Collaborative Geospatial Feature Search

George Lamprianidis
Institute for the Management of Information Systems
Research Center “Athena”
Artemidos 6, 15125 Maroussi, Greece
glampr@imis.athena-innovation.gr

Dieter Pfoser
Institute for the Management of Information Systems
Research Center “Athena”
Artemidos 6, 15125 Maroussi, Greece
pfoser@imis.athena-innovation.gr

ABSTRACT

The ever-increasing stream of Web and mobile applications addressing geospatial data creation has been producing a large number of user-contributed geospatial datasets. This work proposes a means to query such data using a collaborative Web-based approach. We employ crowdsourcing to the fullest in that used-generated point-cloud data will be mined by the crowd not only by providing feature names, but also by contributing computing resources. We employ browser-based collaborative search for deriving the extents of geospatial objects (Points of Interest) from point-cloud data such as Flickr image locations and tags. The data is aggregated by means of a hierarchical grid in connection with an exploratory and a refinement search phase. A performance study establishes the effectiveness of our approach with respect to the amount of data that needs to be retrieved from the sources and the quality of the derived spatial features.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

General Terms

Algorithms

Keywords

data mining, geospatial data fusion, user-contributed content, map reduce

1. INTRODUCTION

Geospatial data is out there, maybe not always in the format, quality and amount we expect it, but by employing intelligent data collection and mining algorithms, we are able to produce high-quality, and what is sometimes more important, unusual data.

User-Generated Content (UGC) along with crowdsourcing are elegant concepts that when properly applied lead to staggering results. This work aims at exploiting already existing data and employ user-contributed computing resources and search parameters to produce quality geospatial datasets. The Web has created a number of services that facilitate the collection of location-relevant content, i.e., data for which location is just but one, and most often not the most important attribute. Prominent examples include (i) photo sharing sites such as Flickr and Panoramio, and many others, (ii) microblogging sites with a geotagging feature, e.g. Facebook, Google+, Twitter as well as related photo sharing sites (twitpic) and (iii) more recently check-in services, e.g., Foursquare, Loopt, and Gowalla. Our goal is to utilize this user-contributed geospatial data and to derive meaningful geospatial datasets from it.

It is interesting to see that User-Generated Geospatial Content, or, Volunteered Geographic Information (VGI) [10, 7] is *large in volume but comparatively poor in quality.* A basic lesson that can be learned from crowdsourcing is that the crowd always gets it right, i.e., the estimate of the crowd typically beats the estimate of a single person even that of an “expert” [16]. A similar conclusion can be drawn when considering surveying engineering and the specific task of determining an unknown location by observing the distance and orientation to several known locations. The accuracy of the coordinates produced by the mathematical process of adjustment computation increases with the number of observations, i.e., the less influence a single observation has on the overall outcome.

This work will employ the crowdsourcing concept to the fullest in that user-generated point-cloud data will be mined by the crowd not only by providing concepts (searched for geospatial features) but also by providing computing resources. We employ browser-based collaborative search for deriving the extents of geospatial objects, or, Points of Interest from point-cloud data such as Flickr image locations. In a nutshell, the user provides the search terms (e.g., “Plaka, Athens”) and respective point databases are queried based on their tag information and using their APIs to retrieve a point cloud that characterizes this location. This point cloud stands for the wisdom of the crowd and somehow this data needs to be aggregated to derive the actual location of the searched-for geospatial object, e.g., a polygon derived from the point cloud and representing the spatial extent of “Plaka” . Here, a particularity in our approach is the use of a browser as a computing platform. Borrowing from the MapReduce computing paradigm, we create our own browser-based version in which data collection and computation tasks are delegated to the connected search clients (browsers) and the server only coordinating the search. As a side effect, no actual data is retrieved by the server directly from any point-cloud data source, but all traffic is handled by the clients, effectively balancing the data traffic of the search over the network. The search is collaborative in that the more clients are connected, i.e., users searching for the same term, the faster the search finishes.

Overall, the objective of this work is to transform available user-contributed geocontent ("point cloud") into meaningful chunks of information (e.g., a polygon representing the location of a spatial object). We aim for geospatial datasets obtained with simplicity and speed comparable to that of Web-based search. To achieve this task, we propose our search method utilizing crowdsourcing concepts implemented as a Web-based, collaborative search tool termed *Jeocrowd*[1].

The outline of the remainder of this work is as follows. Section 2 discusses related work. Section 3 surveys the type of user-generated data we will exploit in this work. Section 4 describes the advocated computational approach, while the actual computing framework is given in Section 5. Section 6 presents an experimental evaluation, and finally, Section 7 gives conclusions and directions for future work.

2. RELATED WORK

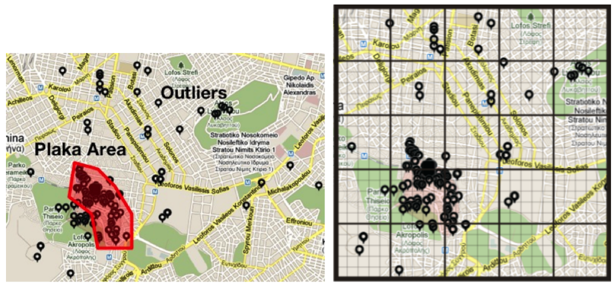
In recent years various approaches have been developed that aim at exploiting UGC and more specifically VGI for use in different applications. The following discussion should give an indication as to what has been achieved and makes no claim of completeness.

One big issue raised in [9], is the concern about the credibility of VGI. Several techniques and strategies are examined that help identify the quality and reliability of such user-generated content. In [4] there is an effort to classify the types of people, companies or agencies that contribute GI data and the nature of their contribution. [11] touches the interesting issue of the localness of the GI contributed data. By examining two major websites, Flickr and Wikipedia, the authors find that more than half of Flickr users contribute local information on average, while in Wikipedia the authors' participation is less local.

Because of the nature of the data and the increase in the availability of computing resources, many techniques that process this information in parallel have been proposed to reduce the execution time and provide results faster. The MapReduce model is a perfect candidate to be used as a programming model in these scenarios, as shown in [1] and [17] where it outperforms the traditional computational approaches. In our work, we implemented the MapReduce framework a bit differently. Instead of distributing our computational work load to server farms, or the cloud, we let the users' browsers perform all the calculations for us and then return us the results.

