面向近似近邻查询的分布式哈希学习方法

文庆福[1] 王建民[1),2)] 朱 晗[1] 曹 越[1] 龙明盛[1),2)]

[1)](清华大学软件学院 北京 100084)

[2)](清华大学信息科学与技术国家实验室(筹) 北京 100084)

摘 要 近似近邻查询是信息检索领域中的一项重要技术.随着文本、图像、视频等非结构化数据规模的迅速增长,如何对海量高维数据进行快速、准确的查询是处理大规模数据所必须面对的问题.哈希作为近似近邻查询的关键方法之一,能够在保持数据相似性的条件下对高维数据进行大比例压缩.以往所提出的哈希方法往往都是应对集中式存储的数据,因而难以处理分布式存储的数据.该文提出了一种基于乘积量化的分布式哈希学习方法SparkPQ,并在 Spark 分布式计算框架下实现算法.在传统的乘积量化方法的基础上,该文首先给出了分布式乘积量化模型的形式化定义.然后,作者设计了一种按行列划分的分布式矩阵,采用分布式 K-Means 算法实现模型求解和码本训练,利用训练出的码本模型对分布式数据进行编码和索引.最终,该文构建了一套完整的近似近邻查询系统,不仅可以大幅降低存储和计算开销,而且在保证高检索准确率的条件下加速查询效率.在较大规模的图像检索数据集上进行的实验验证了方法的正确性和可扩展性.

关键词 近似近邻查询;哈希学习;高维索引;分布式计算;Spark

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2017.00192

Distributed Learning to Hash for Approximate Nearest Neighbor Search

WEN Qing-Fu[1)] WANG Jian-Min[1),2)] ZHU Han[1)] CAO Yue[1)] LONG Ming-Sheng[1),2)]

[1)](*School of Software, Tsinghua University, Beijing* 100084)

[2)](*National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing* 100084)

Abstract Approximate nearest neighbor (ANN) search is an important technique in Information Retrieval. With rapid growth of volumes of unstructured data like texts, images, and videos, how to perform efficient and accurate search from large-scale data becomes an inevitable problem. As a key approach to approximate nearest neighbor search, hashing can perform similarity-preserving compression for high-dimensional data. Previous hashing methods are usually applied to centralized data, hence they cannot process distributed data. In this paper, SparkPQ, a novel distributed learning to hash method based on Product Quantization (PQ) is proposed, which is implemented in the Spark distributed computing framework. Based on the seminal Product Quantization (PQ) method, we first give a formal definition of distributed Product Quantization model. Then, we design a distributed matrix partitioned by rows and columns and apply distributed K-Means algorithm to solve the SparkPQ model and train a codebook. We encode and index the distributed data from database with the codebook model. Finally, we build an integrated ANN search system which not only reduces the storage and computation cost substantially, but also speeds up the

收稿日期:2015-09-21;在线出版日期:2016-05-17.本课题得到清华大学信息科学与技术国家实验室大数据科学与技术专项基金、国家自然科学基金(61325008,61502265)和中国博士后基金特别资助项目(2015T80088)资助.**文庆福**,男,1993 年生,硕士研究生,主要研究方向为机器学习与信息检索.E-mail:wqf15@mails.tsinghua.edu.cn.**王建民**,男,1968 年生,博士,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为大数据与知识工程.E-mail:jimwang@tsinghua.edu.cn.**朱 晗**,男,1992 年生,硕士研究生,主要研究方向为机器学习与信息检索.**曹 越**,男,1992 年生,博士研究生,主要研究方向为机器学习与信息检索.**龙明盛**,男,1985 年生,博士,助理教授,中国计算机学会(CCF)会员,主要研究方向为机器学习与大数据系统.

search efficiency with guaranteed search accuracy. A comprehensive empirical study on large-scale image retrieval datasets validates the effectiveness and scalability of the proposed distributed learning to hash method.

Keywords approximate nearest neighbor search; learning to hash; high-dimensional indexing; distributed computing; Spark

1 引　言

在信息技术迅速发展的今天，非结构化数据如文本、图像、视频以及音频等都呈现出指数级的增长. 如何从海量的互联网数据中快速、准确地获取用户想要的信息，是非结构化大数据管理与检索中的一个重要技术问题. 谷歌、百度等互联网公司所提供的文本、图像等搜索服务为我们获取信息带来了极大的便利. 而在这些搜索服务的背后，都需要近似近邻查询(Approximate Nearest Neighbor Search)技术的支持. 在大规模高维数据的应用场景下，精确的近邻查询需要耗费大量存储和计算资源，且查询时间太长、索引系统吞吐量过低，实际应用价值偏低. 近似近邻查询技术可以大幅度缩短查询时间、降低存储和计算开销，同时保证查询结果与精确查询结果近似，因此具有更高的实用性. 除了信息检索以外，近似近邻查询技术被广泛应用于机器学习、数据挖掘、多媒体管理等领域.

近年来，近似近邻查询技术一直是相当活跃的研究方向，新的实现方法不断出现，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，传统的树索引结构一般都是面向小规模数据而设计的单机结构. 大规模数据一般无法做到单机存储，这些数据往往存储于分布式系统中，同时也需要一种分布式的索引结构来支持查询和检索. 海量的数据不仅给数据存储带来了压力，同时也给实时数据查询带来了挑战. 另一方面，在图像、视频、音频等非结构化数据处理过程中，往往都会对数据进行特征提取. 针对特定的任务，为了获得更高的准确度，往往提取出的特征维度都比较高. 例如，在图像和视频数据处理过程中，常用的 SIFT、SURF 特征有 128 维，GIST 特征有 960 维，近年来取得突破的深度卷积网络特征为 4096 维，而 BOW(Bag of Words)词袋特征的维度更是高达成千上万维. 怎样对如此高维度的数据进行快速、高效的检索是一个十分具有挑战性的问题. 哈希方法可以对高维数据进行保持相似性的编码压缩，从而减少了近似近邻查询的存储空间和计算时间.

在数据规模不断增长的今天，越来越多的应用都是基于存储在分布式系统中的大规模数据，例如互联网文本、图像和视频检索等. 现有的很多哈希算法都是在单机的环境下实现的，而在分布式环境下，现有大部分哈希算法都要求将所有数据迁移到同一台机器中进行集中式学习，但这违背了数据的分布式存储方式，带来了很高的数据迁移代价，并且单机能够处理的数据总是很有限的，因此一种有效的分布式哈希方法是非常有必要的.

为有效管理和分析分布式数据，Hadoop、Storm、Spark 等大数据分布式处理系统相继涌现. 本文基于 Spark 分布式计算平台设计并实现了一种分布式哈希学习方法，从而实现对分布式高维数据进行快速准确的检索. 利用 Spark 分布式计算框架的特点构建的系统能够更为高效地应对大规模数据的近似近邻查询任务.

本文第 2 节分别对基于树结构和基于哈希的两大类近似近邻查询方法进行综述；第 3 节介绍向量量化和乘积量化的哈希方法以及 Spark 弹性分布式数据集模型；第 4 节阐述分布式哈希学习方法及其在 Spark 上的设计与实现；第 5 节分析讨论了分布式哈希算法和近似近邻查询系统在大规模图像检索数据集上的实验结果；最终，第 6 节对本文工作进行了总结和展望.

2 相关工作

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线性扫描的方式来解决. 比如，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，并根据距离大小选取出距离最近的前 r 个数据组成一个结果列表. 但由于数据集合 S 的规模非常大，这种朴素的查询方法的查询时间太长、计算代价太大而无法实用. 但是，如果首

先从规模比较大的数据集合 S 上筛选出一个显著小的候选集合 S'，然后在集合 S' 上进行朴素的线性扫描选取出前 r 个近邻数据，这时的线性扫描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筛选出候选集合 S' 的过程. 候选集合 S' 大小与查询准确率有着密切关系，一般而言，集合 S' 越大查询准确率越高，但后续线性扫描阶段的时间就会越长；反之，集合 S' 越小则查询准确率会越低，但后续线性扫描阶段的时间就会越短.

近邻查询问题的关键就在于如何选取出一个近邻候选集合 S'. 为了快速地筛选出近邻候选集合，就需要将待检索数据库索引起来. 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

传统的树结构索引方法有很多，比如 R 树、KD 树、Ball 树等. 下面介绍 FLANN[1] 近似近邻搜索算法库中用到的两种树结构索引方法.

经典的 KD 树会将原始的数据空间按数据的每个维度划分成一棵二叉树. 它在低维的空间上检索效率非常高，但随着维度不断增加，KD 树的检索效率会迅速降低并退化为线性扫描. 因此，许多改进 KD 树的工作涌现出来. 其中，Silpa-Anan 等人[2] 提出了一种随机化 KD 树的改进方法. 原始的 KD 树方法在空间划分时会选取数据方差最大的维度，在该维度上将空间一分为二，而随机化 KD 树在数据空间划分的时候，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的前几个维度中随机地选取一个维度进行空间划分.

