Pigeon: A Spatial MapReduce Language*

Ahmed Eldawy Mohamed F. Mokbel

Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
eldawy@cs.umn.edu mokbel@cs.umn.edu

***Abstract*—With the huge amounts of spatial data collected everyday, MapReduce frameworks, such as Hadoop, have become a common choice to analyze big spatial data for scientists and people from industry. Users prefer to use high level languages, such as Pig Latin, to deal with Hadoop for simplicity. Unfortunately, these languages are designed for primitive non-spatial data and have no support for spatial data types or functions. This demonstration presents Pigeon, a spatial extension to Pig which provides spatial functionality in Pig. Pigeon is implemented through user defined functions (UDFs) making it easy to use and compatible with all recent versions of Pig. This also allows it to integrate smoothly with existing non-spatial functions and operations such as Filter, Join and Group By. Pigeon is compatible with the Open Geospatial Consortium (OGC) standard which makes it easy to learn and use for users who are familiar with existing OGC-compliant tools such as PostGIS. This demonstrations shows to audience how to work with Pigeon through some interesting applications running on large scale real datasets extracted from OpenStreetMap.**

I. Introduction

Hadoop [1] became a standard choice for large scale data processing due to its efficiency, simplicity and open source nature. It has been adopted in many applications including graph processing [2], machine learning [3] and terabyte sorting [4]. Hadoop employs the MapReduce programming paradigm in which programs are designed as two functions, map and reduce. The map function maps each input record to a set of intermediate key-value pairs while the reduce function collects values with a similar key and produces the final answer. To analyze large scale spatial data, several spatial operations were implemented as map and reduce functions including range query [5], kNN [6], spatial join [7] and kNN join [8], [9].

Dealing with Hadoop through MapReduce programs is cumbersome for non-technical users. Thus, several SQL-like high level languages were developed including Pig Latin [10], HiveQL [11] and Y-Smart [12] to simplify large scale data processing in Hadoop. These languages allow users to describe their programs in terms of standard operations such as filter, sort and join. These primitive operations are combined together to perform more complex operations the same way it is done in SQL. Unfortunately, existing languages do not support spatial data types or functions making it difficult to analyze large scale spatial datasets.

In this demonstration, we present Pigeon, an extension to Pig Latin to support spatial data processing in Hadoop. It is available as a free and open source software at *http://github.com/aseldawy/pigeon*. Pigeon is compliant with Open Geospatial Consortium (OGC) *simple feature access* standard which is supported in both open source and commercial spatial DBMS. This makes it easier for users familiar with existing SDBMS to migrate their existing code to Pigeon with the least effort. Pigeon supports OGC standard data types including point, linestring and polygon, as well as OGC standard functions for spatial data import/export, querying and manipulation. The spatial functionality is implemented as user defined functions (UDFs) which are seamless to integrate with existing non-spatial operations in Pig and also makes it compatible with all recent versions of Pig that support UDFs. This extension is part of the SpatialHadoop system [13] which provides efficient spatial processing in Hadoop.

In this demo we describe how the spatial functionality is implemented in Pigeon and how it can be used to perform complex spatial queries efficiently on a cluster of machines. During the demonstration, a cluster of 20 machines will be running on Amazon EC2 and it will be accessible through a laptop used as a front end. We start with a 400GB semi-structured XML file provided by OpenStreetMap [14] and contains all the information about the planet. We show how a simple Pigeon script parses the file, extracts different datasets, and stores them back to disk in a structured format. After that, we demonstrate more Pigeon scripts to query and analyze the extracted datasets efficiently on the cluster. We provide a range of scripts ranging from a simple query on one relation to more complex queries operating on multiple relations. In all cases, the scripts run efficiently by exploiting the underlying parallelism of the Hadoop framework.

