In Memory Processing of Massive Point Clouds for Multi-core Systems

Kostis Kyzirakos Foteini Alvanaki Martin Kersten

Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
{first name}.{last name}@cwi.nl

ABSTRACT

LIDAR is a popular remote sensing method used to examine the surface of the Earth. LIDAR instruments use light in the form of a pulsed laser to measure ranges (variable distances) and generate vast amounts of precise three dimensional point data describing the shape of the Earth. Processing large collections of point cloud data and combining them with auxiliary GIS data remain an open research problem.

Past research in the area of geographic information systems focused on handling large collections of complex geometric objects stored on disk and most algorithms have been designed and studied in a single-thread setting even though multi-core systems are well established. In this paper, we describe parallel alternatives of known algorithms for evaluating spatial selections over point clouds and spatial joins between point clouds and rectangle collections.

1. INTRODUCTION

Light Detection and Ranging (LIDAR) data provide a wealth of information for various application domains like urban planning, smart cities and natural resource management. The production of large collections of point cloud data has increased over the past years due to its easy collection via airborne laser scanning. Airborne laser scanning is a remote sensing technology that enables users to collect rapidly large amounts of point data at global scale. Many datasets with national coverage have been released during the last few years as open data. For example, the AHN dataset [1] is the Dutch elevation map that is freely distributed as open data. The fist version of AHN contained one point per 16-25m^2, while the second version contained 6-10 points/m^2 resulting in approximately 640 billion points for the whole country. The dataset is distributed as a collection of 60,000 files encoded according to the Laser (LAS) file format [2].

File based solutions were initially developed for point clouds and recently multiple DBMS were extended with support for LIDAR data. PostgreSQL and Oracle physically reorganize point cloud data into blocks that contain multiple points following a condensed representation. Afterwards, indices are built using the spatial extent of each block. In this paper, we follow a different direction and opt for a purely columnar storage scheme like the one used by MonetDB. In addition, our algorithms are inspired by the processing model followed by MonetDB.

In this paper, we focus on implementation details of known algorithms for spatial selections and joins for exploiting the characteristics of modern hardware. Given the recent increase of main memory capacities, we focus on main memory databases. In these systems the bottleneck is no longer the I/O bandwidth but the memory bandwidth. Column store databases use memory bandwidth more efficiently by maximizing the amount of useful data that is transferred. Compression is used to limit further the utilization of memory bandwidth increasing at the same time the data elements that fit in cache. Although compression requires extra CPU time, there are many compression schemes that allow some operations to be performed directly on compressed data.

Indices have been heavily used in databases to minimize the amount of data that needs to be accessed. The shift from disk-based to memory-based databases lead to the need of adapting the index structures and the algorithms used to access them. Traditionally, indices were focusing on minimizing I/O but used in a main memory setting caused a lot of random memory accesses compared to e.g. simple scanning approaches. Indices are still used to limit the mount of accessed data, however nowadays spatial locality of the data accesses plays an important role.

Multi-core systems pose one more challenge to index designers, that of efficiently utilizing the available parallel resources. With increasing number of threads the main memory bandwidth problem becomes more prominent. Having multiple threads processing independent chunks of the index may result in memory bandwidth contention.

The rest of the paper is organized as follows: In Section 2, we discuss related work and in Section 3, we present how to index a point cloud using a data-driven data structure and known algorithms for evaluating spatial selections and spatial joins using a grid index. In Section 4, we present how we implemented the aforementioned algorithms and in Section 5, we evaluate the algorithms experimentally in two server-class machines. In Section 6, we conclude the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DaMoN'16, June 27,2016, San Francisco, USA

© 2016 ACM. ISBN 978-1-4503-3638-3/15/06...$15.00

DOI: http://dx.doi.org/10.1145/2933349.2933356

2. RELATED WORK

Traditional GIS indexing structures are divided in two broad categories: space driven indices and data driven indices. Space driven indices partition the space into rectan-

gular cells regardless of the spatial distribution of the geometric objects. Afterwards, geometric objects are assigned to one or more overlapping cells. On the contrary, data-driven indices partition the geometric objects by taking into account their spatial distribution in the enclosing space.

The *fixed grid* index [17] is a space driven index that tessellates the space into equi-area rectangular cells. Each cell is assigned a disk page and each geometry is assigned to all overlapping cells. The *grid file* [15] is an improvement of the fixed grid where data are not assumed to follow a uniform distribution. When the points assigned to a cell do not fit in a single disk page, the cell is split into two cells and the points are reassigned to the new cells accordingly. This leads to the creation of a non-regular grid that adapts to the spatial distribution of the indexed points. A well-known grid-based spatial index is the quadtree [14]. A quadtree is created by diving the space in four cells. In each level, each cell is further subdivided to four cells resulting in a tree where each parent has exactly four children nodes. A different approach is followed by data structures based on space-filling curves. A *space-filling curve* [13] is a function that defines a total order on the cells of a regular grid. As a result, cells that are close in space are probably close in the total order. Space filling curves is a method of dimension reduction, allowing a DBMS to index multi dimensional data using a B+tree index.

On the other hand, data driven data structures partition the geometric objects by taking into account their spatial distribution. The most notable index in this category is the R-tree [11]. An R-tree groups together neighbouring objects. A rectangle that encloses all geometries in a group is used for representing the geometries in the higher level. One main advantage of a data-driven data structure like the R-tree is that complex geometric objects (e.g., lines, polygons) appear only once in the index. On the contrary, when a space-driven structure is used, the object identifier may be inserted multiple times in several leaves. As a result when indexing objects that overlap multiple cells, the duplication rate increases significantly. However, this drawback does not apply when indexing point geometries since a point will overlap exactly one cell.

