CLUSTERING AND INDEXING HISTORIC VESSEL MOVEMENT DATA WITH SPACE FILLING CURVES

Martijn Meijers[1,*], Peter van Oosterom[1]

[1] Delft University of Technology, Faculty of Architecture, OTB, GIS-technology
(b.m.meijers, p.j.m.vanoosterom)@tudelft.nl

Commission IV, WG IV/7

KEY WORDS: vessel movement, space filling curves, clustering, indexing, space-time queries, query framework

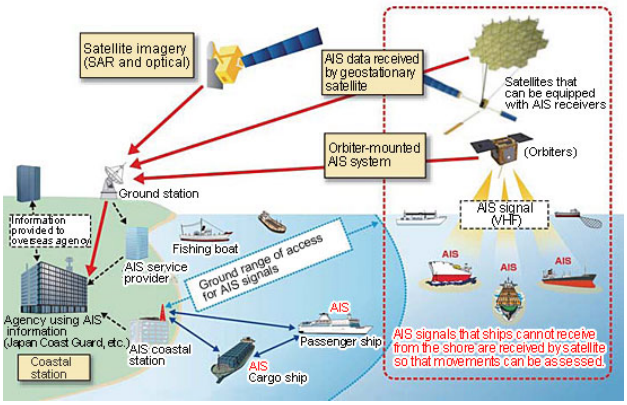
ABSTRACT:

This paper reports on the result of an on-going study using Space Filling Curves (SFCs) for indexing and clustering vessel movement message data (obtained via the Automated Identification System, AIS) inside a geographical Database Management System (Geo-DBMS). With AIS, vessels transmit their positions in intervals ranging from 2 seconds to 3 minutes. Every 6 minutes voyage related information is broadcast.
Relevant AIS messages contain a position, timestamp and vessel identifier. This information can be stored in a DBMS as separate columns with different types (as 2D point plus time plus identifier), or in an integrated column (as higher dimensional 4D point which is encoded as the position on a space filling curve, that we will call the SFC-key). Subsequently, indexing based on this SFC-key column can replace separate indexes (where this one integrated index will need less storage space than separate indexes). Moreover, this integrated index allows a good clustering (physical ordering of the table). Also, in an approach with separate indexes for location, time and object identifier the query optimizer inside a DBMS has to estimate which index is most selective for a given query. It is not possible to use two indexes at the same time — e.g. in case of a space-time query. An approach with one multi-dimensional integrated index does not have this problem. It results in faster query responses when specifying multiple selection criteria; i.e. both search geometry and time interval.
We explain the steps needed to make this SFC approach available *fully inside* a DBMS (to avoid expensive data transfer to external programs during use). The SFC approach makes it possible to better cluster the (spatio-temporal) data compared to an approach with separate indexes. Moreover, we show experiments (with 723,853,597 AIS position report messages spanning 3 months, Sep–Dec 2016, using data for Europe, both on-sea and inland water ways) to compare an approach based on one multi-dimensional integrated index (using a SFC) with non-integrated approach. We analyze loading time (including SFC encoding) and storage requirements, together with the speed of execution of queries and granularity of answers.
Conclusion is that time spend on query execution in case of space-time queries where both dimensions are selective using the integrated SFC approach outperforms the non-integrated approach (typically a factor 2–6). Also, the SFC approach saves considerably on storage space (less space needed for indexes). Lastly, we propose some future improvements to get some better query performance using the SFC approach (e.g. IOT, range-glueing and nD-histogram).

1. INTRODUCTION

AIS stands for Automated Identification System. The AIS system is mainly used for improving safety at sea and inland waters (enabling vessels to discover quickly how other vessels are behaving to avoid collisions) and can also be used as input for traffic management.

Figure 1 illustrates that the system consists of different components: Vessels equipped with Global Navigation Satellite System (GNSS) receiver and AIS transponder, as well as base stations. For seagoing vessels and inland vessels larger than a certain size it is mandatory to carry an AIS transponder. These transponders send up to date information via Very High Frequency (VHF) radio at certain intervals. Depending on the cruising speed of the vessel or whether it is anchored or moored, a transponder broadcasts its identity and position in intervals ranging from 2 seconds to 3 minutes. In addition, every 6 minutes voyage related information is broadcast. Also other devices, e.g. man-over-board devices, can broadcast their position to others nearby. Furthermore, in assigned mode, a base station can request more frequent



Source: Japan Aerospace Exploration Agency

Figure 1. Using AIS equipment vessels automatically broadcast ship specific information, such as their name and type, as well as their position to nearby vessels or coastal stations. On oceans Satellite-based AIS (S-AIS) can be used.

*Corresponding author

updates, or can command AIS transponders to be more quiet. Together with the (implicit) receiving time the AIS position reports form a rather large spatial-temporal data set.

