

Histogram 树在海量 LiDAR 点云查询中的设计与应用

刘海澄¹, 关雪峰², Martijn Meijers¹, Peter van Oosterom¹

1. Faculty of Architecture and the Built Environment, Delft University of Technology, 2628 BL, Delft, the Netherlands;

2. 武汉大学 测绘遥感信息工程国家重点实验室, 武汉 430079

摘要: 高精度点云作为除矢量与栅格数据外的第三类空间信息数据被越来越广泛地采集与应用, 然而其快速增长的庞大数据量却让现有数据管理方案遭遇瓶颈。近期有学者提出基于空间填充曲线(SFC)编码的 B+树管理方案, 即将点云的多个组织维度如 XYZ, 采用 SFC 编码合并为一个键值, 然后在此键上建立 B+树以达到对数级查询效率。此方法查询时的一个关键步骤是将查询窗口从原始的多维空间转换为 SFC 键值的一维范围段, 再利用 B+树取出范围段内的点输出。传统的 SFC 范围计算方法基于数学原理等分空间, 并未考虑点云的实际分布, 故而产生的范围段冗余, 有些并不含数据, 导致查询时间长、结果粗糙。针对此问题, 本研究提出首先根据点云构建 Histogram 树, 统计点云在空间的分布, 在查询时利用 Histogram 树生成 SFC 范围段, 使查询效率提高。研究设计并实现了两种 Histogram 树构建算法, 在测试后发现考虑点云形状特征的混合构建算法在内存与时间消耗上明显优于自底向上的构建算法。点云查询测试验证了使用 Histogram 树在保证结果精度的情况下, 可有效降低 SFC 范围段数目, 提升整体查询效率。后期会在此基础上继续优化, 根据检索出的叶节点采用数学方法进行更深层搜索, 提高查询精度。可以预期, 面对点云信息量越来越大, 步入 N 维空间后稀疏化、不均匀化的问题, Histogram 树会更加发挥其优势, 指导查询更高效地进行。

关键词: 海量点云, Histogram, LiDAR, 查询算法, 空间填充曲线, Morton 编码

中图分类号: P208 **文献标志码:** A

1 引言

近年来, 高精度激光扫描仪的普及以及多波束回声测深仪和 3D 摄像机等新一代采集设备的出现促进了点云数据的生产。例如 Fugro 等公司从 2008 年到 2013 年采集制作第二代覆盖荷兰全境的机载激光点云数据产品(AHN2), 总量达 6400 亿点(Wijga-Hoefsloot, 2012; Van Oosterom 等, 2015)。2018 年 1 月, 美国国家标准与技术研究院(NIST)初步制定了一项公共安全研究计划(NIST, 2018), 通过采集室内点云构建室内点云标准模型, 认为点云会成为下一代室内应用的基础。与此同时, 基于点云的应用不断增加, 例如无人驾驶和虚拟现实, 提出了更智能和高效的点云数据管理需求。然而现有工具难以达到所需: 目前大多数点云管理是基于文件系统, 例如 TEXT、LAS/LAZ, 甚至供应商自定义的文件格式, 完成排序和索引则需要创建另外的数据结构。数据库主流产品如 Oracle、PostgreSQL 所采取的主要解决方案有平表和块式存储两种: 平表是将每个维度存为一列数值; 块式存储是将点云打包压缩为块, 之后再母表对块进行索引。但如 Liu 等(2018)和 Van Oosterom 等(2015)指出, 平表存储方式在查询时会进行全表扫描, 效率低下, 而块式存储的解

压过程相当耗时。

研究人员为此探索更高效的管理方案, Psomadaki(2016)收集了荷兰海岸带监测的用户需求, 开发了基于 Oracle Index-Organized Table (IOT)的大型动态点云管理方案, IOT 本身采用 B+树结构, 以叶节点存储 X/Y/Z 或 X/Y/Time 的 Morton 编码, 另外的维度以附加属性值存储。结果表明最采用 X/Y/Time Morton 编码的方案在各类查询中综合表现最优, 此外该方案具有良好的扩展性。Guan 等(2018)通过对空间与细节层次维度进行 4D Hilbert 编码并用 Oracle IOT 管理, 同样证实该方案可以高效查询大规模点云数据。Van Oosterom 等(2015)在 PostgreSQL 上进行的基准测试也表明基于空间填充曲线(SFC)编码的 IOT 方案相对平表方案查询效率大幅提升。此类方案要实现查询, 需有转换器将查询窗口的多维空间范围转换为一维 SFC 键值范围段。上述研究均是通过对点云的最小包围框进行迭代剖分并与查询窗口进行相交计算, 直到达到某层深度然后返回相对应的 SFC 范围段用于查询。Meijers 等(2018)通过基准测试指出低于或超过最优剖分深度, 查询效率均会降低, 最优深度则需要通过不断试验得到。通过探讨, 我们指出若能实现表征点云分布的 Histogram 树, 则可望借助其使查询自动达到最优深度并提高效率。

