OSMX: Spark-based Geospatial Data Extractor from OpenStreetMap*

Samriddhi Singla
ssing068@ucr.edu
Computer Science and Engineering
University of California, Riverside
Riverside, California

Yaming Zhang
yzhan737@ucr.edu
Tencent Technology Co.

Ahmed Eldawy
eldawy@ucr.edu
Computer Science and Engineering
University of California, Riverside
Riverside, California

ABSTRACT

With the rising amount of publicly available data, data-driven modeling is becoming increasingly popular. Geospatial data is one of the most important facets that can be combined with virtually all data science real-world applications. However, there is a lack of customized geospatial data that can be used in various data science applications from different domains, e.g., hydrology, political science, climatology, and agriculture. This paper introduces a Spark-based extractor that can extract rich geospatial datasets from OpenStreetMap (OSM). OSM hosts crowd-sourced geospatial data that represent a variety of natural and human-made features, e.g., lakes, buildings, and roads. The size of this data is extremely huge and requires complex processing before being ready to use in data science. The proposed extractor runs on Apache SparkSQL which allows it to scale to the Planet.osm file which spans the entire world. In addition to the extractor, we make the data available in various standard formats, e.g., GeoJSON, CSV, KML, and Shapefile. Furthermore, we host these datasets on UCR-Star which allows users to visually explore these datasets and download any subset of the data for any geospatial region.

CCS CONCEPTS

• **Information systems** → *Database query processing*.

KEYWORDS

OSM Data, Data Extraction, OpenStreetMap, Big Data, Spatial Data

ACM Reference Format:
Samriddhi Singla, Yaming Zhang, and Ahmed Eldawy. 2022. OSMX: Spark-based Geospatial Data Extractor from OpenStreetMap. In *The 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '22), November 1–4, 2022, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3557915.3560954

*This work is supported in part by the National Science Foundation (NSF) under grant IIS-2046236 and by Agriculture and Food Research Initiative Competitive grant No. 2020-69012-31914 from the USDA National Institute of Food and Agriculture

SIGSPATIAL '22, November 1–4, 2022, Seattle, WA, USA

ACM ISBN 978-1-4503-9529-8/22/11.
https://doi.org/10.1145/3557915.3560954

1 INTRODUCTION

The recent evolution of technology has led to a significant increase in the amount of collected data. This has shifted most scientific domains to rely more on data science and data-driven models [9, 11–14]. The success of these applications heavily depends on the availability of suitable data that contains the right set of features. Geospatial features are among the most popular features that are needed in virtually all data science applications. However, having the right set of geospatial features in the region of interest is still a challenge that limits many data science applications.

OpenStreetMap (OSM) is an open system that relies on citizens worldwide to collect geospatial data including roads, administrative boundaries, parks, and lakes. However, despite being publicly available, extracting useful data that can be used in data science is still an active challenge due to the complexity of this data. First, OSM data is available as one large file, called Planet.osm, in Protocol Buffers Format (PBF) which is not readily accessible to big-data systems such as Spark. Second, OSM data is stored in a canonical form that consists of three major sections, nodes, ways, and relations. A single geographical feature, e.g., a lake, is split among those sections and complex processing is needed to produce features in a standard geospatial format used by data scientists, e.g., GeoJSON. Third, the scale of the data is huge. The standard format of the file is a single XML file that is nearly one terabyte in size. Other formats, such as the compressed PBF file are nearly 60GB and are not easy to parse or process.

This paper presents OSMX, the first Spark-based end-to-end solution for extracting ready-to-use data from the OSM PBF file. The proposed extractor has three unique advantages. 1) It uses Spark, the most advanced distributed query engine to process large amounts of data. 2) It works directly on the compressed PBF file which minimizes the download time and storage requirements. 3) We make the source code and extracted data available for free and we host the datasets on UCR-Star [8] which allows users to explore the data and download arbitrary subsets of it.

The proposed extractor is built using Beast [3], a Spark extension for spatial data exploration. First, it adds a distributed reader for the Planet.osm file and uses it to read the three major sections of the file, namely, nodes, ways, and relations. Then, it uses SparkSQL to join these sections efficiently and produce the desired features. The SQL queries can be easily modified by the user to limit them to a specific geographical region or a subset of features, e.g., buildings or roads. The produced features can then be stored in several standard geospatial formats including GeoJSON and Shapefile. Unlike existing work, the proposed extractor can generate data for any

geographical region, and scientists are no longer tied to the limited publicly available data.

