大数据分析

大作业系统设计报告

(2019 年度春季学期)

组　　员　1160300312 靳贺霖

组　　员　1160300314 朱明彦

学　　院　计算机学院

教　　师　杨东华、王金宝

计算机科学与技术学院

目录

分布式空间近似关键字查询系统

第 1 章　问题描述

1.1　数据

空间对象集合 $D=\{o_1,o_2,\ldots,o_n\}$，对于 D 中任意一个对象 $o_i=(loc_i,kw_{i,1},\ldots,kw_{i,m})$，即包含 $\mathbb{R}^d$ 维欧式空间中一个点 loc_i 和一组关键字 $kw_{i,1},\ldots,kw_{i,m}$，记为 $o_i.loc=loc_i$ 和 $o_i.kw=\{kw_{i,1},\ldots,kw_{i,m}\}$。**在本项目中主要关注 $d=2$ 的情况，$\mathbb{R}^2$ 对于现实中的应用有着很大的价值，但项目中提出的方法可以扩充到任意有限维欧式空间。**

1.2　范围查询

输入：　$Q=(Q_{rs},Q_{rt})$，其中 Q_{rs} 是一个空间范围（$\mathbb{R}^d$ 维欧式空间中的超立方体）；Q_{rt} 为关键字近似条件，$Q_{rt}=\{(kw_1,\theta_1),\ldots,(kw_K,\theta_K)\}$，其中 θ_i 为阈值。

输出：　$O=\{o|o\in D,o.loc\in Q.Q_s,\forall(kw_i,\theta_i)\in Q.Q_t,\exists o.kw_j,\mathrm{ED}(kw_j,kw_i)\le\theta_i\}$，其中 $\mathrm{ED}(kw_j,kw_i)$ 表示两个关键字 kw_j 和 kw_i 之间的编辑距离。

1.3　kNN 查询

输入：　$Q=(Q_s,Q_t,k)$，其中 $Q_s=loc$ 是 $\mathbb{R}^d$ 维欧式空间中一个点，即查询发出的位置；$Q_t=\{(kw_1,\theta_1),\ldots,(kw_K,\theta_K)\}$；$k$ 为表示最近邻居的数量。

输出：　对 $O_t=\{o|o\in D,\forall(kw_i,\theta_i)\in Q.Q_t,\exists o.kw_j,\mathrm{ED}(kw_j,kw_i)\le\theta_i\}$，根据 $|O_t|$ 的大小进行定义，

- 如果 $|O_t|\le k$，则 $O_{k\mathrm{NN}}=O_t$ 即为最终结果。
- 如果 $|O_t|>k$，$O_{k\mathrm{NN}}=\{o|o\in O_t,\forall o_i\in O_t-O,\mathrm{Dis}(loc,o_i)\ge\mathrm{Dis}(loc,o_j)$ 对 $\forall o_j\in O$ 成立 $\}$ 并且 $|O_{k\mathrm{NN}}|=k$。

1.4　Reverse kNN 查询

输入：　与1.3节输入相同，不再赘述。

输出：　$O_{\mathrm{R}k\mathrm{NN}}=\{o_{R_1},\ldots,o_{R_M}\}$，对于 $O_{\mathrm{R}k\mathrm{NN}}$ 中的任一元素 o_{R_i} 均有 $o_{k\mathrm{NN}}\in O_{R_i-k\mathrm{NN}}$ 且 $o_{R_i}\in D$，其中 $o_{k\mathrm{NN}}.loc=Q_s,o_{k\mathrm{NN}}.kw=Q_t$；$O_{R_i-k\mathrm{NN}}$ 是以 $(o_{R_i}.loc,o_{R_i}.kw,k)$ 为输入的 kNN 查询结果。

主要思路

- 存储部分，类似 Spark、HDFS 进行处理
- 索引和算法部分，两层索引结构，组织不同节点间的索引使用 `RT-CAN` [5]，在本地使用以 R 树为核心，结合 MHR-Tree [6] 进行范围查询，结合 Voronoi Diagrams [4] 进行 kNN 查询和 Reverse kNN 查询。

