LIFT:一种用于高维数据的索引结构

薛向阳,罗航哉,吴立德
(复旦大学计算机系,上海 200433)

摘 要: 本文提出一种新的高维空间中点数据的索引方法,其基本原理是用格矢量量化(Lattice vector quantization)均匀划分数据空间、用倒排文件(Inverted File)存储格点、用 Trie 树实现倒排文件的组织和存储、用 Trie 并行搜索算法实现倒排文件的快速访问. 和传统索引方法相比,新方法具有许多优点,例如它能以较低的复杂度建立索引结构、支持非常高维的数据索引、充分利用高维空间中点分布的稀疏性等. 实验结果表明,在较高维数时,LIFT 性能优于传统索引方法.

关键词: 索引结构; 相似性检索; 矢量量化
中图分类号: TP311 **文献标识码**: A **文章编号**: 0372-2112 (2001) 02-0192-04

LIFT:An Index Structure for High Dimensional Data

XUE Xiang-yang, LUO Hang-zai, WU Li-de
(Department of Computer Science, Fudan University, Shanghai 200433, China)

Abstract: A new method for indexing large amounts of points in high-dimensional space is proposed. The basic principle is as follows: uniformly partition the data space by Lattice vector quantization, store the lattice points by Inverted File, organize the inverted file by Trie tree, and fast access the inverted file by Trie parallel search algorithm. We called this index structure LIFT. Compared with the traditional index methods, the LIFT can build the index structure with low complexity, support very high dimensionality, and take advantage of sparsity of data points in high-dimensional space, etc. The experiments show that for high-dimensional data, the LIFT outperforms the well-known R-tree.

Key words: index structure; similarity retrieval; vector quantization

1 引言

在基于内容的检索和数据挖掘等领域,经常需要在对象集中查找与某个给定对象"相似"的对象,这样的查找过程叫"相似性检索". 通常,对象间的相似性不是由对象本身直接定义的,而是从对象中提取出"特征",然后在对象的"特征"上定义相似性. 大多数情况下,特征用高维空间的点(或特征矢量)来描述,特征矢量之间的接近程度反映了对象内容的相似度大小,因此基于内容的检索就简化为空间中点的快速搜寻问题.

用 $R^Q(Q_o)$ 表示查询结果集,Q_o 表示查询对象,本文定义二类相似查询方法. 一类叫"范围查询(range query)":

$$R^Q_{\text{Range}}(Q_o) = \{x \mid x \in O, D(f(x), f(Q_o)) \le T\} \quad (1)$$

其中 T 为范围门限. 另一类叫做"k-近邻查询",k 为给定的正整数. $k=1$ 时的最近邻查询定义为:

$$R^Q_{1\text{-Nearest}}(Q_o) = \{x_0 \mid x_0 \in O, \forall y \in O: D(f(y), f(Q_o)) \ge D(f(x_o), f(Q_o))\} \quad (2)$$

设 $R^Q_{(k-1)\text{-Nearest}}$ 是"$(k-1)$-最邻近查询"的结果,则"k-最邻近查询"定义为:

$$R^Q_{k\text{-Nearest}}(Q_o) = R^Q_{(k-1)\text{-Nearest}}(Q_o) \cup \{x_0 \mid x_0 \in O', \forall y \in O': D(f(y), f(Q_0)) \ge D(f(x_0), f(Q_0))\} \quad (3)$$

其中,$O' = O - R^Q_{(k-1)\text{-Nearest}}(Q_0)$,$O$ 表示对象集合,f 表示对象到特征矢量的映射. 本文只讨论索引结构,不考虑映射 f.

