分布式空间近似关键字查询系统

王丙昊[1]（50%），干寅雷[2]（50%）
[1]学号：1160300302，班级 1603108
[2]学号：1160300308，班级 1603108

ABSTRACT

在本次实验中，我们设计了一个基于无共享集群的分布式近似关键字查询系统。近些年来，已经有许多工作有效地回答了大规模空间数据库中的近似关键字查询问题，如基于 min-hash signature 和线性散列技术的 MHR-Tree 索引**[4]**，该索引结构可以有效地支持空间近似关键字范围查询以及 KNN 查询，我们使用该索引作为每个计算节点上的本地索引。在一个分布式系统中，全局索引的维护往往对系统性能有显著的影响，我们在所设计的系统中使用了 RT-CAN**[5]**全局索引结构，它建立在本地 R-Tree（或其扩展）索引之上，可以在大规模无共享集群中提供有效的数据检索服务。对这样一个系统来说，各个计算节点上的负载均衡以及每个计算节点上数据的空间近似性对系统的性能也有关系，在数据划分时，我们使用了一种基于聚类的空间数据划分算法**[1]**，它充分考虑了负载均衡以及空间近似性。最后，我们总结了在该系统上进行范围查询以及 KNN 查询的算法过程。

1. INTRODUCTION

目前许多应用都要求支持空间近似关键字查询服务，这样的查询包括两部分，一组关键字条件及空间近似条件，目标是返回同时满足关键字查询条件及空间近似条件的对象，如一些基于位置的网络服务 Google Maps 等。近似字符串查询在模糊查询或用户数据错误的情况下，或数据库中的数据存在不确定性时是非常有用的。

对于如何衡量关键字之间的相似度的问题，编辑距离是最常使用的，可以使用动态规划的算法来求解，但是当数据规模较大时，会产生相当大的计算开销。空间近似关键字查询有两种直观的解决方案：1）先找到符合空间范围的所有点，然后确定是否相似，这种方法也叫做 R-Tree 方法；2）先找到满足关键字近似条件的点，再看其是否在空间范围内，这种方法为字符串索引方法。

在本次实验中，我们结合了许多相关的研究成果，将它们有机结合起来，组成了一个完整的分布式空间近似关键字查询系统，**Contributions：**

- 在系统中使用了一种基于统计聚类的空间数据划分算法，将空间数据划分到计算节点上，既保持了负载平衡，又满足每个计算节点上空间近似的特性。提出了一种基于阈值的方法，来保证新到来的数据不会影响负载平衡以及空间近似的特性，并对计算节点上的数据进行数据备份。
- 采用 MHR-Tree 作为计算节点上的本地索引，在其基础上使用 RT-CAN 构建了全局索引。我们讨论了这两层索引结构所涉及到的关键技术以及构建、动态维护方法。
- 我们总结了在该分布式空间近似关键字查询系统中进行查询的算法，包括范围查询以及 KNN 查询。

我们在 Section 2 中对该系统索要处理的问题做了形式化定义，其中包括近似关键字查询、空间近似关键字查询、空间近似关键字范围查询以及空间近似关键字 KNN 查询；在

Section 3 中介绍了我们所使用的基于聚类的空间数据划分算法，其中详细描述了该算法所涉及到的关键技术，以及导入新的数据的方法和数据备份的方法；在 Section 4 中详细描述了本系统所使用的两层索引结构——MHR-Tree 索引以及 RT-CAN，包括两层索引的基本思想、涉及到的关键技术、构建方法以及维护方法等；最后我们在 Section 5 中，详细介绍了如何在该分布式系统上执行空间近似关键字范围查询以及空间近似关键字 KNN 查询。

2. PROBLEM FORMULATION

对于空间对象集合 D = {$p_1, p_2, \cdots, p_n$}，每条空间数据都可以表示为（p_i，σ_{i1}，σ_{i2}，$\cdots$,σ_{iw}），其中 p_i表示该空间数据的位置，而σ表示与该空间数据相关的关键字信息。一个空间数据库中的空间数据与一个或多个关键字相关，且不同数据可能有重复的关键字。

Definition 1（空间近似关键字查询）一个空间近似关键字查询 Q 包含两个部分：空间查询 Qr 以及关键字查询 Qs。空间查询 Qr 可以通过指定空间谓词以及谓词参数，来执行不同种类的空间查询，比如我们接下来要介绍的范围查询及 KNN 查询，对于范围查询来说，Qr 通常是一个矩形或圆形，对 KNN 查询来说，Qr 为一个（t, K）对，其中 t 表示一个点，K 为 KNN 查询的参数；关键字查询 Qs 有一个或多个关键字及其编辑距离阈值组成，Qs = {(σ1，τ1), (σ2，τ2)，……, (σβ，τβ)}，其中σi 表示查询关键字，τi 表示关键字近似条件。

Definition 2（空间近似关键字范围查询）一个空间近似关键字范围查询 Q：(Qr = r, Qs)，它将返回空间中点的一个集合 A：

$$\forall p \in \mathcal{A} \begin{cases} p \in r & p \text{ is contained in } r; \\ p \in \mathcal{A}_s & p \text{ has similar strings to all query strings.} \end{cases}$$

Definition 3（空间近似关键字 KNN 查询）一个空间近似关键字 KNN 查询 Q：(Qr = t, Qs) 返回空间中距离 t 最近的且满足关键字查询条件的 K 个点：

$$\begin{cases} \mathcal{A} = \mathcal{A}_s & \text{if } |\mathcal{A}_s| \le k; \\ |\mathcal{A}| = k \text{ and} & \\ \forall p' \in \mathcal{A}, \forall p'' \in \mathcal{A}_s - \mathcal{A}, \|p', t\| < \|p'', t\| & \text{if } |\mathcal{A}_s| > k. \end{cases}$$

