Efficient classification of billions of points into complex geographic regions using hierarchical triangular mesh

Dániel Kondor\textsuperscript{1}, László Dobos\textsuperscript{1}, István Csabai\textsuperscript{1}, András Bodor\textsuperscript{1}, Gábor Vattay\textsuperscript{1}, Tamás Budavári\textsuperscript{2}, Alexander S. Szalay\textsuperscript{2}

\textsuperscript{1} Eötvös Loránd University, Department of Physics of Complex Systems
Pf. 32, H-1518 Budapest, Hungary
\textsuperscript{2} The Johns Hopkins University, Department of Physics & Astronomy
3800 San Martin Drive, Baltimore, MD 21218, USA

Outline

1 Motivation and problem statement
2 Our proposed solution
3 Implementation
4 Performance evaluation
5 Conclusion
Our use-case

- 1.1 billion geo-tagged Twitter messages (tweets) stored in a MS SQL Server database

For some of the analyses, we need to assign the tweets into geographic regions.

Calculate aggregated statistics by administrative areas, e.g., tweet activity, time series, etc.

Analysis by regions, e.g., differences in content, regional variations of language use.


D. Kondor et. al. (ELTE TTK, JHU) Point classification using HTM SSDBM ’14
Motivation and problem statement

Our use-case

- 1.1 billion geo-tagged Twitter messages (tweets) stored in a MS SQL Server database

- For some of the analyses, we need to assign the tweets into geographic regions

- Calculate aggregated statistics by administrative areas, e.g. tweet activity, time series, etc.

- Analysis by regions, e.g. differences in content, regional variations of language use

---


General motivation and overview

- Classify large amount of points into complex geographic regions
- More generally: carry out spatial joins on massive datasets
- Do this inside an RDBMS system (note: most RDBMS systems already offer GIS capabilities)
- Better understand and improve spatial indexing possibilities inside a database
- In our case, current solutions seemed inefficient
- With a custom index, we achieved a significant speedup
Overview of our solution

- We use the Hierarchical Triangular Mesh (HTM)\(^3\) for indexing both the geographic regions and the points.
- We use the Spherical Library\(^4\) for generating a convex cover of the region and creating a basic, low-resolution index on it.
- We then iteratively refine the index until a given precision.
- We implemented this refinement procedure as a loadable module for SQL Server.

---


\(^4\)http://voservices.net/spherical
Our proposed solution

Overview of our solution

- The index generation is first run as an SQL statement (call to an SQL Server CLR UDF), which fills a table with the index.
- Most of the points are then classified using the index (with a highly efficient range join query).
- Points on regions boundaries still need to be checked with the SQL Server GIS functions. This step can also be sped up using our index.
- The goal is to minimize the number of points which need to be checked by GIS functions. A good tradeoff between index size and number of partially classified points need to be chosen depending on the use-case.
Our proposed solution

The HTM index

- Recursive subdivision of the unit sphere into triangle-shaped cells (trixels)$^5$
- Multiple levels; resolution (number of cells) on level $L$ is $8 \times 4^L$
- For normal use-cases $L \leq 20$; on $L = 20$, $\sim 10$ m precision (average cell area: $\sim 60 \text{m}^2$)
- Each cell is assigned a unique ID (64-bit integer)
- Converting between a cell ID and coordinates is fast
- Open source library (written in C#, also includes loadable functions for SQL Server):
  
  http://voservices.net/spherical

---

Our proposed solution

The HTM index

- Aggregation and division is very easy (it can be achieved by dividing or multiplying the IDs by 4)
- A cell on lower resolution can be represented by a contiguous range of cells from higher resolution levels
- We represent the points by level 20 IDs (i.e. using the level 20 cell which contains the point)
- We represent the regions with a list of covering cells (of different levels); these are stored as ranges of level 20 IDs
- Determining if a point is contained in such a covering cell is achieved by checking if its ID falls in this range
Creating and using the index