实现相似查询的方法很多,最基本方法是"顺序扫描算法(SSA)":顺序检查 O 中每个对象,如果它符合相似性查询要求,则将该对象加入 $R^Q(Q_o)$ 中. SSA 的 CPU 和 I/O 复杂度通常都很大,为了降低查询开销,多年来,人们陆续提出多种支持快速相似查询的索引结构,例如 R-树[1]、k-d-树[2]、K-D-B-树[3]、TV-树[4]、SS-树[5]和 SR-树[6]等. 无论采用哪种索引结构和实现哪种相似性查询,其实现方法是基本一致的:(1) 迅速找到 $R^Q(Q_o)$ 的一个超集 $R^Q_{\text{super}}(Q_o)$;(2) 在 $R^Q_{\text{super}}(Q_o)$ 上做 SSA 得到 $R^Q(Q_o)$. 因此,人们总是期望做到:(1) $R^Q_{\text{super}}(Q_o)$ 包含 $R^Q(Q_o)$;(2) $R^Q_{\text{super}}(Q_o)$ 足够小;(3) 快速得到 $R^Q_{\text{super}}(Q_o)$.

然而,在实际应用中 $R^Q_{\text{super}}(Q_o)$ 经常大于 $R^Q(Q_o)$,特别是当维数增加时,$R^Q_{\text{super}}(Q_o)$ 迅速接近 O,这一现象叫做"维数灾

收稿日期:1999-11-01;修回日期:2000-03-06
基金项目:自然科学基金重点项目(No. 69935010);国家 863-317-01-07-99;自然科学基金项目(No. 60003017)

难”,前面提及的索引结构几乎都存在这一问题. 尽管 VA-File[9]能够一直保证 $R^Q_{\text{super}}(Q_o)$ 非常接近 $R^Q(Q_o)$,但是其 CPU 代价很大. 本文则利用格矢量量化[14]、倒排文件和 Trie 数据结构[15],构造出一种新的索引结构,力图在一定程度上克服维数灾难,同时确保较小的 CPU 代价.

2　LIFT 的基本原理

2.1　用格矢量量化实现空间划分

格矢量量化就是将数据空间划分成同样大小的 Vonoroi 胞腔. 对格 A_2 来说,胞腔为正六边形;对格 Z_2 来说,胞腔为正方形;对高维来说,胞腔通常是凸多面体. 每个 Vonoroi 胞腔的质心就是相应格的格点,如果任一矢量 x 落在某个 Vonoroi 胞腔 V_i 中,设其质心为 l_i,那么 x 就被量化为 l_i,即 $l_i = LVQ(x)$,这里 $LVQ(\cdot)$ 表示格矢量量化. 由于格的良好代数性质,格矢量量化算法的复杂度通常很低[10,11].

假定数据库中有 N 个对象,每个对象对应一个唯一序号 id,从每一个对象上提取出一个特征矢量,那么这 N 个特征矢量将组成数据集合 $DB = \{x_i, i = 1,2,\cdots,N\}$. 经过格矢量量化后,多个特征矢量可能会落在同一个胞腔中,量化为同一个格点. 设量化后格点集合为 $DB_{vq} = \{l_k = LVQ(x_i) \mid x_i \in DB\}$,假设 DB_{vq} 的大小为 M,显然 $M \leq N$.

2.2　用倒排文件实现格点的存储

2.2.1　倒排文件

LIFT 用倒排文件来组织和存储格点 DB_{vq},它将格点看作倒排文件的键值,每个格点可能会对应若干个序号,见表 1.

表 1　倒排文件存储格点集合和图像序号

键值	图像序号
l_1	$id_{1,1}, id_{1,2}, \cdots$
l_2	$id_{2,1}, id_{2,2}, \cdots$
…	…
l_M	$id_{M,1}, id_{M,2}, \cdots$

快速访问倒排文件的方法很多,例如二分查找法、Hash 和 Trie 等,本文采用 Trie 方法,因为它能够充分利用特征矢量在高维空间中分布的稀疏特性,实现快速搜索.

