TAREEG: A MapReduce-Based System for Extracting Spatial Data from OpenStreetMap

Louai Alarabi, Ahmed Eldawy, Rami Alghamdi, Mohamed F. Mokbel

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
{louai,eldawy,alghamdi,mokbel}@cs.umn.edu

ABSTRACT

Real spatial data, e.g., detailed road networks, rivers, buildings, parks, are not easily available for most of the world. This hinders the practicality of many research ideas that need a real spatial data for testing and experiments. Such data is often available for governmental use, or at major software companies, but it is prohibitively expensive to build or buy for academia or individual researchers. This paper presents TAREEG; a web-service that makes real spatial data, from anywhere in the world, available at the fingertips of every researcher or individual. TAREEG gets all its data by leveraging the richness of OpenStreetMap data set; the most comprehensive available spatial data of the world. Yet, it is still challenging to obtain OpenStreetMap data due to the size limitations, special data format, and the noisy nature of spatial data. TAREEG employs MapReduce-based techniques to make it efficient and easy to extract OpenStreetMap data in a standard form with minimal effort. Experimental results show that TAREEG is highly accurate and efficient.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

Keywords

Spatial, MapReduce, Hadoop, GIS, OpenStreetMap

1. INTRODUCTION

There is a major need to have full access to underlying real road networks to experiment and evaluate various algorithms, which include shortest path queries [14, 28, 35, 36, 38, 39], k-nearest-neighbor queries [4, 11, 12, 13, 17, 19, 37, 40], reverse nearest neighbor query [15, 30], range queries [3], skyline query [5, 33], among others (e.g., [16, 32, 39]). Unfortunately, it is always challenging to get such real road network, which imposes a major obstacle in advancing the research in such algorithms. Usually, researchers either: (a) buy actual and accurate spatial data from specialized companies (e.g., NavTeq Here [20]). Yet, this is prohibitively expensive for academia and small startups. (b) share some common road network data among themselves, which result in the case that many research papers use the same road network data for their evaluation. While this may be used as a benchmark, but it also limits the applicability of the developed algorithms to a wide variety of road networks, or (c) rely on their own efforts on extracting road networks from publicly available datasets, e.g., Tiger/Line files [34] (available only for USA road network) and OpenStreetMap [24] (available for the whole world).

However, such approach needs a learning curve, which is not easy for all researchers. In addition, Tiger/Line files [34] are only limited to USA, and hence is of no use to get the road network anywhere else in the world. To fill in this gap, OpenStreetMap [24] has been launched in 2004 to allow volunteers to combine their efforts in building an exhaustive and trustworthy map for the whole world with an increased focus given to road networks in a single 500 GB file *Planet.osm*. Despite its richness, using OpenStreetMap data is not an easy task due to its huge size and non-standard format. For example, to extract the road network of Istanbul, Turkey, one needs to: (1) understand the XML file format and its tags, (2) parse the whole 500 GB file to extract only the parts within the area of Istanbul, (3) within the extracted parts, parse records carefully to extract the XML tags that are related to road networks and exclude everything else, and (4) overcome the noisy data as many of the OpenStreetMap data are voluntarily contributed.

Several attempts were proposed to extract data from OpenStreetMap. For example, the GeoFabrik project [8] allows users to download a predefined area from OpenStreetMap, yet it does not provide any kind of extraction of specific map features (e.g., road network). It is basically a range query service over existing data without any attempt to read the noisy format. Also, OSMOSIS [26] and OSM2PGSQL [25] are two proposed tools that attempt to process the whole *Planet.osm* by loading it on spatial databases. However these tools may take up to several days for loading the entire *Planet.osm* file. For instance, in [18], the authors report that they spent 305 hours (approximately 12 days) for

*This work is supported in part by KACST GIS Technology Innovation Center at Umm Al-Qura University, under project GISTIC-13-14, and was done while all the authors were visiting the center

SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA
Copyright 2014 ACM 978-1-4503-3131-9/14/11$15.00
http://dx.doi.org/10.1145/2666310.2666403.

loading and extracting the data for their experiments. In another benchmark of OSM2PGSQL, importing the *Planet.osm* to database takes up to 7 days of processing [25].

In this paper, we present TAREEG; an easy and efficient system to extract spatial data from OpenStreetMap. TAREEG is set up as an online web service available 24/7 at [31]. An initial version of TAREEG has been demonstrated in SIGMOD 2014 [1]. Unlike all previous approaches, TAREEG overcomes the challenges of processing OpenStreetMap dataset (i.e., *Planet.osm*) with a high accuracy, efficiency, and performance. Users of TAREEG can: (1) extract geographical information for anywhere in the world using a nicely designed web service, (2) export the extracted data in various data formats that include Comma Separated Values (CSV), Keyhole Markup Language (KML), Esri Shapefiles, and/or Well-known-Text (WKT) formats and (3) visualize the extracted geographical data on TAREEG using four different mapping engines, which include OpenStreetMap, Google Maps, Google Hybrid Maps, and Esri Maps. All TAREEG services are made available online 24/7 at [31] to the community in large to give a full access to spatial data. Available spatial data does not only include all road networks in the whole world, but also other spatial features (e.g., lakes, buildings, rivers, and parks). The requested data are sent back to the requesting user in a form of an email with hyperlinks to download the requested data in different formats. The turnaround time to send the requested data back to the users highly depends on the size of the requested data. Yet, it is always a a matter of few seconds for city level requests.

TAREEG is composed of the following four main module (1) The *Data Extraction* module, which runs on a weekly basis to download a 500 GB file from OpenStreetMap, extracts its spatial features, and cleans its noisy data, (2) The *Indexing* module, which runs right after the *Data Extraction* module to index the extracted data using an R-tree-like indexing technique over multiple machines as a means for achieving scalability, (3) The *Query Processor* module, which receives the user requests for obtaining spatial data, converts the request to a range query with the user specified area and a predicate filter for the spatial feature, and finally exploits the R-tree-like index structure to retrieve the requested data in an efficient way, and (4) The *Front-end Visualizer* module, which is a nicely designed web interface that allows users to express their data requests, calls the *Query Processor* module for execution, sends an email to the users with links to the requested data and finally visualize the extracted data on various mapping engines that include Google Maps, Google Hybrid Maps, OpenStreetMap, and Esri Maps.

