基于Spark的3D点云数据空间索引技术

赵尔平[1] 孟小峰[2]
(西藏民族大学信息工程学院 陕西 咸阳 712082)[1] (中国人民大学信息学院 北京 100872)[2]

摘 要 针对Spark引擎不支持多维空间查询的问题,提出基于R树的二级空间索引,即在每个Worker节点上创建R子树,并将这些子树作为孩子,在Master节点上创建R树。针对LRU算法内存替换粒度粗、结果不够精确的问题,提出基于数据使用权重的内存替换方法。该方法将每次实际使用数据量与其总量的比值作为替换权重,将热点场景数据以RDD形式持久化至内存中,提高了基于内存查询的效率。根据远粗近细的视觉原理提出细节层次查询,该方法将最能代表物体特征的点云数据先传输给客户端,或者仅把简化模型点数据传给客户端,以解决网络带宽不足和数据加载延迟的问题。实验证明,文中方法能有效解决Spark多维空间的查询问题,查询效率得到了明显提高。

关键词 Spark,多维空间索引,3D点云数据,数据使用权重,细节层次,虚拟旅游
中图法分类号 TP392 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.09.035

Spatial Index of 3D Point Cloud Data Based on Spark

ZHAO Er-ping[1] MENG Xiao-feng[2]
(School of Information Engineering, Xizang Minzu University, Xianyang, Shaanxi 712082, China)[1]
(School of Information, Renmin University of China, Beijing 100872, China)[2]

Abstract Two level spatial index based on R tree was presented according to the problem that spark engine doesn't support multi-dimensional spatial query, that is, the R subtree is created on each worker node, and these subtrees are used as children to create the R tree on the master node. Memory replacement granularity of LRU algorithm is coarse, and the result is not accurate enough. For this reason, the method of memory replacement based on data usage weight was proposed. The ratio of actual used amount of data and its total amount is used as replacement weight. The method stores the hot scene data in RDD form into memory and improves the query efficiency based on memory. According to the visual principle of far thick and near fine, the level of detail query was presented. The point cloud data that best represent the object characteristics are firstly transmitted or the simplified model data are only transmitted to the client, so as to solve the problem of insufficient network bandwidth and data loading delay. Experimental results show that the proposed method can effectively solve the problem of multi-dimensional spatial query on spark, and the query efficiency is improved obviously.

Keywords Spark, Multi-dimensional spatial index, 3D point cloud data, Data usage weight, Level of detail, Virtual tourism

三维激光扫描技术作为一项成熟的技术,已被用于许多领域内的三维建模,如文物古迹保护、虚拟旅游、数字城市、建筑设计、三维GIS、三维测量等[1]。它是利用三维激光扫描仪采集被扫描物体表面海量密集点的三维坐标、颜色(RGB)、激光反射强度和法线等信息,即3D点云数据。三维激光扫描仪每秒可采集百万级数量的点云数据,机载扫描仪更快[2],荷兰3D点云模型由大约6400亿个点云数据组成[3]。由此可见,3D点云模型的数据量非常庞大,某些模型的数据量已达到TB级;加之3D点云数据具有离散、无拓扑结构及海量的特点,数据管理与索引受到了极大的挑战,成为了数据库领域新的研究热点。

人们通常采用索引机制来提高点云数据的查询效率,如k-d树、八叉树、R树和R^+树[4-6]等索引结构。为了降低网络带宽的有限压力,缓解客户端长时间等待数据加载的问题,研究者还将细节层次(Level of Detail,LOD)技术引入点云数据索引[7-9]机制中。这些方法在一定程度上提高了点云数据的查询性能,但是它们都是基于传统的集中式数据管理方式,面对海量点云数据,集中式系统的硬件性能一直是其查询的瓶颈,即使采用各种索引机制,性能提高也非常有限,依然不能满足多用户交互和大规模数据的实时查询需求。目前,通用类型的大数据采用分布式文件系统HDFS(Hadoop Distributed File System)进行存储管理。HDFS是一种高容错、高可

到稿日期:2017-08-04 返修日期:2017-11-13 本文受国家自然科学基金(61762082),西藏自治区自然科学基金(12KJZRYMY07)资助。
赵尔平(1976-),男,硕士,副教授,主要研究方向为数据库技术,E-mail:xdzep@163.com(通信作者);**孟小峰**(1964-),男,博士,教授,博士生导师,CCF会员,主要研究方向为云数据管理、Web数据管理、闪存数据库和隐私保护等。

靠、动态负载均衡、多个节点协同工作的文件系统[10]，仅允许追加新数据操作，不允许更新数据操作[11]，适合构件在廉价计算机集群上，为用户提供底层透明的访问机制，具有管理TB甚至PB数量级数据的能力。Spark是继Apache开发的Hadoop之后的又一大数据开源框架，它超越了点燃大数据革命的Hadoop/MapReduce[12]；而且Spark是基于内存数据处理的计算引擎，采用弹性分布式数据集(Resilient Distributed Datasets，RDD)作为内存数据结构，利用多个计算任务共享数据的策略来避免重复读写数据，特别适合应用于交互式查询、机器学习和数据挖掘等领域。