LAStools [3] is a file based solution for handling point cloud data distributed as LAS files. LAStools provide two closed source tools for indexing LAS files: lassort and lasindex. The former tool sorts the points of a LAS file using a space filling curve while lasindex creates a square quadtree for a sorted LAS file. PostGIS [9] is an extension of PostgreSQL for handling geospatial information. PostGIS provides an implementation of an R-tree but its performance was not satisfactory when indexing large point clouds. As a result, PostgreSQL was recently extended with a new module for handling point cloud data [8]. Points are organized in patches that consist of 5,000 points and an R-tree index is created over the rectangles that enclose all points of each patch. Oracle Spatial and Graph was recently extended with support for point cloud data [7]. Oracle follows a similar approach with PostgreSQL and groups points in patches of 5,000 points. Each patch is compressed using various compression methods and a Hilbert R-Tree is constructed over the rectangles that enclose all points of each patch. MonetDB [4] was recently extended with a data vault for LAS files. The Data Vaults [12] is a mechanism that provides a true symbiosis between a DBMS and existing file-based repositories. By utilizing the LIDAR data vault and the recent extensions of the geospatial module of MonetDB, the user can efficiently manage and process vast collections of point cloud data. MonetDB stores the point cloud data using a flat table approach. A lightweight and cache conscious secondary index called Imprints is used for executing a coarse filtering step, followed by the construction of a fixed grid for the refinement step [10]. An implementation of a secondary index based on a fixed grid is ongoing work. Recently, a functional and performance benchmark was developed for comparing all aforementioned systems [18]. In [18], the authors propose a benchmark based on the Dutch elevation map that consists of 640 billion points.

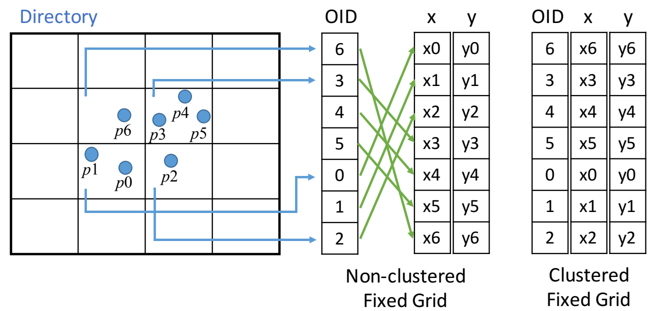


Figure 1: Fixed Grid Index

3. PROCESSING POINT CLOUD DATA

In this section, we describe the data structures and algorithms that we will examine in this paper for processing point clouds. Point cloud data are represented by a set of coordinates (x,y,z) and a set of accompanying attributes. We store the point cloud data following a fully decomposed model i.e., we create a relation `PC` having one column for each coordinate and dataset attribute. Given that the density of point cloud data is fairly uniform, we experimented with space-driven data structures for point cloud indexing. For storing a collection of rectangles we create a relation `RECT` with a single attribute `r` that stores rectangular geometric objects of the form $(x_{min}\ y_{min}, x_{max}\ y_{max})$. In the rest of this section, we describe the fixed grid index that we implemented and we describe the two types of queries that we examine in this paper, namely spatial selections and spatial joins.

3.1 Indexing

We choose to index point cloud data using a fixed grid. For creating the fixed grid, we decompose the search space into rectangular cells. The resulting grid is a regular grid that consists of $n_x \times n_y$ equi-area cells. Each cell is associated with a position of an array that stores the object identifiers (OIDs) of each point. A point p is assigned to cell c if c contains the map geometry of p. For indexing n points using a $n_x \times n_y$ regular grid, we need an array $DIR[n_x, n_y]$ as a directory and an array $OID[n]$ for storing the OIDs of all points. Each element $DIR[i, j]$ of the directory contains a pointer to the first element of the array OID that stores the objects assigned to cell $c_{i,j}$.

Figure 1 depicts a fixed grid indexing a collection of seven points. The grid directory is a 2D array that contains the position of the array OID storing the first point assigned to each cell. The array OID contains the oids of all point

Algorithm 1 Scan

```
Input: X                              ▷ Array with x coordinates
       Y                              ▷ Array with y coordinates
       r                              ▷ Query rectangle
Output: OID    ▷ Bit vector indicating the points that are inside r
Set OID ← 0
for x_i ∈ X do
    if r.x_min ≤ x_i ≤ r.x_max then
        OID[i] = 1
    end if
end for
for y_i ∈ Y do
    if r.y_min ≤ y_i ≤ r.y_max then
        OID[i] &=1
    else
        OID[i] = 0
    end if
end for
```

ordered according to the cell that each point is assigned into. Each oid indicates a position on the arrays x and y that store the coordinates of the points.

3.2 Spatial Selections

The term spatial selection is used to describe a selection based on a spatial predicate. A spatial predicate can be a combination of topological, metric, or directional relations between two spatial objects. In this paper we focus on *window queries* that ask for spatial features of a dataset that are enclosed by a user-specified area.

EXAMPLE 3.1. *The following SQL query is a spatial selection:*

```
SELECT *
FROM PC
WHERE ST_Contains(
        ST_Envelope(ST_GeomFromText("POLYGON((
           xmin ymin, xmax ymin, xmax ymax,
           xmin ymax, xmin ymin))"),
        ST_Point(PC.x, PC.y));
```

The above query selects all points from the point cloud relation `PC` that are within the user defined rectangle `(xmin ymin, xmax ymax)`. The functions prefixed with `ST_` are functions defined by the Simple Features Access standard of the Open Geospatial Consortium [5].

Let us now discuss how we can evaluate a spatial selection that takes as input a collection of points and returns as a result a list of qualifying OIDs. We distinguish between the cases of querying an unindexed and indexed point cloud.

Unindexed Point Cloud.

When the point cloud is not indexed, a full scan is required for evaluating a spatial selection. The scan operator starts by reading sequentially values from the x and y columns and exhaustively evaluating the spatial predicate. Since data are read sequentially, the scan operator exhibits good spatial locality and allows efficient prefetching. We follow a fully decomposed model where point coordinates are stored in separate columns and only one column is accessed at any time. We start by creating a bit vector that will be used for storing the result-set. Afterwards, we examine an x coordinate and if it satisfies the spatial predicate, the corresponding bit is set. We repeat this process for the y coordinate. Algorithm 1 demonstrates this idea.

Algorithm 2 Grid Index Lookup

```
Input: Index                  ▷ Grid index for point cloud data
       X                              ▷ Array with x coordinates
       Y                              ▷ Array with y coordinates
       r                              ▷ Query rectangle
Output: OID    ▷ Bit vector indicating the points that are inside r
Set OID ← 0
Compute Cells ← compute Index cells that overlap r
for cell_j ∈ Cells do
    for x_i ∈ cell_j do
        if r.x_min ≤ x_i ≤ r.x_max then
            OID[i] = 1
        end if
    end for
    for y_i ∈ cell_j do
        if r.y_min ≤ y_i ≤ r.y_max then
            OID[i] &=1
        else
            OID[i] = 0
        end if
    end for
end for
```

Indexed Point Cloud.

