清 华 大 学

综 合 论 文 训 练

题目：基于 Spark 的分布式近似近邻查询系统

系　　别：软件学院

专　　业：计算机软件

姓　　名：文庆福

指导教师：王建民教授

2015 年 6 月 15 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

近似近邻查询是处理非结构化数据的一项基本而重要的技术，在模式识别、数据挖掘等前沿研究领域有着广泛的应用。随着数据指数式的增长，如何从大规模高维数据中进行尽可能快速、精确地查询成为备受关注的问题。

本文中，我们通过对比研究了基于树结构的索引方法和基于哈希的索引方法，并介绍了其中具有代表性的几种方法。本文采用向量量化的哈希方法，在 Spark 平台上建立了一套近似近邻查询系统，不仅可以降低存储代价，而且在保证高检索准确率的情况下加速查询效率。最终介绍了在三个数据集上进行的实验，验证系统的正确性和可用性。

关键词：近似近邻查询；Spark；乘积量化；索引

ABSTRACT

Approximate nearest neighbor search (ANNS) is a basic and important technique in processing unstructured data, which is widely used in frontier research fields, such as Pattern Recognition and Data Mining. With exponential growth of data, much attention has been paid to the question that how to search in large-scale high-dimensional data as fast and accurately as we can.

In this paper, we study tree-based indexing method and hash-based indexing method comparatively, and introduce several representative method of them. We build an ANNS system on Spark using hashing method based on vector quantization, which can not only reduce storage costs but also improve query efficiency with high accuracy of retrieval. Finally, we introduce experiments that are performed on three dataset to validate correctness and practicability of the system.

Keywords: Approximate Nearest Neighbor Search; Spark; Product Quantization; Indexing

目 录

主要符号对照表

ANNS	近似近邻查询 (Approximate Nearest Neighbor Search)
PQ	乘积量化 (Product Quantization)
SDC	对称距离度量 (Symmetric Distance Computation)
ADC	非对称距离度量 (Asymmetric Distance Computation)
ITQ	迭代量化 (Iterative Quantization)
SVD	奇异值分解 (Sigular Value Decomposition)
API	应用程序编程接口
RDD	弹性分布式数据集 (Resilient Distributed Datasets)
$\|\cdot\|_2$	L_2 范数
$\|\cdot\|_F$	Frobenius 范数
h	子空间中聚类中心数量
m	子空间个数

第 1 章　引言

1.1　研究背景

在今天这个数据爆炸的时代，文本、图像、视频以及音频等非结构化数据呈现出指数级的增长。如何快速、准确地从这个海量的互联网数据库中获取我们想要的信息，是我们不得不面对的一个问题。Google[①]、Bing[②]、Baidu[③] 等提供的文本、图像等搜索服务为我们获取信息带来了极大的便利。而在这些搜索引擎背后的都需要用到的一项技术——近似近邻查询（Approximate Nearest Neighbor Search）。在大规模数据的应用场景下，精确的近似查询需要耗费时间太长，不具有实际应用价值。近似近邻查询可以大幅度缩短查询时间，同时保证查询结果与精确查询结果近似，因此更具有实用性。除了信息检索以外，近似近邻查询技术被广泛应用于模式识别、机器学习、数据挖掘等领域。

近年来，近似近邻查询技术一直都是研究热点之一，但是该技术面临的挑战却没有改变。

一方面，随着互联网上的数据越来越多，需要处理的数据量也越来越大。然而，传统的索引结构一般都是基于小规模数据而设计的单机结构。大规模的数据一般无法做到单机存储，更不用说加载到内存当中进行索引了。这些数据往往存储于分布式系统当中，同时也需要一种分布式的索引结构来支持查询。海量的数据不仅给存储数据带来了压力，同时也给实时数据查询带来了巨大挑战。

另一方面，在非结构化数据处理过程时，往往都会针对非结构化数据提取特征进行处理。为了更好地表示数据，一般特征数据的维度越高，特征表示的准确性也越高。例如，在图像数据处理过程中，我们常常可能用到的 SIFT、SURF 特征都是 128 维的，GIST 特征有 960 维，而 BOVW（Bag of Visual Words）的维度更是高达成千上万维。在如此高维度的数据如何进行快速、高效的检索是一个必须解决的问题。

① https://www.google.com
② http://cn.bing.com
③ https://www.baidu.com

面对大数据处理的现状，Hadoop、Storm、Spark 等一系列的大数据并行计算框架相继涌现出来。本文中，我们基于 Spark 平台，在其上构建一套近似近邻查询系统来实现快速的高维数据检索。我们希望利用 Spark 并行计算的特点，使我们构建的系统能够应对大规模数据进行高效的近似近邻查询。

对于近似近邻查询问题，在不考虑时间效率的情况下，这一问题可以直接通过一种暴力搜索的方式来解决。比如，我们可以直接计算查询数据 q 与数据集合 S 中每一条数据的距离，最终根据距离大小，选取出距离最近的前 n 个数据。但在现实中，由于数据集合 S 的规模非常大，这种朴素的查询方法的计算时间太长而无法实用。但是，如果从规模比较大的数据集合 S 上删选出一个非常小待选集合 S'，之后再在集合 S' 上进行朴素的暴力搜索选取出前 n 个近邻数据，这时的暴力搜索的时间效率是可以接受的，整个查询过程的时间效率和准确率就取决于删除出待选集合 S' 的过程。待选集合 S' 大小与查询准确率有着密切关系，一般来说，集合 S' 越大，查询准确率越高，但是最终的暴力搜索阶段的时间会变长；集合 S' 越小，则查询效率越低，暴力搜索阶段的时间越短。因此，如何快速地选取出一个大小合适、相关性高的待选集合 S' 就成了近似近邻查询问题的关键。

1.2 主要工作

本文的主要的关注点是在高维空间中的近似近邻查询问题，通过研究对比现有的近似近邻查询方法，了解几种不同方法各自的特点。在 Spark 框架下实现了一套的近似近邻查询系统。最终，通过实验来验证准确性以及测试时间效率。

总体而言，本文的主要工作包括三个方面：

- 算法研究：通过阅读文献，调研现有的近似近邻查询方法，并对现有方法进行比较和分析。
- 系统设计与实现：基于算法研究的基础，在 Spark 平台上设计并实现了一套基于乘积量化的近似近邻查询系统。该系统不仅可以对数据进行编码压缩以减少空间占用，而且利用并行计算可以大幅度加快查询速度。
- 实验验证：通过在不同的数据集上进行实验，验证系统的正确可靠。

1.3 论文组织结构

本文首先介绍了研究背景和本文的主要工作。在接下来的章节中，第二章中对现有的索引方法的进行了综述并对其中代表性方法进行介绍；第三章首先介绍了 Spark 并行计算框架中弹性分布式数据集与分布式机器学习库 MLlib，之后介绍了乘积量化的近似近邻查询方法以及我们在 Spark 平台上设计与实现；第四章介绍了我们实现的系统在三个不同数据集上进行的实验，对实验结果进行了分析；最终，第五章节对本文工作进行了总结，并指出了本文工作中还需要改进的地方。

第 2 章　索引相关综述

前文已经提到，近邻查询问题最关键的就是在于如何选取出一个近邻候选集合 S'。为了快速地删选出近邻候选集，我们通常会先把原始的待检索数据索引起来。依据不同的索引结构分类，近邻查询的方法一般可以分为基于树结构的索引和基于哈希的索引两大类。

2.1　基于树结构的索引

传统的基于树结构的索引方法有很多，比如 R 树、KD-树、Ball 树等。下面将介绍在 FLANN[1] 近似近邻搜索中用到的两种树结构的索引方法。

2.1.1　随机化 KD-树

经典的 KD-树会将原始的数据空间不断划分成一棵二叉树。它在低维的空间上检索具有非常高效，但随着维度不断增加，KD-树的检索效率会迅速大幅度降低。因此，许多基于 KD-树改进的工作涌现出来。其中，Silpa-Anan 等人提出的一种随机化的 KD-树[2] 的改进方法。原始的 KD-树的方法在空间划分时会选取数据方差最大的维度，在这一维度上将空间一分为二。相比于传统的 KD-树方法，随机化的 KD-树在数据空间切分的时候，并不是固定选择数据方差最大的维度，而是从数据方差比较大前几个维度中随机地选取一个维度进行切分。之所这样处理，是因为在维度较高的情况下，例如 50 维，其实数据在许多维度上的方差可能差别不大。通过随机化的选择，这样有更高的概率可以把数据空间中的点切分地比较均匀，从而可以提高近似近邻查询的效率。

2.1.2　分层 K-Means 树

顾名思义，分层 K-Means 树[3] 是在数据需要切分时，使用 K-Means 聚类的方法将数据切分成 K 份。依照这一方法，在每一层中都使用 K-Means 聚类，当数据的数量少于 K 个时，我们就可以直接将这 K 个节点作为叶子节点。这样，分层 K-Means 树就可以看作是一棵 K 叉树。利用分层 K-Means 树作近似近邻查询的时候，当遍历到某个父亲节点，首先在其子节点当中，选取出一个距离查

询数据最近的子节点，然后再优先遍历这个子节点。显然，这样的查询效率是非常高的，能够快速地找到近邻候选集。但是从分层 K-Means 树的建树过程来看，在每一非叶子节点都要执行一次 K-Means 聚类，这种算法不管从时间效率上还是空间效率上来计算，代价都是非常大的。

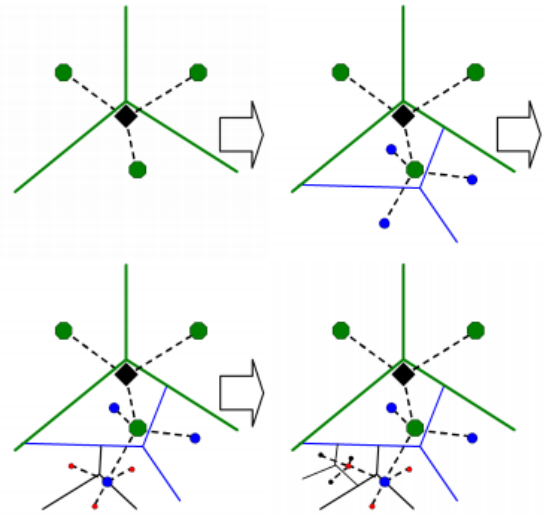


图 2.1　聚类中心为 3 的层次 K-Means 树建树过程

注：图像来源[3]

2.2　基于哈希的索引

传统基于树结构的索引方法最大的不足就是空间占用过大，随着维度的不断增长，空间代价成倍增长。因此，我们需要对原始的数据集空间进行压缩编码以节省空间。基于哈希的索引方法是将高维的数据压缩成二进制编码的形式，这类方法在大规模的图像、文本、视频、音频等非结构化数据检索上取得不错的效果。根据哈希函数形式的不同，我们可以粗略地将哈希索引分为两类——汉明嵌入和向量量化。

2.2.1　距离度量

在近似近邻查询中，计算两个数据的距离是不可避免的。在介绍汉明嵌入和向量量化之前，首先介绍一下距离度量方式。

由于哈希的方法将数据进行了编码压缩，我们可能无法直接在原始空间中计算距离。因此这时候可能需要编码空间中距离与原始空间中距离的转换，根

据不同转换方式产生两种距离度量方式，分别是对称距离度量（SDC）和非对称距离度量（ADC）。

对于两个数据 x 和 y，x 是查询数据，y 是原始数据空间中的一个数据，$q(\cdot)$ 是哈希函数。对称距离的度量方式就是指 x 和 y 之间的距离可以近似地用 $q(x)$ 和 $q(y)$ 之间的距离来表示，用公式表示也就是 $d(x,y) \approx d(q(x),q(y))$。汉明嵌入中用汉明距离[①]近似原始距离就是一种典型的对称距离的度量方式。非对称距离的度量方式是指 x 和 y 之间的距离用 x 和 $q(y)$ 之间的距离近似表示，也就是 $d(x,y) \approx d(x,q(y))$。