此外，FLANN 中的层次 K-Means 树[3] 是在数据需要划分时，使用 K-Means 聚类的方法将数据划分成 K 份. 依照这一方法，在每一层中都使用 K-Means 聚类，当数据量少于 K 个时，就可以直接将这 K 个节点作为叶子节点. 这样，层次 K-Means 树就可以看作是一棵 K 叉树. 利用层次 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合. 但是在层次 K-Means 树的建树过程中，在每个非叶子节点都要执行一次 K-Means 聚类，这种算法不管在时间效率上还是空间效率上代价都是非常大的.

2.2 基于哈希的索引

传统树结构索引方法最大的不足就是存储空间占用过大，随着维度的不断增长，空间代价成倍增长. 因此，我们需要对原始数据通过哈希进行编码压缩以节省空间. 现有哈希方法主要分为数据无关哈希和数据驱动哈希. 数据无关哈希方法以局部敏感哈希(Locality Sensitive Hashing，LSH)[4] 为代表，其变种之一是随机投影法，该方法在不考虑数据分布的情况下将原始空间中的数据随机投影到超平面获取相应编码. 数据驱动哈希方法主要通过判别数据结构及分布信息来自动学习哈希函数，代表性的方法主要有谱哈希(Spectral Hashing，SH)[5]、迭代量化(Iterative Quantization，ITQ)[6]、乘积量化(Product Quantization，PQ)[7]、笛卡尔 K 均值(Cartesian K-Means)[8] 以及组合量化(Composite Quantization)[9] 等.

局部敏感哈希(LSH)的基本思想是保持相似性的空间转换，对于原始空间中相似的两个数据点，经过相同的哈希函数映射后，这两个数据点在映射后的空间中依然是相似的；反之，如果两个点在原始空间中不相似，那么映射后的两个点也是不相似的. 局部敏感哈希算法首先将原始数据嵌入到汉明空间，然后在汉明空间中选取多个位置的值进行组合，作为哈希映射. 最后，将上述经过哈希映射得到的序列进一步通过哈希函数转化成一个实数. 这样，就实现了原始向量到哈希桶的转换. 由于局部敏感特性的存在，原始空间中越相似的两个数据点，经过哈希之后，越有可能出现在同一个哈希桶中.

谱哈希(SH)的基本思想与局部敏感哈希类似，但对数据的分布特点进行建模，在保证原始空间的向量相似性条件下，它将整个编码过程转化为一个图分割的过程. 它首先对原始空间的高维数据进行谱分析，通过松弛约束条件转化为求解一个拉普拉斯矩阵的特征值分解问题. 谱哈希的求解过程为：首先根据数据对的相似度构造一个 K 近邻图，该图上每个顶点代表一个数据点，该顶点仅与其最相似的 K 个数据点有边相连，边的权重由数据点之间的相似度确定；其次由 K 近邻图的邻接矩阵得到拉普拉斯矩阵，计算拉普拉斯矩阵的特征值和特征向量(图分割算法)；最后选取前若干个最小特征值对应的特征向量，通过对其进行二值化得到哈希编码.

迭代量化(ITQ)首先通过主成分分析(PCA)对原始数据空间进行降维，将原始的 p 维数据 $\boldsymbol{x}\in\mathbb{R}^p$ 降成 d 维向量 $\boldsymbol{v}\in\mathbb{R}^d$. 然后，ITQ 对降维数据集 $\{\boldsymbol{v}\}$

迭代如下两个步骤直到收敛：(1) 通过正交变换矩阵 $\boldsymbol{R}$ 将任一向量 $\boldsymbol{v}$ 旋转得到 $\boldsymbol{Rv}$；(2) 将 $\boldsymbol{Rv}$ 通过符号函数 sgn 进行二值化，得到对应的二进制编码 $\boldsymbol{b}=\mathrm{sgn}(\boldsymbol{Rv})\in\{-1,1\}^d$. ITQ 的学习目标是使整个数据集的量化误差最小，即要求旋转后的向量 $\boldsymbol{Rv}$ 与编码后的哈希码 $\boldsymbol{b}$ 之间的均方误差 $\|\boldsymbol{b}-\boldsymbol{Rv}\|^2$ 最小.

3　背景知识

3.1　向量量化

向量量化(Vector Quantization)[10]就是对原始向量进行量化压缩，维度为 p 的原始向量 $\boldsymbol{x}\in\mathbb{R}^p$. 通过量化函数 q 被映射为 $q(\boldsymbol{x})\in C=\{\boldsymbol{c}_i\}$，其中集合 C 被称为码本，C 中的每个元素 $\boldsymbol{c}_i$ 被称为码字，映射 $q(\boldsymbol{x})$ 就是将向量 $\boldsymbol{x}$ 用码本 C 中的某个码字来表示. 在向量量化中，对一个包含 n 个 p 维数据点的数据集 $\mathcal{D}=\{\boldsymbol{x}_j\}_{j=1}^n$，K-means 算法会将这 n 个数据点聚成 k 类簇，同时用聚类中心来代表每一个类簇的数据. 记矩阵 $\boldsymbol{C}\in\mathbb{R}^{p\times k}$ 的所有列向量由 k 个聚类中心构成，每一列都是一个聚类中心，即 $\boldsymbol{C}=[\boldsymbol{c}_1,\boldsymbol{c}_2,\cdots,\boldsymbol{c}_k]$. 向量量化模型简单有效，使用最朴素的枚举方法就可以将数据点映射到相应聚类中心. 这种映射过程将每条原始数据的大小压缩到 $\log_2 k$ bit，所需要消耗的存储空间会随着 k 的增长而呈现出对数级增长.

3.2　乘积量化

乘积量化(Product Quantization，PQ)[7]是比向量量化更有效的一种量化方法. 假设需要量化压缩 p 维的向量到 64 bit，如果采用向量量化方法，则需要有 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用的空间来看，都是不可行的. 为了解决上述聚类中心(即码字)数量膨胀问题，在乘积量化算法中，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 p 维的向量切成 m 个长度为 p/m 的子向量；在每个子空间里，分别对其中的子向量集合进行 K-Means 聚类，聚类中心数量为 h. 这样就可以用聚类中心编号 $1\sim h$ 对子向量进行编码，m 个子向量的编码串接在一起就构成了原始向量的哈希编码. 这样，原始空间的 p 维向量就可以压缩为 $m\log_2 h$ bit 的哈希编码，从而大大节省了存储空间. 乘积量化(PQ)学习方法的目标函数形式化如下：

$$\ell_{\mathrm{PQ}}=\min\sum_{i=1}^{n}\|\boldsymbol{x}_i-\boldsymbol{C}\boldsymbol{b}_i\|_2^2$$

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

其中 $\boldsymbol{x}_i$ 是 p 维原始向量；$\boldsymbol{b}_i^j\in\{0,1\}^h$ 是 $\boldsymbol{x}_i$ 在第 j 个子空间中聚类后所属的聚类中心编号(即在码本中字编号)，每条数据在每个子空间中仅能属于一个聚类中心，因此 $\|\boldsymbol{b}_i^j\|=1, j\in\{1,\cdots,m\}$. 在编码过程中，整体的码本 C 就可以用多个子空间中码本的笛卡尔积的形式表示，$C=C^1\times C^2\times\cdots\times C^m$. 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，所以码本大小就是 $k=h^m$. 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. 整个算法的空间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为 $O(mhp)$.

从表 1 中乘积量化算法和向量量化算法的空间占用对比可知，当 $m=1$ 时，乘积量化就退化成普通的 K-Means 向量量化了. 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，选择合适的 m 和 h 的参数值是非常重要的. 文献[6]中指出，为了能够用一个字节表示 $b_i^j\in\{0,1\}^h$，通常取 $h=256$；而对于 128 维的向量数据，在采用 64 bit 进行哈希编码时，$m=8$ 是比较合适的取值.

表 1　向量量化与乘积量化的复杂度对比

	聚类中心	编码长度	空间占用
向量量化	k	$\log_2 k$	$O(kp)$
乘积量化	h^m	$m\log_2 h$	$O(mhp)$

3.3　Spark 与弹性分布式数据集

Spark 是一个实现了 MapReduce 编程范式的通用的大数据分布式计算框架，最初由 UC Berkeley AMP Lab 开发完成. Spark 继承了 MapReduce 编程简单的优点，并增加了对分布式内存计算的支持. MapReduce 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中间数据，从而提高计算效率. 在 Spark 数据处理过程中，数据来源不仅可以是本地文件系统或者 HDFS 上各种格式的数据，还可以是 HBase、Cassandra 等数据库中的数据. 在 Spark 内核的基础上，还集成了 Spark Streaming、Spark SQL、MLlib、GraphX 等数据处理的组件，形

成一栈式的生态系统.