Example: determine which points are inside California.
Our proposed solution

Creating and using the index

We start from a covering of a convex hull of California.
Creating and using the index

For each cell we determine if it’s inside (full cell), intersecting (partial cell) or outside.
Creating and using the index

We recursively subdivide partial cells until the desired resolution is reached.
Creating and using the index

Determining whether a point is inside a cell can be done very effectively.
Creating and using the index

Pre-filtering: we identify points in full and partial cells.
Our proposed solution

Creating and using the index

For points in full cells, we are done; for points in partial cells, we need to check if it’s really inside.
Creating and using the index

We use traditional GIS functions to test points in partial cells.
Our proposed solution

Creating and using the index

Trick to speed up processing: we already know which cell the points are in, so it’s sufficient to test if they are inside the intersection of the cell and the original region.
Creating and using the index

If we first compute the intersection of a partial cell with the whole region, and then use this for further testing, we can gain a further significant speedup.
Our proposed solution

Index size vs. resolution

HTM level 14: 31,251 cells to cover California, 100M for all countries in the world

HTM level 16: 139,447 cells to cover California
Implementation details

- We use the open source Spherical Toolkit\(^6\) for manipulating HTM cells.
- We implemented the HTM index generation in C#, as a loadable module for Microsoft SQL Server; it can be run as a table-valued UDF.
- We use the SQL Server GIS library for the calculating intersections.
- Points in full cells are classified by a highly efficient range join query (no GIS function calls here, only regular DB tables).
- Points in partial cells are classified by containment tests using the SQL Server GIS library; function calls are integrated into SQL queries as join predicates.

\(^6\)http://voservices.net/spherical
function EvalTrixels(region, trixellist, maxlevel)
    retlist ← ∅
    for all t in trixellist do
        if region.STContains(t) then
            retlist.Add(t, 1)
        else
            region2 = region.STIntersection(t)
            if region2 ≠ ∅ then
                if t.Level ≥ maxlevel then
                    retlist.Add(t, 0)
                else
                    tlist2 = t.Extend(t.Level+1)
                    retlist.AddRange(EvalTrixels(region2, tlist2, maxlevel))
            ▶ Flag as partial trixel
        ▶ Flag as full trixel
    ▶ Partial or disjunct trixel
    return retlist
Example queries

Creating the index:

CREATE TABLE regionindex(region_id int not null,
    lo bigint not null, hi bigint not null, full bit not null);
INSERT INTO regionindex SELECT region_id, lo, hi, full
    FROM regions CROSS APPLY HTMIndexCreate(geom, 14, 0);

Classifying points in full cells:

SELECT region_id, pt_id FROM regionindex r JOIN
    points p ON p.htmid BETWEEN r.lo AND r.hi AND r.full = 1;

Classifying points in partial cells:

SELECT region_id, pt_id FROM regionindex r JOIN
    points p ON p.htmid BETWEEN r.lo AND r.hi AND r.full = 0
    JOIN regions g ON r.region_id = g.region_id
    WHERE g.geom.STContains(p.point) = 1;
Performance evaluation

- We ran tests for the USA (the regions are the states), for various sample sizes.
- We compared our solution to the native GIS functions in SQL Server (which use a built-in indexing scheme).
- Significant speedup (up to 100 times); running the SQL Server version for all the 1.1 billion tweets did not seem feasible (projected runtime: 10 days on our system).
- After the performance tests, we ran our method for all countries in the world and all of the tweets; total runtime was only a few hours.
Performance evaluation

Index generation time and index size:

<table>
<thead>
<tr>
<th>index type</th>
<th>time [s]</th>
<th>index rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Server geography $8 \times 8$</td>
<td>13,352</td>
<td>412,055</td>
</tr>
<tr>
<td>SQL Server geography $16 \times 16$</td>
<td>6,215</td>
<td>410,040</td>
</tr>
<tr>
<td>HTM level 12</td>
<td>4,366</td>
<td>267,763</td>
</tr>
<tr>
<td>HTM level 14</td>
<td>5,151</td>
<td>1,331,632</td>
</tr>
<tr>
<td>HTM level 16</td>
<td>9,952</td>
<td>6,354,932</td>
</tr>
</tbody>
</table>
False positives

False positive rate of index-only queries for some states. Note, that false positive rates depend on the actual distribution of points and not only on the geometry of the states.

