Apache spark on planet scale

Denis Chaplygin
Software engineer @ Wolt

Jan 2020

Wolt

OpenStreetMap

- **OpenStreetMap (OSM)** is a collaborative project to create a free editable map of the world
- Database of all the features found on the surface (and below) of planet Earth.
- The Earth is big, so the database is big: ~1.2TB
- Looks like a perfect case for Apache Spark

Apache Spark

- **Apache Spark** is an open-source distributed general-purpose cluster-computing framework
- In-memory processing with automatic data partitioning and sharding
- Virtually unlimited in RAM and CPU cores.
- Designed to process huge amounts of data effectively
  - By ‘huge’ we mean - more data that you can usually fit to RAM.

Using OSM data in Apache Spark

1. **Loading from external database**

The simplest way to use your data is to import OSM database into PostGIS or Osmosis database and load geometries using JDBC datasource.

Pros:

- Everything is already implemented
- Filtering of geometry on the database side

Cons:

- Extremely long import process
- Need to maintain external DB,import process etc

2. Converting to format understood by Spark

With Magellan extension Apache spark can read geometries in GeoJSON or WKB formats.

Pros:

- Everything is already implemented

Cons:

- Even slower preparation process, that also needs to be maintained
- No filtering on load

3. Load OSM data directly to Spark

Directly load OSM database as Spark Dataframe.

Pros:

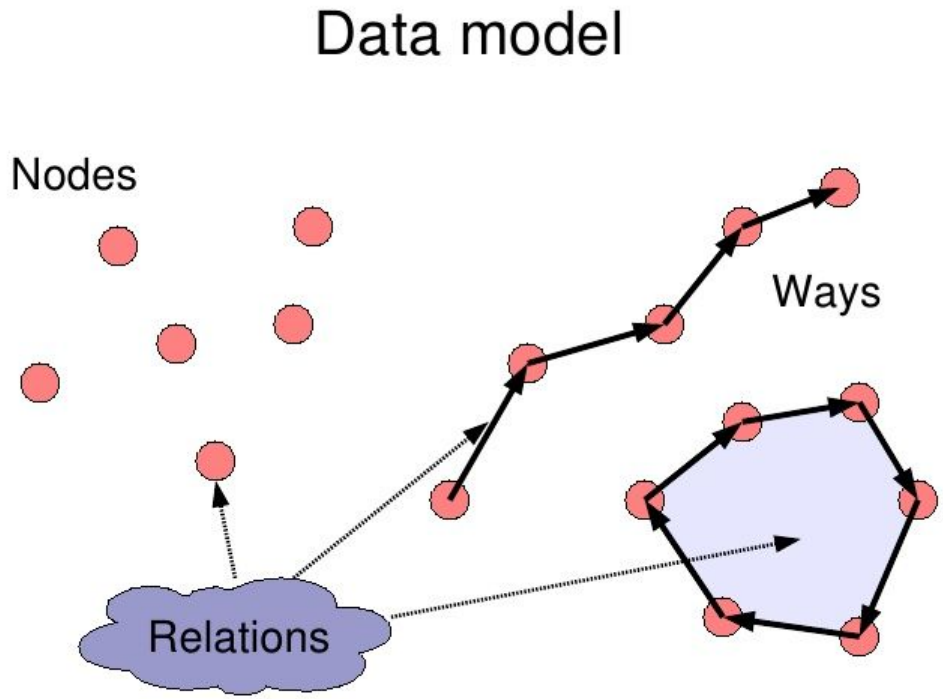
- Simplest way to get the data, no external dependencies
- Partial filtering support on load

Cons:

- I had to code it myself :(

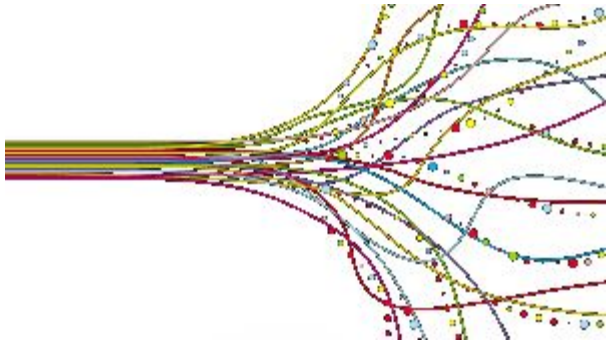
The problem #1 with OSM data

- OSM data consists of 3 types of entities:
  - Nodes with coordinates
  - Ways referring to the nodes
  - Relations referring to the nodes and ways
- Typical OSM data file is sorted in the way, that it stores nodes first, then ways, then relations.
- Due to the size of the OSM database you are forced to process it sequentially, as you can't fit it in to you RAM.
- But with Apache Spark you actually can store the whole planet in the cluster, so you don't care about sequentiality anymore and can read OSM file in random multithreaded manner



The solution #1: parallelpbf

- OSM PBF format multithreaded reader written in Java 8.
- Supports all current OSM PBF features and options
- Available under GPLv3 at https://github.com/woltapp/parallelpbf and Maven **com.wolt.osm:parallelpbf:0.2.0**
- Easy to use, callback based API:



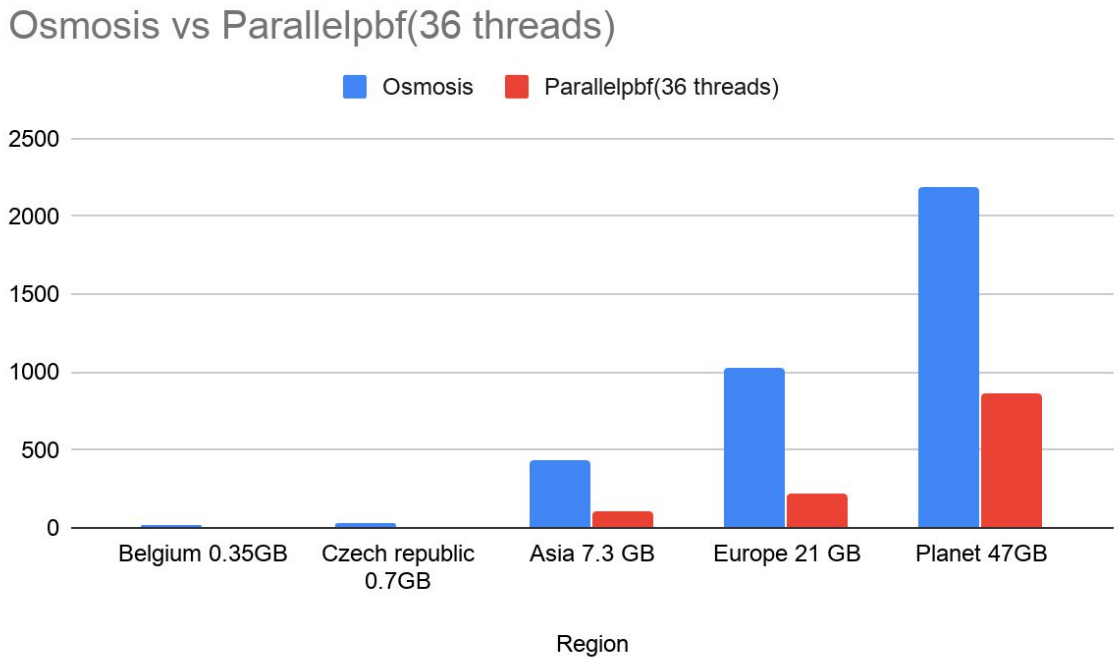
```
InputStream input = Thread.currentThread().getContextClassLoader().getResourceAsStream("sample.pbf");
new ParallelBinaryParser(input, 36)
        .onHeader(this::processHeader)
        .onBoundBox(this::processBoundingBox)
        .onComplete(this::printOnCompletions)
        .onNode(this::processNodes)
        .onWay(this::processWays)
        .onRelation(this::processRelations)
        .onChangeset(this::processChangesets)
        .parse();
```

- Skips reading of entities without callback set.

parallelpbf performance improvements

- Test format: reading OSM files and counting ‘fixme’ tags for each entity type
- Comparing Osmosis library reader and parallelpbf
- Running on c5d.9xlarge instance with 36 cores and local SSD