When an index over the point cloud dataset exists, it is used for limiting the data that needs to be accessed for evaluating the spatial selection. Given a rectangle $r = (x_{min}, y_{min}, x_{max}, y_{max})$, we compute the grid cells that intersect r. For each cell, we access the index to find all points that are located inside the cell. The coordinates of each point are retrieved and used to determine whether a point is inside r. Algorithm 2 presents how the grid index is used for evaluating a spatial selection.

3.3 Spatial Joins

A spatial join is a theta join between collections of spatial objects where theta is a combination of topological or directional predicates. We focus on spatial joins between point clouds and collections of rectangles using the *contains* spatial predicate.

EXAMPLE 3.2. *The following SQL query is a spatial join:*

```
SELECT *
FROM PC, RECT
WHERE ST_Contains(RECT.r, ST_Point(PC.x, PC.y));
```

The above query selects all pairs of points and rectangles which satisfy the spatial predicate `ST_Contains`.

Index on Point Cloud.

If there is an index on the point cloud but the rectangle collection is not indexed, we perform the join using the *Indexed Nested Loops* algorithm. According to the algorithm, we iterate the rectangle collection and for each rectangle, we perform a grid index lookup. In contrast to Algorithm 2, we do not use a bit vector for computing the results. First, the x coordinate of the points is examined and a list of qualifying OIDs is created. This set of OIDs is called set of *candidates*. In a subsequent step, the y coordinates are examined. The candidates are used to access only the y coordinates of the

Algorithm 3 Indexed Nested Loops

```
Input: Index   ▷ An array with the sets of idxs of the points in each cell
       X                          ▷ Array with x coordinates
       Y                          ▷ Array with y coordinates
       R                          ▷ Array with rectangles
Output: OIDs_r                    ▷ List of OIDs of rectangles
        OIDs_p                    ▷ List of OIDs of points
OIDs_r ← EmptyList
OIDs_p ← EmptyList
for r_j ∈ R do
    candidates ← EmptyList
    Compute Cells ← the cells of the Index that overlap r
    for cell ∈ Cells do
        for x_i ∈ cell do
            if r_j.x_min ≤ x_i ≤ r_j.x_max then
                candidates.add(i)
            end if
        end for
    end for
    for i ∈ candidates do
        if r_j.y_min ≤ y_i ≤ r_j.y_max then
            OIDs_r.add(j)
            OIDs_p.add(i)
        end if
    end for
end for
```

points with qualifying x coordinates. Candidates with qualifying y coordinates belong to the result-set. Algorithm 3 presents the idea of the Indexed Nested Loops.

Index on Both Relations.

When both the point cloud and the rectangle collections are indexed, we evaluate the join using the Partition Based Spatial-Merge Join (PBSM) [16] algorithm. The PBSM algorithm assumes that both relations are indexed over the same grid. The algorithm performs the join by comparing all elements of each cell of one relation with all elements of the same cell of the other relation. The idea of PBSM is sketched in Algorithm 4.

Algorithm 4 PBSM

```
Input: Index                      ▷ Grid Index
       X                          ▷ Array with x coordinates
       Y                          ▷ Array with y coordinates
       R                          ▷ Array with rectangles
Output: OIDs_r                    ▷ List of OIDs of rectangles
        OIDs_p                    ▷ List of OIDs of points
OIDs_r ← EmptyList
OIDs_p ← EmptyList
for cell ∈ Index do
    candidates_r ← EmptyList
    candidates_p ← EmptyList
    for r_j ∈ cell do
        for x_i ∈ cell do
            if r_j.x_min ≤ x_i ≤ r_j.x_max then
                candidates_r.add(j)
                candidates_p.add(i)
            end if
        end for
    end for
    for k ∈ sizeof(candidates_p) do
        j ← candidates_r[k]
        i ← candidates_p[k]
        if r_j.y_min ≤ y_i ≤ r_j.y_max then
            OIDs_r.add(j)
            OIDs_p.add(i)
        end if
    end for
end for
```

4. IMPLEMENTATION

Let us now discuss how we implemented the algorithms presented in Section 3. We discuss how we parallelised the aforementioned algorithms, how we placed data in a NUMA local manner and how we modified the grid index to reduce the number of random memory accesses.

4.1 Parallelisation

We parallelised all algorithms using OpenMP [6]; an interface that allows multi-platform shared-memory parallel programming. Below, we describe the methodology we employed for parallelising each one of the four algorithms.

Scan: The parallelisation of the *scan* operator is performed by splitting the point cloud data in pieces. The arrays containing the x and y coordinates are logically split in chunks of the same size and a separate thread is assigned to each chunk. Each thread processes an independent part of the dataset and writes the results at its own local structures eliminating the need for synchronisation.

Grid Index lookup: We split the grid index lookup algorithm in two phases. During the first phase, a single thread tessellates the query rectangle and computes a discrete representation of it. The tessellation allows us to approximate the query rectangle by a finite number of cells. In the second phase, the intersecting cells are logically divided among all threads. Each thread processes the points that are inside the cells that have been assigned to it.

Indexed Nested Loops: We logically divide the set of rectangles evenly among all threads. Each thread performs a grid index lookup for each rectangle that has been assigned to it. The index is shared among all threads and multiple threads might access the same cells concurrently. Since only read accesses are performed on them there is no need for synchronisation.

Given that point cloud data are usually joined with collections of polygons that are roughly of similar area (e.g., cadastral property boundaries), this approach distributes the load evenly among all threads. For other cases, one should add a pre-processing step where rectangles are split in chunks according to their area.

PBSM: We logically divide the grid cells evenly among all threads. Each thread processes sequentially all assigned cells and for each cell it performs a nested loops joins between the indexed points and rectangles.

4.2 Optimisations

We applied various well-known optimisation techniques for making all algorithms more efficient. We employed techniques like *predication*, *loop unrolling* and *blocking* wherever applicable. For vectorisation, we rely on the capabilities of the compiler.

Furthermore, we made additional optimisations that are specific to the algorithms that we examine. In the case of the Grid Index lookup and Indexed Nested Loops, we can safely omit the evaluation of the spatial predicate over points that are indexed at cells that are completely within the query rectangle. Such points are immediately copied to the result set. Only the cells that intersect the border of a query rectangle need to be examined in detail since they might contain

points that are located inside or outside the query rectangle.

Clustered Grid Index.

