2019 数据库系统实现技术

期中作业

学院：计算机科学与工程学院

专业：计算机科学与技术

班级：计算机 1606 班

姓名：戚子强

学号：20164625

两种索引结构的分析

一、H-Tree 索引

1.1 简介

随着计算机网络的迅猛发展和大数据时代的到来，数据越来越频繁地呈现出多属性异构的特点。这种包含多种不同类型属性的大数据流称为异构大数据流(Heterogeneous Big Data Strems)。在面向大规模数据在线监测分析的应用中，通常需要在异构大数据流上注册大规模监测规则。因此，对于每一个数据流元组，必须用最小的计算开销满足所有的规则。同时，由于大数据流上监测规则集异常庞大，提高规则监测的性能是大规模数据流在线监测的关键。基于此，该文提出一种层次化的索引结构 H-Tree 及其在线规则匹配算法。具体的，H-Tree 将大数据流上的属性集划分为离散型属性和连续型属性。基于不同的属性集，构建两层索引结构：在第 1 层，通过改进的红黑树对离散型谓词构建触发索引；在第 2 层，通过量化连续型谓词构建多维索引结构。H-Tree 的在线规则匹配算法利用关联关系表对两层索引的监测结果进行融合过滤。实验分析表明，与经典的 R+方法相比较，H-Tree 通过层次化的索引结构，在不降低准确度的前提下，显著提升了大数据流的监测效率。

1.2 索引结构

从结构上来讲，H-Tree 包括 3 个重要组成部分以及 3 个重要操作。其中，3 个组成部分分别是：（1）第 1 层的 Treap 索引结构；（2）第 2 层的高维向量索引结构；（3）监测规则和谓词的关联表。基于 H-Tree 的 3 个主要操作分别是：（1）搜索；（2）插入；（3）删除。H-Tree 总体上是 1 个两层的索引结构。图 1 给出了基于表 1 中的监测规则构建的 H-Tree。如图 1 所示，其第 1 层是由离散型谓词构建的 Treap 树索引；第 2 层是根据连续型谓词对应的多维向量构建的高维空间树；另外 1 个很重要的组成部分是监测规则与谓词的关联表，用来完成两层索引监测规则结果的快速融合。

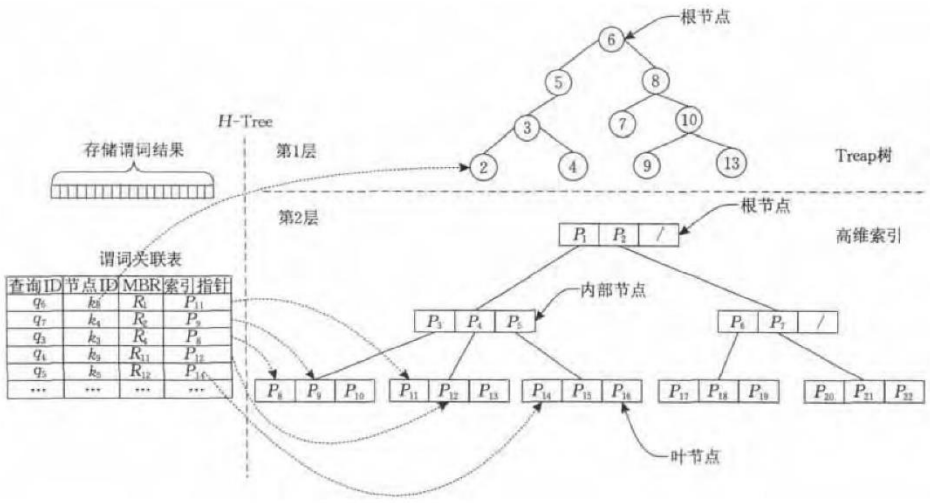

图 1 H-Tree 数据结构

H-Tree 树结构主要有 3 个部分组成。左边是 H-Tree 中存储监测规则与谓词的关联表。右边是 H-Tree 的两层索引框架。第 1 层是 Treap 树，是根据离散型属性上的谓词集构建。第 2 层是高维索引树，是根据连续型属性的谓词构建。H-Tree 中的节点可以分为 3 类：首层节点 top_node，第 2 层的中间节点 mid_node 和叶子节点 leaf_node。节点的主要定义和主要包含元素如图 1 所示。其中，attr 为首层 Treap 树节点对应的离散型属性，value 为该 Treap 节点对应的离散值，weight 为该节点代表的谓词的优先级，left,right 为该节点的左右孩子节点。branch 代表着第 2 层索引对应的高维树结构的中间节点指针。bmr 为第 2 层叶子节点对应的高维向量。