2.2.2　Trie 树的构造

为了快速访问倒排文件表,本文用 Trie 表示所有键值的集合 $DB_{vq} = \{l_i = (l_{i,j}, j = 1,2,\cdots,n), i = 1,2,\cdots,M\}$,则 Trie 的字母表为 $\sum = \{\lambda_L, (\lambda_L + 1), \cdots, 0, \cdots, (\lambda_H - 1), \lambda_H\}$,其中 $\lambda_L = \min_{i,j}(l_{i,j})$,$\lambda_H = \max_{i,j}(l_{i,j})$,因此有 $DB_{vq} \subseteq \sum^n$. 事实上,基于 DB_{vq} 构造 Trie 树是一个递归过程. 不妨假设当前正处理第 d 维 $(1 \leq d \leq n)$,再设 $S \subseteq DB_{vq}$,于是为 S 建立一个 Trie 节点的过程如下:

(1) 令 $|S|$ 表示集合 S 中的元素个数. 如果 $|S| > 1$,将 S 按第 d 维分成 $(\lambda_H - \lambda_L + 1)$ 个子集:$S_{\lambda_L}, S_{\lambda_L+1}, \cdots, S_{\lambda_H-1}, S_{\lambda_H}$,其中 $S_t = \{l_i \mid l_{i,d} = t\}$. 再分别基于 $S_{\lambda_L}, S_{\lambda_L+1}, \cdots, S_{\lambda_H-1}, S_{\lambda_H}$ 按照第 $(d+1)$ 维来递归构造 $(\lambda_H - \lambda_L + 1)$ 个 Trie 节点. 最后,建立一个中间节点将上述 $(\lambda_H - \lambda_L + 1)$ 个 Trie 子节点连接到中间节点的相应位置.

(2) 如果 $S = \Phi$,则建立一个空节点.

(3) 如果 $|S| = 1$,则建立一个终结节点. 设 $l \in S$ 为 S 的唯一元素,$\{id_1, id_2, \cdots\}$ 为 l 所对应的图像序号集,将 $(l_{d+1}, l_{d+2}, \cdots, l_n)$ 和序号 $\{id_1, id_2, \cdots\}$ 存储到此终结节点中.

用 $S = DB_{vq}$ 和 $d = 1$ 调用上面过程,就可以递归建立一个完整的 Trie 树,称之为 $\text{Trie}_{\text{lift}}$. 至此,就完成了用 Trie 重新组织和存储倒排文件表.

2.3　实现范围查询的基本原理

2.3.1　范围查询的基本步骤

假设查询对象为 Q_O,查询矢量 $y = f(Q_O)$. 现在,基于公式(1)进行范围查询,范围门限为 T,于是:

(1) 用同样的矢量量化器对查询矢量 y 进行量化,得到:$l_O = LVQ(y)$.

(2) 根据查询范围门限 T,首先计算出 $\delta = \lceil T/T_0 \rceil$.

(3) 决定需要搜索的超矩形窗口 W_{Rect}:

$$l_{\text{Low}} = (\max((l_{0,1} - \delta), \lambda_L), \max((l_{0,2} - \delta), \lambda_L), \cdots, \max((l_{0,n} - \delta), \lambda_L))$$

$$l_{\text{High}} = (\min((l_{0,1} + \delta), \lambda_H), \min((l_{0,2} + \delta), \lambda_H), \cdots, \min((l_{0,n} + \delta), \lambda_H))$$

其中,l_{Low} 和 l_{High} 是超矩形窗口 W_{Rect} 的两个对角顶点.

(4) 在倒排文件中,用 Trie 并行搜索算法(TPSA)搜索 W_{Rect} 中所有格点,得到 $R^Q_{\text{super}}(Q_o)$.

(5) 根据公式(1),对 $R^Q_{\text{super}}(Q_o)$ 进行顺序搜索,得到最终查询结果 $R^Q_{\text{Range}}(Q_o)$.