- Belgium, 0.35GB file
  - **18 seconds** osmosis reader
  - **7 seconds** 36 threads
- Czech Republic, 0.7GB file
  - **34 seconds** osmosis reader
  - **11 seconds** 36 threads
- Asia, 7.7GB file
  - **431 seconds** single thread
  - **104 seconds** 36 threads
- Europe, 21GB file
  - **1024 seconds** single thread
  - **224 seconds** 36 threads
- Planet, 49GB file
  - **2194 seconds** single thread
  - **864 seconds** 36 threads



The problem #2 with OSM data and parallelpbf

- Parallelpbf reader will only be executed on a single Spark cluster node (master node) and all other executors nodes will be waiting for it
- The dataset have to fit master node RAM
- Dataframe creation will redistribute data from master node to executor nodes, causing unneeded data moves.

The solution #2: spark-osm-datasource

- OSM PBF format Spark datasource, built on top of parallelpbf
- Supports Scala 2.11 and 2.12, will support 2.13 when Spark will catch up.
- Available under GPLv3 at https://github.com/woltapp/spark-osm-datasource and Maven **com.wolt.osm:spark-osm-datasource:0.3.0**
- Supports partitioning, thus loading partitions of OSM file in parallel on all executors.
- Supports Spark local file distribution mechanism, to save time on S3 transfer
- Supports partial filtering
- Running same tags counter as for parallelpbf on a planet file with cluster of 720 cores and 1440GB of Ram takes only **2.5 minutes**

Spark osm datasource example

```
  val osm = spark.read
      .option("threads", 6)
      .option("partitions", 32)
      .format(OsmSource.OSM_SOURCE_NAME)
      .load(HADOOP_URL).drop("INFO")
   val counted = osm.filter(col("TAG")("fixme").isNotNull).groupBy("TYPE").count().collect()
```

- Any Hadoop accessible file is supported, like local files, HDFS, S3, etc
- With ‘useLocalFile’ option you can use Spark built-in file distribution mechanism, saving time on retrieving file from external source several times
- Number of threads specifies, how many threads each Spark executor should use for loading it’s assigned partitions
- Instead of guessing input file size or hardcoding some specific number of partitions, you can explicitly specify, how OSM file should be splitted.

Making you life easier: spark-osm-tools

- Collections of useful Spark snippets for processing OSM data
- Supports Scala 2.11 and 2.12, will support 2.13 when Spark will catch up.
- Still work in progress
- Available under Apache 2.0 at https://github.com/woltapp/spark-osm-tools, but not published to the Maven Central yet.
- Includes procedures for merging OSM datasets, limiting/extracting by some polygon boundary, relation hierarchy processing
- Ways to geometry conversion
- Multipolygon solver
- Writer to the Osmosis format database
- Even a simple renderer!

Using it all together: public transport coverage for a city

- The goal is to analyze public transport coverage
- For each building (starting line) distance to a nearest public transport platform should be calculated.
- Buildings should be color coded with that distance