Finally, there is a great number of related work dealing with extracting tag information to derive meaningful geospatial data, or, in more general terms, place information. [5] outlines an approach to derive geospatial shape information from geotagged Flickr photos. Yahoo! GeoPlanet WOEIDs as part of the metadata are exploited by means of direct access to the Flickr database. [15] extracts place semantics

[1]The tool is named Jeocrowd, which is a combination of Geo + Crowd, using a J instead of G, to emphasize the use of Javascript as its core calculation platform.



(a) Point cloud (b) Grid approach

Figure 1: Point cloud for the "Plaka" area in Athens, Greece.

from Flickr tag information based on the spatial distribution of the data. In [12], again, Flickr data is used to identify places and also relationships among them (e.g., containment) based on tag analysis and spatial clustering. While the above approaches have certain similarities with the basic premise of our work, i.e., to extract geospatial information from user-contributed datasets, our contribution will be towards defining a collaborative data mining framework that actively involves the user and their resources in the process. Finally, an early prototype showcasing the Jeocrowd approach has been presented as a demo paper [13].

3. USER-GENERATED AND GEOSPATIAL DATA

Users generate data by means of many different applications in which the geospatial aspect does not play a major role, but is simply used to index and access the collected data. As mentioned, prominent examples here are photo sharing sites and micro blogging services using geotagging features. To illustrate the potential for geospatial data generation, consider the use of Flickr data for the computation of feature shapes of various spatial objects including city scale, countries and other colloquial areas [5]. The approach is based on computing primary shapes for point clouds that are grouped together by Yahoo! GeoPlanet WOEIDs (Yahoo! Where On Earth IDs), which are part of the Flickr metadata. Flickr currently contains 150+ million geotagged photos.

Querying the API with a specific search term returns in essence a "point cloud" referring to a specific POI. In other words, the geotagging of each photo represents an independent observation of the "true" location of the POI. The ambition of this work is to take point clouds as input and to identify an approach for extracting geospatial location information from it. Here we essentially use the wisdom of the crowd to determine the true location of a POI. Consider the example point cloud in Figure 1(a), which shows geotags of retrieved flick images for the query "Plaka". The result includes also locations outside the actual area (shaded polygon). The task at hand is how to derive the area information from this point cloud data. Complicating things is the fact that the location of a POI might be that of a point or area feature. Prominent examples here are "Central Park" (area feature) and "Parthenon" (point feature) but also larger features such as cities ("London") or entire

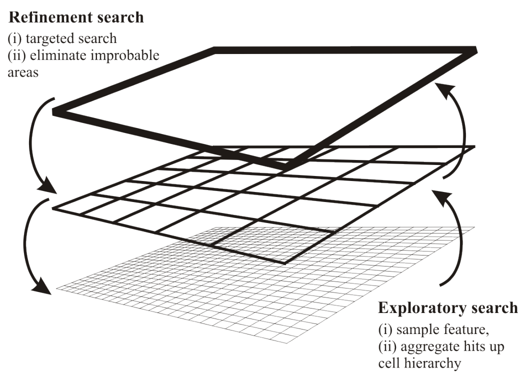


Figure 2: Exploratory and refinement search traversing up and down the hierarchical grid.

natural areas ("Alps") [2].

Overall, to extract place information from point clouds, we propose a method based on *online* (live search), *collaborative* (many users querying the same term speed up the search), *Web-based* (browser-based computing) *crowdsourcing* (user-contributed expertise - the search term, and resources - computing power and bandwidth). The core contributions of this work are (i) a collaborative spatial search implemented by means of (ii) Web and browser-based computing utilizing a browser-based MapReduce approach as described in the following two sections.

4. COLLABORATIVE SPATIAL SEARCH

The basic task at hand is how to derive spatial datasets from user-contributed point cloud data that have basically no quality guarantee. A typical approach would entail the *clustering of the points* (cf. [5]) in combination with outlier detection. The latter should eliminate points that have been assigned false coordinate information in connection with image labels.

In Jeocrowd, we employ a *grid-based clustering approach* (cf. [18], [14], [8]). Point cloud locations are aggregated by means of a hierarchical grid. The fanout n and the grid spacing at the lowest level a are grid parameters. For example, with $n = 5, a = 50m$, we generate a grid that at the lowest level (Level 0 or base level) consists of $50m \times 50m$ cells, at the level above (Level 1) of $250m \times 250m$ cells, etc. At every level, except the base one, a cell will enclose n^2 cells of the level below.

Performing a spatial search entails now intelligently populating the cells by retrieving photos and more specifically their locations using a point cloud API and a specific search term that is matched to tag data. We essentially count how many photos are located in a specific cell. Figure 1(b) shows the example of "Plaka, Athens". Using various thresholds such as occupancy in connection with neighbor count information, the result will comprise as set of connected cells whose spatial extent will be the position of the searched for spatial feature (POI). The typical result will be a polygon with the accuracy of the shape being limited by the spacing of the grid.

[2]The authors acknowledge however that whether a location can be represented by a point or an area depends on the scale and (practical) degree of abstraction at which the data is represented.

Using a grid has the advantage that the search (i) can be *parallelized*, i.e., cells can be populated simultaneously from various data sources and (ii) is in fact *continuous*, i.e., as additional user-generated data becomes available, the search result will be refined. For example, city boundaries may change in subsequent searches to reflect newly observed data.

Since dealing with remote data sources, our goal is to *retrieve as little data as possible* so as to speed up search and reduce traffic. To this effect we split our search into an *exploratory* part, which tries to quickly find a good estimate of the extent of the spatial feature and a *refinement* part, which then tries to explore the shape of the spatial feature in greater detail. As shown in Figure 2, the hierarchical grid is used (i) to expand the search area after the exploratory phase and (ii) to eliminate unlikely areas during the refinement stage.

4.1 Exploratory Search

Exploratory search retrieves some initial points matching the search terms to get an overview of the spatial extent of the spatial feature, i.e., is it a point location (statue, coffee shop) or are we looking for an area feature (neighborhood, city, country). These points are used to construct the base grid. Each point is assigned to a grid cell according to its coordinates. The total count of points (photos) within a cell is called *degree* of the cell. During this phase we aim at collecting a representative but small in size point cloud relevant to our search terms. Hence when using Flickr as our point cloud provider, the data is requested and retrieved sorted first by relevance and then by descending interestingness. These two sorting criteria are the parameters of the API that provide us the most relevant dataset for this provider. In the current implementation using only Flickr data, we collect approximately 4000 unique points (photos). Figure 7(a) shows the result of the exploratory search for "Plaka" (Level 0). Specifically a neighborhood density map is shown, in which colors from blue to red are assigned to cells based on how many neighboring cells have also collected point cloud data. Isolated cells are colored blue and surrounded cells are shaded red.