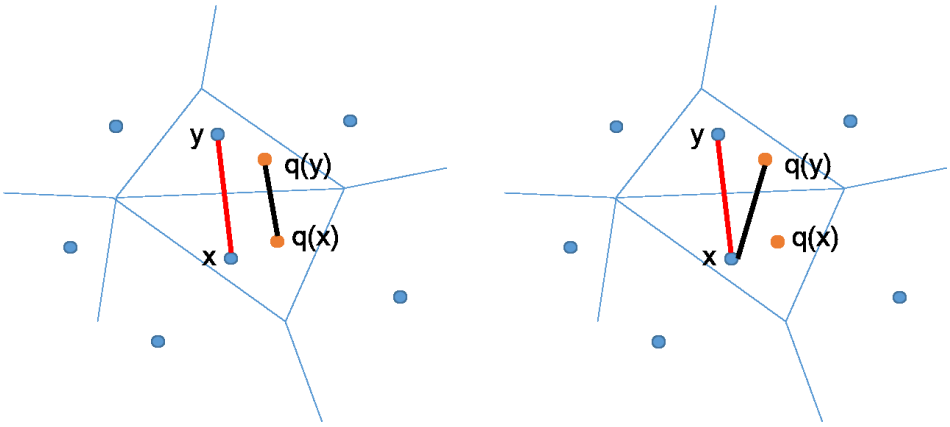


图 2.2　对称距离度量与非对称距离度量

图 2.2 中分别展示了对称距离度量方式和非对称距离度量方式。其中左侧是对称距离对量方式，右侧是非对称距离度量方式。红色的线表示实际距离，黑色线表示近似距离。一般而言，非对称距离的度量方式会比对称距离的度量方式更准确。

2.2.2　汉明嵌入

汉明嵌入的哈希方法就是要寻找一个映射，对于任意的对象 $x \in S$ 都能映射到汉明空间中的二进制串 $b(x) \in \{0,1\}^d$。这种映射是保持距离的，也就是在原始空间中距离大小与汉明空间中的距离大小有对应关系。那么，任意两个对象之间的相似度就可以通过其对应二进制串之间的汉明距离来近似计算：

$$sim(x,y) \approx 1 - \frac{2\delta_{Ham}(b(x),b(y))}{d} \tag{2-1}$$

① 汉明距离指两个二进制串进行 XOR 操作之后，结果串中 1 的个数。比如，1011001 和 1001101 的汉明距离是 2。

汉明嵌入的哈希方法比较多，谱哈希（Spectral Hashing）[4] 方法就是其中的代表性方法之一。

2.2.2.1 谱哈希

假设 $\{y_i\}_{i=1}^n$ 是 n 个数据的二进制编码（编码长度为 k），邻接矩阵 $W_{n\times n}$ 用来原始数据点之间的距离。W_{ij} 就表示原始数据空间中第 i 个数据和第 j 个数据之的相似度。原始数据空间中的数据 i 用编码结果用 $y_i \in \{-1,1\}^k$ 表示。为了做到编码保持距离大小对应，也就是说 $W_{i,j}$ 相似度越大，$\|y_i - y_j\|^2$ 就应该越小。因此，所有近邻关系中的平均汉明距离可以用 $\sum_{ij} W_{ij}\|y_i - y_j\|^2$ 表示。那么，我们也就可以得到这个问题的目标函数如下：

$$\begin{aligned} \text{minimize} : & \sum_{ij} W_{ij}\|y_i - y_j\|^2 \\ \text{s.t.} : & \ y_i \in \{-1,1\}^k \\ & \sum_i y_i = 0 \\ & \frac{1}{n}\sum_i y_i y_i^T = I \end{aligned} \tag{2-2}$$

其中 $\sum_i y_i = 0$ 约束条件会使编码中 0 和 1 的数量各占一半，$\frac{1}{n}\sum_i y_i y_i^T = I$ 则表示任意两个编码之间是不相关的。观察发现，上面的式子是一个 NP 难问题，无法在多项式时间内求解。

我们将公式 2-2 进行变形，引入一个 $n \times k$ 大小的矩阵 Y，Y 的第 j 列是 y_j^T 。$n \times n$ 的对角矩阵 D，其中 $D(i,i) = \sum_j W_{ij}$。变换后的公式：

$$\begin{aligned} \text{minimize} : & \ tr(Y^T(D-W)Y) \\ \text{s.t.} : & \sum_i Y^T 1 = 0 \\ & Y^T Y = I \end{aligned} \tag{2-3}$$

相比于公式 2-2 ，我们去掉了 $Y_{ij} \in \{-1,1\}$ 的约束，Y_{ij} 可以在实数集上取值。这样原来的问题就简化成了求解矩阵 $D - W$ 的 k 个特征向量。对这 k 个特征向量根据正负二值化就可以得到我们原来想要的编码矩阵。

2.2.3 向量量化

向量量化的哈希索引是对原始向量空间进行量化压缩，原始向量 $\mathbf{x} \in \mathbb{R}^D$ 通过量化函数 q 被映射到 $q(\mathbf{x}) \in C = \{\mathbf{c}_i\}$，其中 i 是下标，$\mathbf{c}_i$ 可以称作为码字，而 C 则被称为码本。这种映射可以形式化定义成：$\forall \mathbf{x} \in \mathbb{R}^D$，$\exists \mathbf{c}_i \in C$，$q(\mathbf{x}) = \mathbf{c}_i$。整个量化过程中的平均量化误差 E 就可以定义为：

$$E = \frac{1}{n} \sum_{\mathbf{x}} \|\mathbf{x} - q(\mathbf{x})\|^2 \tag{2-4}$$

其中 $\|\cdot\|$ 表示欧氏距离，而 n 是数据的总量。对于给定的原始数据集合 S，我们的目标是找到一个码本 C 以及对应量化函数 $q(\mathbf{x})$ 使得量化误差 E 最小。在最小化量化误差过程中，不同的限制条件就对应了不同的量化方法。

2.2.3.1 K-Means

假设有一个包含 n 个 p 维数据点的集合，$\mathcal{D}\{\mathbf{x}_j\}_{j=1}^n$，k-means 算法会将这 n 个数据点聚成 k 类，同时用聚类中心来代表每一个聚类的数据。假如矩阵 $C \in \mathbb{R}^{p \times k}$ 的列向量由 k 个聚类构成，每一列都是一个聚类中心，$C = [\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_k]$。k-means 量化的目标函数如下：

$$l_{\text{k-means}} = \sum_{\mathbf{x} \in \mathcal{D}} \min_i \|\mathbf{x} - \mathbf{c}_i\|_2^2 \tag{2-5}$$

$$= \sum_{\mathbf{x} \in \mathcal{D}} \min_{\mathbf{b} \in \mathcal{H}_{1/k}} \|\mathbf{x} - C\mathbf{b}\|_2^2 \tag{2-6}$$

其中 $\mathcal{H}_{1/k} = \{\mathbf{b} | \mathbf{b} \in \{0,1\}^k \text{ 且 } \|\mathbf{b}\| = 1\}$，$\mathbf{b}$ 是一个 1 对 k 编码的二进制向量（包含 1 个 1，k-1 个 0）。

k-means 量化模型浅显易懂，使用最朴素的枚举方法就可以将数据点映射到聚类中心。这种映射过程将每一个原始数据压缩到了 $\log_2 k$ 比特的数据，所需要消耗的存储空间随着 k 的增长而线性增长。

2.2.3.2 ITQ

ITQ（Iterative Quantization）[5] 的量化方法是在 2011 年的 CVPR 会议上提出，这一方法较之前的哈希方法在准确率上有了显著提升。

ITQ 方法首先是对原始数据集空间进行 PCA 降维，将原始的 $\mathbb{R}^{n \times d}$ 空间降维成 $\mathbb{R}^{n \times c}$ 空间。此时，再考虑将降维后的数据集进行二进制编码。从降维后的空

间中取出 $\mathbf{v} \in \mathbb{R}^c$，其对应的编码 $\text{sgn}(\mathbf{v})$ 可以看做事超立方体 $\{-1,1\}^c$ 中的顶点。此时，欲使整体的量化误差最小，就是要使原始数据与编码后的数据的欧氏距离最小，$\min \|\text{sgn}(\mathbf{v}) - \mathbf{v}\|^2$。对于编码前的原始数据集我们用 $X \in \mathbb{R}^{n\times d}$ 来表示，经过 PCA 降维后用 $V \in \mathbb{R}^{n\times c}$ 表示，编码后的数据集用 $B \in \{-1,1\}^{n\times c}$ 来表示。整体量化误差：

$$Q(B,R) = \|B - VR\|_F^2 \tag{2-7}$$

其中 $\|\cdot\|_F$ 是 Frobenius 范式，而 R 是正交矩阵，作用是对 V 进行旋转对齐。为何要引入这个旋转矩阵 R 呢？从下图（2.3）可以看出，左侧是 PCA 降维后的数据集 V，中间是随机旋转后的数据集，右侧是最优化旋转后的数据集 VR。右乘一个旋转矩阵 R，让待编码的数据绕数据中心进行旋转到合适状态，每个数据点就可以用距离其最近的数据顶点来表示。下图中可以用 $(-1,-1),(-1,1),(1,1),(1,-1)$ 这是四个点来表示。此时，所有的数据点到超立方的顶点的距离和最小，也就是量化误差最小。

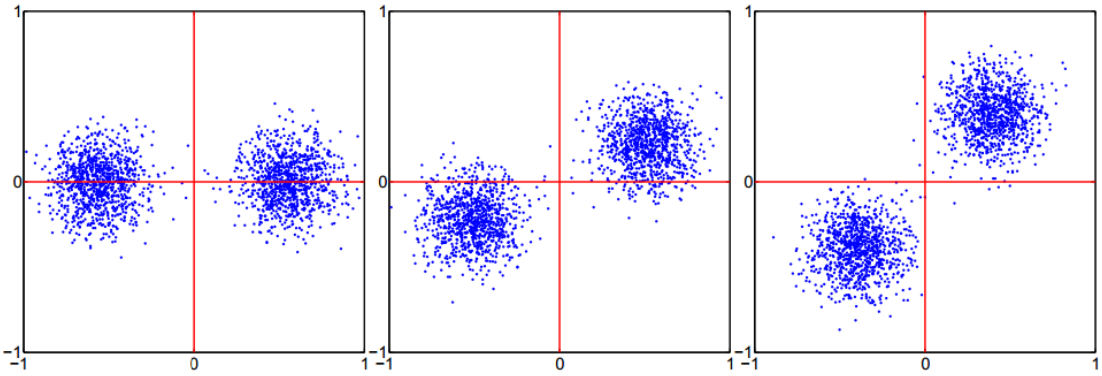


图 2.3　ITQ 旋转过程

注：图像来源[5]

这样，整个问题的目标函数就变成了 $\min\|B - VR\|_F^2$。这个公式中有两个未知量，编码后的矩阵 B 和旋转矩阵 R。所以，这个问题的求解要考虑采用交替迭代的方法。先对一个随机生成的矩阵进行 SVD 分解得到一个正交矩阵作为 R 的初始值，此时 R 已知，就可以用 $B = \text{sgn}(VR)$ 来求解 B；当 B 已知后，可以对 $B^T V$ 进行 SVD 分解求解 R。既然 R 求出，又可以重新一次迭代，固定 R 求 B，如此交替迭代就可以求解出该问题。