The efficiency and scalability of TAREEG is mainly due to the fact that it leverages the power of MapReduce-based processing. In particular, TAREEG is powered by SpatialHadoop [6]; an extended MapReduce framework that deals efficiently with spatial data. TAREEG takes advantage of SpatialHadoop and its distributed nature, along with large number of machines, in the following: (1) each machine independently downloads part of the OpenStreetMap data, efficiently parses its own part, and extracts the spatial features from its share of the data, (2) TAREEG uses the indexing capabilities of SpatialHadoop to partition the extracted data on multiple computing nodes (machines) using a spatial index partitioning scheme, and (3) Satisfying the user requests is done through a querying engine that exploits the partitioned data over multiple nodes done by SpatialHadoop. As a result, downloading and indexing the whole OpenStreetMap (a weekly offline process) takes few hours in TAREEG instead of few days if done without the MapReduce-based way in TAREEG. Also, querying the whole dataset to satisfy user requests takes few seconds in TAREEG instead of hours if done in a traditional way.

```
<?xml version="1.0" encoding="UTF-8"?>
 <osm version="0.6"/>
       <node id="1" lat="21.4219827" lon="39.8336534">
       <tag k="traffic" v="light"/>
       </node>
       <node id="2" lat="21.4221823" lon="39.8331833">
              <tag k="highway" v="motorway_junction"/>
       </node>
       . . .
       <way id="6">
              <nd ref="1"/>
              <nd ref="2"/>
              <tag k="highway" v="service"/>
       </way>
       <way id="8">
              <nd ref="4"/>
              <nd ref="5"/>
       <tag k="type" v="multipolygon"/>
       </way>
       . . .
       <relation id="2">
              <member type="relation" ref="1" role="inner"/>
              <member type="way" ref="6" role="inner"/>
       <tag k="highway" v="primary"/>
       </relation>
       . . .
</osm>
```

Figure 1: OpenStreetMap Data Format.

The rest of this paper is organized as follows: Section 2 gives a brief background about OpenStreetMap. Section 3 gives system overview of TAREEG. The main modules of TAREEG, *Data Extraction*, *Indexing*, *Query Processing*, and *Front-end Visualization* are described in Sections 4, 5, 6, and 7, respectively. Experimental evaluation is presented in Section 8. Finally, Section 10 concludes the paper.

2. OVERVIEW OF OPENSTREETMAP

OpenStreetMap (OSM) [24], lunched in 2004, is a collaborative community project to create a free editable map of the world. It is considered as the Wikipedia project for maps, where the community can help in building the maps around the world. OpenStreetMap has over 1.6 million registered users, where around 30% of them have made actual contributions to the maps [21]. As it stands now, OpenStreetMap has a very high accuracy [10] that is comparable to proprietary datasources [10]. OpenStreetMap whole world dataset is free and accessible as an XML 500 GB file called *Planet.osm*, updated on weekly basis.

Figure 1 gives a snippet of the *Planet.osm* XML file. The file consists of the following three primitive data types: (1) *Node*, which is defined as a point in the space associated with a node identifier, latitude and longitude coordinates, (2) *Way*, which represents a line between two nodes, and associated with the way identifer and the two nodes identifers of the two end points of the line. The line could be simply a road segment, part of a boundary of a building, city/country boundary, or part of a lake contour, (3) *Relation*, which represents the relation between *nodes*, *ways*, or even other *relation*, and is used to express polygons. For example, to express the boundaries of a certain lake, the nodes

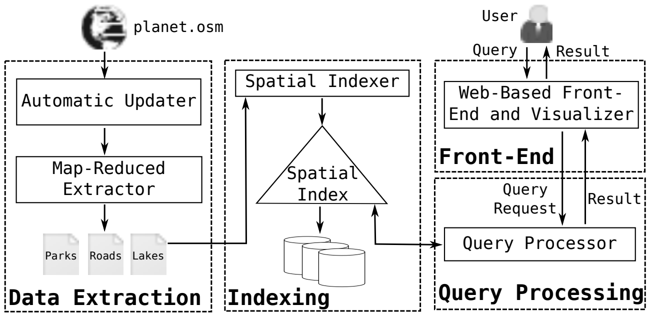


Figure 2: System Overview

need to be defined, then the ways that connect nodes to each other, then a relation that connects the ways together to express the lake boundary. As the dataset is contributed by different volunteers, one lake may be expressed in various relations that could be separated or nested (e.g., a *relation* inside a *relation*), where each relation is composed of either ways or nodes. For example, one lake is composed of two separate relations X and Y. Relation X includes a set of ways that form part of the lake, while relation Y is composed on one way and a relation Z. Then, relation Z is composed of a set of ways that form other part of the lake. Together, relations X, Y, and Z form the whole shape of the lake.

Each of the three primitive data types *node*, *way*, and *relation* is associated with a set of tags. A tag is basically a (*key*, *value*) pair that gives extra information about the primitive data type. Unlike the three basic primitives, tags are not predefined, where volunteers can add new tags and modify existing one. This makes OpenStreetMap data noisy with non-standard tags. In TAREEG, we have experienced several tags that are misinterpreted by volunteers. For example, some volunteers added a skyway as road network bridge and vice versa.

3. SYSTEM OVERVIEW

Figure 2 gives TAREEG system overview. TAREEG is set up as an online web service available 24/7 at [31] to receive user requests of obtaining certain kind of spatial data from anywhere in the world. From inside, TAREEG is composed of four main modules, namely, *Data Extraction*, *Indexing*, *Query Processor*, and *Front-End Visualizer*, briefly discussed below:

Data Extraction. This module runs as a background process that wakes up on a weekly basis to download a 500 GB file *Planet.osm* from OpenStreetMap, extracts its spatial features, and cleans its noisy data. This module also classifies the extracted data into separate files, each represents one kind of spatial data, e.g., parks, road, or lakes. This module faces two main challenges: (1) The large volume of the dataset and (2) the noisy dataset coming from using non-standard tags. Details of the *Data Extraction* module are presented in Section 4.

Indexing. This module runs immediately after the *Data Extraction* module to index the extracted data. Hence, it is also a background process runs on a weekly basis to index the new downloaded data. Given the large size of the extracted data, TAREEG leverages SpatialHadoop [6] to partition and index the extracted data over a set of computing nodes in an R-tree-like way. It is important to note that each type of spatial data (e.g., parks, roads, and lakes) are partitioned and indexed separately. Hence, there will be one index designated for road network data over all available computing nodes, while another completely separate index will be designated for lakes data, and so on. Details of the *Indexing* module are presented in Section 5.

Query Processor. This module receives the user requests for obtaining spatial data, converts the request to a range query with the user specified area and a predicate filter for the spatial feature (e.g., road network, lakes), and finally exploits the R-tree-like index structure to retrieve the requested data in an efficient way. In this module, TAREEG Takes advantage of the fact that the *Indexing* module has partitioned the extracted data into multiple computing nodes to execute its range query over multiple nodes in parallel. Hence, an efficient query processing can be achieved. Details of the *Query Processing* module are presented in Section 6.