II. OpenStreetMap Dataset

OpenStreetMap [14] is a project for collecting map information around the world by volunteers. This data is publicly available in the form of one big file (i.e., planet file) stored in XML format. The planet file has three main sections, namely, *nodes*, *ways*, and *relations*. In this demo, we only deal with the first two sections. The *nodes* section contains a set of points where each point is defined by an ID, latitude, longitude and textual tags. Some points have a physical meaning (e.g., a house or bus stop) while others are just logical points used later in the ways section. The *ways* sections contains a set of ways where each way is defined by an ID, list of points (referenced by their IDs) and textual tags. The list of points define the spatial information of this way while the tags are used to identify the its type (e.g., park, lake or road) as well as meta information about this way (e.g., lake name or speed limit). Relations are used to combine several ways and points to represent more complex entities. For example, a university campus can be represented as a relation that contains buildings, roads and bus stops. For the sake of this demonstration, we only use points and ways.

*This work is supported in part by the National Science Foundation, USA, under Grants IIS-0952977 and IIS-1218168.

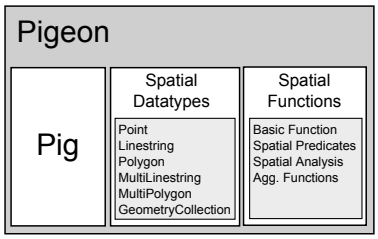

Fig. 1. Overview of Pigeon

III. Overview

Fig. 1 gives an overview of Pigeon. The two main components of Pigeon are spatial datatypes and spatial functions. Pigeon provides support to the standard OGC datatypes including Point, Linestring, MultiLinestring, Polygon, MultiPolygon, and GeometryCollection. For compatibility with existing OGC-compliant systems (e.g., PostGIS), Pigeon can import spatial objects stored in the Well-Known Text (WKT) and Well-Known Binary (WKB) formats. Details of the datatypes are provided in Section IV.

Spatial functions are implemented as user-defined functions (UDFs) which mesh well with existing operations provided in Pig such as filter, join and group by. There are four groups of spatial functions implemented in Pigeon: (1) Basic spatial methods, (2) Spatial predicates, (3) Spatial analysis, and (4) Aggregate functions. Details of how the spatial functions are implemented and used are given in Section V.

IV. Spatial Datatypes

The first step in a Pig script is to load the input file (relation) and optionally specify its schema defined by a name and a data type for each column. Columns with unspecified types default to `bytearray` which is just the raw bytes stored in that field. Traditional Pig provides support for primitive datatypes such as numbers and strings. It falls short in supporting spatial datatypes such as point and polygon. It does not provide any means of supporting user-defined custom datatypes either. To overcome these limitations, Pigeon overloads the `bytearray` data type to work with spatial data types. The `bytearray` datatype represents the value of a field as a raw array of bytes. When a spatial function is called, the spatial object is retrieved from its binary format, the spatial function is executed on the spatial object, and finally, the result is converted back to binary. This has an advantage of being compatible with the standard Pig but imposes an overhead of converting between binary and geometry back and forth.

When a user loads a spatial file, the spatial column is initially loaded as a *bytearray*. When a spatial function is called, it automatically parses the value in that column to retrieve the spatial object stored in it. Pigeon supports the three common formats used by other tools, namely, Well-Known Text (WKT), Well-Known Binary (WKB) and hex-encoded WKB. To export the value of a spatial column, Pigeon provides three methods `AsBinary`, `AsText` and `AsHex` to convert a spatial object back to one of the three standard formats.

V. Spatial Functions

Pigeon utilizes the extensibility of Pig to implement the spatial functions as UDFs which makes it easy to setup and use in an existing Pig installation. Pigeon provides four categories of spatial functions: (1) Basic spatial functions, (2) Spatial predicates, (3) Spatial analysis, and (4) Aggregate functions. A full list of all supported functions can be found in the documents of OGC standards [15].

A. Basic Spatial Functions

A basic function retrieves some basic information of a single spatial object such as the perimeter length or area. Functions in this category are implemented as simple `Eval` functions where the input is one primitive value and the output is another primitive value. In methods where the input is expected to be a spatial object (e.g., `Area`), the function automatically converts the input from WKT or WKB to a spatial object. Similarly, if the output is a spatial object (e.g., `MakePoint`), the output is converted to WKB. This automatic conversion allows Pigeon to work without the need to define a new datatype for spatial data which is currently not supported by Pig.