4.2 Refinement Search

During the refinement phase, *the search is forced into areas that are suspected of being part of the feature*, but for which we have not retrieved (many) points during the exploratory phase. Users tend to take disproportionally many pictures of the most popular locations. The refinement search uses the *search term in connection with location estimates* determined by the exploratory search to retrieve additional photos for this "unpopular areas".

4.2.1 Grid Construction

The refinement search is based on the *hierarchical grid* idea using a top-down approach. However, to make the top-down approach work, we first need to work our way bottom-up and construct the hierarchical grid based on the exploratory search results that populated the base level. The two procedures given in pseudo code in Figure 4 and Figure 3 are used to iteratively build each level of the hierarchical grid by aggregating degrees of lower-level cells to create higher-level ones using an aggregation criterion and a termination criterion.

```
BuildGridLevel($level, aggregationFunc$)
 1  $grid \leftarrow \emptyset$
 2  $gridBelow \leftarrow$ GetGridAtLevel($level$ - 1)
 3  for each $cell$ in $gridBelow$
 4      $parentId \leftarrow$ DigitizeCoordinates($cell.lat, cell.lon$)
 5      if $\neg$CellExists($grid, parentId$)
 6              $\wedge$Apply($aggregationFunc, cell$)
 7          $siblings \leftarrow$ getSiblings($cell$)
 8          $degree \leftarrow siblings$ .Sum($'degree'$)
 9          $points \leftarrow siblings$ .Collect($'points'$)
10          $parentCell \leftarrow$ MakeCell($parentId, degree, points$)
11          AddCell($grid, parentCell$)
12  return $grid$
```

Figure 3: Construction of a single level of the hierarchical grid

```
BuildGrid()
 1  $grids[level] \leftarrow \emptyset$
 2  $aggregationFunc \leftarrow$ AtLeastTwo $\triangleright$ $\theta = 2$
 3  for $level \leftarrow 1$ to 7
 4      $grids[level] \leftarrow$
                BuildGridLevel($level, aggregationFunc$)
 5      if CountCells($grids[level]$) == 0
 6          $terminationLevel \leftarrow level - 1$
 7          break
 8  $aggregationFunc' \leftarrow$ AtLeastOne $\triangleright$ $\theta = 1$
    $\triangleright$ $terminationLevel$ is the stopping factor needed
    $\triangleright$ using $aggregationFunc'$
 9  for $level \leftarrow 1$ to $terminationLevel$
10      $grids[level] \leftarrow$
                BuildGridLevel($level, aggregationFunc'$)
11  return $grids$
```

Figure 4: Construction of the hierarchical grid (iteratively build each level)

The *aggregation* criterion determines whether a higher-level cell will be populated based on the attributes of its constituting lower-level cells. An example criterion would be to create a higher-level cell based on a threshold θ for the aggregated degree of its constituting lower-level cells. The BuildGridLevel procedure (Figure 3) is called within an iteration, in order to create each of the multiple levels of the hierarchical grid, until a grid at a certain level contains zero cells. The procedure loops through all the cells of a level, grouping them together to form the larger cells of the level above. If the aggregation criterion is met, then the group is instantiated as a cell in the level above. If the aggregation criterion cannot guarantee that eventually a level will have zero cells, the iteration must use a termination criterion as well.

The *termination* criterion is introduced to terminate cell aggregation with a weak aggregation criterion. For example, if we use $\theta = 2$, i.e., propagate cells if they enclose at least 2 populated cells from the level below, the aggregation will eventually terminate. However, for $\theta = 1$, the aggregation will not terminate as there is always at least one populated cell that gets "aggregated" to an upper level. In this case a termination criterion is needed to stop aggregation. Our experiments showed that with $\theta = 2$ many interesting areas were "cut off" because they had not attracted sufficient points in the exploratory search. Hence, the choice of $\theta = 1$ and the introduction of a termination criterion.

Because the termination criterion needs to be variable rather than fixed for better results, we opted for a filter-refinement strategy in our search. An initial search with $\theta = 2$ derives the termination (maximum) level the hierarchical grid reaches in our search, i.e., the size estimate of the spatial object. This level becomes the termination criterion for the second aggregation phase that re-runs the algorithm with $\theta = 1$ and stopping the search at the level corresponding to the termination criterion. This flow is described in pseudo code in Figure 4.

When searching for *large-scale spatial objects*, the construction of the hierarchical grid might prematurely terminate, i.e., the estimated extent is too small due to sparse point clouds. In those cases, we want to allow the grid to grow beyond what is estimated. To accommodate for such scenarios, after computing the maximum level and grid, we employ the following heuristic. We measure the distance between the centers of the h "hottest cells". If the average distance is larger than s times the length of the edge of a cell for that maximum level then the grid is considered *sparse* and an additional level is added to the hierarchical grid. This procedure is described in pseudo code in Figure 5. The default values are $h = 5$ and $s = 10$.

```
DetectSparseGrid($h, s$)
    $\triangleright$ order cells by descending degree
 1  $cells \leftarrow grids[terminationLevel]$.GetCells()
 2  $cells \leftarrow cells$ .SortBy($'degree', DESC$)
    $\triangleright$ get the first $h$ cells
 3  $hot \leftarrow cells[0..h-1]$
    $\triangleright$ calculate average distances
 4  for each pair $c1, c2$ in $hot$
 5      $d[c1, c2] \leftarrow$ DistanceBetween($c1, c2$)
 6  $length =$ LengthOfCellEdge($terminationLevel$)
 7  if Average($d$) > $s \times length$
 8      $terminationLevel \leftarrow terminationLevel + 1$
 9      $grids[terminationLevel] \leftarrow$
                BuildGridLevel($terminationLevel$,
                        $aggregationFunc'$)
10  return $grids$
```

Figure 5: Expanding the grid one more level if the existing top level has a lot of disjoint components.