如果空间中点的个数小于 K，那么该查询返回所有满足关键字查询条件的空间数据。

3. DATA STORAGE

在这一节中，我们描述了数据存储方面的技术，其中包括数据划分、导入新的数据，以及数据备份。数据划分时，我们使用了一种**基于统计聚类的空间数据划分算法[1]**，它可以满足计算节点上的负载均衡以及空间近似性（Section 3.1）。该算法要求在进行数据划分时数据是静态的，也就是在算法执行过程中数据不发生变化，在下一节中介绍了在数据划分完之后，如何导入新的数据（Section 3.2）。**最后，我们介绍了数据备份（TODO）**

3.1. 基于统计聚类的空间数据划分算法

A. 数据划分的基本准则

进行空间数据划分时，我们应该保证负载均衡，各个计算节点上的数据量相近；并且要把算法的时间复杂度以及计算资源开销考虑在内。基于以上观点，我们总结了空间数据

划分的四条基本准则:

1）并行计算节点上的数据负载平衡;

2）同一个计算节点内的数据有较好的空间近似性

3）不同的计算节点间没有数据冗余（仅在数据划分情况下，不把数据备份考虑在内）

4）算法的应具有大规模数据的可扩展性（时间复杂度等）。

B. 数据划分的前提条件

执行该空间数据划分算法，要求执行过程中数据是静态的，也就是在整个算法的执行过程中，数据不能发生增删改操作。对于新到来的数据，需要在初始数据执行完划分算法之后才能导入，关于新数据如何导入，在 Section 3.2 中介绍。

C. 算法关键技术

1) 聚类方法的选取

基于统计的方法可以处理不同的聚类要求，如划分算法，分层算法，基于密度的算法等。这些不同种类的方法都可以用来进行空间数据划分，将数据集聚类成 K 个 block。当 k 等于并行计算节点数的时候，数据划分工作结束。

聚类问题的一个好的划分准则为最小化平方误差

$$E = \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in C_i} ||\boldsymbol{x} - \boldsymbol{\mu}_i||_2^2$$

该式在一定程度上刻画了数据划分的紧密程度，E 值越小则簇内样本近似度越高。最小化该式子是一个 NP 难问题，事实上最广泛使用的方法是基于贪心思想的 K-Means 算法，该算法非常简单，并且具有很好的可扩展性，但容易受到离群数据的影响。

这里使用一种基于层次的聚类算法来获取初始 K 值和初始聚类块，然后通过迭代重定位来改进聚类结果。

在选取了聚类方法的基础上，为了满足空间数据划分的需求及基本准则，我们需要对算法做一些细致的改进，如 K 值的选取，空间数据聚类的度量准则及聚类中心的选取等，下面将分别介绍这些问题。

2) K-Value 的选取

K 值的选取对最终簇的数量、聚类结果以及算法执行时间都有影响。该算法使用一种基于层次的聚类算法来获取初始 K 值（K_0-Value）和初始聚类块，然后通过迭代重定位来改进聚类结果，下面分别介绍 K_0 以及 K_i 的选取。

K_0-Value：可以通过最小化平方误差来获取精确的 K_0-Vaue，但这样不实际，并且在我们的数据划分要求中，对聚类精度没有那么高的要求，所以这里我们根据经验来获取 K_0-Value：当数据规模为 10_1，10^2，10^3 时，K_0-Value 取空间对象数量的 1/10，1/100，1/1000.

K_i-Value：$K_{i+1} = K_i / P$。P 通常取 2~5.

3) 空间数据聚类的度量准则

选取一种空间度量准则来进行聚类，如最短距离，最短距离累计准则，最小面积准则（聚类过程中将对象表示成 MBR）。

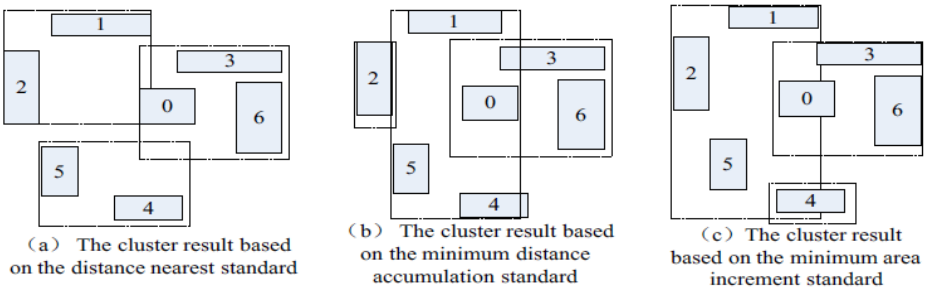

Figure1. 基于不同度量准则的聚类结果

Fig1. (b)可以看出，可能将某一维的数据相近的聚类到一个簇中，而其他维相近的分布在不同的簇中；Fig1. (c)可以看出，MBR 中会有许多空白区域。而 Fig. (a)效果相对于另外两个要好，所以最终我们选取最短距离作为度量准则，这里的距离指的是两个 MBR 中心的欧氏距离。

4) 聚类中心的选取

在聚类算法中，初始中心点的选取对聚类效果有非常显著的影响。在这里，我们希望选择的中心点分布均匀，这里使用一种排序空间的间隔下标的选择方法。首先确定间隔 T = N / K。将空间数据按照一维的坐标进行排序，每隔 T 个对象选取一个空间数据加入到初始中心点集合中。虽然数据是按照某一维进行排序的，但中心点序列近似地均匀覆盖了整个空间。