Rijkswaterstaat (RWS) is part of the Dutch Ministry of Infrastructure and the Environment and maintains the main waterway network in The Netherlands. Within RWS Automated Identification System (AIS) messages are received in real time with the Dutch Inland AIS Monitoring Infrastructure (DIAMONIS) network.

The current architecture of this system is not suited for archiving large amounts of historic AIS messages. As AIS messages are received frequently for many vessels, the total data volume is significant. Per week more than 80 million messages are received by DIAMONIS (leading to over 1.5GB of raw message data per week stored in flat text files, see Meijers et al. (2017)).

Compared to a flat text files based organization a Database Management System (DBMS) can offer concurrent user access, so that many users can have access to the same data. It makes permission management possible, which makes it possible to specify which user accounts are allowed to access which data (this might be important given privacy concerns related to AIS message data). Furthermore, the availability of a declarative query language (SQL) allows the rapid development of applications. Easier integration with other types of geographic data (e.g. raster data – making heat maps of traffic intensity) is another advantage.

However, to make fast queries possible a DBMS should provide Spatial Access Methods (SAM): Spatial indexing and clustering techniques. Spatial indexing, which is necessary to guarantee fast access to individual records, will also require storage space. In this research we investigate how indexing and clusterting based on a Space Filling Curve can be implemented in a DBMS for space-time queries and how its performance compares to more conventional indexing structures as B-tree (Bayer and McCreight, 1972) and R-tree (Guttman, 1984).

The remainder of this paper is structured as follows: Section 2 shows recent work using SFCs for indexing and clustering space-time data, Section 3 describes the SFC approach in more detail, Section 4 gives an overview of the experiments we carried out together with analysis of the results and Section 5 concludes the work and gives future work.

2. RELATED WORK

Van Oosterom et al. (2015) introduce the challenge to manage massive point cloud data in a database, present a benchmark, and propose to use a Space Filling Curve (SFC) for efficient 3D point cloud data management with as test case the AHN2 data.

Psomadaki et al. (2016) and Psomadaki (2016) propose to use a SFC for the management of dynamic point cloud data and analyse organizations options: Ranging from 2D (just xy in SFC key) to 4D (xyzt in SFC key) with test case coastal monitoring data. The SFC computation was in both cases programmed outside the Oracle database and implemented in Python (drawback: a lot of data transfer between database and external SFC programs).

Guan et al. (2018) implemented the nD SFC Library[1] in C++ (more efficient than Python) and tested up-to 5D by adding also the importance dimension to the SFC key (xyzti), but expensive data transfer between database and external SFC software remained.

In the above solutions the Oracle database with the Index Organized Table (IOT) was used. A IOT brings several performance benefits: a. data is clustered (in order of index), b. table and index are integrated in single structure, so more compact with less reading needed, c. no combination of 2 structures needed (no ‘join’ of index and table needed), which means less processing.

De Vreede (2016) implemented using Python the SFC key technique in combination with the MongoDB database and tested this with moving ship trajectory data of the Automatic Identification System (system). Also, functionality for computing SFC keys was outside of database. As De Vreede (2016) observed in her thesis that far fewer records needed to be retrieved, while using an integrated multi-dimensional index based on the Morton SFC for organizing AIS messages.

To summarize, for nD-points an organisation approach using Space Filling Curves (SFC) has been proposed. However, this approach has not been implemented fully *inside* a Geo-DBMS. Up-to-now most relevant parts were programs running outside of database server causing a lot of data transfer between DBMS and SFC computation (one of main performance bottlenecks). In this work we use the SFC approach for indexing and clustering AIS points, while running fully inside the PostgreSQL DBMS.[2] PostgreSQL is a mature database system, available under an open source license. Furthermore, it has spatial capabilities by means of an extension, called PostGIS. This extension allows to use spatial data types, spatial indexing and offers spatial predicates (e.g. to determine whether a point is inside a polygon). Although PostgreSQL does not have equivalent of Oracle’s IOT, it allows to cluster a table (sorting table along index order), so quite good performance for data retrieval is to be expected.

3. METHOD

3.1 Objective

In this on-going research, for making fast queries possible on a large historic archive of AIS position message reports, we implement an approach for indexing and clustering based on Space Filling Curves (SFC) inside a Geo-DBMS. Examples of use cases that need a large historic archive of AIS messages related to vessel traffic management are counting passages, travel time analysis between two lines, obtaining traffic densities and intensities, investigating how many tracks of vessels there are that cross each other within a limited time frame (e. g. within 10 minutes), prediction of use of fairway, etc. For these use cases two queries are often a starting point:

1. Location query: Find the position of a set of vessels in a specific time window.
2. Trajectory query: Give the historic positions of a vessel and their corresponding timestamps. The trajectory can be given as a ordered/unordered set of point locations or as set of line segments.