图 1 是 Spark 集群系统架构图. 驱动程序(Driver)会和集群的管理器(Cluster Manager)相连接,驱动管理器为集群其他节点分配资源. 在分配完毕以后,驱动程序会将应用程序发送到各个节点的执行进程(Executor). 之后驱动程序会调配任务给各个执行进程执行任务.

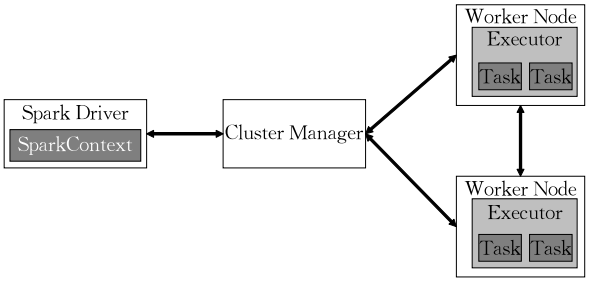

图 1 Spark 集群系统架构图

弹性分布式数据集(Resilient Distributed Datasets,RDD)[11] 是 Spark 中的分布式内存的抽象. 相比于 MapReduce 的计算过程,RDD 可以被缓存在内存中,每一次的计算产生的结果都可以保留在内存中,从而避免了大量的磁盘读写操作,大大节省了计算时间. 在 Spark 程序中,RDD 的创建是通过静态类 SparkContext 来实现,主要包含有两种创建来源:一是从指定的文件系统(或指定的数据库)读取数据来创建;二是从内存数据集合直接生成. 不同于 MapReduce 中仅有 map 和 reduce 两种操作,RDD 还支持多种丰富的常用操作,主要分为转换操作、控制操作和行为操作 3 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. 控制操作主要是将 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 存储到外部文件系统或数据库的操作. Spark 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).

4 分布式哈希学习及其 Spark 实现

4.1 算法整体设计

算法的总体流程设计如图 2 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应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,将编码后的数据存储起来. 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. 在近似近邻查询过程中,主要分为两步:首先是通过索引找出候选集合;然后在候选集合上进行重新排序.

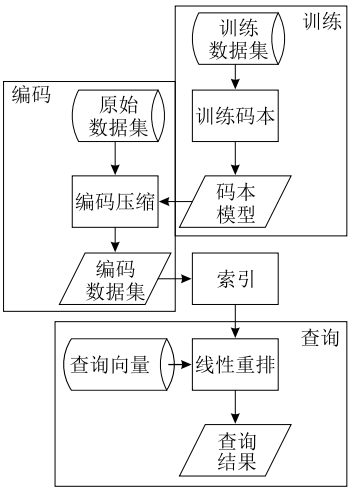

图 2 分布式哈希学习算法整体流程图

4.2 数据结构设计

在 Spark 上,分布式程序的编写必须依赖分布式数据结构 RDD,RDD 分布式地存储在不同节点上,这样才能使得程序分布式地执行. 因此,在编写 Spark 程序过程中如何合理设计程序的 RDD 数据结构非常重要. 在 Spark MLlib 库中提供了一种自带的 RDD 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,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 等.

图 3 是一个示例的 BlockMatrix 的划分方式,图中的 8×10 的矩阵被按行划分成 4 个子空间,每个子空间上有 10 个子向量,具体实验过程中的参数大小与此不同. 那么我们需要一个 4×10 的 BlockMatrix 来存储,图中深色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用 BlockMatrix 来存储.

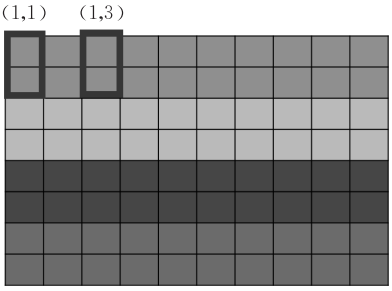

图 3　BlockMatrix 的划分方式

4.3　训练码本

首先，我们将乘积量化模型的目标函数进行分布式表示，把式(1)改写成弗罗贝尼乌斯范数(Frobenius norm)的形式：

$$\ell_{\mathrm{PQ}}=\min\left\|\boldsymbol{X}-\begin{bmatrix}\boldsymbol{C}^1\boldsymbol{B}^1\\ \vdots\\ \boldsymbol{C}^m\boldsymbol{B}^m\end{bmatrix}\right\|_F^2 \tag{2}$$

其中 $\boldsymbol{X}=[\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_n]$，$\boldsymbol{B}=[\boldsymbol{b}_1,\boldsymbol{b}_2,\cdots,\boldsymbol{b}_n]$. 下面证明如何从式(1)推导到式(2).

证明.　由弗罗贝尼乌斯范数的定义可知

$$\|\boldsymbol{A}\|_F^2=\sum_i\sum_j\|\boldsymbol{a}_{ij}\|^2=\sum_i\|\boldsymbol{a}\|_{i2}^2,$$

而 $\left\|\boldsymbol{X}-\begin{bmatrix}\boldsymbol{C}^1\boldsymbol{B}^1\\ \vdots\\ \boldsymbol{C}^m\boldsymbol{B}^m\end{bmatrix}\right\|_F^2=\left\|[\boldsymbol{x}_1,\cdots,\boldsymbol{x}_n]-\begin{bmatrix}\boldsymbol{C}^1[\boldsymbol{b}_1^1,\boldsymbol{b}_2^1,\cdots,\boldsymbol{b}_n^1]\\ \vdots\\ \boldsymbol{C}^m[\boldsymbol{b}_1^m,\boldsymbol{b}_2^m,\cdots,\boldsymbol{b}_n^m]\end{bmatrix}\right\|_F^2$

$$=\left\|\begin{matrix}\boldsymbol{x}_1^1-\boldsymbol{C}^1\boldsymbol{b}_1^1,\boldsymbol{x}_2^1-\boldsymbol{C}^1\boldsymbol{b}_2^1,\cdots,\boldsymbol{x}_n^1-\boldsymbol{C}^1\boldsymbol{b}_n^1\\ \vdots\\ \boldsymbol{x}_1^m-\boldsymbol{C}^m\boldsymbol{b}_1^m,\boldsymbol{x}_2^m-\boldsymbol{C}^m\boldsymbol{b}_2^m,\cdots,\boldsymbol{x}_n^m-\boldsymbol{C}^m\boldsymbol{b}_n^m\end{matrix}\right\|_F^2,$$

故 $\left\|\boldsymbol{X}-\begin{bmatrix}\boldsymbol{C}^1\boldsymbol{B}^1\\ \vdots\\ \boldsymbol{C}^m\boldsymbol{B}^m\end{bmatrix}\right\|_F^2=\sum_{i=1}^n\left\|\boldsymbol{x}_i-\begin{bmatrix}\boldsymbol{C}^1\boldsymbol{b}_i^1\\ \vdots\\ \boldsymbol{C}^m\boldsymbol{b}_i^m\end{bmatrix}\right\|_2^2.$

因此式(1)可以改写为式(2).　　证毕.

在分布式的系统中，数据是分布式地存储在拥有 S 个节点的计算集群上. 假设第 t 个节点上存储的 n_t 个数据，原来的数据矩阵 $\boldsymbol{X}$ 就可以被划分成 S 个小的矩阵进行分布式存储，即 $\boldsymbol{X}=[\boldsymbol{X}_1,\boldsymbol{X}_2,\cdots,\boldsymbol{X}_S]$，其中 $\boldsymbol{X}_i\in\mathbb{R}^{p\times n_t}$. 同样我们最终所需要编码矩阵 $\boldsymbol{B}$ 也可以用 $\boldsymbol{B}=[\boldsymbol{B}_1,\boldsymbol{B}_2,\cdots,\boldsymbol{B}_S]$ 来表示. 根据数据分布式这一特点，由式(2)，我们可以写出如下形式的分布式乘积量化的目标函数：

$$\ell_{\mathrm{PQ}}=\min\sum_{t=1}^S\left\|\boldsymbol{X}_t-\begin{bmatrix}\boldsymbol{C}^1\boldsymbol{B}_t^1\\ \vdots\\ \boldsymbol{C}^m\boldsymbol{B}_t^m\end{bmatrix}\right\|_F^2=\min\sum_{t=1}^S\left\|\begin{matrix}\boldsymbol{X}_t^1-\boldsymbol{C}^1\boldsymbol{B}_t^1\\ \vdots\\ \boldsymbol{X}_t^m-\boldsymbol{C}^m\boldsymbol{B}_t^m\end{matrix}\right\|_F^2 \tag{3}$$

从式(3)中，我们可以看出，在每一个子空间中需要求解的等式形式都是相同的. 以第一个子空间为例，对于 $\sum_{t=1}^S\min\|\boldsymbol{X}_t^1-\boldsymbol{C}^1\boldsymbol{B}_t^1\|_F^2$，不难看出要求解出 $\boldsymbol{C}^1$ 和 $\boldsymbol{B}_t^1$，只需要对 $\boldsymbol{X}_t^1$ 进行分布式 K-Means 聚类就可以得到结果.