第 3 章　基于 Spark 的分布式近似近邻查询系统

3.1　Spark 的弹性分布式数据集与分布式机器学习库

Spark 是一个基于 MapReduce 的通用的大数据并行计算框架，最初由 UC Berkeley AMP Lab 开发。Spark 的架构是在 Hadoop 基础上的改良，继承了 MapReduce 的优点，它与 Hadoop 最大不同之处就是内存计算，Hadoop 将计算过程的中间数据存储在磁盘上，而 Spark 一般是用内存来存储数据，从而提高计算效率。

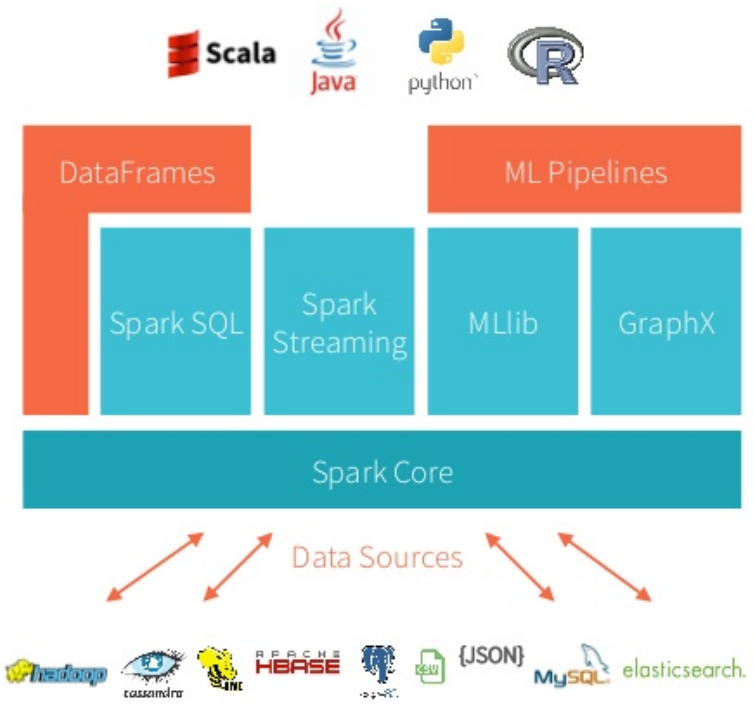


图 3.1　Spark 生态圈

注：图像来源①

① http://www.slideshare.net/rxin/stanford-cs347-guest-lecture-apache-spark?utm_source=tuicool

在 Spark 的生态圈中，Spark 的数据来源可以不仅可以从本地系统或者 HDFS 上各种格式的数据，还可以从 HBase、Cassandra 等数据库中的数据。基于在 Spark 的内核，还提供了诸如 Spark Streaming、Spark SQL、MLlib、GraphX 等数据处理的库。开发者可以使用 Scala、Java、Python 甚至是 R 语言编写 Spark 应用程序完成数据处理的任务。

3.1.1 弹性分布式数据集

弹性分布式数据集（Resilient Distributed Datasets，下文简称 RDD）[6] 是 Spark 中的分布式内存的抽象。相比于 Hadoop 中的计算过程，RDD 可以被缓存在内存当中，每一次的计算产生的结果都可以保留在内存当中。对于迭代计算，这样多次迭代的计算每次可以将结果保存在内存中，下一次迭代又可以直接从内存中读取数据计算，从而避免了大量的磁盘读写操作，大大节省了计算时间。

一般来说，RDD 的创建是通过 SparkContext 来实现，主要包含有两种创建来源：一是从支持的文件系统（或支持的数据库）读取创建；二是从内存数据集合生成。不同于 Hadoop 中仅有 Map 和 Reduce 操作，RDD 还支持其他类型的操作，主要分为转换操作、控制操作和行为操作三类。转换操作顾名思义，就是将一个 RDD 操作之后转换为另一个 RDD，包括 map、flatMap、filter 等操作。控制操作主要用来将 RDD 缓存到内存中或者磁盘上，比如 cache、persist、checkpoint 等操作。行为操作主要分为两类：一类是变成集合或标量的操作，另一类是将 RDD 外部文件系统或数据库的操作。Spark 中的所有对 RDD 的操作，只有当执行行为操作时，才会执行之前的转换或控制操作。例如，我们先对 RDD 执行 map 操作，然后执行 reduce 操作，在 map 操作时，Spark 并不会真正执行，只是记录，只有执行 reduce 操作时才会真正一起计算。这一特性称为惰性计算（lazy computing）。图 3.2 是对 RDD 的操作流程示例。

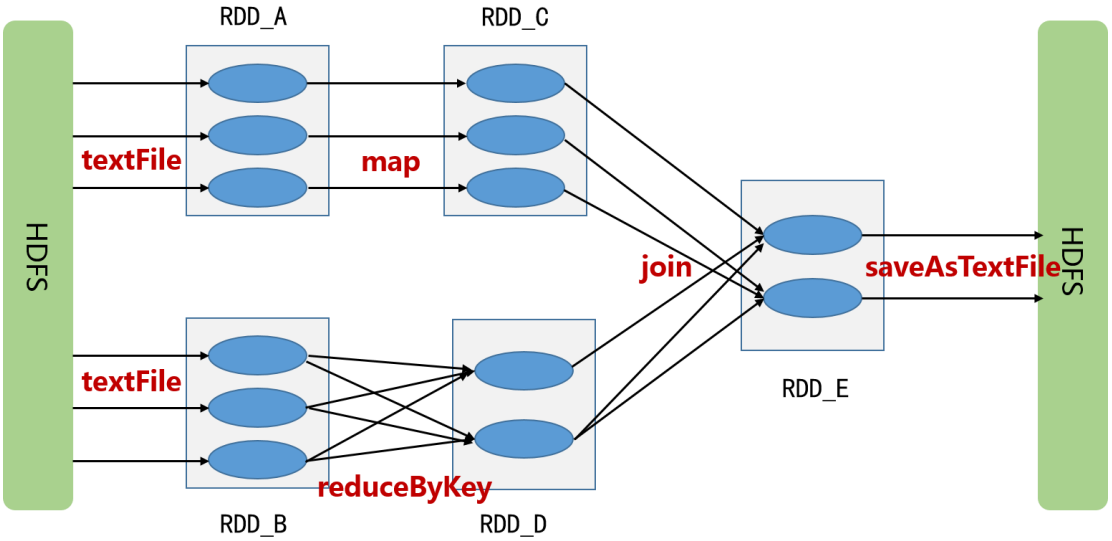


图 3.2　RDD 操作流程示例

RDD 的持久化是 Spark 中 RDD 的一个重要特性。通过 RDD 的持久化，我们可以将 RDD 缓存到内存或者磁盘上。默认地，RDD 并不会进行缓存，每次计算需要用到 RDD 时，都需要重新计算获取 RDD，这样是非常低效的，特别是对于一些迭代的计算。因此，我们需要考虑在计算过程中，将一些重复用到的 RDD 进行缓存操作，这样我们我们只要第一次计算出 RDD，以后每次用到相同的 RDD 时就不用在重复计算，从缓存中直接读取就可以了。RDD 的 cache() 和 persist() 函数就是用于缓存的，其中 cache() 函数是指将 RDD 缓存在内存当中，而 persist() 根据参数可以将 RDD 进行不同级别的缓存。

RDD 本身自带有容错机制，通过记录 RDD 的演变过程以便在任务执行失败的时候能够恢复出原有的数据，而不需要通过备份的形式实现容错。例如，当前的 RDD 的因任务执行失败而数据丢失，系统则会根据记录的 RDD 演变关系回溯到未丢失的祖先 RDD，重新根据转换操作计算出一个新的 RDD 进行恢复。

3.1.2　分布式机器学习库 MLlib

随着数据量的不断增大，传统的单机式的机器学习算法实现已经无法满足大数据的要求，分布式机器学习算法为解决大数据机器学习的提供了可能。MLlib 是基于 Spark 构建的一个常用分布式机器学习算法和工具库，目前支持的算法包括分类算法、回归算法、聚类算法、协同过滤算法、降维算法等。MLlib

中的分布式机器学习算法，相比于以往的算法，在时间效率有了明显提升。下图（3.3）是逻辑回归算法在 Hadoop 和 Spark 上的执行时间对比。

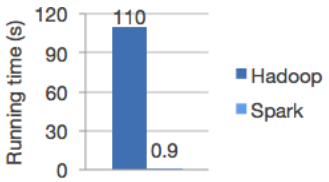


图 3.3　Hadoop 和 Spark 上逻辑回归算法效率对比

注：图像来源①

MLlib 中已经实现的一些常用机器学习算法可以供使用者调用，在 Spark 上利用 MLlib 实现分布式地实现大规模数据的机器学习的任务正是一个不错的选择。

3.2　乘积量化

与之前提到的 K-Means 的量化方法一样，乘积量化（Product Quantization）[7]也是向量量化方法的一种。假设我们需要量化压缩 128 维的向量到 64 比特，采用 K-Means 的量化方法的话，需要有 2^{64} 个聚类中心，这样不管是从 K-Means 聚类所需要的时间还是从存储聚类中心所占的空间来看，都是不可行的。

对于上面同样的问题，在乘积量化的算法中，我们首先将原始的数据空间划分为 m 个不相交的子空间，也就是将 128 维的向量切成 m 个长度为 $128/m$ 的子向量。在每个子空间里，分别对子空间中的子向量集合进行 K-Means 聚类，聚类中心数量为 h。这样我们就可以用 $1\cdots h$ 这些编号来对子向量进行编码，m 个子向量的编码组合在一起就构成了原始向量的编码。这样，原始空间中一个 128 维的向量就可以压缩到一个 $m\log_2 h$ 比特编码表示，从而大大节省了空间。

① http://spark.apache.org/

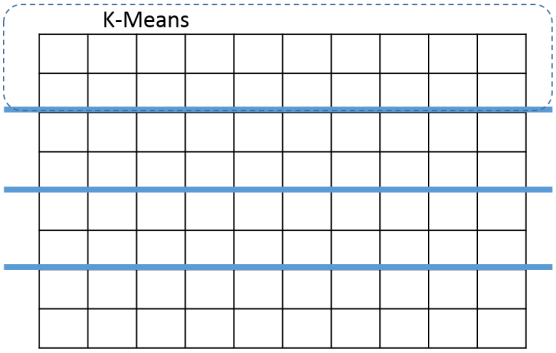


图 3.4　PQ 算法中子空间的划分

更为一般来说，我们可以得到乘积量化方法的目标函数如下：

$$l_{\mathrm{PQ}}=\sum_{i=1}^{n}\min\left\|\mathbf{x}_i-\begin{bmatrix}C^1\mathbf{b}_i^1\\ \vdots\\ C^m\mathbf{b}_i^m\end{bmatrix}\right\|_2^2 \tag{3-1}$$

其中 $\mathbf{x}_i$ 的维度是 p，$\mathbf{b}_i^j\in\{0,1\}^h$ 且 $\|\mathbf{b}_i^j\|=1$，$j\in\{1,\cdots,m\}$。在上面式子中，$C^j(j\in\{1,\cdots,m\})$ 就是我们需要求解的矩阵。在编码过程中，整体的码本（codebook）C 就可以用笛卡尔积的形式表示，$C=C^1\times\ldots\times C^m$。码本的大小就是所有子空间中聚类中心数量的乘积，根据前面的假设，共有 m 个子空间，每个子空间聚类个数为 h，所以码本数量就是 $k=h^m$。求解过程其实并不复杂，正如前文提到，在每个子空间中做 K-Means 聚类就可以求解出码本，这样我们就可以利用码本对每个子空间中的子向量进行编码，从而对原始向量进行编码表示。整个算法的复杂度就和向量维度 p、子空间数量 m、子空间聚类中心数量 h 有关，存储码本所需要的空间复杂度为 $O(mhp)$。