2.3.2　Trie 并行搜索算法

算法复杂度主要取决于对倒排文件表的查找速度. 查找倒排文件表最慢的方法是顺序扫描,它需要将格点 l_O 和倒排文件表中键值进行逐个比较. 事实上,随着维数增加,查询窗口 W_{Rect} 中格点数目迅速增加,它可能会远远大于 DB_{vq} 中格点数 M. 然而 $|W_{Rect} \cap DB_{vq}|$ 总是非常有限的,因为 W_{Rect} 中有效格点非常稀少,最多也不会超过 M. 有很多算法用于快速访问倒排文件:二分查找、Hash 查找和 Trie 树查找等. 本文采用 Trie 树方法的原因是:二分查找和 Hash 查找必须用 W_{Rect} 中所有格点去搜索倒排文件,因此总复杂度为 $|W_{Rect}|$ 次倒排文件查找,尽管 W_{Rect} 中大量格点并不存在于倒排文件之中.

为了快速地搜索有效格点集合 $(W_{Rect} \cap DB_{vq})$,得到 $R^Q_{\text{super}}(Q_o)$,我们提出 Trie 并行搜索算法(TPSA). TPSA 是一个递归过程. 用 d 表示当前正在处理的维,用 $\text{Root}_{\text{trie}}$ 表示当前子树的根,那么:

(1) 如果 $\text{Root}_{\text{trie}}$ 为 Null,则返回.

(2) 如果 $\text{Root}_{\text{trie}}$ 是终结节点,设 $l' = (l_{d+1}, l_{d+2}, \cdots, l_n)$ 为存储在 $\text{Root}_{\text{trie}}$ 中的剩余部分,对于 $i = d+1, d+2, \cdots, n$ 均有 $l_{\text{Low},i} \leq l_i \leq l_{\text{High},i}$,则该终结节点中所有图像序号对应的矢量都加入 $R^Q_{\text{super}}(Q_o)$.

(3) 如果 $\text{Root}_{\text{trie}}$ 是中间节点,则以 $\text{Root}_{\text{trie}}$ 的第 $l_{\text{Low},d}$ 个分支到第 $l_{\text{High},d}$ 个分支为根,令 $d \leftarrow d+1$,,调用本递归过程,继续查找.

令 $d = 1$,$R^Q_{\text{super}}(Q_o)$ 的初值为 NULL,$\text{Root}_{\text{trie}}$ 等于 $\text{Trie}_{\text{lift}}$ 的根,调用上述递归过程,得到 $R^Q_{\text{super}}(Q_o)$. 在实验中,发现 Trie 树中有大量空节点,为了进一步减少 TPSA 的复杂度,对原始 Trie 树又进行了改进,通过增加少量空节点指示信息(即指明空节点所处位置信息),从而删除所有空节点,得到压缩的

Trie 树. 然后,再对 TPSA 略加修改,就可以在压缩后 Trie 上进行快速搜索,从而提高计算速度.

3 实验研究

用计算机模拟方法比较 LIFT 和 *R*-树的索引性能,包括 I/O 代价和 CPU 复杂度两个指标. 其中,I/O 代价定义为 $RIOCost = |R^Q_{super}(Q_o)|/|R^Q(Q_o)|$,它直接反映一个索引结构在一次查询过程中所涉及到的对象,是衡量 I/O 代价的关键指标,理想情况下 $RIOCost = 1$;CPU 代价则是用 $Speed\text{-}up = t_{SSA}/t_{fast}$ 测量,其中 t_{SSA} 表示 SSA 算法实现一次查询所需要的时间,t_{fast} 表示采用索引结构后实现一次查询所需要的时间,注意测量 t_{SSA} 和 t_{fast} 时,所有相关数据都在主存中,即这些数据和辅助存储器无关,因此 *Speed-up* 主要反映了一次查询所花费的计算复杂度.

在实验中,用随机数发生器产生 $(N + L_Q)$ 个高斯矢量,矢量中每个分量是相互独立的,且都服从均值为 0、方差为 σ^2 的高斯分布;然后,产生 N_0 个矢量,它们均匀分布于空间 $[-A, +A]^n$ 中;对每个高斯矢量,从 N_0 个均匀分布的矢量中随机选取一个,两者相加;最终得到 $(N + L_Q)$ 个矢量,取其中 N 个矢量作为数据集 *DB*,剩余 L_Q 个矢量用作为查询矢量.