[1] `http://sfclib.github.io/`

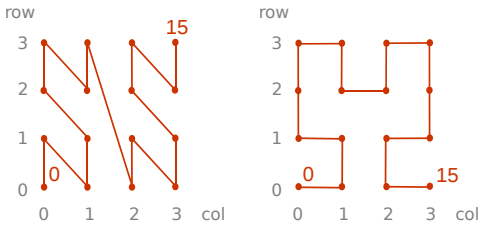
[2] The Python and Rust code for enabling the SFC approach inside PostgreSQL is available at `https://bitbucket.org/bmmeijers/sfc-rs` and `https://bitbucket.org/bmmeijers/sfc-rs-ffi`.

In this paper, we focus on the Location query. Moreover, we take the following requirements into account:

1. Volume of storage has to be efficient and comparable to size of raw AIS message data in text files (and loading times should be reasonable)
2. Spatial-temporal queries can be answered (e. g. give all vessels in geographic region B and inside time window $T, \ldots$)
3. Queries can be executed reasonably efficient (i.e. the database system should be able to give answers within reasonable amount of time)

3.2 Space Filling Curves – Indexing and clustering in a DBMS

Figure 2 illustrates that a Space Filling Curve (SFC) is a curve that traverses linearly through a n-dimensional (nD) grid (hyper cube) with a given resolution. Each cell is in the grid is visited exactly once by the curve. An important property is that locality from the n-D space is preserved in the location on the curve (Dai and Su, 2003). Space Filling Curves, such as Gray-coded curves, Hilbert curves, Peano curves and z-order curves (also known as Morton curve) have been well investigated. In this paper, we only consider the Morton and Hilbert curves. Both are so-called quadrant recursive curves.



(a) Morton (b) Hilbert

Figure 2. Space filling curves.

By means of the SFC, we can map a nD coordinate to a 1D ordinate (i.e. position on the curve). The main idea for using a Space Filling Curve as organization structure is to get fast access to selections of nD points as the records for the points can be sorted along the curve and, because locality is preserved, records close on the curve are also close in nD space (persistent memory), meaning that for retrieval less communication is needed. In an approach with separate indexes for location, time and object identifier, the query optimizer has to estimate which index is most selective for a given query. Such an index is then used to determine which records from the table fulfill the selection criteria of the query. It is not possible to use two indexes at same time, e. g. in case of a space-time query. Indexing based on a 1D Space Filling Curve position (which we call the ‘SFC-key’) can integrate separate indexes and thus allow clustering in a balanced way (space and time at the same time).

For this to work, it is necessary to scale and translate each dimension of every input point to a nD hyper cube. Then we can encode this nD position (of integers) to a 1D location value (integer) on the Space Filling Curve (the SFC-key, distance from the start of the curve). For encoding (and decoding) we implemented 2 functions in the database:

- `sfc_encode`, taking as input an array of integers (the nD-coordinate), returning an integer, the SFC-key.
- `sfc_decode`, taking as input an SFC-key (integer) and an integer specifying the number of dimensions, returning an array of integers (original nD-coordinate).

For the Morton curve the SFC-key can be obtained from the integer coordinates of the nD point by interleaving the bits of the coordinates. For the Hilbert curve, the process of producing a SFC-key also involves rotating and mirroring the bit pattern at subsequent levels in the tree (Guan et al., 2018).

After calculating the SFC-keys for all points (according to Hilbert or Morton order) and storing the result in additional column in the table, we can index this SFC-key column by using a B-tree index. Note, it would also be possible to only store the SFC-keys, this means that only the SFC-key is stored and not the related attributes (which are already encoded in the key). To obtain the attribute values, we can use the decode function, e. g. using this function inside a database view.

A custom query procedure with filter and refine steps is implemented to make efficient use of the SFC-key (cf. Psomadaki (2016), §4.4). Space-time queries (nD hyper box) need to be translated to ranges of SFC-key values, that then are joined to the original data table (where the B-tree index is used to make this join fast). The query procedure uses the relationship that exists between quadrant recursive SFC-keys and 2^n-trees: A SFC-key gives location in an nD grid. The total number of cells in the grid is determined by the amount of bits that are used (per dimension). Furthermore, a linear order of visiting the grid cells is defined by all SFC-keys. As also noted by Van Oosterom and Vijlbrief (1996) and Baert et al. (2013), a 2^n-tree has a relationship with this linear order. The leaf nodes in the corresponding tree (which in 2D is a quadtree and in 3D is an octree) are encountered in this order, when the tree is traversed depth-first, post-order. Every n bits, starting at the most significant bits of the SFC-key describe which nD cell at that level overlaps with the SFC-key: A cell its index at a higher level in the tree thus corresponds to a coarser version of the highest resolution nD grid. Hence, a full resolution SFC-key gives a full path from the root to the leaf nodes (a highest resolution cell) in the tree. For range searching, we take the tree as starting point: With a given query geometry, descend the 2^n tree and perform geometric overlap tests of the cells of the tree, remember the traversed path. If you stop the traversal of the tree at interior node of the tree, this means that the start of the SFC range is known and the depth gives also the length of the range (so the end of the range can be determined).