Front-End Visualizer. This module is basically a nicely designed web interface that allows users to express their data requests, calls the *Query Processor* module for execution, sends an email to the users with links to the requested data, and finally visualize the extracted data on various mapping engines that include Google Maps, Google Hybrid Maps, OpenStreetMap, and Esri Maps. This module is also responsible on producing the output data in various formats that include Comma Separated Values (CSV), Keyhole Markup Language (KML), Esri Shapefiles, and/or Well-known-Text (WKT). Users can also upload their extracted data any time to just visualize it on TAREEG. This also helps in checking the accuracy of the extracted data as it can be contrasted to a ground truth from Google maps and other mapping services, available in TAREEG. Details of *front-end visualizer* module are presented in Section 7.

4. DATA EXTRACTION

Extracting information from OpenStreetMap is not a trivial task. The whole OpenStreetMap dataset is stored sequentially in one large volume file in a semi-structured XML format. The XML file starts with nodes, then ways, and finally, relations, while tags are nested in each of these data types. A main challenge in extracting information from OpenStreetMap is identifying the annotations (i.e., tags) that imply categorized spatial data.

The *Data Extraction* module in TAREEG basically triggers a script that runs weekly to execute two map-reduce jobs, which take the URL of the compressed *Planet.osm* file as an input, and outputs several categorized files. Each output file contains a homogeneous spatial set of information, e.g., road network data will be stored in one file and lakes in another one, and so on. The first map-reduce job is concerned with spatial features in a form of points and limes, e.g., road network and rivers, while the second map-reduce job is concerned with spatial features in a from of polygons, e.g., lakes, parks, and building. In the rest of this section, we will discuss each map-reduce job separately.

4.1 Line-based Data Extractor

To extract line-based data, e.g., road network and rivers, TAREEG runs a map-reduce job with three main components, namely, *splitter*, *record reader*, and *mapper*, described below.

4.1.1 *Splitter*

The *splitter* component breaks the input file *Planet.osm*

```
Algorithm 1: Line-based Extractor-Record Reader
  Input  : Split $S$
  Output: Position of last byte $Key$ , Primitive OSM
          element $Value$
1  $Record\ reader\ R$;
2  $Element\ e$ ;
3  $End\ byte\ end$ ← $S$ length;
4  $Current\ byte\ position$ ← 0;
5  while $R$ has next OR $e$ has child do
6      $line$ ← $R$ line read by record reader;
7      $position$ ← number of read byte;
8      if $line$ is a root $e$ then
9          Value ← append $line$;
10         if $e$ doesn't have child then
11             Key ← set $position$;
12             return Value;
13     else if $e$ has child then
14         Value ← append $line$;
15         if $line$ is end of root $e$ then
16             return Value;
17     else
18         $e$ doesn't have $child$ And $position > end$
19     return Null;
```

into chunks of size 64 MB (the default block size in HDFS). Since the file is compressed in a *block zip* format, it is possible that each chunk is downloaded and decompressed separately to extract part of the XML file. Such splitting may cause inconsistent XML structures in each split, which is handled later by the *Record Reader*. Thus, splitting the *Planet.osm* will parallelize and distribute the processing load into several map tasks, which is much faster than processing on a single node machine.

4.1.2 Record Reader

The default record reader in Hadoop processes text files line-by-line, so we can not process the data of *Planet.osm* inline. Due to the inconsistent representation of each split, we implemented an XML element reader instead of the default line reader provided by Hadoop framework. The output of this component is an XML *element*, which will be sent later to the mapper. In addition, record reader is responsible on completing inconsistent elements in each split. To elaborate more, if a split has some missing information about one concise type (i.e., spatial feature) such as *buildings*, the record reader will fetch these missing information from the next split. Then the element(s) are passed to the mapper. Therefore, when the next split is being processed by the *Record Reader*, the head of the inconsistent *element* will be ignored, as it has been already processed and sent to the mapper with the previous split.

Algorithm. Algorithm 1 presents the pseudo code of the Record Reader, where we process each split and return a structured XML *elements* as a result. Since processing splits are parallelized and distributed into several mapper task, we need to keep track of the last byte processed by the record reader. First, an XML *Element* that stores the result, and the current end byte position are initialized (lines 1-4). Each processed split consists of a number of lines. We iterate split lines one by one, while tracking the last position of the iterated line (lines 6-7). The last byte position will be returned in the result with the compact XML elements. The iterated line will be appended to the result if it is part of the compact *element* type (i.e., OpenStreetMap data type), otherwise, if the iterated line exceeds the split length and *element* does not have a root XML *element*, then it will be ignored (lines 8-18). While processing a split, there could be one of the following three possibilities:

1. Compacted *element*: Append the line that consists of concise XML *element* e , where e has root of an XML, and it does not have any nested elements e'. In other words, if the iterated line represents a full semi-structured OpenStreetMap *element* (i.e., nodes, ways, relations), then line will be reported to the result and no more lines need to be processed (lines 8-12). For example, if we have node information stored in one line, and this node does not have any associated tags.

2. Semi-compacted *element*: If the split contains the root of an element e, with either closing tag not found in the same split, or the same line. The *Record Reader* will iterate more lines from the same split or next split until closing tag of the root element e found (lines 13-16). For instance, if iterated line begins with OpenStreetMap way information. This way information extends into several lines. *Record Reader* will iterate lines till it finds the end tags of that way.

3. Uncompacted *element*: Ignoring iterated lines if it does not have a root element e. The main reason for ignoring lines is that these lines must be processed with the previous split (line 17). For example, if split begins with subset of a way followed by a relation, then relation lines will be reported in the result. On the other hand, way information will be ignored.

4.1.3 Mapper

This component receives *elements* sent by the *Record Reader* and classifies each element based on its annotation (i.e., tags). Then, it extracts and writes the results into separate files, based on the spatial feature of the extracted data. These spatial features are stored randomly as a set of nodes N, which consists of $N = \{node\_id, longitude, latitude, tags\}$ and another set of extracted spatial features R that consists of $R = \{edge\_id, node1\_id, node2\_id, tags\}$. The main challenge of the mapper is how to deal with extracting spatial features from the noisy dataset. For example, in TAREEG, we have experienced many misinterpreted annotations (i.e., tags) used to describe a downtown skyway, while it is has other tags that are related to road networks. Also in other cases, some tags are used to describe tunnel in a road network, while in a map matching, it shows that it is a tunnel between two buildings. TAREEG mapper handles this kind of noise data by studying carefully each spatial feature tags and filters tags on the fly while data is passed to the mapper by the *Record Reader*.

Algorithm. Algorithm 2 gives the pseudo code of the mapper. Once primitive XML elements received from *Record Reader*, the *Mapper* will process this element. Mapper classifies primitive elements by checking the associated tags with each element. XML element could be either compact element or semi-compact element. Therefore, mapper writes

Algorithm 2: Line-based Extractor-Mapper

```
Input  : Primitive Elements
Output: geographical data File
1 Event reader R;
2 while R has next element e do
3     type flag of classification;
4     if e is start element then
5         CategorizeElementFeature() → Get element
          annotation;
6     if e is end element then
7         File_type ← Write classified data;
8 return geographical data File
```

the classification of the processed element once the whole element is being categorized based on its annotation (i.e., tags).