2 RELATED WORK

Although OpenStreetMap (OSM) is one of the biggest sources of publicly available geospatial data, there are very few solutions aimed at extracting data from it. One of the first solutions included TAREEG [1], a MapReduce extractor that could extract datasets from the XML file format of OSM data. It was based on Spatial-Hadoop [6] and Pigeon [5] but it is no longer maintained and did not scale well due to the limitations of Hadoop. The latest extract from TAREEG dates back to 2015. Other efforts included [4] and [2], which aimed at integrating OSM data with other sources of Linked Geo Open Data (LGOD). While [4] converted the OSM data format to an RDF (Resource Description Framework) data model to create a large spatial knowledge base, [2] proposed a framework to convert sources of Linked Geo Open Data (LGOD) into the OSM data format and query the combined dataset. However, [4] has not updated its database since 2015 and [2] is limited to only integrating points of interest and cannot work with relations (roads, bus routes, etc.).

Photon [10] is another open-source system that can extract datasets from OSM. It has built its own worldwide search index (53 GB) that the users can query, however, it might take days for this index [1] to be ingested by Photon before the users can query it. There are also some open-source tools that are available for parsing XML or PBF files but they run on a single machine and do not scale well, e.g., Osm2pgsql and Osmium. There are also services like GeoFabrik [7] that provide data extracts from OSM for predefined regions and datasets. A limited version of these data extracts is available for free and the full version is priced according to its size.

In contrast to these systems, the proposed extractor is implemented in Spark and can process the whole OSM planet file without a separate indexing step and makes the datasets available for public use in various formats. It is not limited to points of interest and can extract datasets such as buildings, lakes, roads, etc. The datasets extracted by the proposed system are free to download without any limitations.

3 GEOMETRY DATA EXTRACTION

To implement the proposed extractor, we defined several user-defined functions (UDFs) such as ST_Connect, ST_CreatePoint, ST_CreateLinePolygon, and HasTag that can be run with SparkSQL. We use these functions to generate the geometries that represent physical objects in OSM and store it in a DataFrame. Below, we describe the process of extracting the desired geometry features from the OSM PBF file using these functions.

3.1 Spark PBF Reader

The first step is to parse and read the records from the PBF file. We create a custom input format that works as follows. First, it splits the PBF file into equi-sized blocks of size 128 MB. Notice that this is just a logical splitting that defines the ranges in the file that will be read in parallel but the file is not physically split. This step runs on the driver machine and it takes only a fraction of a second since the input is not actually read from the disk.

[1]As mentioned on their Github page

After that, the Spark executors process these splits in parallel to extract records inside each split. The challenge in this step is to ensure that the entire file is processed correctly and that each record is read exactly once while the executors run in parallel. This is not a trivial task since PrimitiveGroups in the PBF file might span two splits. To ensure that the file is processed correctly, we define an *anchor point* as the first byte of each PrimitiveGroup. The split that contains the anchor point of the PrimitiveGroup becomes responsible for reading it. This ensures that exactly one executor will read each PrimitiveGroup. To implement that, each executor starts reading from the beginning of its assigned split and skips until it finds the first *special marker* that marks the beginning of the PrimitiveGroup. If it falls within the split boundary, it reads the entire PrimitiveGroup even if it spans to the next split. This process is highly scalable since it does not require any communication between the executor nodes.

3.2 Read OSM Entities

In this step, we read the three main sections of the input file that contain nodes, ways, and relations. To do that, we first utilize our PBF reader with Spark's `newAPIHadoopFile` function to read all entities in the input file as a Resilient Distributed Dataset (RDD) in Spark. Then, we run three filter operations to split that RDD into three groups for nodes, ways, and relations. Finally, we convert each group into a DataFrame with the correct schema to be able to use SparkSQL for the extraction process. The conversion to DataFrame is done by creating three classes, one for each data type.