1.3 构建过程

基于 H-Tree 的层次化索引结构，提出一种可行的索引构建算法，共分为 3 个步骤：规则预处理、谓词集合划分和层次化索引构建。先对预处理后的规则集按照属性分类进行一定的划分，在此基础上再对划分后的数据集分层构建索引。规则集划分时，对于预处理后的 n 条规则，按照属性类别和值域划了离散型谓词集 A 和连续型谓词集 B 。其中，有

$$\sum \|A\| + \|B\| = \sum_{q \in Q} p(q)$$

假设将第 1 维上的谓词集分为了 s 个区间，其中每个区间只有离散型谓词或者连续型谓词。由于经过了预处理的维度转换，使得任何一个的区间内含有的谓词性质基本相同或者相似，便于分层次索引的构建。同时，异构数据流注册的规则集通常包含着优先级信息，规则的优先级越高，表示其被监测计算的需求越紧迫。当一个新的监测规则注册到系统中，H-Tree 首先预处理模块将监测规则按照属性类型划分为离散型谓词 p d 和连续型谓词 pc。然后将离散型谓词 p d 插入到 H-Tree 的第 1 层索引中，也就是插入到离散型属性对应的 Treap 树索引中；最后将连续型谓词 pc 插入到 H-Tree 的第 2 层索引中。当将 pd 插入到第 1 层的

Treap 树的时候，首先按照排序二叉树的标准插入法进行插入，但这时可能违反 Treap 树的堆性质，因此需要自底向上进行旋转，直到堆性质得到满足。删除是相反的，先把优先级设置为最低，自上而下转移到叶子，然后删除。H-Tree 的第 2 层插入算法与 R-Tree 的插入过程类似：首先利用 searchLeaf(pc)去定位到要插入的目标叶子节点，定位的过程本身是个递归过程。第 2 层索引插入 pc 的过程从第 2 层的根节点开始，顺次按照广度优先搜索，按照多维空间的包含关系进行搜索，当找到一个叶子节点 n 以后，检查 n 的分支数。如果发现已经超过分支阈值M，则直接进行节点分裂，产生新节点，并将 n 已有的节点和 pc 的向量利用启发式策略平均分派到两个节点中，最后依次更新父亲节点信息。如果 n 的分支数没有超过阈值M，则直接调用 updateParentNode(L)来完成插入操作。H-Tree 插入和删除的具体细节如算法 1 和算法 2 所示。

算法 1. *H*-Tree 的插入算法.

输入：监测规则 q，离散型谓词 p_d，连续型谓 p_c，*H*-Tree *T*, Insert node *n*

输出：NULL

```
1. IF n==NULL THEN
2.   DirectInsert(p_d);
3.   rt←GetRoot(T).
     L←searchLeaf(p_c;T)
4.   IF left branch of L less than M THEN
5.    Update ParentNode of L
6.   ELSE
7.    Split Node L;
8.    Adjust child tree of L;
9.   END IF
10. ELSE
11.  IF the value of p_d is less than n: value THEN
12.   Insert (n: leftChild, p);
13.  ELSE
14.   Insert (n: rightChild, p);
15.  END IF
16. END IF;
```

算法 2. H-Tree 的实时监测算法.

输入：数据流元组 t，索引 H，检索结果 res

输出：NULL

```
1. FOR all 针对当前滑动窗口中的元组 t DO
2.   r←H_first: TreapSearch(t).
3.   IF r is not empty THEN
4.     H_second: Search(t; res);
5.   ELSE
6.     t←Get Next Tuple t;
7.   END IF
8.   IF res is not empty THEN
9.     res←Combine(res; r)
10.  ELSE
11.    t←Get Next Tuple t;
12.  END IF
13. END FOR;
```