第 2 章 Background

2.1 R-Tree

R-Tree [1] 是如今处理空间查询最常用的索引，它将 $\mathbb{R}^d$ 中的数据划分到 d 维超立方体（Minimum Bounding Rectangle），并将每个超立方体存储在叶子节点。将每个数据划分到叶子后，再递归地寻找 MBR 能够覆盖若干个叶子，直到仅剩一个 MBR，即为 R-Tree 的根节点。

2.2 Voronoi 图

Voronoi Diagram [2] 是一种根据点之间特定的距离度量方式将空间划分成若干区域的划分方式。对于 $\mathbb{R}^d$ 中的一个集合 $D = \{p_1, p_2, \ldots, p_n\}$ 上的 Voronoi 图，将 $\mathbb{R}^d$ 划分为 n 个区域，每个区域中包含着距离 D 中某一个数据在 $\mathbb{R}^d$ 最近的所有点，其中距离 $\mathrm{Dis}(.,.)$ 的定义方式可以自行给定。

换而言之，如果给定 $q \in \mathbb{R}^d, p_i \in D$，如果 q 在 Voronoi 图中包含 p_i 的区域里，则

$$\forall j \neq i, p_j \in D, \mathrm{Dis}(p_j, q) \geq \mathrm{Dis}(p_i, q)$$

由上述性质以及在 [3] 中提到的性质，Voronoi 图在处理 kNN 查询和 Reverse kNN 查询有着很好的效果。**在本项目中，主要关注 $d = 2$ 并且使用欧式距离度量的情况**。

2.3 MHR-Tree

MHR-Tree [6] 本质上是在 R 树上增加关键字集合的信息（min-hash 签名），并使用基于 q-gram 集合的剪枝策略来处理近似关键字的条件。从根节点开始的查询，根据查询关键字 σ 和某一个内节点 u 的 q-gram 集合 $|g_\sigma \cap g_u|$ 的大小来进行剪枝。

- 当 $|g_\sigma \cap g_u| < |\sigma| - 1 - (\tau - 1) * q$ 时，无需访问 u，其中 τ 为编辑距离的阈值。
- 否则，需要访问内节点 u。

而根据 [6] 中可以通过 $s(g_\sigma)$ 和 $s(g_u)$ 来估计 $|g_\sigma \cap g_u|$，主要依赖于下式

$$|\widehat{g_\sigma \cap g_u}| = \hat{\rho}(g_\sigma, g_u) \times |\widehat{g_\sigma \cup g_u}|$$

2.4 CAN

CAN(Content Addressable Network) 是一种自组织性很强的 P2P 覆盖网络，一个 d 维的 CAN 的划分是将一个 d 维的坐标系划分到不同的节点上去。数据是通过一个 $hash$ 函数来将数据的主键划分为 d 维的坐标，从而划分数据到不同的 CAN 节点上的。对于一个 CAN 节点 N_i 来说，它会保存所有和 N_i 对应的空间的邻接空间的 ip 地址，从而形成一个 P2P 网络。因为 CAN 网络的自组织性很强，所以一个节点可以很自由地加入或者离开一个 CAN，就像 P2P 网络一样，只需要发送信息更改网络拓扑即可。

我们使用的 RT-CAN 索引的构建是基于一种 CAN 的变种，C^2 来实现的。C^2 除了有 CAN 的特征外，它还在划分坐标系的每个维度上添加了和弦邻居链接 (Chord-like neighbor link)。具体而言，在每个维度上距离为 $2^0, 2^1, \cdots$ 的节点上添加链接。因为我们要实现的是分布式空间近似关键字查询，所以我们选择中 2 维的 C^2 来实现的 RT-CAN 索引。

我们要处理的是空间近似关键字查询以及 kNN，RkNN 近似关键字查询，所以我们的数据的 key 值就可以看作是 2 维的位置坐标，这样很自然的就可以想到对 CAN 的划分也是在 2 维坐标中而且可以和位置信息对应起来，即 $hash$ 函数不对输入进行转换。

第 3 章 存储

3.1 数据划分

数据的划分我们使用的是一种类似于 KD 树划分的方法。我们知道 [7]，KD 树有一种划分的算法按照如下的方法进行：

1. 从方差最高的维度开始，并按照这个维度的数据值对空间进行划分，一般取方差最高的那个维度的值的中值作为划分值；
2. 重复在下一个方差最高的维度继续进行划分，直到空间被划分成想要的份数。