表 3.1　乘积量化与 K-Means 量化的空间占用对比

	聚类中心	编码长度	空间占用
K-Means 量化	k	$\log k$	$O(kp)$
乘积量化	h^m	$m\log h$	$O(mhp)$

注：$h^m=k$

从表 3.1 是乘积量化算法和 K-Means 量化算法的空间占用对可知，当 $m=1$ 时，乘积量化就退化成普通的 K-Means 聚类量化了。此外，h 取值越大，不仅计算时间复杂度越大，而且空间复杂度也越大，进而也会使得在查询时的时间复

杂度变大。因此，选择一个合适的 m 和 h 是非常重要的。论文[7] 中指出，$m = 8$ 和 $h = 256$ 是比较合适的选择。

3.3 基于乘积量化的近似近邻查询

刚刚我们已经介绍了乘积量化的编码压缩方法，那么如何将这一方法运用在近似近邻查询当中呢？采用乘积量化的方法进行近似近邻查询也是一个机器学习类方法，遵循学习类算法的主体流程。整个近似近邻查询算法的主要流程如下：

- 利用部分数据集，依照乘积量化的方法训练出码本。
- 在原始的数据集上，采用训练出的码本对其进行编码压缩。
- 对于任意的查询向量，度量它与编码压缩后的码字之间的距离，从而得到近邻集合。

3.3.1 训练码本

首先是训练集的选择，训练集的选取对整个算法是非常关键的，最终近邻查询的准确率一定程度上取决于训练集的好坏。在选择训练集上有两点需要注意，一是训练集的规模大小，二是训练集的代表性。训练集的规模既不宜过大也不能太小，过大会带来过拟合问题，过小则会欠拟合。通常，训练集的规模大小约为原始数据集的 1/10 比较合适。此外，训练集还应该尽可能得有代表性，尽可能广泛地分布于整个数据空间，这样才能使得训练出来的码本才能更好的量化原始数据空间，更准确地对原始数据集进行编码。

在具体训练码本的过程中，如章节 3.2 中所介绍的一样，首先将训练集中的数据划分到 m 个子空间。在每个子空间中，对所有的子向量数据在进行 K-Means 聚类，可以得到 h 个聚类中心，也就得到每个子空间的码本。算法的伪代码如算法 1 所示。

3.3.2 编码压缩

如算法 2 所示，训练得出每个子空间中的码本之后，将其应用到原始数据集上进行编码压缩。首先，同样也是将原始数据集划分到 m 个子空间。然后在每个子空间中，对每一个数据的子向量，分别用训练出来的码本进行编码，也就

算法 1 训练码本

输入: 训练集 C 矩阵，子空间聚类数量 $subCenters$，聚类算法最大迭代次数 $maxIterations$

输出: 码本模型数组 $model$

function PQ_TRAIN($C, subCenters, maxIterations$)
    uniformly split C by rows into m part
    for $i = 1 \rightarrow m$ **do**
        $model[i] \leftarrow KMEANS\_TRAIN(C[i], subCenters, maxIterations)$
    end for
    return $model$
end function

是用训练好的 K-Means 模型进行预测，可以知道每个子向量隶属于哪个聚类中心，即子向量应该用哪个码字来编码了。这样，整个数据集中的数据都可以用编码来进行表示。完成编码压缩之后，编码后的数据集相比于原始数据集，存储空间成倍的缩小了。

算法 2 编码压缩

输入: 码本模型数组 $model$，原始数据集 X 矩阵

输出: 编码后的矩阵 B

function PQ_ASSIGN($model, X$)
    uniformly split X by rows into m part
    for $i = 1 \rightarrow m$ **do**
        for $j = 1 \rightarrow n$ **do**
            $B[i][j] \leftarrow model[i].predict(X[i][j])$
        end for
    end for
    return B
end function

3.3.3 近邻查询

在近似近邻查询的过程中，对于任意一个查询向量 q，我们首先计算出 q 在子空间中对应子向量和子空间中聚类中心之间的距离，将计算出的距离用一个查找表存储好。现在我们计算查询向量 q 和编码数据集上的每个数据之间的距离。在每个子空间中，因为我们已经计算出了子向量与聚类中心之间距离，利用查找表，我们就可以迅速得出每个数据在子空间中与向量对应距离。最终将不同子空间中同一向量距离加和。这样就得到了最终的每个向量和查询向量 q 之间的距离。利用一个优先队列，我们就可以快速获取出前 k 个近邻集合。具体算法流程可以参考算法 3

算法 3 近邻近邻查询

输入: 码本模型数组 *model*，编码数据集 B 矩阵，查询数据 q

输出: 近邻集合 *result* 数组

```
1:  function PQ_SEARCH(model, B, q)
2:      for i = 1 → m do
3:          for j = 1 → model[i].subcenters do
4:              d[i][j] ← COMPUTE_DIST(model[i].center[j], q)
5:          end for
6:      end for
7:      for i = 1 → n do
8:          dist[i] ← 0
9:          for j = 1 → m do
10:             dist[i] ← dist[i] + d[i][B[j][i]]
11:         end for
12:     end for
13:     result = GET_TOPK(dist)
14:     return result
15: end function
```

3.4 基于 Spark 的分布式近似近邻查询

3.4.1 算法流程设计

算法的总体流程设计图如图 3.5 所示，首先在训练数据集上进行码本的训练，得到码本模型后将其运用在原始数据集上进行编码压缩，从而可以将原始数据进行编码表示。最终，对于任意一个查询向量，通过近似近邻查询算法在编码数据集上找出近邻候选集合。

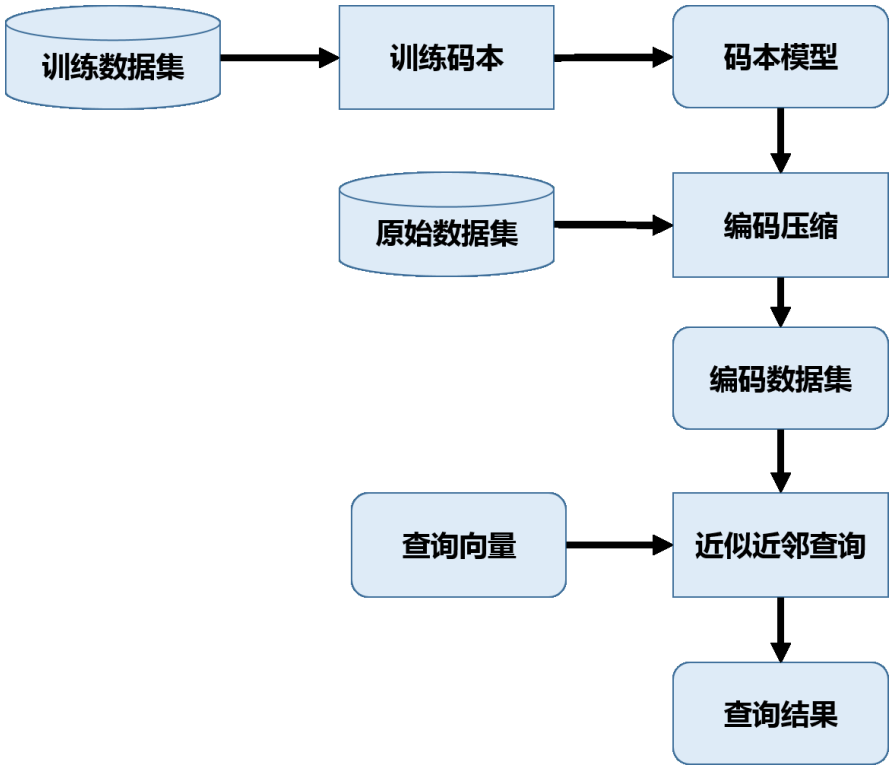


图 3.5 算法总体设计图

3.4.2 数据结构

在 Spark 上，分布式程序的编写必须依赖分布式的数据结构 RDD，RDD 分布式地存储在不同节点上，这样才能保证程序分布式地执行。因此如何合理设计程序的 RDD 非常重要。

在 Spark MLlib 库中提供了一种自带的数据结构 BlockMatrix。BlockMatrix 是用 RDD 构建的分布式矩阵，其中 RDD 的类型是（(Int, Int）, Matrix)。(Int, Int）是 Matrix 的下标索引，那么 BlockMatrix 的每一个元素都是一个带下标索

引的矩阵。BlockMatrix 还提供了一些自带的函数可供调用，如 add、multiply。

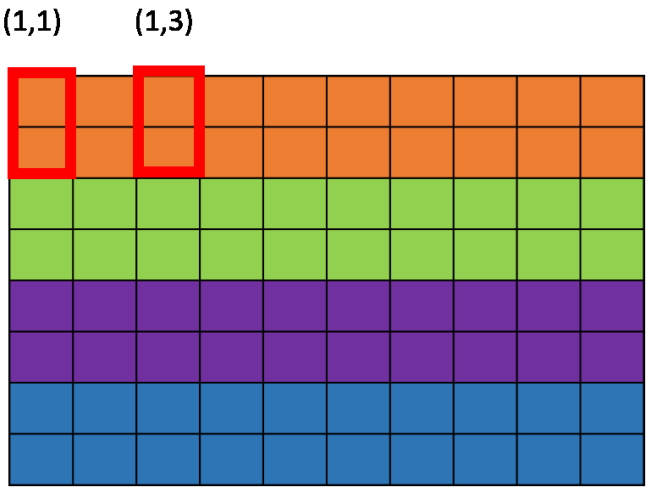


图 3.6　BlockMatrix 划分方式

图 3.6 是一个示例的 BlockMatrix 的划分方式，图中的 8×10 的矩阵被划分成 4 个子空间，每个子空间上有 10 个子向量。那么我们需要一个 4×10 的 BlockMatrix 来存储，红色的框中的向量就对应了一个 matrix，相应 matrix 的下标索引标示在图中了。在上一章节的算法中已经说明，我们需要划分 m 个子空间，每个子空间中有 n 个子向量，因此我们用 $m \times n$ 的 BlockMatrix 数据结构来表示数据是比较合适的。具体而言，我们算法中的训练集数据、原始数据、编码后的数据等都是需要用 BlockMatrix 这个数据结构来存储表示的。

3.4.3　系统优化

集群系统上的分布式程序一般来说都会需要考虑通信的问题，即使 Spark 中 RDD 已经进行了很好的封装，让开发者可以不用直接进行底层的数据管理（通信、容错），只需要通过操作上层的一些 API 就可以了，但是数据通信的问题在 Spark 中还是一个不能忽视的，只有了解底层的通信机制才能利用 API 编写出高效的程序。一般来说，我们优化的原则是尽可能地减少数据的通信，特别是通信量比较大的数据通信。

数据都是分布式存储，如果某个操作与两个分布式数据同时关联，比如两个大小相同分布式矩阵对应元素相加，那么必须要两个分布式数据中分块的一一对齐。具体到本算法中，根据上一小节所提的数据结构的划分方式，假设存储训练数据集的矩阵用 C 表示，存储原始数据集 BlockMatrix 用 X 表示，存储