The Result of the *Mapper* is a set of files, each file represents a relation R that has a specific spatial feature. We use Pig [2] to combine the extracted data in R with its spatial information (i.e., geolocation) from nodes N in a way similar to the join operation in any relational database. The output of the Pig map-reduce program is a spatial set R' that consists of $R' = \{edge\_id,\ node1\_id,\ longitude1,\ latitude1,\ node2\_id,\ longitude2,\ latitude2,\ tags\}$. Each record $r' \in R'$ is now associated with the corresponding geo-location. Yet, R' is neither sparsely nor spatially stored in Hadoop Distributed File System (HDFS).

4.2 Polygon-based Data Extractor

This section discusses the second Map-Reduce job run by the *Data Extraction* module to extract polygon data, e.g., lakes, buildings, and parks.

The line-based extractor described above works well for extracting some datasets such as road networks and rivers. However, other datasets contained in the *Planet.osm* file such as lakes and buildings are better represented in terms of primitive geometric shapes such as polygons and line strings.

Therefore, TAREEG provides this polygon-based extractor, which extracts datasets as objects. Each object is represented by a triple (*id*, *geometry*, *tags*), where *id* is a unique identifier, *geometry* is the geometric shape and *tags* are the associated tags as a list of ⟨ *key*, *value*⟩ pairs. In addition to the large size and non-standard format of *Planet.osm* file, there are additional challenges which face the polygon-based extractor. First, the spatial attribute (i.e., location) is only included in the *nodes* section of the Planet.osm file while other attributes (i.e., ID and tag) are in other sections. Second, depending on the size and complexity of the geometric shape, an object (e.g., lake) might be located in the nodes section (simplest), in the ways section (complex) or in the relations section (very complex). The data from the three sections must be unified and merged in order to generate one output file.

To overcome these challenges, we use Pigeon [7]; a high level MapReduce language for spatial data. It includes standard relational operators such as selection, projection and join and is backed up by OGC-compliant [23] spatial data types and functions. Figure 3 provides a block diagram of how the polygon-based extractor works. Each block indicates an operation applied to the data. In step 1, the XML

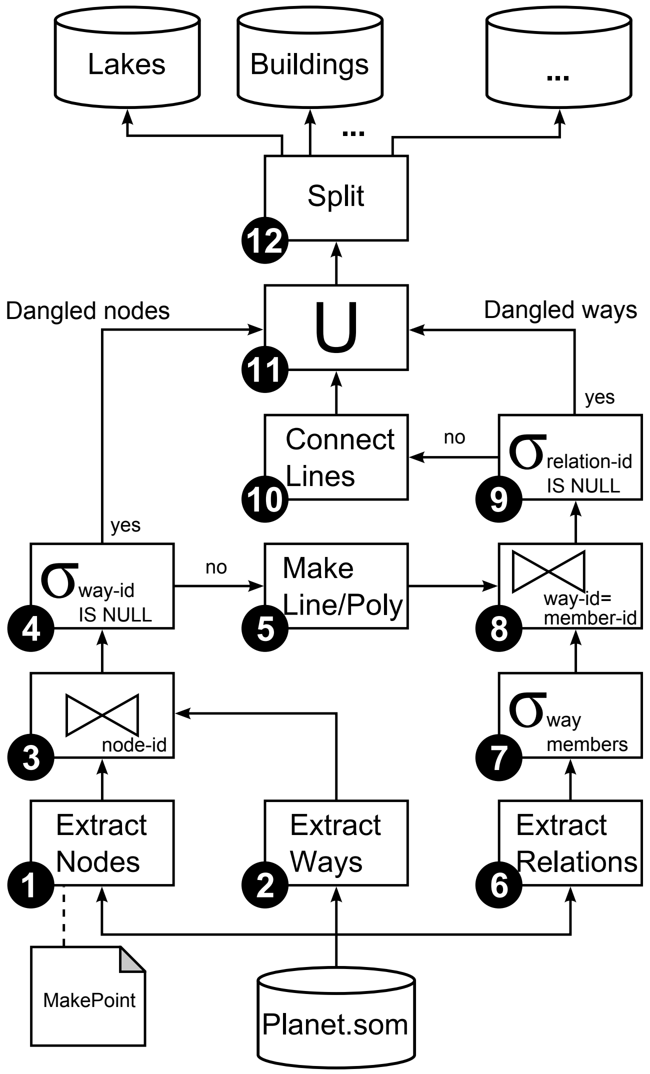


Figure 3: Block diagram of the polygon-based data extractor

elements in the *nodes* section of the *Planet.osm* file are read and parsed to extract nodes information, node ID, latitude, longitude, and tags. In this step, the `MakePoint` function in Pigeon is called to combine the numeric values of latitude and longitude into a `Point` object. In step 2, the same thing is done with the *ways* section and the information extracted are way ID, list of node IDs in this way, and tags. In step 3 the extracted ways are joined with the nodes on the node ID column to add the spatial location of each point. In this step, we use an *outer join* which causes all nodes to be included in the output even if they do not match with any extracted way. Those nodes are called *dangled nodes* and they represent an object by themselves without being assigned to a way such as mountains, traffic lights, and bus stops. Step 4 splits the join output to separate dangled nodes from other nodes. Dangled nodes are later written to output while nodes assigned to ways are further processed in the next step. Step 5 calls the Pigeon function `Make-LinePolygon`, which groups points by way ID and generates a polygon in case points form a closed loop or line if they do not.

In step 6, relations are extracted from the *Planet.osm* file, where each relation contains the attributes relation