3.3 Experiment setup

We used 723,853,597 AIS position messages, with their coordinates (ϕ, λ) only in the positive quadrant ([0°, 90°], [0°, 180°]). The messages are spanning 3 months of time, Sep–Dec 2016.

SFC approach Appendix B gives full details for the steps to setup the SFC table and related B-tree index. Loading data using the SFC approach means performing the following few steps:

1. Load the AIS message data in a staging table (unpack its 2D point and MMSI from the NMEA message, see Appendix A on AIS messages for more details), together with their timestamp

2. Make SFC-key per record (scale and translate coordinates of point and timestamp plus cast to integer, as SFC-key needs to be computed based on integers), using the `sfc_encode` function.
3. Make B-tree index on SFC-key column
4. Cluster the table on the B-tree index (sort the records of the table in the order of their position on the SFC)
5. Vacuum the table (reclaim storage space cluster/ sorting step)

Result is a table with 4 columns: AIS point, timestamp, object identifier (MMSI number), and also the added column to hold the SFC-key value. We created 2 variants, using the Morton SFC-key and Hilbert SFC-key. We scaled and translated the data for each dimension to use at maximum 21 bits (to be able to fit SFC-key for 3 dimensions in 64 bit number), finest space resolution: $\pm$ 2 cm and finest resolution for time: $\pm$ 3 seconds.

Reference approach To compare the SFC approach to, we have also created a regular table with 3 columns: AIS point, timestamp and object identifier (MMSI number). For the regular table we created 4 variants:

a. A table with 2 indexes on space and time column, clustered on space (variant 2S)
b. A table with 2 indexes on space and time column, clustered on time (variant 2T)
c. A table with 1 index on space column, clustered on space index (variant 1S)
d. A table with 1 index on time column, clustered on time index (variant 1T)

For the space index we used the available R-tree of PostGIS, and for the time index we used the available B-tree of PostgreSQL.

4. RESULTS

For the tests described in this document we have used a server with the following details: HP DL380p Gen8 server with 2$\times$ 8-core Intel Xeon processors, E5-2690 at 2.9 GHz, 128 GB of main memory, and a RHEL 6 operating system. The disk storage, which is directly attached, consists of a 400 GB SSD, 5 TB SAS 15K rpm in RAID-5 configuration (internal), and 41 TB SATA 7,200 rpm in RAID-5 configuration (in a Yotta disk cabinet). The PostgreSQL version used is 10.1 and the version of PostGIS is 2.4.2.

4.1 Load times

Figure 3 gives an overview of the data, and also indicates the regions we used for the queries. Table 1 shows how much time it takes to load, index, cluster and vacuum all 723,853,597 records. As can be seen from Table 1, it is slightly more work to compute SFC-keys according to the Hilbert curve compared to the Morton curve (0.4 hour more). As comparison we also loaded the same records in a regular table and created relevant indexes. It is interesting to see that clustering is quite an expensive step when R-tree is available, but is very fast when only the B-tree is present. Note, that the absence of an index could result in long query times, in case a spatial or temporal selection is required. In this sense, it is unrealistic that both variant 1S and 1T will perform well for a variety of queries.

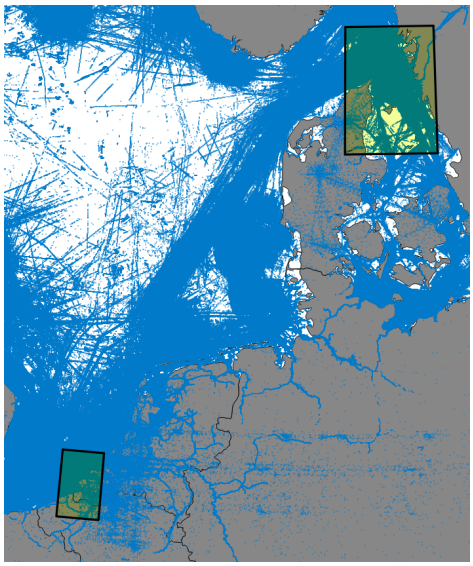


Figure 3. A visualization of the AIS position reports that we used as test data. The regions we used for queries are indicated by the yellow rectangles. Map projection: ETRS89 / LAEA Europe.

4.2 Data size

Table 2 shows that the integrated SFC approach (data plus B-tree) saves storage space. The table with added SFC-key needs more storage space (5GB extra in our case), due to the addition of the SFC-key column. However, we only need one B-tree index on the SFC-key column, which is leaner than R-tree on geometry and B-tree on timestamp columns together for regular table. In total it is approximately 30% cheaper to store the SFC approach with integrated space-time index, compared to regular table with separate indexes for space and time. Note, in case of SFC-key only storage the AIS point and timestamp column could be removed from the table, resulting in additional gains in space.