需要注意的是：基于离散型谓词的特点，H-Tree 的第 1 层索引用 Treap 树构建，加快离散型谓词的快速监测规则计算。我们假设空树的优先级为无穷大，则上面的添加和删除算法可以正确处理只有一个儿子的情形。第 1 层索引的添加和删除操作的期望时间复杂度均为 O(log n)。从编程复杂度来说，Treap 是工程上最容易实现的数据结构，不仅没有任何特例需要考虑，而且在期望 O(log n)时间内同时支持插入、删除、分离和合并。插入过程最重要的问题是第 2 层索引中节点的分裂策略。在这里本文采用类似 R-Tree 的启发式策略。详细来讲，首先取出所有要分裂的块。然后选择交叠面积最小、同时覆盖两个块的面积最大的两个块。最后将剩余的块按照面积交叠差异依次划归到不同的节点中。由于删除操作与构建操作都比较简单，在此，本文不再赘述。

1.4 性能分析

本节针对提出的 H-Tree 中的实时监测算法和插入、删除算法的时间和空间开销进行分析。在这里，监测规则集合本文用 Q 来表示，监测规则和谓词的映射结构用 C 表示，H 用来表示多属性哈希索引。如算法 2 所示，对于每个实时到来的数据流元组 t，监测规则处理的时间开销可以分为 4 个部分：（1）计算每个谓词结果的时间；（2）检查监测规则和谓词映射关系的时间；（3）检索哈希索引的时间；（4）检索第 2 层高维索引树的时间。为此，本文设计了评估方程，对监测规则处理过程的时间开销进行评估。因此，时间开销的评估方程如下：

$$idx\_searching(H)+sum=\sum_{p\in TP(C)} v(p)\times scanning(p,Q)+multi\_dim(t)\ (3)$$

其中，idx_searching(H)是多属性 Treap 树的搜索开销，scanning(p,Q)代表着对谓词 p 对应的关联表计算的开销，multi_dim(t)代表着 t 在 H-Tree 的第 2 层连续高维索引的检索开销。在算法 2 中，本文做了以下假设：（1）多属性 Treap 树索引的搜索开销 idx_searching(H)与离散型属性个数成正比；（2）scanning(p,Q)的大小与 Q 的数量成正比。基于以上假设，式（3）可以转换为式（4）。

$$sum=|H|\times \log C_{idx}+\sum_{p\in TP(C)} v(p)\times \lambda_p\times \kappa_p\times n\ (4)$$

Cidx 代表多属性 Treap 树的节点数量。λp 代表着谓词 P 在监测规则集 Q 的频率，κp 是谓词 p 的宽度。 n 表示监测规则集 Q 的数量。同时，我们用式（5）对 H-Tree 的空间开销进行评估，空间开销主要由以下 3 个部分组成：（1）第 1 层索引即多属性 Treap 索引的空间；（2）存储谓词与监测规则关系的空间开销；（3）连续型高维索引的开销。

$$sum=\sum_{h\in H} M_{\text{treap}}(h)+\sum_{q\in Q} M_{\text{bitmap}}(q)+\sum_{p\in TP(H)} M_{\text{mapping}}(Q)+M_{\text{dim}}\qquad (5)$$

其中，MTreap 构建多属性 Treap 树索引所需要的空间，Mbitmap 是被用来对所有的监测规则结果进行融合计算的空间。Mmapping 是用来对所有的谓词与监测规则集 Q 的关系进行缓存的空间。Mdim是 H-Tree 的第 2 层高维索引的内存开销。

1.5 相关改进

在以实现大型集团企业的运营状态在线监测为背景的大数据流环境中，尤其是多媒体数据流环境下，数据流包含的属性越来越多，不同属性之间存在差异的异构大数据流也日益壮大，由于监测规则规模异常庞大，传统的高维索引方法对监测规则集合构建的索引性能无法满足实时性的要求。为了解决这个问题，从异构大数据流本身的特点出发，结合不同属性上谓词的特点，本文提出了一种层次化的索引结构(H-Tree)。在 H-Tree 中，首先根据离散型谓词构建 Treap 树作为第 1 层索引；在第 2 层，将每一个连续型谓词映射为多维空间中一维，进而构建多维索引。当数据流元组到来时，首先扫描 H-Tree 的第 1 层触发索引，完成离散型谓词的监测规则；当命中触发索引以后再搜索第 2 层索引。文中方法既充分利用了第 1 层离散型触发索引的快速扫描能力，又借助第 2 层连续型谓词的高维区块划分，将连续型谓词映射为不同的区间块，进一步加快检索速度。在公开数据集和人工数据集上的实验结果表明，在不降低准确率的情况下，H-Tree 的性能优于已有的解决方法。