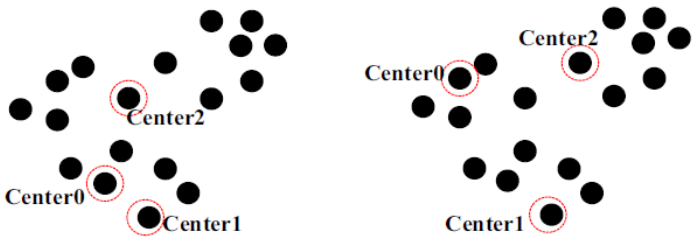

Figure2. 中心点选择方法的效果对比

D. 算法流程

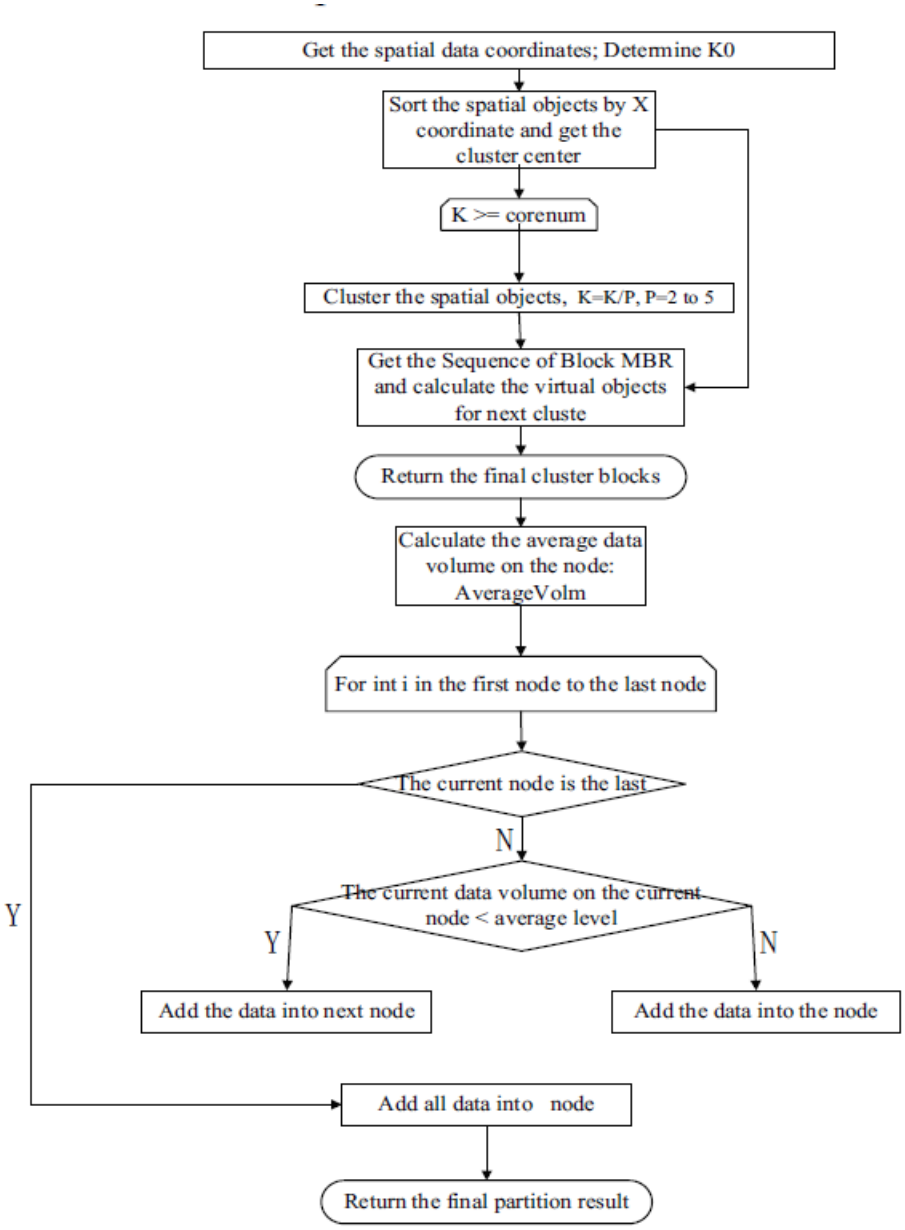

Figure3. 空间数据划分算法流程图

3.2. 导入新的数据

在执行完数据划分算法之后，将各个 Block 分配到不同的计算节点上，并在计算节点上构建本地索引（MHR-Tree，Section 4）以及进一步构建全局索引（RT-CAN Section4）。将新的数据导入该分布式近似关键字查询系统时，也要考虑各个计算节点的负载均衡以及空间近似性。

A. 空间近似性

我们可以采用与划分算法相同的度量准则来将数据划分到计算节点上，以保证空间近似性：每个计算节点上数据都被组织成 MBR，通过计算新到来的数据与各个计算节点 MBR 的最短距离来决定新数据所属的计算节点，假设有 N 个计算节点，新到来的数据规模为 M，这种精确导入的时间复杂度为 O(MN)；由于该系统对一个节点上的空间数据的近似性要求没有那么高，我们可以通过设置一个阈值 T，如果计算得到的新数据与计算节点 MBR 的最短距离小于 T，则不必去计算到其他计算节点 MBR 的最短距离，这相当于是一种精确

度与时间复杂度的 Tradeoff。

B. 负载均衡

在执行完数据划分算法后，当将新的数据导入计算节点时，我们需要动态地维护各个计算节点的负载均衡。首先定义一个阈值 T，当计算节点的数据规模大于 T 时，我们对该计算节点上的数据进行划分，将其 MBR 划分成两个分区，然后添加子分区到下一个计算节点。这个阈值 T 不可以太大，不然在划分前可能会导致数据的不均衡；也不可以太小，不然在划分后数据规模会远小于其他未划分的数据规模。

3.3. 数据备份

数据备份由监控器，备份模块和恢复模块三部分组成。:

1）第一部分，监视器用于检测用户应用程序中磁盘文件的操作，例如创建文件，写入文件，更改文件属性和其他事件。

2）第二部分，增量备份模块根据监视器先前记录和发送的事件进行备份。

3）第三部分，当由于数据丢失而需要恢复数据时，恢复模块从备份介质读取数据并将它们恢复到存储介质。

使用了一种数据滚动技术，实现了多个增量数据滚动来恢复数据到指定版本。在数据恢复技术中，备份数据文件根据原始树结构目录进行组织。 创建文件夹节点与每个文件的原始文件名相同。 增量文件形式的多个增量数据将存储在同一文件的该文件夹中。 这些增量文件按增量生成顺序组织。 当我们根据需要恢复数据时，我们只是进行数据滚动。 也就是说，首先将原始完整备份版本文件复制到临时文件中。 然后从第一个增量开始，依次对该临时文件执行根据记录事件的相同操作。 因此，所有增量数据都将转换为指定的文件版本

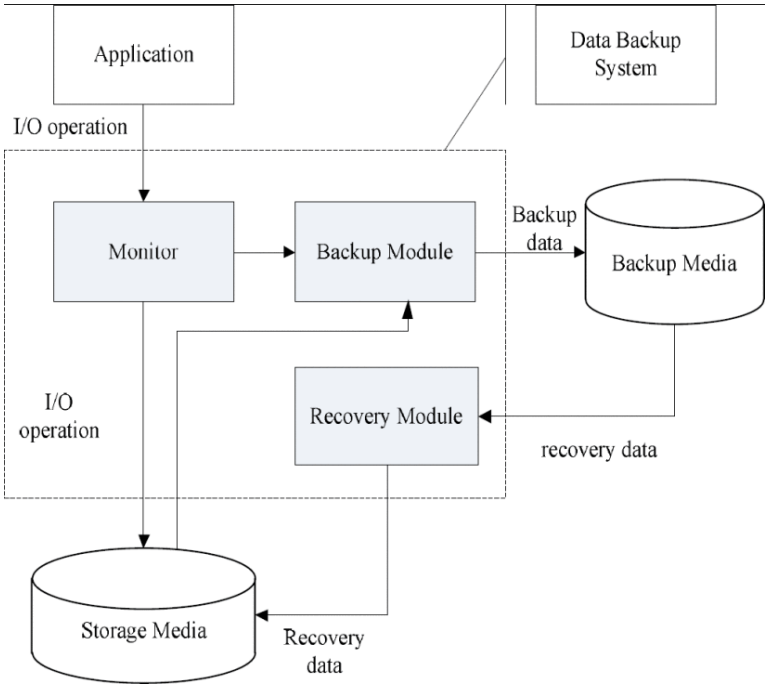

Figure4. 数据备份实现原理图

4. INDEX：两层索引结构

在我们的分布式空间近似关键字查询系统中，使用了 RT-CAN 的两层索引结构，其中本地索引用于指定数据在哪个磁盘块，哪个内存部分，本地索引使用 MHR-Tree 来实现；在本地 MHR-Tree 索引的基础上，我们从中选择全局索引，并映射到计算节点上。在本节

中，我们主要介绍这两种索引结构的关键技术、索引的构建及维护方法。

4.1. 本地索引：MHR-Tree

A. MHR-Tree 的基本思想

MHR-Tree 是基于 Min-wise signature 以及 liner Hashing 技术的 R-Tree，是一种在空间索引结构上加入关键字集合的概要信息来实现近似关键字查询功能的一种索引结构。MHR-Tree 上节点 u 的 min-hash 签名表示了以该节点为子树的 q-grams 的概要信息，我们通过求解 query string 与树节点 min-hash 签名的相似度来估计它们 q-grams 集合的相似度，以在空间搜索的过程中实现近似关键字查询和剪枝功能。下面将详细介绍 MHR-Tree 的基本思想。

在 Section 1 中我们提到了空间近似关键字查询的 R-Tree 方法，先找符合空间条件的所有点的集合，然后再通过关键字查询条件进行筛选。但这种方法会访问到不必要的树节点，导致 IO 代价较大，导致进行筛选时，关键字近似匹配的 CPU 代价也会较大。会出现这种状况的原因是，R-Tree 没有使用剪枝策略来避免访问不可能出现在最终结果集合中的叶节点上的数据。MHR-Tree 在此基础上做了改进，它在查询时使用了空间查询谓词和近似关键字查询谓词提供的剪枝功能，使得查询的 IO 代价和 CPU 代价降低。

MHR-Tree 使用的是一种基于 q-gram 集合相似度的剪枝策略。使用该策略的一种直观的思想是直接在 R-Tree 的节点中存储以该节点为根的子树中空间数据 q-gram 集合的并集，但在高层节点上该集合的规模非常大，这样既增大了存储开销，还增加了查询成本，这样得不偿失。

为了解决这个问题，有一项研究采用了将 R-Tree 的节点分类的思想，仅在部分节点中保存关键字信息以及 q-gram 的倒排索引，其他的节点仅保留 MBR 以进行空间检索**[7]**。与之不同，MHR-Tree 采用的是在中间节点中存储子树中 q-gram 集合的 min-hash 签名，min-hash 签名具有体积小而且固定的特点，可以在树节点中存储，通过计算 query string 与子树 min-hash 签名的相似度来估计 q-gram 集合的相似度，以实现剪枝功能。

下面将详细介绍如何使用 min-hash 签名来估计 q-gram 集合的相似度。

B. MHR-Tree 的核心技术

1）基于 q-gram 的近似关键字查询

直接计算两个关键字σ1 和σ2 的编辑距离的代价比较大，时间复杂度为 O(|σ1||σ2|)，通常都是通过一种基于 q-gram 的过滤-验证框架先选出候选结果，然后再验证。使用关键字条件进行过滤时，主要使用的是：

Lemma 1：如果 ED(s, t)⩽t，则 s 和 t 的公共 q-gram 不少于 max{|s|, |t|} – q +1 – kq.