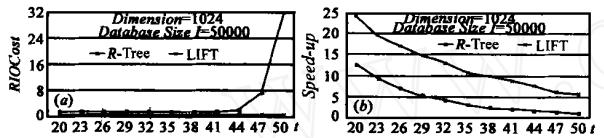

图 1 (*a*) 1024 维时的 I/O 代价; (*b*) 1024 维时的 CPU 代价

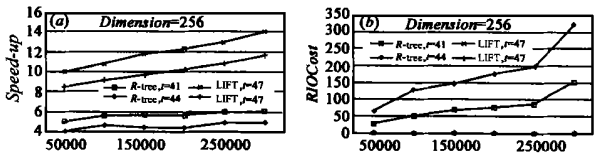

图 2 (*a*) 不同数据库规模时的 CPU 代价;
(*b*) 不同数据库规模时的 I/O 代价

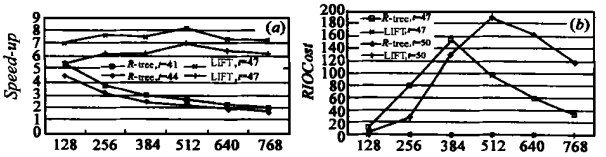

图 3 (*a*) 不同维数时的 CPU 代价; (*b*) 不同维数时的 I/O 代价

首先比较 LIFT 和 *R*-树的 I/O 代价. 图 1 (*a*) 和 (*b*) 分别给出了 *R*-树和 LIFT 的 I/O 代价和 CPU 性能 ($n = 1024$, $N = 50000$),可以看到:不管查询范围 T 如何变化,LIFT 的 *RIOCost* 非常接近于 1(即理想情形),而 *R*-树的 *RIOCost* 曲线随着 T 变大而迅速增加;LIFT 的计算复杂度明显比 *R*-树低. 图 2 (*a*) 和 (*b*) 给出了不同数据库规模时 LIFT 和 *R*-树的索引性能 ($n = 256$, $T = 41$、44),可以看出:(1) 当数据库规模 N 增加时,LIFT 的加速比 *Speed-up* 比 *R*-树增加快;(2) 范围门限 T 增加导致加速比降低;(3) 不管 T 和 N 多大,LIFT 的 *RIOCost* 始终接近于 1,而 *R*-树的 I/O 性能随着 N 增加而变坏. 图 3 (*a*) 和 (*b*) 给出不同维数时的索引性能:图 3 (*a*) 表明 LIFT 的加速比 *Speed-up* 几乎保持常数,而 *R*-树的加速比随维数增加而下降;图 3 (*b*) 表明 (1) LIFT 的 *RIOCost* 还是接近于 1;(2) *R*-树的 *RIOCost* 则表现出复杂的变化趋势,它首先快速增加,然后又迅速减少,"增加"恰好说明 *R*-树的 I/O 性能随维数增加而迅速恶化,而"减少"的原因是在于:随着维数增加,数据点在空间中分布密度急剧减少,从而导致查询窗口中数据点急剧减少,导致 *RIOCost* 在一定维数后迅速下降. 最后,图 4 给出了构造 LIFT 和 *R*-树这两种数据结构所需要花费的时间. 可以看出:随着维数增加,构造 *R*-树的复杂度急剧增加;随着数据库规模增加,构造 *R*-树的复杂度也增加很多;而构造 LIFT 的复杂度总是比构造 *R*-树要低.

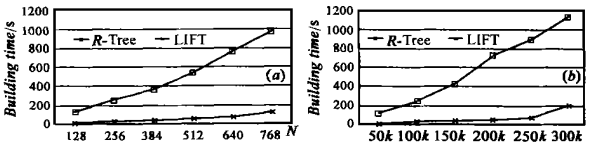

图 4 (*a*) 不同维数时构造索引结构的时间
(*b*) 不同数据库规模构造索引结构的时间

上述模拟结果表明:和 *R*-树相比,LIFT 具有较低 I/O 代价、较低 CPU 代价和较低构造代价,支持很高维数的索引,在一定程度上克服了维数灾难.