The grid index is used for reducing the number of points that are examined when processing a spatial predicate. As a result, only data that are likely to be an answer to a query are transferred to the CPU. However, there is no guarantee that points with coordinates stored in neighboring positions in the x and y arrays will also be close in the plane. As a result, the structure OID, that organises point OIDs according to the cell that they belong into, contains sets of non consecutive OIDs. Thus, for examining a spatial predicate over a grid cell, we need to perform many random accesses to the x and y arrays. In specific, each access to the x and y arrays requires dereferencing a pointer resulting to a pointer chasing situation. This makes prefetching impossible since the hardware has no means of knowing where the data point is placed, and increases cache and DTLB misses. In a multi-threaded setting this problem is magnified and quickly the memory bandwidth is saturated since each thread performs random memory accesses and utilizes only a small portion of the transferred data.

To overcome this problem, we also experiment with a *Clustered Grid Index*. When creating a clustered grid index, the contents of the arrays x and y are physically reorganized so that all points that are close in the plane are also physically stored in neighbouring positions. As a result, the grid directory contains the position of the aligned arrays OID, x and y storing the first point assigned to each cell. A spatial predicate can now be evaluated by accessing directly sequential parts of the x and y arrays while the OID array is accessed only for writing the qualifying OIDs to the result set. Figure 1 depicts the organisation of a non-clustered and clustered grid index.

NUMA local data placement.

High-end servers are equipped with interconnected sockets of multi-core processors, each of which has its memory controller and memory. Sockets are interconnected through a Quick Path Interconnect (QPI) allowing them to access remote memory on other sockets. This decentralised architecture is called non-uniform memory access (NUMA). The bandwidth of the QPI is significantly lower than the bandwidth of the memory bus that connects each socket to its local DRAM. Thus, the bandwidth of the QPI can be separately saturated affecting the scale capabilities of the algorithms that we examine in this paper.

For investigating the effect of the limited QPI bandwidth we implemented a NUMA local version of all algorithms where data are replicated in all NUMA nodes and each thread processes a subset of the data stored in its local DRAM. We opted for NUMA local data placement instead of NUMA aware data sharding because a NUMA aware implementation of grid based algorithms would require changing the parallelisation methodology employed in each algorithm, adding an extra variable to the comparison. A straightforward NUMA aware implementation would require distributing the input x and y columns evenly on each NUMA node and creating locally a separate grid index. The grid index for the rectangle collection is in principle much smaller compared to the grid index for the point cloud, thus it can be replicated to all NUMA nodes.

5. EXPERIMENTAL EVALUATION

In this section we study the speedup of the algorithms presented in Sections 3 and 4 as we increase the number of worker threads.

5.1 Experimental platforms

We tested all algorithms on two hardware platforms with different configurations, namely a server class machine and a high-end server. The server is equipped with two Intel Xeon E5-2650 at 2-2.8 GHz, each of which has 8 cores and 16 hardware threads. Each core has 32 KB L1I and L1D cache, and 256 KB L2 cache. Each CPU has 20 MB of shared L3 cache. The server has 256 GB RAM in total. The high-end server is equipped with four Intel Xeon E5-4657L at 2.4-2.9 GHz, each of which has 12 cores and 24 hardware threads. Each core has separate 32 KB L1D and L1I cache, and 256 KB L2 cache. Each CPU has 30 MB of shared L3 cache. The high-end server has 1 TB RAM in total.

5.2 Dataset

In practice, LIDAR data cover large convex quadrilateral surfaces. The points cloud datasets are distributed in the form of multiple files, each covering an axis aligned rectangular area. For example, the Dutch elevation map (AHN) distributes point cloud data for the city of Amsterdam in four files, each of which contains on average 500 million points. Following this practice, we generated a point cloud dataset that consists of 500 million points covering an axis aligned rectangular area. The points are uniformly distributed in the plane and each point is assigned an 8-byte OID and a pair of 8-byte integer coordinates. The resulting point cloud dataset occupies 7.6 GB of storage.

For generating rectangles, we utilized the Dutch cadastral property boundaries collection (Digitale kadastrale kaart). This collections contains approximately 40,000 property boundaries in the area covered by each AHN file for the city of Amsterdam. We measured the number of AHN points contained by each polygon and on average, the selectivity factor was 0.011%. For this reason, we generated 40,000 rectangles, each of which is assigned a 4-byte OID and a minimum bounding box that is represented by four 8-byte integer coordinates.

5.3 Varying Parameters

For spatial selections, we measured the time required by each algorithm to compute the OIDs that satisfy the spatial predicate. We experimented with spatial predicates of various selectivity factors, namely 1%, 10%, 25%, 50%, 75% and 100%. For spatial joins, we generated three collections of rectangles. On average, the selectivity factor for each polygon is the measured 0.011%, its half 0.0055% and its double 0.022%.

Additionally, we vary the number of worker threads used. We start by running each algorithm using a single thread, then we assign one, two, half the available cores, all available cores and all available threads per socket. This means that for the server we use 1,2,4,8,16 and 32 threads and for the high-end server we use 1,4,8,24,48 and 96 threads.

5.4 Grid creation

Indexing 500 million (resp. 1 billion, 2 billion) points required 18.12 (resp. 36.53, 79.11) seconds. Indexing 40,000 rectangles required about 5 msec. The grid index occupies

3.8 GB of space for storing the OID array and 129 KB for storing the grid directory.

5.5 Spatial Joins

We run each configuration on each machine five times and report the average response time. Both indexed nested loops and PBSM were tested using every possible combination of NUMA agnostic vs NUMA local data placement and random vs clustered grid index. We start the discussion for the case where the average selectivity factor of each rectangle is 0.011%.

In Figures 2a and 2b, we observe that the response time, for the Indexed Nested Loops algorithm, is affected by the data placement policy. Since the algorithm is memory bound, the benefit comes from replacing remote memory accesses with local memory accesses. The utilized memory bandwidth increases since requests are directed to all available NUMA memory nodes. This observation becomes more prominent when increasing the number of worker threads since the QPI is quickly saturated when a large number of threads perform remote memory requests. The measured CPI when 8 threads are used dropped from 16.7 to 13. For the same reason the speed-up of the algorithm at the server is increased by 30% when the default grid is used and by 20% when the clustered grid is used (Tables 1-4). Accessing remote memory at the high-end server involves up to two QPI links so the measured speed-up increases by 70% and 50% when using the default or the clustered index respectively (Tables 5-8).

