A column-store meets the point clouds

Oscar Martinez-Rubi
NLeSC Amsterdam
The Netherlands
o.rubi@esciencecenter.nl

Martin L. Kersten
CWI Amsterdam
The Netherlands
martin.kersten@cwi.nl

Romulo Goncalves
NLeSC Amsterdam
The Netherlands
r.goncalves@esciencecenter.nl

Milena Ivanova
NLeSC Amsterdam
The Netherlands
m.ivanova@esciencecenter.nl

ABSTRACT

Dealing with LIDAR data in the context of database management systems calls for a re-assessment of their functionality, performance, and storage/processing limitations. The territory for efficient and scalable processing of LIDAR repositories using GIS enabled database systems is still largely unexplored. Bringing together hard core database management experts and GIS application developers is a sine qua non to advance the state of the art. In particular to assess the relative merits of both traditional row-based database engines and the modern column-oriented database engines.

1. INTRODUCTION

The GIS application field is quickly expanding. Macro structures such as urban modeling, land use exploration, road/rail infrastructure maintenance are combined with micro structure analysis, such as factory infrastructures, in car sensors, and historical sites. The input for model based analysis is increasingly coming from modern scanning technology, such as LIDAR (Light Detection and Range), which produces huge point clouds.

Modern GIS database platforms, such as PostGIS, Oracle Spatial and Graph, and the ESRI toolkit, contain a plethora of features to handle anything from regular raster images up to highly complex geometric models. However, the scale at which point clouds are produced is hitherto an area largely unexplored in these generic data management platforms. This applies even more to the computational requirements to distill the geometric models from the raw point cloud data using declarative query languages.

The Netherlands eScience Center (NLeSC)[1] has taken the lead to start a few national projects to push the technology envelop by calling for innovations close to the core of database management systems in the context of concrete urban and historic applications.

This work-in-progress paper sheds some light on the possibilities a modern column store can offer and the challenges it needs to deal with. Column stores have become the de-facto standard for managing large data warehouses. All major database vendors (Oracle, IBM, Microsoft) provide columnar extensions and some vendors (SAP HANA) are straight column store implementations. Although column stores have a proven track record in business analytics, their pros- and cons- for GIS applications are not yet well understood.

Our approach centers around understanding the impact of the point cloud data on the different layers of a Database Management System (DBMS). It touches key issues from (adaptive) data loading to optimization of queries over point clouds. The preliminary results obtained through a micro benchmark illustrate both the capabilities to handle point cloud queries efficiently, but also the relative merits of traditional index structures and compression techniques on the performance characteristics. The results are obtained from two easily accessible open-source database systems, MonetDB and PostgreSQL, but are indicative on what to expect from other systems as well.

The remainder of the paper is organized as follows. Section 2 provides a short introduction about the column-store technology and the architecture of the column-store MonetDB in particular. In Section 3 we summarize the challenges posed by point clouds on the DBMS. The preliminary evaluation in Section 4 shows the promising results. We conclude with an outlook in Section 5.

2. BACKGROUND

In this section we identify the major differences between column-stores and row-stores and why they are suitable for GIS systems. At the same time we identify the features of MonetDB, an open-source column-store, which offer advantages in comparison to other system in the family.

2.1 Column-stores

In the recent years we have seen the introduction of a number of column-oriented database systems [10, 17]. For read-intensive analytical processing workload, such as the ones encountered in data warehouses, a column-store offers order-of-magnitude gains compared to traditional row-store architectures.

The most relevant optimizations specific to column-oriented DBMSs are late materialization, block iteration and column-specific compression [5]. For late materialization, columns read off disk are joined together into rows as late as possible in a query. Together with block iteration, i.e., multiple values from a column are passed as a block from one operator to the next, vectorized query processing is achieved. Instead of using Volcano-style per-tuple iterators as in row-stores [9], values of fixed-width are iterated through as an array. For column-specific compression, column-stores can

[1] http://www.esciencecenter.nl/

take advantage of techniques, such as run-length encoding, for direct operation on compressed data when using late-materialization plans [6].