2）min-hash 签名

在 MHR-Tree 中的节点中存储子树 q-gram 集合的 min-hash 签名来代替 q-gram 集合。两个集合的相似度可以通过 Jaccard 相似度来衡量

$$\rho(A,B)=\frac{|A\cap B|}{|A\cup B|}.$$

而使用 min-hash 签名可以无偏地估计这两个集合的相似度，即：

$$\widehat{\rho}(A,B)=\Pr(\min\{\pi(A)\}=\min\{\pi(B)\}).$$

我们可以通过一种线性哈希的方式来模拟哈希签名的生成。指定 min-hash 签名的长度 l，

通过 liner hash 来生成 l 个函数 h1, h2, …, hl，那么集合 A 的 min-hash 签名为

$$s(A) = \{\min\{\pi_1(A)\}, \min\{\pi_2(A)\}, \ldots, \min\{\pi_\ell(A)\}\}$$

其中Π指的就是 liner hash 生成的函数 h。两个集合相似度的无偏估计最终可以表示为

$$\widehat{\rho}(A,B) = \frac{|\{i|\min\{\pi_i(A)\} = \min\{\pi_i(B)\}\}|}{\ell}.$$

我们使用 min-hash 签名来代替真实的 q-gram 集合，将其存储在 R-Tree 的节点中，记为 S(Gu)。S(Gu)的大小是固定的，所以这样就解决了在空间索引中保存关键字概要信息的问题。

C. MHR-Tree 索引的构建

构建 MHR-Tree 的过程中要解决的问题如下：

1) 指定 min-hash signature 的长度 l，通过线性散列技术生成计算 min-hash 签名的 l 个函数。
2) 构造 R-Tree，内节点的儿子列表中要预留 min-hash 签名的空间。
3) 在叶节点中计算每个关键字的 min-hash 签名，然后合成叶节点的 min-hash 签名。
4) 自底向上地完成所有节点的 min-hash 签名的计算。

在由每条空间数据的 min-hash 签名生成叶节点的 min-hash 签名，以及自底向上地计算所有节点的 min-hash 签名时， 需要使用到下面的定理：

Lemma 2：假设集合 A 和 B 的 min-hash 签名分别为 S(A)和 S(B)，那么集合 A∪B 的 min-hash 签名为 S(A∪B) = {min{S(A)[1], S(B)[1]}，…， min{S(A)[l], S(B)[l]}}。这种计算方式可以扩展到 k 个集合的并集。基于上面的思想，总结 MHR-Tree 索引构建的算法如下：

Algorithm 1: Construct-MHR

```
1  Use any existing bulk-loading algorithm A for R-tree;
2  Let u be an R-tree node produced by A over P;
3  if u is a leaf node then
4      Compute G_p and s(G_p) for every point p ∈ u_p;
5      Store s(G_p) together with p in u;
6  else
7      for every child entry c_i with child node w_i do
8          Store MBR(w_i), s(G_{w_i}), and pointer to w_i in c_i;
9      for i = 1, ..., ℓ do
10         Let s(G_u)[i] = min(s(G_{w_1})[i], ..., s(G_{w_f}[i]));
11         Store s(G_u) in parent of u;
```

D. MHR-Tree 索引的维护

由于在 R-Tree 中加入了 min-hash 签名，使得 MHR-Tree 的动态更新更复杂。这里我们主要考虑了插入和删除操作。

插入新数据：我们遵循 R-Tree 插入算法，然后计算新插入数据的签名，并将其签名与

包含它的叶节点的签名联合起来，使用上面提到的 **lemma 2** 重新计算叶节点的 min-hash 签名，改变以类似的方式自底向上地传播到父节点。当某次传播过程中父节点的 min-hash 签名不发生变化时，传播停止。

删除数据：删除操作相比于插入要涉及到更多操作。如果删除的数据的 min-hash 签名与其所属的叶节点的 min-hash 签名某些位上的值相同，那么就需要通过从其他数据的 min-hash 值中找到这些位置上的新值。这些更新可能会在树中进一步传播，传播与更新过程与插入操作相同。

这里需要注意的是，添加 min-hash 签名不会影响 R-Tree 的更新性能，因为签名永远不会导致结构更新。因此，MHR-Tree 的更新性能和 R-Tree 的更新性能完全相同。

4.2. 全局索引：RT-CAN

A. RT-CAN 的基本思想

RT-CAN 索引是基于 R 树和 CAN（peer-to-peer 覆盖网络）建立的一种索引结构，它是一种分布式索引结构。RT-CAN 是建立在本地索引上的一种全局索引，RT-CAN 索引需要解决的一个重要的问题是，如何将本地 R 树选择提取出的节点映射到 CAN 覆盖网络上。

通过构造一个投影函数，函数定义了一个 CAN 覆盖网络节点最大半径 Rmax，然后计算出 R 树节点的半径 r，比较 r 和 Rmax，假如 r 小于 Rmax，说明 R 树节点被 CAN 网络上的某一个节点完全覆盖，那么这个 R 树节点就被投影到 CAN 网络的那个节点上。假如 r 大于 Rmax，对于以 R 树节点为中心，r 为半径的圆可以得到这个圆形区域与 CAN 网络上的多个区域相交，最后将 R 树节点投影到所有的相交的区域。Rmax 取值要求：尽量设置成一个比 R 树所有叶节点半径大的一个值。