我们的划分和这种对 KD 树的划分有所不同。对于分布式系统，我们更关心的是每个计算节点的数据分配和负载平衡，所以我们对其作一定的修改。首先，设置一个阈值参数 P_{max}，然后按照如下的方法进行划分：

1. 判断数据点的数量是否小于 P_{max}，如果小于则无需划分。否则找到数据中方差最高的维度，按照这个维度的中值作为划分值划分数据；
2. 重复对每一个数据点个数大于 P_{max} 的块进行划分：找到块中方差最高的维度，按照这个维度的中值作为划分值划分这个块。

这样，我们就能保证每个块中的数据点数量不少于 $P_{max}/2$ 而且不多于 P_{max}，保证了一定的负载均衡而且保证数据量不超过阈值 P_{max}，防止节点负载过大。当数据量大的时候可以选择尽可能较大的 P_{max} 充分利用每个节点的存储计算性能；当数据量小的时候可以适量调小 P_{max} 以便数据更分散地分配到各个节点中提高并行性。

第 4 章 两层索引结构

4.1 全局索引（RT-CAN）

我们使用 RT-CAN 索引作为我们的第一层索引，即寻找存储节点的索引。RT-CAN 索引是一种建立在本地索引之上的索引 [5]。它使用 C^2 网络作为底层存储节点分布，通过定义 R 树数据节点如何划分到各个存储节点来得到数据的组织方式，并且定义相关算法能对其中的数据进行查询。下面首先介绍 RT-CAN 节点的结构，然后介绍如何构造 RT-CAN 索引。

4.1.1 RT-CAN 节点结构

RT-CAN 索引是建立在一个 shared-nothing 的集群上的。如图 1 所示，集群中的每一个节点 N_i 都包含了两个部分：一个是存储节点 N_{si}，另一个是覆盖节点 N_{oi}。N_{si} 表示的是分布式存储的特征，它存储着所有数据划分的一部分。为了满足空间近似查询，以及空间近似的 kNN 和 RkNN 查询，N_{si} 使用了一种 R 树的变种来存储局部数据，从而满足我们的查询需求。N_{oi} 是用来表示 CAN 结构化覆盖的部分，它负责的是 CAN 划分的一部分。对于 CAN 的网络通信来说，N_{si} 会适应性的选择一部分局部 R 树的结点，然后通过 N_{oi} 来将这些节点信息发送到 CAN 网络中。发送的信息结果为一个二元组 (ip, mbr)，其中 ip 是 N_i 节点的 IP 地址，mbr 是这个 R 树结点的范围。当 Nsi 收到 N_{oi} 的发送请求时，Nsi 就会选择相应的 R 树结点并将其 map 成为一个 CAN 节点，并通过 CAN 的路由协议将请求发出。N_{oi} 维护着全局索引，当它收到一个广播请求，它就通过 map 方法判断这是否是它要接受的请求。如果时，它就保留一份广播的 R 树的结点当作索引并保存。这样就能做到用一些 R 树的结点来当作索引并将其分布在集群中。

4.1.2 RT-CAN 索引构造

基于我们的需求，我们使用的是二维的 C^2 来构建我们的 RT-CAN 索引。前面提到我们需要一个 map 方法将一个 R 树结点 map 为一个 CAN 节点，这样的 map 方法一般要以这个 R 树节点的中心和半径来确定。对一个二维的 R 树节点 n，范围为 $[l_1, u_1], [l_2, u_2]$，中心和半径分别表示为 $c_n = (\frac{l_1+u_1}{2}), r_n = \frac{1}{2}\sqrt{(u_1-l_1)^2+(u_2-l_2)^2}$。首先，对于 R 树节点 n，我们首先把它 map 到包含 n 的中心 c_n 的 CAN 节点 N_c 上，然后 N_c 会比较 n 半径和定义的一阈值参数 R_{max}，如果 n 的半径小于 R_{max}，就只需要 map 给这一个 CAN 节点，否则就需要将 n 发送给所有和 n 范围覆盖的所有 CAN 节点。这样可能会导致一些副本的出现，但同时也会提升查询的效率，因为只保存一个索引会导致所有相关的搜索都要在网络中查询这个索引，会降低查询效率。

