GeoStream - Exploiting User-Generated Geospatial Content Streams

FP7-SME-2012-315631

Web Computing Framework
D3.1

Deliverable lead contractor: IMIS / ATHENA R.C.

Main contributors:
George Lamprianidis glampr@imis.athena-innovation.gr
Dimitris Skoutas dskoutas@imis.athena-innovation.gr

Due data: 30.09.2014

Abstract

This deliverable presents a novel application of the map-reduce computation concept in connection with browser computing to accomplish mining and integration of user-generated content by means of Web site visitors that contribute their "expertise" and computing power (browser) to perform this task.

Research Center "ATHENA," Greece
FU BERLIN, Germany
Fraunhofer, Germany
Michael Mueller Verlag, Germany
TALENT, Greece
WIGEOGIS, Austria

Table of Contents

List of Tables

Table of Figures

1 Introduction

The constantly increasing participation of users in social networking sites and in efforts and initiatives for open and collaborative data publishing has led to an explosion of crowdsourced geospatial data on the Web. This new wealth of sources and content opens up new opportunities for improving, enriching and enhancing services in the geospatial domain, such as location-based services, trip planning, etc. However, in its raw form, this content exists in very disparate sources and heterogeneous formats, often being incomplete and/or inaccurate. Thus, several challenges need to be tackled to allow for such data to be harvested, processed, analysed, and used in applications.

In GeoStream, we develop methods and tools to address these challenges, focusing in particular on (a) harvesting, integrating and mining user-generated geospatial content from the Web, (b) developing a Web and mobile computing framework for managing such content and supporting applications, (c) designing and implementing rich authoring tools to further facilitate and encourage users in providing such content, in ways that make it also easier to process it.

In this deliverable, we present a framework that utilizes browser-based computing for the compilation of user-generated geospatial content. More specifically, we design and implement a novel application of the map-reduce computation concept in connection with browser computing to accomplish mining and integration of user-generated content by means of Web-site visitors that contribute their "expertise" and computing power (browser) to perform this task.

This deliverable builds on and extends the work presented in deliverables D2.1 and D1.2, which describe the architecture of the GeoStream content store and the processes for integrating and mining user-generated geospatial content from the Web. In particular, it does so by proposing and describing a browser-based computation framework based on a map-reduce architecture to perform the processing of the collected crowdsourced content in a more efficient and flexible manner. This work is accompanied by an evaluation of the framework which is reported in deliverable D3.2.

The deliverable is structured as follows:

- Section 2 provides briefly preliminaries regarding the steps applied to collect and integrate content in the GeoStream data store. More details on these issues can be found in previous deliverables; here this information is briefly presented to make the current document self-contained.
- Section 3 presents the architecture of the developed framework, which follows the MapReduce paradigm, detailing the main components and the workflow of the process.
- Section 4 explains the browser-based computation process, presenting both the client- and the server-side components, as well as the Web monitoring interface.
- Finally, Section 5 concludes the document.

2 Preliminaries

In this section, we briefly outline the steps taken by the GeoStream data miner to collect, process, and analyse crowdsourced geospatial data from Web sources. In particular, this workflow comprises the following main operations: (a) data collection, (b) mapping of categories, (c) matching of spatial entities, and (d) clustering of spatial entities.

2.1 Data collection

In the context of the project, we are interested in four types of (geo-)content:

- **points of interest (POIs)**, which include places that users would like to visit (e.g., museums, monuments) or use services from (e.g., restaurants, shops, banks, train stations);
- **photos**, which allow users to obtain visual information about a place;
- **events**, e.g., concerts, exhibitions, which users may be interested to attend;
- **text feeds**, in particular, Twitter messages, which may provide further information about a location.

To collect such data, we have considered sources that are very popular and widely used, and which provide their content in an explicit manner (e.g. via a link to download data, a query interface or an API). The list of sources used is presented in Table 1.

Source	URL	POIs	Photos	Events	Text
DBpedia	http://dbpedia.org	√			
OpenStreetMap	http://www.openstreetmap.org	√			
Wikimapia	http://wikimapia.org	√			
Google places	https://developers.google.com/places	√			
Foursquare	https://foursquare.com	√			
Flickr	http://www.flickr.com		√		
Panoramio	http://www.panoramio.com		√		
Eventful	http://eventful.com	√		√	
Last.fm	http://www.last.fm	√		√	
Twitter	https://twitter.com				√

Table 1: Selected sources and types of content.

Table 2 summarizes the various input and output parameters involved in the retrieval process for the different sources. We use the symbol "√" to indicate that this information is provided in all or most cases, and the symbol "~" to indicate that it is provided in some cases.

For each of the data sources described above, we have implemented a client application that accesses the corresponding API (or SPARQL endpoint) and retrieves data for a particular region. The retrieved data are then stored in the GeoStream Content Store (see Deliverable D2.1). The core functionality of these applications is similar, with some adaptations according to the different access methods, parameters, and restrictions of each source.

		DBpedia	OSM	Wikimapia	Google places	Foursquare	Flickr	Panoramio	Eventful	Last.fm	Twitter
Input	Location	√	√	√	√	√	√	√	√	√	√
	Type	√	√		√						
	Date						√	√			
	Language	√	√	√	√						
	Format			√							
	Page count						√	√	√		
	API key			√	√	√	√	√	√	√	√
Output	Label	√	√	√	√	√	√	√	√	√	√
	Type	√	√	√	√	√			~		
	Description	√		√		√	√	√	√	~	
	URL	√	√	√	√	√	√	√	√	√	√
	Location	√	√	√	√	√	√	√	√	√	√
	Address				√	√			√	√	
	Tags			√		√	√	√		√	
	Ratings/Comments			~	~	√	√			~	
	Photos	~		~	~		√	√	~		
	Date						√	√	√	√	√
	Opening hours				~	~					

Table 2: Input and output parameters for data retrieval.

A main issue to deal with is the limitations imposed on the maximum number of allowed requests within a specific time period and the maximum number of results that are returned for each request. As a consequence, if a call is issued requesting data for the whole region of interest, it is very likely that only a small portion of the actually available data is retrieved (even in cases where paging of results is provided and more pages beyond the first one can be requested). One possible way to overcome this limitation is to use a small enough sliding window to iteratively cover the whole area. However, given that not all parts of an area are equally dense w.r.t. the number of POIs, photos, etc. located within them, using a sliding window of fixed size is not optimal. Instead, we apply a divide-and-conquer algorithm that splits the input region into a *hierarchical grid* and recursively retrieves data for each cell. In particular, the algorithm works as follows. Given an input cell, a call to the source API is issued to retrieve data for it. If the number of results is equal to the maximum number of records returned by this API for each call, then the cell is split into four equal cells and the process is repeated for each one. Otherwise, the retrieved records are collected and stored. This allows achieving complete coverage for a large area. Moreover, appropriate time delays are inserted between subsequent calls to ensure that the limits of each source are respected. Thus, the content retrieval process can be quite time consuming (e.g., retrieving data from Foursquare in the area of London took approximately 3 days). Note that, since communication errors may also occur in the meantime, we keep track of the progress (i.e., which subareas have been covered) so that the retrieval can resume from the same point after a failure instead of starting over.

2.2 Mapping of categories

One of the main attributes used in virtually all sources to describe an entity is its type(s) or category(-ies), which indicates the nature or usage of the entity, e.g. museum, restaurant, shopping mall, metro station. This information is very important for allowing users to search and browse the available content. However, each source typically employs its own classification or does not strictly specify a pre-defined set of hierarchically organized categories but rather allows users to freely categorize resources, e.g., by assigning tags to them, using either pre-defined classes or classes created by other users or introducing their own. Therefore, before data collected from different sources can be processed and analysed, a reconciliation mechanism is required to map these various taxonomies to a common classification scheme.