在具体训练码本的过程中，如前文中所介绍的一样，首先将训练集中的数据划分到 m 个子空间. 然后在每个子空间中，对所有的子向量数据进行 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. 图 4 表示了对训练数据集 $\boldsymbol{X}$ 进行划分为 m 个子空间，然后在子空间中分别聚类得到码本模型 $\boldsymbol{C}$.

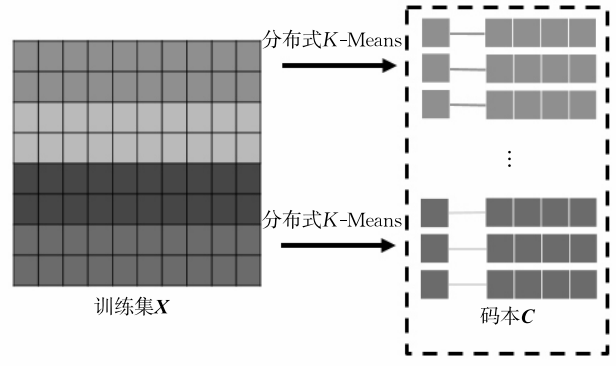

图 4　码本训练过程

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. 一般而言，模型确定以后，训练集的规模不宜过大也不能太小. 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更好地量化原始数据空间，更准确地对原始数据集进行编码. 在后面的实验过程中，我们采用对原始数据集随机采样的方法来构建训练集. 码本的训练算法描述如算法 1 所示.

算法 1.　训练码本.

输入：训练集 $\boldsymbol{X}$ 矩阵，子空间聚类数量 h，聚类算法最大迭代次数 $maxIter$

输出：码本模型数组 *model*

1. uniformly split $\boldsymbol{X}$ by rows into m subspaces
2. FOREACH subspace $i=1:m$ in parallel DO
3. $model[i] \leftarrow kmeans\_train(\boldsymbol{X}[i], h, maxIter)$
4. END FOREACH
5. RETURN *model*

4.4 编码压缩

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 $\boldsymbol{Z}$ 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，也就是用训练好的 K-Means 模型进行预测，可以计算出每个子向量的所属聚类中心，从而可以使用对应的聚类中心序号对该子向量进行编码. 这样，整个数据集中的向量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，存储空间成倍减少. 图 5 中编码压缩过程的示意图，将码本模型 $\boldsymbol{C}$ 广播到原始数据集 $\boldsymbol{Z}$ 的每个子空间，在子空间中对应编码压缩形成编码后的矩阵 $\boldsymbol{B}$. 具体算法如算法 2 所示.

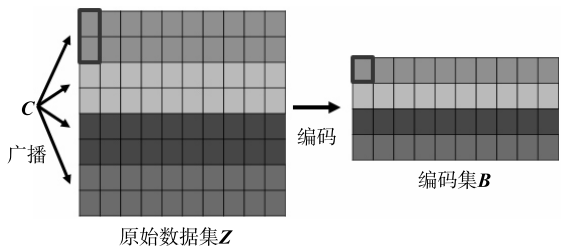

图 5 编码压缩过程

算法 2. 编码压缩.

输入：码本模型数组 *model*，原始数据集 $\boldsymbol{Z}$ 矩阵

输出：编码后的矩阵 $\boldsymbol{B}$

1. uniformly split $\boldsymbol{Z}$ by rows into m subspaces
2. broadcast *model* to all nodes
3. $n \leftarrow$ num of columns of $\boldsymbol{Z}$
4. FOREACH subspace $i=1:m$, column $j=1:n$ in parallel DO
5. $\boldsymbol{B}[i,j] \leftarrow model[i].predict(\boldsymbol{Z}[i,j])$
6. END FOREACH
7. RETURN $\boldsymbol{B}$

4.5 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 $\boldsymbol{q}$，计算 $\boldsymbol{q}$ 和任意的数据库中向量 $\boldsymbol{x}_i$ 之间的距离时，使用非对称距离度量（Asymmetric distance computation）[7] 方式进行距离计算. 哈希方法的出发点就是避免直接计算 $\boldsymbol{q}$ 和 $\boldsymbol{x}_i$ 之间的欧式距离 $D(\boldsymbol{q},\boldsymbol{x}_i)$，因此如果 $\boldsymbol{q}$ 与数据库中的每一个向量都计算一次距离，查询的时间代价太大. 在近似近邻查询过程中，使用 $\boldsymbol{q}$ 和 $\boldsymbol{x}_i$ 之间的非对称距离 $AD(\boldsymbol{q},\boldsymbol{x}_i)$ 近似表示原始距离 $D(\boldsymbol{q},\boldsymbol{x}_i)$，其中 $AD(\boldsymbol{q},\boldsymbol{x}_i)=D(\boldsymbol{q},\boldsymbol{x}_i')$，$\boldsymbol{x}_i'$ 是 $\boldsymbol{x}_i$ 所属的聚类中心. $D(\boldsymbol{q},\boldsymbol{x}_i')$ 可以先计算出来存储在查找表中，在之后查找比较时，用查找表中的非对称距离近似表示原始距离.

在算法具体流程上，我们首先计算出 $\boldsymbol{q}$ 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 $\boldsymbol{q}$ 和数据库中每个数据之间的距离. 在每个子空间中，因为子向量与聚类中心之间距离已经存储在查找表中，可以查找出每个数据与向量 $\boldsymbol{q}$ 之间的近似距离. 最后，将不同子空间中同一向量距离求和. 这样就得到了查询向量 $\boldsymbol{q}$ 和数据库中每个向量之间的距离. 通过线性扫描一遍距离数组，我们就可以快速获取出前 k 个近邻向量. 图 6 是近似近邻查询阶段的流程图. 算法如算法 3 所示.

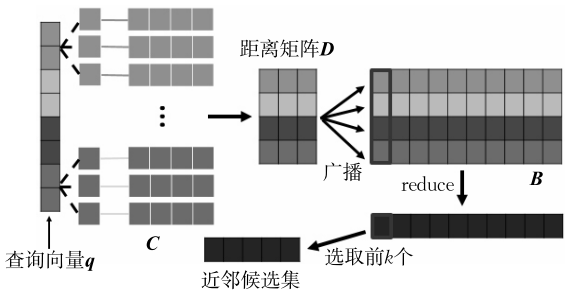

图 6 近似近邻查询过程

算法 3. 近邻查询.

输入：码本模型数组 *model*，编码数据集 $\boldsymbol{B}$，查询向量 $\boldsymbol{q}$

输出：近邻集合 *result* 数组

1. FOR $i=1 \rightarrow m$ DO
2. FOR $j=1 \rightarrow model[i].numOfCenters$
3. $\boldsymbol{D}[i,j] \leftarrow computeDist(models[i].center[j], \boldsymbol{q})$
4. END FOR
5. END FOR
6. $n \leftarrow$ num of columns of $\boldsymbol{B}$
7. broadcast dist to all nodes
8. FOREACH subspace $i=1:m$, column $j=1:n$ in parallel DO
9. $dist[i] \leftarrow dist[i] + \boldsymbol{D}[i, \boldsymbol{B}[i,j]]$
10. END FOREACH
11. $result \leftarrow$ getTopK($dist$)
12. RETURN *result*

上述的查询过程中，非对称距离与原始距离之间的误差可以用 $|AD(\boldsymbol{q},\boldsymbol{x}_i)-D(\boldsymbol{q},\boldsymbol{x}_i)|$ 表示.

定理 1.　$\boldsymbol{x}_i-\boldsymbol{C}\boldsymbol{b}_{i2}$是非对称距离 $AD(\boldsymbol{q},\boldsymbol{x}_i)$与原始距离 $D(\boldsymbol{q},\boldsymbol{x}_i)$之间误差的上界.

证明.
$$\begin{aligned}&|AD(\boldsymbol{q},\boldsymbol{x}_i)-D(\boldsymbol{q},\boldsymbol{x}_i)|\\&=|D(q,x_i')-D(q,x_i)|\\&\leq D(\boldsymbol{x}_i',\boldsymbol{x}_i)\\&=\|\boldsymbol{x}_i-\boldsymbol{C}\boldsymbol{b}_i\|_2.\end{aligned}$$
证毕.

由定理 1 可知，非对称距离与原始距离之间的误差可以被式(1)中目标函数所约束，最小化目标函数的同时，也最小化了非对称距离与原始距离之间的误差，从而保证检索的准确性.