二、改良的 Patricia 树结构

2.1 简介

针对空间索引响应近邻查询效率低的问题，基于二进制 Morton 码和 Patricia 树，提出一种一维空问索引结构。通过改良 Patricia 树结构及其相关算法提高索引结构的操作率。基于 Morton 码特点，融合索引结构和 Morton 码，使得索引结构拥有高效响应近邻查询的能力，并同时提出基于 MPT 的近邻算法。将二维空间进行预定规则下的不同粒度的划分，把分块后的二维空间区域转换为一维编码，使 MPT 索引具备高效响应区域查询能力。分析区域查询误差出现的原因，并给出相应解决方案。实验结果表明，与 B+树、Hash 表、Trie 树相比，该方法在查询速度上更具优势，基于 MPT 的近邻搜索比基于 R-Tree 近邻搜索效率更高。

2.2 索引结构

使用 Morton 码将二维位置信息转化为一维编码，以 2 个编码相同前缀的长短来表示 2 个二维位置的近远，Morton 码不仅保留了移动对象二维位置的邻近关系，另一方面也能很好地反映不同大小的二维空间区域之间的从属关系。Patricia 树父子结点分别代表的字符串具有前缀对应关系，这也恰好适于 Morton 码前缀匹配操作。基于上述考虑，将移动对象二维位置信息转换为二进制 Morton 码并通过 MPT 插入算法构建索引结构。并改良索引结构和相关算法使 MPT 具有高效响应空间查询的能力。

MPT 的整个结构是以二进制的位操作为原理构建的，这样同时也能节省空间和加快操作效率。因为在计算机中对位的操作更快，所以树将字符串进行“二进制”化再插入到索引结构中。MPT 子结点保存的是插入 MPT 的二进制 Morton 码串，非叶子结点保存的是 Moron 码串前缀对应的“状态”，这些状态是 Morton 码串在插入过程中根据不同 Morton 码串基于位的比较而得到的，只有 2 个码串出现二进制位的不同时，才会出现一个新的结点，来保存这个“分歧”状态，而那 2 个码串也因为这个“分歧”的二进制位的 0 或 1 而由这个结点的 2 个不同的子结点方向各自向下继续比较。也就是说对整个树的操作也就类似于对二叉树的操作。在索引结构中的搜索路径的选择是根据对应位的 0 或 1 来决定的。分析同样具有前缀搜索能力的 trie 树的结构，可以发现 trie 树结点保留字符串中相同的字符，每条检索路径都是字符串的前缀。如果改变结点本身强调的内容，将字符串之间的不同点的信息保留在分支结构中，那么就可以把所有相同的字串匹配省略，减少了所需存储信息的数量。