For this purpose, we have defined a reference category hierarchy, including the top level categories shown in Figure 1, and we compute mappings from the categories used by each source to this common classification.

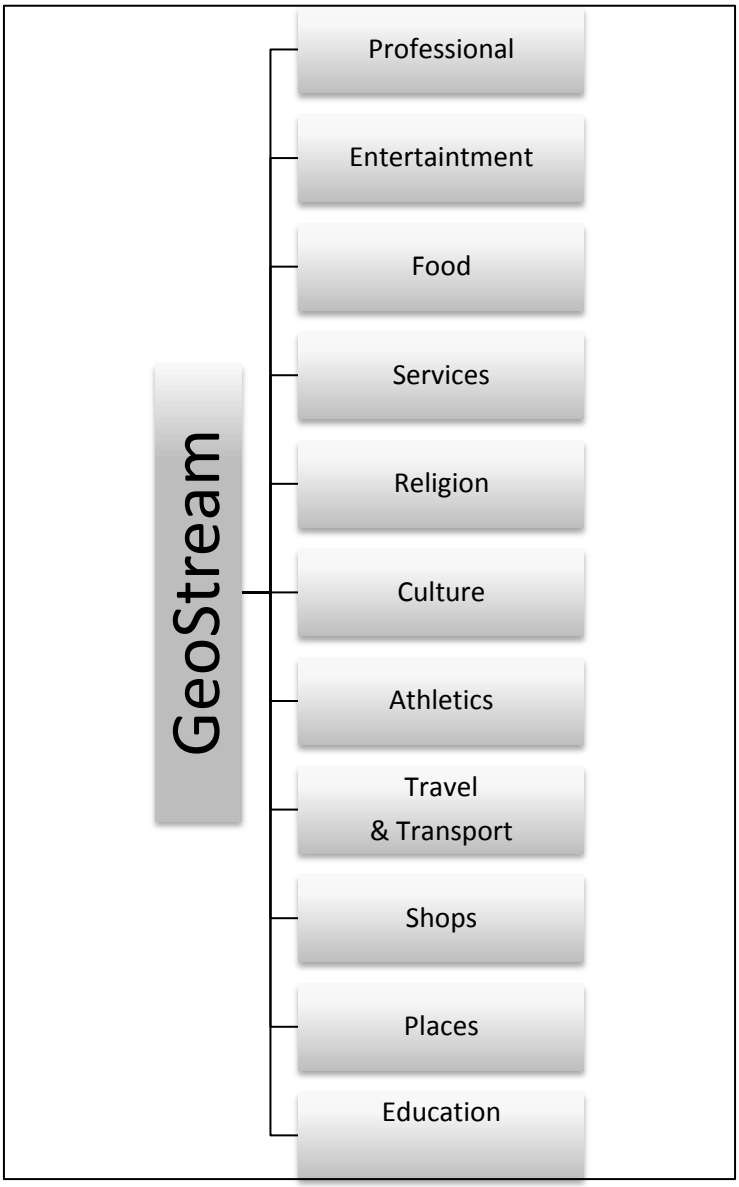

Figure 1: Top level GeoStream categories.

2.3 Matching of geospatial entities

The next major step in the process of integrating and reconciling data collected from the different sources is to match entities across them. This problem arises from the fact that often the same entities, especially popular POIs, appear in many sources, perhaps with different, complementary or sometimes even conflicting representations.

For this purpose, we take into consideration for the matching process two attributes: the location of an entity and its name/label. More specifically, two entities are considered to match if:

- their distance is below a specified threshold, and
- their name similarity is above a specified threshold.

In particular, we perform the matching process as follows. For each entity P_i, we consider a square centred around it and with side length r. For each entity P_j with coordinates inside this square, if the name similarity $sim(P_i, P_j)$ is equal or higher than s, then we consider P_i and P_j to match.

2.4 Clustering of geospatial entities

Content provided by users in a volunteered manner may often be of low accuracy and quality, e.g. containing misspellings, incomplete, or erroneous information. However, although this holds for individual pieces of information, the vast volumes of data and the fact that the same or similar information can be found in various sources and/or by various users make it possible to derive meaningful, high-quality information by aggregating the data and considering statistical results.

Another motivation for aggregating content is the increasing interest to move from the notion of Points of Interest (POIs) to Regions of Interest (ROIs). This is driven by the fact that users often prefer areas with high density of POIs in order to have many alternatives to pick from and to be able to visit multiple places within a shorter period of time.

In our case, the main underlying technique for achieving this task of identifying ROIs is clustering. Clustering groups together similar (according to some specified notion of similarity) items. It comprises one of the most widely used techniques in statistical data analysis. Several clustering techniques and algorithms have been developed over the years. Among the most typical ones are:

- *centroid-based clustering*, where groups of objects are created around central points, and
- *density-based clustering*, where clusters are defined as areas of higher density.

We have used density-based clustering, since the intuition is to identify areas with high concentration of some particular entities of interest. More specifically, the clustering algorithm we have implemented and used is based on the well-known DBSCAN algorithm [1][2] with some adaptations and modifications.

3 MapReduce–like Architecture

The design of our architecture is inspired by the well-known MapReduce computing framework [3], developed by Google Inc. in 2004. 'MapReduce' is a framework for processing parallelizable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogeneous hardware). Computational processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of locality of data, processing it on or near the storage assets in order to reduce the distance over which it must be transmitted.

Borrowing from this idea, we implement a system that can split large jobs into smaller tasks and offer an API for clients to contribute to individual tasks. Given that most of the tasks are spatially bound, we have an inherent way to split jobs into smaller tasks, by applying a grid mask to the area we are interested in, and create tasks that each handles a unique grid cell.

3.1 Overview of the approach

The motivation behind this approach is to build a scalable system that can grow its throughput alongside its user-base and therefore its needs. To accomplish this we can assign extremely time consuming parts of the system to clients in order to offload the server and promote parallelization. Hence we aim to implement a pluggable architecture that includes a master server that can coordinate long running processes and delegate tasks to modular workers that know how to handle each job. The number of workers can be adjusted according to the load of the system and should in most cases provide near linear scalability.

One of the biggest advantages of this approach is the lack of need to explicitly write parallel code. All that is required is to build a tasks assignment system that will split a job into smaller tasks, and make sure that the individual tasks are as less dependent of one another as possible. This last condition is extremely important to achieve better performance when increasing the number of workers, since new workers will not have to wait for previous ones to finish their job before they can start processing their respective task.

We have identified two main areas that our system can benefit from using this approach:

1. The online collection process of available user contributed data from the various external providers
2. The clustering of the collected data to identify regions of interest, i.e. regions with high density of POIs of the same category.

The online collection process is not especially computationally expensive, since the main task is just to contact all the external sources, provide the correct parameters and download the available data. However, due to the high amount of network requests involved, the process can be extremely time consuming. Furthermore, writing parallel code to simultaneously access all these sources is not trivial. As described earlier, our approach removes the need to write parallel code, since if needed we can just increase the number of running workers, to achieve higher throughput.