In Figures 2a and 2b, we also observe that the response time is heavily affected by the data access pattern imposed by using the default or the clustered index. When using 8 threads at the server using NUMA local data placement we measured that 28% of CPU stalls is attributed to DTLB misses when the default grid index is used, while this percentage drops to 1.9% when the clustered index is used. As a result, the percentage of retired micro-operations increases from 1.6% to 30% and CPI drops from 16.70 to 0.88. Similar results were observed in all other cases. PBSM exhibits that the same behaviour but at a lesser extent as we observe in Figure 3 and Tables 9-16. We observed the same behaviour with all selectivity factors tested. For brevity, we omit the respective plots.

5.6 Spatial Selections

We run each configuration on each machine seven times and report the average response time. We tested all possible combinations of NUMA agnostic vs NUMA local data placement and random vs clustered grid index. We start the discussion for scan operator.

The change of the data layout from NUMA agnostic to NUMA local improves the response time of the scan operator in both the server and the high-end server (cf. Figure 4). The observed behaviour matches that observed for the indexed nested loops and the PBSM algorithms. We observe that the algorithm scales-up almost linearly when up to 8 threads are used and degrades slightly when hyper-threads are utilized. We observe that the scale-up between the NUMA aware and NUMA local differ significantly at the high end server. When using 48 threads, the scale-up of the NUMA agnostic approach is only 6x while the NUMA local approach has a scale-up factor 27x.

The change of the data layout from NUMA agnostic to NUMA local does not improve significantly the response time of the grid index (cf. Figure 5). The grid index lookup examines exhaustively the points that are located close to the border of the query rectangle while the rest points are used to set the appropriate bits of the bit vector used for storing the result set without examining their values. Thus, using the clustered grid index is beneficial only when processing points close to the border of the query rectangle where the x and y values are retrieved. For this reason, we observe at Tables 21-28 that the speed-up of the algorithm is slightly affected by the usage of the clustered index.

6. CONCLUSIONS

In this paper, we studied how to parallelise grid based algorithms for evaluating spatial selections and spatial joins over massive point clouds in the context of a main-memory columnar DBMS. We investigated the effect of parallelisation when point cloud data are stored according to a fully decomposed model. We also investigated how to use a regular grid for quickly indexing point clouds and how it can be used in a multi-core setting. We observed good scale-up for a small number of threads, but the random accesses that incur when using many threads quickly saturate the memory bandwidth.

Using a clustered grid index increases the spatial locality of the index accesses, leading to much faster solutions. Additionally, with the NUMA architectures it is important to partition the data in a NUMA local configuration and modify the algorithms to minimise the accesses to remote memory.

7. REFERENCES

[1] Actueel Hoogtebestand Nederland. http://www.ahn.nl/.

[2] American Society for Photogrammetry and Remote Sensing LASer File Format Exchange. http://www.asprs.org/committee-general/laser-las-file-format-exchange-activities.html.

[3] Lastools software suite. https://rapidlasso.com/lastools/.

[4] MonetDB. https://www.monetdb.org/.

[5] OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option. http://portal.opengeospatial.org/files/?artifact_id=25354.

[6] OpenMP. www.openmp.org/.

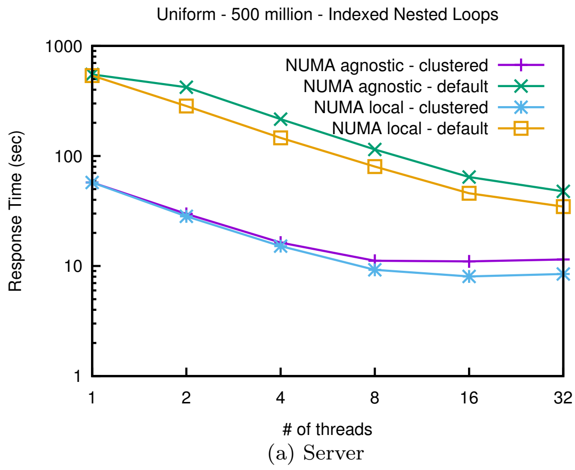
[7] Point Clouds package of Oracle 12c. https://docs.oracle.com/cd/B28359_01/appdev.111/b28400/sdo_pc_pkg_ref.htm.

[8] Pointcloud extension for PostgreSQL. https://github.com/pgpointcloud/pointcloud/.

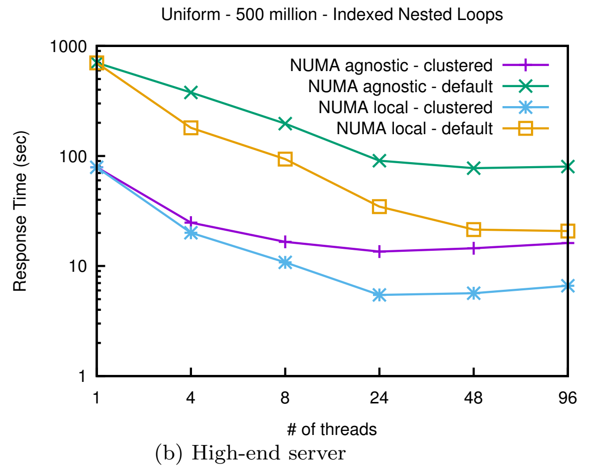
[9] PostGIS extension for PostgreSQL. http://postgis.net/.

[10] F. Alvanaki, R. Goncalves, M. Ivanova, M. L. Kersten, and K. Kyzirakos. GIS navigation boosted by column stores. *PVLDB*, 8(12):1956–1967, 2015.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