基金项目: 国家留学基金, Fugro BV 研究基金

第一作者简介: 刘海澄(1991—), 男, 博士, 现从事海量点云的数据结构设计和应用研究。E-mail: H.Liu-6@tudelft.nl

2 SFC 查询转换器原理

SFC 是指一条贯穿多维空间中所有基础单元仅一次的曲线。SFC 可以保持其覆盖的地理实体的空间邻近关系，已经广泛用于地理信息数据索引 (Nivarti 等, 2015; Huang, 2014)。以 Morton 曲线 (也称为 Z 阶或 N 阶曲线) 为例, Morton 键的计算通过交错各维度二进制坐标的位来实现。假设一个点坐标为(3,2), 其二进制表示为(11,10)。通过位交错, 即可得出 Morton 键 1101(十进制 13), 它对应曲线上的第 14 个节点(图 1)。依次方法, 所有点根据 Morton 键排序聚合, 空间相邻的点在 Morton 曲线上也相距不远。通过控制二进制位长度, 可以构造不同的层级。比如图 1 左下角框中的四个点的起始两位都是 0, 这样截断末尾两位, 在更高层级就可以用键值为 0 的一个点代表。因此空间填充曲线同时也可以表达多层次空间划分, 即 2^n 叉树结构, n 为维数。

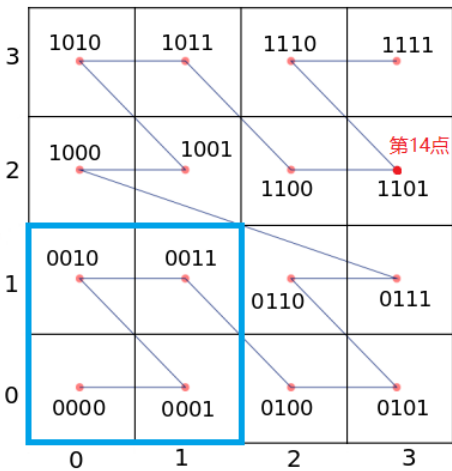


图 1 二维 Morton 曲线
Fig.1 2D Morton curve

计算出键值后, 所有键, 亦即点, 会通过 B+树进行管理, 比如 Oracle 数据库提供的 IOT(图 2), 它采用 B+树结构将数据存在叶节点中, 以普通表的形式呈现, 使用键值组织与索引。其主要特点是索引与数据结合在一起, 无需单独构建索引结构, 这会节省空间及提高查询效率。

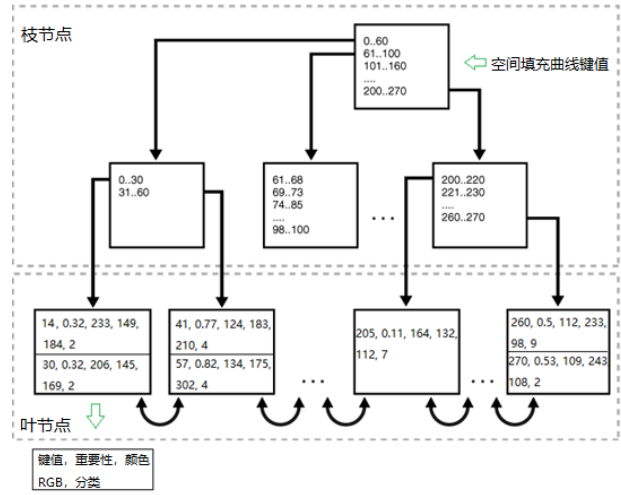


图 2 空间填充曲线索引组织表
Fig.2 The SFC IOT structure

SFC B+树管理方法的查询需用借助转换器, 将查询窗口从原始的多维空间映射到 SFC 一维空间(图 3), 之后再一维的 SFC 范围段放入 SQL 查询的 WHERE 条件中进行查找。以往研究对查询窗口递归分解, 利用多层空间填充曲线单元逼近查询窗口范围, 整个分解过程不考虑目标数据分布。以图 3 为例, 算法首先判断整个数据集窗口是否与查询窗口相交, 若相交则把数据集窗口均分为四块, 即四叉树结构, 再判断每块与查询窗口关系, 依此递归进行, 直到达到某种预定深度。

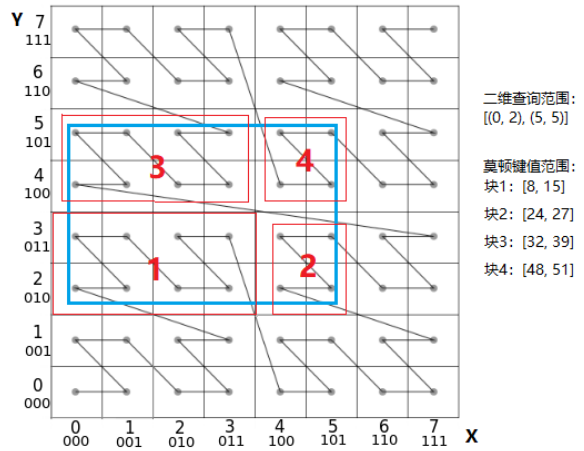


图 3 查询范围转换

Fig.3 The transformation of query range

Guan 等(2018)和 Meijers 等(2018)普遍反映此过程相当耗时。由于点云空间分布的异质性, 此转换算法往往会导致 SFC 范围的无效计算, 例如图 3 中为得到块 4 范围会迭代 3 步, 而若其中并无数据,