1 研究现状

近几年，基于Hadoop框架的空间数据索引研究逐渐增多。Whitman等[13]利用PMRquadtree作为Hadoop框架的空间索引结构，将空间数据划分为相等的数据集后分配在集群节点上，并为它们分别建立局部quadtree索引，按照空间位置关系将每个节点的本地quadtree构建成一个完整的全局PMRquadtree索引树。Zhong等[14]在利用Hadoop框架处理地理空间大数据时，提出了基于四叉树全局索引和希尔伯特排序局部索引的二级空间索引结构，数据分块后被均匀部署在集群节点上。SpatialHadoop[15]是明尼苏达大学在Hadoop上开发的大规模空间数据处理平台，采用R树作为二级空间索引，实现了按空间范围查询的功能，但它与文献[14]一样，仅支持二维空间查询，不支持多维空间查询。

虽然Spark引擎在某些应用中搜索数据的速度比Hadoop快很多倍，但它也存在一个局限，即未能提供高级索引功能的存储引擎，仅能顺序扫描HDFS文件获取数据[16]，在面对非常庞大的数据集时搜索速度也不能满足人们的要求，于是人们在Spark框架上引入索引机制以加速它对海量数据搜索的速度。文献[17]为了提高从海量高光谱影像数据中抽取端元的速度，把像元纯度指数算法作为Spark框架的索引算法。网格索引方法SFTGridIndex[18](Spark-Fat-Thin-Grid-Index)将遥感影像每个子分区包围的所有多边形保存到分区文件中，给每个分区文件创建记录(唯一分区编号和区域范围等信息)并写入索引文件。该方法避免了大量多边形的求交运算，提高了大规模遥感影像数据的查询速度。

上述研究成果都是通过给大数据处理引擎增加索引来提高它们的查询性能。但是，目前已有的研究成果都是在Hadoop引擎上增加空间索引结构，且仅支持二维空间查询而不支持多维空间查询，在Spark引擎上实现空间查询的研究成果比较罕见，有待继续探索。由于R树创建、插入、删除和更新等操作的实现相对简单，因此其成为了目前公认的、理想的空间索引结构，被广泛应用于商业系统。例如，Oracle数据库企业版于2007年增加了基于R树的空间索引功能，SQLserver2008支持基于R树的空间索引功能[19]。基于以上原因，本文在Spark引擎上增加了基于R树的二级空间索引结构，使Spark能快速处理多维空间查询，并解决三维虚拟旅游、数字城市和三维GIS等系统因采用集中式数据管理而造成多用户交互漫游时存在查询性能瓶颈的问题。

2 Spark实现空间索引的优势

与传统平台相比，Spark在实现空间索引方面具有很多优势：1)传统平台上的空间索引采用程序设计的编程规范；而Spark使用RDD编程范式，实现方便、规范。2)传统平台空间索引是基于集中式管理的，数据量与访问峰值是其软肋；Spark采用HDFS分布式文件系统，多台机器负载均衡与并行处理，非常适合于多用户交互查询系统。3)Spark强大的内存计算能力能使点云数据实现基于内存的查询，允许中间结果保存在内存，此特点可以使相同的后续查询直接在内存中完成。4)Spark支持大数据延迟计算，有助于查询结果在内存中进一步优化处理。5)HDFS之上的应用程序必须按流式方式访问数据，此优点恰好与3D点云漫游系统需要以流式方式传输数据吻合，由于3D点云模型的数据量非常庞大，必须采取一边传输一边渲染即流式传输的方式。6)虚拟旅游、数字城市和三维GIS等3D模型建成后一般不再变动，点云数据一旦被写入HDFS文件就不会再被修改，与HDFS文件一次写、多次读的特性相吻合；Spark是既可在内存中又可在外存上执行操作的引擎，因此它特别适合处理数据量远大于集群内存总容量的点云数据。

文中给Spark引擎增加了基于R树的二级空间索引结构，其架构如图1所示。R树作为全局索引(第一级)部署在Spark的Master节点上，它的每棵R子树作为局部索引(第二级)部署在集群各个Worker节点上，Master将3D点云数据均匀分配给每个Worker节点，局部索引则有效地组织并管理各自节点上的点云数据，全局索引则管理所有数据。

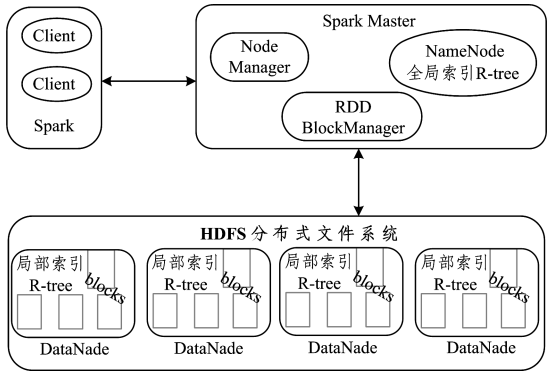

图1 具有空间索引机制的Spark架构

Fig. 1 Spark architecture with spatial indexing mechanism

3 基于R树的二级空间索引结构

二级空间索引结构是指在Worker节点上创建局部索引R子树，以这些子树作为孩子在Master节点上创建R树，R树作为全局索引。按照独立场景数据分割和数据等分的负载均衡原则，将3D点云数据均匀分配给每个Worker节点，并为它们创建对应的R子树，进一步创建整棵R树。

3.1 R树在磁盘上的定位

R树的非叶子节点存储它的所有子节点的索引记录；叶子节点存储它的所有外接立方体的索引记录，这些外接立方体代表互不相交的空间区域。父节点空间区域完全覆盖所有子节点的空间域。索引需要记录数据集在磁盘上的位置，这