4.6　系统优化

集群系统上的分布式程序一般都会需要考虑节点间数据通信的问题，Spark 中对 RDD 已经进行了较完备的封装，让开发者可以不用直接进行底层的数据管理(通信、容错)，只需通过操作上层的一些接口即可，但是数据通信的问题在 Spark 中仍是一个不能忽视的问题，只有了解底层的通信机制才能利用 API 编写出高效的程序. 程序优化的原则是尽可能地减少数据的通信，特别是数据量比较大的数据通信. 以上是对系统的第 1 点优化.

第 2 点优化是利用 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 的时候不必多次重复计算，从而节省时间效率，这一机制特别适合在迭代算法中使用. 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.

第 3 点优化是对 Spark 程序参数的调优. Spark 程序需要设置一些如 executor 数量、每个 executor 的核数、executor 内存大小、RDD 分区数等在内的系列参数. 一般而言，executor 的数量乘以每个 executor的核数应该与集群的总核数相同；executor 的内存大小应该与集群中每台机器的内存大小除以每台机器分配的 executor 数相同；RDD 的分区数会直接影响程序并行度，合适的分区数才能保证程序充分并行执行，这一参数设置应该与 executor 数量相关.

5　实验结果与分析

在本节中，我们通过在 4 个数据集上进行大量对比试验，以此来观察我们在 Spark 上所实现的分布式乘积量化方法 SparkPQ 的性能和可扩展性. 一方面，我们通过和单机版的 PQ、ITQ、SH、LSH 这 4 个算法在不同数据集上的对比来观测算法的性能好坏；另一方面，我们通过在 Spark 集群上，改变集群的节点数量、Spark 参数配置、数据集大小、算法参数等来观测的 SparkPQ 的可扩展性和对参数变化的敏感程度.

5.1　实验环境和数据集

实验环境. 本次实验中 Spark 部分的实验是在由 4 台机器构成的 Spark on YARN 集群系统上完成的. 其中每台机器配置相同，如表 2 所示.

表 2　集群计算节点的配置信息

内容	配置信息
操作系统	Red Hat Enterprise Linux Server release 7.0
处理器	Intel(R) Xeon(R) CPU E5-2609 0@2.40 GHz
CPU 核数	8
内存	60 GB
Scala 版本	2.10.3
Spark 版本	1.4.0

非分布式的 PQ、ITQ、SH、LSH 算法的单机实验是在一台机器上完成，配置信息如表 3 所示.

表 3　单机的配置信息

内容	配置信息
操作系统	Windows 8
处理器	Intel(R) Core(TM) i3 CPU M 380 @ 2.53 GHz
CPU 核数	4
内存	6 GB
MATLAB 版本	R2012b

数据集. 实验过程中，共使用了 4 个数据集：SIFT1M、GIST1M、CIFAR-10 和 SIFT100M，这 4 个数据集近年来被广泛用于衡量近似近邻查询方法的有效性[7-9,12-13].

SIFT1M 中的每条数据为 128 维的 SIFT 特征向量，其原始图片来源于图片分享网站 Flickr.

GIST1M 中的每条数据是 960 维的 GIST 特征向量.

CIFAR-10 数据集是一个 8×10^8 千万极小图像数据集[14]的子集，被广泛应用于计算机视觉领域测试目标识别[15]、图像分类[16]等任务的效果，由 60000 张 32×32 大小的彩色图像组成，分属于 10 个类别，每个类别包含 6000 张图片. 在本次实验中，我们对 CIFAR-10 数据集中的每张图片提取出 320 维的 GIST 特征，并从中随机选出 1000 张图片作为测试数据集，其余 59000 张作为训练数据集和待检索数据集.

SIFT100M 是 SIFT1B[7]数据集的一个子集，包含数据集中的 1/10 数据，该数据集中的待检索集由

10^8 个 SIFT 特征向量构成，用来验证我们提出的方法在大规模数据集上的可扩展性.

整个实验部分使用的所有数据集规模如表 4 所示.

表 4 数据集的规模

数据集	维度	训练集	检索集	测试集
SIFT1M	128	10^5	10^6	10^4
GIST1M	960	5×10^5	10^6	10^3
CIFAR10	320	5.9×10^4	5.9×10^4	10^3
SIFT100M	128	10^7	10^8	10^4

表 4 中维度是表示向量的维数，训练集、检索集和测试集表示的是其中向量的数量大小. 训练集是用于训练哈希模型的数据集，检索集包含所有可以被编码检索到的数据库数据，测试集是指用于查询的所有数据.

5.2 实验设置

实验部分包括算法性能对比实验和算法可扩展性实验，下文将详述各部分实验的具体设置.

在性能对比实验部分，我们参照文献[6]，使用召回率(Recall)、查准率(Precision)、平均准确率(Mean Average Precision)等评价指标. 通过与 PQ、ITQ、SH、LSH 等方法在不同数据集上的召回率、查准率和平均准确率的对比来观测算法的性能.

下面列举出实验中对比的所有算法：

(1) SparkPQ. Spark 上分布式乘积量化方法.

(2) PQ. 单机版的乘积量化方法.

(3) ITQ. 第 2.2 节中介绍的迭代量化方法.

(4) SH. 第 2.2 节中介绍的谱哈希方法.

(5) LSH. 第 2.2 节中介绍的局部敏感哈希方法.

在验证算法可扩展性的实验中，我们通过在 Spark 集群中改变集群的节点数量、Spark 中分配的 executor 数量大小、训练集数据大小和子空间数量 m 等参数来验证算法的可扩展性和对参数的敏感程度. 集群节点数量为实验中使用的机器数量，在实验中默认总节点数量为 4. Executor 数量为 Spark 分配给程序的任务执行进程个数，实验中默认设置为 32. 训练集大小是指哈希学习过程中训练集数据的多少. 子空间数量 m 和子空间聚类中心数量 h 会影响整体数据压缩的编码长度，默认 $m=8$，$h=256$，使用 $8\log256=64$ bit 长度编码原始向量数据.

5.3 算法性能对比实验

5.3.1 SIFT1M 数据集实验结果

首先我们在 SIFT1M 数据集上进行对比实验，通过比较 Spark 集群上实现的分布式乘积量化方法和单机 MATLAB 实现的乘积量化方法的查询召回率，以验证算法的正确性. 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，所有参数如 5.2 节所述，均采用默认设置，编码长度为 64 bit.

在本次实验中，SIFT1M 数据中有 10^4 个查询向量，对于任意一个查询向量，通过整个近似近邻查询计算，我们在数据库向量中找到前 100 个近邻向量，并分别取检出数量 R 为 1、2、5、10、20、50、100，计算出前 R 个向量中出现最近邻向量的次数 s，那么衡量标准召回率 $=s/10^4$.

表 5 中是 Spark 集群上分布式乘积量化方法和 MATLAB 上实现的乘积量化方法的召回率对比. 从表中可以看出，Spark 上分布式算法的召回率与 MATLAB 单机版本的相差不大，从而证明了算法正确有效.

表 5 Spark 与 MATLAB 上实验召回率

	召回率($R=1$)	召回率($R=2$)	召回率($R=5$)	召回率($R=10$)	召回率($R=20$)	召回率($R=50$)	召回率($R=100$)
Spark	0.224	0.319	0.466	0.593	0.713	0.854	0.921
MATLAB	0.226	0.327	0.476	0.607	0.719	0.853	0.920

此外，我们还将分布式乘积量化算法与 ITQ、SH、LSH 等经典算法的性能对比，图 7 是改变查询检出相似向量的数量 R，不同算法的实验召回率的变化曲线图. 从图 7 中可以看出，我们提出的分布式乘积量化方法的召回率要比其它几种经典方法的召回率都要高，证明了我们方法的有效性. 除此之外我们发现检出数量越大，召回率越高，这与召回率的定义相符.

5.3.2 CIFAR-10 数据集实验结果

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 SIFT1M 实验中完全相同. 对于 CIFAR-10 中 60000 张图片，随机产生 1000 张图片作为测试数据，其他 59000 张图片用于训练和检索，对于图片，我们提取出 320 维的 GIST 特征，使用欧氏距离先计算出 1000 个测试数据的与剩余待检索数据的近邻关系. 使用此近邻关系作为衡量基准，对于每个算法，我们分别计算检索的查准率、召回率以及平均准确率，以此对比不同算法的性能好坏.

在图 8 中，我们可以看到不同算法的查准率变

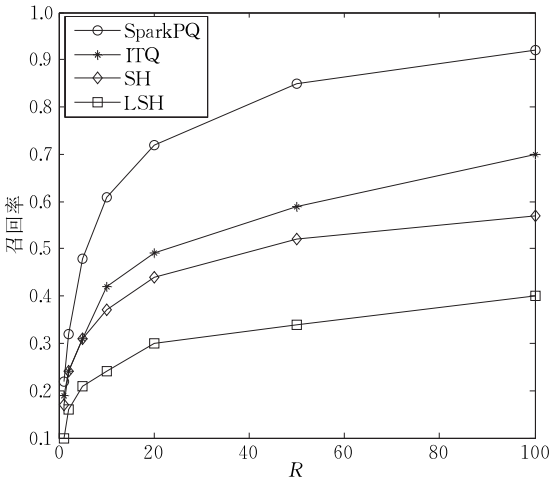

图 7　不同算法的召回率随 R 的变化对比

化，检出数量 R 从 0 增加到 1000，随着检出数量增大，所有算法的查准率都在不断降低. 然而不管检出数量如何变化，分布式乘积量化算法的查准率均比其它算法高.