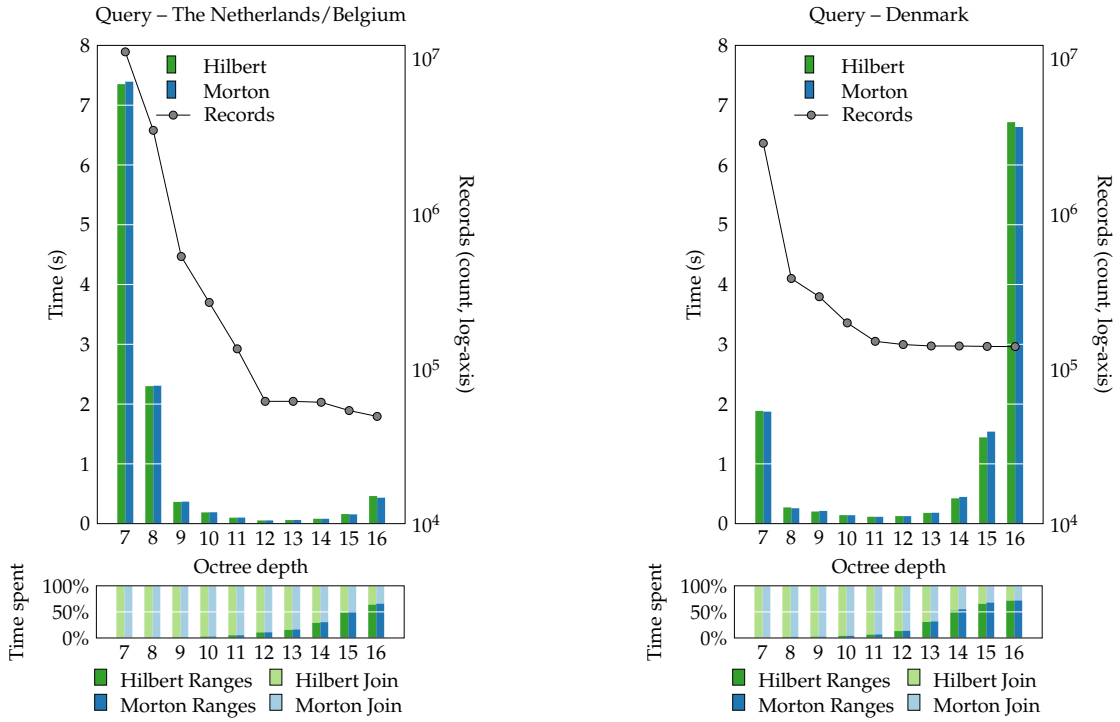
4.3 Query times

For the area in the south overlapping with The Netherlands/ Belgium (3% of the total records lie in the query region), we used a time window of 3 hours (accounting for a 0.1% of the total number of records). For the region in the north overlapping with Denmark/ Sweden (1% of total records within box), we used a time window of 24 hours (selection on this time window results in 1% of total records). As Table 3 indicates, the SFC approach, in our case with space-time query, outperforms an approach with separate R-tree and B-tree indexes with a factor 2 to 6, even with more records retrieved for the answer (as the SFC approach approximates the query geometry).

Figure 4 show the results of the two queries we performed with the SFC approach to analyze in more detail what happens. The bar chart at the top shows how much time the complete query procedure takes in seconds, where each green bar refers to a SFC-key based on the Hilbert curve and each blue bar is based on the Morton curve. The x-axis indicates at what level the traversal of the 2^n tree was stopped to generate SFC ranges. The bar chart below the main graph shows whether time is spent on the SFC range generation step, or on the join of the ranges to the table with SFC-key column.

		SFC		Regular			
		Morton	Hilbert	2S	2T	1S	1T
Table		4.8	5.2	0.2	0.2	0.2	0.2
Index	· SFC-key	0.1	0.1				
	· point			4.1	4.0	4.1	
	· timestamp			0.1	0.1		0.1
Cluster on	· SFC-key	0.3	0.3				
	· point			3.8		7.2	
	· timestamp				6.3		0.2
Vacuum		0.7	0.7	0.8	0.8	0.7	0.7
Total time		5.2	5.6	8.8	11.1	12.2	1.2

Table 1. Loading 723,853,597 records, time in hours



(a) The Netherlands / Belgium. 3 hour time window. Fastest result (with: 61,290 resulting records) is obtained at depth 12, taking 54 milliseconds for the complete query process.

(b) Denmark. 24 hour time window. Fastest result is obtained at depth 11 (generating 149,188 records), taking 115 milliseconds.

Figure 4. Query results using the SFC approach.