样在处理查询时就可以快速定位到目标数据，避免额外的 I/O 操作，但是 HDFS 为用户提供了底层透明的访问机制，应用程序事先无法定位数据在磁盘上的具体位置，导致 R 树索引不能直接将它管理的数据集定位到具体的磁盘页面。HDFS 默认将文件分割成多个 block 块存储，除了最后一个 block 块外，每个块均为 64MB，可以借助 block 块在磁盘上的位置来间接实现 R 树节点与磁盘页面的对应关系，即利用 R 树的每个叶节点固定管理 HDFS 文件中的一个 block 数据块的方法间接实现 R 树节点与其在磁盘上所管理数据的映射关系，从而实现 R 树索引在磁盘上的定位。R 树中规定它的每个节点独占一个磁盘页面，因此每个节点存储的索引记录条数基本相等，其扇出计算公式为：

$$fan=\frac{B_{\text{disk}}}{B_{\text{record}}} \tag{1}$$

其中，B_{disk} 为集群机器的磁盘页面大小，B_{record} 为一条索引记录所占的字节数。由于 R 树兄弟节点空间重叠引起的多路查询会造成其索引性能降低，尤其是对于形状不规则、数据量大的物体，最坏情况下 R 树的性能退化为顺序查询。若构造 R 树前能对数据进行适当预处理，打破空间重叠条件，其性能将得到大幅度提高。

3.2　点云模型的空间剖分

为了使 R 树兄弟节点管理的点云数据空间区域不重叠，构建 R 树前先要对其管理的 3D 点云数据进行去重叠处理，常用方法是利用八叉树对 3D 模型按预设阈值进行空间剖分。R 树不支持按点查询，因为其仅维护多维数据边界框和指向实际数据的指针[20]，所以 R 树是典型的按范围查询。从根节点到叶节点逐层匹配空间区域的重叠对象，R 树的查询精确度由其叶子节点外接最小边界矩形 MBR（Minimal Bounding Rectangle）的覆盖范围决定，MBR 可以扩展到多维空间，如三维为立方体。为了保证 R 树既有一定的查询精确度，又能充分利用磁盘碎片空间，查询粒度的选取不宜太大，规定 R 树叶节点的每个外接立方体包围 64 kB 点云数据。将八叉树空间剖分阈值预设为 64 kB，利用它的快速收敛性对 3D 模型数据进行空间剖分，最终得到若干个互不重叠的连续空间区域。经过这样的预处理后，就能确保 R 树的兄弟节点空间区域不重叠，解决因多路查询导致的索引性能下降的问题。

3.3　二级空间索引的创建

批量读取八叉树叶节点数据，让八叉树一个叶节点管理的数据对应 R 树叶节点的一个外接立方体，并计算每个外接立方体 *MBR* 的空间范围 I，I 可以用它的左下角（$X_{\min}$，$Y_{\min}$，$Z_{\min}$）和右上角（$X_{\max}$，$Y_{\max}$，$Z_{\max}$）坐标表示。将这些立方体管理的数据批量、有序地写入集群某个 Worker 节点对应的 HDFS 文件中，构造这些外接立方体的索引记录（I，*tuple-identifier*），其中 *tuple-identifier* 表示某个立方体覆盖的数据在 HDFS 文件系统中的起始位置，然后把索引记录批量插入 R 树叶节点。由于 Spark 库中没有提供创建 R 树的方法，因此需要在 Spark 中增加 CreateRtree 函数，实现 R 树的创建功能。插入前，若 R 树的某叶节点的索引记录数大于扇出 *fan*，或者其管理的数据大于 64 MB，则该叶节点先分裂，将其插入新分裂的节点。若叶节点索引记录条数满足[*fan*/2，*fan*)，则计算这个叶节点的所有外接立方体合并后的范围，构建这个叶节点的索引记录（I，*child-pointer*）并将其插入父节点，其中 *child-pointer* 为指向该叶节点的指针。为了充分利用磁盘空间，R 树每个节点的索引记录条目至少要大于 *fan*/2。如此重复来构建这个 Worker 节点的 R 子树，依据负载均衡条件结束该节点的 R 子树的创建，同理创建集群中其他 Worker 节点的 R 子树。若集群中每个 Worker 节点的 R 子树创建完成，则计算每个 Worker 节点的 R 子树覆盖的空间域范围，构建每个 Worker 节点的索引记录，同理将这些索引记录插入 Master 节点，完成 R 树的创建。创建二级空间索引的具体算法描述如算法 1 所示。

算法 1　二级空间索引 R 树创建算法

输入：3D 模型的点云数据 D

输出：二级空间索引 R 树

```
1. partition(D,τ)/* D为点云数据,τ为剖分阈值*/
2. if(τ< 64)
3. return {d1,d2,d3,…,di,…} /* d1∪d2∪di∪…∪dj∪…=D*/
4.    else return dj∪partition(D-dj,τ)
5.       j=1
6. for(i=1;i=n;i++) /* n为集群Worker节点个数*/
7.    { do
8.       {CreateRtree(Ri) /*创建第i个R子树*/
9.       j++
10.      SaveAsObjectFile(Di)/*点云数据写入HDFS文件*/
11.      Di=Di-dj}/*Di为分配给第i个Worker节点的数据*/
12. While(Di=Φ)}
13. CreateRtree(R1,R2,…,Rn)/*创建全局索引R树*/
14. return R-Tree
```