B. Spatial Predicates

A spatial predicate function takes one or two spatial objects and returns a Boolean value based on the relationship of the input object(s) (e.g., IsClosed or Touches). These functions are implemented as simple UDF `Eval` functions which take one or two values and return a Boolean value. Similar to the basic functions, the input objects are automatically converted from WKB or WKT for easy integration with the standard Pig. No need to convert the output value as it is always of type Boolean which is natively supported by Pig.

C. Spatial Analysis

A spatial analysis function performs some spatial transformation on spatial objects. Some functions are unary (e.g., `Centroid`) while others are binary (e.g., `Intersection`). Functions in this category are implemented as `Eval` functions which take one or two objects as input and return one object as output. Similar to other functions, inputs and outputs are automatically converted to and from spatial objects as needed.

D. Aggregate Functions

A spatial aggregate function takes a set of spatial objects and returns a single value that summarizes all the input; e.g., the function `ConvexHull` returns one polygon that represents the minimal convex hull of all input spatial objects. These functions are implemented as `Algebraic` aggregate functions which are computed efficiently in Pig using incremental computations. First, the function is applied locally in each machine to compute partial results, then the same function is applied globally on the partial results to produce the final answer. For example, multiple local convex hulls are first computed in each machine, then the global convex hull is computed by combining all local hulls. This technique is very efficient as the local computation step runs in parallel and reduces the data size which speeds up the final step.

Algorithm 1 Planet Extraction

```
1:  nodes = LOAD 'planet.xml' USING XMLLoader('node');
2:  points = FOREACH nodes GENERATE id, MakePoint(latitude, longitude) AS
    location, tags;
3:  STORE points INTO 'points.csv';
4:  nodes = LOAD 'planet.xml' USING XMLLoader('ways');
5:  ways = FOREACH nodes GENERATE way_id, Flatten(nodes), tags;
6:  way_nodes = JOIN points nodes BY id, ways BY node_id;
7:  grouped_ways = GROUP way_nodes BY way_id;
8:  ways_with_geom = FOREACH grouped_ways {
9:     GENERATE group AS way_id, MakeLine(grouped_ways) AS geom, tags;
10: }
11: STORE ways_with_geom INTO 'ways.csv';
```

Algorithm 2 Largest city

```
1: cities = LOAD 'cities.csv' AS (city_id: int, city_geom);
2: city_area = FOREACH cities GENERATE city_id, Area(city_geom) AS area;
3: ordered_cities = ORDER city_area BY area DESC;
4: largest_city = LIMIT ordered_cities 1;
```

Algorithm 3 Overlapping roads and rivers

```
1: roads = LOAD 'roads.csv' AS (road_id: int, road_geom);
2: rivers = LOAD 'rivers.csv' AS (river_id: int, river_geom);
3: cross_prod = CROSS roads, rivers;
4: overlapping = FILTER cross_prod BY Overlap(road_geom, river_geom);
```

VI. SHOWCASE: OPENSTREETMAP

As an example of how Pigeon processes large scale spatial datasets, we take OpenStreetMap data as a showcase. We start with the planet file which contains all the information about the globe in a semi-structured XML file. First, we show how a simple Pigeon script extracts all the data from the XML file and stores it in a structured format. Then, we show different examples of analyzing the extracted datasets using Pigeon.

A. Planet Extraction

The first task we run using Pigeon is to extract all nodes and ways from OpenStreetMap planet file. To accomplish this task, we run the Pigeon script showed in Algorithm 1. The script runs in two phases, *point extraction* and *way extraction*. *Point extraction* (lines 1-3), starts by loading the XML nodes representing points from the XML file. In line 2, it selects the columns of interest from XML node, namely, ID, latitude, longitude and tags. Notice the use of the new method `MakePoint` to combine the latitude and longitude in one column of type `Point`. Finally, the extracted nodes are stored back to disk in line 3.

Way extraction 4-11 starts by loading the XML nodes representing ways. In line 5, it extracts way ID, tags and flatten the list of node IDs. This statement expands the list of nodes IDs by replicating the whole record for each node ID and setting the column node-id to a different value in the list. This is similar to normalization of 1NF in databases. Line 6 joins the `ways` relation with the `nodes` relation (extracted in phase 1) to substitute each node ID with its geo location. Then, line 7 groups ways by way_id to ensure that all points belonging to one way are grouped together. Lines 8-10, generate a linestring for each way by combining all nodes in this way in one linestring. Here we use the new method `MakeLine` which creates a linestring from a list of points. Finally, results are stored back to disk in line 11.