		SFC	Regular	Δ SFC - Regular
Table		57	52	+5
Index	· sfc	15		-36
	· geo		36	
	· ts		15	
Total		72	102	-31

Table 2. Size comparison (in GB), comparing the SFC approach against the regular indexed tables

	Denmark			The Netherlands		
Variant	depth	records	time	depth	records	time
Morton	11	149,188	0.115	12	61,290	**0.053**
Hilbert	11	149,188	**0.114**	12	61,290	0.054
1S		137,658	3.35		44,924	6.20
1T		137,658	2.57		44,924	**0.357**
2S		137,658	3.06		44,924	3.06
2T		137,658	**2.51**		44,924	0.362

Table 3. Query time comparison. Time in seconds. Fastest time per approach in bold. The amount of records that is retrieved by the regular approach is the ‘exact’ answer. As the SFC approach approximates the query geometry, more records are retrieved (8% more for Denmark query, 38% more for The Netherlands query).

The line chart at the top shows how much records were retrieved by the query (note, logarithmic scale on the right of the chart). Again, the SFC approach approximates the query geometry, so query result will contain more points than the ‘exact’ answer. From the graphs it is clear, that the result starts to approach the exact query answer (as obtained by regular approach) better with a higher depth value.

Some more observations can be made:

- Descending too little or too much: Both is not good. Too little: the join step is extremely expensive as too coarse ranges, this leads to many points to be retrieved (for both queries time is completely dominated by join step for level 7). Too much: SFC range generation becomes expensive (nearly 75% of time is caused by generating ranges for level 16) and this refinement does not contribute to more exact answer (roughly same amount of records will be retrieved).
- The ‘correct’ depth seems to be the level where the reduction of selected records does not go down drastically any more (you might possibly know this point by having a nD histogram at hand).
- Although there is one depth that clearly gives fastest query result, there exists a range of depths where query is reasonably fast (e. g. depth 9, 10, 11, 12, 13 all produce answer within 0.5 second).
- Currently, there does not exist a notable difference between Hilbert and Morton (although Hilbert potentially is a bit more expensive to compute, as was also apparent from loading time, cf. Table 1).
- SFC approach gives approximate answer: Exact answer plus some additional records. Second filter step (which performs a point in polygon test) should have been applied to generate answer with same number of points as exact outcome. However, we did not yet test this.
- For the fastest query (at level 11, 12), currently most time (more than 90%) is spent in join phase. This could allow for speeding up the query. We can join consecutive ranges (cf. Psomadaki (2016), §4.4.3) making the join step faster. Expectation is that Hilbert would benefit more from this optimization compared to Morton, as Hilbert curve is more compact (less big jumps in curve, better locality property).

In short, tuning the granularity of SFC ranges is required for querying with the SFC approach. An optimal exists between generating not so many, but long SFC ranges (easy to generate the ranges, yet quite some work for data join) and generating shorter, but many more SFC ranges (which is more work to generate the ranges, but at same time can be more selective for data join step). Furthermore, there is some room for further improvement by joining consecutive ranges.

5. CONCLUSION AND FUTURE WORK

We have implemented for the first time the SFC approach *fully* inside the database and have shown that it is possible to implement the whole approach inside the database server. The SFC approach saves on storage space compared to regular table together with multiple indexes approach. Tight integration within the database server has clearly advantages: No data transfer between external program, leading to efficient load and performant queries, that are faster than using conventional/regular approach for storing space-time data (with separate indexes).

To make the SFC approach work, we have implemented SFC encode, SFC decode and SFC range generation functions. Moreover, we have used the B-tree, cluster and join functionalities, already supplied by the DBMS. Also, per dataset we have to scale and translate the coordinates and round to integer grid.

Although the SFC approach is very promising, we have a list of points to address in the future:

- Investigate SFC-key-only storage, plus additional database view for decode (and its impact on data storage and performance).
- In 4D (and higher) size of SFC key gets too big to be represented by 64-bit integer. Other key encodings need to be used (e. g. raw bytes, varchar).
- To make the encoding / decoding work, we apply scaling and translation (offset) to all dimensions (xyt) of the 3D input points. Although these are simple numerical operations, it is important that the scaling and translation that is applied at load time, is the same when querying the data. Storing these values inside a metadata table in the database, and let the encode / decode / ranges functions use this metadata will make the approach more robust (less chance of inputting wrong values by end user) and more user friendly.
- Using the non-obvious dimension ‘object identifier’ inside SFC-key (leading to a 4D point). Will the resulting clustering be good for performing queries?
- As was shown, chosing correct depth in 2^n-tree is crucial for querying in a successful and efficient way with SFC approach. This tuning of SFC query approach has been manual. In future, this should be more automated / transparent for end user: A histogram on the data distribution for various levels in the 2^n-tree can help. Implement and test this idea.
- The queries we used are space-time queries with both the space and time dimension reasonably selective. In case space is given, but not time for making the query (complete time window present in data set to be considered for answer), we also want to generate an answer reasonably fast (although might result in many records, so expensive to generate anyway). Test what happens in such a worst case.
- A second filter step (point-in-polygon-test) was not applied in this work. It should be tested how much work the point-in-polygon-test entails. Note that the SFC ranges can be annotated with whether they are fully inside query geometry or on boundary (as query geometry is approximated by them). Only the SFC ranges that are overlapping with the boundary of query geometry need to be evaluated against second filter.
- As mentioned before, consecutive SFC ranges can be glued together. This will reduce the amount of work for the join step (as the join step needs to consider less ranges). Hypothesis: Hilbert will perform better than Morton, due to better locality. Note that also, non-consecutive ranges (with small gap between them) can be glued together. However, this will increase false positives for query result (and these records need to be removed by a second filter step).
- Test the SFC-approach with an Indexed Organized Table (IOT). This type of structure with integrated index and table is not supported in PostgreSQL at the moment, but this could be tested with Oracle DBMS.