4　Spark 引擎实现 3D 点云数据的查询

3D 点云数据查询是典型的基于范围的查询。由于 Spark 函数库没有空间区域匹配函数，因此需要在它的函数库中增加空间范围查询函数 RangeQuery 和空间连接函数 SpatialJoin。RangeQuery 的功能是根据查询立方体的左下角和右上角坐标来计算其范围内所包含的点数据，结合二级索引快速定位到数据块在磁盘的起始位置，然后批量读取这些数据。SpatialJoin 函数的功能是将多个输出文件合并成一个文件，并将这个文件作为并行查询的输出结果。

4.1　基于二级空间索引的查询

二级空间索引查询首先将索引结构载入集群内存，根据用户查询范围请求利用 RangeQuery 函数从全局索引 R 树的根节点到子节点逐层匹配空间域重叠的节点，由这些节点索引记录转到一个或多个 Worker 节点上的局部索引 R 子树。RangeQuery 函数利用局部索引 R 树继续按空间范围进行搜索，匹配与查询请求空间域重叠的所有叶子节点，由这些叶子节点的索引记录对应到 HDFS 文件相应的 block 数据块，从中获取与用户请求空间域重叠的点云数据，将它们批量读入内存，并用 SpatialJoin 函数将它们连接成一个数据块，对其进行一定优化处理后发送到客户端。如果查询任务被分解在多个 Worker 节点上执行，则多个 Worker 节点能够并行处理，因此基于 R 树的二级空间索引的查询时间由两部分组成：一

部分是 Master 节点分配查询任务的时间和数据收集时间;另一部分是由多个 Worker 节点并行完成子空间查询任务时开销最大的 Worker 节点的时间花费。基于二级空间索引查询的代价估算公式为:

$$T_Q = T(Q_M) + \mathrm{Max}\{T_i(Q_S)\} \quad (2)$$

因为查询任务是面对外存操作,查询过程需要频繁进行 I/O 操作,式(2)中的时间开销主要是完成子任务查询的花费,并且由承担最大查询任务的 Worker 节点决定。索引机制是 Spark 框架降低查询代价的途径之一,利用 Spark 共享内存的特性,若查询前将部分数据从外存提前载入集群内存来实现基于内存空间的查询也是提高其查询性能的另一种有效途径。

4.2 点云数据 RDD 的创建与查询

虽然通过二级空间索引能有效提高 Spark 引擎的查询速度,但是外存查询存在大量 I/O 操作,而查询代价主要来自于 I/O 访问时间,因此索引结构提高的空间查询性能非常有限,不能满足多用户交互式的查询要求。

弹性分布式数据集是 Spark 的核心组件,是基于内存的共享模型,支持高容错和并行操作的数据集,允许用户将重复操作的数据集以 RDD 形式持久化至内存或磁盘中,并对数据进行分区(Partition)以实现并行操作;编程人员可以在集群上实现基于内存的操作。RDD 数据集有助于提升 Spark 处理大数据的速度,已被广泛应用于多种大数据处理场景。4.1 节提到若能将点云数据提前载入集群内存,则能大大提高查询性能,因此利用 RDD 共享内存的特性,尽可能把更多的点云数据以 RDD 形式持久化至集群内存,后续查询直接在内存 RDD 数据集上进行,从而提高查询效率。RDD 的创建方式主要有基于 HDFS 文件、已有 RDD、数据库和数据流等。虚拟旅游、数字城市等 3D 点云漫游系统的空间查询属于数据密集型操作而非计算密集型操作,与机器学习、数据挖掘等操作会产生大量中间计算结果不同,基于内存的空间查询仅进行点数据的三维坐标匹配操作,不存在其他复杂计算,利用已有 RDD 创建子 RDD 会重复占用集群内存。例如:RDD_1 点数据集的空间区域为 V_1,查询请求 Q_2 的空间区域为 V_2,且 $V_2 \subseteq V_1$,显然基于 Q_2 查询结果创建的 RDD_2 包含于 RDD_1。如果后续用户再次提出 Q_2 查询请求,那么他们在 RDD_1 数据集上查询 Q_2 与在 RDD_2 数据集上查询 Q_2 的代价几乎相同,因为他们都是在内存中进行空间区域匹配的相同操作。但是 RDD_2 需要额外占用集群内存,减少了其他点云数据提前载入内存的机会,这种情况与点云数据庞大而集群内存有限的状况矛盾。为了节省集群内存,让更多点云数据以 RDD 形式持久化在内存中,必须避免产生重复 RDD,本文不通过已有 RDD 衍生新 RDD,而是利用 HDFS 文件创建。用户借助二级空间索引快速从 HDFS 文件查询相关点云数据,对其进行处理后发送给客户端,同时利用这些查询结果创建对应的 RDD,用 cache() 方法将该 RDD 加入 cache 列表;然后 CacheManager 把该 RDD 分区的 Partitions 放入 BlockManager 中进行管理,作为后续多用户交互式查询复用的内存数据。

定义 1 V 代表集群内存 RDD 数据集表示的空间范围,用户查询请求窗口的范围为 V_u,如果 $V_u \subseteq V$,则利用 RDD 数据集实现基于内存的空间查询,否则利用基于 R 树的二级空间索引在外存的 HDFS 文件中实现空间查询。