在保证索引结构操作效率的情况下，使结点中保存的信息尽量少，以减少空间占用。MPT 索引结构如图 2 所示。

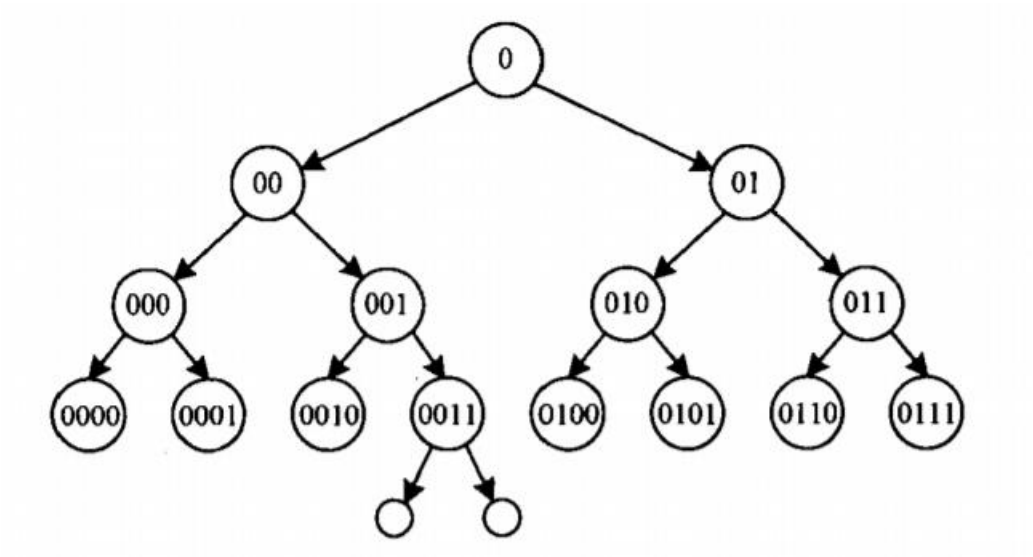

图 2 MPT 结构

非叶子结点结构为(Ptr[2]，pos，mask)，其中，Ptr 是指向子结点的指针，每个结点固定包含 2 个指针；Pos 是结点对应字符在整个字符串中的位置；mask 是一个用于位操作的 8 位二进制码，用来获取比较的字符的最高不同位。叶子结点的结构为(Ptr，Mojlnfo)，其中，Ptr 是指向 Morton 码串的指针；Mojlnfo 是一个结构体数组。每个数据项结构为(ID，X，Y)，其中 ID 为移动对象的标识号；X，Y 为移动对象的二维位置。为了节省空间，本文使用指针变量的最低有效位来表示这个指针是指向内部结点还是指向叶子结点。大致结构如图 2 所示。为了示意，画出了一个“满”MPT，但是实际如果没有出现对应的字符串对比。则不会出现因“分歧”而产生的对应的结点。

2.3 构建过程

整个 MPT 构建过程主要依赖插入算法，由于结构特点，对于同样的 Morton 码集的输入，对应产生 MPT 也是完全相同的，因此 MPT 在操作过程中不需要实现复杂的平衡算法。并且整个 MPT 结构的高度由输入的字符串的最大长度来决定，而不是由输入字符串的数目来决定。插入操作先处理时空树的情况，然后使用类似于上面描述的搜索方法来查到到合适的插入位置，计算相关的 pos 和 mask 参数，然后生成对应新的结点，最终将结点插入。具体算法的伪代码如下：

```
    输入  MPT 结构指针 T,指向字符串指针 u,移
动对象 ID,移动对象二维位置信息 x,y
    输出  True/False
    Begin
     len = strlen(u);//获取输出 Morton 码长度
     node * p = T.root;//获取 MPT 结构根结点
    If(! p)//如果是空树
  以 u 指向的字符串直接构建新结点,并插入
  End If
  While(p 是内部结点)
    对 MPT 结构进行搜索,找到合适插入 u 的结点位置 loc;
     config(pos,mask);//判断loc中存储的Morton码与插
//入u指向的Morton码之间有无区别,如果有,通过位操作计
//算出新插入字符串对应结点中的pos和mask参数,并进行设
//置,如果无则直接返回 False.
     init(newnode);//初始化一个结点的空间,并使其拥有2个
//子结点指针,结点中保存mask和pos信息,结点的2个子结点
//指针其中一个指向移动对象信息构建的子结点,另一个子
//结点指针 p 保留在将 newnode 插入 MPT 的过程中使用
     insert(newnode);//搜索至插入位置newloc,newnode代替
//到 inewloc 中结点位置
      set(newnode);//将 p 另一指针指向原 newloc 处的结点
End.
```

2.4 性能分析

2.4.1 时间效率

将 MPT 与 B+树、Hash 表、Trie 树进行查询效率的比较。将生成的移动对象的位置的经纬度数据转换成 32 位二进制 Morton 码，构建索引结构，进行查询效率的比较。和 B+树不同，MPT 的普通操作，例如查询、插入、删除需要 0(K)的时间(K 表示字符串长度)，而 B+树中这些操作需要 0(109n)的时间(n 为字符串数)，一般情况下 O(k)>O(109n)，但是在 B+树中对于字符串的比较最差情况需要 O(K)的时间复杂度，当字符串的长度较长，则会明显增加 B+树中普通操作所花费的时间，这也是为什么 B+树的效率相对不高的原因。而在 MPT 中，搜索过程中所有的比较操作所消耗的时间都是常数时间。哈希表的插入、查找、删除的时间为 O(1)，但这是以计算哈希值的过程为常数时间当作前提的，当考虑了哈希值的计算时间，则需要 D(k)的操作时间，加上如果在操作过程中发生哈希值的碰撞，则需要消耗更多的时间。而 MPT 则最坏时间为 D(K)。而实验结果也证实了这一点。MPT 比哈希表的效率更高。MPT 相对于