B. Filtered Extraction

In this task we extract a subset of the data by applying some spatial and non-spatial filters. First, we can apply a spatial filter to extract data in a specific area defined by a polygon using a `Filter` statement after line 2. For example, the statement:

```
Filter points BY Contains(
  Rectangle(west, south, east, north),
  location);
```

keeps only points contained in the defined rectangle. The inner join operation in line 6 will automatically remove ways that do not have any points in the area of interest. The final result will contain only data in the specified area.

We can also apply a non-spatial filter after line 4 which filters ways according to their tags and leaves only ways with specific tags. This can be used, for example, to extract ways that represent rivers by keeping only ways that has a tag of key 'waterway' and value 'river'. For example, the statement:

```
Filter ways BY tags#'waterway' == 'river';
```

keeps only ways that represent rivers. We use non-spatial filters to extract datasets that represent roads, rivers, lakes, parks and cities. These datasets are used in analysis tasks that are described next.

C. Analysis

We use the datasets extracted above to run large scale spatial analysis queries using Pigeon. In the demonstration we show three examples of spatial analysis queries. The first example shows a simple spatial function call on one relation. The second example is a traditional spatial join query which operates on two relations. The third example is a more comprehensive example which shows a non-trivial spatial query that operates on two relations. Notice that the three queries can be easily executed in SDBMS using SQL but the advantage of running them in Pigeon is that they automatically scale to hundreds of machines according to the cluster size.

1) Largest city: This simple example shows how to select the city with the largest area. Simply, it calculates the area of each city, orders them by area and selects top one. The Pigeon script is shown in Algorithm 2 where line 1 loads the cities relation and line 2 calculates the area of each city. After that, lines 3 and 4 order cities by their area in descending order and selects the first one (i.e., largest city).

2) Overlapping roads and rivers: In this example, we show a typical spatial join query which finds roads that overlap with rivers. As shown in Algorithm 3, this query can be represented easily using Pigeon by computing the cross product of the two relations (line 3) and applying the spatial predicate (i.e., overlap) as a post processing filter (line 4).

3) Lake area per country: This example calculates the total area occupied by lakes in each country. The script is shown in Algorithm 4. Lines 1, 2 load the relations of lakes and countries. Notice that lakes are not directly related to countries in OpenStreetMap data. Thus, we need to perform a spatial join step using the overlap predicate (lines 3, 4) to find the lakes which overlap with each country. After that, we

Algorithm 4 Lake area per country

```
1: lakes = LOAD 'lakes.csv' AS (lake_id: int, lake_geom);
2: countries = LOAD 'countries.csv AS (country_id: int, country_geom);
3: lake_country = CROSS lakes, countries;
4: overlap_lakes = FILTER lake_country BY Overlap(lakes_geom, country_geom);
5: grouped_lakes = GROUP overlap_lakes BY country_id;
6: total_area = FOREACH grouped_lakes
7:   GENERATE group AS country_id,
8:   Sum(Area(Intersection(overlap_lakes.lakes_geom, overlap_lakes.country_geom)))
   AS lakes_area;
```

group the join result by country ID in line 5. To calculate the total area for each country, we first compute the intersection between each pair of overlapping country and lake using the `Intersection` method to consider only the portion of the lake inside the country. Then, we calculate the area for this portion using the `Area` method. Finally, we sum all the areas using the standard `Sum` method. Notice how multiple spatial functions are nested in one line and how we mix spatial and non-spatial functions in one statement.

VII. Demonstration Scenario

To demonstrate Pigeon, we deploy a cluster of 20 machine that is running on Amazon EC2 with the latest versions of Apache Hadoop (1.2.1) and Pig (0.12) installed on them. Pigeon is then installed on that cluster to allow spatial scripts to run. We also build a web interface (shown in Fig. 2) to assist attendees to write their Pigeon scripts, submit their queries, and visualize the results. The cluster will be preloaded with several datasets extracted from OpenStreetMap including the raw XML planet file, as well as the extracted files of buildings, cities, countries, lakes, parks, rivers, and roads. We will have the Pigeon scripts described in Section VI ready for demonstration while the attendees can modify these scripts or provide their own.