基于内存空间查询的时间开销主要由读取数据代价和数据处理代价两部分组成,其查询代价的估算公式为:

$$T_q = T_{\mathrm{read}} + T_{\mathrm{process}} \quad (3)$$

基于内存空间查询时,被查询数据以 RDD 形式存在于内存中。若数据集在内存中,则读取数据的代价可以忽略[21],因此基于内存查询代价中读取数据的代价为零,即 $T_{\mathrm{read}} = 0$。查询代价 T_{process} 包括空间域匹配和查询结果后续处理时间,而这些操作都是在集群内存中进行的,时间花费非常少。因此,利用 RDD 数据集实现基于内存空间的查询能够大幅提高查询效率。但是,与海量 3D 点云数据相比,集群内存空间非常有限,将其全部载入内存不现实,在多用户交互式查询中存在某次查询请求的数据不在或不全在集群内存从而不能实现基于内存查询的情况,所以文中空间查询操作需按定义 1 中定义的内、外存查询两种情况分别处理。

定义 2 将每次实际使用的数据量与总量的比值称为数据使用权重。

4.3 基于数据使用权重的内存替换算法

点云数据提前载入内存可以最大化地加快查询速度,但是 Spark 集群内存远远不够容纳海量点云数据,查询时需要不断地进行内存数据的替换,这给 Spark 内存管理带来了挑战。Spark 采用最近最少使用算法 LRU 替换内存中的 RDD[22],这是一种粗粒度替换策略,仅考虑了 RDD 数据集的最近使用频率,没有考虑每次真正使用的 RDD 数据量在整个 RDD 数据集中所占的比重,即数据使用权重,因此 LRU 替换算法不够精细。例如,RDD_1 被任务 A 和任务 B 各调用 1 次,任务 A 使用 RDD_1 10% 的数据,任务 B 使用 RDD_1 30% 的数据;相反,RDD_2 仅被任务 A 调用 1 次,但是使用了 RDD_2 80% 的数据。很显然,RDD_2 数据的实际使用权重大于 RDD_1,但是按 LRU 替换算法淘汰的是 RDD_2 而非 RDD_1。假设 RDD_1 和 RDD_2 数据集被查询前都存储在外存且大小相等,那么任务 A 和任务 B 使用 RDD_1 时有 40% 的数据需通过 I/O 操作获取,而任务 A 使用 RDD_2 时有 80% 的数据需通过 I/O 操作获取,显然操作 RDD_2 读数据的代价大于 RDD_1,再次证明了替换 RDD_2 不可取。鉴于 LRU 算法评价策略过粗,造成 Spark 管理内存申请与释放不够精准,本文提出基于数据使用权重的内存替换算法,该算法能更精细、更准确地选取使用率最低的 RDD 并将其淘汰出内存。

RDD 创建后系统会按 Partitioner 属性值对其进行分区,这些 Partitions 被系统分配在不同的 Worker 节点上实现分布式并行操作。为了准确地统计 RDD 在某个时间段的数据使用权重,需要先计算它的每次数据使用权重,每次数据使用权重之和便是某个时段该 RDD 的数据使用权重。由于 RDD 每次被调用时实际使用的数据量不方便统计,但它的 Partition 被分配在不同的 Worker 节点上,每次调用的 Partition 数量比较容易统计,因此用每次实际使用的 Partition 个数与该 RDD 的 Partition 总数的比值来近似计算该 RDD 本次数据的使用权重。用 N 表示 RDD 的 Partition 总数,假设它在

某个时间段被调用 m 次，那么该 RDD 在这个时间段的数据使用权重的计算公式为：

$$W_R=\sum_{i=1}^{m}\frac{n_i}{N} \tag{4}$$

其中，n_i 为第 i 次实际使用的 Partition 个数。根据数据使用权重内存替换算法，当集群内存不足时，比较每个 RDD 的数据使用权重，将权重值小的 RDD 先淘汰出内存。该方法的内存替换粒度小，替换结果精确，集群内存的使用效率高，能把大多数用户感兴趣的热点场景的 3D 点云数据以 RDD 形式持久化于内存中。

4.4　细节层次查询

3D 模型的点云数据非常庞大，网络带宽又非常有限，数据传输到客户端时需要较长的时间。对此，人们寻求了多种技术手段，其中 LOD 技术是解决途径之一。LOD 技术是指在不影响视觉效果的前提下，根据景物在场景中所处的位置和重要度，借助多种技术简化景物表面的细节来降低场景的几何复杂性，让最能代表景物特征的简化模型的点云数据先传输、先渲染，使得系统和网络资源合理分配，相对减少数据加载时间，现已被广泛应用于虚拟现实和可视化等领域。点云数据自带了激光反射强度属性，构成物体凸面、光滑面的点云数据的激光反射强度值相对较大，光线能最先进入人的眼线，它们最能代表物体的轮廓与特征；相反，构成物体凹面、粗糙面的点云数据的激光反射强度值相对较小。利用这个光学原理，将查询结果数据集按激光反射强度属性值先进行排序，使查询结果中激光反射强度值大的点数据先传输，相反，值小的点数据后传输，从而实现连续细节层次的渲染，让用户尽早体验到景物轮廓，消除了用户长时间等待数据加载的烦恼。

定义 3　漫游时，相机最高视点距离为 L，N 为 3D 点云模型 LOD 的层次数，$l=\lfloor L/N \rfloor$，相机视点高度用 c 表示，则 $c\in[L-l,L)$ 对应模型的最低细节层次，$c\in[L-2l,L-l)$ 对应模型的次低细节层次，…，$c\in(0,L-N\times l)$ 对应模型的最详细节层次。