4.2.2 Filtering Cells

At this stage of the search a set of large cells gives an estimate of the shape of the spatial object. What needs to be done now is to "carve out" the actual shape, i.e., direct the search intelligently so as to eliminate as large as possible improbable areas as early as possible in our search. To dissect larger cells into smaller cells and eliminating cells in the process the two criteria *degree* and *neighborhood density* (i.e., count of adjacent cells) of each cell are used. The filtering algorithm needs a function employing those two parameters that when applied on a cell decides if it should be discarded or not. If $\overline{\delta(h)}$ is the average degree of the first

h "hottest cells", i.e., cells with the largest degree, then the function used in the current implementation discards a cell if (i) it is isolated (has 0 neighbors) and its degree is less than $0.5 \times \overline{\delta(h)}$, or, (ii) if it is not isolated and its degree is less than $k \times \overline{\delta(h)}$. The default values are $k = 0.02$ and $h = 5$. The filtering procedure is described in pseudo code in Figure 6.

```
FilterLevel($level, k$)
  $grid \leftarrow grids[level]$
  $cells \leftarrow grid$ .GetCells()
  $\triangleright$ order cells by descending degree
  $cells \leftarrow cells$ .SortBy($'degree', DESC$)
  $\triangleright$ get the first $h$ cells
  $hot \leftarrow cells[0..h-1]$
  $avg \leftarrow$ Average($hot$)
  $\triangleright$ iterate all cells and discard as necessary
  for each $cell$ in $cells$
      $neighborCount \leftarrow cell$ .GetNeighbors().Count()
      if ($neighborCount == 0 \wedge cell.degree < 0.5 \times avg$)
            $\vee(neighborCount > 0 \wedge cell.degree < k \times avg)$
            $grid$ .DeleteCell($cell$)
  return $grid$
```

Figure 6: The filtering procedure that runs at each level of the grid before progressing to the next one.

The refinement search then proceeds with searching these *super cells* that remained after the filtering process, starting from the grid at the maximum level and working its way top-down. Figure 7(b) shows a super cell for the search term "Plaka". Because in this phase of the search each search request to the external provider (e.g., Flickr) is spatially bounded by the border of a cell we only need to request the total count of points in that cell, thus reducing the size of the data requested and transmitted considerably. As we will see in Section 6, on average a request at this phase will download only 22bytes, as opposed to the exploratory phase of the search where the payload is approximately 89KB per request.

Initially at this stage the *top level* cells are dissected, so that a "hollow" grid at level *top level - 1* is created. The "hollow" grid cells are then sequentially searched, and their degree gets updated. Once all the cells of a specific level have been searched, the algorithm runs the filtering procedure again to discard the most irrelevant, then recursively dissects the remaining ones and proceeds to the level below.

In addition to that, the filtering procedure also tries to detect cells that are completely surrounded by populated neighboring cells. These *core cells* will not be dissected at lower levels as they represent significant areas of the spatial object discovered early on. Discarded cells will not be examined further at a lower level. Table 4 shows that this elimination process is very beneficial for the algorithm, as it discards about 85% of cells on average. The boost in execution time as well as in total data downloaded is significant. Using a hierarchical grid, the goal is to rule out cells as early as possible, i.e., at high levels, since examining larger areas at lower levels is costly (many small cells). The result of the "Plaka" search after completing the refinement step is shown in Figure 7(c).

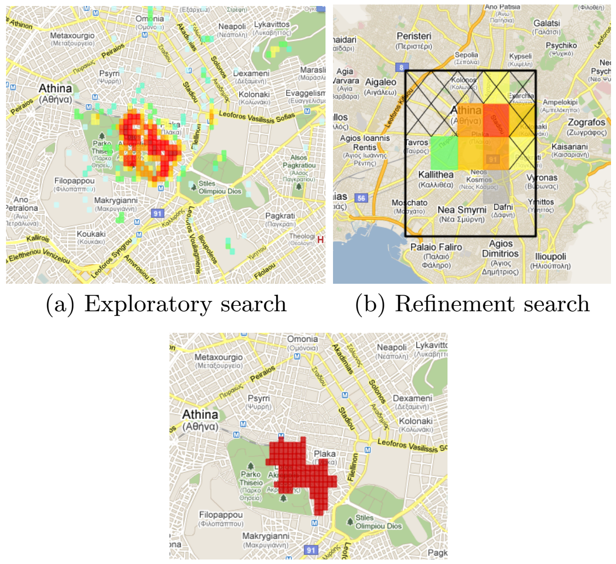

(a) Exploratory search
(b) Refinement search
(c) Result

Figure 7: Searching for Plaka, Athens

Our approach can also detect multi locations if the search term refers to more than one spatial entity. For example, the search for "Olympic Stadium, Athens" will return two locations, the Marble Stadium and the new Olympic Stadium.

5. BROWSER-BASED MAPREDUCE

To facilitate collaborative search, one not only needs to design a search strategy and respective data structures, but also an efficient computation framework. Search is a typical task performed by Web users. We build the computation framework around this notion and try to not only provide a service for users, but to involve them (and their resources) in the search process as much as possible. Hence, the core aspects of the approach described in the following are (i) parallelization, (ii) Web computing and to some extent (iii) user-contributed content.

We propose browser-based computing in connection with the MapReduce [6] concept. We introduce a Web application to speed up the procedure by utilizing the users' computing resources and, thus, offloading the server. The difference to the original MapReduce idea is that due to the browser-based nature of the application, the task assignment uses a *pull* rather than a *push* mechanism. In conventional implementations a master worker pushes the several map tasks to a list of available machines. Here each client (web browser) requests a new task once it connects to the service. The server is only used for coordinating tasks and storing results. All computation takes place in the browser. This has many advantages, the main one being the increase in computational capacity. The basic work flow of the exploratory and the refinement search applying our browser-based MapReduce paradigm is shown in Figures 8 and 10, respectively.

5.1 Parallelizing Search

MapReduce can only be applied provided one finds a way to parallelize problem solving. Our collaborative spatial search consists of two distinct search phases, each affording a different approach to parallelization.