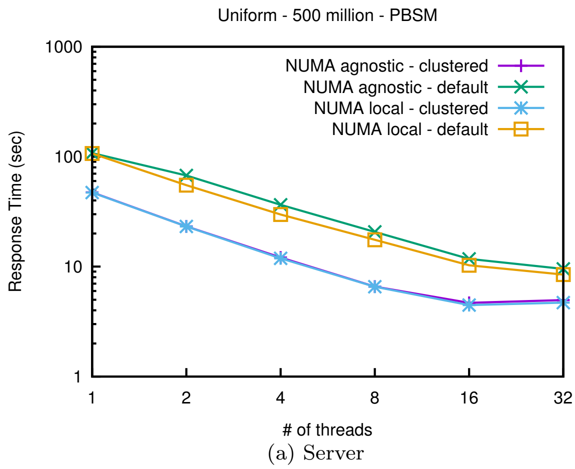


(a) Server

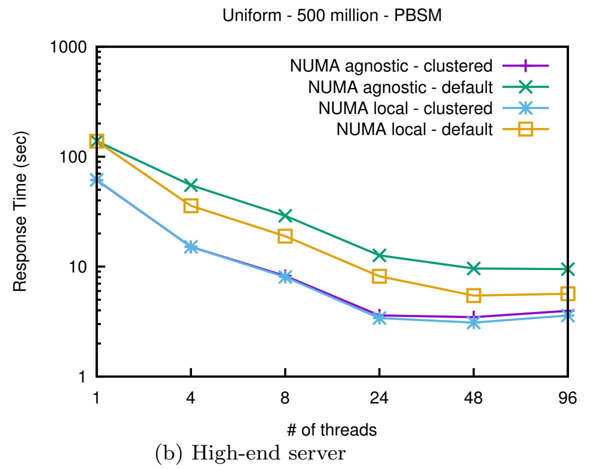


(b) High-end server

Figure 2: Indexed Nested Loops

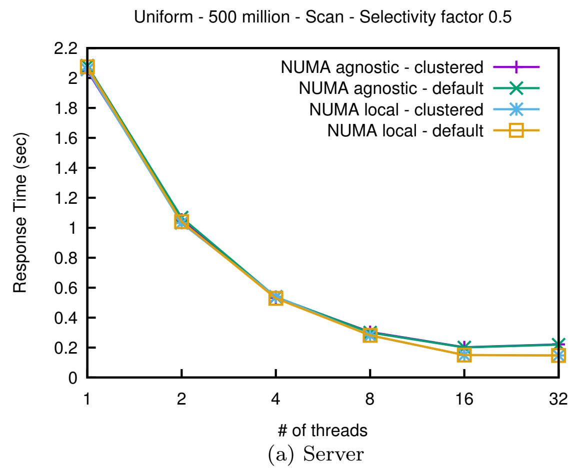


(a) Server

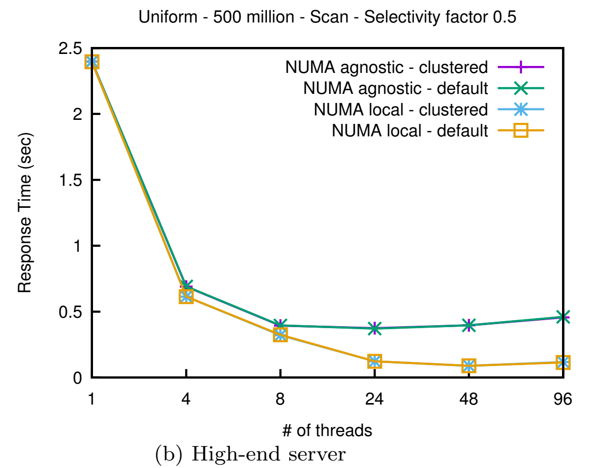


(b) High-end server

Figure 3: PBSM

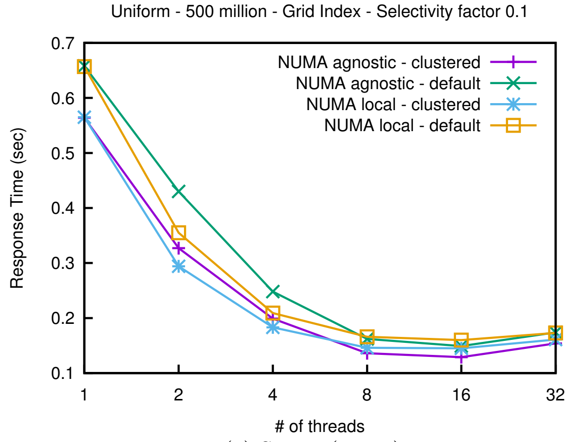


(a) Server

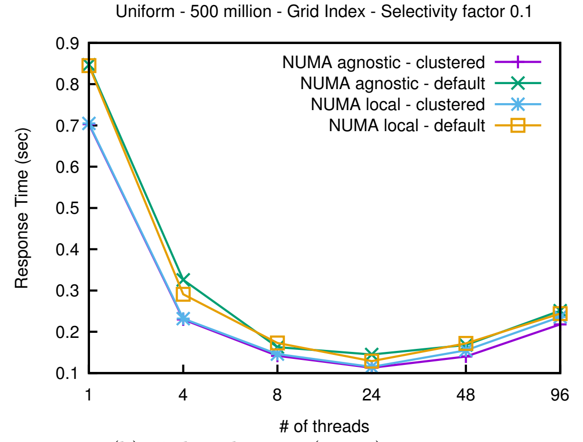


(b) High-end server

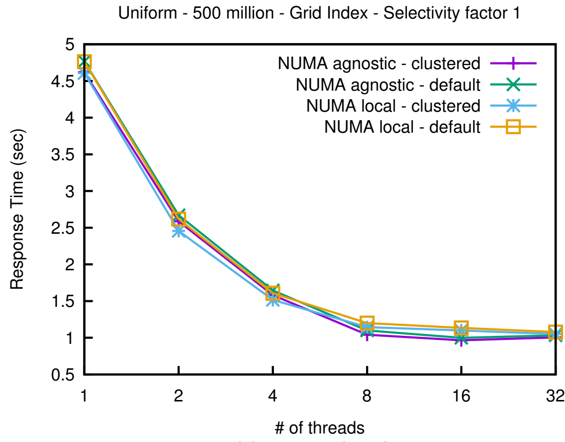
Figure 4: Scan



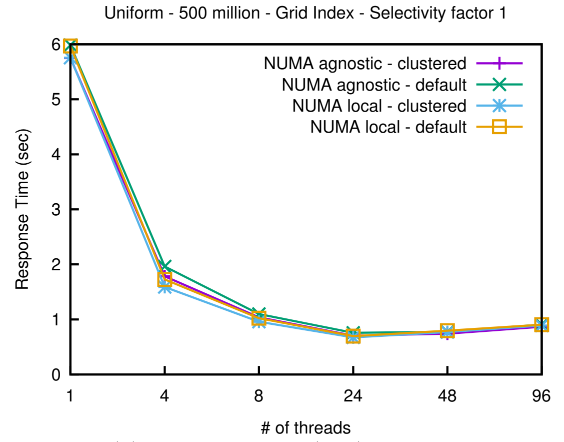
(a) Server (s=0.1)