3.3 Extract Nodes and POIs

The first two datasets we extract are *nodes* and points-of-interest (POIs). To extract *nodes*, we apply the `ST_CreatePoint` function on the DataFrame for *nodes* to combine the two coordinates, longitude and latitude, into a *Point* geometry and write it to the output. The following SQL function illustrates this logic[2].

```
SELECT id, ST_CreatePoint(longitude, latitude) AS geom
FROM nodes
```

Similarly, POIs are created by adding a filter for a set of tags that identify popular POIs. The following code snippet shows part of the WHERE clause that is appended to the previous SQL query.

```
... WHERE HasTag(tagsMap, "name,amenity,cuisine,...", "")
```

3.4 Extract Roads

The next dataset to extract is the roads dataset. Roads are extracted from the DataFrames for *nodes* and *ways*. Since the *ways* DataFrame contains only node IDs without their location, we need to join the *nodes* and *ways* DataFrames to get the node location. To accomplish that, we normalize the *ways* DataFrame by repeating each *way* for each *nodeId* it contains. The SparkSQL function `explodepos` accomplishes this requirement while maintaining the position of each *nodeId* to allow us to construct the geometry in the right order. The following code snippet illustrates this step.

[2]For brevity, we only include the ID for each extracted entity in SQL queries but the actual extractor includes all other attributes such as tagsMap and timestamp

```
WITH explodedWays AS (
  SELECT id, posexplode(nodeIds) AS (pos, nodeId)
  FROM ways
)
```

Then, we join the resulting DataFrame with *nodes* to bring the location back into the dataset using the following SQL query.

```
SELECT array(longitude, latitude) AS node_location,
  longitude, latitude, way.id, way.nodeId, pos
FROM nodes AS node LEFT OUTER JOIN explodedWays AS way
WHERE node.id = way.nodeId
```

After that, we bring together all the *nodes* for each *way* and their locations into one record while maintaining their correct position. This step uses the windowing functionality in SQL to group by *wayId* and sort by *pos* within each group. The following SQL query illustrates this step.

```
SELECT longitude, latitude, id
  collect_list(nodeId) OVER w AS nodeIds,
  flatten(collect_list(node_location)) OVER w AS node_locations,
  first(nodeId) OVER w AS first_nodeId,
  last(nodeId) OVER w AS last_nodeId
FROM joinWayDf
WINDOW w AS (PARTITION BY id ORDER BY pos)
```

Finally, we use the function `ST_CreateLinePolygon` to create a geometry for each *way* and filter them by tags that represent roads as illustrated by the following SQL query.

```
SELECT id, ST_CreateLinePolygon(nodeIds, node_locations)
FROM ways
WHERE HasTag(tagsMap, "highway,junction,...",
  "yes,street,...")
```

3.5 Extract Relations

The last step is to extract relations by connecting multiple *ways* that can represent bigger objects, e.g., a country boundary with more than 2,000 *nodes*. Similar to what we did with *ways*, we first normalize the *relations* DataFrame by repeating the *way* information for each relation. We use the explode function as illustrated in the following SQL query.

```
WITH explodedRelations AS (
  SELECT explode(arrays_zip(memberIds,
      entityTypes, memberRoles)) AS exp, *
  FROM relations
)
SELECT id, wayId, exp.entityTypes,
    first_nodeId, last_nodeId, geometry
FROM explodedRelations RIGHT OUTER JOIN wayDf
ON exp.memberIds = wayId AND exp.entityTypes = "Way"
```

Notice that we only keep relation members of type "Way" since these are the ones we are interested in connecting in this step. We also use the right outer join so that *ways* that do not belong to any *relations* will be in the result. They represent top-level *ways* that should be written to the output as-is.

After that, we use the `ST_Connect` function to connect all *way* geometries that belong to the same *relation* as illustrated by the following SQL query.

```
WITH groupedRelations AS (
  SELECT id, collect_list(first_nodeId) AS first_nodeIds,
      collect_list(last_nodeId) AS last_nodeIds,
      collect_list(geometry) AS geometry
  FROM joinRelationWay456
  WHERE id IS NOT NULL
  GROUP BY id
)
SELECT id, ST_Connect(first_nodeIds, last_nodeIds,
    geometry) as geometry
FROM groupedRelations
```

```
All Objects
 ├── id: long (nullable = true)
 ├── version: integer (nullable = false)
 ├── timestamp: long (nullable = false)
 ├── changeSetId: long (nullable = false)
 ├── uid: integer (nullable = false)
 ├── uname: string (nullable = true)
 ├── tagsMap: map (nullable = true)
 │    ├── key: string
 │    └── value: string (valueContainsNull = true)
 └── geometry: geometry (nullable = false)
```