The detection of regions of interest process is far more computationally expensive. Despite the fact that this process can also be parallelized by dividing

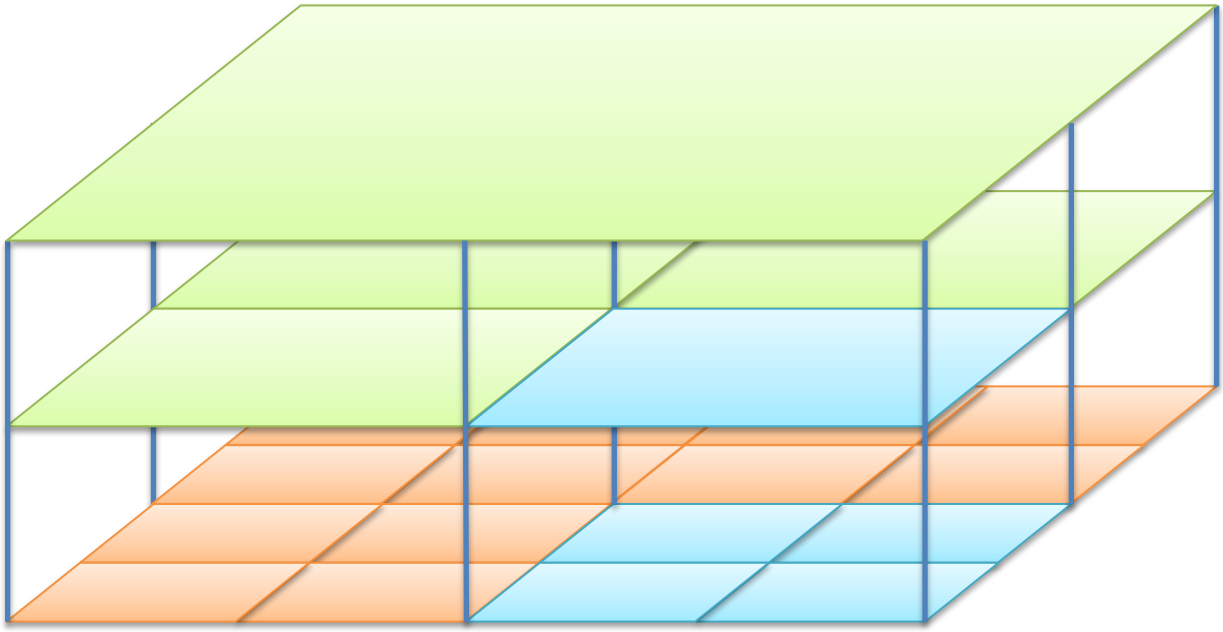

Figure 2. Splitting an area into quadrants.

the original area into smaller parts, there exists a tight coupling between the outputs of each of the smaller tasks, as clusters of each individual cell eventually have to be merged. So this presents a nice case to benchmark our approach in scenarios where the individual tasks are not completely independent.

The next section presents the workflow and the main concepts in more detail.

3.2 Main components and workflow

The task at hand here is to design a generic mechanism that given an area and a task will initially divide the area into smaller regions and then assign the task and each of the small regions to available workers for processing.

This mechanism will be exposed via a programmatic Ruby API and an HTTP endpoint so that it can coordinate not only standard Ruby scripts (for example our existing download clients for Flickr, Panoramio, etc.) but also workers implemented in any kind of programming language or platform.

To demonstrate this, we have re-implemented our download clients in Ruby to be conformant with the new API and we implemented from scratch the same clients in JavaScript. This allows both clients to run simultaneously and in as many "copies" as needed. For example, for a given area, we can initiate two Ruby Panoramio clients, and open three browsers to load the JavaScript Panoramio clients, resulting in having 5 clients collecting Panoramio data for the area at the same time.

We also implemented a JavaScript version of the DBSCAN clustering algorithm to identify regions of interest in a given area. Further details regarding the use of browsers as a computing platform are given in Section 4 . The rest of this section describes the workflow for each of the two cases, regardless of the implementation of the workers. The main concept includes splitting a given area either iteratively or recursively into quadrants (Figure 2) then running the problem's algorithm within these smaller cells. Due to the fact that the collection and the clustering process slightly vary they will be described separately.

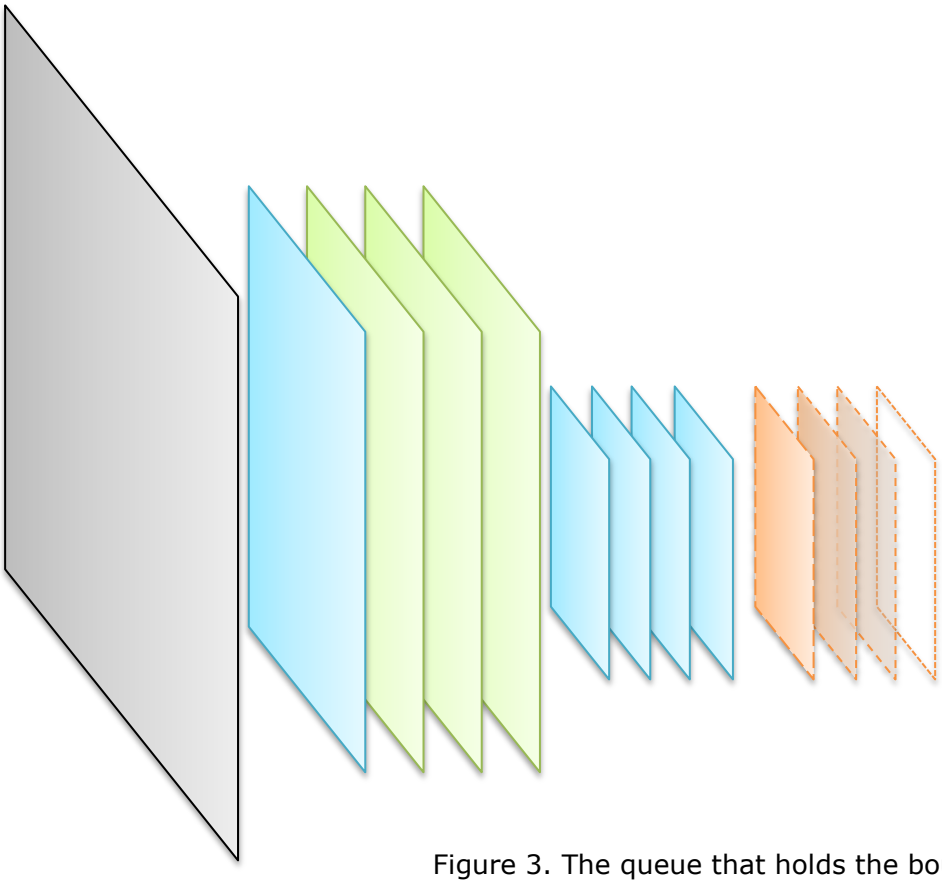

Figure 3. The queue that holds the bounding boxes that are pending for processing.

3.2.1 Data collection

The data collection process uses a top-bottom approach. As the data distribution is not known beforehand, the splitting of an area into smaller cells has to be done as new data is downloaded. The entire processed is governed by four constants:

- `QUERY_RESULTS_LIMIT`, maximum number of results within a cell that we want to retrieve
- `QUERY_RADIUS_LIMIT`, maximum radius or length of cell that we are allowed to search within
- `QUERY_RADIUS_SIGNIFICANT`, minimum radius or length of cell that we want to search within
- `QUERY_MAX_PER_PAGE`, maximum number of results that we are allowed to retrieve with a single request from the source

The process begins by adding the original bounding box of the area that we want to collect data from into a queue (Q) as shown in Figure 3. As long as there are items in the queue, workers can dequeue bounding boxes from Q for processing. Once a bounding box B is dequeued, we check if its longer edge is bigger than `QUERY_RADIUS_LIMIT`; if it is, then the bounding box is split into four quadrants, gets marked as "exceeded_radius", and the four quadrants are in turn enqueued into Q; if not, then we can start processing it.