(b) High-end server (s=0.1)



(c) Server (s=1)



(d) High-end server (s=1)

Figure 5: Grid index lookup

-	1	2	4	8	16	32
0.000055	1.00	1.27	2.53	4.80	8.46	11.62
0.00011	1.00	1.31	2.55	4.81	8.56	11.54
0.00022	1.00	1.32	2.61	4.90	8.73	11.64

Table 1: Indexed Nested Loops - NUMA Agnostic Random Accesses - Server

-	1	2	4	8	16	32
0.000055	1.00	1.86	3.38	4.86	4.93	4.75
0.00011	1.00	1.93	3.52	5.14	5.21	5.00
0.00022	1.00	2.28	4.21	6.23	6.28	5.98

Table 2: Indexed Nested Loops - NUMA Agnostic Clustered Accesses - Server

-	1	2	4	8	16	32
0.000055	1.00	1.89	3.68	6.72	11.92	15.74
0.00011	1.00	1.89	3.68	6.70	11.71	15.49
0.00022	1.00	1.93	3.76	6.84	11.85	15.46

Table 3: Indexed Nested Loops - NUMA Aware Random Accesses - Server

-	1	2	4	8	16	32
0.000055	1.00	1.99	3.68	6.09	7.06	6.81
0.00011	1.00	2.03	3.79	6.23	7.14	6.80
0.00022	1.00	2.38	4.46	7.37	8.33	7.90

Table 4: Indexed Nested Loops - NUMA Aware Clustered Accesses - Server

-	1	4	8	24	48	96
0.000055	1.00	1.83	3.53	7.71	8.81	8.63
0.00011	1.00	1.86	3.57	7.75	9.06	8.76
0.00022	1.00	1.89	3.66	7.85	8.95	8.76

Table 5: Indexed Nested Loops - NUMA Agnostic Random Accesses - High-end Server

-	1	4	8	24	48	96
0.000055	1.00	2.99	4.41	5.20	4.88	4.33
0.00011	1.00	3.19	4.76	5.85	5.45	4.88
0.00022	1.00	3.31	5.11	6.45	6.03	5.37

Table 6: Indexed Nested Loops - NUMA Agnostic Clustered Accesses - High-end Server

-	1	4	8	24	48	96
0.000055	1.00	3.87	7.45	20.26	32.55	34.41
0.00011	1.00	3.89	7.49	20.27	32.75	33.77
0.00022	1.00	3.90	7.49	20.24	32.36	33.32

Table 7: Indexed Nested Loops - NUMA Aware Random Accesses - High-end Server

-	1	4	8	24	48	96
0.000055	1.00	3.81	7.11	13.86	13.50	11.40
0.00011	1.00	3.93	7.31	14.45	13.92	11.91
0.00022	1.00	4.02	7.49	14.98	14.16	12.04

Table 8: Indexed Nested Loops - NUMA Aware Clustered Accesses - High-end Server

-	1	2	4	8	16	32
0.000055	1.00	1.47	2.79	4.99	8.91	11.48
0.00011	1.00	1.60	2.95	5.21	9.16	11.30
0.00022	1.00	1.79	3.30	5.76	10.01	11.75

Table 9: PBSM - NUMA Agnostic
Random Accesses - Server

-	1	2	4	8	16	32
0.000055	1.00	2.03	3.98	7.05	9.99	9.58
0.00011	1.00	2.04	3.91	7.20	10.11	9.54
0.00022	1.00	2.17	4.28	7.75	10.69	10.19

Table 10: PBSM - NUMA Agnostic
Clustered Accesses - Server

-	1	2	4	8	16	32
0.000055	1.00	1.91	3.55	6.14	10.58	13.44
0.00011	1.00	1.94	3.59	6.09	10.38	12.61
0.00022	1.00	2.01	3.80	6.45	10.64	12.35

Table 11: PBSM - NUMA Aware
Random Accesses - Server

-	1	2	4	8	16	32
0.000055	1.00	2.05	3.95	7.26	10.82	10.40
0.00011	1.00	2.03	3.97	7.19	10.54	10.02
0.00022	1.00	2.18	4.27	7.74	10.98	10.39

Table 12: PBSM - NUMA Aware
Clustered Accesses - Server

-	1	4	8	24	48	96
0.000055	1.00	2.25	4.31	9.80	12.71	13.33
0.00011	1.00	2.52	4.78	10.94	14.41	14.59
0.00022	1.00	2.81	5.31	12.53	16.03	15.90

Table 13: PBSM - NUMA Agnostic
Random Accesses - High-end Server

-	1	4	8	24	48	96
0.000055	1.00	4.00	7.45	16.38	16.43	14.01
0.00011	1.00	4.05	7.46	17.06	17.67	15.48
0.00022	1.00	4.03	7.53	17.97	18.64	16.26

Table 14: PBSM - NUMA Agnostic
Clustered Accesses - High-end Server

-	1	4	8	24	48	96
0.000055	1.00	3.82	7.18	16.93	26.43	26.01
0.00011	1.00	3.86	7.31	16.94	25.38	24.43
0.00022	1.00	3.93	7.47	17.63	24.27	22.74

Table 15: PBSM - NUMA Aware
Random Accesses - High-end Server

-	1	4	8	24	48	96
0.000055	1.00	4.05	7.70	18.23	20.39	17.24
0.00011	1.00	4.07	7.65	18.08	19.86	17.15
0.00022	1.00	4.05	7.61	18.56	19.60	16.92

Table 16: PBSM - NUMA Aware
Clustered Accesses - High-end Server

-	1	2	4	8	16	32
0.01	1.00	1.93	3.82	6.92	10.21	9.42
0.1	1.00	2.02	3.74	7.17	10.24	9.45
0.25	1.00	1.97	3.62	6.94	10.08	9.35
0.5	1.00	1.95	3.88	6.94	10.36	9.51
0.75	1.00	1.95	3.82	7.01	10.17	9.34
1	1.00	1.95	3.70	6.91	10.16	9.33

Table 17: Scan - NUMA Agnostic
Server

-	1	2	4	8	16	32
0.01	1.00	2.00	3.84	7.14	13.57	14.04
0.1	1.00	2.03	4.00	7.34	13.89	14.18
0.25	1.00	1.99	3.81	7.35	13.77	13.96
0.5	1.00	1.99	3.92	7.36	13.83	14.12
0.75	1.00	1.96	3.86	7.22	13.71	13.99
1	1.00	2.01	3.86	7.20	13.63	14.01