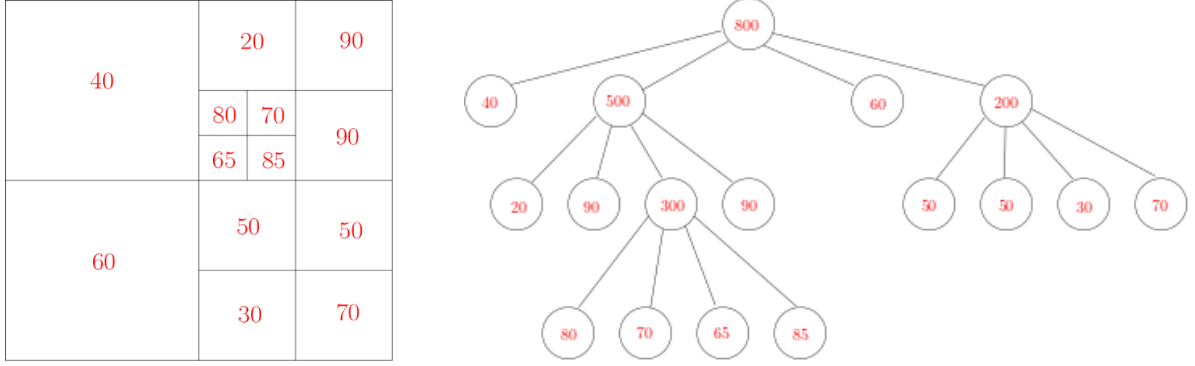


图 4 二维点云 Histogram 树构建示例, 阈值为 100

Fig.4 A histogram building example for 2D point cloud, where the threshold is 100

则计算并无实际价值, 反而浪费时间。另外无效计算还会导致产生多余的 SFC 范围段, WHERE 语句中过多的范围条件也会使查询效率下降。为了提高转换过程的效率, 我们决定构建数据点的 Histogram 树以反映点的分布以避免多余的范围计算和查询(图 4、图 5)。主要应用思路是将 Histogram 树作为二级索引结构:

1. 在数据加载期间构建, 在 Oracle SQL loader 中实现。
2. 采用树结构进行存储。
3. 进行查询时, 首先根据 Histogram 树计算范围段, 然后再与原始数据表 JOIN 进行选择。
4. 更新数据表后, Histogram 树会自动更新。
5. Histogram 树应具有通用性, 可在 N 维点云上构建。

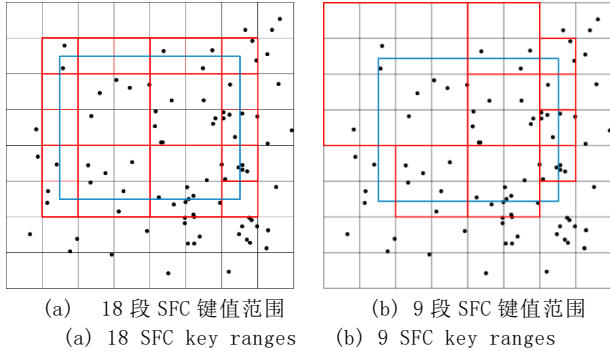


图 5 使用 SFC Histogram 树的预期效果, 蓝色框为查询窗口

Fig.5 The expected improvement by adopting the histogram tree to compute SFC ranges; blue box indicates the query window

3 Histogram 树算法设计

Histogram 树构建的整体思路是递归统计每个单元格中点的数目, 如小于阈值则停止计算, 该单元节点即为叶节点, 反之则继续递归统计, 最后构建出变深度树。树节点的键是 SFC 值, 存储的则是

点的数目(图 4)。查询时从根节点依次向下搜索整个树, 根据查询精度确定搜索深度, 最优精度取决于构建所用阈值。

3.1 构建算法

算法 1 HistCompBottomUp($PCL, DimNum, t$)

输入: 点云坐标列表 PCL , 点云的维度数目 $DimNum$ 和阈值 t

输出: Histogram 树 L

```

1:  $L \leftarrow \emptyset$ 
2: while not end of  $PCL$  do
3:   initialize HistNode{key, child, neighbor, pnum, height} ▷ 节点数据
   结构, 包括SFC键值、一个子节点、一个兄弟节点、节点内包含的点数
   目、节点在树中的高度, 均初始化为0
4:    $L$  insert (HistNode.key, HistNode) ▷ 容器插入键值对
5:   read next point  $P$ 
6:   while key_prefix  $\neq 0$  do
7:      $PL \leftarrow \emptyset$  ▷ 父节点容器
8:     for HistNode  $\in L$  do
9:        $p\_key \leftarrow$  cut last DimNum bits of HistNode.key ▷ 父节点键值
10:      if  $p\_key = PHistNode.key$  in  $PL$  then
11:         $PHistNode.pnum ++$ 
12:         $HistNode.neighbor \leftarrow PHistNode.child.neighbor$ 
13:         $PHistNode.child.neighbor \leftarrow HistNode$ 
14:      else
15:        initialize PHistNode ▷ 创建父节点
16:         $PHistNode.key \leftarrow p\_key$ 
17:         $PHistNode.child \leftarrow HistNode$ 
18:         $PHistNode.pnum \leftarrow HistNode.pnum$ 
19:         $PHistNode.height \leftarrow HistNode.height + 1$ 
20:         $PL$  insert ( $p\_key, PHistNode$ )
21:     for  $PHistNode \in PL$  do
22:       if  $PHistNode.pnum < t$  then
23:         release all children of  $PHistNode$ 
24:      $L \leftarrow PL$ 
25:     key_prefix  $\leftarrow L[0].key$ 
26: return  $L[0].HistNode$ 