Processing starts by checking the number of available data already collected inside B (for example from previous runs of the search or from queries for bounding boxes that enclosed B, i.e. ancestors). If the number of existing results inside B is greater or equal to `QUERY_RESULTS_LIMIT` then we immediately split B into quadrants and don't search it as an entity, since it will anyway gather more than `QUERY_RESULTS_LIMIT` results in the end and will eventually have to be

dissected. Otherwise we can start querying the external source for new data within B, by making requests that will download at most `QUERY_MAX_PER_PAGE` results at a time until reaching `QUERY_RESULTS_LIMIT` or exhausting all available data. If we manage to collect `QUERY_RESULTS_LIMIT` results then it is likely that more results are available so B is marked as "exceeded_results" and split into four quadrants that are enqueued into Q.

During this process, every time a bounding box is split into quadrants we check if the resulting children bounding boxes have their longest edge less than the `QUERY_RADIUS_SIGNIFICANT` parameter, in which case the splitting is prevented. The process continues until Q is empty. At that point data for the entire area has been downloaded, according to the granularity specified by the four aforementioned constants. The workflow of the procedure is outlined in Figure 4.

For each of the available data providers the workflow remains the same. The only differentiation is the "Collect data" part and the values of the four constants.

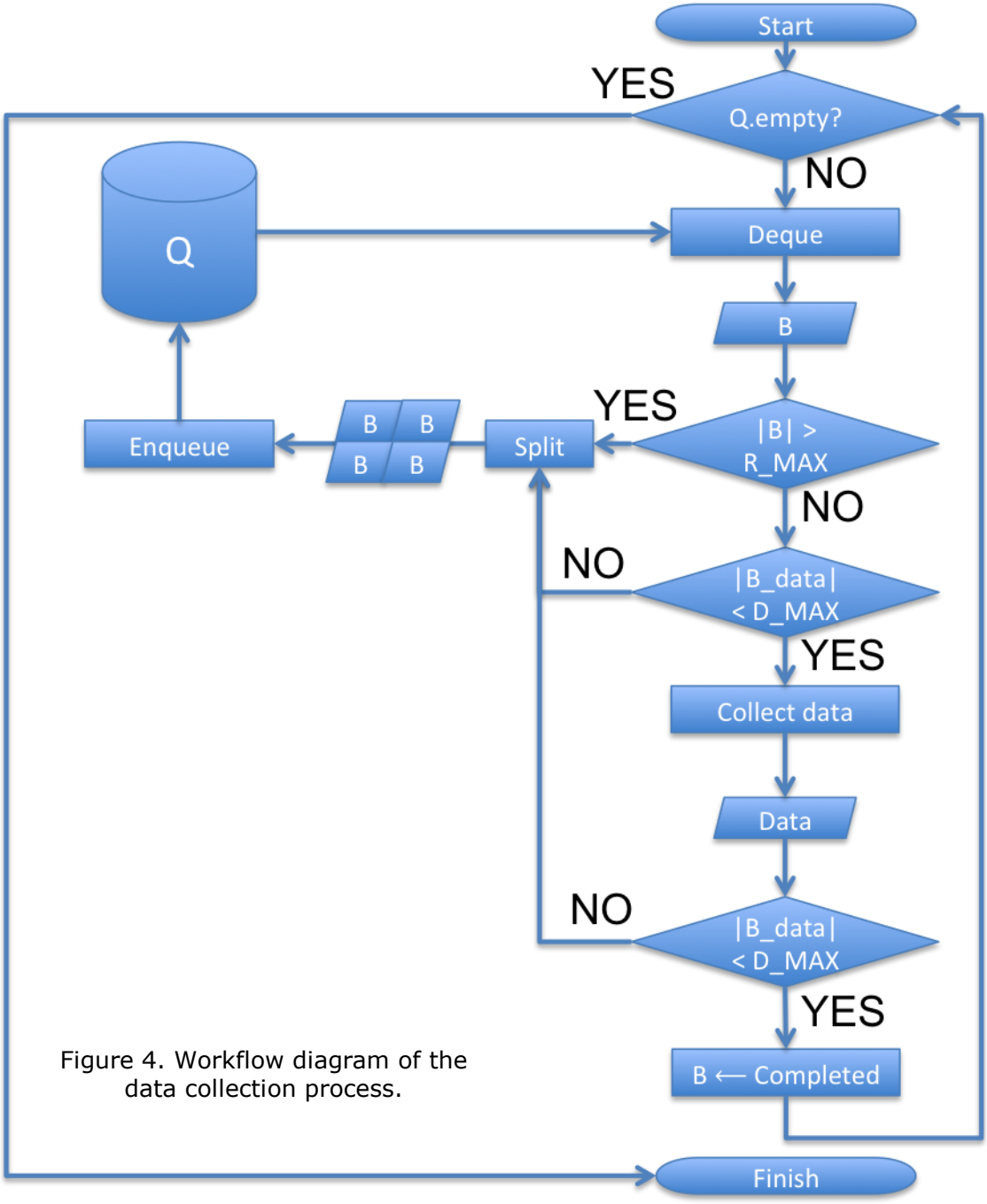

Figure 4. Workflow diagram of the data collection process.

These are implemented in a different class for each provider, to handle its particularities, e.g. parameter naming, specific pagination, quota limits, etc.

3.2.2 Data clustering – Regions of interest

The clustering procedure as opposed to the data collection follows a bottom-up approach. Here the data is known a priori, so the entire area can be split based on some criteria, before starting the assignments to the workers.

The only parameter required for the split is P_{max}, the total number of POIs enclosed within a grid cell. Given this number a recursive operation begins that checks if the current bounding box contains more than P_{max} POIs. If it does, it is split into four quadrants and the recursion continues.

Once this process is done, we can start assigning cells to workers. The grid cells can be viewed as a quad-tree with the cell corresponding to the entire area as the root and the smallest cells as leafs. In this notion, only leaf cells need to be assigned to workers for clustering. Intermediate nodes need only to run a merge procedure once all children are complete. The merge procedure can be run by the scheduling system or assigned to a worker. In our implementation we decided to leave the merging to the main infrastructure, i.e. the scheduling system.

The workflow of the clustering procedure is shown in Figure 5. To keep the flow as similar as possible to the previous example we can add all leaf cells to a queue Q. After the initial dissection of the area, starting from the deepest leaf cells, we run the DBSCAN algorithm to each one of them. Once all quadrants have computed clusters, a merging procedure runs that fuses the ones with a non-blank intersection. The merging repeats recursively and in parallel with assignments of other disjoint leaf cells. The procedure ends when all leaf cells have computed clusters, and subsequently all merge procedures have run for all the intermediate levels of the quad-tree up to the root.