然后，对于索引的构造，对于每个 CAN 节点，若假设其存储的 R 树是 L 层的，我们就选择将 R 树的 $L-1$ 层的所有 R 树节点发送，因为它们不是经常被更新的，这样就减少了更新索引的次数。然后对每个 CAN 节点都执行发送的操作，然后就可以按照我们前面提到的 map 算法来判断如何构造我们的全局索引。

4.1.3 RT-CAN 的性质

定理 1. *对一个点查询 $Q(key)$，如果我们查询了所有以 key 为圆心，R_{max} 为半径的的圆覆盖的所有的 CAN，那么我们就一能得到完整的结果。*

定理 2. *对于一个范围查询 $Q(range)$，如果我们查询了所有以 $range$ 的中心为圆心，$R_{max}+range.radius$ 为半径的源覆盖的所有的 CAN，那么我们就一定能得到完整的结果。*

4.2 本地索引（*MVR-Tree*）

本地索引结构主要是基于 R-Tree [1]，结合 [6] 和 [4] 两篇文章的工作，分别取 MHR-Tree 在处理范围查询上的良好表现，以及 Voronoi 图在处理 kNN 和 RkNN 上的良好效果，所以将本地的索引结构称为 ***MVR-Tree(Min-wise signature with linear hashing and Voronoi diagram R-Tree)***。

由 RT-CAN 给每个单机分配的内容，我们可以得到的一个 R 树；根据 [3] 中提到的如 Fortune's sweepline 算法，可以用来构建 D 上的 Voronoi 图，并将 Voronoi 图中的邻居 $VN(o_i)$ 和每个 cell 对应的区域 $V(o_i)$ 记录在每个节点 o_i 中；另外根据 R 树每个叶子节点 o_i 的关键字信息，可以计算其对应的 q-grams g_{o_i} 和对应的 min-hash 签名 $s(g_{o_i})$，并根据所有叶子节点的 $s(g_{o_i})$ 可以递归地自底向上的构建出所有 R 树内节点的 min-hash 签名 [6]。

因此，**_MVR-Tree_ 是一个 R 树的变种，并在每个叶子节点 o_i 记录 Voronoi 图的信息 $VN(o_i), V(o_i)$ 和 min-hash 信息 $g_{o_i}, s(g_{o_i})$，以及在每个内节点 u 记录 $s(g_u)$**。

第 5 章 查询处理

在本地使用 MVR-Tree 进行范围查询、kNN 查询和 RkNN 查询，分别是使用 [6] 中针对 range query 的方法和使用 [4] 中针对 kNN 和 RkNN 的方法来实现。

可以这样做的正确性是依赖于 R 树的性质：无论是 [6] 中提出的 MHR-Tree 和 [4] 中提出的 VoR-Tree，都是在叶子节点增加了额外的信息用于查询式剪枝，而没有改变 R 树的性质。并且在 [4] 中提到所有原本在 R 树上可以进行的查询，都可以在 VoR-Tree 上照常进行。而 MVR-Tree 只是结合了两种索引，并没有改变本质上作为 R 树的性质。所以直接利用 [6] 和 [4] 中的相应算法，在 MVR-Tree 上就可以进行查询，并且可以保证正确性。

5.1 Range Query

对于1.2节所提到的范围查询，$1-8$ 行根据 [5] 中 Range Query 的方式将查询定位到若干台机器上，$9-27$ 行是在本地利用 MVR-Tree 进行查询。其中通过 $14-15$ 行来实现近似关键字查询剪枝；通过 12 行来实现范围上的查询。具体的步骤见算法1。

Algorithm 1 Range Query(RT-CAN *Root*, (Q_{rs}, Q_{rt}))

1: $S_t = \emptyset$
2: $N_{init} = CAN.lookup(Q_{rs}.center)$ ▷ $Q_{rs}.center$ 是查询范围 Q_{rs} 的中心