Each optimization, depending on the workload, has different performance impact. With compression, when possible, column-stores are order-of-magnitude faster than row-stores, but not in all the cases. The block iteration optimization offers about factor of 1.5 improvement on average, while late materialization offers about factor of 3 performance improvement in most of read-intensive analytical processing workloads [7].

In addition to being a mature representative of the column-store family, MonetDB was chosen for its specific features, such as using the operator-at-the-time paradigm, partitioned execution for efficient many-core parallelism, and in-memory secondary indexes. Such features help MonetDB to outperform PostgreSQL on our evaluation, Section 4.3.

2.2 MonetDB

MonetDB is a modern in-memory column-stored database system, designed in the late 90's with a proven track record in various fields [8, 15] [2]. Its software stack consists of three layers. The bottom layer is formed by a library that implements a column storage execution engine, including a rich set of highly optimized relational operators. The middle layer provides a convenient abstraction over the kernel libraries, and a concise programming model for plan generation and execution. The top layer consists of an SQL compiler and data management system.

All SQL queries are translated into a parametrized version by factoring out all constants. Such query templates can be re-used more easily at the cost of better run-time optimization. Furthermore, it exploits an operator-at-a-time execution paradigm where complete intermediates are a by-product of every step in the query execution plan. Full materialization benefits fast algorithms and it enables easy re-use of intermediates among the queries [13]. Furthermore, MonetDB exploits partitioned parallelism which offers much better opportunities for speedup and scale up than *pipeline* parallelism. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big query into many independent little jobs. This is an ideal situation for speed up and scale up [11].

2.3 MonetDB's in-memory indexes

MonetDB has an unconventional approach towards index management. Unlike contemporary wisdom, the system itself decides what and how to index at runtime. The rational behind this choice is the design for memory intensive systems, where re-creation of an index in memory is often much cheaper than maintenance of a persistent copy on disk. The indices are maintained under update, but will not be kept on disk.

MonetDB uses column imprints [16], a novel and powerful, yet lightweight, cache conscious secondary index. A column imprint is a collection of many small bit vectors, each indexing the data points of a single cacheline. An imprint is used during query evaluation to limit data access, and thus minimize memory traffic. The compression for imprints is CPU friendly and exploits the empirical observation that data often exhibits local clustering or partial ordering as a side effect of the construction process. Most importantly, column imprint compression remains effective and robust even in the case of unclustered data, while other state-of-the-art solutions fail. The storage overhead, when experimenting with real world data sets, is just a few percent over the size of the columns being indexed.

[2]The system including our extensions can be downloaded from http://monetdb.cwi.nl

3. THE LIDAR CHALLENGE

In this section we elicit the DBMS issues raised by handling LIDAR data.

3.1 LIDAR data management

Airborne laser scanning technology makes it easy to collect large amounts of point data sampling the elevation of the terrain beneath, as well as using stationary laser scanner devices to gather the ground-based structures. The LAS file format has become the defacto standard for storing and distributing the acquired points [1]. As the sampling density of LIDAR sensors increases so does the size of the resulting files. Typical LAS files contain millions of points today, but soon billions will be commonplace.

The LIDAR sensors measure many variables and these vary by sensor and capture process. Some data sets might contain only X, Y, and Z values. Others will contain many more variables: X, Y, Z; intensity and return number; red, green, and blue values; and others. The challenge for a point cloud database management system extension is efficiently storing this data while allowing fast access to these variables. The complexity in handling LIDAR comes from the need to deal with multiple variables per point.

3.2 Compressed LIDAR

Column stores excel in using various public and proprietary compression schemes. The prime reason is to reduce the IO cost when accessing large blocks on slow secondary storage or remotely over the network. Unlike traditional systems, they either read multiple MB in one go, exploit the opportunities of a compressed file system or rely on the memory mapping capabilities of the underlying operating system. Compression is also a key component in LIDAR data management. The LIDAR data easily results in 0.5GB sized files. An application domain compression scheme reduces LAS files to 5-15% of their original size [2]. The LASzip compressor is loss-less, non-progressive, streaming, order-preserving, and provides random-access.