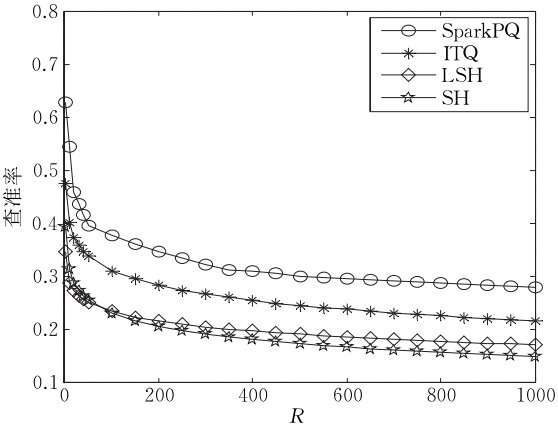

图 8　不同算法 CIFAR-10 上的查准率对比

图 9 则反映的是检出数量 R 从 0 变化到 1000，不同算法的召回率变化情况. 同样，分布式乘积量化方法的召回率高于其他所有算法.

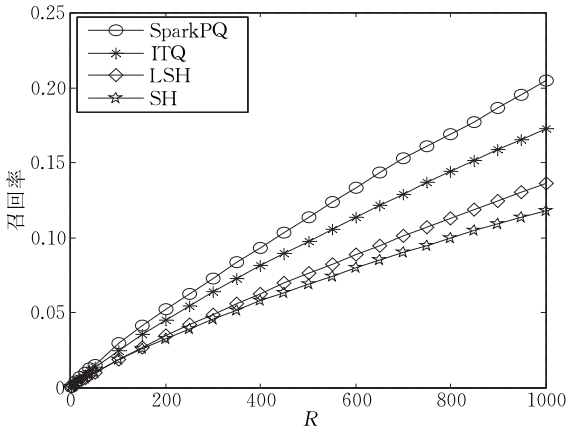

图 9　不同算法 CIFAR-10 上的召回率对比

图 10 展现了不同算法在不同编码长度下的平均准确率变化情况. 在编码长度分别为 8、16、32、64、128 情况下，我们可以看到，对于所有算法，编码比特数越大，算法的平均准确率越高，即平均准确率随着比特数的增大而升高. 然而，不同算法的增长幅度是不同的. 我们可以比较 SparkPQ 和 ITQ，ITQ 在从 8 bit 到 16 bit 长度情况下，平均准确率增长幅度较大，之后逐渐趋缓；SparkPQ 在 8 bit 长度的平均准确率较高，最初的增长幅度也不大，但是在 64 bit 变化到 128 bit 长度时，平均准确率增长的幅度较大. 由此可以看出，SparkPQ 在编码长度较长时，增加编码长度的时仍能保持较好的平均准确率增幅.

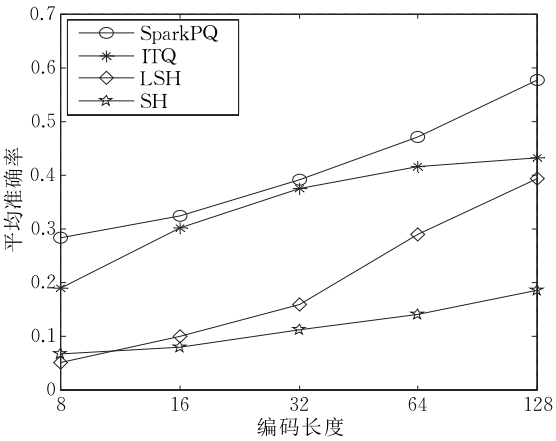

图 10　不同算法 CIFAR-10 上的平均准确率对比

图 11 是我们从实验中选取出的 3 个相似图像检索的示例，对于每个查询请求的图片，选取出前 36 个最相似的图片. 同样，在这个实验中对图片提取 320 维的 GIST 特征，采用特征向量的欧氏距离近邻关系作为参考标准. 图中深色框标识出的图片与查询图片并不近邻. 从图中可以看出，对于这 3 个查询请求，其中分布式乘积量化的返回结果是最精确的，迭代量化方法次之，局部敏感哈希和谱哈希的准确度较差.

5.4　算法可扩展性实验

5.4.1　SIFT1M 数据集实验结果

本实验通过比较不同节点数量的 Spark 集群上的乘积量化的近似近邻查询方法实验的召回率和时间消耗的对比，用于验证算法的可扩展性. 本次实验分布式乘积量化算法部分采用默认配置，通过改变集群中节点数量，观察不同检出数量的召回率变化与计算时间随集群节点数量的变化. 表 6 和表 7 分别显示的是在不同节点数量的 Spark 集群系统上实验召回率对比以及所用时间对比.

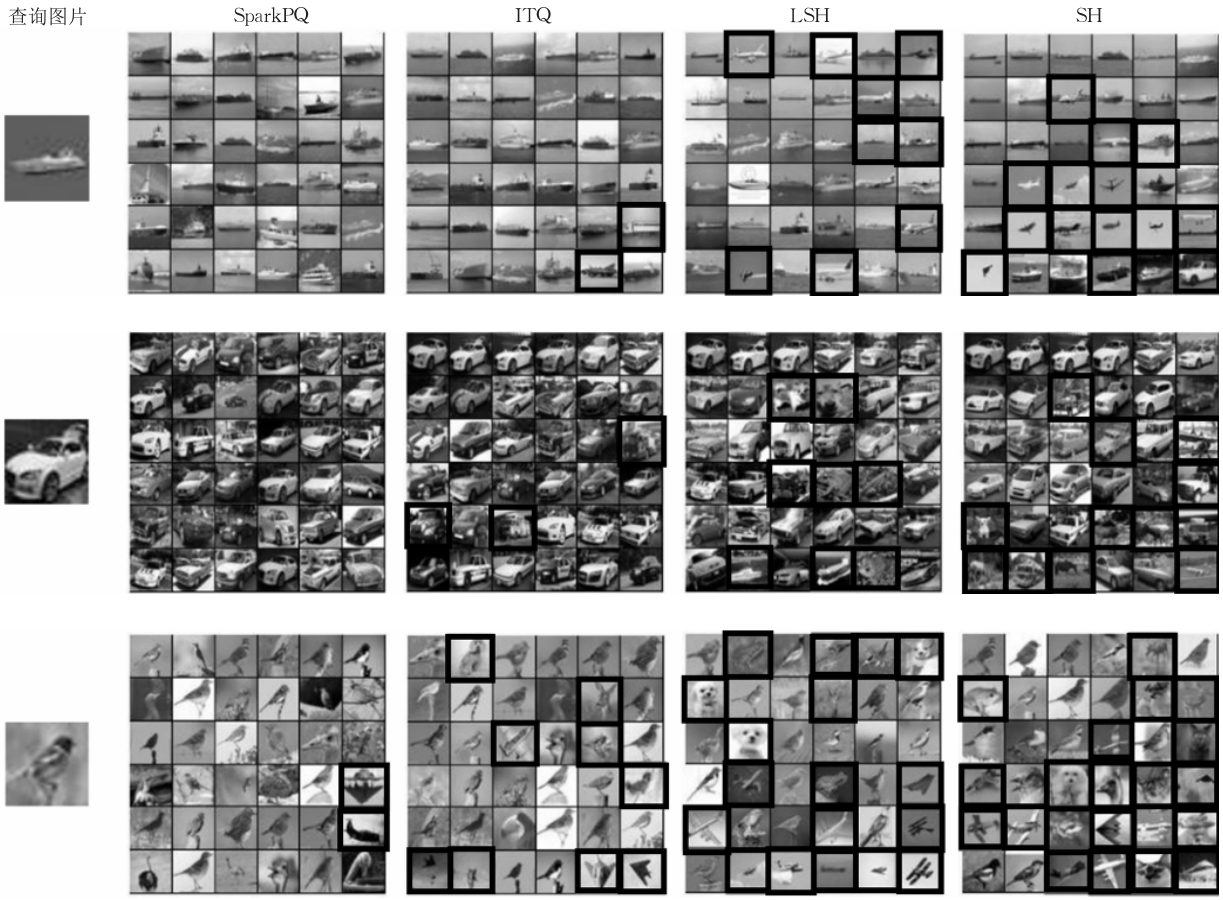

图 11 CIFAR-10 数据集上 64 位编码长度图像检索示例

表 6 不同节点数量 Spark 集群系统上的实验召回率

节点数	召回率(R=1)	召回率(R=2)	召回率(R=5)	召回率(R=10)	召回率(R=20)	召回率(R=50)	召回率(R=100)
1	0.227	0.330	0.481	0.604	0.719	0.847	0.919
2	0.231	0.329	0.479	0.602	0.721	0.855	0.923
3	0.220	0.322	0.474	0.600	0.719	0.849	0.921
4	0.224	0.319	0.466	0.593	0.713	0.854	0.921