编码后数据集的矩阵用 B 表示，这些都是分布式地存储在集群系统当中的。由于子空间的划分，我们会在每个子空间中生成自己的码本模型。如果码本模型是分布式存储，那么在编码阶段，我们需要将子空间对齐。Spark RDD 的 join 操作确实提供了按照键值进行对齐操作，但这一过程可能就会产生一些数据分区重组操作。这种分区重组的操作需要进行磁盘读写和网络通信，因此我们应该尽量避免分区重组操作。我们考虑采用 RDD 的广播操作，一次性将只读的数据广播到集群上的所有节点，每个节点上的任务在执行过程中可以直接读取，而不需要考虑对齐问题。在我们算法中，我们选择将计算出的码本模型广播到所有节点以便编码阶段使用。

第二点优化就是利用之前提到的 Spark RDD 的持久化机制。Spark RDD 的持久化可以将 RDD 数据缓存到内存或者磁盘上，以后用到 RDD 的时候不要再次重复计算，从而节省时间效率，这一机制特别适合迭代计算中使用。在我们的算法中，我们选择将一些反复用到 RDD 缓存在内存当中。

3.5 本章小结

本章中首先介绍了 Spark 的弹性分布式数据集合分布式机器学习库 MLlib，接着介绍了乘积量化的哈希方法，通过乘积量化的方法将原始数据集进行编码压缩。之后，我们还介绍了乘积量化哈希方法在近似近邻查询中应用。通过训练码本、编码压缩、近似近邻查询三个过程，我们就可以实现一套完整的近似近邻查询算法。此外，本章节还介绍了在 Spark 计算框架上的这一算法的实现，主要介绍了算法总体流程设计、数据结构的选择以及系统的优化方法。

第 4 章　实验与分析

本章节主要介绍在 Spark 集群上进行近似近邻查询算法的实验，主要包括实验环境介绍、实验数据集介绍、实验结果展示以及实验分析。

下图是 Spark 集群系统的工作模式图。驱动程序（driver）会和集群的管理器（cluster manager）相连接，驱动其为集群其他节点分配资源。在分配完毕以后，驱动程序会将你的应用程序（Scala 中的 jar 包）发送到各个节点的线程池（executor）。之后驱动程序会调配任务给各个节点执行。

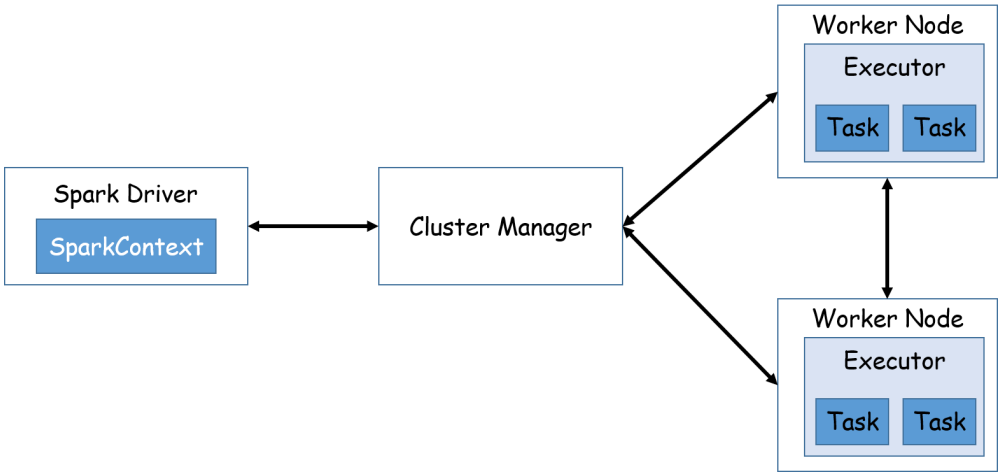


图 4.1　Spark 集群系统工作模式

4.1　实验环境

本次实验中关于 Spark 部分的实验都是在由 4 台机器构成的 Spark on YARN 集群系统上完成的。其中，每台机器配置均相同，配置如下：

- 操作系统：Red Hat Enterprise Linux Server release 7.0
- 处理器信息：Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz
- 逻辑核数：8 核
- 内存大小：60G

关于 MATLAB 部分的实验是在一台单机上完成，配置如下：

- 操作系统：Windows 8

- 处理器信息：Intel(R) Core(TM) i3 CPU M 380 @ 2.53GHz
- 逻辑核数：4 核
- 内存大小：6G

4.2 实验数据集

实验过程中主要使用到了三个数据集，分别是 SIFT1M[①]、GIST1M[①]、CIFAR-10[②]。实验主要对比上一章节中介绍的方法在 Spark 集群系统上与 MATLAB 单机系统上的近似近邻查询的召回率以及相应时间消耗，此外也对比在不同的参数下，Spark 集群系统上的运行结果。

4.2.1 SIFT1M

SIFT1M 是近年来被广泛应用于近似近邻查询方法准确率和召回率的衡量。这个数据集中的数据都是 128 维的 SIFT 特征向量，包含三部分：待索引的原始数据集（base）、训练数据集（learn）、查询数据集（query）。其中待索引的原始数据集合有 10^6 个向量，训练数据集有 10^5 个，查询数据集共有 10^4 个。

4.2.2 GIST1M

GIST1M 这个数据集中的数据都是 960 维的 GIST 特征向量，包含三部分：待索引的原始数据集（base）、训练数据集（learn）、查询数据集（query）。其中待索引的原始数据集合有 10^6 个向量，训练数据集有 5×10^5 个，查询数据集共有 10^3 个。

4.2.3 CIFAR-10

CIFAR-10 数据集被广泛用于在计算机视觉领域做目标识别、图像分类等任务的。它是从 80 million tiny images[③] 数据集中挑选出的一个带标签的子集，由 60000 张 32×32 的彩色图像组成。它们被分成 10 类，每个类别有 6000 张图片。10 个类别分别是 airplane、automobile、bird、cat、deer、dog、frog、horse、ship、truck。

① http://corpus-texmex.irisa.fr/
② http://www.cs.toronto.edu/ kriz/cifar.html
③ http://groups.csail.mit.edu/vision/TinyImages/

在本次实验中，我们对每张图片提取出 320 维的 GIST 特征表示，这样我们就得到了 60000 个 320 维的 GIST 向量。我们从中选取 1000 个作为查询数据，剩余的作为训练数据和待检索数据集。在编码数据集上进行近似近邻查询出前 500 张图片，根据原有的类别标签来衡量查询的准确率。

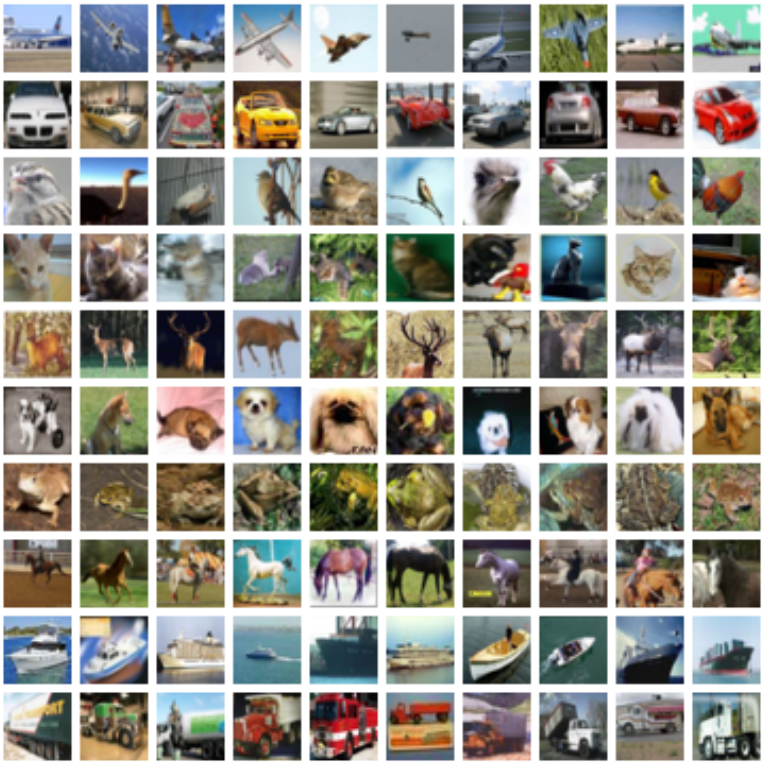


图 4.2　CIFAR-10 数据集中图片示例

4.3　实验结果与分析

4.3.1　SIFT1M 对比实验

本实验室在 SIFT1M 数据上进行，通过比较 Spark 集群上的乘积量化的近似近邻查询方法与同样方法在 MATLAB 单机上的实验的召回率和时间消耗的对比。

在 Spark 集群系统上的程序参数设置，我们采用 yarn-client 模式在集群上运行程序，设置 executor 的数量为 40，每一个 executor 的内存大小限制为 5G。在对 SIFT1M 数据集近似近邻查询实验中，我们设置子空间切分数量 m 为 8，子空间中的聚类中心数量为 h 为 256，因此总的聚类中心的数量就是 2^{64}，编码字

节的大小就是 64 比特。h 取 256 是比较合适的，恰当的子空间聚类中心数量一方面可以使得查找表的大小不会过大，另一方面保证训练和编码的时间也不会太长。在这次实验中，我们采用 recall@R 作为查询效果好坏的衡量标准，其中 R 是检出数据集的大小。SIFT1M 数据中有 10^4 个查询向量，对于一个查询向量，通过整个近似近邻查询计算，我们在原始向量中找到前 100 个近邻向量，并分别取 R 为 1、2、5、10、20、50、100，计算前 R 个向量中出现最近邻向量的次数 s，那么衡量标准就可以由公式 recall@$R = s/10^4$ 计算得出。表 4.1 和表 4.2

表 4.1　Spark 与 MATLAB 上召回率对比

	Spark	MATLAB
recall@1	0.23	0.29
recall@2	0.33	0.40
recall@5	0.48	0.53
recall@10	0.60	0.62
recall@20	0.72	0.73
recall@50	0.85	0.81
recall@100	0.92	0.88

表 4.2　Spark 与 MATLAB 上时间对比

	Spark	MATLAB
训练时间（s）	1193.04	1574.09
编码时间（s）	12.42	49.93
单次查询时间（s）	2.89	181.71

分别显示的是在 Spark 上与 MATLAB 中的召回率对比以及所用时间对比。仅从表 4.1 来看，Spark 集群系统上程序的召回率与 MATLAB 单机版本的相差不大。在保证召回率差不多的情况下，我们再看表 4.2，从表中我们可以看出在训练阶段的时间消耗上，Spark 比 MATLAB 耗时要少，但是并没有成倍数减少。在编码阶段时间消耗，Spark 集群上用时仅为 MATLAB 的 $1/4$ 左右。到了查询阶段，单次耗时 MATLAB 是 Spark 集群系统上单次耗时的 70 余倍。

4.3.2　GIST1M 实验

对 GIST1M 数据集的实验都在 Spark 集群系统上完成，Spark 集群系统的配置上面的实验相同，在 yarn-client 模式执行程序，executor 的数量为 40，executor 的内存大小限制为 5G。在本次实验中，我们分别采用 32 比特、64 比特、96 比特、128 比特、256 比特这些不同编码长度进行实验，观察实验中 recall@R 的变化，R 是检出数据集大小。