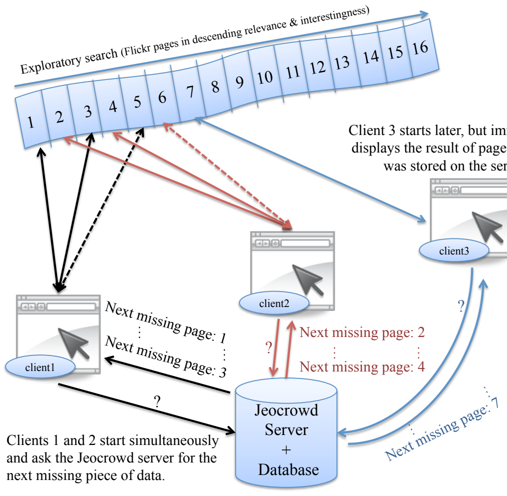

Exploratory search (Flickr pages in descending relevance & interestingness)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Client 3 starts later, but immediately displays the result of pages 1-6 that was stored on the server.
client3
client2
client1
Next missing page: 1
Next missing page: 3
Next missing page: 2
Next missing page: 4
?
Jeocrowd Server + Database
Clients 1 and 2 start simultaneously and ask the Jeocrowd server for the next missing piece of data.
Next missing page: 7

Figure 8: Browser-based MapReduce in Jeocrowd during the exploratory phase and of the search.

```
ExploratorySearch($p, results$)
      ▷ $p \in [0, \text{MAX\_PAGE}]$
1     $scheduling[page] \leftarrow p$
2     $results[page] \leftarrow results$
      ▷ assign new time-stamp for client and
      ▷ insert it into scheduling array
3     $t \leftarrow$ CurrentTimeInMilliseconds()
4     if $\neg$ scheduling.Full?
5         scheduling.Push($t$)
6     else
7         for $page \leftarrow 0$ to $\text{MAX\_PAGE} - 1$
8             if $(t - scheduling[page] > \text{TIMEOUT}) \wedge$
9                 $(scheduling[page] \notin [0, \text{MAX\_PAGE}])$
10                $scheduling[page] \leftarrow t$
11    return scheduling
```

Figure 9: Server-side flow when a client tries to store a page of results during the exploratory phase.

During each search phase (cf. Section 4) data is requested from an external provider. For the *exploratory search* the data is requested as a set of pages sorted by relevance and descending interestingness. Each page contains a fixed number of point locations (we do not need to retrieve actual images). Each page is assigned to a client, i.e., a Web browser connected to the service, which queries the external provider for any points matching the search term and then submits the results to the server.

During the *refinement phase* the cells are searched/processed in order of their coordinates. To parallelize the search, each client is assigned a block/set of cells it then retrieves from the external data provider. As stated, in contrast to the exploratory search, during the refinement phase, we only retrieve counts of points within the boundaries of each cell. The number is reported back to the server.

The ordering in both cases permits the *search task to be split into multiple subtasks*, each of which can be assigned a non-overlapping range of pages or blocks to be retrieved from the external point-cloud data sources, e.g., Flickr API. Each browser handles its portion of the search separately only coordinated by the server, thus parallelizing the search. (cf. Figure 8 and Figure 10).

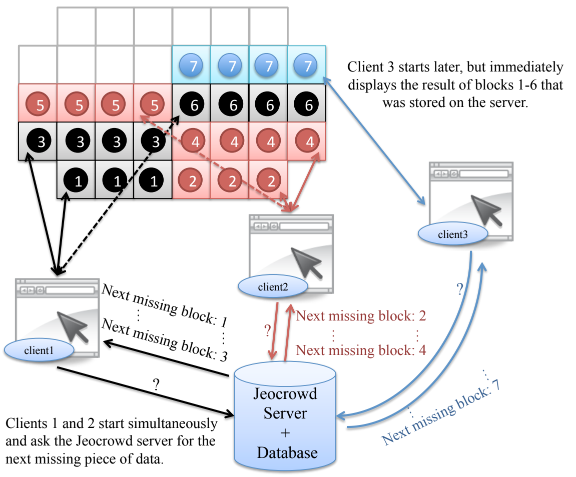

7 7 7 7
5 5 5 5 6 6 6 6
3 3 3 3 4 4 4 4
1 1 1 2 2 2
Client 3 starts later, but immediately displays the result of blocks 1-6 that was stored on the server.
client3
client2
client1
Next missing block: 1
Next missing block: 3
Next missing block: 2
Next missing block: 4
?
Jeocrowd Server + Database
Clients 1 and 2 start simultaneously and ask the Jeocrowd server for the next missing piece of data.
Next missing block: 7

Figure 10: Browser-based MapReduce in Jeocrowd during the refinement phase of the search for a single level of the hierarchical grid.

```
RefinementSearch($level, newResults, blockSize$)
      ▷ newResults is a hash, eg. key - value pairs
      ▷ newResults are appended to the results array
1     $results[level]$.Push($newResults$)
      ▷ the ids of the cells just save are removed
      ▷ from the markingArray
2     $markingArray[level]$.PullAll($newResults.keys$)
      ▷ the first blockSize cell ids are removed from the
      ▷ assingingArray and copied for transfer to the client
3     $newBlock \leftarrow assingingArray[level]$.Splice($0, blockSize$)
      ▷ if the new assignment is empty copy all cell ids not
      ▷ marked as saved yet and try assigning again
4     if $newBlock$.Empty?() $\wedge \neg$ $markingArray[level]$.Empty?()
5         $assingingArray[level]$.PushAll($markingArray[level]$)
6         $newBlock \leftarrow assingingArray[level]$.Splice($0, blockSize$)
7     return newBlock
```

Figure 11: Server-side flow when a client tries to store a block of results during the refinement phase.

The mechanism of task assignment is handled by the server. A no-SQL document database [2] that supports atomic in-place updates, e.g., MongoDB [3], alongside with a write-only technique to eliminate race condition effects from the multiple clients as much as possible, is used.

This approach aims at providing a scalable architecture with the ability to *resume tasks* from their last saved state by losing the minimum amount of work possible. To persist the state of a search, we keep all the information needed to resume it, on the server. This includes which pages and blocks have already been retrieved and which are to be retrieved next. Once a new client connects to the service, regardless if it's the only one searching for a specific term, or if there are others as well, receives the latest snapshot of the search and is assigned the next available task. Depending on the phase of the search a different approach is used.