表 7 不同节点数量 Spark 集群系统上的实验时间

节点数	训练时间/s	编码时间/s	单次查询/s
1	875.9	12.35	3.01
2	585.2	5.69	1.51
3	556.9	5.20	1.45
4	541.3	5.08	1.49

仅从表 6 来看，不同节点数量的 Spark 集群系统上实验的召回率相差不大. 在保证召回率变化不大的情况下，从表 7 中我们可以看出在训练、编码和查询 3 个阶段的时间消耗上，随着节点数量的增多，训练时间迅速下降并逐渐趋缓，并没有保持理想的线性下降趋势，这是由于节点增多时，Spark 集群的额外计算开销增多，包括网络通信时间等. 整体而言，时间消耗随着节点增加而下降说明该系统的可扩展性良好.

此外，我们还在 SIFT1M 数据集上进行训练集大小与训练时间关系的实验. 通过改变不同训练数据集的大小，从而对比实验的召回率和时间消耗情况，用以验证算法可扩展性. 在实验参数设置方面，我们采用与上一实验中的 4 节点 Spark 集群实验相同的配置. 同样我们采用召回率作为查询结果好坏的衡量标准. 不同之处，我们分别取训练集大小为 0.1 M、0.2 M、0.5 M、1 M 进行实验，观察实验召回率和时间消耗的变化.

从表 8 来看，改变实验中的训练集合大小，随着训练集增大，召回率呈现出增长的趋势，但是并不明显. 从表 9 中我们可以看出在随着训练集的增大，训练时间迅速地增大，随后逐渐趋于缓慢增长. 编码时间和单次查询之所以时间变化不大，是因为并不受训练数据集大小变化的影响. 因此，实验表明系统的可扩展性较好.

表 8　Spark 集群系统上不同训练集大小实验召回率

训练集大小/M	召回率(R=1)	召回率(R=2)	召回率(R=5)	召回率(R=10)	召回率(R=20)	召回率(R=50)	召回率(R=100)
0.1	0.224	0.319	0.466	0.593	0.713	0.854	0.921
0.2	0.223	0.321	0.469	0.596	0.719	0.856	0.923
0.5	0.222	0.323	0.473	0.600	0.726	0.859	0.924
1.0	0.222	0.326	0.472	0.603	0.727	0.855	0.927

表 9　Spark 集群系统上不同训练集大小实验时间

训练集/M	训练时间/s	编码时间/s	单次查询/s
0.1	541.3	5.08	1.49
0.2	1235.7	5.22	1.48
0.5	2601.5	5.18	1.47
1.0	5616.3	4.93	1.47

5.4.2　GIST1M 数据集实验结果

对 GIST1M 数据集的实验都在 Spark 集群系统上完成，Spark 集群系统的配置与 SIFT1M 的实验相同，在 yarn-client 模式运行程序，executor 的数量为 32. 在本次实验中，我们分别采用 32 bit、64 bit、96 bit、128 bit、256 bit 这些不同编码长度进行实验，观察实验中压缩编码长度对召回率的影响，如图 12 所示. 在同等检出数量下，编码长度越长，召回率越高.

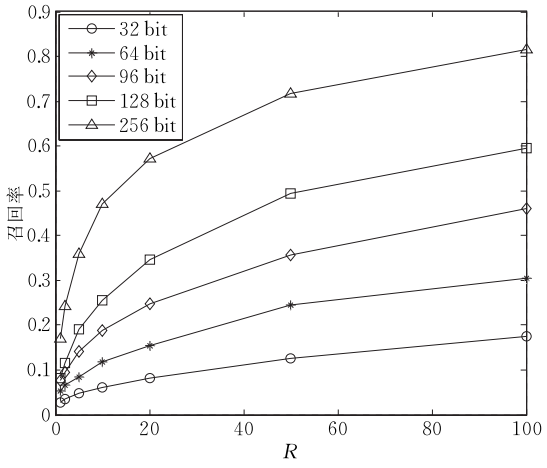

图 12　GIST1M 上不同编码长度下的召回率

图 13 中是记录了在 GIST1M 上不同编码长度的算法运行时间. 从图 13 可以看出，随着编码长度的增大，算法的训练时间、编码时间、单次查询时间都是在不断增长的，但是增长的幅度较小. 时间不断增长是因为编码长度 $m\log h$ 增大，子空间 m 数量就会增大，从而需要在更多子空间中训练码本、压缩编码和检索近邻，增加了计算时间.

5.4.3　CIFAR-10 数据集实验结果

与上一个实验配置相同，我们在 CIFAR-10 数据集上的分别对不同长度的编码进行比较，采用准确率(Precision)来衡量检出效果.

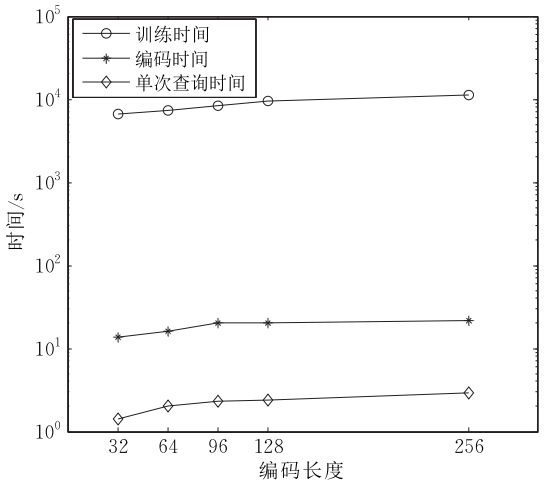

图 13　GIST1M 上不同编码长度的算法运行时间

在图 14 中，R 是检出数据的数量，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.

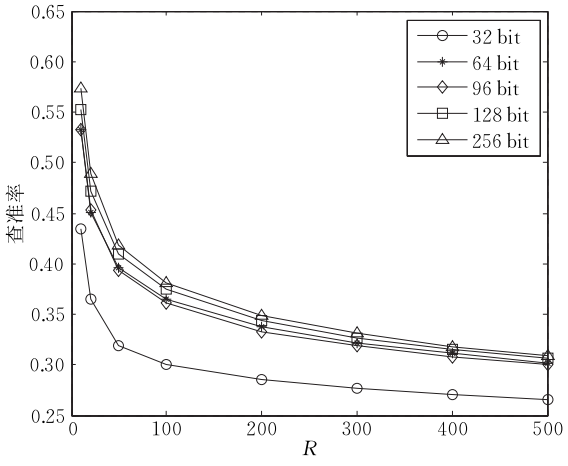

图 14　CIFAR-10 上不同编码长度下的查准率

图 15 是不同编码长度下的算法运行时间变化情况，从图 15 中可以得出与图 13 中实验相似的结论，编码长度分别为 32 bit、64 bit、96 bit、128 bit、256 bit，算法的训练时间、编码时间、单次查询时间都是在不断增长的，运行时间增长的速度与编码长度的增长幅度有关.

图 16 中是在压缩编码长度分别为 32 bit、64 bit、

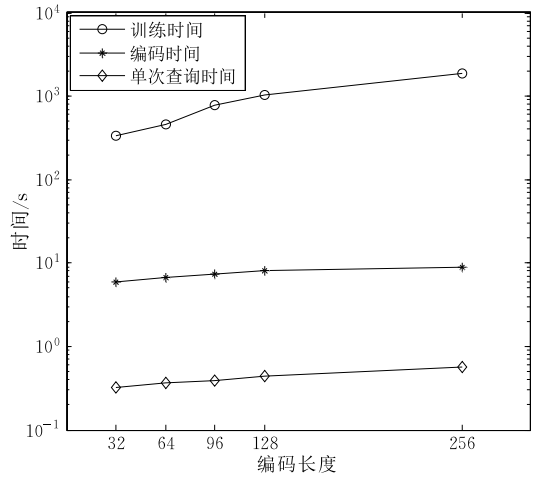

图 15 CIFAR-10 上不同编码长度的算法运行时间

96 bit、128 bit、256 bit 情况下，训练时间随 executor 数量的变化. 从图 16 中可以看出，大体上 executor 的数量越多，训练时间就越短，但是也并非 executor 数量越多越好，与 Spark 集群系统的总处理器核数有关. 当 executor 数量超过集群的总核数时，训练时间反而会增加. 实验中集群的总核数为 32，因此当 executor 数量为 48 时，训练时间略大于 executor 数量为 32 的情况.