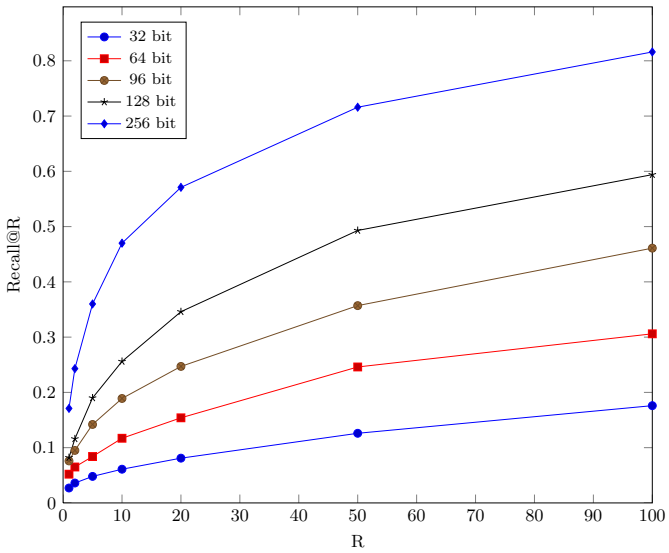


图 4.3　GIST1M 上不同编码长度下的召回率对比

从上图 4.3 中可以看出，在同等检出数量下，编码长度越长，召回率越高。由此，也可以证明算法的正确性。

4.3.3　CIFAR-10 实验

在 CIFAR-10 数据集上的实验中，Spark 集群系统的配置以及程序的参数设置和 GIST1M 中完全相同。分别对不同长度的编码进行对比实验，根据数据集原有的类别标签，我们采用准确率 precision 来衡量检出效果，图中 R 是检出数据的数量。

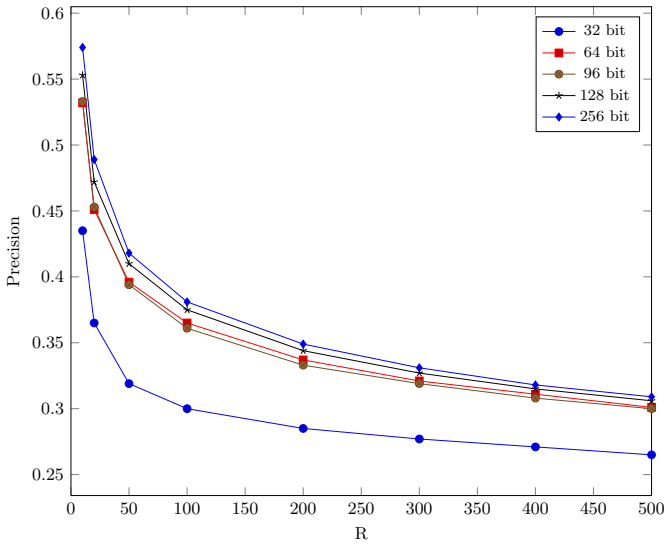


图 4.4　CIFAR-10 上不同编码长度下的查准率对比

从图 4.4 中，我们可以看到在同样的压缩编码下，准确率随着检出数量增大而不断降低。在相同的检出数量条件下，编码长度越长，检出的准确率越高。下图 4.5 是实验中对于一个查询图片检索出前 25 个相似图片的示例。

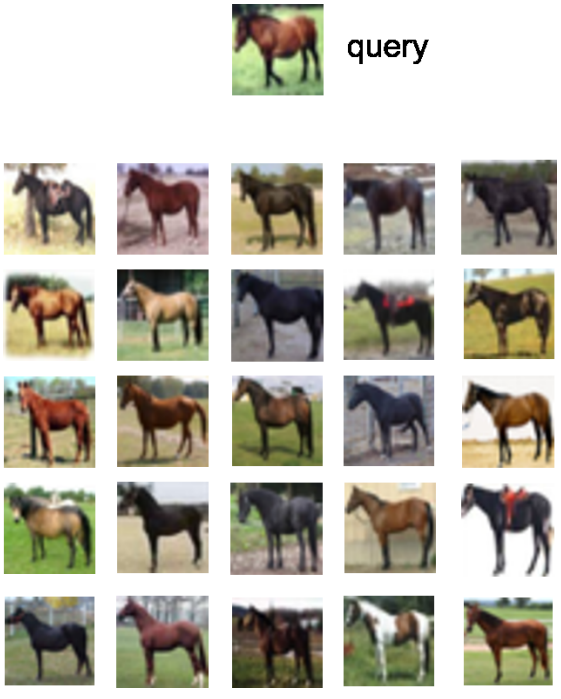


图 4.5　CIFAR-10 数据集上图像查询实例

4.3.4 本章小结

本章中主要介绍了在 SIFT1M、GIST1M、CIFAR-10 三个数据集上分别进行实验。在 SIFT1M 数据集上，通过对比 Spark 集群系统上和 MATLAB 上的召回率以及查询时间，可以看出我们的近似近邻查询系统的正确性以及查询效率的加速情况。在 GIST1M 上的实验说明，我们系统有比较好的扩展性，同时结果也表明了查询系统的正确性。在 CIFAR-10 数据集上的实验表明我们的系统能够应用于大规模图像检索当中。

第 5 章　总结

5.1　本文工作总结

本文针对大规模高维数据的近似近邻查询问题，首先对比研究了两大类索引结构——基于树结构的索引和基于哈希的索引。在树结构的索引中，本文介绍了两种用于近似近邻查询的索引树，分别是随机化 KD-树和分层 K-Means 树。在哈希的索引方法中，本文着重介绍谱哈希、K-Means 量化和 ITQ 量化三种哈希方法。在高维空间中，传统的基于树结构的索引方法会使得树的层数随着维度增加而不断增加，空间占用较大。从查询时间上来看，单次查询时间不仅受索引树深度的影响，同时也和查询数据的维度有关。基于哈希的方法对原始空间的数据集进行编码压缩，这类方法不仅减少了数据的占用空间，同时也一定程度上可以提高查询效率。

通过对乘积量化的哈希方法的深入研究，我们在 Spark 平台上实现了一套基于乘积量化哈希方法的近似近邻查询系统。系统中采用基于 RDD 构建的分布式矩阵 BlockMatrix 来存储数据，使用 RDD 的持久化和减少通信数据量来不断优化。最终，通过在 SIFT1M、GIST1M 两个数据集上的实验证明，该系统一方面对数据进行了编码压缩，从而可以大幅度降低空间占用；另一方面在保证查询准确率的同时，通过并行计算的方式可以大大提高查询的时间效率。同时，我们也在 CIFAR-10 数据集上进行图像检索的任务，从而可以看出本系统的实用性。

5.2　未来工作展望

本文已经实现了一套基于 Spark 的近似近邻查询系统，采用乘积量化的方式进行哈希编码，但并没有在 Spark 上实现一套高效的索引方法。目前已被广泛应用的倒排索引方法或是新提出的多重倒排索引方法[8] 都是可用于近似近邻查询的索引方法。此外，编码方法也还存在改进的空间，近年来提出的 Cartesian K-Means[9] 在乘积量化的基础上，引入了旋转矩阵，在切分的子空间中将数据

旋转矫正来减小数据和聚类中心的距离使得编码更加准确，这种方法在检索准确率上比乘积量化更高。

本文的实验是在 SIFT1M、GIST1M、CIFAR-10 三个数据集上进行，未来的工作中可以考虑在 SIFT1B[①] 数据集和更大规模的图像或文本数据集上进行实验。最终，可以考虑基于近似近邻查询系统，在一个大规模的真实数据集上构建一个以图搜图的搜索引擎。

① http://corpus-texmex.irisa.fr/

插图索引

表格索引

公式索引

参考文献

[1] Muja M, Lowe D G. Fast approximate nearest neighbors with automatic algorithm configuration. International Conference on Computer Vision Theory and Application VISSAPP'09). INSTICC Press, 2009. 331–340

[2] Silpa-Anan C, Hartley R. Optimised kd-trees for fast image descriptor matching. CVPR. IEEE Computer Society, 2008

[3] Nister D, Stewenius H. Scalable recognition with a vocabulary tree. Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2, Washington, DC, USA: IEEE Computer Society, 2006. 2161–2168

[4] Weiss Y, Torralba A, Fergus R. Spectral hashing. In: Koller D, Schuurmans D, Bengio Y, et al., (eds.). NIPS. Curran Associates, Inc., 2008. 1753–1760

[5] Gong Y, Lazebnik S. Iterative quantization: A procrustean approach to learning binary codes. Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition, Washington, DC, USA: IEEE Computer Society, 2011. 817–824

[6] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), San Jose, CA: USENIX, 2012. 15–28

[7] Jégou H, Douze M, Schmid C. Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2011, 33(1):117–128

[8] Babenko A, Lempitsky V S. The inverted multi-index. CVPR. IEEE Computer Society, 2012. 3069–3076

[9] Norouzi M, Fleet D J. Cartesian k-means. CVPR. IEEE, 2013. 3017–3024

致　谢

感谢我的导师王建民教授，王老师严谨治学、勤恳工作的态度深深影响了我。

感谢龙明盛学长对我毕设的耐心指导。同时，也要感谢研究组里的各位学长和学姐，没有他们的答疑解惑，我的毕设工作不可能顺利完成。

感谢大学四年言传身教的各位老师，他们不仅教给我文化知识，而且也教会我为人处世之道。

感谢软件学院一字班里的每一位同学，四年来我们共同学习、互帮互助、共同成长。

感谢林梓佳辅导员四年来的悉心指导和照顾，在我最困难的时候，给予我鼓励和帮助，谢谢林导！

最后，还要感谢在背后一直默默支持我的家人和朋友，谢谢你们！

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

附录 A　外文资料的书面翻译

在快速近似近邻查询中选取近邻候选集最高效的方法是什么？

A.1　摘要

近似近邻查询是一项被广泛运用于许多应用场景中的基础而重要的技术。它主要包括两个阶段：选择近邻候选集合和在候选集合进行最朴素的暴力搜索。只有第一个阶段有提升的空间。在现有的大部分方法中，大多是通过将空间近似量化。计算查询数据与被量化的数据（如聚类中心或者比特序列）之前的距离，然后选择适当大小的候选集。这类方法准确性的衡量就是通过在候选集合度量计算来实现。这看似合理但却忽略了一个重要的问题：没有考虑到选取过程的计算时间代价。在本文中，我们提出一种新的近似近邻查询方法，同时关注选取候选集过程的代价。现有的一些方法都用到一些代价比较大的技术，比如说排序或者堆，但我们提出的方法并非如此，是一种非常高效的检索方法。在 10^8 的 SIFT 特征数据检索上，相比于现有最好的方法，我们成功减少了 1/3 左右的计算时间。

A.2　简介

给定一个查询数据，查找出与之相近的数据，这就叫做近邻查询。近似近邻查询是在近邻查询基础上的近似。近似近邻查询问题被主动地研究和广泛地应用于许多场景，如近似重复检测、大规模物体识别、文档检索、光学字符识别。近似近邻查询的特点就是在节省时间和空间地情况下准确地找出真正的近邻候选集。在本文中，我们关注于计算效率与准确率之前的关系。

下面我们来讨论一下近邻查询问题中查询准确率与计算时间之前固有关系。如果不考虑计算时间，那么近邻查询问题可以用暴力枚举的方法解决，直接计算查询数据与原有数据之间的距离。因此耗时太长，这种朴素的解决方法并不实用，特别是在处理大数据集时。减少计算时间其实就是要减少最终暴力枚举

的数据量。我们先选出一个小的近邻候集合，在这个数据集上进行暴力枚举。只有在选取候选集合的过程需要我们去优化时间效率和准确率。在选取候近邻选集合时，近邻候选集越大，最终查询的准确率就越高，但是这样就会使得最终暴力枚举的时间变长。

目前已有的大多数方法都是通过衡量候选集合的准确性来衡量算法的性能。候选集合的大小决定了暴力枚举的计算时间。然而，选取近邻候选集的过程的计算时间是独立的。因此，忽略掉选取候选集合时间的做法只是觉得了近似近邻查询问题的一部分。