```

图 6 Histogram 树自底而上构建算法

Fig.6 The bottom-up algorithm to build histogram tree

前期调研得出两种方法: 一种是比较直观的自上而下的构建过程, 即从根节点开始, 逐层统计直到达到阈值, 此算法每一层都会把所有点遍历, 上层的统计结果对下层并无帮助, 故而时间复杂度为

$O(\log N)$, N 为点数目; 另一种构建方法是从原始数据开始, 自下而上逐层统计, 上层结果基于下层, 时间开销为 $O(N)$, 伪代码如图 6 所示。然而在实现时我们发现, 由于原始点云数目过大, 从最底层构建树会消耗大量内存, 故而提出了一种自中间层开始向上构建的混合算法, 伪代码见图 7。

算法 2 HistCompHybrid($PCL, DimNum, scaling, t$)

输入: 点云坐标列表 PCL , 点云的维度数目 $DimNum$, SFC 编码时点的扩展系数 $scaling$ 和统计直方图阈值 t

输出: Histogram 树 L

```

1:  $PointNum \leftarrow$  point number of  $PCL$ 
2:  $MBB \leftarrow$  minimum bounding box of  $PCL$ 
3:  $TrueDimNum \leftarrow$  true dimension number of  $PCL$  ▷ 统计学方法
4: for  $i \leq DimNum$  do
5:    $DimLength[i] \leftarrow$  length of dimension in  $MBB$ 
6: for  $j \leq TrueDimNum$  do
7:    $Area \leftarrow Area \times DimLength[j]$ 
8: if  $PointNum/Area < 1$  then
9:    $BaseHeight \leftarrow \text{floor}(\log_2 \frac{Area}{PointNum}/TrueDimNum)$ 
10:  $ThresholdHeight \leftarrow \text{floor}(\log_2 t/TrueDimNum)$ 
11:  $StartHeight \leftarrow BaseHeight + ThresholdHeight$  ▷ 起始构建高度
12:  $L \leftarrow \emptyset$  ▷ 中间层
13: while not end of  $PCL$  do
14:   create HistNode{key, child, neighbor, pnum, StartHeight}
15:    $L$  insert ( $HistNode.key, HistNode$ )
16:    $SFCL$  insert  $SFCencode(P)$ 
17:   read next point  $P$ 
18: for  $HistNode \in L$  do
19:   if  $HistNode.pnum > t$  then
20:     FixPool insert HistNode ▷ 插入修复容器
21: for  $SFCP \in SFCL$  do
22:    $p\_key \leftarrow$  cut last  $DimNum \times StartHeight$  bits of  $SFCP$ 
23:   if  $p\_key \in FHistNode.key$  in FixPool then
24:     create HistNode{SFCP, child, neighbor, pnum, height}
25:      $FHistNode.child \leftarrow HistNode$ 
26: for  $FHistNode \in FixPool$  do
27:   build child layers according to  $FHistNode.child$  and  $t$ 
28:  $RootNode \leftarrow HistCompBottomUp(L, DimNum, t)$ 
29: return  $RootNode$ 

```

图 7 Histogram 树混合构建算法

Fig.7 The hybrid algorithm to build histogram tree

最佳的中间层应保证绝大部分节点包含的点数目未超过阈值, 其计算的关键在于获取点云的实际维度。考虑点云在空间的分布不均匀, 比如机载 LiDAR 点云多分布在地表层, 故而曾现二维特性, 这样在构建父节点层时, 点云的实际聚合度为 4 而非 8。即虽然 Histogram 是八叉树结构, 但实际构建出的的树中基本所有节点的子节点数目都不超过 4。通过第一遍读取数据获取点云数目, 最小包围框等统计信息, 我们采用统计方法确定点云的实际维度。经测试发现, LiDAR 点扫数据均呈现面特性, 在三维空间的分布依然是由面决定, 大部分 LiDAR 点云的实际维度为一(电力线或杆状结构)或二。之后通过第二遍读取原始点云构建中间层节点, 此时的中间中会有少量节点超过阈值。超过阈值的节点会做

修复, 通过第三遍读取原始数据, 找出中间层过载的节点涵盖的所有原始点并加载到内存中, 并建立指针联系, 之后即可快速对过载节点由上而下构建出 Histogram 子树, 如图 7 中 18~27 行所示。最后再由中间层自底而上构建出完整树结构。