Trie 的优势则体现在 2 种结构的特点，具体原因如 3。2 节所述。实验结果如图 3 所示。由于插入操作和删除操作的大部分时间消耗都是用于查询，因此得到的结果与查询效率对比实验类似，故此不再赘述。

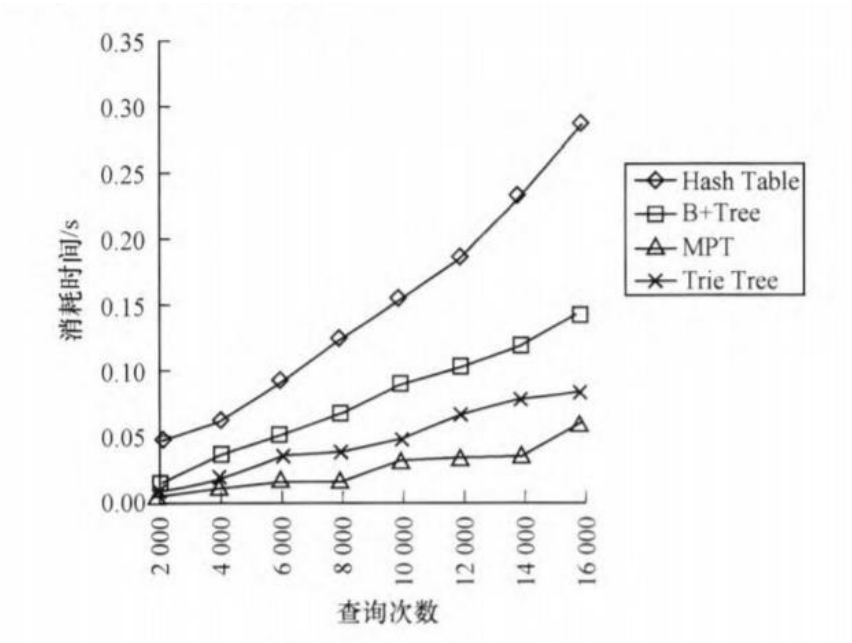

图 3 查询效率比较

其显示了基于 MPT 的近邻搜索算法与基于 R-Tree 的近邻搜索算法效率的对比结果，因为 MPT 的搜索是根据二进制位的比较，而基于 R-Tree 的搜索算法是需要 MBR(Minimum Bounding Rectangle)的比较，相比之下后者需要花费更多时间，并且 R-Tree 存在 MBR 互相重叠的问题，尤其处理大数据量时，基于 R-Tree 的近邻查询算法性能急剧恶化，而 MPT 因其结构的特性并没有上述问题。实验结果表明，基于 MPT 的近邻查询算法效率更高。

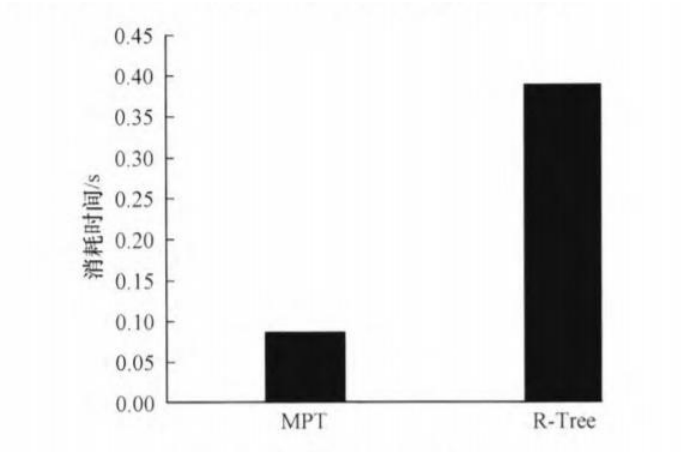

图 4 临近查询效率对比

2.4.2 空间效率

Morton 码的长度越长，则表示的位置越精确，但是使用 MPT 在近邻搜索的

过程中，使用过长的 Morton 码对 MPT 近邻搜索的过程并没有帮助，反而因为过长的码长而导致 MPT 中的结点数过多，所以会产生更多空间的占用。表 1 表示经纬度某一方向经过一定次数的划分后对应的 Morton 码在这一方向上表示的范围。可以看到当划分次数为 15 时，Morton 码在这一方向上的精度为 1223m，就是约为 1km 左右，这时候对应的码长为 30，这样的码长是比较适合作为在交通道路网络中近邻搜索的起始码长的。