Defination 4（CAN 覆盖网络）：CAN 覆盖网络是一个可扩展的 p2p 覆盖网络，CAN 可以是一个多维的覆盖网络，每一个节点有个 P 维的覆盖空间，在各个维度方向上维护一条链表结构，用于存储每一个维度上的邻接节点，用于方便 RT-CAN 的索引查找。对于空间近似关键字查询来说，CAN 网络上的节点N_i存储两部分内容，第一部分N_{Si}用于存储局部 MHR 树索引结构（MHR 树索引结构是 R 树索引结构上的扩展），第二部分N_{Oi}用于存储从局部 MHR 树索引中提取出的 R 树节点，并且将N_{Oi}发布到 p2p 覆盖网络上，形成最终的全局索引。

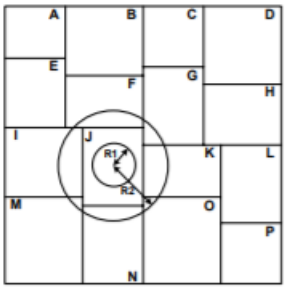

Figure5. 一个二维的 CAN 覆盖网络举例

B. RT-CAN 在本地索引的选择

如何从本地索引中选择全局索引，需要满足两个要求：

1、 **完整性**：对于 R 树中的所有叶节点，它们要么是全局索引的一部分，要么它们的父

亲节点是全局索引的一部分

2、**唯一性**：对于 R 树中的某个叶节点如果它是全局索引的一部分，那么他的父节点必定不在全局索引中，如果某个父亲节点包含在全局索引中，那么它的所有子节点必然不包含在全局索引中。

索引的选择是基于某个权值计算函数：

$$C(n) = |Q_{FP}(n)| + 3(p_{split}(n) + p_{merge}(n)).$$

假设 n 是 R 树上的某个节点，它的代价估计函数 C(n)由两个部分组成，第一部分$|Q_{FP}(n)|$它表示在范围查询过程中，有些 CAN 覆盖网络上的节点在查询过程中可能会被访问，但是其中的全局索引并没有包含 n，也就是错误的查询的节点的数量。第二部分 $3(p_{split}(n) + p_{merge}(n))$表示节点 n 的分裂或者合并产生的代价，因为节点变化的次数以及时间不可知，所以通过马尔科夫假设，将这个代价估计成在未来的某个时间下该节点分裂或者合并的概率。通过代价估计函数，可以得到索引的选择算法如下：

Algorithm 2: R-Tree Node Selection Algorithm

Input: local R-Tree T_R
Output: A set of R-Tree nodes S
1: **for** each R-tree node n_i in T_R **do**
2: update cost of n_i using query and update histograms
3: **end for**
4: $\{S, C\}$=$IndexSelect(T_R.root)$ /* *invoke Algorithm* */
5: return S

Algorithm 3: Index Select

Input: R-Tree node n
Output: A set of R-Tree nodes S, Total Cost C
1: $S = \{n\}$
2: $C = C(n)$
3: **if** n is a leaf **then**
4: return S and C
5: **else**
6: $S_{temp} = \emptyset$, $C_{temp} = 0$
7: **for** $\forall n_i \in n.child$ **do**
8: $\{S', C'\}$=$IndexSelect(n_i)$
9: $S_{temp} = S_{temp} \cup S'$
10: $C_{temp} = C_{temp} + C'$
11: **end for**
12: **if** $C_{temp} < C(n)$ **then**
13: return S_{temp} and C_{temp}
14: **else**
15: return S and C
16: **end if**
17: **end if**

首先对 R 树中的每个节点计算他的代价，然后从根节点开始执行索引选择算法，索引选择算法是一个迭代的过程。首先全局索引 S 中加入自身节点 n，索引代价 C 等于节点 n 的代价。然后S_{temp}用于记录节点 n 的子节点的全局索引，C_{temp}用于记录子节点的索引代价之和，

对于节点 n 的所有子节点，将它们的全局索引添加到S_{temp}中，计算他们的代价之和赋给C_{temp}然后比较C_{temp}和 C 的大小，如果 C 的代价较小，全局索引仍为节点 n，如果 C 的代价较大时，全局索引更新成S_{temp}。

索引选择算法从根节点出发，一直深入到所有叶节点，然后计算叶节点的代价之和与它们的父亲节点的代价进行比较，自底向上选择代价最小的全局索引，最后更新到根节点，最终的全局索引即为代价最小的全局索引。

C. RT-CAN 索引的发布

在上面一部分介绍了如何在本地索引中选取要发布的 R-Tree 节点，在本节中，我们介绍如何将所选取的全局索引映射到计算节点。索引发布算法如下所示：

Algorithm 4: Index Publication

```
1: S_i = getSelectedRTreeNode(N_i)
2: for each n ∈ S_i do
3:     c_n = n's center; r_n = n's radius
4:     key_n = compositeKey(c_n, r_n)
5:     N_j = CAN.lookup(key_n)
6:     N_j inserts n into its global index set
7:     if r_n > R_max then
8:         S_n = getOverlappedNode(n)
9:         for ∀N_k ∈ S_n do
10:            N_k inserts n into its global index set
11:        end for
12:    end if
13: end for
```

算法具体实现：已经通过 map 函数将对应的 R 树节点投影到了对应的 CAN 覆盖网络上。对 CAN 覆盖网络上的某个节点实现 RT-CAN 索引的发布，遍历 CAN 网络节点N_i中的所有 R 树节点 n，定义 R 树节点 n 的中心点C_n和半径r_n，假设 R 树节点是一个 d 维的空间矩形节点：$([l_1 - u_1], [l_2 - u_2], \cdots, [l_d - u_d])$，那么节点 n 的中心$C_n$：$(\frac{l_1+u_1}{2}, \frac{l_2+u_2}{2}, \cdots, \frac{l_d+u_d}{2})$