点云的维数会从元数据表中获取, 元数据表提供整套 SFC B+树管理方案所需的必要参数, 还包括维度单位、SFC 类型、数据存储类型、参考系等信息。算法实际输出为 Histogram 树的根节点, 通过它可以访问整个树结构。现阶段整个树储存在内存中, 并无导出。

3.2 查询使用

构建出 Histogram 树后, 查询时会从其根节点开始, 通过判断枝节点是否与查询框相交, 逐层深入, 图 8 为查询伪代码。

算法 3 Polyquery($P, DimNum, HistRoot$)

输入: 采用边界表达形式的任意 N 维多面体查询窗口 P , 点云的维度数目 $DimNum$, 以及 Histogram 树的根节点 $HistRoot$

输出: 涵盖 P 的 SFC 范围段列表 $SFCL$

```

1: SearchTree insert  $HistRoot$  ▷ 搜寻路径
2: while SearchTree  $\neq \emptyset$  do
3:    $HistNode \leftarrow SearchTree.pop()$  ▷ 取末尾节点并在栈中删除
4:    $cell \leftarrow$  bounding box of  $HistNode$  ▷ 根据键值及节点高度得出
5:   Compute spatial relationship between  $cell$  and  $P$  ▷ 光线投射算法
6:   if  $cell$  inside  $P$  then
7:     RawSFCL insert ( $SFCencode(cell_{LL}), SFCencode(cell_{UR})$ ) ▷ 左下及右上角 SFC 编码, 以元组形式插入
8:   else if  $cell$  intersects  $P$  then
9:     if  $HistNode.child \neq \emptyset$  then
10:       SearchTree insert all children of  $HistNode$ 
11:     else
12:       RawSFCL insert ( $SFCencode(cell_{LL}), SFCencode(cell_{UR})$ )
13:   else
14:     continue
15:  $SFCL \leftarrow$  merge adjacent ranges in RawSFCL ▷ 合并相邻 SFC 范围段
16: return  $SFCL$ 

```

图 8 利用 Histogram 树做 N 维窗口查询

Fig.8 Utilizing the histogram tree to perform a query confined by an nD window

查询算法中采用光线投射法是考虑到不规则查询窗口的情况, 以往研究的矩形框查询是一种特殊情况, 只需对比节点和矩形框的端点即可。由于在迭代时计算路径分离, 初始计算返回的范围段(图 8 中 RawSFCL)有可能是相邻的, 所以在最后部分再进行合并操作, 以减少数目, 提高后续根据范围段查找数据表的效率。算法搜索至叶节点层, 但同样可以选用其他搜索限制条件, 比如覆盖点数大小。

4 测试与分析

本研究使用荷兰全国机载点云数据 AHN2 对以

表 1 Histogram 树构建方案效率对比
Table 2 Comparison of efficiency of Histogram tree construction using different approaches

| 级别 | 输入点数目 | 输入文件大小 (MB) | 自底而上算法 | | 混合算法 | |
|----|---------------|-------------|-------------|----------|-------------|----------|
| | | | 最大内存消耗 (MB) | 时间开销 (s) | 最大内存消耗 (MB) | 时间开销 (s) |
| 小 | 14,665,157 | 333 | 4,583 | 123 | 709 | 25 |
| 中 | 138,865,433 | 3,137 | 43,387 | 1,223 | 2,144 | 241 |
| 大 | 1,414,908,540 | 32,438 | - | - | 18,005 | 1,916 |

上算法进行测试。所选样本为荷兰西南角(图 9)，坐标系为 Amersfoort / RD New, EPSG:28992，数据最小包围框为 (13427.64, 363052.95, -2.33; 19999.99, 380252.64, 56.69)，共包含 1,414,908,540 个点。测试平台为一台 HP DL380p Gen8 服务器，RHEL6 操作系统，配有 16 核 Intel Xeon E5-2690 处理器，主频 2.9Hz，主存 138GB，硬盘为 41 TB SATA 7200 rpm，配置为 RAID6 磁盘阵列。

4.1 Histogram 树构建测试

为了解不同数据大小下构建 Histogram 树资源消耗的差异，共设置了三组数据样本，输入大小间约为 10 倍差距，均是从总样本中切割得出，其中最大样本即为总样本 (见表 1)。表中最大内存消耗指构建 Histogram 树过程中占用的最大内存，而 Histogram 树本身内存占用会小很多。输入为文本文件，只存储 XYZ 坐标值。两种算法实现时均采用节点池技术，即底层的节点在聚合生成父节点时若因点数过少而被抛弃，则此节点放入节点池中供循环使用，这样可充分利用内存并减少时间消耗。

由表 1 可见，自底而上算法最大内存消耗超过数据本身大小近 10 倍，并且随着输入数据的增加，内存与时间消耗也呈线性增加。最后一组测试由于占用内存超过服务器容量，故而没有进行。其内存与时间消耗均明显大于混合算法。另一方面，混合算法虽一开始内存使用较大，但由于起始构建高度为 7，省略了 Histogram 树底部的绝大部分节点，故而内存消耗明显降低。虽然混合算法的时间复杂度依旧为 $O(N)$ ，但同样由于省略了树底层部分的合并计算，实际构建时间显著降低，如表 1 所示，总样本的 Histogram 树构建时间在半小时左右。