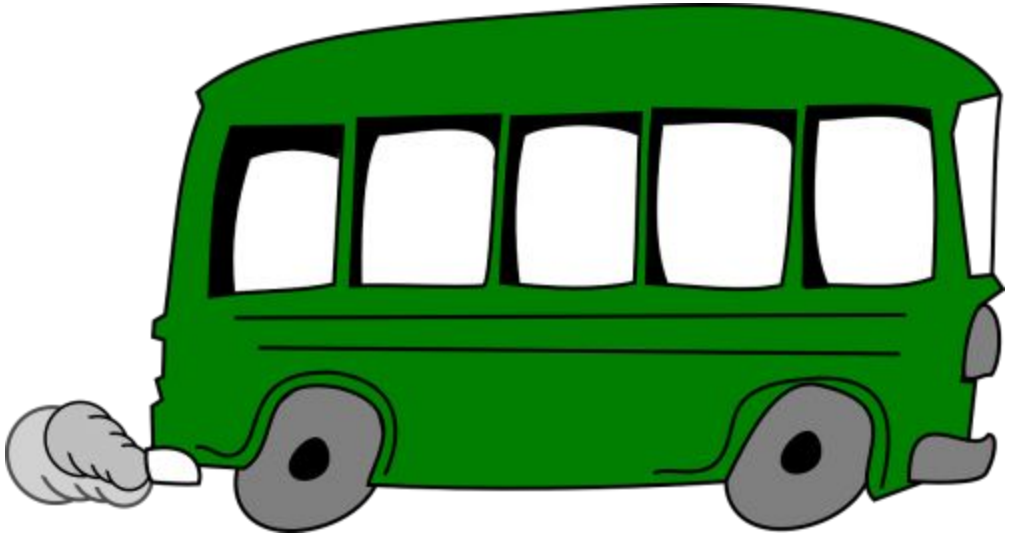


Image by Clker-Free-Vector-Images (pixabay.com)

public transport coverage for a city cont'd

- Load the map:

```
val osm = spark.read
  .option("threads", 2).option("partitions", 12).format(OsmSource.OSM_SOURCE_NAME)
  .load("belgium-latest.osm.pbf").drop("INFO")
```

- Build city boundaries polygon and extract city from the loaded data:

```
val brBoundary = osm.filter(col("TYPE") === OsmEntity.RELATION)
 .filter(lower(col("TAG")("boundary")) === "administrative" && col("TAG")("admin_level") === "4" && col("TAG")("ref:INS") === "04000") //Brussels
val brPolygon = ResolveMultipolygon(brBoundary, osm).select("geometry").first().getAs[Seq[Seq[Double]]]("geometry")
val area = Extract(osm, brPolygon, Extract.CompleteRelations, spark)
```

- Find locations of all public transport stops:

```
val stop_positions = area.filter(col("TYPE") === OsmEntity.NODE).filter(lower(col("TAG")("public_transport")) === "stop_position").select("LON", "LAT")
  .collect().map(row => (row.getAs[Double]("LON"), row.getAs[Double]("LAT")))
```

- Get building geometry

```
val way_buildings = area.filter(col("TYPE") === OsmEntity.WAY).filter(lower(col("TAG")("building")).isNotNull)
 .select("WAY").filter(size(col("WAY")) > 2).filter(col("WAY")(0) === col("WAY")(size(col("WAY")) - 1))
val buildingsGeometry = WayGeometry(way_buildings, area).drop("WAY")
```

public transport coverage for a city cont’d

- Do the actual analysis
    - Find buildings mean points

```
val meanPointUdf = udf{(geometry: Seq[Seq[Double]]) => {
  val lon = geometry.map(_.head).sum / geometry.size.toDouble
  val lat = geometry.map(_.last).sum / geometry.size.toDouble
  Seq(lon, lat)
}}
val buildingsMeanPoints = buildingsGeometry.withColumn("MEAN_POINT", meanPointUdf(col("geometry")))
```

    - Find distance to the nearest public transport platform

```
val distanceUdf = udf { (lon: Double, lat: Double) => stop_positions.map(position => haversine(lon, lat, position._1, position._2)).min }
val buildingsWithDistances = buildingsMeanPoints.withColumn("DISTANCE", distanceUdf(col("MEAN_POINT")(0), col("MEAN_POINT")(1)))
  .drop("MEAN_POINT")
```