5.1.1 Exploratory Search

The *exploratory search* downloads point ids with coordinate information. Here we need to keep all this information saved to avoid using duplicate points when calculating the counters to build the hierarchical grid. Here, the task assignment works as follows. There are *two arrays* with 16 slots each. The *scheduling array* is used for coordinating the clients and the *results array* is used for storing the actual results. Each array matches its index to the page number of the results fetched. To distinguish which client is responsible for which page we use millisecond timestamps. When a request arrives from a client, the server appends a new timestamp to the scheduling array and sends back the array and the timestamp to the client. The client reads the timestamp, finds its index in the array and requests the page with the same index from the external provider. Once the result has been retrieved, the client sends it back to server where it gets saved in the results array. At this point the timestamp that rests at the specified index in the scheduling array is replaced with the index value itself, thus marking that specific page as complete. In the exploratory phase only the base grid (at level 0) is stored in the database. Every other level can be computed and displayed on demand, e.g., for visualization purposes.

5.1.2 Refinement Search

The *refinement search* uses point counts for specified bounding boxes rather than actual coordinate information. For each level of the hierarchical grid, the search terminates after all cells have been searched. However the number of cells in each grid is not a fixed number. The only hard limit posed by the implementation is the total number of levels in the hierarchical grid, which is 7. The task assignment is accomplished as follows. Three arrays exist for *each* level. The first two are used to coordinate the client jobs; the *assignment array* is used to hold all the cell ids that are to be search, and the *missing array* contains all the cell ids for which results have not been saved yet. The last one, the *result array*, is used to store the actual results as {cell id, count} tuples. This structure helps in reducing (i) update racing conditions and (ii) the overlap in the data stored by the different clients. In terms of implementation detail, this is achieved by using the atomic, in-place modifiers that MongoDB offers when altering the arrays. Every time we want to assign a new block of cells to a client for processing, we first read all the cell identifiers, then explicitly pull designated identifiers from the assignment array and send these identifiers to the client. The client then queries the external provider for the total count of photos contained in each cell's bounding box. Once its assigned block of cells has been searched, the client submits the results back to the server. The cells are appended to the result array, and the cells' ids are pulled from the missing array. When both the assigning and the missing array are empty, the search of this specific cell level is considered complete and the filtering and dissection process proceed to the next lower level.

6. EXPERIMENTATION

The objective of the following experimentation is to establish the performance of the collaborative spatial search algorithm in terms of (i) time it takes to compute a result and (ii) the data exchanged with the external provider(s). The performance will be evaluated for a varying number of search terms represented ranging from point features such as "Eiffel tower" to large-scale features such as "Alps". Important for the performance assessment will be a parallelization and, thus, utilization of the map-reduce approach by using a varying number of browsers that concurrently perform a search.

6.1 Setup

The experiments conducted will include four sets of six search terms each, grouped by the anticipated spatial extent of the results: point features, small-scale areas, city-scale areas feature and large-scale area features. Each set will have its own profile (configuration settings), optimized to give good results for the selected spatial extent. For each one of these sets, we will then run the search 5 times, increasing the number of clients used by a factor of 2.

All search parameters, summarized in Tables 1 and 2, were established through experimental evaluation or by making basic assumptions. In addition, the optimal refinement search parameters as shown in Table 2 depend on the size of the spatial feature in question.

Param.	Explanation
$n = 5$	grid fanout in each dimension
$a = 50m$	grid spacing at level 0
$\theta = 1$	min. cell occupancy, termination criterion
h	nof. "hottest" cells
s	$s \times a$ distance between sparse cells
k	discarding isolated cell threshold

Table 1: Search parameter settings and overview

Param./Feature	point	sm. area	city	lg. area
h	4	10	5	5
s	10	10	15	20
k	0.25	0.10	0.05	0.01

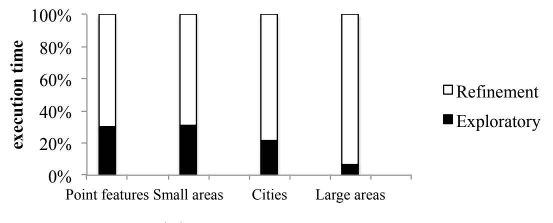
Table 2: Values of adjustable search parameters

To obtain the following experimental results, the web application was installed on a 64-bit Ubuntu server virtual machine with QEMU Virtual CPU clocked at 2.3GHz with 4 cores, and with a total RAM capacity of 4GB. The web application is a Ruby on Rails 3.1 backed up application, using the jQuery 1.7 Javascript framework and the database used is a MongoDB 2.0 server. The use of multiple browsers is simulated by creating a script that would automatically launch the number of browsers required. For simplicity, all the browsers run on the same machine. The browser of choice was Google Chrome.

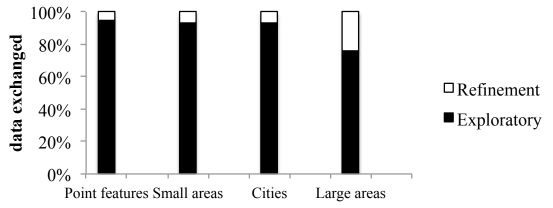
What follows are the results of our performance experiments that detail the *cost attributed to the various searches in terms of transferred data and execution time.*

6.2 Exploratory and Refinement Search

Exploratory search establishes a rough position estimate that is subsequently refined using a hierarchical grid-based search.



(a) Time distribution



(b) Data distribution

Figure 12: Time and data distribution between each phase of the search split by category.

Figure 12(a) shows the percentage of *time* spent for the varying types of spatial objects split by search type. As the time spent on exploratory search is the same for all searches the *refinement search largely determines the overall search time.* The key parameter of the refinement search is the hierarchical grid configuration for a specific search determined by (i) the starting level (top), the finishing level (bottom), and the spatial extent of the object(s) being searched. The starting level is determined by the outcome of the exploratory search comprising a base grid that defines the rough estimate of the spatial extent of the object. The starting level of the refinement search is calculated by the routine that aggregates the base grid into coarser higher level grid cells as explained in Section 4.2. The finishing level is selected based on pre-existing knowledge with respect to the size of the object. In terms of search *time* using one browser, the exploratory phase takes about 1m to complete, i.e., 16 requests taking approximately 4.2s each. The refinement search can take from a couple of minutes up to almost half an hour for some very large scale objects. This assumes that an average refinement search request takes about 300ms.

