Efficient Query Processing in Geographic Web Search Engines

Yen-Yu Chen (yenyu@photon.poly.edu)
Torsten Suel (suel@poly.edu)
Alex Markowetz (alexmar@cs.ust.hk)
Polytechnic University
Brooklyn, NY
Web Search Engines

- used by millions of people every day
- a few large and many small engines
- cover billions of pages
- are completely changing the way people find information
Search Engine Architecture

- Crawling pages
- Parsing for link information
- Building Index
- Computing ranking scores
- Processing Queries
Standard Search Does Not Work Well

Example: *looking for a Yoga school around Brooklyn*

- try various terms with geographic meaning:
  - Yoga *Brooklyn*
  - Yoga “New York”
  - Yoga “Park Slope”
  - Yoga *Queens*

- inconvenient & inefficient
- will still miss many results
Geographic Search Engines

- Geographic data mining
  - Geo extraction, matching, propagation
- Geographic footprints
- Geographic ranking function

![Diagram of Geographic Search Engines](image)
Geographic Footprints

• how to represent geo. locations of documents?

• “document footprint”
  represented by some kind of spatial structure (bitmaps, polygons)

• how to use footprints to answer queries?

• look for documents that
  - contain the textual search terms
  - and whose geographic footprint overlaps the query footprint
  - score according to textual occurrences and degree of overlap

• we claim: this gives general and flexible approach

• how to extend a standard SE to answer such queries?
Geographic Footprints

- Raster data model
- Representing geographic footprint of a page as a bitmap on an underlying 1024x1024 grid of Germany
- Each point on the grid has an integer amplitude
- Bitmaps are kept as quad tree structures
Geographic Footprints of Web Pages

Two advantages:

1. Aggregation and other operations are efficient
2. Highly compressed
   - less than 100 bytes on average after simplification

0-badewanne.baby--shop.de
<table>
<thead>
<tr>
<th>Text Search</th>
<th>Geographic Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>User enters key words</td>
<td>User enters key words and interested geographic positions</td>
</tr>
<tr>
<td>Boolean operations on inverted index.</td>
<td>Boolean operations on inverted index and Footprints</td>
</tr>
<tr>
<td>Ranking according to subject-relevance</td>
<td>Ranking according to subject-relevance and intersection between footprints</td>
</tr>
</tbody>
</table>
Geographic Query Processing

- each doc consists of textual terms and a geographic footprint
- each query consists of textual terms and a query footprint
- standard engine: combination of textual (cosine) + links etc
- geo engine: combine with geographic score function $gs(d, q)$
  $$s(d, q) = ts(d, q) + pr(d) + gs(d, q)$$
- properties of geographic score
  - only depends on footprints
  - is zero if intersection empty
  - is “partitionable”
- examples:
  - intersection volume
  - better: dot product
  - with suitable weights
- claim: fairly flexible
Challenges

• How can spatial index be integrated into a search engine query processor?
  – Optimizing spatial index towards inverted index structure.

• How should the data flow during query execution?
  – Most selective filter, textual or spatial?
  – Interlacing within DAAT text query processor
## Standard vs. Geographic SEs

<table>
<thead>
<tr>
<th></th>
<th>Standard Search</th>
<th>Geographic Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data acquisition</strong></td>
<td>Standard crawling</td>
<td>Maybe restricted to particular area</td>
</tr>
<tr>
<td><strong>Data mining</strong></td>
<td>Link analysis, spam and duplicate detection, etc</td>
<td>Additional data mining for geo extraction</td>
</tr>
<tr>
<td><strong>Index construction</strong></td>
<td>Build inverted index</td>
<td>Build additional spatial data structures</td>
</tr>
<tr>
<td><strong>Query execution</strong></td>
<td>Traverse inverted lists</td>
<td>Traverse inverted lists plus spatial structures</td>
</tr>
</tbody>
</table>
• if you can cook it into a footprint, we have an efficient way to execute it
• separation of concerns
Experiment Setup

- seven machines
  - 512 MB memory each
- partition 31,000,000 documents from .de domain
  - 70GB of web pages (uncompressed) per machine
  - 6.5 GB compressed inverted index per machine
- 50% to 90% of pages have footprints of average size 130-500 bytes
- Inverted index and footprints need to reside on disk
  - main memory is too small
  - mostly used for caching
- footprints smaller than index but larger than individual inverted lists
Goals and Roadmap

- Minimizing footprint data to be retrieved
- Replacing random I/O by sequential scans
- Minimizing memory usage
- Early results

- Text First baseline algorithm
- Geo First baseline algorithm
- Toeprints
- Spatial disk layout
- K-sweep algorithm
- Tile-index algorithm
- Spatially ordered inverted index
Naïve Algorithms

- Text-First Baseline
  1. Go through inverted list with all keywords.
  2. Retrieve all footprints.
  3. Compute scores.

- Lots of random I/Os (footprints)
- DAAT (both steps can be interleaved per document)
Naïve Algorithms (2)

• Geo-First Baseline
  1. Get doc IDs from R*-tree.
  2. Filter doc IDs with inverted index.
  3. Retrieve all footprints.
  4. Compute scores.

