- [5] LIU Y, CUI J, HUANG Z, et al. SK-LSH: an efficient index structure for approximate nearest neighbor search[J]. Proceedings of the VLDB Endowment, 2014, 7(9): 745-756.
- [6] LIU Y, CHENG H, CUI J. PQBF: i/o-efficient approximate nearest neighbor search by product quantization[C]//Proceedings of the 2017 ACM on Conference on Information and Knowledge Management. 2017: 667-676.
- [7] JEGOU H, DOUZE M, SCHMID C. Product quantization for nearest neighbor search[J]. IEEE transactions on pattern analysis and machine intelligence, 2010, 33(1): 117-128.
- [8] GAN J, FENG J, FANG Q, et al. Locality-sensitive hashing scheme based on dynamic collision counting [C]//Proceedings of the 2012 ACM SIGMOD international conference on management of data. 2012: 541-552.