The *data* retrieved for each search case is shown in Figure 12(b). While a considerably smaller search time is attributed to the exploratory search, the respective data retrieved amounts to 95% of all data retrieved from the point-cloud data source. This is due to different nature of the data being fetched during these phases. In exploratory search we fetch 250 photo attributes per request, including their unique id, their location information, and the url to the actual photo stored on the servers, while during refinement search this information is fetched essentially for only one photograph plus the total count for the specified bounding box.

In terms of size, one exploratory search request comprises 89KB, while a refinement search request fetches only 22bytes of data from the Web data source. This results on average in data sizes of 1.36MB and 26KB for the exploratory and the refinement search respectively.

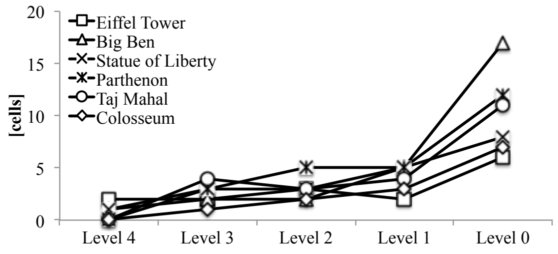
Table 3 summarizes the absolute times spent and data exchanged for four example cases. It is evident from the table that the costly part (time) of the algorithm is indeed the refinement phase. Nevertheless, this phase retrieves very little data compared to the exploratory search.

Search query	Time (mm:ss)		Data (bytes)	
	expl.	refin.	expl.	refin.
Eiffel Tower	00:56.2	01:39.0	1,407,770	9,538
Disneyland	01:05.0	02:13.7	1,427,622	10,717
London	01:14.5	05:51.3	1,423,609	36,102
California	01:02.3	23:51.0	1,433,566	101,674

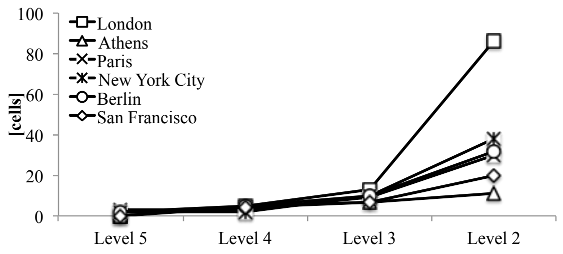
Table 3: Time and data values for example search queries, one browser

6.3 Refinement Search

Refining the positional estimate is the time-wise costly part of our geospatial search. Some intuition is given in Figure 13. The figure plots the number of visited cells per hierarchy level as the refinement search progresses, ranging in each case from the starting level (the level the exploratory search identifies as the starting point for the refinement search) to the desired finishing level. Each subsequent level in the hierarchical grid grows in terms of number of cells visited. This is of course expected as we move from coarser levels to more fine-grained ones to better define the object's spatial extent. Using a 5 $\times$ 5 grid (n = 5), one might expect that the number of visited cells increases with each step by 25. However, in our algorithm, we employ two techniques that try to reduce these numbers: the *filtering process* and the use of *core cells.*



(a) Point features



(b) Cities

Figure 13: Line graphs that show how each hierarchical grid level's size tends to grow as the search progresses split by category.

The filtering process (cf. Section 4.2) runs after all cells of a specific level have been searched and discards irrelevant cells, thus slowly carving out the shape of the spatial object. Core cells try to achieve the opposite by finding very relevant cells, i.e., cells that are completely surrounded by neighboring cells, and whilst keeping them in the result of the search, avoid them being searched over and over again. This implies that the object in question has a solid spatial extent, with no "holes", as is true for most cases. This technique tries to identify the "core" of the object as quickly as possible and then focuses on refining the edges of its shape. Using these two techniques, the increase in number of cells by progressing to the next levels is kept to an average of 2 (instead of 25) (cf. Figure 13).

	Pt. feat.	Sm. areas	Cities	Lg. areas
Average	92.2%	88.3%	90.9%	71.7%
Cum. Average	99.6%	98.2%	98.8%	81.5%

Table 4: Percentage of cells that got discarded and eventually not searched thus improving the speed and reducing the data downloaded by the algorithm.

In this context, one interesting metric is the percentage of cells that ends up being eliminated. This metric, as shown in Table 4 can be computed in two different ways. The first method is to simply compute the average percentage of the cells discarded at each level. The second approach calculates this percentage in a cumulative way by figuring out the number of cells discarded compared to the number of cells the grid would have if its highest level were to be dissected directly into base grid cells. This is cumulative in the sense that discarded cells of one level count also as n^2 discarded cells of the level below and so on, where n^2 is the amount of cells of level $l - 1$ enclosed in one cell of level l.

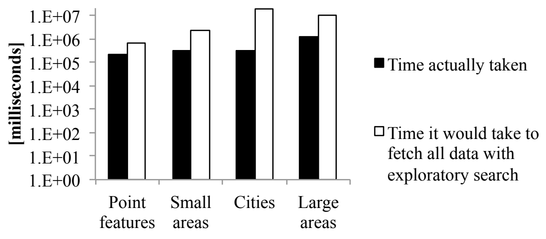
6.4 Overall Search Results

One question that still needs to be answered is whether our search is an actual improvement over the simplest way of computing spatial positions, namely retrieving all location data that is stored on the external providers for a specific search term. Our experiments clearly show that our algorithm performs not only faster, but also uses a lot less data when compared to the exhaustive approach.

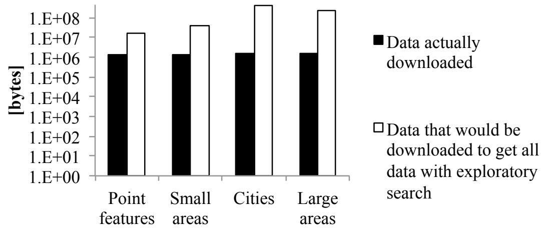
Figure 14 shows the results of that comparison. Both the speedup in the execution time and the decrease in data downloaded are very significant. Table 5 gives the actual percentages. The dominant case observed is the "Cities" category. Here, both the search time speedup and the decrease in downloaded data is very high. This abnormality can be explained primarily by the fact that "city-type" searches have a very high number of total available photos since city names are an expressive tag for photo descriptions. Thus, when just searching with a city name as a keyword one can expect that there will be a vast number of results and that most probably this search will include results from the first two categories - namely point features and small areas - that can be found in the specified city.