节点 n 的半径为：

$$\frac{1}{2}\sqrt{(u_1 - l_1)^2 + (u_2 - l_2)^2 + \cdots + (u_d - l_d)^2}。$$

key_n表示以C_n为中心r_n为半径的多维空间上的一个球体，对于N_i节点设置了一个半径 Rmax：假如半径r_n大于 Rmax，那么key_n必定在空间 CAN 覆盖网络上与多个N_k空间节点相交，将 R 树节点 n 添加到所有相交计算节点N_k的全局索引中。当 CAN 空间覆盖网络的所有节点都发布了本地选取的全局索引后，RT-CAN 全局索引构建完成。

5. ALGORITHMS

在这一部分中，我们介绍了如何在所设计的分布式空间近似关键字查询系统里面进行范围查询以及 KNN 查询。查询包括两步，第一步是全局查询，找到符合空间查询条件的计算节点上选出的全局索引；第二步根据找到的全局索引，在本地的 MHR-Tree 上找到相应的树节点，从它开始进行本地查询。在下面的部分，先介绍了在本分布式空间近似关键字查询系统中进行范围查询的过程；然后介绍了 KNN 查询的过程，KNN 查询要比范围查询更复杂，在下面会详细介绍。

5.1. 范围查询

在本分布式系统中，**范围查询的执行过程**：首先在 RT-CAN 全局索引上找到符合查询空间条件的全局索引，根据查到的全局索引的信息找到其所在的计算节点的 MHR-Tree 本地索引，接下来从从 MHR-Tree 的相应位置开始根据空间条件和关键字条件同时进行本地查找。接下来我们将分开介绍全局查找和本地查找的算法。

1) 在 RT-CAN 全局索引上根据空间条件查询

在介绍 RT-CAN 的范围查询之前，首先介绍一下 RT-CAN 中所提出的点查询，而范围查询可以看作是点查询的一种扩展或者说是一种特殊情况。

A. 点查询

在点查询算法中，先提出以下结论：对于点查询 Q(key)，使用以 key 为中心，Rmax 为半径的范围进行查询可以保证结果的准确性。证明：假设 key 的半径小于 Rmax 那么 key 完全包含于某个 CAN 空间节点N_i中，如果 key 的半径大于 Rmax 时，在索引发布算法中已经将 key 添加到所有相交的 CAN 空间节点N_k中，所以都能得到完整的结果。

点查询算法执行过程：首先将S_i结果集合设为空，在 CAN 网络中找到所有包含 key 的节点N_{init}，对于 key 的 d 维向量的每一维数据$[l_i, u_i]$可以分成三个部分R_0，R_1，R_2。其中R_0表示在$[l_i, u_i]$中的数据，R_1表示小于l_i的部分，R_2表示大于u_i的部分。根据R_0，R_1，R_2可以将一维数据划分成三个部分，然后在结合其余各维数据就可以将一个空间球体数据按照某一维特征划分成三个部分。数据的划分从第一维数据开始，每次将数据划分为三个部分，其中R_0将所有的全局索引添加到S_i中，并且R_0在下一轮划分的过程中成为了新的需要被划分的数据，因为经过第一轮的划分已经去除了在第一维中不在范围中的数据，第二轮就需要根据第二维数据进行划分，以此类推，直到所有维数据都被划分完成结束。

对于R_1，R_2的数据，因为 CAN 空间覆盖网络在每一个数据维度上维护了一个链表记录邻接节点，所以只要在划分的维度上找到两端的邻接节点，然后对两个邻接的节点N_1，N_2使用迭代查询算法，不过查询空间已经确定了就是R_1，R_2，然后将包含在这两空间中的所有全局索引添加到S_i中。

经过 d 轮迭代，所有维的数据都已经经过划分，并且将可维上的邻接节点的全局索引都添加到S_i中，然后遍历S_i中所有的全局索引，如果索引包含 key 保留否则删除，最终得到了结果S_i

Algorithm 5: RT-CAN Point Query Processing

Input: $Q(key)$
Output: S_i (the result of indexed R-tree nodes)

$S_i = \phi$
$N_{init} = CAN.lookup(key)$
$C = generateSearchCircle(key, R_{max})$
for i=1 to d **do**
    partition C into R_0, R_1 and R_2 based on l_i and u_i
    $C = R_0$
    $N_1 = getNeighbor(N_{init}, R_1)$
    $N_2 = getNeighbor(N_{init}, R_2)$
    Forward query message (N_1, R_1) to N_1
    Forward query message (N_2, R_2) to N_2
end for
$S_i = N_{init}.globalIndex$
for $\forall I \in S_i$ **do**
    if I's bounding box does not contain key **then**
        $S_i = S_i - \{I\}$
    end if
end for
return S_i

B. 范围查询

对该点查询算法稍作扩展，便可以得到 RT-CAN 的范围查询算法。

假设查询是一个范围查询 Q（range），根据索引的发布可以得到如下结论：对于某个范围查询 Q（range），计算范围查询的半径 r，以范围的中心作为中心节点，r+Rmax 作为半径进行范围查询可以得到准确的结果。

在范围查询的过程中点查询算法仍然适用，只需要将 CAN 空间覆盖网络上的某个包含这个范围的节点N_{init}作为初始节点，一半径为 r+Rmax 的空间进行查询就能得到准确的结果。

2) 在本地索引上根据空间条件及关键字条件同时查询

在 MHR-Tree 本地索引上进行查找，就是在 R-Tree 查询的基础上，使用 q-gram 集合的 min-hash 签名进行剪枝，于是关键问题就转变成了如何使用两个 min-hash 签名来估计 q-gram 集合的相似度。估计方式如下：

$$\widehat{|G_u \cap G_\sigma|} = \widehat{\rho}(G_u, G_\sigma) * \widehat{|G_u \cup G_\sigma|}.$$