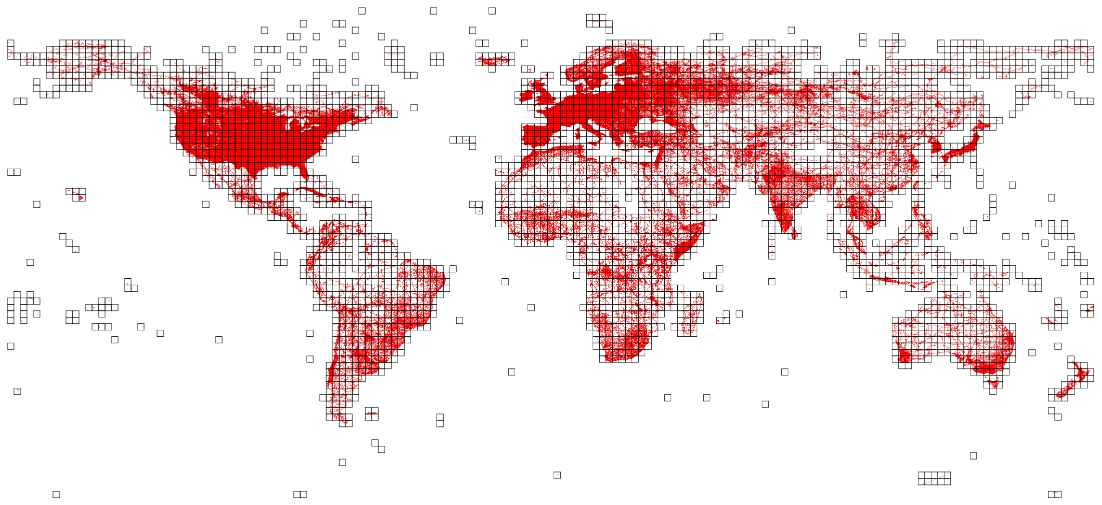


Figure 4: Grid Files Spatial Data Partitioned

ID, members, and tags. Members is a list of tuples, $\langle member-ID, member-type \rangle$ where member-type can take one of the values *point*, *way* or *relation*. Step 7 selects members of type *way* only as these are the ones needed in next steps. Step 8 performs an outer join between relations and ways on 'way ID = member ID' to add the spatial dimension. As done in step 4, the result is filtered based on relation ID to split *dangled ways* from ways assigned to relations. Dangled ways represent objects that are totally defined in the ways section and do not need to be combined with other ways. Step 10 calls a Pigeon function named `Connect` which connects a set of lines and polygons to form a more complex shape based on the following cases: (1) If two line strings share one end point, they are connected together to form one longer line string, (2) If two line strings share two end points, they are connected together to form a polygon, (3) If two polygons are combined together and one polygon is contained in the other polygon, the inner polygon is added as a *hole* inside the first one, and (4) In any other case, the two shapes are combined together to form a *Geometry Collection* standard data type as defined by the OGC standard [23].

In step 11, the output of the `Connect` operation is *unioned* with dangled nodes and dangled ways to produce the final output that contains all objects found in the *Planet.osm* file. The output has a unified schema (ID, Geometry, Tags), where the geometry can be a point, line string, polygon, or a geometry collection based on how it is generated. Finally, step 12 splits those objects based on tags to the datasets we are interested in such as lakes and buildings where each tag results in a separate file.

5. INDEXING

This section describes the *Indexing* module in TAREEG, which is triggered immediately after the execution of the *Data Extraction* module. The files generated by the *Data Extraction* module are heap files, which are not organized in any specific order. This means that if a range query is executed to return data from a specific region, the query would have to scan the whole file to retrieve the result. This will considerably slow down the system, especially for larger datasets, such as road networks. To speed up the processing of extracted datasets, the *Indexing* module in TAREEG builds spatial indexes for each dataset that efficiently support range queries. One method to index datasets is to store each input file as a relation in a Spatial Database Management System (DBMS), e.g., PostGIS [27]. Then, we can build a spatial index, e.g., R-tree [9] on that relation. Once the relation is indexed, range queries can be expressed in SQL and executed efficiently inside the spatial DBMS. Unfortunately, such technique rendered infeasible due to the long time consumed to build the indexes.

In TAREEG, to be able to construct the indexes efficiently, we use SpatialHadoop [6] to take advantage of its spatial indexing techniques. SpatialHadoop supports different types of spatial indexes including Grid file [22], R-tree [9], and R+-tree [29]. Once datasets are extracted from the *Planet.osm* file, we use the SpatialHadoop '`index`' command on each file separately to build an index for it. The partition size is set to 64 MB (default for Hadoop). Each index is stored as one *master* file that stores the boundaries of each partition, and multiple data files, each with its corresponding set of spatial data overlapping with the partition boundaries. We briefly discuss below each index structure and its suitability to OpenStreetMap dataset.

Grid index. In a grid index, data are partitioned into a uniform grid file structure, where it is distributed based on the geolocation of spatial features into equal spatial dimension size. In other words, data are not equally partitioned by block size. Figure 4 gives the grid index of road network data. Obviously, data are not uniformally distributed based by block size between partitions. For instance, in a dense area like Houston Texas, the partition might exceed the blocking size. On the other hand, in rural areas the partition size is larger then the data size, due to the lack of the data in that spatial region. This results in a huge waste of storage as most partitions are under utilized. This also affects the query processing as more partitions need to be visited even for a small range query.

R-tree index. In the R-tree index, we first partition the data in a balanced tree. Each partition represents a minimum boundary rectangle (MBR) that includes a block of data of size 64 MB. The partition boundaries are decided based on the data distribution, and may be overlapped. Data items that overlap with more than one partition are stored once.

R+tree index. Similar to the case of R-tree index, data are partitioned in a balanced tree with each partition rep-

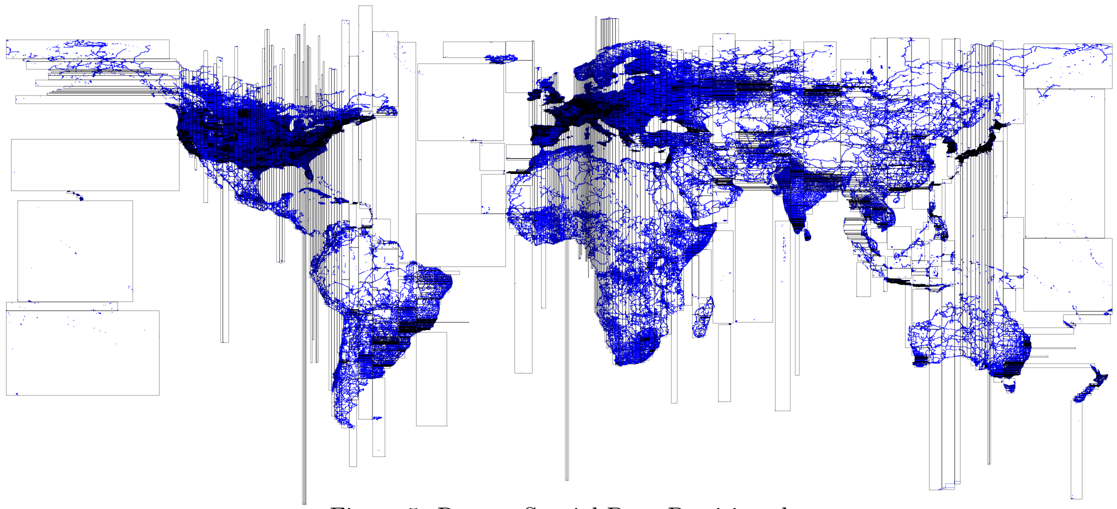


Figure 5: R+tree Spatial Data Partitioned

resenting a minimum boundary rectangle of a set of points of size 64 MB. The only difference is that the partitions in the R+-tree are not overlapped. Hence, one object may be stored with all the partitions it overlaps with.