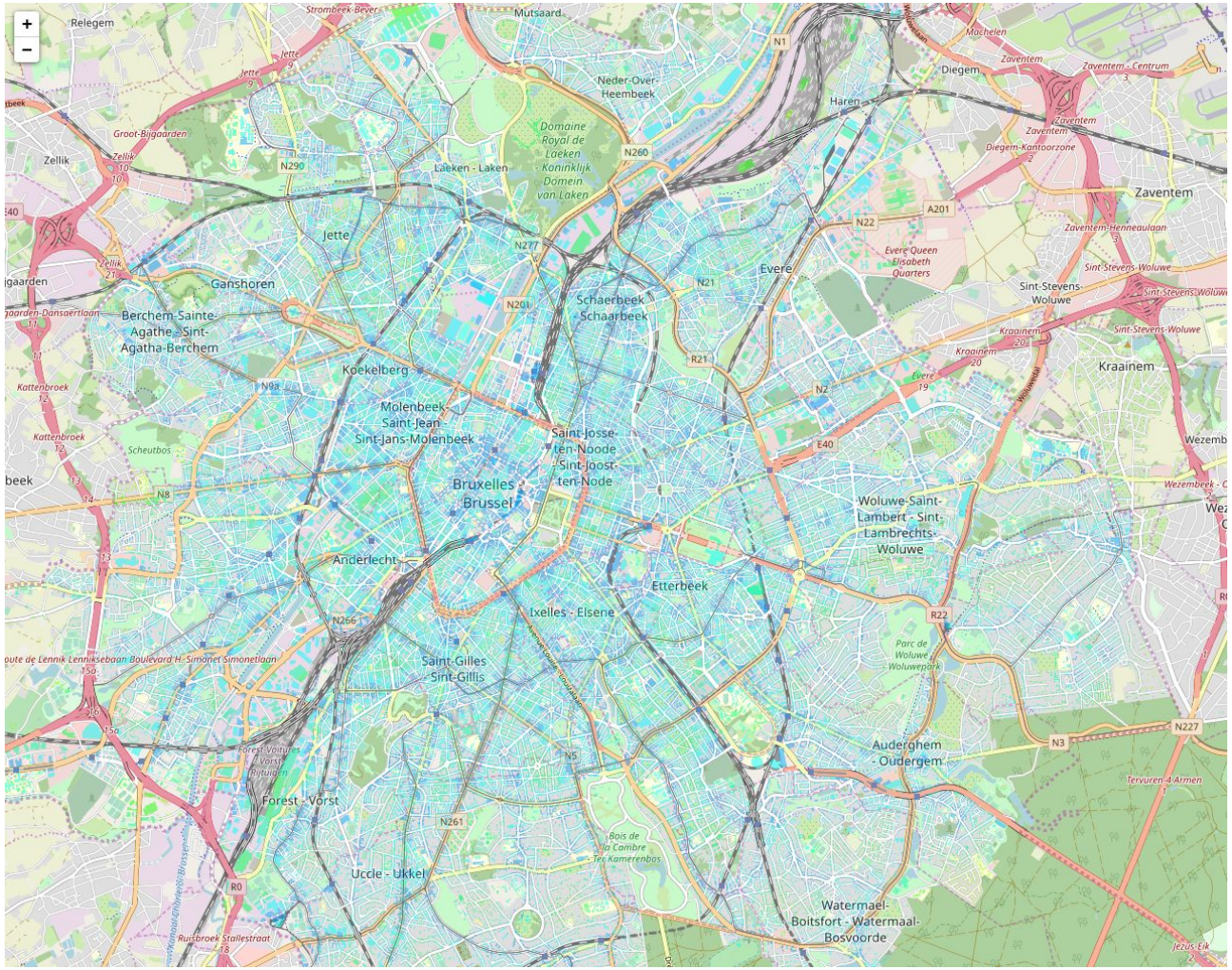
- Finally mark building for rendering and render them:

```
val distanceToRenderParametersUdf = udf(distanceToRenderParameters _) //Here colors are assigned
val symbolized = buildingsWithDistances.withColumn("symbolizer", lit("Polygon")) //Render buildings as polygons
  .withColumn("minZoom", lit(13)) //Render only at zoom levels starting from 13
  .withColumn("parameters", distanceToRenderParametersUdf(col("DISTANCE")))


Renderer(symbolized, 13 to 19, "/home/chollya/tiles/public_transport_coverage")
```