4.2 点云查询基准测试

通过对总样本 AHN2 点坐标进行伸缩变换，即坐标扩大 100 变为整数，再对三个维度进行 Morton 编码可得到三维 Morton 键，最后使用 Oracle 创建 IOT 表，过程如下：

1. CREATE TABLE AHNSFC (SFC NUMBER)
2. 使用 SQL Loader 将 Morton 键值载入 AHNSFC 表
3. CREATE TABLE AHNIOT (SFC, CONSTRAINT xyz_PK PRIMARY KEY

```
(SFC) ORGANIZATION INDEX AS
SELECT SFC FROM AHNSFC
```

研究构建了四种方案，分别为平表、SFChist、SFCplain13 和 SFCplain14。后三种方案均基于 AHNIOT 表，分别采用不同算法计算出查询窗口对应的 Morton 键值范围，然后存入表 RANGE_PACKS，再与 IOT 表结合查找结果。SFChist 采用算法 3 (图 8) 计算范围，查询时最深搜索到叶节点层，返回对应的范围段。SFCplain 采用传统的纯数学方法，13、14 指从根节点向下搜索的层数，显然层数越多，结果越精确。测出的时间仅是返回 SFC 键值的时间，并不包括解码，具体如下：

平表：

```
数据表: CREATE TABLE AHNFLAT (X NUMBER,
Y NUMBER, Z NUMBER)
```

```
查询: CREATE TABLE QUERY_RES AS SELECT
X, Y, Z FROM AHNFLAT WHERE (X BETWEEN...
AND... AND Y BETWEEN...)
```

SFChist:

```
数据表: AHNIOT
```

```
查询: CREATE TABLE QUERY_RES AS SELECT
/*+ use_nl (t r)*/ t.SFC FROM AHNIOT
t, RANGE_PACKS r WHERE (t.SFC BEWTEEN
r.LOWER and r.UPPER)
```

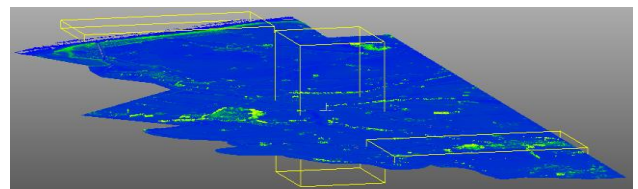


图 9 查询窗口，从右向左：城镇、郊区、海岸
Fig.9 Query windows, from left to right: urban, rural and coastal area

SFCplain13 与 SFCplain14 执行方法 SFChist 相同。为了解不同数据分布情形下查询效率的差别，我们设置了三个查询窗口，分别涵盖城镇、郊区与海岸地理区域(图 9)。设置城镇区域主要考虑点云分布更加趋于三维，除去了大部分地面点($Z > 1.5$)，以及把最高海拔值设为 100 (表 2)。郊区窗口查询实际为二维查询，不设 Z 选择范围，但由于 Morton 编码基于三维坐标，故而人为设置海拔从 -999 到 999，点云在此三维空间分布极为不均，主要集中在地面。海岸地区点云分布同样分布不均，在 XY 平面有一

半区域没有点云，但设置了海拔的取值范围减小 Z 方向的不均匀度。三个区域包含的实际点数都维持在 3×10^7 左右。

表 2 查询窗口范围

| 地理区域 | 查询窗口 |
|------|--|
| 城镇 | (16902.29, 365439.3, 1.5; 19999, 367189.4, 100) |
| 郊区 | (16664.54, 370486.56, -999; 17997.76, 372036.45, 999) |
| 海岸 | (15281.52, 378658.19, -10; 18320.86, 380248.44, 100) |

查询测试结果如表 3 所示，平表返回的结果即为真值，由于 Histogram 树的阈值设置，以及传统 SFC 方法搜索层级限制，其返回点数均会多于真值。我们采用式(1)描述相对误差(Liu, 2009)，其中 y 为真值， y' 为近似查询结果：

$$Error = \left| \frac{y - y'}{y} \right| \quad (1)$$

在城镇查询中，利用 Histogram 树查询所得结果显著偏大，相对误差为 65%。SFCplain13 返回范围段数目与 SFChist 近似，但假阳性点数目过多，其相对误差达到 200%，说明返回的 SFC 范围段长度过大，经查验，其根本原因在于 Z 坐标数值较小，在 Morton 键值中的影响力过小，因此返回了很多 Z 坐标在 1.5 以下的假阳性点。另外由于不考虑数据分布，很多范围段内部并无点，过多的范围段并没有起到实质作用，徒增时间消耗。SFCplain14 由于搜索更深一层，故而范围段更加精细，其结果更趋于真值，相对误差仅 0.06%，但由于其范围段数目过大，写入时间增长，故而其 SFC 范围计算时间消耗较大。