其中

$$\widehat{\rho}(G_u, G_\sigma) = \frac{|\{i| \min\{h_i(G_u)\} = \min\{h_i(G_\sigma)\}\}|}{\ell}.$$

$$\widehat{|G_u \cup G_\sigma|} = \frac{|G_\sigma|}{\widehat{\rho}(G, G_\sigma)}.$$

Algorithm 6: Range-MHR

```
1  Let B be a FIFO queue initialized to ∅, let A = ∅;
2  Let u be the root node of R; insert u into B;
3  while B ≠ ∅ do
4      Let u be the head element of B; pop out u;
5      if u is a leaf node then
6          for every point p ∈ u_p do
7              if p is contained in r then
8                  if
                   |G_p ∩ G_σ| ≥ max(|σ_p|, |σ|) − 1 − (τ − 1) ∗ q
                   then
9                      if ε(σ_p, σ) < τ then
10                         Insert p in A;
11     else
12         for every child entry c_i of u do
13             if r and MBR(w_i) intersect then
14                 Calculate s(G = G_{w_i} ∪ G_σ) based on
                   s(G_{w_i}), s(G_σ) and Equation 3;
15                 Calculate |G_{w_i} ∩ G_σ| using Equation 8;
16                 if |G_{w_i} ∩ G_σ| ≥ |σ| − 1 − (τ − 1) ∗ q then
17                     Read node w_i and insert w_i into B;
18 Return A.
```

5.2. KNN 查询

KNN 查询执行过程：算法要求找到与查询点空间上最相近的且满足关键字查询条件的 K 个数据。首先我们先在全局索引上进行查找，找到 k 个与 Key 空间上相近的全局索引，注意这里的 k 与 K 不同，因为不一定能保证所找到的空间上相近的全局索引下的数据满足关键字查询条件，或者可能找到的全局索引下的数据有多个满足关键字查询条件。找到全局索引之后，在本地索引上执行本地查找。

KNN 查询要注意的一点是：如果执行完本地查询后，结果数量小于 K，那么将扩大全局查找的半径，进一步执行全局查询和本地查询。下面将分别介绍全局查找和本地查找的过程。

1) 在 RT-CAN 全局索引上查询

对于一个 KNN 查询 Q(key,k)算法得到与 key 最接近的 k 个全局索引，注意这里 k 与 K 不相等。如果 k 较大，那么全局查找的代价较大，但是可能会减少整体算法执行轮数；如果 k 较小，每轮全局查找和本地查找的代价会减小，但是可能会执行多轮算法。

首先根据关键字 key 找到包含 key 的 CAN 网络上的节点N_{init}，然后根据 k 计算选择的半径：

$$D_k \approx \frac{2\sqrt[d]{\Gamma(\frac{d}{2}+1)}}{\sqrt{\pi}}(1-\sqrt{1-\sqrt[d]{\frac{k}{N}}})$$

然后得到半径 r=δ=$\frac{D_k}{k}$，以关键字 key 为中心，r 为半径进行范围查询，如果得到的全局索引的数量大于 k，找到最接近的 k 个全局索引；如果索引的个数不足 k，将 r 的半径扩大 r=r+δ，继续范围查询直到找到满足要求的 k 个全局索引。

Algorithm 7: RT-CAN KNN Query Processing

Input: $Q(key, k)$
Output: S_i (the result of indexed R-tree nodes)

```
1: S_i = φ
2: N_init = CAN.lookup(key)
3: δ = estimateRadius(k)
4: r = δ
5: while true do
6:     S_i = Search(key, r)
7:     if |S_i| ≥ k then
8:         return top k results of S_i
9:     else
10:        r = r + δ
11:    end if
12: end while
```

2) 在 MHR-Tree 本地索引上查询

与本地范围查询类似，MHR-Tree 上的 KNN 查询也可以在 R-Tree 的 KNN 查询基础上进行扩展，也就是加入关键字查询条件进行过滤。具体实现过程中可以通过维护一个 MBR 或叶节点上的数据到查询点 key 的最小堆。

每一步执行过程中，如果堆顶的元素是数据项，则将其弹出加入到结果集合中；如果堆顶元素是 MBR，先通过估计其与 query string 的 q-gram 集合的相似度，如果达到了阈值，则将该叶节点弹出，将其儿子添加到堆中。

当结果集合的元素个数达到 K，或者最小堆中没有元素了，在该本地节点上的查询结束。如果在所有计算节点上均查询结束后，结果集合中的元素仍小于 K，那么就需要扩大全局查询的半径，执行下一轮的查询，直到结果集合的个数达到 K 或没有数据项。

Algorithm 8: R-Tree KNN Query Processing

```
Initialize Empty Priority Queue PQ;
Add root node of the R-tree R into PQ with its minDist as key;
int i = 1;
repeat
   element ele = removeMin(PQ);
   if (ele.type == internal node) then
      add all its entries to PQ with their respective
      minDist (from q) as keys;
   else if (ele.type == external node) then
      add all its entries to PQ with their respective
      euclideanDist (from q) as keys;
   else if (ele.type == datapoint) then
      report ele as i^th nearest neighbor;    i++;
      if (i>k) then
         break;
      end
   end
until false;
```

6. REFERENCES

[1] A Spatial Data Partition Algorithm Based on Statistical Cluster.
[2] Efficient Merging and Filtering Algorithms for Approximate String Searches.
[3] Answering Approximate String Queries on Large Data Sets Using External Memory.
[4] Approximate String Search in Spatial Databases.
[5] Indexing Multi-dimensional Data in a Cloud System.
[6] The R*-Tree: an efficient and robust access method for points and rectangles.
[7] Supporting Location-Based Approximate-Keyword Queries.
[8] A Data Backup Method Based on File System Filter Driver.