Figure 1: Schema for the produced data

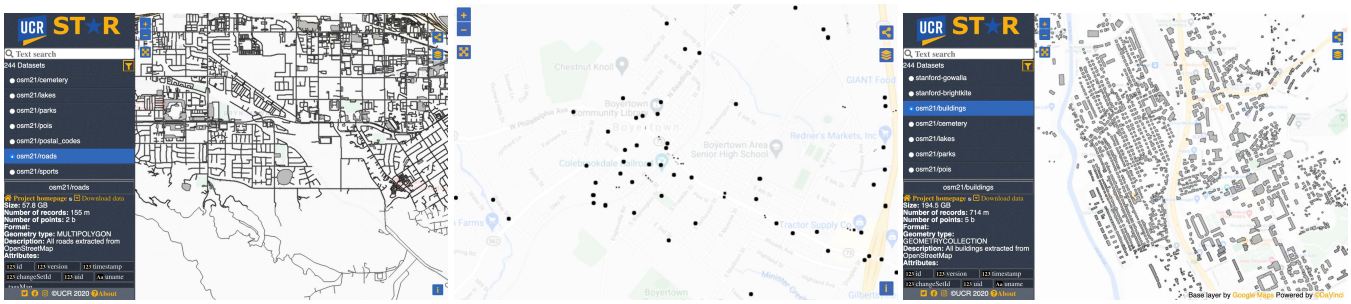
3.6 Extract All Objects

Since an object can be represented as a node, way, or relation, we perform a final union step that combines objects of the three types. To avoid repetition, we only include *dangled nodes*, that is, *nodes* that were not included in any *way*. Similarly, we only include dangled ways that are not included in any relations. This union step produces one very large dataset with all objects in the OSM PBF file. We can also retrieve subsets of coherent objects based on the tags including buildings, lakes, parks, cemeteries, sports, and postal_codes. The unified schema of all produced data is given in Figure 1. Finally, the output DataFrame can be written in any standard format, e.g., GeoJSON.

To summarizes the overall process: First, the input PBF file is parsed into an RDD of entities. After that, we split it into three DataFrames: nodes, ways, and relations. Nodes and POIs are directly exported to the output. After that, we perform a left join between *nodes* and *ways* to bring the location information to the *ways* dataset. *Nodes* that do not belong to any *way* are called *dangled nodes* and are kept on the side. After that, we join the *ways* with the *relations* to bring the geometry information into the *relation* DataFrame. Similarly, *dangled ways* are kept on the side. Finally, we perform a union of the three datasets, *dangled nodes*, *dangled ways*, and *relations* into one dataset that becomes our final master dataset. This final dataset can be further filtered based on the tags to produce buildings, lakes, etc., and written to the output.

4 EXPERIMENTS

We ran the proposed extractor on an Amazon AWS cluster of 51 m5.4xlarge nodes. Each node was equipped with 16 cores, 64 GB of RAM, and 256GB SSD storage. We processed the entire Planet.osm.pbf file in 4,412 seconds and extracted all the datasets indicated in Table 2. This has a total cost of nearly $70 which is a decent price to extract all this data in slightly more than an hour. Table 1 shows the timeline of the extraction process. All timestamps



(a) Roads (b) Points of Interest (POIs) (c) Buildings

Figure 2: Visualization of parts of the extracted datasets from UCR-Star

Table 1: Extraction timeline

Timestamp (seconds)	Event
0	Extraction process started
603	Extracted all nodes
707	Extracted points of interest
2142	Extracted roads
4194	Extracted all objects
4264	Extracted buildings
4294	Extracted lakes
4331	Extracted parks
4359	Extracted cemetery
4386	Extracted sports
4412	Extracted postal codes
4412	**Extraction process finished**