3.3 Query workload

In our initial exploration we purposely limited ourselves to the core of GIS applications, i.e. retrieving objects in a well-defined spatial area. Furthermore, we do not tweak the internals of either system beyond normal database management practices. A spatial index structure is often required to reduce the overhead in scanning disk-based data. However, such structures are becoming less of an issue in the memory sizes of modern servers, which easily reach >1TB of RAM. More coarse grain index structures and exploitation of the tens of processor cores are to be called for.

4. PRELIMINARY EVALUATION

In this section we report on our preliminary experiments using the latest MonetDB software release (Jan2014) and PostgreSQL 9.3.2 / PostGIS 2.1.1 with PointCloud 1.0 extension [3]. Both systems support the GEOS 3.4.2 library and use LASzip 2.1.0 for loading. The hardware configuration is a HP DL380p Gen8 server (2 x 8-core Intel Xeon processors, E5-2690 at 2.9 GHz, 128 GB main

memory) and RHEL 6 operating system. The storage system used in the experiments is comprised of 88 TB SATA 7200 rpm in RAID 6 configuration.

4.1 Database loading

A hindrance for large scale adoption of database management systems in handling point clouds is the time it takes to bulk-load the data into the proprietary database structures. Often this comes with a size explosion, because the systems are generally poor in handling compressed data files.

Unfortunately, most database systems still require a decompression step and bulk loading operation before they can access the data itself. MonetDB is no exception to this rule at this moment, although good progress in adaptive loading is underway [12]. PostgreSQL uses an external tool called PDAL (see http://www.pdal.io) to load the point cloud data into the database. The specially designed point cloud module retains part of the compression opportunities by blocking large number of points in chunks.

The first experiment sheds some light on the scope of the problem. We acquired from our companion project [3] a sample of the Dutch elevation map, called the Algemene Hoogtekaart Nederland [4]. This data set can be freely used for experimentation. We considered several sizes where each data set includes the previous one:

Data set	Points	LAZ files	Disk size
20M	20,165,862	1	37 MB
210M	210,631,597	17	366MB
2201M	2,201,135,689	153	3310 MB
23090M	23,090,482,455	1492	35673 MB

Table 1: Data sets description

In this test we compare MonetDB using a flat table with the point cloud solution of PostgreSQL which is based on grouping the points in blocks. For the latter we used 3000 points per block and dimensional compression (see [3] for details on the used compression). In table 2 we show the loading details for the different databases and data sets.

DB	Data set	Time[s]	Size [MB]	
			Total	Index
MonetDB	20M	78.77	517	56
MonetDB	210M	809.70	4956	136
MonetDB	2201M	8378.05	50838	459
MonetDB	23090M	88395.00	545177	16678
PostgreSQL	20M	82.76	102	1
PostgreSQL	210M	857.15	1012	5
PostgreSQL	2201M	8452.95	10268	52
PostgreSQL	23090M	125797.82	106422	561

Table 2: Data loading performance for LAZ files

The times of MonetDB and PostgreSQL are similar and they scale well in the file sizes. In MonetDB the dominant part of loading stems from the conversion of the LAZ files into CSV format, i.e. around 15 sec for 20MB, and the subsequent parsing of the CSV records by the database engine. On the other hand PostgreSQL uses the PDAL tool to directly read from LAS/LAZ files. It also illustrates the effectiveness of the PostgreSQL PointCloud block-based storage structure, which achieves a factor of 5 compression over the fully exploded storage structure of MonetDB. A secondary index is created by MonetDB the first time it encounters a query. It uses the imprint indexing scheme, which comes with a 5-12% storage overhead.

In addition we tested PostgreSQL using a flat table with and without the spatial functionality provided by PostGIS. However, we decided to skip them for this comparison because the loading times and the required storage were much higher than the solutions that we present here. Concretely, the loading times were up to ten times higher than the other solutions and the storage requirements were up to five times larger than the requirements of MonetDB.