For this procedure we only implemented JavaScript clients, as will be described in the next section, but similarly to the collection process, our design allows for implementations in any programming language to be used as workers for the system, by using the HTTP endpoint to communicate with

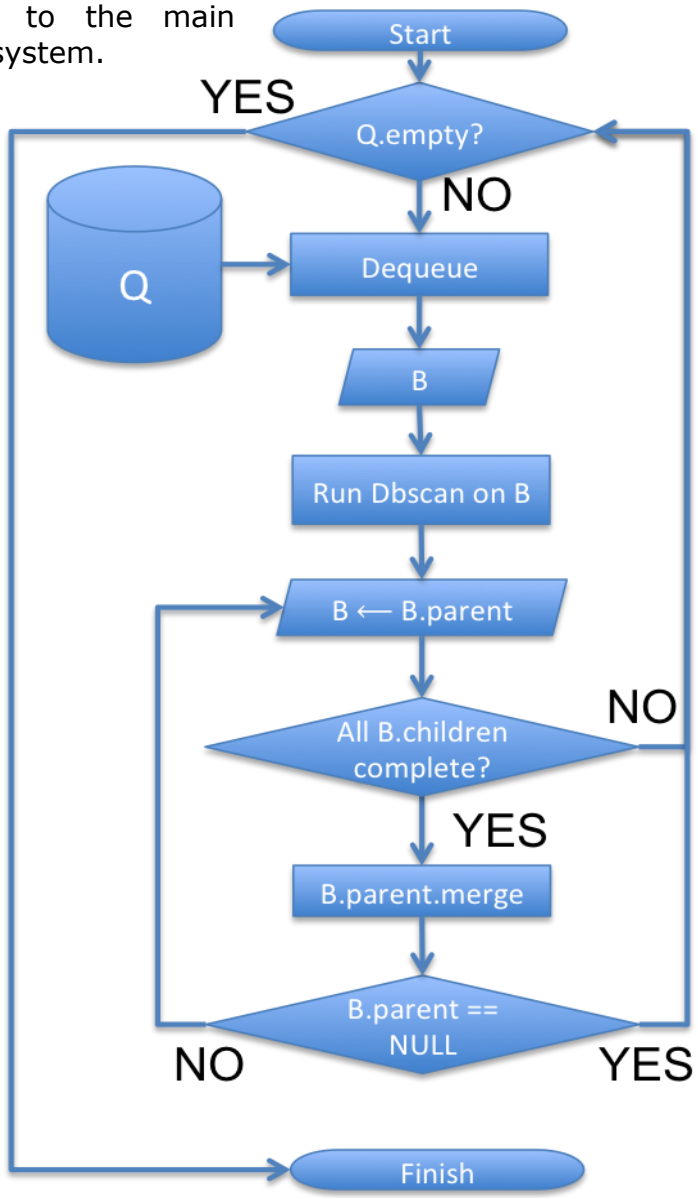

Figure 5. Workflow diagram of the data clustering process

the scheduling mechanism.

4 Browser-based Computation

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 coordinates the process. As a side effect, wherever data retrieval from external sources is needed, no actual data is downloaded directly by the server, but all traffic is handled by the clients, effectively balancing the data over the network.

With the scheduling mechanism already in place – as described in the previous section – all that is needed is to expose this mechanism to the browsers via an HTTP restful API. The API consists of two endpoints as shown in Table 3: the first one is used to GET the next pending task, while the second one is used to POST the results back to our system and receive the next pending task, if any, as a response. The `areaId` parameter of the URL corresponds to the internal identifier of the area we are interested in processing. The `controller` is set depending on the task executed, e.g. data collection process requires the `requests` controller, while the POI clustering process requires the `cluster_cells` controller.

HTTP Verb	Path	Description
GET	`/admin/areas/{areaId}/{controller}/next_pending.json`	Retrieve details about next pending job
POST	`/admin/areas/{areaId}/{controller}/submit.json`	Submit results back to server

Table 3. Overview of HTTP API for tasks management.

Finally we have to implement each Web client to handle its respective tasks. Since all clients work in a similar fashion, we have used inheritance to declare all this common behaviour implemented in the main class WebClient. Subclasses for the various external providers, e.g. WebClient.Panoramio, WebClient.Flickr, etc. are used in the data collection process, while the clustering procedure to identify regions of interest uses a separate WebClient.Clustering client.

The source code of the clients is implemented in CoffeeScript, a small language that compiles one-to-one into JavaScript, with no interpretation at runtime. One can use any existing JavaScript library seamlessly from CoffeeScript (and vice-versa). The compiled output is readable, works in every JavaScript runtime, and tends to run as fast or faster than the equivalent handwritten JavaScript. The barebone structure of the WebClient, written in CoffeeScript can be viewed in Figure 6.

4.1 The clients

Typically the workflow remains the same for all clients and follows this order:

1. Contact the server and request the next pending job details (`nextJob`).
2. Build the download URL based on the request details (`buildDownloadURL`).
3. Fetch data from the download URL (`download`).
4. Parse the results (`parseResults`):
   a. Process or perform any required calculations.
   b. Transform them into a compatible format used by our storage engine.
5. Submit only the desired results to the server (`submit`).

6. The server will reply with the next pending job details if any.
7. If a new pending job is given restart from 2.

```
class window.WebClient

  constructor: (@name, @area, @osmMap, @access, options = {}) ->
    @status = "stopped"
    @areaId = @area.id
    @controller = options.controller || "requests"

  buildNextRequestURL: () ->
    "/admin/areas/#{@areaId}/#{@controller}/next_pending.json?provider=#{@name}"

  buildSubmitResultsURL: () ->
    "/admin/areas/#{@areaId}/#{@controller}/submit.json?provider=#{@name}"

  nextJob: () ->
    jQuery.ajax({
      "url"       : @buildNextRequestURL(),
      "dataType"  : "json",
      "success"   : (data) =>
        @request = data.request
        if @request?
          @download()
        else
          @status = "stopped"
          alert("No more pending jobs found! #{data.message}")
    })

  download: () ->
    jQuery.ajax({
      "url"       : @buildDownloadURL(),
      "dataType"  : @getProviderRequestDataType(),
      "jsonp"     : @getJSONPcallback(),
      "success"   : (data) =>
        results = @parseResults(data)
        @updateRequest(data)
        @submit(results)
    })

  submit: (results) ->
    jQuery.ajax({
      "url"      : @buildSubmitResultsURL(),
      "type"     : "POST",
      "dataType" : "json",
      "data"     : {request": @request, "results": results, "status": @getRequestStatus()},
      "success"  : (data) =>
        if @status == "running"
          @request = data.request
          if @request?
            @download()
          else
            @status = "stopped"
            alert("No more pending jobs found! #{data.message}")
```

```
})
```

Figure 6. JavaScript (CoffeeScript) WebClient listing

The names given in parenthesis correspond to the methods called in WebClient and its subclasses to perform the respective task.

Most of the methods described in Figure 6 are used directly from the WebClient superclass without modification. On the other hand the most common overridden methods for each client, include:

- `buildDownloadURL: () ->`

  Based on the next pending job details that were retrieved from the server, the client builds a unique URL to contact in order to fetch all the data needed to perform the assigned task. The most common parameters used to build it include the bounding box of the assigned area and an offset information if paging is required to obtain more data.

- `parseResults: (data) ->`

  This is where the processing of the data on the client side takes places. This method provides specific instructions to the client to manipulate the retrieved data and finally transform it into a form that can be immediately stored in our server.

  For example, in the data collection process, this method is used to transform the retrieved data from the external sources into objects that follow our common schema. That mainly involves renaming field names, transforming values, and removing irrelevant information to reduce network traffic, when submitting.

  On the other hand in regions of interest identification process, this is the method that actually computes the respective clusters for each of the smaller areas.

  The output is being sent to the server via the `submit` method.

- `updateRequest: (data) ->`

  This method is used to send information about the status of the job executed at the client back to the server, in order to schedule the remaining jobs correctly.