<table>
<thead>
<tr>
<th>index type</th>
<th>Colorado</th>
<th>Illinois</th>
<th>Maryland</th>
<th>Washington</th>
</tr>
</thead>
<tbody>
<tr>
<td>geography $8 \times 8$</td>
<td>&lt;0.01%</td>
<td>0.16%</td>
<td>3.62%</td>
<td>1.11%</td>
</tr>
<tr>
<td>geography $16 \times 16$</td>
<td>&lt;0.01%</td>
<td>4.66%</td>
<td>22.43%</td>
<td>3.14%</td>
</tr>
<tr>
<td>HTM level 12</td>
<td>0.01%</td>
<td>1.71%</td>
<td>4.82%</td>
<td>1.30%</td>
</tr>
<tr>
<td>HTM level 14</td>
<td>&lt;0.01%</td>
<td>0.18%</td>
<td>1.84%</td>
<td>0.47%</td>
</tr>
<tr>
<td>HTM level 16</td>
<td>&lt;0.01%</td>
<td>0.04%</td>
<td>0.53%</td>
<td>0.23%</td>
</tr>
</tbody>
</table>
## Performance evaluation

### Runtimes

#### Pre-filtering times:

<table>
<thead>
<tr>
<th>index type</th>
<th>300k [s]</th>
<th>1M [s]</th>
<th>5M [s]</th>
<th>1G [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>geography $8 \times 8$</td>
<td>223</td>
<td>780</td>
<td>5009</td>
<td>-</td>
</tr>
<tr>
<td>geography $16 \times 16$</td>
<td>223</td>
<td>883</td>
<td>4053</td>
<td>-</td>
</tr>
<tr>
<td>HTM level 12</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>194</td>
</tr>
<tr>
<td>HTM level 14</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>266</td>
</tr>
<tr>
<td>HTM level 16</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>232</td>
</tr>
</tbody>
</table>

#### Total runtimes:

<table>
<thead>
<tr>
<th>index type</th>
<th>300k [s]</th>
<th>1M [s]</th>
<th>5M [s]</th>
<th>1G [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>geography $8 \times 8$</td>
<td>295</td>
<td>915</td>
<td>4276</td>
<td>-</td>
</tr>
<tr>
<td>geography $16 \times 16$</td>
<td>301</td>
<td>773</td>
<td>4273</td>
<td>-</td>
</tr>
<tr>
<td>HTM level 12</td>
<td>12</td>
<td>24</td>
<td>139</td>
<td>2370</td>
</tr>
<tr>
<td>HTM level 14</td>
<td>7</td>
<td>13</td>
<td>58</td>
<td>1299</td>
</tr>
<tr>
<td>HTM level 16</td>
<td>8</td>
<td>10</td>
<td>42</td>
<td>1032</td>
</tr>
</tbody>
</table>
Rapidly increasing volume of spatial data, and use-cases for storing it in a RDBMS

Most systems nowadays offer a solution via GIS extensions

These are usually general-purpose solutions which can be inefficient for specialized tasks on massive datasets

For these problems, a specialized solution can achieve significant speedup
Outlook

- Generalize to region-region intersection and containment tests
- Optimize the index generation (e.g. parallelize the main procedure)
- Run more benchmarks with different configurations to gain a better understanding of how various factors and use-case requirements affect index size and performance
- Figure out good heuristics for required index depth; implement self-tuning index creation and easy dynamic refinement
- Further tune the queries, implement them as UDFs / stored procedures, include some decision logic for calculating intersections
- Port to other database systems and GIS libraries
Thank you!

Thank You!

http://www.vo.elte.hu/htmpaper

kdani88@elte.hu

dobos@complex.elte.hu