ACKNOWLEDGEMENTS

This work has been carried out in the framework of the ‘RWS-TUD Raamovereenkomst betreffende Samenwerking en Kennisuitwisseling op gebied van Ruimtelijke Informatievoorziening’ (Reference 31103836).

We thank our colleagues Dongliang Peng for helping with making the charts and Haicheng Liu for suggesting the use of a nD histogram while querying.

REFERENCES

Baert, J., Lagae, A. and Dutré, P., 2013. Out-of-core construction of sparse voxel octrees. *Proceedings of the 5th High-Performance Graphics Conference on HPG ‘13*.

Bayer, R. and McCreight, E. M., 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1(3), pp. 173–189.

Dai, H. K. and Su, H. C., 2003. On the Locality Properties of Space-Filling Curves. In: *Algorithms and Computation*, Springer Berlin Heidelberg, pp. 385–394.

De Vreede, I., 2016. Managing Historic Automatic Identification System data by using a proper Database Management System structure. Master’s thesis, Delft University of Technology.

Guan, X., van Oosterom, P. and Cheng, B., 2018. A Parallel N-dimensional Space-Filling-Curve Library and Application in Massive Point Cloud Management. Manuscript under preparation, to be submitted to ISPRS Int. J. Geo-Inf.

Guttman, A., 1984. R-trees. *ACM SIGMOD Record* 14(2), pp. 47.

International Telecommunication Union, 2014. Recommendation ITU-R M.1371-5: Technical characteristics for an automatic identification system using time-division multiple access in the VHF maritime mobile band.

Meijers, M., Quak, W. and van Oosterom, P., 2017. Archiving AIS messages in a Geo-DBMS. In: A. Bregt, T. Sarjakoski, R. van Lammeren and F. Rip (eds), *Proceedings of the 20th AGILE Conference on Geographic Information Science*, Wageningen University & Research, p. 3.

Meijers, M., van Oosterom, P. and Quak, W., 2016. Management of AIS messages in a Geo-DBMS. Technical report, Delft University of Technology.

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.

Psomadaki, S., van Oosterom, P. J. M., Tijssen, T. P. M. and Baart, F., 2016. Using a space filling curve approach for the management of dynamic point clouds. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* IV-2/W1, pp. 107–118.

Raymond, E. S., 2016. AIVDM/AIVDO protocol decoding.

Van Oosterom, P. and Vijlbrief, T., 1996. The Spatial Location Code. In: M. J. Kraak and M. Molenaar (eds), *Proceedings of the 7th International symposium on Spatial Data Handling: Advances in GIS research*, pp. 1–12.

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

APPENDIX A – AIS MESSAGES

The International Telecommunication Union (2014) defines 27 different top level AIS message types. These messages can be classified in 6 different main groups: 1. position messages, 2. meta data messages, 3. acknowledgment messages, 4. defined slot binary messages, 5. addressed messages and 6. broadcast messages. Note that not all message types are received as frequent as others. The National Marine Electronics Association (2012) defines an ASCII encoding, such that AIS messages can be sent over a serial link to other equipment. In NMEA terms, an AIS message is a group of sentences.

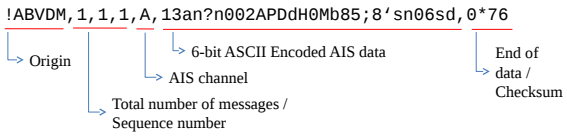


Figure 5. NMEA sentence containing AIS position report

Figure 5 shows that such a sentence contains the origin type (which codes the origin of the sentence, in this case ‘AB’ in ‘!ABVDM’ stands for base station), the number of sentences a message consists of (a message can span multiple NMEA sentences), the AIS channel that was used to broadcast the message, the raw AIS data and a check sum to check the integrity of the raw message. Which parameters are contained in a raw message depends on the message type.