  For example in the data collection process, in cases where paging is required to retrieve all the data within a bounding box, it will modify the request object accordingly to signify if:

  - we need to request the next batch (“page”) of results for this bounding box from the external source, or
  - we exhausted all pages and the bounding box needs to be examined deeper (i.e. split into quadrants), or
  - we have collected all the available data for the bounding box and it should be marked as “completed”

  This method is not used in the data clustering process.

One important thing to be aware of is to avoid intense and long running computations in the client’s code. Since JavaScript execution in the browser is single-threaded, this is essential to avoid bad user experience for our visitors, due to their browser becoming unresponsive while trying to execute computationally expensive algorithms (e.g. big nested loops). If long running

algorithms cannot be avoided by better scheduling individual tasks, one might consider the new WebWorkers API, which allows having scripts run in a background thread and communicate with the main thread via a message-passing interface.

4.1.1 Implementation details

The following section presents a listing of the source code in CoffeeScript of two web clients, with explanatory comments for the most critical parts.

```
class window.WebClient.Flickr extends window.WebClient

  @QUERY_RESULTS_LIMIT       = 1500
  @QUERY_RADIUS_LIMIT        = 20000
  @QUERY_RADIUS_SIGNIFICANT  = 200
  @QUERY_MAX_PER_PAGE        = 150

  defaultParams: {
    method: "flickr.photos.search", has_geo: true, accuracy: 12,
    extras: "description,date_upload,date_taken,geo,tags,views",
    sort:   "date-posted-desc", format: "json", per_page: @QUERY_MAX_PER_PAGE
  }

  constructor: (@area, @osmMap, @access) ->
    super("flickr", @area, @osmMap, @access)

  buildDownloadURL: () ->
    offset  = @request.param_offset
    polygon = @request.param_bbox
    offset  = 1 unless offset?
    @from   = parseInt(offset)
    polygon = @osmMap.olWKT.read(polygon)
    bounds  = polygon.geometry.getBounds()
    @params = jQuery.extend({}, @defaultParams)
    @params.bbox = bounds.toBBOX()
    @params.page = @from
    @params.api_key = @access.api_key
    "https://api.flickr.com/services/rest?" + jQuery.param(@params)

  parseResults: (data) ->
    for photo in data.photos.photo
      delete(photo.owner)
      … …
      delete(photo.context)
    data.photos.photo

  updateRequest: (data) ->
    @request.data_count = data.photos.photo.length
    maxPage = Math.floor(@QUERY_RESULTS_LIMIT / @QUERY_MAX_PER_PAGE)
    if data.photos.page < Math.min(data.photos.pages, maxPage)
      @requestStatus = "more"
    else if data.photos.photo.length == @constructor.QUERY_MAX_PER_PAGE
      @requestStatus = "dissect"
    else
      @requestStatus = "complete"
```

Define the constants that handle the workflow and are specific to Flickr

Using the details from the `@request` object that was retrieved from the server and some Flickr default options specific to our task, construct the URL to request the data from

Delete all info that we do not need to store in our database (and rename any fields if necessary) so that the results submitted to our server can be saved with the least manipulation possible

Update the `@request` object sent back to server with info about the retrieval (e.g. `data_count`) and with instruction on how to proceed: `"more"`, `"dissect"` or `"complete"`

Figure 7. Implentation of the Flick web client in CoffeeScript.

The first one shown in Figure 7 contains the source code of the Flickr web client. The web client used for the POI clustering is shown in Figure 8. Both figures show only the most interesting parts of the implementations.

```
class window.WebClient.Clustering extends window.WebClient

  constructor: (@area, @osmMap, @access, options) ->
    @category = options.category
    @minPoints = options.min_pts || 20; eps = options.eps || 0.001
    super("clustering", @area, @osmMap, @access, options)

  buildDownloadURL: () ->
    polygon = @osmMap.olWKT.read(@request.bounds)
    bounds  = polygon.geometry.getBounds(); xbounds = bounds.clone()
    xbounds.extend(bounds.add(@eps, @eps)); xbounds.extend(bounds.add(-@eps, -@eps))
    extPoly = @osmMap.olWKT.write(new OpenLayers.Feature.Vector(xbounds.toGeometry()))
    "/admin/areas/#{@area.id}/venues.json?q[bbox]=#{extPoly}&q[category]=#{@category}"

  parseResults: (data) ->
    @results = @dbScan(data)
    for cluster in @results
      for poi in cluster
        poi.poi_id = poi.id;
        poi.poi_coordinates = poi.coordinates
        … …
    @results

  dbScan: (data) ->
    @clusters = []
    for poi, i in data
      if !poi.visited?
        neighbors = @getNeighbors(poi, data)
        if neighbors.length >= @minPoints
          poi.visited = "PART_OF_CLUSTER"
          cluster = [poi]
          cluster = @expandCluster(cluster, neighbors, data)
          @clusters.push(cluster)
        else
          poi.visited = "NOISE"
    @clusters

  expandCluster: (cluster, neighbors, data) ->
    seeds = $.merge([], neighbors)
    while seeds.length
      poi = seeds.splice(0, 1)[0] # remove and return the first seed
      if !poi.visited?
        neighbors = @getNeighbors(poi, data)
        if neighbors.length >= @minPoints
          seeds.push neighbor for neighbor in neighbors
      if poi.visited != "PART_OF_CLUSTER"
        poi.visited = "PART_OF_CLUSTER"
```

The client manipulates the bounding box of the task received by the server to add a buffer zone around it

Delete all info that we do not need to store in our database (and rename any fields if necessary) so that the results submitted to our server can be saved with the least manipulation possible

CORE PROCESS:

This is the implementation of the DBscan algorithm expressed in CoffeeScript

```
        cluster.push poi
    cluster
```

Figure 8. Implentation of the POI clustering client in CoffeeScript.

4.2 The server

The server that coordinates the JavaScript clients is part of the Ruby on Rails web application that powers the GeoStream website. As described earlier and shown in detail in Table 3 the API consists of two endpoints, one to retrieve details about pending tasks and another one to submit the results.

Each JavaScript client needs to contact the server to get information about the next pending task. All the available jobs are stored in the database. Each task record is tagged with a unique code that identifies the bounding box the task should act upon. The initial area has a NULL tag, while its four children are given tags that correspond to the position of each quadrant, relatively to their parent (SW, SE, NE, NW). Subsequent tasks, concatenate their tag to that of their parent, for example children of task SE, have tags: SESW, SESE, SENE, SENW. The entire set of available tasks can be modeled as a quad-tree, based on the enclosure relationship of their bounding boxes.

As described earlier the data collection process follows a top to bottom approach since the data is not known a priori but only as it gets downloaded. This is why pending jobs with smaller tags are selected first; they are closer to the root of the quad-tree and dealing with them first can create more tasks sooner than later. The more tasks created earlier in the process, the faster the process can complete, if enough workers are available to work in parallel, until all data is retrieved based on the granularity defined by the search parameters.

The clustering process works in a reverse manner. The data is known beforehand thus the process follows a bottom up approach. The final splitting is done before the jobs are assigned to the clients. Here tasks with longer tags are assigned first for processing. This way tasks in intermediate levels do not block unnecessarily while waiting for all their children to compete. As soon as leaf tasks are completed their ancestors immediately start the merging process, until finally the root is merged.