Figure 5 gives the main reason behind the efficiency of spatial data indexing and querying in TAREEG. The figure shows an R+-tree partitioning scheme for the whole road network file. All road networks are depicted in blue lines, while the black rectangles indicate partition boundaries. While some road segments cross multiple partitions, partition boundaries remain disjoint due to the properties of the R+-tree. As each partition is sufficient for ONLY 64 MB worth of data, we can see that dense areas (e.g., Europe) are contained in very small partitions, while sparse areas (e.g., oceans) are contained in very large partitions. One way to look at this figure is that this is the way that Spatial-Hadoop divides a dataset of 500 GB into small chunks, each of 64 MB. Recall that in Hadoop, this 500 GB file will be divided into chunks of 64 MB as sequential heap file, which definitely does not fit spatial operations.

6. QUERY PROCESSING

Queries sent to TAREEG are basically range queries that request extracting a certain type of spatial data (e.g., road networks, lakes, buildings, borders) within a certain area of interest, presented as a rectangular area. When a range query is sent to TAREEG, it first executes that query on the master file to retrieve partitions that overlap with the range query. For each matching partition, the records in the corresponding data file are compared to the query range and all matching records are stored as part of the answer. If matching records lie in more than one data partition, TAREEG exploits the parallelism of computing nodes to execute the query simultaneously on different partitions. Since in R+-tree some records overlapping multiple partitions are replicated, a post processing duplicate avoidance step is executed to ensure the correctness of the answer as described in [6].

7. SYSTEM FRONT-END

The system front-end provides a set of tools for users to extract and visualize their requested spatial data. As TAREEG is deployed as 24/7 online web service, it is designed for simplicity, where user can extract, download, and visualize the spatial features. The system front-end of TAREEG consists of two main modules: (1) *Data extraction request*, where users can submit extraction requests through nicely designed web interface, by selecting any arbitrary region of world map and the type of spatial data they want to extract. (2) *Spatial data download and visualization*, where users can download and visualize exported data on either OpenStreetMap, Google Maps, Google Hybrid Maps (i.e., hybrid satellite images with labels), and Esri Maps (i.e National Geographic map).

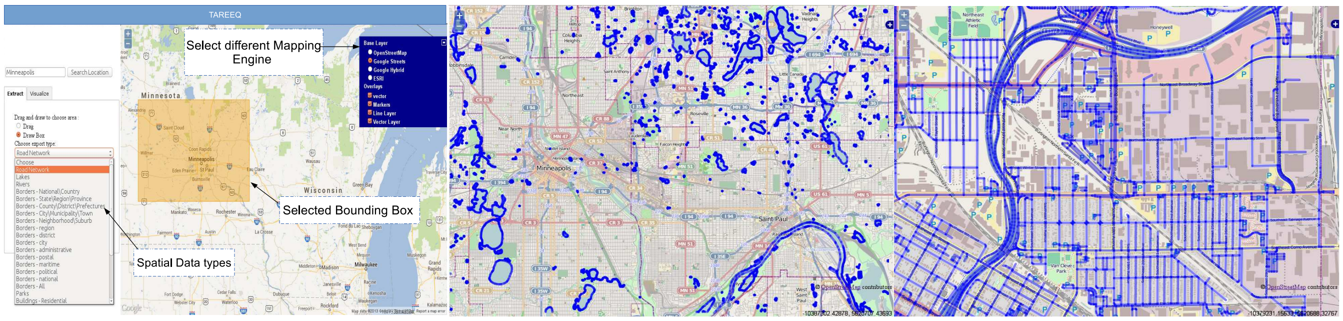
7.1 Spatial Data Extraction Request

Initially, users interact with TAREEG system by requesting different spatial data types as shown in Figure 6a, following few steps: (1) the user selects any arbitrary region from the world map, either by zooming and draging the map to the geographical area of interest, or by searching the area of interest using the search bar, (2) the user specifies a boundary rectangle of the selected region, (3) the user provides an email address, in which TAREEG will send an email when the request is satisfied with links to download the requested data, and (4) the user submits the request by hitting the 'Extract button'.

Once a request is received, TAREEG processes this request by querying the requested spatial data from the back-end of the system, and notifies the requester through email once the data is available. It is important to note here that all requests to TAREEG are satisfied from its own local data, as there is no need to contact OpenStreetMap servers, and hence requests are usually satisfied within a few seconds. Once the requested data is available, the user will receive an email from TAREEG with hyperlinks to download the requested data. The time of processing the request depends on the size of the selected region and complexity of the requested dataset.

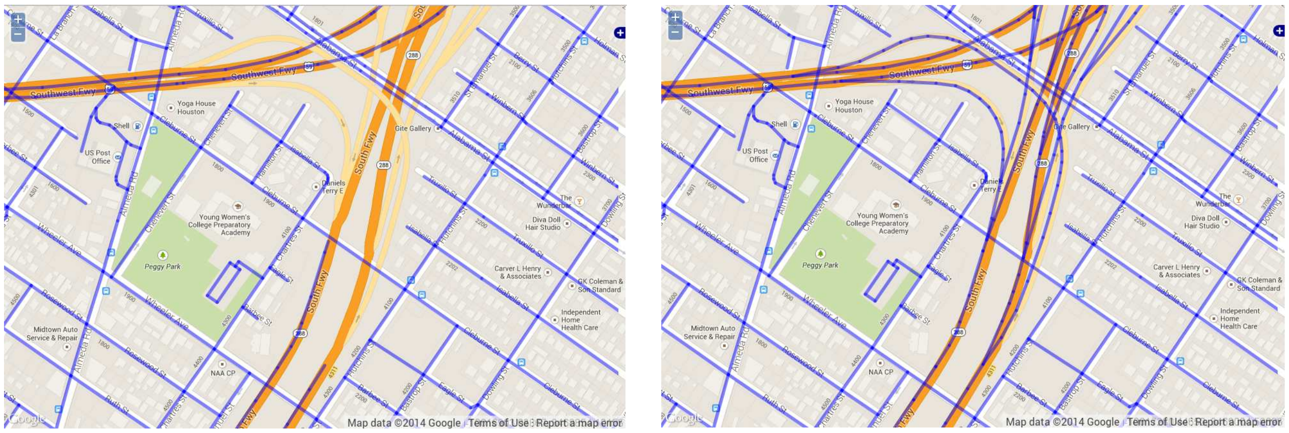
7.2 Spatial Data Download & Visualization

TAREEG users receive their extracted data in four common standard formats, a Comma Separated Values (CSV) files, a Keyhole Markup Language (KML), Esri shape files, and Well-Know Text (WKT) formats. TAREEG only supports visualizing (CSV) format, where others can be already visualized through various applications (e.g., QGIS, Google Earth, and Esri ArcGIS). The CSV format has the following



(a) TAREEG Web Interface (b) Extracted lakes (c) Extracted Road Networks

Figure 6: TAREEG System Web Interface



(a) Straightforward road network extraction (b) TAREEG road network extraction

Figure 7: Accuracy of TAREEG road network

structure:

```
Node < Node_id , Latitude , Longitude >
Edge <Edge_id , Start_Node_id ,  End_Node_id ,  [Tags] >
```

Well Known Text (WKT) serialization is standardized by OGC and provides a textual representation for geometric objects with the following structure:

```
Triple ( id , geometry , tags )
Point ( Latitude , Longitude )
Line (Point_1, Point_2, Point_3, ..., Point_n)
Polygon (Point_1, Point_2, Point_3, ..., Point_n)
```

Figures 6b and 6c give two examples of the visualization tool of TAREEG, where an extracted set of lakes and road networks are visualized, respectively. TAREEG visualizer shows the spatial data on the map by uploading the CSV files, that were sent to the user as hyperlinks in an email. TAREEG visualizer supports four different mapping engines, where user can experience the easiness of switching the gear between these mapping engines.