有些文章也适当地考虑了计算时间代价问题。在他们当中，最有效的方法就是多重倒排索引（inverted multi-index，IMI）。然而，在”选取近邻候选集最高效的方法是什么？“这一问题上，我们发现这个问题没有抓住选择近邻候选集过程的本质。下面我们直观上来解释下，假设有 n 个数据，任务是选取近邻候选集，集合大小为 k（$k < n$）。最差的方法就是将他们全部排序，这需要耗时是 $O(n\log n)$。一种更好的方法就是采用有限队列对他们部分排序。这样，每一条数据都被加入到优先队列当中，最终 k 个最近邻的数据留下了。这需要的时间就是 $O(n\log k)$。多重倒排索引用的就是这种方法。然而，还有一种更快的排序方法——桶排序。因为不需要比较序数，它可以在 $O(n)$ 时间复杂度内完成。因此，如果我们能够选择合适的桶，那么选取近邻候选集的过程就可以加快。我们的目标是选取出近邻候选集，所以我们没有必要对桶内的元素按照距离去排序。

对于大规模数据的近似近邻查询问题，我们提出了一种比以往更加高效的近似近邻查询方法。提出的这种方法称为桶距离哈希（bucket distance hashing, BDH）。在实验中，我们在相同平台上比较了这种方法与几种代表性的近似近邻查询方法，分别比较了召回率与计算时间的关系以及召回率与候选集合大小的关系。虽然我们的目标的解决近邻查询问题，但是同样的方法也可以运用到 k 近邻搜索问题当中。

A.3 相关工作

要想高效地选取近邻候选集，通常需要先将数据进行索引。按照索引结构的不同，近似近邻查询方法可以被分为基于树的方法和基于哈希的方法。基于哈希的方法同时也分为数据无关和数据相关两类。前者在索引的过程中不会使

用原始数据而后者需要使用。

A.3.1 基于树的方法

FLANN 是一种基于树索引的代表方法。它可以自动选择随机 kd-树，层次 k-means 和暴力枚举中一种最优的方法以及在给定数据集上调参。在实验过程中，我们会将随机 kd-树和层次 k-means 与我们的方法进行比较。

A.3.2 基于哈希的数据无关方法

局部敏感哈希（locality sensitive hashing，LSH）是一种代表性的基于哈希的数据无关的方法。近年来也有很多人对其进行改善。然而众所周知它的性能比数据相关的索引方法要差。此外，拒局部敏感哈希比数据相关的方法也要执行速度上也要慢很多。

A.3.3 基于哈希的数据相关方法

数据相关的方法可以分为在欧氏空间和汉明空间进行选取近邻集合两种。**在欧氏空间中选取**

这种方法通常使用量化和数据压缩的技术。向量量化（vector quantization，VQ）应用到近似近邻查询中，例如 VQ-index 和 IVFADC。乘积量化（product quantization）被运用在 IMI 当中。正如前面所提到的，IMI 方法要比 IVFADC 方法的效果更好。转换编码被应用在 [3] 中。这一过程可以看做是标量量化（scalar quantization，SQ）。在这个类别中，我们选取 IVFADC 和 IMI 进行比较评价，二者会在下一章节中再介绍。虽然我们实现转换编码，但是它无法和二者比较。**在汉明空间中选取**

这种方法中，数据通常用二进制编码表示。由于位运算 XOR 操作，这样可以减少内存占用和在汉明空间中的距离计算时间。由于这些好的特性，许多近似近邻查询方法都是基于二进制编码的。在他们当中，谱哈希（spectral hashing，SH）就是一种代表性的方法。

为了实现高召回率的长编码，编码长度为 128 比特或者 256 比特是比较常见的。其实处理这么长的编码是不简单的。在汉明空间中选取近邻候选集合的

基准方法是顺序查找。这种方法虽然很快，但却无法再大数据集上应用。有人也许认为树结构或哈希结构能够实现子顺序查找。然而，在汉明空间进行等距离数据的查找，随着距离增大，查找的数据量会呈现出爆炸增长。这使得选取近邻候选集的过程变得不太高效。正如 [16] 中之处一样，使用原始的 LSH 来哈希。然而前面提过，数据无关的方法并不高效。另一个也许稍微复杂一点的方法就是现有的在欧氏空间的近似近邻查询方法。我们将自己提出的方法运用在谱哈希上。但是，并没有将其与现有的代表性方法进行比较。

A.4 现有基于向量和乘积量化的哈希方法

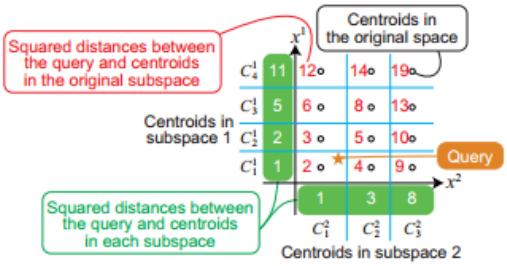
回顾一下 IVFADC 和 IMI 方法可以帮助更好地理解我们提出的方法 BDH。因此在这一章节，我们会比第二章中更加详细介绍他们。

A.4.1 IVFADC

IVFADC 使用 VQ 索引数据可以高效地选取近邻候选集，使用 k-means 聚类算法将数据分成许多个聚类。给定一个查询数据，与之相近的聚类会被检索到，隶属于该聚类的所有数据会被选为近邻候选集。这有可能使得有些数据与查询数据近邻却没有被选取到近邻集合中。然而在相同聚类中心数量下的量化误差最小，VQ 被认为是最好的量化方法。

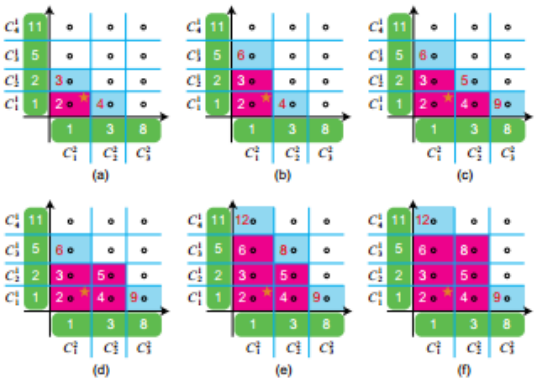
A.4.2 多重倒排索引

作为比 IVFADC 更好的解决方法，Babenko 和 Lempitsky 提出的 IMI 使用乘积量化代替向量量化。PQ 是一种介于 SQ 和 VQ 之间的量化方法。向量空间被划分成很多个子空间，之后再每个子空间上运用 VQ 编码。在相同数量的聚类中心下，PQ 一般会产生比 VQ 更大的量化误差。然而为了达到相同召回率，计算时间可以通过多序列算法（MSA）来减少。



IMI 算法的介绍

上图中回顾介绍了 IMI 算法。根据图片所示，在子空间 1 中有 4 个聚类中心，在子空间 2 中有 3 个聚类中心。聚类中心的用 C_j^i 表示，其中 i 表示子空间标号，j 表示第 i 个子空间中的聚类标号。首先，计算用星号表示的查询数据与子空间聚类中心之间的平方距离。原始的平方距离是每个子空间中的平方距离之和。然后，如果在原始空间中的所有距离都计算完了，中心离查询数据近的就会被查找出。但是，没有必要去计算在原始空间中所有的距离。因为子空间中的中心距离大的中心并不能在过程中发挥作用。因此，他们在 MSA 算法中被忽略不计。



MSA 算法的介绍

图中是 MSA 算法的概览。在 MSA 算法中，每个子空间中的平方距离事先被排序好了。然后，子空间的中心按距离递增顺序一次排列。正如图（a）所示，第一个查找的聚类就是 $C_1^1 \times C_1^2$ 的积，与查询数据的距离为 2。相邻的聚类中心 $C_2^1 \times C_1^2$ 和 $C_1^1 \times C_2^2$ 会在下一次中作为候选集被查找出。

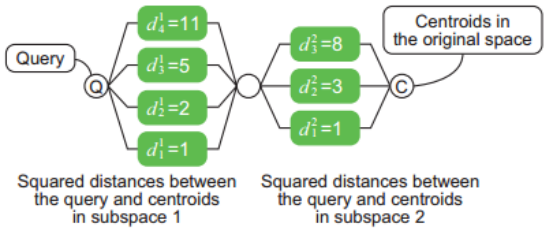
如上所示，MSA 算法比较聚类中心之间的距离并将其按照升序排列。这在近邻查询中是不必要的。在这点上，这种方法就没有抓住近邻查询的本质。图

中描述了子空间数量为 2 的情况。如果空间划分多于 2 的话，那么整个计算的代价就会变大。因此，IMI 算法最佳情形就是子空间数量为 2。我们将会展示子空间被划分成多余 2 同时获得更好的查询效率的算法。

A.5 提出的方法

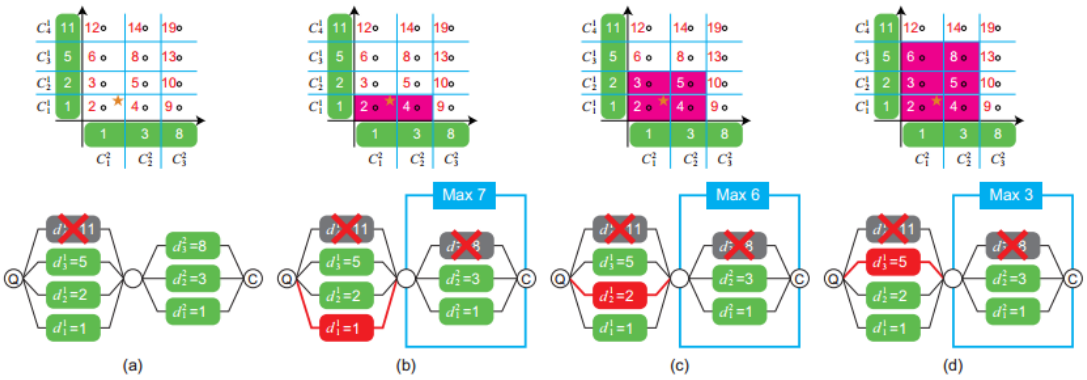
A.5.1 回顾

正如第一章节中所提，我们提出的方法最关键的想法就是在选取近邻候选集的时候不做数据的排序。我们采用分支定界法来代替 MSA 算法解决相同问题。



子空间的距离计算

图中，介绍了我们所提出的算法。图中绘制了查询数据（Q）和聚类中心（C）路径。路径的左右半边分别表示在查询数据和聚类中心之间子空间 1 和子空间 2 中的平方距离 $\{d_j^i\}$。d_j^i 和 C_j^i 的标号示意相同。然后，我们会确定平方距离的上界。整个过程中，上界会不断地增大。在所有路径中，距离小于上界的聚类中心就会被查找出来。



BDH 算法示意图

图中描述了提出的算法流程，平方距离的上界是 8。整个算法流程从（a）到（d）。在（a）中，当上界被设置成 8，路径 $d_4^1 = 11$ 就会立即被删除。之后，子空间 1 中的每条路径都会遍历一遍。在（b）中，路径 d_1^1 被遍历。由于上界是 7，路径上子空间 2 中 $d_1^2 = 1$ 和 $d_2^2 = 3$ 会被选出到近邻候选集。在（c）中，d_2^1 被遍历，由于上界是 6，路径上子空间 2 中 $d_1^2 = 1$ 和 $d_2^2 = 3$ 会被选出到近邻候选集中。