Table 18: Scan - NUMA Aware
Server

-	1	4	8	24	48	96
0.01	1.00	3.46	6.06	6.41	6.06	5.27
0.1	1.00	3.43	6.09	6.45	6.04	5.27
0.25	1.00	3.47	6.04	6.43	6.09	5.24
0.5	1.00	3.47	6.07	6.46	6.06	5.21
0.75	1.00	3.42	6.09	6.41	6.03	5.27
1	1.00	3.48	6.04	6.43	6.07	5.21

Table 19: Scan - NUMA Agnostic
High-end Server

-	1	4	8	24	48	96
0.01	1.00	3.90	7.29	19.33	27.87	20.66
0.1	1.00	3.88	7.37	19.32	26.92	21.02
0.25	1.00	3.90	7.46	19.32	26.62	20.31
0.5	1.00	3.90	7.40	19.48	26.92	21.02
0.75	1.00	3.91	7.40	19.32	26.33	20.31
1	1.00	3.92	7.51	19.16	27.22	20.65

Table 20: Scan - NUMA Aware
High-end Server

-	1	2	4	8	16	32
0.01	1.00	1.28	2.10	2.98	3.05	1.94
0.1	1.00	1.53	2.65	4.06	4.42	3.78
0.25	1.00	1.60	2.77	4.31	4.63	4.26
0.5	1.00	1.63	2.79	4.41	4.73	4.48
0.75	1.00	1.67	2.82	4.46	4.79	4.59
1	1.00	1.79	2.90	4.34	4.77	4.62

Table 21: Grid Index - NUMA Agnostic
Random Accesses - Server

-	1	2	4	8	16	32
0.01	1.00	1.44	2.19	2.88	2.97	1.70
0.1	1.00	1.72	2.83	4.15	4.37	3.66
0.25	1.00	1.67	2.91	4.36	4.64	4.20
0.5	1.00	1.73	2.94	4.44	4.74	4.45
0.75	1.00	1.76	2.91	4.47	4.79	4.53
1	1.00	1.79	2.92	4.43	4.79	4.60

Table 22: Grid Index - NUMA Agnostic
Clustered Accesses - Server

-	1	2	4	8	16	32
0.01	1.00	1.74	2.71	2.98	2.71	1.79
0.1	1.00	1.85	3.14	3.96	4.11	3.80
0.25	1.00	1.87	3.15	4.08	4.25	4.25
0.5	1.00	1.88	3.19	4.17	4.33	4.46
0.75	1.00	1.90	3.20	4.16	4.33	4.53
1	1.00	1.82	2.96	3.97	4.20	4.42

Table 23: Grid Index - NUMA Aware
Random Accesses - Server

-	1	2	4	8	16	32
0.01	1.00	1.77	2.56	2.63	2.42	1.39
0.1	1.00	1.92	3.09	3.87	3.90	3.51
0.25	1.00	1.91	3.15	4.01	4.10	4.02
0.5	1.00	1.93	3.16	4.08	4.21	4.30
0.75	1.00	1.92	3.19	4.14	4.25	4.41
1	1.00	1.88	3.04	4.02	4.19	4.39

Table 24: Grid Index - NUMA Aware
Clustered Accesses - Server

-	1	4	8	24	48	96
0.01	1.00	2.00	3.95	3.19	2.16	1.11
0.1	1.00	2.60	5.20	5.84	5.04	3.37
0.25	1.00	2.79	5.46	6.46	5.86	4.52
0.5	1.00	2.86	5.59	6.96	6.45	5.30
0.75	1.00	2.91	5.65	7.17	6.66	5.66
1	1.00	3.05	5.45	7.90	7.71	6.60

Table 25: Grid Index - NUMA Agnostic Random Accesses - High-end Server

-	1	4	8	24	48	96
0.01	1.00	2.33	3.05	2.98	1.83	0.85
0.1	1.00	3.06	4.95	6.22	5.02	3.22
0.25	1.00	3.16	5.32	7.00	6.10	4.56
0.5	1.00	3.21	5.49	7.51	6.65	5.37
0.75	1.00	3.22	5.63	7.59	6.84	5.77
1	1.00	3.22	5.54	8.17	7.75	6.62

Table 26: Grid Index - NUMA Agnostic Clustered Accesses - High-end Server

-	1	4	8	24	48	96
0.01	1.00	2.48	3.39	3.25	1.95	1.01
0.1	1.00	2.90	4.88	6.55	4.91	3.46
0.25	1.00	3.12	5.26	7.48	6.03	4.88
0.5	1.00	3.45	5.70	8.05	6.77	5.84
0.75	1.00	3.56	5.93	8.33	7.02	6.28
1	1.00	3.45	5.83	8.57	7.50	6.60

Table 27: Grid Index - NUMA Aware Random Accesses - High-end Server

-	1	4	8	24	48	96
0.01	1.00	2.59	3.05	2.64	1.45	0.73
0.1	1.00	3.04	4.83	6.13	4.55	2.99
0.25	1.00	3.24	5.27	7.27	5.84	4.54
0.5	1.00	3.54	5.72	7.88	6.62	5.56
0.75	1.00	3.67	5.97	8.22	6.86	6.01
1	1.00	3.62	6.00	8.55	7.48	6.47

Table 28: Grid Index - NUMA Aware Clustered Accesses - High-end Server

[12] M. Ivanova, M. Kersten, and S. Manegold. Data Vaults: A symbiosis between database technology and scientific file repositories. In *Scientific and Statistical Database Management*, volume 7338. Springer, 2012.

[13] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In *Proceedings of the 17th British National Conference on Databases: Advances in Databases*, BNCOD 17, pages 20–35, London, UK, UK, 2000. Springer-Verlag.

[14] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 1986, pages 197–206, New York, NY, USA, 1986. ACM.

[15] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[16] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 259–270. ACM Press, 1996.

[17] S. Shekhar and H. Xiong. *Encyclopedia of GIS*. Springer Publishing Company, Incorporated, 2007.

[18] P. van Oosterom, O. Martinez-Rubi, M. Ivanova, M. Horhammer, D. Geringer, S. Ravada, T. Tijssen, M. Kodde, and R. Gonçalves. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers & Graphics*, 49:92 – 125, 2015.