4.2 Database querying

To assess the state of the art in using (open-source) DBMS as a geo-spatial filtering device, we ran a micro benchmark of just four SQL queries against the height map of the Netherlands:

Q1:Small rectangle, axis aligned, 51 x 53 m
Q2:Large rectangle, axis aligned, 222 x 223 m
Q3:Small circle at (85365 446594), radius 20 m
Q4:Large circle at (85759 447028), radius 115 m

Table 3 shows the SQL phrasing for these straightforward queries. We purposely assembled the result set as a table at the server side to get a glimpse on their internal performance behavior. Sending the result set to the client merely blurs the picture with additional IO. In PostgreSQL the different queries are defined as Well-Known-Text (WKT) and they are loaded in a table as PostGIS geometry types.

MonetDB Rectangle

```
CREATE TABLE results (x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO results SELECT * FROM flat WHERE
 x between [x0] and [x1] and y between [y0] and [y1];
```

MonetDB Circle

```
CREATE TABLE results (x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO results SELECT * FROM (SELECT * FROM flat
 WHERE (x between [cx]-[r] and [cx]+[r])
   AND (y between [cy]-[r] and [cy]+[r])) a
WHERE power(x-[cx],2) + power(y-[cy],2)<power([r],2);
```

PostgreSQL Rectangle - Circle

```
CREATE TABLE results AS (SELECT PC_GET(qpoint,'x') as x,
 PC_GET(qpoint,'y') as y, PC_GET(qpoint,'z') as z FROM (
  SELECT PC_EXPLODE(PC_INTERSECTION(pa,geom)) as qpoint
   FROM patches,qpolygons WHERE PC_INTERSECTS(pa,geom)
     and qpolygons.id=[qid]) as qtable);
```

Table 3: SQL phrases for the different databases and queries

The results of these experiments are shown in Table 4. The difference in the number of points in the circles is due to a polygon approximation used in PostgreSQL.

Each query is ran twice, i.e., in a cold state (Run 1) and a warm database state (Run 2). During the first query of the first run, MonetDB takes the time to built an imprint index, which is subsequently used in any filter operation against the table. The current implementation creates an imprint for individual columns, which is suboptimal for spatial search where range conditions on 2 and 3 dimensions are often used. The 23090M run for MonetDB illustrates a situation when the system fails to create the imprints index and falls back to a linear scan.

During a warm run, the in-memory imprint structure in MonetDB has been created and the PostgreSQL buffers are partly filled. For MonetDB we used its SQL trace option to gain insight into the

[3]Massive Point Clouds for eSciences (see http://pointclouds.nl)

DB	Data set	Run	Number of points			
			Q1	Q2	Q3	Q4
MonetDB	*	*	74872	718021	34691	563037
PostgreSQL	*	*	74872	718021	34667	563014
			Time[s]			
MonetDB	20M	1	6.11	0.24	0.11	0.27
MonetDB	20M	2	0.04	0.08	0.05	0.12
MonetDB	210M	1	34.88	0.44	0.36	0.41
MonetDB	210M	2	0.14	0.23	0.15	0.26
MonetDB	2201M	1	201.77	1.26	0.89	1.23
MonetDB	2201M	2	0.67	0.71	0.60	0.81
MonetDB	23090M	1	10.41	7.60	5.37	5.74
MonetDB	23090M	2	5.17	5.18	5.22	5.42
PostgreSQL	20M	1	0.84	7.24	0.44	17.07
PostgreSQL	20M	2	0.82	7.19	0.44	16.98
PostgreSQL	210M	1	0.83	7.06	0.44	16.89
PostgreSQL	210M	2	0.82	7.18	0.46	16.99
PostgreSQL	2201M	1	0.86	6.72	0.47	16.87
PostgreSQL	2201M	2	0.82	7.13	0.46	16.82
PostgreSQL	23090M	1	1.21	6.99	0.63	18.24
PostgreSQL	23090M	2	0.86	7.22	0.52	17.92

Table 4: Query results for cold- and hot- runs

cost factors of the query components. An immediate observation is that MonetDB effectively uses all cores in scanning the LIDAR columns. The PostgreSQL figures indicate that such parallel scans can be an attractive alternative performance-wise.