定义 4　空间查询结果按点数据激光反射强度属性值降序排序后的数据集为 D，其包含的点数据个数用 M 表示，若相机视点处于 $c\in[L-l,L)$ 高度范围，则选取 D 集中前 $\lfloor 1/N\times M \rfloor$ 个点数据传输给客户端；若相机视点处于 $c\in[L-2l,L-l)$ 高度范围，则选取 D 集中前 $\lfloor 2/N\times M \rfloor$ 个点数据传输给客户端，…；若相机视点 $c\in(0,L-N\times l)$ 高度范围，则选取 D 集的全部数据传输给客户端。

漫游过程中不同用户的漫游方式有所不同，有的用户喜欢远视点、大范围粗略浏览，有的用户喜欢近视点、小范围细致漫游。根据远大近小、远粗近细的视觉原理，远视点对应场景简化细节模型，近视点对应场景详细细节模型。如果把模型细节层次数设置为 3，由定义 3 可知，漫游时相机视点高度可以对应模型的低、中和最高 3 个细节层次。利用 Spark 强大的内存计算功能，按查询结果点数据集的激光反射强度值对其进行降序排序并统计点数据个数，根据定义 4 实现细节层次查询和数据传输，例如远视点漫游时只需传输大约 1/3 的点数据，中距离视点漫游时传输约 2/3 的点数据，近视漫游时则传输全部点数据。如果用户由远到近同范围漫游，则要对已经传输简化模型的点云数据进行过滤，以避免给客户端重复传输数据。传统的 LOD 技术远视点传输简化模型，近视点时再传输另一个详细模型，由于不同的简化模型之间存在数据冗余，因此传统的 LOD 技术存在大量数据被重复传输，增加了网络负载。文中方法是真正意义上的 LOD 技术，有效地解决了漫游系统网络带宽不足的问题。当然 LOD 技术是在不影响视觉效果的前提下使用的，代表各个细节层次的数据不一定等分，例如可以设置远视点传输 1/2 的数据甚至更多。细节层次数也不一定设置为 3 层，这些因素需要根据具体的 3D 模型、用户需求、视觉效果综合考虑。

5　实验结果与结论

实验环境选用 5 台服务器搭建 Spark 集群，其中 1 台服务器作为 Master 节点，另外 4 台作为 Worker 节点，每台机器内存均为 16 GB，存盘页面为 64 kB。利用 TrimbleVX 激光扫描仪采集不同场景的 3D 点云数据，利用由 TrimbleRealWorks 三维重建软件构建的 3D 模型作为实验数据。

5.1　基于索引与无索引的查询性能分析

通过比较基于索引和无索引两类查询效果来验证文中二级空间索引的性能。在不同数据集上进行多次查询并取其平均值，这两类查询都是直接在 HDFS 文件中进行的，属于外存查询。为了确保每次查询都能在外存中进行，选择的所有查询必须都是在系统中首次进行，并且每轮实验结束时需重置系统。由文中创建的 RDD 方式可知，每次查询结束后都必须利用查询结果数据集创建 RDD，如果每轮实验结束后不重置系统，清理上轮查询结果数据在内存中创建的 RDD，那么下轮查询的数据便在集群内存中存在，由定义 1 可知下轮查询就会按基于内存查询的方式进行。实验的平均查询性能如图 2 所示，无索引查询的时间开销比借助索引查询的时间多出大约 4～10 倍，因为无索引查询每次都要从头顺序扫描 HDFS 文件，直至搜索到查询目标，此过程中的若干次 I/O 操作需要花费大量时间，因此查询代价非常大。无索引查询开销对数据量非常敏感，它会随着数据量的增加快速变大。相反，借助索引可以准确且快速地定位到查询目标，即使数据量增加，其对查询代价的影响也很小。实验结果证明，二级空间索引结构提高 3D 点云数据查询性能的效果显著。

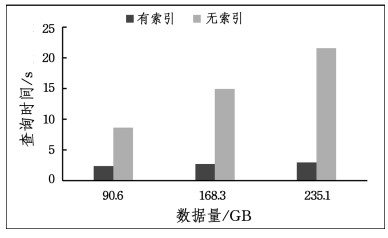

图 2　有索引与无索引时平均查询性能的比较

Fig. 2　Comparison of average query performance with Index and without index

5.2　基于 Spark 内存查询的性能分析

定义 1 规定若查询数据已在集群内存，则实现基于内存的查询，否则利用基于 R 二级空间索引在 HDFS 文件中查

询,并把在 HDFS 文件查询的结果创建为新 RDD。由于此次实验要求所有查询都是基于 Spark 集群内存的查询,即在持久化于集群内存的 RDD 数据集上查询,若把 5.1 节的实验在 3 种数据集上设计的查询分别再次执行,便能得到本次实验的结果。实验中设置 Spark 集群内存的 60%用于缓存 RDD 数据集,平均查询开销如图 3 所示。基于内存查询的整体时间开销都非常小,实验中大部分基于内存空间查询的开销都在毫秒级。