Depending on the task (data collection or POI clustering), after the unique ordering is applied to the records (either by tag length ascending or tag length descending respectively) the first one is locked and marked as assigned, and finally returned to the client for processing. The assignment is marked with the current timestamp. The tasks stay marked as assigned for a predefined time interval, currently set to 3 minutes. If it is suspected that the processing might take more than that, workers can declare that they are still working on a specific task by updating the `assigned_at` timestamp. If an assigned task that is not yet marked as completed, is found to have a stale `assigned_at` timestamp more than 3 minutes old, we consider that the worker it was assigned to was interrupted or has crushed and thus the task can be re-assigned to a different worker.

Once a worker has finished processing a specific task it can submit the results back to the server. The client submits the results along with the requests details and an indication as to how the server should continue distributing tasks. For example in the data collection process the client, alongside the results, submits a request status parameter, that directs the server as to if it should split the current bounding box further and create children tasks, or if it should create task

that will request more data for the same bounding box (e.g. when the external API supports paging), or finally, if the task is complete without needing further splitting (i.e. all available data was gathered).

4.3 Web monitoring interface

The progress and validity of these JavaScript algorithms can be observed and verified via our web application, in a separate page for each process.

4.3.1 Data collection

For the "browser - data collection" process, the progress can be observed on the interactive map on the same page that is used to monitor the "server - data collection". Two extra buttons have been added to the interface as shown in Figure 9:

- "Start browser collecting!" is used to start the JavaScript client
- "Show space distribution" is used to display the bounding boxes – colour-coded with their status – on the interactive map

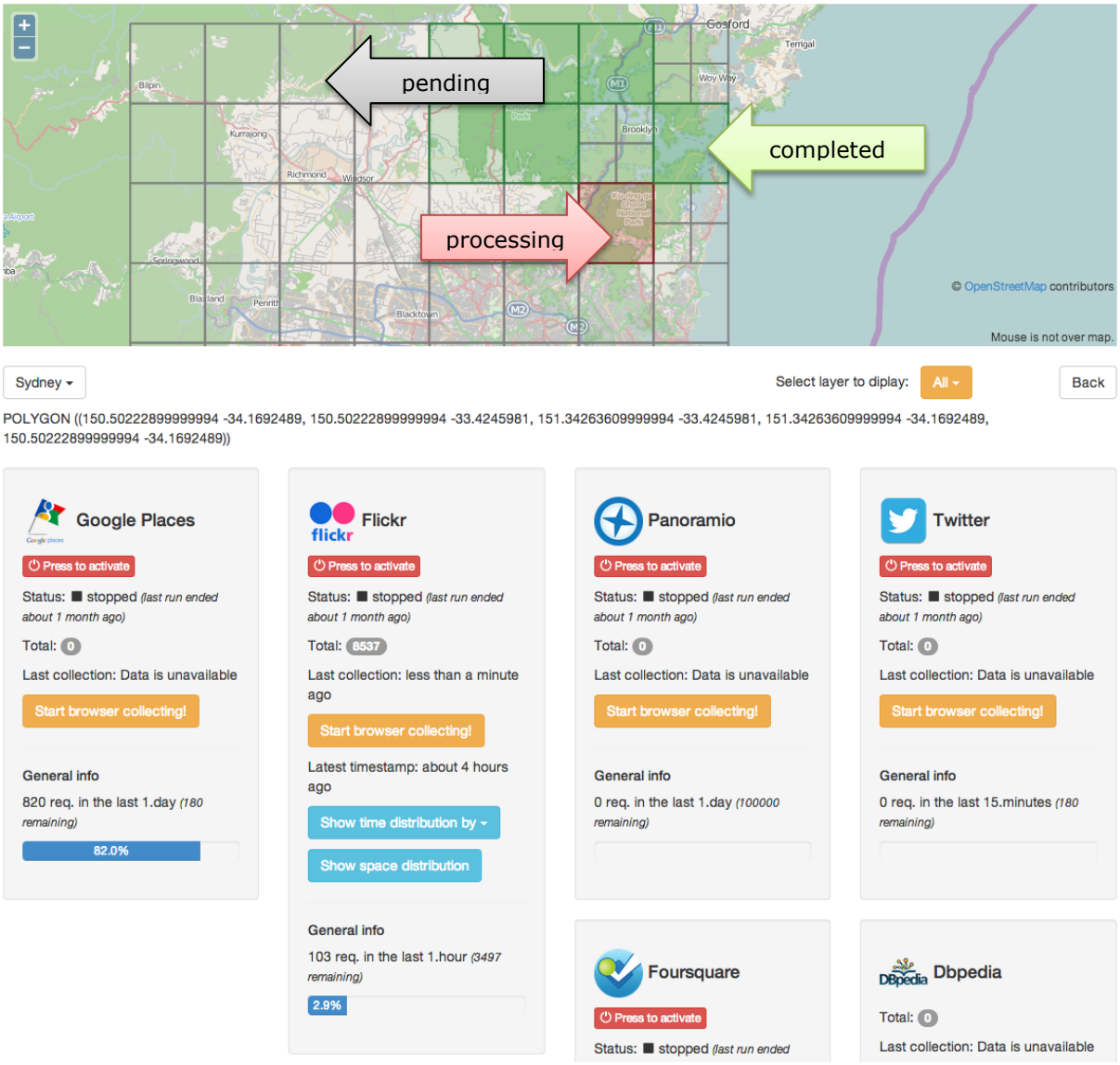

Figure 9. Initiating and monitoring the JavaScript data collection progress

The colour codes are explained in Figure 9. Initially all bounding boxes are pending and are coloured grey. When a client is assigned one and it starts

processing it, it turns red. Once the task has completed, its bounding box on the map is coloured green. The bounding boxes shown in the map are automatically refreshed every few seconds.

4.3.2 Identifying regions of interest

A new page (Figure 10) has been built to allow the JavaScript version of the DBSCAN algorithm to run and monitor its progress. In this page, initially we give the parameters for splitting of the data, specifically P_{max} the maximum number of POIs inside a cell and the category we are interested in. The original area is then split into four quadrants recursively until all cells contain at most P_{max} POIs.