表 1　Morton 码精度与划分次数对应表

划分次数	对应精度/m	划分次数	对应精度/m
1	20 037 726.370 00	11	19 568.09
2	10 018 863.190 00	12	9 784.04
3	5 009 431.593 00	13	4 892.02
4	2 504 715.796 00	14	2 446.01
5	1 252 357.898 00	15	1 223.00
6	626 178.949 10	16	611.50
7	313 089.474 50	17	305.75
8	156 544.737 30	18	152.87
9	78 272.368 63	19	76.43
10	39 136.184 32	20	38.21

上表显示了使用不同码长的 Morton 码所构建的 MPT 索引所占的内存大小，可以发现在码长为 30 左右的时候，当插入 MPT 移动对象数据量到达一定的时候，索引所占的空间增长明显减缓，这是因为索引结构中大量减少了重复信息的存储的缘故。基于这样的存储优势，MPT 索引结构可以在应对大量移动对象信息的情形下使用。

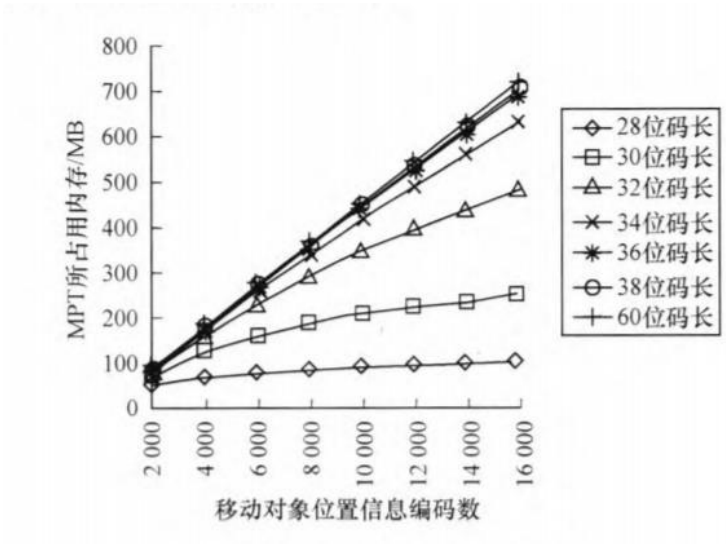

图 5　移动对象位置信息编码数

2.5 相关改进

在 Morton 和 Patricia 树的基础上提出一种一维索引结构 MPT，通过改良 Patricia 树结构，使得在索引结构操作上有着更好的性能，并且合理利用 Morton 码编码方式，使索引结构拥有比 Geohash 更加细粒度地区域划分精度的控制,同时提出基于 MPT 的近邻算法,其性能优于基于 R-Tree 的近邻搜索算法。通过将二维空间合理划分，并将划分的区域编码,使索引结构具有高效响应区域查询的能力。因为考虑到近邻查询和区域查询都是实时性较强，所以暂时并没有考虑时间因素，下一步将把时间、位置信息编码成一维信息，使得索引结构能够得到更广泛的应用。

参考文献

[1]易显天,徐展,郭承军, 等.基于 Patricia 树的空间索引结构[J].计算机工程,2015,41(12):69-74. DOI:10.3969/j.issn.1000-3428.2015.12.014.

[2]侯雄文.一种改进的四叉树索引结构[J].科教导刊-电子版（下旬）,2018,(11):266.

[3]李正欣,张凤鸣,李克武,张晓丰.一种支持 DTW 距离的多元时间序列索引结构[J].软件学报,2014,25(03):560-575.

[4]臧文羽,李军,方滨兴,谭建龙.H-Tree:一种面向大数据流在线监测的层次索引[J].计算机学报,2015,38(01):35-44.