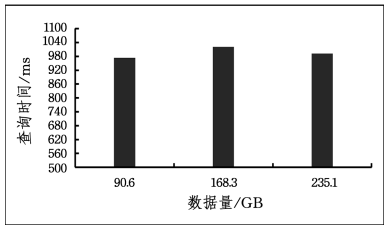

图 3 基于 Spark 集群内存的平均查询性能的比较

Fig. 3 Comparison of average query performance based on Spark cluster memory

5.3 内存大小对查询性能的影响

Spark 集群内存分为保存 RDD 的缓存和运行内存,通过 setProperty 方法设置 spark. storage. memoryFraction 的值来调整缓存大小,实验将缓存占内存的大小分别设置为 50%,60%和 70% 3 种。由图 4 所示的测试结果可知,利用 Spark 框架实现 3D 点云数据空间查询时,集群内存分配的最佳方案是缓存分配 60%,运行内存 40%。缓存分配 70%时执行内存仅剩 30%,此时集群内存自身已是查询任务执行的瓶颈,因此查询任务从开始就处于高代价状态;当缓存接近饱和状态的 70%时,集群的网络带宽又成为查询的瓶颈,造成查询开销突然变大,使得后续查询任务的开销一直处于高位状态。缓存为 50%,60%两种情况时的执行内存都充足,内存不会造成查询瓶颈,始终以较高的效率执行查询任务。缓存分配 60%时,点云数据的查询性能最好,因为缓存为 60%时既能保证较多查询实现基于内存的查询,又能确保运行内存有足够空间。文中采用数据使用权重内存替换方法,能使多数用户共同感兴趣的热点场景数据以 RDD 形式存于内存,且能保证集群内存不溢出,它们后期的查询任务都能在相对稳定的时间内完成。

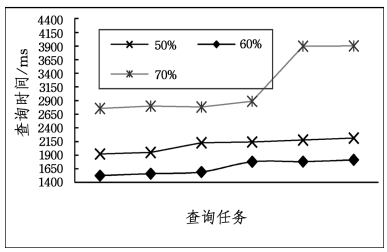

图 4 不同内存查询性能的比较

Fig. 4 Comparison of query performance with different memory

5.4 替换算法对查询性能的影响

为了测试 LRU 与数据使用权重两种内存替换算法对查询性能的影响,实验前先让持久化 RDD 的缓存处于饱和状态,以使后续查询中创建的新 RDD 才有机会替换已经存在于缓存中的旧 RDD。实验时 Spark 集群内存的 60%用作缓存,在两种替换算法下分别进行多次查询,它们的平均查询结果如图 5 所示。很显然,文中提出的数据使用权重内存替换算法对查询性能的提高明显优于 LRU 算法,因为数据使用权重内存替换算法能替换更精细 RDD,被大多数用户认可的热点数据会以 RDD 形式持久化存在于缓存,使得基于 Spark 集群内存查询的比例高于 LRU 算法,所以整体查询性能较优。数据使用权重的算法是利用实际使用的 Partition 个数与该 RDD 的 Partition 总数的比值计算替换权重,计算权重时先要统计 Partition 被使用的次数,再计算它们在 RDD 的 Partition 总数中所占的比例,而 LRU 算法只需统计 RDD 整体的使用次数。很显然,LRU 算法的计算开销比数据使用权重的算法小,但是 LRU 算法是粗粒度替换算法,代表热点数据集的 RDD 对象与非热点 RDD 对象会被等概率替换出缓存,直接导致外存查询比例增加,外存查询的 I/O 开销远远大于数据使用权重的计算开销,因此它的查询性能相对较差。实验结果表明,数据使用权重算法对平均查询性能的影响会随着查询次数的增加而变得更优,但是达到一定极值时将不再随着查询次数的增加而提高。

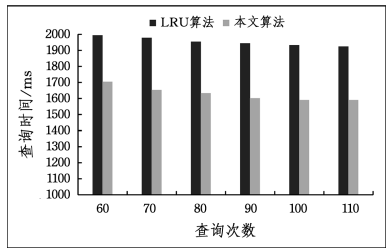

图 5 不同内存替换算法对查询性能的影响

Fig. 5 Effect of different memory replacement algorithms on query performance

结束语 在三维激光扫描技术的推动下,3D 虚拟旅游、数字城市、三维 GIS 等漫游系统越来越多,产生的 3D 点云数据呈爆炸式增长,传统的集中式平台在空间大数据查询方面明显不足,而分布式大数据处理引擎 Spark 却不支持多维空间查询。为此,文中提出基于 Spark 的 3D 点云数据空间索引技术,为它增加了基于 R 树二级空间索引结构以支持多维空间查询。用空间剖分思想解决 R 树兄弟节点空间重叠引起的多路查询性能降低的问题,利用 HDFS 文件的 block 巧妙解决 R 树与磁盘页面定位问题;提出数据使用权重替换算法,该算法能把多数用户感兴趣的热点场景的 3D 点云数据以 RDD 形式持久化于内存中,使 Spark 框架既能实现基于内存空间的查询又能利用空间索引在外存快速查询,大大提高了查询效率;引入 LOD 技术将查询结果按点云数据自带激光反射强度属性值排序,漫游时根据视点远近仅把部分查询数据传输到客户端,以实现细节层次的查询和渲染,解决点云大数据造成的网络阻塞问题;最后,通过实验证明文中提出的方法可行且高效,非常适合海量 3D 点云数据。

"互联网+"旅游是未来发展的新模式,3D 点云数据量也在快速增长,下一步将结合用户对热点场景的兴趣,把热点度作为负载均衡的因素来研究集群机器的负载均衡问题,进一

步提高 Spark 空间的查询能力。LOD 查询中视点由远到近移动时可视范围会缩小，为了减少查询冗余数据，降低网络资源开销，查询区域裁剪也是下一步的研究重点。

参 考 文 献

[1] ELWIRA H, JERZY W, ROBERT S, et al. Color based Algorithm for Automatic Merging of Multiview 3D Point Clouds[J]. ACM Journal on Computing and Cultural Heritage, 2014, 7(3): 1-21.