郊区查询结果证明了在点云分布不均时，使用 Histogram 树会显著降低 SFC 范围段数目，而 SFCplain13 与 14 由于生成很多空范围段，增加磁盘读写时间，故而总体上增大了时间消耗。三种 SFC 方案返回的结果均与真值差距不大，相对误差在

0.22%与 0.36%之间。查询时间相似，结合城镇查询结果，可以明显得出，在此输入数量级，范围段数目对查询时间影响不大，返回点数目是决定因素，关系着读写时间。

海岸查询窗口同样因为点云分布不均，Histogram 树的使用会使范围段数目显著减少，但由于有了海拔限制，SFCplain13 与 14 较郊区查询返回的范围段数目大幅降低。

总体而言，在此输入数量级下，使用 Histogram 在去除结果假阳性点数目上所起作用不大，经查验，其叶节点业主要分布在第 13 和 14 层上，对应的 Histogram 树高度为 8 和 7(树总高度为 21)。比如在城镇查询时，SFChist 返回的所有叶节点中，在第 13 层的数目占 40%，在第 14 层的比例达 52%，故而其结果精度介于 SFCplain13 与 SFCplain14 之间。之所以城镇查询结果较郊区与海岸查询更为粗糙，是因为落在城镇查询窗口边缘上的节点数目过多，而这些节点全都囊括了查询窗口以外的临近点。郊区窗口因为 Z 方向的边界无点数据，海岸窗口也只是一部分边缘与数据集相交，故而落在查询窗口边缘的节点数远远减少，所以返回结果也更为精确。若要验证 SFChist 在查询精度上优于 SFCplain，则点云在空间的分布须更加不均，换言之，AHN2 分布还是相对均匀。Histogram 树的现有优势主要表现在降低范围段数目，减少了其在生成时的读写时间，从而整体上降低了查询时间。平表方案的主要时间消耗是搜索与结果写入，可通过在 SFChist 方案最后加设全局过滤功能，使其返回准确结果，那么其整体性能会超过平表。

5 结 论

针对海量点云数据 SFC B+树管理方案中的查询窗口维度转换问题，本研究提出构建 Histogram 树记录点云空间分布，并用其辅助计算 SFC 范围段，从而提高点云查询效率。

研究设计并实现了两种 Histogram 树构建算法，实验证明，混合算法在内存与时间消耗上比理论确定最优的自底而上构建算法显著降低，在十亿点的输入数据以及 100 的阈值设置下，内存使用减少 10

表 3 查询方案性能对比

| 地理区域 | | 平表 | SFChist | SFCplain13 | SFCplain14 |
|------|---------|--------------|--------------|--------------|--------------|
| 城镇 | 返回点数 | 32, 857, 103 | 54, 732, 006 | 98, 579, 615 | 32, 875, 576 |
| | 范围段数目 | - | 272, 799 | 264, 276 | 883, 980 |
| | 时间消耗(s) | 54.19 | 32 + 19.42 | 38 + 30.59 | 163 + 10.9 |
| 郊区 | 返回点数 | 32, 308, 240 | 32, 424, 323 | 32, 465, 984 | 32, 380, 793 |
| | 范围段数目 | - | 2, 668 | 189, 251 | 2, 080, 974 |
| | 时间消耗(s) | 50.5 | 1 + 9.23 | 26 + 10.15 | 110 + 10.08 |
| 海岸 | 返回点数 | 31, 874, 797 | 31, 963, 855 | 32, 016, 089 | 31, 919, 992 |
| | 范围段数目 | - | 2, 464 | 52, 228 | 222, 443 |
| | 时间消耗(s) | 55.14 | 0 + 10.75 | 21 + 9.14 | 83 + 9.4 |

注：时间消耗 = 范围段生成时间 + 查询时间

到 20 倍，速度提升约 5 倍。

在查询测试中，Histogram 树现阶段体现的主要优势是有效减少 SFC 范围段数目，降低读写时间，从而总体提高查询效率。然而其返回结果精度仍需继续提升，而与之作为对比的传统的 SFC 查询方法在搜索层级设置合理的情况下，结果精度表现良好。因此得出 Histogram 树最重要的作用在于给出点云的大致分布描述，而在具体查询时，若查询窗口与数据集相交边缘过长，可在树的叶节点处继续向下采用数学方法计算范围段，从而提升结果精度。照此方向，生成 Histogram 树的阈值可以适当增大，这样亦可减少内存消耗。阈值的选取也受应用的影响，若应用对精度要求较高而点云分布不均，则阈值要尽量降低，更精细的描述点云分布。受 AHN2 测试启发，另一种提升查询结果精度的方法是在 SFC 编码时通过伸缩变换，使得各维度取值范围大致相等，比如将 AHN2 的 Z 坐标扩大 100 倍，提升其在 Morton 键值中的选择力，达到减少假阳性点的目的。