While PostgreSQL presents constant query times, in MonetDB we observe much faster query times for the smaller data sets and increasing query costs for larger sets that can be fully attributed to scanning a large portion of the imprint index. We foresee multi-column imprint structures as an important direction for improvement. The experimentation platform is large enough to even accommodate the 2201M point cloud challenge but for very large data sets (like 23090M) the query performance in MonetDB decreases considerably. Therefore, for such data sets, alternative storage structures like the blocking approach used by PostgreSQL, should be investigated since they offer more stable query responses.

4.3 Conclusion

From the results above, even though some effort is still required to deal with its current limitations, MonetDB can be considered more modern than PostgreSQL, because it is designed from an in-memory perspective and relies on the operating system to move data between the storage hierarchies in an efficient manner. All queries are also highly parallel, using the cores available wherever possible. On the contrary, PostgreSQL represents the traditional buffer-based and iterator query engine approach. Tuning the buffer size to use all available memory by itself does not help because the logic of chasing data in buffers remains. Furthermore, PostgreSQL does not by default support multi-core query processing.

5. SUMMARY AND OUTLOOK

In this paper we have identified column stores as a viable alternative for point cloud data management. The initial experiments against a well-known open-source column-store, MonetDB, show that out of the box performance is competitive to PostgreSQL with the point-cloud module. The in-memory parallel scans using a single dimension indexing scheme seems effective, re-iterating the experience that fast scans in memory are a viable alternative to the well-known spatial access methods. It does not require special steps during data bulk loading, nor parameter tuning by the database administrator. However, much more leverage can be expected from column store technology. A good example are two on going projects, DataVault and SciQL, which extend MonetDB with new data analysis features in other sciences.

- DataVault [12] is an ongoing project to automate efficient access to foreign file repositories in MonetDB. It has already shown its capabilities in handling FITS (astronomy), MSEED (seismology) and NetCDF (meteorology) file formats. The technology is well suited to be expanded to directly work with LAZ files on a need to know basis for individual queries.
- SciQL [14] is an ongoing project to improve database query expressiveness using ARRAYS as first class citizens. Its prototype implementation has shown its capabilities in e.g. remote sensing, where large regular grids of satellite images are analyzed.

Within the NLeSC projects Massive Point Clouds for e-Sciences and Big Data Analytics in the Geo-spatial Domain we will further deepen our understanding of the potentials of several GIS database management systems. The ultimate goal is to be able to directly query the 640B points comprising the 'flat' Netherlands at a 5 cm height precision and density of 10 points/m^2.

Acknowledgments

We thank the members of the CWI database group and the COMMIT project for their support in extending and releasing MonetDB, as well as the partners from the Massive Point Clouds project for providing the point cloud challenge. This work is in part funded by the research programme of the Netherlands eScience Center.

6. REFERENCES

[1] http://www.asprs.org/a/society/committees/lidar/lidar_format.html.
[2] http://www.laszip.org/.
[3] https://github.com/pramsey/pointcloud.
[4] http://ahn.geodan.nl/ahn/.
[5] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. SIGMOD, 2006.
[6] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.
[7] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. Row-stores: How Different Are They Really? SIGMOD, 2008.
[8] P. Boncz, S. Manegold, and M. Kersten. Optimizing main-Memory join on modern hardware. *TKDE*, 2002.
[9] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. CIDR, 2005.
[10] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 1999.
[11] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 1992.
[12] M. Ivanova, Y. Kargin, M. Kersten, S. Manegold, Y. Zhang, M. Datcu, and D. E. Molina. Data Vaults: A Database Welcome to Scientific File Repositories. SSDBM, 2013.
[13] M. Ivanova, M. Kersten, N. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. *TODS*, 2010.
[14] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. SciQL, a Query Language for Science Applications. AD, 2011.
[15] S. Manegold, M. Kersten, and P. Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. VLDB, 2009.
[16] L. Sidirourgos and M. Kersten. Column Imprints: A Secondary Index Structure. SIGMOD, 2013.
[17] M. Stonebraker, D. J. Abadi, et al. C-store: A Column-oriented DBMS. VLDB, 2005.