• A smarter (non-dynamic) spatial index could have been used

• Blocking (sorting by doc ID after spatial index)
Toeprints

• MBRs of footprints can be quite large
  – Page on Hamburg and Munich spans all of Germany
• Degrade the spatial index
• Idea: toeprints
• Split up MBRs recursively
• Partitioning stops when
  1) No MBR has a side length > $S_0$
  2) No MBR larger than $S_1$ is more than X% empty
Data Layout on Disk

- We have to keep the toeprints on disk
- We have to retrieve some
- We want to avoid random I/Os and prefer sequential scans
- **Space-filling curves** are curves whose ranges contain the entire two dimensional unit square
- Arrange data on disk according to a space filling curve
K-Sweep Algorithm

- **IDEA:** Retrieve toeprints through a fixed number of contiguous scans
  1. Retrieve intervals from in-memory spatial structure.
  2. Perform up to k sweeps to fetch all toeprints.
  3. Filter doc IDs with inverted index.
  4. Compute the geo score.

<table>
<thead>
<tr>
<th>Tile ID</th>
<th>Interval one</th>
<th>Interval two</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3476</td>
<td>23400</td>
</tr>
<tr>
<td>1</td>
<td>4688</td>
<td>13202</td>
</tr>
<tr>
<td>2</td>
<td>13274</td>
<td>28408</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Disadvantage:
- fetches complete toeprint data without first filtering by query terms
- Blocking (sorting)
Tile Index Algorithm

- Uses a grid
- Stores doc IDs for every tile
  - Instead of intervals in k-sweep
  - Size reduction by coarser grid (256 * 256)

1. Retrieve doc IDs in tiles intersecting the query footprint.
2. Filter Doc IDs with inverted index.
3. Fetch the toeprints.
4. Compute and combine the geo score.

<table>
<thead>
<tr>
<th>Tile ID</th>
<th>Toeprint IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>354, 4562, 7821, ...</td>
</tr>
<tr>
<td>1</td>
<td>134, 592, 8211, ...</td>
</tr>
<tr>
<td>2</td>
<td>51, 62, 78, 101, ...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td>351, 362, 478, ...</td>
</tr>
</tbody>
</table>

Decreases amount of toeprints to be retrieved
But not dramatically
(they are clustered and k-sweep performs scans)
Space-Filling Inverted Index

- Idea: order documents in Inverted index along space filling curve

- Assign doc IDs along a space-filling curve

- Pros
  - Apply to either k-Sweep or Tile Index algo.
  - Fed into inverted index seamlessly
  - Speed up reading inverted index
  - Good trade-off if textual part is bottleneck

- Cons
  - Increase the size of inverted index
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time per Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Only</td>
<td>0.33</td>
</tr>
<tr>
<td>Text-First</td>
<td>11.18</td>
</tr>
<tr>
<td>Geo-First</td>
<td>6.47</td>
</tr>
<tr>
<td>1-Sweep</td>
<td>1.61</td>
</tr>
<tr>
<td>3-Sweep</td>
<td>0.67</td>
</tr>
<tr>
<td>4-Sweep</td>
<td>0.61</td>
</tr>
<tr>
<td>Tile Index (8x8)</td>
<td>0.47</td>
</tr>
<tr>
<td>Tile Index (16x16)</td>
<td>0.42</td>
</tr>
<tr>
<td>Space-Filling Inverted Index (basic)</td>
<td>0.35</td>
</tr>
<tr>
<td>Space-Filling Inverted Index (improved)</td>
<td>0.34</td>
</tr>
</tbody>
</table>

Table 4.1: Overview of execution times for various algorithms, on query footprints of size 10 × 10.
K-Sweep

![Bar charts showing average time per query for different split ratios and sweep counts.]

- **Left Chart**: 
  - 1:4.11 split, 1:5.48 split, 1:6.02 split
  - Legend: One Sweep, Two Sweeps, Three Sweeps

- **Right Chart**: 
  - 1:3.01 split, 1:3.48 split, 1:4.11 split
  - Legend: One Sweep, Two Sweeps, Three Sweeps
Tile Index vs. Space-Filling Inverted Index

![Graphs showing performance comparison between Tile Index and Space-Filling Inverted Index](image)

**Legend:**
- 4 x 4
- 8 x 8
- 16 x 16 Tile Index

**X-axis:** Size of Query Footprint
- 10x10
- 20x20
- 30x30

**Y-axis:** Average Time Per Query
Comparison

- **Average Time Per Query**
  - **10x10**: Tile Index
  - **20x20**: Space-Filling Inverted Index (basic)
  - **30x30**: Space-Filling Inverted Index (improved)

- **Size of Query**: 10x10, 20x20, 30x30
Caching

![Bar chart showing average time per query for different cache sizes and indexing methods. The x-axis represents cache sizes (None, 64M, 128M, 256M) and the y-axis represents average time per query. The chart compares different indexing methods: Tile Index (16x16), Space-Filling Inverted Index (basic), and Space-Filling Inverted Index (improved).]
Related Work

• M. van Kreveld, et al. “Distributed ranking methods for geographic information retrieval”, In European Workshop on Computational Geometry
Conclusion and Future Work

- Geographic query processing can be performed at about the same level of efficiency as text-only queries.
- Pruning techniques for geographic search engines.
- Parallel geographic query processing.