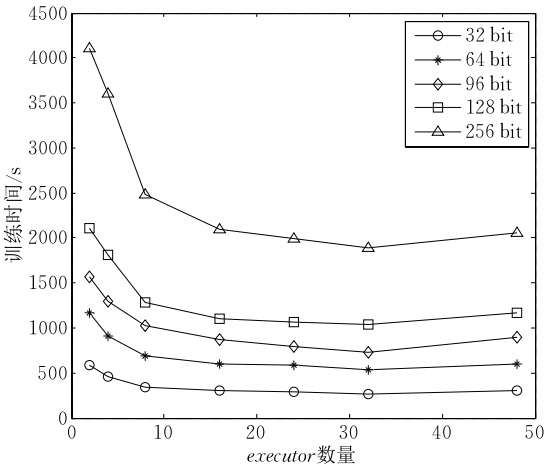

图 16 CIFAR-10 上训练时间随 executor 数量的变化

5.4.4 SIFT100M 数据集实验结果

使用 SIFT100M 数据集进行实验主要是为了在一个更大规模数据集上验证算法的可扩展性. 由于输入数据规模变大，因此 Spark 集群相关参数需要有所调整. 集群节点数仍为 4，executor 的数目为 15，每个 executor 分配内存大小为 12 GB，其它参数不变. 在此我们比较了不同编码长度下召回率的变化以及计算时间的变化.

图 17 是 SIFT100M 上不同编码长度下召回率的变化图. 由于数据集较大，当编码长度为 16 时，召回率几乎接近于 0，随着编码长度增大，召回率也不断提高. 图 18 是不同编码长度的算法计算时间变化图，随着编码长度增大，计算时间呈现出缓慢增长. 以上观察与小数据集时的情况相似，从而证明了在较大规模的数据集上，我们的方法依然具有良好的可扩展性.

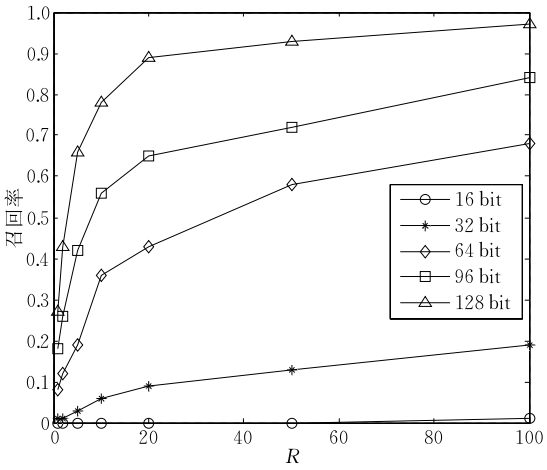

图 17 SIFT100M 上不同编码长度下的召回率

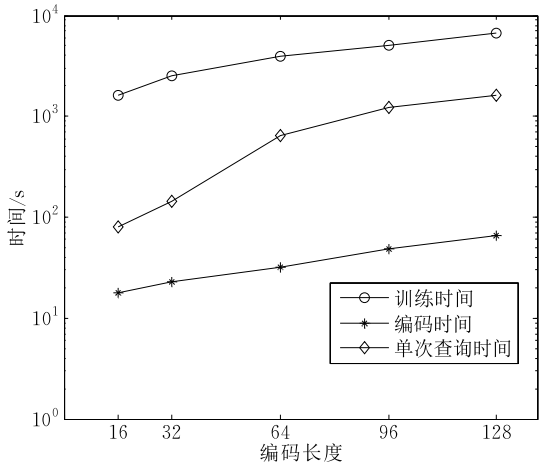

图 18 SIFT100M 上不同编码长度的算法运行时间

6 结束语

本文针对大规模高维数据的近似近邻查询问题，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分布式哈希方法的近似近邻查询系统. 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 Spark 集群系统并行计算的方式可以大大提高查询的效率.

本文已经实现了一套基于 Spark 的近似近邻查询系统，采用乘积量化的方法进行分布式哈希编码，但并没有在 Spark 上实现一套高效的索引方法. 在接下来的研究中，我们准备实现一套高效索引方法[17]以提高查询的性能.

致　谢　感谢清华大学信息科学与技术国家实验室大数据科学与技术专项、国家自然科学基金项目、中国博士后基金特别资助项目的支持. 感谢《计算机学报》编辑部和审稿专家的宝贵意见！

参　考　文　献

[1] Muja M, Lowe D G. Fast approximate nearest neighbors with automatic algorithm configuration//Proceedings of the 4th International Conference on Computer Vision Theory and Applications. Lisboa, Portugal, 2009: 331-340

[2] Silpa-Anan C, Hartley R. Optimised KD-trees for fast image descriptor matching//Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Anchorage, USA, 2008: 1-8

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

[4] Gionis A, Indyk P, Motwani R. Similarity search in high dimensions via hashing//Proceedings of the 25th International Conference on Very Large Data Bases. Edinburgh, UK, 1999: 518-529

[5] Weiss Y, Torralba A, Fergus R. Spectral hashing//Proceedings of the Conference on Neural Information Processing Systems. Vancouver, Canada, 2008: 1753-1760

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

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

[8] Norouzi M, Fleet D J. Cartesian *k*-means//Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Porland, USA, 2013: 3017-3024

[9] Zhang T, Du C, Wang J. Composite quantization for approximate nearest neighbor search//Proceedings of the 32nd International Conference on Machine Learning. Beijing, China, 2014: 838-846

[10] Gray R M, Neuhoff D L. Quantization. IEEE Transactions on Information Theory, 1998, 44(6): 2325-2383

[11] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing//Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. San Jose, USA, 2012: 15-28

[12] Jégou H, Douze M, Schmid C. Searching with quantization: Approximate nearest neighbor search using short codes and distance estimators. Villers-lès-Nancy, France: INRIA, Technical Report: inria-00410767, 2009

[13] Paulevé L, Jégou H, Amsaleg L. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. Pattern Recognition Letters, 2010, 31(11): 1348-1358

[14] Torralba A, Fergus R, Freeman W T. 80 million tiny images: A large dataset for nonparametric object and scene recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2008, 30(11): 1958-1970

[15] Bo L, Lai K, Ren X, et al. Object recognition with hierarchical kernel descriptors//Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Colorado Springs, USA, 2011: 1729-1736

[16] Wan L, Zeiler M, Zhang S, et al. Regularization of neural networks using dropconnect//Proceedings of the International Conference on Machine Learning. Atlanta, USA, 2013: 1058-1066

[17] Babenko A, Lempitsky V S. The inverted multi-index//Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Providence, USA, 2012: 3069-3076

WEN Qing-Fu, born in 1993, M. S. candidate. His research interests include machine learning and information retrieval.

WANG Jian-Min, born in 1968, Ph. D., professor. His research interests include big data and knowledge engineering.

ZHU Han, born in 1992, M. S. candidate. His research interests include machine learning and information retrieval.

CAO Yue, born in 1992, Ph. D. candidate. His research interests include machine learning and information retrieval.

LONG Ming-Sheng, born in 1985, Ph. D., assistant professor. His research interests include machine learning and big data.

Background

Approximate Nearest Neighbor (ANN) search is a fundamental technique for large-scale unstructured data management, which is widely applied in many research fields such as Data Mining and Multimedia Retrieval. With the current exponential growth of big data, there has been an increasing interest in studying efficient and accurate search in large-scale high-dimensional database.

Hashing is an important approach to approximate nearest neighbor search that can significantly compress data to reduce storage and computation costs. Many hashing methods have been proposed for fast search, including the seminal Locality Sensitive Hashing, Spectral Hashing, and Iterative Quantization.

Recently, Spark has become a popular platform which is applied to efficient processing of large scale datasets. Spark is an open-source distributed computing platform that supports high scalability, high availability and fault tolerance. Previous works on large scale datasets show that Spark is suitable for solving problems that depend on iterative algorithms, while machine learning fits well into this iterative programming paradigm.

In this paper, we proposed SparkPQ, a novel distributed learning to hash method based on Product Quantization (PQ), which is implemented in the Spark distributed computing framework. Based on the SparkPQ method, an ANN search system is built, which can not only reduce the storage and computation cost substantially, but also can speed up the search efficiency with guaranteed search accuracy. Besides, we conducted extensive empirical experiments on large-scale image retrieval datasets, which validates the effectiveness and scalability of the proposed SparkPQ method and system.

This research was supported by the Tsinghua National Laboratory (TNList) Special Fund for Big Data Science and Technology under Grant "Development and Operation Platform for Domain-Oriented Big Data Systems", the National Natural Science Foundation of China (Grant Nos. 61325008 and 61502265), and the China Postdoctoral Science Foundation (Grant No. 2015T80088).