本研究虽基于 LiDAR 点云，但可以扩展到广义点云数据，比如轨迹、数理模型抽象等。随着融入信息的增加，点云会向 N 维发展，比如加入时间维度、重要性维度，点云的分布可能会更加稀疏与不均，这样采用 Histogram 树的作用会更加明显，在未来的工作中，将会继续对此情形进行探索。此外，本研究的输入点数在十亿级，当其继续增大，现有的混合构建算法消耗亦会成倍增加，这就需要分块构建 Histogram 树，并采用并行方式增加构建效率。未来工作还包括实现 Histogram 树随数据表变化的自动更新功能，查询测试时尝试不规则查询窗口，以及分布式环境下的构建与查询实现。

参考文献 (References)

- Guan X, Van Oosterom P and Cheng B. 2018. A parallel N-dimensional Space-Filling Curve library and its application in massive point cloud management. *ISPRS International Journal of Geo-Information*, 7(8): 327-345 [DOI: 10.3390/ijgi7080327]
- Huang Y K. 2014. Indexing and querying moving objects with uncertain speed and direction in spatiotemporal databases. *Journal of Geographical Systems*, 16(2): 139-160 [DOI: 10.1007/s10109-013-0191-6]
- Liu H, Van Oosterom P, Meijers M and Verbree E. 2018. Management of large indoor point clouds: an initial exploration. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, 42(4): 365-372 [DOI: 10.5194/isprs-archives-xlii-4-365-2018]
- Liu Q. 2009. Approximate query processing. *Encyclopedia of Database Systems*, New York, USA: Springer, 113-119 [DOI: 10.1007/978-0-387-39940-9_534]
- Meijers, M. and van Oosterom, P., 2018. Clustering and indexing historic vessel movement data with space filling curves. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, 42(4): 417-424 [DOI: 10.5194/isprs-archives-XLII-4-417-2018]
- NIST. 2018. Notice of funding opportunity (nofo) NIST public safety innovation accelerator program (PSIAP) – point cloud city. Technical report, National Institute of Standards and Technology. [2019-04-26]. https://www.nist.gov/sites/default/files/documents/2018/01/05/2018-nist-psiap-pc2_nof.pdf
- Nivarti G V, Salehi M M and Bushe WK. 2015. A mesh partitioning algorithm for preserving spatial locality in arbitrary geometries. *Journal of Computational Physics*, 281: 352-364 [DOI: 10.1016/j.jcp.2014.10.022]
- Psomadaki S. 2016. Using a space filling curve for the management of dynamic point cloud data in a relational DBMS. Master's thesis, Delft University of Technology, the Netherlands.
- Van Oosterom P, Martinez-Rubi O, Ivanova M, Horhammer M, Geringer D, Ravada S, Tijssen T, Kodde M and Goncalves R. 2015. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers & Graphics*, 49:92-125 [DOI: 10.1016/j.cag.2015.01.007]
- Wijga-Hoefsloot M. 2012. Point clouds in a database: Data management within an engineering company. Master's thesis, Delft University of Technology, the Netherlands.

The design and application of histogram trees for querying massive LiDAR point clouds

LIU Haicheng¹, GUAN Xuefeng², MEIJERS Martijn¹, VAN OOSTEROM Peter¹

1. Faculty of Architecture and the Built Environment, Delft University of Technology, 2628 BL, Delft, the Netherlands;

2. Liesmars, Wuhan University, Wuhan 430079, China

Abstract: Other than vectors and rasters, high accurate point clouds, as the third spatial data type, are now being more and more widely collected and utilized. However, its fast increasing volume causes bottlenecks for data management. Researchers recently proposed a state-of-the-art data management solution. That is using Space Filling Curve (SFC) to encode several organizing dimensions such as XYZ of point clouds into a key, and then building a B+ tree on the keys to achieve logarithmic query efficiency. A critical process during querying is to convert the query window from original multidimensional space into one dimensional SFC ranges, so that later we could utilize B+ tree to efficiently retrieve points inside the SFC ranges. However, conventional method is purely based on mathematics to equally split the space to get the ranges without considering the distribution of points. Consequently, the SFC ranges generated are excessive and some contain no data, which leads to long query process and rough result.

To address this problem, we propose to first build a histogram tree for the point cloud to record the distribution of points in the space. Then we use the histogram tree for generating SFC ranges, which could improve the query efficiency. In this paper, two algorithms to build the histogram tree were designed and implemented. After tests, we found that the hybrid algorithm which took the shape of point clouds into account cost less memory and time compared with the bottom-up algorithm. Point cloud querying benchmark tests confirmed that the use of the histogram tree can effectively reduce SFC ranges generated while the accuracy was satisfactory. Hence the total time cost was decreased. Optimization will be made in the future to improve the query accuracy, specifically by searching deeper from the leaf nodes retrieved. It could be expected that with more information involved in the point clouds, points will be more sparse and unevenly distributed in an nD space. The histogram tree would then become more indispensable to guarantee an efficient querying process.

Key words: massive point clouds, Histogram, LiDAR, query algorithm, space filling curve, Morton

CLC number: P208

Document code: A

本论文_____参加_____第五届全国激光雷达大会优秀青年论文评选。第一作者的身份证号码为：_370881199104231118_（参加优秀青年论文评选者填写）。