The panel on the left displays available relations (files) and running queries. When a user selects a dataset, the first few lines are displayed in the preview area to help the user understand the file schema and column types. For example, in Fig. 2, the `parks` relation is selected and the preview area shows that it is a tab separated file where the first column is an integer ID and the second column is a spatial object stored in Well-Known Text (WKT) format. If a user selects a running query, the preview area displays its status and progress as reported by Pig in the back-end.

At the bottom, there is a text area where users can write a Pigeon script. The script is then submitted to the back-end for execution. Users can provide a user-friendly name for a query to be displayed in the list of running queries. During the demonstration, we will provide the five Pigeon scripts described in Section VI for users to run, namely, (1) Planet Extraction, (2) Filtered Extraction, (3) Largest city (4) Overlapping roads and rivers, (5) Lake area per country. Users are also allowed to edit these scripts to provide different behavior or even provide their own scripts.

As spatial analysis queries are executed, users can track their execution in the cluster. It will be shown during demonstration that both simple and complex analysis queries scale efficiently by executing in parallel on all nodes in the cluster. Attendees have two options of monitoring the progress of

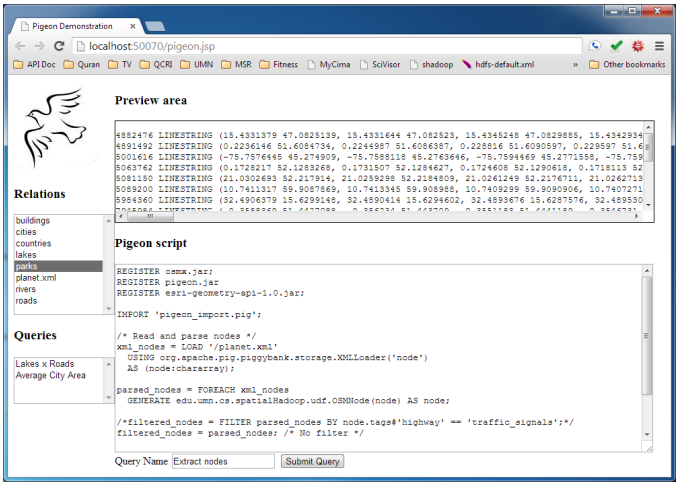

Fig. 2. Front-end of Pigeon

queries. First, they can track the progress of the script as a whole as reported by Pig. In addition, users can also track the progress of individual MapReduce jobs as reported by Hadoop. While simple queries are mapped to a single MapReduce job, more complex queries are mapped to multiple MapReduce jobs.

References

[1] "Apache. Hadoop," http://hadoop.apache.org/.

[2] "Giraph," http://giraph.apache.org/.

[3] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative Machine Learning on MapReduce," in *ICDE*, 2011.

[4] O. O'Malley, "Terabyte Sort on Apache Hadoop," 2008.

[5] Q. Ma, B. Yang, W. Qian, and A. Zhou, "Query Processing of Massive Trajectory Data Based on MapReduce," in *CLOUDDB*, 2009.

[6] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Spatial Queries Evaluation with MapReduce," in *GCC*, 2009.

[7] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "SJMR: Parallelizing spatial join with MapReduce on clusters," in *CLUSTER*, 2009.

[8] C. Zhang, F. Li, and J. Jestes, "Efficient Parallel kNN Joins for Large Data in MapReduce," in *EDBT*, 2012.

[9] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient Processing of k Nearest Neighbor Joins using MapReduce," *PVLDB*, 2012.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-so-foreign Language for Data Processing," in *SIGMOD*, 2008.

[11] A. Thusoo, J. S. Sen, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A Warehousing Solution over a Map-Reduce Framework," *PVLDB*, 2009.

[12] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, "Ysmart: Yet another sql-to-mapreduce translator," in *ICDCS*, 2011.

[13] A. Eldawy and M. F. Mokbel, "A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data," in *VLDB*, 2013.

[14] "OpenStreetMap," http://www.openstreetmap.org/.

[15] "Open Geospatial Consortium," http://www.opengeospatial.org/.