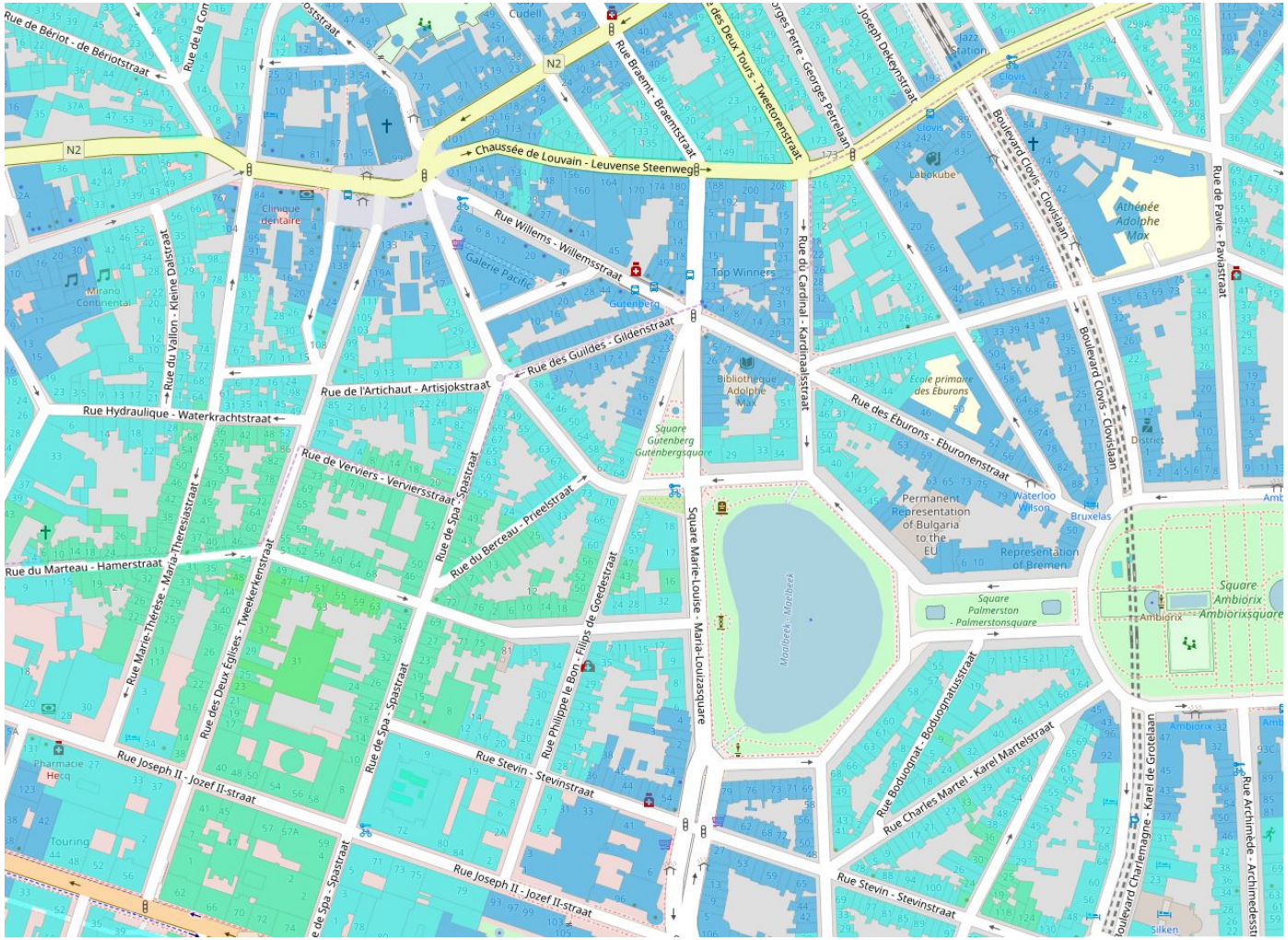
Public transport coverage for a city: results

- Brussels have good public transport coverage:



public transport coverage for a city: results

- There is no way to distinguish residential buildings from other building, like RUIAN in Czech republic



More exciting stuff is coming

- **parallelpbf**
  a. ~~(Unordered) writing support~~ ✓
- **spark-osm-datasource**
  a. Better filtering during load
  b. Geometry conversion on load(?)
- **spark-osm-tools**
  a. Relations solver for different types of relations
  b. GraphX support for relations hierarchy/polygons hierarchy
  c. Geometry conversion/operations
  d. Interoperation with GeoSpark/Magellan

Thank you!

parallelpbf: https://github.com/woltapp/parallelpbf
osm-spark-datasource: https://github.com/woltapp/spark-osm-datasource
osm-spark-tools: https://github.com/woltapp/spark-osm-datasource
Contact author: denis.chaplygin@wolt.com / https://github.com/akashihi
Order your lunch here**: https://wolt.com/**

Wolt