Table 2: Summary of extracted datasets

Name	# records	# points	Size
Nodes	6.8 Billion	6.8 Billion	2.2 TB
POIs	141 Million	141 Million	64 GB
Roads	155 Million	1.8 Billion	120 GB
Objects	1.5 Billion	13.7 Billion	1 TB
Buildings	714 Million	4.6 Billion	413 GB
Cemetery	764 Thousand	7.5 Million	552 MB
Lakes	26.5 Million	846 Million	38 GB
Parks	64 Million	1.3 Billion	64 GB
Postal Codes	28 Thousands	777 Thousands	45 MB
Sports	8.8 Million	88.8 Million	6.3 GB

are relevant to the start time of the extractor. Notice that after extracting all objects, all the remaining datasets take very little time because we persist the objects dataset and use it for subsequent datasets by applying a simple filter function on the tags.

Table 2 summarizes the datasets that we extracted using the proposed extractor and are making available as part of this paper. These datasets are available for public download in a compressed format[3]. Notice that users can easily extract other datasets using the proposed extractor by simply modifying the set of tags that identify their data of interest. All these datasets are available in GeoJSON format and their schema is similar to the one shown in Figure 1.

[3]https://drive.google.com/drive/folders/1QmKFD56bQ9XkpSKMoDa7XnxVuoofCYLQ

Figure 2 shows parts of the visualizations of the extracted data after being hosted on UCR-Star. Users can zoom in/out to inspect the extracted data. They can also download the entire dataset or a subset of their choice.

5 CONCLUSION

The paper proposes a spark-based extractor that can be used to extract rich geospatial datasets from OpenStreetMap (OSM). The proposed system can work directly with the compressed OSM PBF file to extract datasets based on the user's requirements. The experiments show that the extractor is scalable and can extract datasets from the OSM PBF file for the whole planet. The planet-wide datasets extracted from OSM are hosted on UCR-Star which allows users to visually explore and download them.

REFERENCES

[1] Louai Alarabi et al. 2014. TAREEG: A MapReduce-based system for extracting spatial data from OpenStreetMap. In *SIGSPATIAL*. 83–92.
[2] Almendros-Jiménez et al. 2019. Integrating and querying OpenStreetMap and linked geo open data. *Comput. J.* 62, 3 (2019), 321–345.
[3] The Big Data Lab at UCR. 2020. Beast: Big Exploratory Analytics for Spatio-temporal data. http://bitbucket.org/bdlabucr/beast/.
[4] Sören Auer, Jens Lehmann, and Sebastian Hellmann. 2009. Linkedgeodata: Adding a spatial dimension to the web of data. In *International Semantic Web Conference*. Springer, 731–746.
[5] Ahmed Eldawy and Mohamed F. Mokbel. 2014. Pigeon: A Spatial MapReduce Language. In *IEEE 30th International Conference on Data Engineering (ICDE 2014)*. Chicago, IL, 1242–1245.
[6] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*. IEEE Computer Society, Seoul, South Korea.
[7] GeoFabrik. 2021. GeoFabrik OpenStreetMap Extracts. https://www.geofabrik.de/data.
[8] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. 2019. UCR-STAR: The UCR Spatio-Temporal Active Repository. 11, 2 (2019).
[9] Boksoon Myoung et al. 2018. Estimating live fuel moisture from MODIS satellite data for wildfire danger assessment in Southern California USA. *Remote Sensing* 10, 1 (2018), 87.
[10] Photon 2021. Photon by Komoot. https://github.com/komoot/photon.
[11] AJ Prata and IF Grant. 2001. Retrieval of microphysical and morphological properties of volcanic ash plumes from satellite data: Application to Mt Ruapehu, New Zealand. *Quarterly Journal of the Royal Meteorological Society* 127, 576 (2001), 2153–2179.
[12] Mehmet Şahin, Yılmaz Kaya, Murat Uyar, and Selçuk Yıldırım. 2014. Application of extreme learning machine for estimating solar radiation from satellite data. *International Journal of Energy Research* 38, 2 (2014), 205–212.
[13] P Shanmugapriya, S Rathika, T Ramesh, and P Janaki. 2019. Applications of remote sensing in agriculture-A Review. *International Journal of Current Microbiology and Applied Sciences* 8, 1 (2019), 2270–2283.
[14] Andrew A Tronin. 2010. Satellite remote sensing in seismology. A review. *Remote Sensing* 2, 1 (2010), 124–150.