4 结论

本文主要讨论了高维点数据的范围查询问题,提出了一种新的索引结构,称之为 LIFT,它用格矢量量化对数据空间进行划分,用倒排文件组织格点,用 Trie 和压缩 Trie 重新组织倒排文件,用 TPSA 实现范围查询. 实验结果表明 LIFT 无论在 I/O 代价还是 CPU 代价方面都比 R-树要好. 将来研究工作集中在以下两个方面:用 LIFT 实现最近邻查询;将 LIFT 用于实际数据库之中.

参考文献:

[1] Guttman A. R-trees: A dynamic index structure for spatial searching [A]. Proc. ACM SIGMOD Int. Conf. On Management of Data [C], Boston, MA, 1984: 47 - 57.

[2] J L Bentley. Multidimensional binary search trees used for associative searching [J]. Communications of the ACM, 18, Sept. 1975: 509 - 517.

[3] Robinson J. T. The K-D-B-tree: A search structure for large multidimensional dynamic indexes [A]. Proc ACM SIGMOD Int. Conf. on Management of Data [C], 1981: 10 - 18.

[4] K.-I. Lin, H. V. Jagdish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data [J]. VLDB Journal, March 1994: 517 - 542.

[5] White D. A., Jain R. Similarity indexing with the SS-tree [A]. Proc. 12th Int. Conf on Data Engineering [C], New Orleans, LA, 1996.

[6] N. Katayama and S. Satoh. The SR-tree: An index structure for high dimensional nearest neighbor queries [J]. Int. Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson,

Arizon USA,1997:369 - 380.

[7] N. Beckmann and H. P. Kriegel and R. Schneider and B. Seeger. The R*-tree:an efficient and robust access method for points and rectangles [J]. Proceedings of the SIGMOD Conference, Atlantic City, NJ, June 1990:322 - 331.

[8] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: an index structure for high dimensional data [A]. Proceedings of the 22nd VLDB Conference [C], 1996:28 - 39.

[9] Roger Weber, Hans-J. Schek, Stephen Blott. A quantitative analysis and performance study for similarity search methods in high-dimensional spaces [A]. Proc. of the 24th VLDB Conference [C], New York, USA, 1998.

[10] J. Conway and N. J. A. Sloane. Sphere Packings, Lattice and Groups [M]. New York: Springer-Verlag. 1991.

[11] J. H. Conway and N. Sloane. A fast encoding method for lattice codes and quantizers [J]. IEEE Trans Information Theory, Nov. 1983, 29:820 - 824.

[12] M. Flickner, et al. Query by Image and Video Content: the QBIC System [M]. IEEE Computer, 1995.

[13] Pentland, R. W. Picard, and S. Sclaroff. Photobook: Tools for content-based manipulation of image databases [J]. Int. Proc. SPIE Storage and Retrieval for Image and Video Databases II, 1994, 2.

[14] Xiangyang Xue, Hangzai Luo, Lide Wu. Index Point Data Using Algebraic Lattic [J]. SPIE Electronic Imaging 2000: Storage and Retrieval for Media Databases, San Jose, USA: 3972:271 - 283.

[15] Xiangyang Xue, Hangzai Luo, Lide Wu. Index point data using lattice vector quantization and hash [A]. International Workshop on Very Low Bitrate Video Coding [C], VLBV '99, October 29-30, 1999, Kyoto Research Park, Kyoto, JAPAN.

作者简介:

薛向阳　1968 年生,现为复旦大学计算机系副教授,研究兴趣包括:基于内容的图像视频信息检索、高维数据索引结构和视频压缩编码算法.

罗航哉　1977 年生,现为复旦大学计算机系助教,在职研究生,研究兴趣包括:基于内容的图像视频信息检索和高维数据索引结构.

吴立德　1937 年生,现为复旦大学计算机系教授,博士生导师,研究兴趣包括:自然语言处理、计算机视觉和多媒体信息检索.