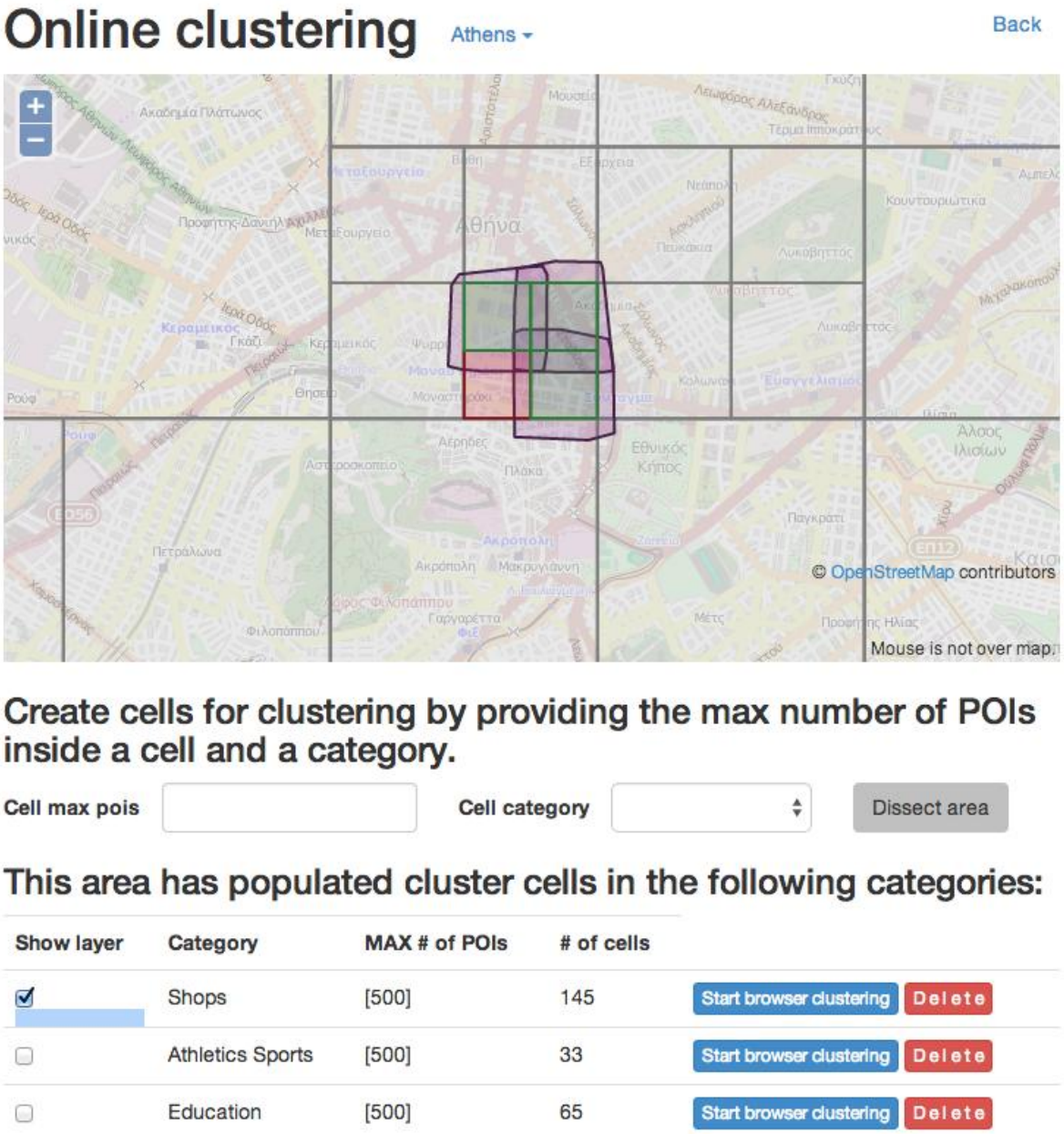

Figure 10.Monitoring the JavaScript POI clustering progress

Once the dissection is complete, we can then initiate the clustering web client and monitor its progress on the interactive map. As shown in Figure 10, each client computes clusters that expand its bounding box by an ε parameter, so that adjacent clusters can be later merged. Once four adjoining cells of the same level

are complete, they get merged into a larger cell. The series of images in Figure 11 illustrates this process.

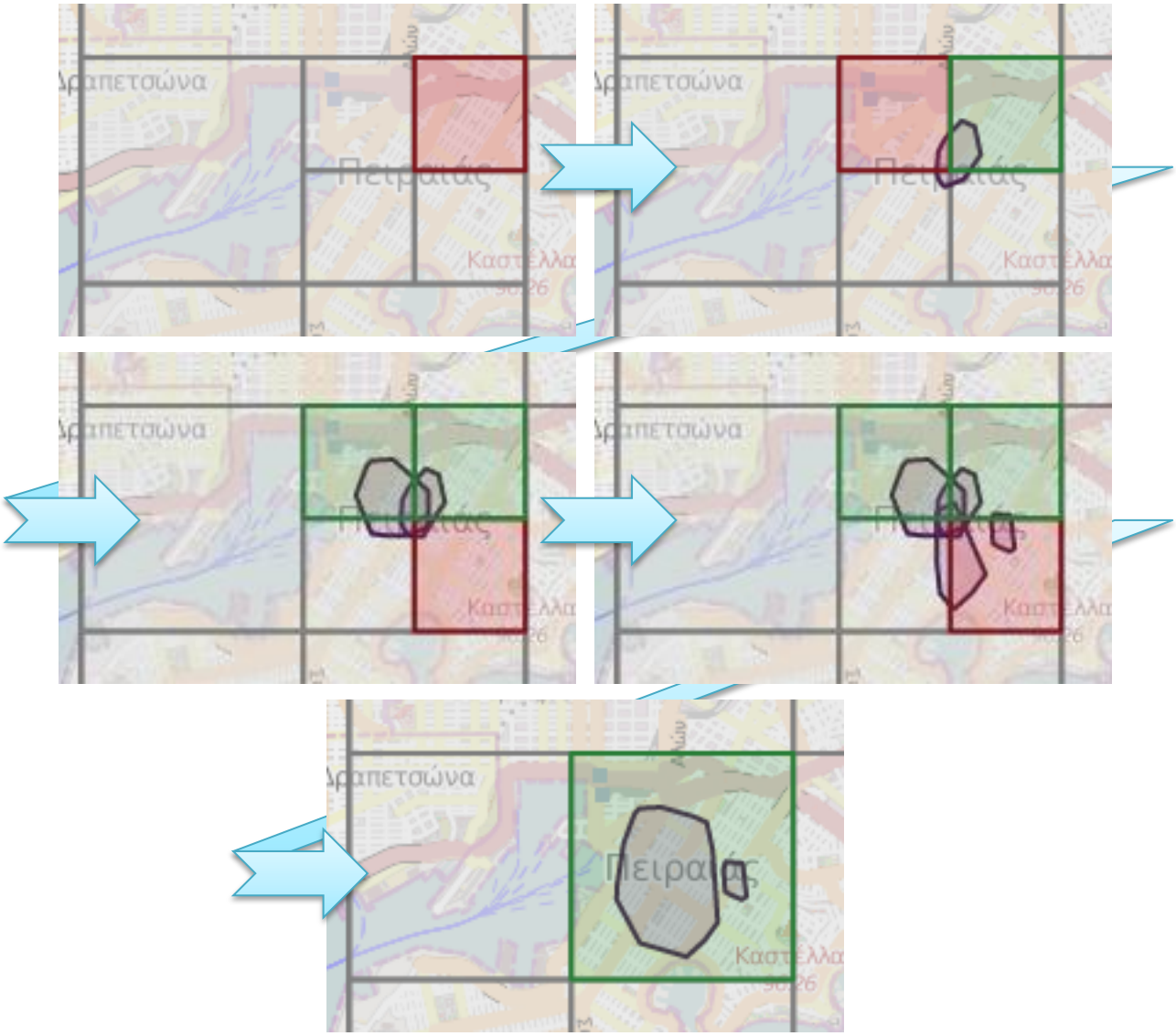

Figure 11. Step by step cluster detection and merging.

The cells on the map are colour-coded with their status following the same convention as described earlier for the data collection process.

5 Conclusions

In this deliverable, we have presented the Web computing framework that was designed and developed for collecting and compiling user-generated geospatial content. The architecture of the framework follows the MapReduce paradigm, but offering also a novel and distinguishing feature, namely the browser-based computation. The framework has been applied to two steps of the processing workflow used in the GeoStream application: content retrieval from Web sources and clustering of the data to identify regions of interest. We analyse the process for each case and present both the client-side and server-side components involved. An evaluation of the presented framework is provided in Deliverable D3.2.

References

[1] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. KDD 1996: 226-231

[2] J. Sander, M. Ester, H.-P. Kriegel, X. Xu: Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. Data Min. Knowl. Discov. 2(2): 169-194 (1998)

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