A.5.2 准备

为了更详细地解释我们提出的方法，需要提前做一些定义。我们假设，查询数据 q 和待查询数据集都用向量表示。用 N 和 D 分别表示待查询数据的数量和维度。我们会在待查询数据集上进行 PCA 降维以减小距离度量的误差和获取最大的 u 个主成分（特征值）降序的 $l_1, l_2, \ldots, l_u$。用 $\mathbf{V} = [v_1 v_2 \cdots v_u]$ 表示与 u 个特征值对应特征向量组成的矩阵。u 对应多维哈希表的维度。我们之后会介绍一种自动确定 u 大小的算法。

A.5.3 索引与调参

PQ 将特征空间划分成 p 维的子空间。例如，一个 u 维的向量 $\mathbf{x}$ 被 u 个特征值表示为 $\mathbf{x} = \{x_1, \ldots, x_m\}$，其中 $m = u/p$。然后，在 p 维的子空间上进行 k-means 聚类。

除去 p 以外，我们提出一套自动调参的算法。首先，我们开始为聚类算法进行自动调参选择。目标是要将待查询数据集在第 i 个子空间聚类成 k_i 个类，得到所有 k_i 以及每个子空间中的聚类中心。用 x_{is} 表示第 s 条数据的第 i 个子向量，用 C_j^i 表示第 i 个子空间中的第 j 个聚类中心。然后量化误差第 i 个子空间的 E_i 定义如下：

$$E_i = \sum_{s=1}^{N} \left(x_{is} - C_{q_i(x_{is})}^i \right)^2$$

其中 $q_i(x)$ 函数定义如下：

$$q_i(x) = \arg\min_j \left(x - C_j^i \right)^2$$

因为过大的量化误差会导致距离度量时的误差过大，所以我们要采取策略使得子空间中的 E_{max} 最小。算法 1 介绍了我们提出的算法。它会增加具有最大量化误差的子空间中聚类中心的数量，并在当桶的总数量达到待查询数据的数量时结束。这样，特征空间就会被分成 N_{bkt} 个桶，对应到多维哈希表的哈希大小。通过对人造数据和真实数据的实验中，我们可以找到算法的终止条件。

这个算法同时也可以确定多维哈希表的维度 u。如果在子空间 1 中有 k_i 个聚类中心，那么子空间中的量化误差就小于 E_{max}。假如子空间不需要划分，假设特征值被降序排列，对于 $i = 1,\ldots,m'$，现有的 m' 满足 $k_i > 1$ 而且对于 $i = m'+1,\ldots,\lfloor D/p \rfloor$ 有 $k_i = 1$。然后可以忽略后半部分（$i > m'+1$），u 设置为 $m'p$。

A.5.4 高效选取近邻候选集合

用 c 表示近邻候选集的集合大小。我们提出的算法可以一步一步地选择至少 c 个最近距离的候选数据。算法 2 和算法 3 展示了整个详细过程。其中有一部分在章节 4.1 中已经有所介绍了。算法 2 可以找到距离在下界 L 和上界 U 之间的桶。L 和 U 的初始值分别是 0 和查询数据与最近桶之间的平方距离。只要当前选取的近邻候选集大小 n 小于 c，整个过程会一直循环执行。Δ 表示 L 和 U 的增量，我们设置 Δ 的大小为特征值之和的 $1/100$。

A.6 实验

我们通过实验对比提出的方法（BDH）与一些代表性的近似近邻查询算法，包括 IVFADC 和 IMI。所有方法都用 C++ 语言实现。我们依照 MATLAB 版本的 IVFADC 和 C++ 源码 IMI 算法实现基于哈希的方法。IVFADC、IMI 和 BDH 共用大部分的代码。这样可以避免在实验过程中的差异并且尽可能公平地对比。对于基于树的方法，我们使用 C++ 源码实现的 FLANN。FLANN 自身无法使用于大数据集合。我们通过实验探索每种方法的最佳参数设置。实验中的参数设置见表格。我们使用包含 10 亿个 128 维 SIFT 特征描述符的 BIGANN 数据集和包含大约 8 亿 384 维 GIST 特征描述符的 80 Million Tiny Images。对于前者，我们使用 1M、10M、100M 大小的数据集，1000 个向量用来做查询。对于后者，我们使用 100K、1M、10M 大小的数据集。对于两类数据，都是用最小的数据

实验中的参数设置

Methods	参数	SIFT1M	SIFT10M	SIFT100M	GIST1M	GIST10M
BDH	$\log_2 \lvert C\rvert, P$	20,5	26,3	28,5	22,5	24,5
IMI	$\log_2 \lvert C\rvert$	14	18	20	18	22
IVFADC	$\log_2 \lvert C\rvert$	10	12	14	12	14
RKD	$No.of trees$	8	8	8	8	16
HKM	k	32	64	N	64	32

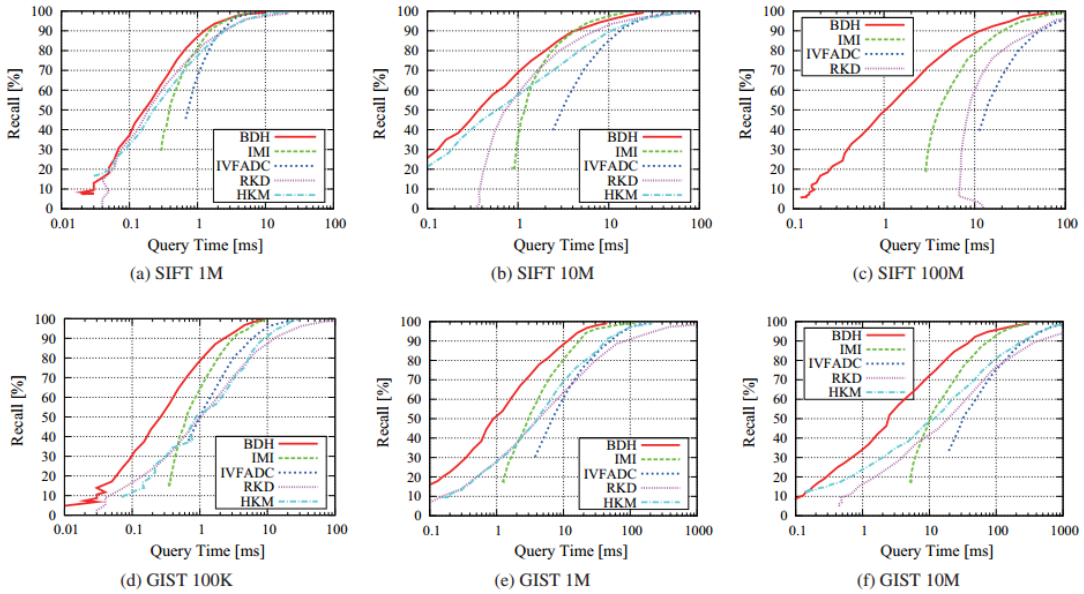
集来做训练。

我们使用有 4 CPUs（AMD Opteron 6174，2.2GHz，12 cores）和 256G 内存的服务器来运行实验。所有的数据都存储在内存当中。所有的程序都是单核单线程执行。

A.6.1 实验 1：召回率与计算时间

我们通过召回率与计算时间关系比较了所有的算法。计算时间是平均一次查询所需要的计算时间。由于计算时间限制，计算时间少的准确率相比之下比较低。图中显示了在 SIFT 和 GIST 数据集上的实验结果。从实验结果看出，我们提出的方法比其他方法的效果都要好很多。其中图（c）中显示在召回率为 90% 时，BDH 算法是 IMI 算法的两倍快，是 IVFDC 的 4.5 快；在召回率为 60% 时，BDH 算法是 IMI 的 2.9 倍快，是 IVFADC 的 9.4 倍快。比较图（a）到图（c），提出的方法的优点随着数据集增大就越明显了。这显示出这个算法的可扩展性好。相反，比较图（d）到图（f），提出的方法的优点并没有变化。

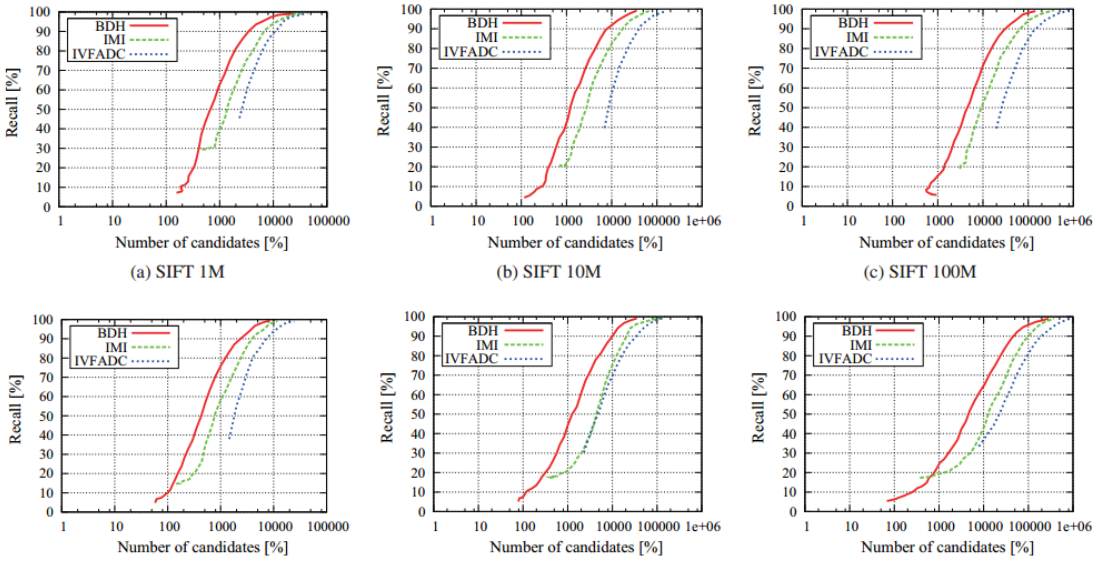
IVFADC 和 IMI 无法扩展是因为相对大规模计算代价。用 G 表示原始空间中的聚类数量，IVFADC 在选择近邻候选集时需要 $O(G)$ 时间来计算距离。IMI 需要 $O(2\sqrt{G})$ 计算距离，需要 $O(\sqrt{G}\log\sqrt{G})$ 在子空间中排序。



实验结果图

A.6.2 实验 2：召回率与候选集大小

我们同时也通过召回率与候选集大小的关系来比较所有算法。为了适合这个标准，我们选择最佳的参数来进行实验。通过图中可以看到，我们的方法要比 IVFADC 和 IMI 方法好。



实验结果图 2

A.7 结论

在近似近邻查询的过程中，只有选择近邻候选集的过程可以变改善。在本文中，我们指出了衡量近似近邻查询算法的计算代价的重要性。计算代价是近似近邻查询算法的衡量中必不可少，研究者忽略计算时间的方法是有缺陷的。我们发现，目前最好的近似近邻查询算法 IMI 并没有把我到选取近邻候选集合的本质。因为近似近邻查询算法只需要选出最近邻的候选集合，所以没有比较去做排序。

最后，我们提出一种新的将计算代价考虑在内的近似近邻查询算法。这种方法基于分支定界算法，可以进行非常高效地检索。我们也提出一种自动调节参数的算法，除了一个参数以外，其余参数可以自动调节到最好。在 100M SIFT 特征实验中，相比于现有最好的算法，我们提出的算法在减少三分一左右的计算时间。在召回率为 90% 和 60% 时，分别是最好算法的 2 倍和 2.9 倍快。

书面翻译对应的原文索引

[1] Iwamura M, Sato T, Kise K. What is the most efficient way to select nearest neighbor candidates for fast approximate nearest neighbor search? The IEEE International Conference on Computer Vision (ICCV), 2013