[2] RUFAEL M, PABLO C. MP3DG-PCC Open Source Software Framework for Implementation and Evaluation of Point Cloud Compression[C]// Proceedings of the 2016 ACM Conference on Multimedia Conference. New York: ACM, 2016: 1222-1226.

[3] ROMULO G, TOMV T, KOSTIS K, et al. Aspatial column store to triangulate the Netherlands on the fly[C]// Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. NewYork: ACM, 2016: 1-4.

[4] ZHANG R, QI J Z, MARTIN S, et al. Towardsa Painless Index for Spatial Objects[J]. ACM Transactions on Database Systems, 2014, 39(3): 1-19.

[5] RIZKI P N M, PARK J S, OH S, et al. STR-Octree Indexing Method for Processing LiDAR Data[C]// Proceedings of IEEE SENSORS. NewYork: IEEE Press, 2015: 1-4.

[6] YANG H, QI W T, GAO X F. Efficient R-Tree Based Indexing Scheme for Server-Centric Cloud Storage System[J]. IEEE Transactions on Knowledge and Data Engineering, 2016, 28(6): 1503-1571.

[7] DENG Y C, JACK C, CHENG P. Automatic Transformation of Different Levels of Detail in 3D GIS City Models in CityGML[J]. International Journal of 3-D Information Modeling, 2015, 4(3): 1-21.

[8] NICHOLAS F P, ANKIT S, PETER S, et al. A Novel Level-of-Detail Technique for Virtual City Environments[C]// Proceedings of the 21st International Conference on Web3D Technology. New York: ACM, 2016: 183-184.

[9] DENG H Y, WU F, ZHAI R J, et al. R-TreeIndex Structure for Multi-Scale Representation of Spatial Data[J]. Chinese Journal of Computers, 2009, 32(1): 177-184. (in Chinese)
邓红艳，武芳，翟仁健，等. 一种用于空间数据多尺度表达的 R 树索引结构[J]. 计算机学报，2009，32(1)：177-184.

[10] SONG B Y, WANG J L, WANG Y, et al. Optimized Storage Strategy Research of HDFS Based on Vandermonde Code [J]. Chinese Journal of Computers, 2015, 38(9): 1825-1837. (in Chinese)
宋宝燕，王俊陆，王妍，等. 基于范德蒙码的 HDFS 优化存储策略研究[J]. 计算机学报，2015，38(9)：1825-1837.

[11] VINITHA R G, NIKHIL T. Indexing HDFS Data in PDW: Splitting the data from the index[C]// Proceedings of the 40th International Conference on Very Large Database. NewYork: PVLDB, 2014: 1520-1528.

[12] JAMES G, SHA N H, LIANGD. Large Scale Distributed Data Science from Scratch using Apache Spark 2.0[C]// Proceedings of the 26th International Conference on World Wide Web Companion. New York: ACM, 2017: 955-957.

[13] RANDALL T W, MICHAEL B P, SARA M A, et al. Spatial indexing and analytics on Hadoop[C]// Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. NewYork: ACM, 2014: 73-82.

[14] ZHONG Y Q, ZHU X M, FANG J Y. Elastic and Effective Spatio-Temporal Query Processing Scheme on Hadoop[C]// Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analyticsfor Big GeospatialData. NewYork: ACM, 2012: 33-42.

[15] AHMED E, MOHAMED F M. A Demonstration of SpatialHadoop: AnEfficientMapReduce Framework for Spatial Data[J]. PVLDB Endowment, 2013, 6(12): 1230-1233.

[16] WAIL Y A, SATTAM A, MICHAEL J C. Largescale Complex Analytics on Semi-structured Datasets using AsterixDB and Spark[J]. Proceedings of the Vidb Endowment, 2016, 9(13): 1585-1588.

[17] GU J P, WU Z B, LI Y L, et al. Parallel Optimization of Pixel Purity Index Algorithm for Hyperspectral Unmixing Based on Spark[C]// Proceedings of Third International Conference on Advanced Cloud and Big Data. New York: IEEE Press, 2015: 159-166.

[18] WANG F Y, WANG X Z, CUI W J, et al. Distributed retrieval for massive remote sensing image metadata on spark[C]// Proceedings of 2016 IEEE International Geoscience and Remote Sensing Symposium. New York: IEEE Press, 2016: 5909-5912.

[19] ZHANG R, QI J Z, MARTIN S, et al. Towards a Painless Index for Spatial Objects[J]. ACM Transactions on Database Systems, 2014, 39(3): 1-41.

[20] PARTH P, DEEPAK G. Perfect Hashing Base R-tree for Multiple Queries[C]// Proceedings of 2014 IEEE International Advance Computing Conference. NewYork: IEEE Press, 2014: 636-640.

[21] BIAN C, YU J, YING C T, et al. Self-Adaptive Strategy for Cache Management in Spark[J]. Acta Electronica Sinica, 2017, 45(2): 278-284. (in Chinese)
卞琛，于炯，英昌甜，等. 并行计算框架 Spark 的自适应缓存管理策略[J]. 电子学报，2017，45(2)：278-284.

[22] SHI J W, QIU Y J, UMAR F M, et al. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics [C]// Proceedings of the 41st International Conference on Very Large Database. New York: PVLDB, 2015: 2110-2121.