$C = generateSearchCircle(Q_{rs}.center, R_{max} + Q_{rs}.radius)$ ▷ $Q_{rs}.radius$ 根据定理2计算
for i = 1 to d **do**
  将 C 根据 l_i, u_i 划分为 R_0, R_1, R_2
  $C = R_0$
  将查询信息发送到 N_1, N_2 在上面重复该查询 ▷ N_1, N_2 是 N_{init} 的邻居
$S_i = N_{init}.globalIndex$
for $\forall I \in S_i$ **do**
  if I 的区域与 Q_{rs} 有交集 **then**
    利用 I 找到本地 MVR-Tree 索引 R
    将队列 L 和本地结果 O 初始化为 $\emptyset$
    将 R 的根节点 u 插入 L
    while $L \neq \emptyset$ **do**
      取 L 的队首元素 u 并且其弹出
      if u 是叶节点 **then**
        for 对于每个 $o \in u$ **do**
          if o 在 Q_{rs} 中 **and** $|g_o \cap g_\sigma| \geq \max(|kw_i|, |kw_j|) - 1 - (\theta_j - 1) * q$ **then**
            if $\text{ED}(kw_i, kw_j) < \theta_j$ **then** ▷ $kw_i \in o.kw, (kw_j, \theta_j) \in Q_{rt}$
              将 o 插入 O 中
      else
        for u 的每个子节点 p_i **do**
          if Q_{rs} 和 p_i 的区域存在交集 **then**
            利用 [6] 中提到的方法估计 $|\widehat{g_{kw_i} \cap g_{kw_j}}|$ ▷ g_{kw_i} 是 p_i 节点关键字的 min-hash 签名
            if $|\widehat{g_{kw_i} \cap g_{kw_j}}| \geq |kw_j| - 1 - (\theta_j - 1) * q$ **then**
              将 p_i 插入 L 中
    输出 O

5.2 kNN Query

对于1.3节提到的 kNN 查询，是利用 MVR-Tree 先找到距离查询 Q_s 最近的邻居（即 1-NN），根据 1-NN 的结果以及其在 Voronoi 图上的邻居来实现接下来查询的剪枝，如此相比 [6] 中直接利用 MHR-Tree 和堆进行的 kNN 查询具有更好的 I/O 代价 [4]。具体的过程如算法2所示。

Algorithm 2 kNN Query(MVR-Tree R, (Q_s, Q_t, k))

将小顶堆 H 初始化为 $\emptyset$，$bestDist = \infty$，$bestNN = null$
将 R 的根节点 r 插入 H，即 $H = \{(r, 0)\}$
while $H \neq \emptyset$ **do**
  取 H 的堆顶元素 u 并且其弹出
  if u 是叶节点 **then**

for 对于每个 $o \in u$ **do**
if $\text{Dis}(Q_s, o) < bestDist$ **then**
$bestNN = o; bestDist = \text{Dis}(Q_s, o)$
if $bestNN! = null$ **and** $V(bestNN)$ 包含 o **then**
Break;
else
for u 的每个子节点 p_i **do**
将 $(p_i, mindist(p_i, Q_s))$ 插入 H 中
if $bestNN! = null$ **then**
将 H 弹空，将 $(bestNN, \text{Dis}(bestNN, Q_s))$ 插入 H
$Visited = \{bestNN\}; counter = 0;$
while $counter < k$ **do**
$counter + +;$
取 H 的堆顶元素 p，并将其弹出
输出 *counter*-NN 即 p
for 对 p 的每个 Voronoi 图邻居 p' **do**
if $p' \notin Visited$ **then**
将 p' 加入 $Visited$，并且将 $\text{Dis}(p', Q_s)$ 插入 H
else
不存在 1-NN，算法结束

5.3 Reverse kNN Query

参考文献

[1] Guttman A. R-trees: a dynamic index structure for spatial searching[M]. ACM, 1984.

[2] Aurenhammer F. Voronoi diagrams—a survey of a fundamental geometric data structure[J]. ACM Computing Surveys (CSUR), 1991, 23(3): 345-405.

[3] Okabe A, Boots B, Sugihara K, et al. Spatial tessellations: concepts and applications of Voronoi diagrams[M]. John Wiley & Sons, 2009.

[4] Sharifzadeh M, Shahabi C. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1231-1242.

[5] Wang J, Wu S, Gao H, et al. Indexing multi-dimensional data in a cloud system[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 591-602.

[6] Li F, Yao B, Tang M, et al. Spatial approximate string search[J]. IEEE Transactions on Knowledge and Data Engineering, 2012, 25(6): 1394-1409.

[7] Mishra S, Suman A C. An efficient method of partitioning high volumes of multidimensional data for parallel clustering algorithms[J]. arXiv preprint arXiv:1609.06221, 2016.