What is further interesting, cities do usually have a large enough spatial extent to allow for core cells to be identified, thus giving an additional performance advantage over searches in other categories. Large areas tend to include several isolated places of interest rather than being one large connected feature, i.e., disfavoring core cells.

6.5 Parallelizing Search



(a) Time speedup



(b) Data gain

Figure 14: Comparison of our algorithm against the simplest way of fetching all data via exploratory search, both time-wise and data-wise.

Time speedup and data gain				
	Pt. feat.	Sm. areas	Cities	Lg. areas
Time	29.32%	12.26%	1.65%	14.45%
Data	8.46%	3.70%	0.33%	0.71%

Table 5: Time taken and data downloaded as a percentage of the values the simplest way of fetching all data via exploratory search would achieve.

The proposed method allows for the parallelization of search, i.e., the more users search for a spatial feature at the same time, the faster the spatial extent of the feature is computed. To measure the speedup in search, we performed experiments with a varying numbers of browsers searching concurrently. We expect the speedup to be more significant for cities and larger areas rather than smaller ones, because of the synchronization points in the algorithm (when switching from exploratory to refinement, and when descending each level during refinement) and because of the increased availability of assignable tasks. For the exploratory phase the speedup will be the same for all categories (same number of pages is fetched in all cases). The refinement search performance however will be affected by the search category. For larger areas, a plethora of blocks is available for assignment to clients for simultaneous processing before reaching the synchronization point of progressing to the next (lower) level. The average speedup measured per category can be seen in Figure 15. The speedup is not very significant for point features, since a lot of overlap exists between assigned search tasks. For larger-scale objects, each browser will be assigned a fair share of the total area. Here a possible speedup of > 7 is possible.

6.6 Summary

Experimentation showed that the proposed method is a very efficient way for searching and discovering geospatial

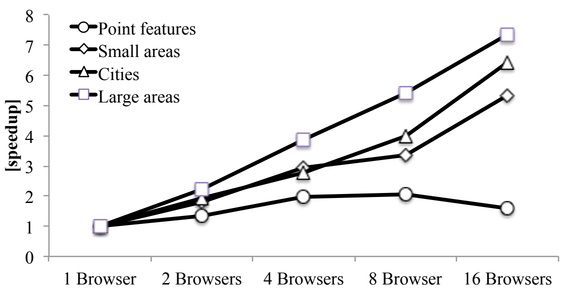


Figure 15: Speedup when using multiple browsers simultaneously

objects in user-contributed point cloud data. It provides a way to minimize, both, computation time and the data fetched from the Web sources. The ability to parallelize the search further decreases the amount of time required and further improves the search performance.

7. CONCLUSIONS

The paper proposes a collaborative, browser-based spatial-feature search mechanism that employs crowdsourcing techniques to harness user-contributed point-cloud data by means of user-contributed expertise and computing resources. To minimize the data transmitted over the network and to speed up the search, a hierarchical grid is used as means to quickly establish a position estimate (exploratory search), to direct the search, and to refine the geometry of the spatial feature in question. Using or own version of what we refer to as a browser-based MapReduce approach, the search is delegated and all computation takes places on Web clients using algorithms implemented in Javascript. Experiments establish the efficiency of the approach, both, in terms of search time and the amount of data that needs to be retrieved to provide an accurate position estimate. These characteristics make it also possible to run a search on mobile devices with limited bandwidth and restricted data plans. Finally, constantly improving browser-technology and specifically the Javascript engines will further improve the search performance. Directions for future work are to introduce a comprehensive mechanism to evaluate the quality of the derived data and to provide quality guarantees. The quality and coverage will be improved by including further data sources and by considering existing geospatial metadata for inferring relationships and hierarchies between spatial objects (“part-of” relationships). As cities are changing and growing, people are discovering these places by means of taking pictures and blogging about them. Using this content to derive geospatial data, one needs to properly consider the evolving nature of this data.

Acknowledgements

The research leading to these results has received funding from the EU FP7 - Marie Curie Actions, Initial Training Network GEOCROWD (http://www.geocrowd.eu) under grant agreement No. FP7-PEOPLE-2010-ITN-264994.

8. REFERENCES

[1] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *Proc. 21st SSDBM conf.*, pages 302–319, 2009.

[2] R. Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[3] K. Chodorow and M. Dirolf. *MongoDB: the definitive guide*. O’Reilly Media, Inc., 2010.

[4] D. Coleman, Y. Georgiadou, and J. Labonte. Volunteered geographic information: the nature and motivation of produsers. *Int’l Journal of Spatial Data Infrastructures Research*, 4:332–358, 2009.

[5] A. Cope. The Shape of Alpha. Where 2.0 conf. presentation, http://whereconf.com/where2009/public/schedule/detail/7212, 2009.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. 6th OSDI Symp.*, pages 137–150, 2004.

[7] S. Elwood. Volunteered geographic information: key questions, concepts and methods to guide emerging research and practice. *GeoJournal*, 72(3):133–135, 2008.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. 2nd KDD conf.*, pages 226–231, 1996.

[9] A. Flanagin and M. Metzger. The credibility of volunteered geographic information. *GeoJournal*, 72(3):137–148, 2008.

[10] M. Goodchild. Uncertainty: the Achilles Heel of GIS? *Geo Info Systems*, 8(11):50–52, 1998.

[11] B. Hecht and D. Gergle. On the localness of user-generated content. In *Proc. 2010 ACM conference on computer supported cooperative work*, pages 229–232, 2010.

[12] S. Intagorn, A. Plangprasopchok, and K. Lerman. Harvesting geospatial knowledge from social metadata. In *Proc. 7th ISCRAM conf.*, 2010.

[13] G. Lamprianidis and D. Pfoser. Jeocrowd: collaborative searching of user-generated point datasets. In *Proc. 19th ACM GIS conf.*, pages 509–512, 2011.

[14] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 144–155, 1994.

[15] T. Rattenbury and M. Naaman. Methods for extracting place semantics from flickr tags. *ACM Transactions on the Web*, 3(1):1–30, January 2009.

[16] J. Surowiecki. *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations*. Doubleday, May 2004.

[17] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song. Accelerating spatial data processing with mapreduce. In *Proc. 16th ICPADS conf.*, pages 229–236, 2010.

[18] W. Wang, J. Yang, and R. R. Muntz. Sting: A statistical information grid approach to spatial data mining. In *Proc. 23rd VLDB conf.*, pages 186–195, 1997.