8. SYSTEM EVALUATION

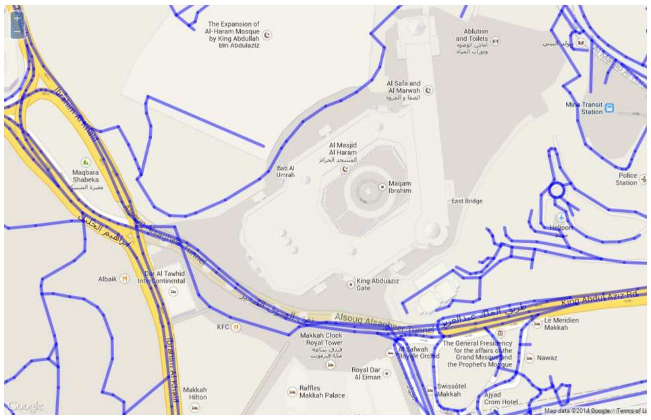
This section evaluates TAREEG in terms of its accuracy in extracting spatial data from OpenStreetMap as well as its performance when relying on MapReduce-based implementation. All experiments in this section was run on Amazon web services with 20 machines.

9. ACCURACY

In this section, we evaluate the accuracy of the *Data Extraction* module in TAREEG. We compare TAREEG against a straightforward data extraction that is just based on the existing tags of OpenStreetMap. Figure 7 shows such comparison for a subarea in Houston, TX. We used Google Maps in the background as the ground truth, while blue lines represent the extracted road network using a straightforward OpenStreetMap tag extraction (Figure 7a) or using TAREEG road network extraction (Figure 7b).

It is clear to see that the quality of TAREEG extraction is higher than the straightforward approach. There are many road segments that are not recognized by the straightforward extraction. Most importantly, the set of highway segments and exit roads in the top middle of the map. Yet, all these segments and exits are all well extracted by TAREEG. The main reason here is that these parts are annotated with different tags from the rest of the roads. We have learned this in TAREEG, and cleaned the segments, when doing the extraction, and hence we did achieve much better accuracy. Though not show in pictures, but we have witnessed many cases where the straightforward extraction mistakenly extracts pedestrian sidewalks and skyways as road segments, while TAREEG extraction makes it always right by efficiently and smartly comparing several associated tags together to ensure the quality of road extraction.

Figures 8a and 8b give two examples of the accuracy of TAREEG when extracting road network for Makkah, Saudi Arabia, and Istanbul, Turkey, respectively. We are not com-


(a) Makkah TAREEG road network

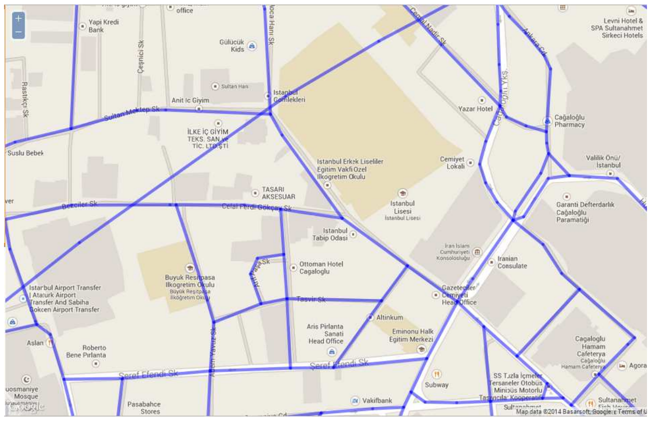

(b) Istanbul TAREEG road network

Figure 8: Accuracy of TAREEG road network

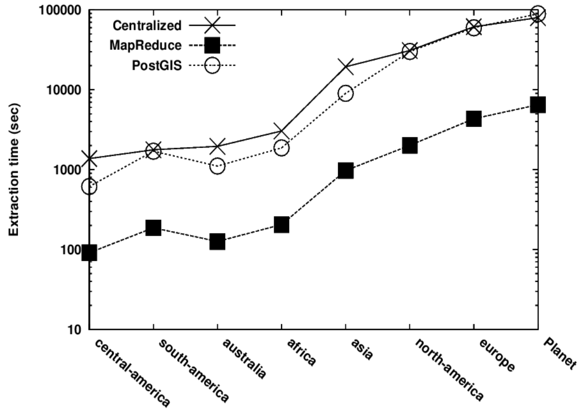


Figure 9: Extraction time

paring here against the straightforward extraction, as we will end up having similar accuracy as that of Figure 7. The purpose of this experiment is just to shows the quality of TAREEG road extraction in various cities around the world.

9.1 Performance

This section discusses the performance of index building, data extraction, and querying time in TAREEG.

9.1.1 *Data Extraction*

Figure 9 compares the extraction time of TAREEG (using MapReduce) with the extraction time of a single-machine process (centralized) and using a traditional spatial database (PostGIS [27]), for different data sets with different sizes starting from a small data set with only Central America to the whole data set of the planet. In all cases, TAREEG is consistently superior than other techniques with at least one order of magnitude better performance. For example, for the whole planet, it takes about two hours from TAREEG to do all the extraction in high accuracy, however, it takes more than a whole day from PostGIS and centralized processing to extract the same data. In fact, in many cases, we had to kill and rerun the experiment for PostGIS, as in sometimes the experiment goes forever without finishing.

9.1.2 *Index Building*

Table 10 compares the usage of three indexing techniques, Grid file, R-tree, and R+-tree for spatial data indexing in TAREEG. The three indexing structures have similar build-

-	Grid File	R-tree	R+tree
Build index (sec)	674	601	690
# number of partition	3559	10654	10583
$Avg\_Partition_{size}$ (MB)	38.5	12.8	12.9

Figure 10: Index Evaluation of Road Network

ing time in the order of 600 seconds. However, it is clear that both R-tree and R+-tree have higher number of partitions than that of the grid structure. This goes in favor for the tree indexing for two main reasons: (1) Higher number of partitions means that the incoming query can be better done in parallel over the 20 machines we have, and (2) Whenever reading from a partition, we only read a small set of data that is relevant to the query we have. This is in contrast to large-sized partitions that result in reading a lot of redundant data.