A base station can add this time stamp to the NMEA sentence upon reception of an AIS message. Many of the AIS message types contain a geographical location (latitude, longitude) and an object identifier of the sender, the MMSI number. MMSI stands for Maritime Mobile Service Identity.

```
-- Database function:
-- Get the MMSI number as integer
CREATE OR REPLACE FUNCTION
    ais_mmsi(payload bit varying)
  RETURNS integer AS
$$
  BEGIN
    RETURN
      substring(payload from 9 for 30)::integer;
  END;
$$ LANGUAGE plpgsql
IMMUTABLE;
```

Figure 6. Example of function within PostgreSQL database to decode MMSI number from bit vector data type.

In earlier work, to store AIS messages inside PostgreSQL, we decided to use the bit vector data type (Meijers et al., 2016). The bit vector type allows direct access to arbitrary long subsets of bits (which is useful to decode the parts of the AIS message). We defined a set of database functions to access the parts of the AIS messages. The online document `http://catb.org/gpsd/AIVDM.html` was of great help interpreting the raw AIS data (Raymond, 2016). We made the following decoding functions in our database: MMSI number, Message type, Easting and northing of a geographic location (and a function to map easting and northing to the point type, provided by PostGIS), 6-bit ASCII encoded strings, Call sign, Name of a vessel and Destination. Figure 6 shows an example function for decoding the MMSI number from the stored bit vector.

APPENDIX B – SFC IN POSTGRESQL DATABASE

Table definition:

```
=# \d isprs_sfc

     Table "isprs_sfc"
  Column   |           Type
-----------+--------------------------
 sfc_key   | bigint
 ts        | timestamp with time zone
 ais_point | geometry(Point,4326)
 ais_mmsi  | integer

Indexes:
    "i__isprs_sfc__sfc_key" B-tree
        (sfc_key) CLUSTER
```

To create this table:

```
create table isprs_sfc as select * from
(
-- transform points + timestamps to 3D cube
with transform_params as
(
select
    sfc_transform_scale(
        -180, 180,
        0, pow(2,21)::numeric) as scale_east,
    sfc_transform_scale(
        -90,  90,
          0, pow(2,21)::numeric) as scale_north,
    sfc_transform_scale(
        extract(epoch from
            '2016-09-30 23:59:59+02'::timestamp
            with time zone)::int,
        extract(epoch from
            '2016-12-31 23:59:59+02'::timestamp
            with time zone)::int,
        0,
        pow(2,21)::numeric) as scale_time,
    180::numeric as translate_east,
    90::numeric as translate_north,
    -extract(epoch from
        '2016-09-30 23:59:59+02'::timestamp
        with time zone)::numeric
            as translate_time
)

select
  -- compute SFC-key
  sfc_hencode(
  array[
    -- X-dimension (space)
    sfc_transform_dim(e,
      (select translate_east
        from transform_params),
      (select scale_east
        from transform_params))::int,
    -- Y-dimension (space)
    sfc_transform_dim(n,
      (select translate_north
        from transform_params),
      (select scale_north
        from transform_params))::int,
    -- T-dimension (time)
    sfc_transform_dim(t,
      (select translate_time
        from transform_params),
      (select scale_time
        from transform_params))::int
  ]) as sfc_key,
  ts,
  ais_point, ais_mmsi
from (
    select
        extract(epoch from ts)::numeric as t,
        st_x(ais_point)::numeric as e,
        st_y(ais_point)::numeric as n,
        ts, ais_point, ais_mmsi
    from ais_staging where
        st_x(ais_point) between 0 and 180
        and
        st_y(ais_point) between 0 and 90
) as staging
) as data
```

Index, cluster, vacuum:

```
create index i__isprs_sfc__sfc_key on
        isprs_sfc (sfc_key);

cluster isprs_sfc using
    i__isprs_sfc__sfc_key;

vacuum analyze isprs_sfc;
```

Making table for ranges:

```
create temp table sfc_ranges as
select * from (

with transform_params as
(
select
    ... -- same as at load
)

select r.* from sfc_hquery(
    -- use offset and scale from transform_params
    -- same as in load step
    array[
        floor(sfc_transform_dim(9.8, ..., ...))::int,
        floor(sfc_transform_dim(56.0, ..., ...))::int,
        floor(sfc_transform_dim(
            extract(epoch from
                '2016-10-15 00:00:00+02'
                ::timestamp
                with time zone)::int,
                ..., ...))::int
    ],
    array[
        ceil(sfc_transform_dim(12.3, ..., ...))::int,
        ceil(sfc_transform_dim(58.5, ..., ...))::int,
        ceil(sfc_transform_dim(
            extract(epoch from
                '2016-10-15 23:59:59+02'
                ::timestamp
                with time zone)::int,
                ..., ...))::int
    ],
    11 -- how deep to descend 2^n-tree
) as r)
ranges;
```

Join ranges to data:

```
create unlogged table sfc_query_result as select
 d.*
from
 sfc_ranges r, isprs_sfc d
where
 d.sfc_key >= r.lower and d.sfc_key < r.upper;
```