In TAREEG, we opt for using R+-tree index as it has a similar performance to R-tree, yet, it ensure that partitions are not overlapping, which is a property we need when dealing with MapReduce environment. Meanwhile, R+-tree gives way much performance better query processing than that of a gird structure.

10. CONCLUSION

This paper presents TAREEG; a web-service that makes real spatial data, from anywhere in the world, available at the fingertips of every researcher or individual. TAREEG gets all its data by leveraging the richness of OpenStreetMap dataset; the most comprehensive available spatial data of the world. Yet, it is still challenging to obtain OpenStreetMap data due to the size limitations, special data format, and the noisy nature of spatial data. TAREEG employs MapReduce-based techniques to make it efficient and easy to extract OpenStreetMap data in a standard form with minimal effort through four main components. *Data Extraction.* is responsible for extracting spatial data and clean the noise data. *Data Indexing.* where each spatial feature is indexed spatially with high efficiency. *Query processor.* that receives the user requests for obtaining spatial data, and converts the request to a range query with the user specified area and a predicate filter for the spatial feature (e.g., road network, lakes). and Finally, *visualization* a nicely designed web interface that allows users to express their data requests. Experimental results show that TAREEG is highly accurate and efficient in terms of the time taken to satisfy data extraction of user requests.

11. REFERENCES

[1] L. Alarabi, A. Eldawy, R. Alghamdi, and M. F. Mokbel. TAREEG: A MapReduce-Based Web Service for Extracting Spatial Data from OpenStreetMap (System Demonstration). In *SIGMOD*, pages 897–900, Snowbird, UT, June 2014.

[2] Apache pig. `http://pig.apache.org/`.

[3] Z. Chen, Y. Liy, R. C.-W. Wong, J. Xiong, G. Mai, and C. Long. Efficient algorithms for optimal location queries in road networks. In *SIGMOD*, 2014.

[4] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD*, pages 591–602, 2009.

[5] K. Deng, X. Zhou, and H. T. Shen. Multi-source skyline query processing in road networks. In *ICDE*, pages 796–805, 2007.

[6] A. Eldawy and M. F. Mokbel. A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data (System Demo). In *VLDB*, Riva del Garda, Italy, Aug. 2013.

[7] A. Eldawy and M. F. Mokbel. Pigeon: A spatial mapreduce language. In *ICDE*, pages 1242–1245, 2014.

[8] Geo fabrik. `http://download.geofabrik.de/`.

[9] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, 1984.

[10] M. Haklay. How good is Volunteered Geographical Information? A Comparative Study of OpenStreetMap and Ordnance Survey Datasets. *Environment and Planning B: Planning and Design*, 37(4):682–703, 2010.

[11] L. Hu, Y. Jing, W.-S. Ku, and C. Shahabi. Enforcing k nearest neighbor query integrity on road networks. In *SIGSPATIAL GIS*, pages 422–425, 2012.

[12] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *SIGSPATIAL GIS*, pages 1–8, 2003.

[13] Y. Jing, L. Hu, W.-S. Ku, and C. Shahabi. Authentication of k nearest neighbor query on road networks. *TKDE*, 26(6):1494–1506, 2014.

[14] C.-C. Lee, Y.-H. Wu, and A. L. P. Chen. Continuous evaluation of fastest path queries on road networks. In *SSTD*, pages 20–37, 2007.

[15] G. Li, Y. Li, J. Li, L. Shu, and F. Yang. Continuous reverse k nearest neighbor monitoring on moving objects in road networks. *Information Systems*, 35(8):860–883, 2010.

[16] S. Luo, Y. Luo, S. Zhou, G. Cong, and J. Guan. DISKs: a system for distributed spatial group keyword search on road networks. *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 5(12):1966–1969, 2012.

[17] X. Ma, S. Shekhar, and H. Xiong. Multi-type nearest neighbor queries in road networks with time window constraints. In *SIGSPATIAL GIS*, pages 484–487, 2009.

[18] P. Mooney and P. Corcoran. Characteristics of heavily edited objects in openstreetmap. *Future Internet*, 2012.

[19] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, pages 43–54, 2006.

[20] Navteq. `http://here.com/navteq-redirect/?lang=en-GB`.

[21] P. Neis and A. Zipf. Analyzing the Contributor Activity of a Volunteered Geographic Information Project Ů The Case of OpenStreetMap. *ISPRS International Journal of Geo-Information*, 1(2):146–165, 2012.

[22] J. Nievergelt, H. Hinterberger, and K. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS*, 9(1):38–71, 1984.

[23] Open geospatial consortium (ogc). `http://www.opengeospatial.org/`.

[24] Openstreetmap. `http://www.openstreetmap.org/export`.

[25] Osm benchmarks, june 2012. `http://wiki.openstreetmap.org/wiki/Osm2pgsql/benchmarks`.

[26] Osm tools, june 2012. `http://wiki.openstreetmap.org/wiki/Osmosis`.

[27] PostGIS, 2007. http://postgis.refractions.net/.

[28] M. N. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *PVLDB*, 4(2):69–80, 2010.

[29] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*, pages 507–518, 1987.

[30] S. Shang, B. Yuan, K. Deng, K. Xie, and X. Zhou. Finding the Most Accessible Locations: Reverse Path Nearest Neighbor Query in Road Networks. In *SIGSPATIAL GIS*, pages 181–190, 2011.

[31] TAREEG. `www.tareeg.org`.

[32] J. R. Thomsen, M. L. Yiu, and C. S. Jensen. Effective caching of shortest paths for location-based services. In *SIGMOD*, 2012.

[33] Y. Tian, K. C. K. Lee, and W.-C. Lee. Finding skyline paths in road networks. In *SIGSPATIAL GIS*, pages 444–447, 2009.

[34] TIGER files. `http://www.census.gov/geo/www/tiger/`.

[35] S. Vanhove and V. Fack. An effective heuristic for computing many shortest path alternatives in road networks. *International Journal of Geographical Information Science*, 26(6):1031–1050, 2012.

[36] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, 2012.

[37] M. L. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks. *TKDE*, 17(6):820–833, 2005.

[38] W. Zeng and R. Church. Finding shortest paths on real road networks: The case for a*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009.

[39] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: Towards bridging theory and practice. In *SIGMOD*, pages 857–868, 2013.

[40] L. Zhu, Y. Jing, W. Sun, D. Mao, and P. Liu. Voronoi-based aggregate nearest neighbor query processing in road networks. In *SIGSPATIAL GIS*, pages 518–521, 2010.