Project: Spatial Data with Hive

Kristian Per Bruun
Supervised by: Marcos António Vaz Salles & Pimin Konstantin Balic Kefaloukos

Department of Computer Science
University of Copenhagen

January 15, 2013

1 Introduction

The purpose with this project is to get some know-how on how to use hive on spatial data. In [6], Ablimit Aji enables spatial querying in a high performance system with large scale spatial data by a developed MapReduce framework named *MIGIS*. The queries are expressed in an SQL like language and then compiled into MapReduce jobs running in Hadoop. The approach in this project is very alike although, we have used the open source tool Hive, to query spatial data.

To do this we have implemented a MapReduce job running in Hadoop for importing spatial data into the Hadoop distributed file system (HDFS). We have used Hive for querying the imported spatial data. Since Hive does not support spatial data querying, we have implemented three different user defined functions(UDF). We have used JTS Topology Suite[4] to implement these UDFs.

The rest of this report is organized as following: section 2 describes the spatial data we use. Section 3 describes how we parse the spatial data and save it in HDFS. Section 4 briefly explains what Hive it and how we use it in this project. Section 5 describes the experiment that we have performed and thoughts derived from the result. And finally, in section 6 we conclude on this project.

2 Spatial data

For this project we have used spatial data from Open Street Map (OSM). A data set is represented in the following way:

```
<osm version="0.6" generator="CGImap 0.0.2">
 <node id="298884269" lat="54.0901746" lon="12.2482632" ... />
 <node id="1831881213" lat="54.0900666" lon="12.2539381" ... >
  <tag k="name" v="Neu Broderstorf"/>
  <tag k="traffic_sign" v="city_limit"/>
 </node>
 ...
 <way id="26659127"...>
```

```
  <nd ref="298884269"/>
  <nd ref="1831881213"/>
  ...
  <tag k="highway" v="unclassified"/>
  <tag k="name" v="Pastower Straße"/>
 </way>
</osm>
```

For simplicity, we have only shown the most interesting xml tags and attributes. For a more complete description of the data set, we refer to[5]. A *node* consists of a unique id together with a value for latitude and longitude that together comprise a coordinate. Each node can have tags associated with it. A tag is a key/value pair that gives some information concerning the given node. A node can also be represented as *well-known text* (WKT), e.g the string `POINT (54.0901746 12.2482632)` would be the WKT representation of the first node from the data set example. In [3], a complete definition of WKT is given.

A *way* is an ordered list of more than two nodes that are given by their unique ids in the `nd` tag. A way is uniquely defined by an id. As for nodes, a way can also contain tags. Each way written as WKT will be represented as either a *linestring* or a *polygon* if the way has more than three nodes and the first node equals the last node.

WKT is a well suited representation for spatial data because Hadoop do not support geometry data types but text strings are supported and well understood by JTS.

3 Importing data

To be able to query spatial data in an easy and efficient manner with Hive, we have transformed the data into a suitable representation. In this case a suitable representation is a text string given as

```
type id uid uname version visible changeset timestamp tags WKTgeometry
```

for each node and way in the data set, where `type` is either the literal *node* or *way* and `WKTgeometry` is the WKT representation of the geometry for a given node or way. `tags` are the key/value pairs that are associated with the node or way. A definition of the other columns can be found in [5].

Parsing of the spatial data is carried out by a four chained MapReduce job. First step is to make sure that all xml tags are non-empty closing tags. That is, for each xml tag of the form `<tag/>` transform it into `<tag></tag>`. By transforming the tags into non-empty closing tags, we can in the subsequent MapReduce jobs work with the tags in a consistent way. In Second step each node is transformed into a text string as the one described above.

Third step is a bit more complex and is best described together with pseudocode for the MapReduce task:

```
map(LongWritable key, Text value):
  // key: offset in file
  // value: either an xml element way or a transformed node text string
  if(value == node)
      EmitIntermediate(TextPair(nodeId, "0"), node.WKTgeometry);
```

```
if(value == WayXmlElement)
  for each nodeId in value.nodeRef
    EmitIntermediate(TextPair(nodeId, "1"), transform(value));

reduce(TextPair key, Iterator<Text> values):
  // key: composite of a nodeId and 0 or 1 depending on type of value
  // value: either a geometry point or a transformed way element
  geom = values.next();
  for each v in values
    Emit(key.nodeId, concat(v, geom));
```

The basic idea is to join a transformed node with each transformed way that has a reference to that particular node. This can be expressed with relational algrebra as $nodes \bowtie_{nodeId} ways$. In the Map phase either a node or a way is given as the value. For a node, the output key is a text pair consisting of a node id and the value 0. The output value is the a WKT representation of the geometry extracted from the node. For a way, the output key is for each reference to a node a text pair consisting of the node id and the value 1. The output value is the transformed way together with the sequence number for the nodeid.

The reasoning why using a composite output key for the mapper is to make sure that the WKT representation of the geometry comes before all transformed ways in the reducer when iterating through the input values. If we did not know the order of the node and ways in the reducer, we would have to store them in main memory and sort them afterwards. This could in situations where data sets are very large exceed the amount of memory and crash the job. So instead, we partition the data by node id and within each partition, we group by the second part of text pair in ascending order which means that a node value will always be the first in the input values to the reducer.

The reduce part gets the first value which is the WKT representation of geometry. For each of the rest of the values the node id is emitted together with the composition of the transformed way and the node WKT geometry.

The forth and last Mapreduce job is to create one way for each unique way id in the data set just created in the third step. In the map-phase the output key is a text pair consisting of the wayid and the sequence number for a given node. the output value is a transformed way. We use the text pair to partition by the way id and group by the sequence number in ascending order. Having sequence number in ascending order, we can in the reduce-phase create a correct WKT geometry for each way. We output the complete transformed way as a text string in the same format as the transformed node.

4 Hive

Hive [7] is an open source data warehousing solution built on top of Hadoop to support SQL-like queries. Hive query language (HiveQL) which is the SQL-dialect used to express queries in a declarative manner are translated into MapReduce jobs and executed using Hadoop. As with traditional relational database management systems, Hive stores data in tables that consist of a number of rows, and each row consists of a specified number of columns. Each column has an associated data type. It can either by a primitive type such as

integers, floating point numbers or strings, or it can be a complex type such as associative arrays, lists or structs.

Joins are supported in HiveQL but there are some limitations that need to be highlighted. Joins have to be specified using the ANSI join syntax. Also, only equality predicates are supported as the join predicate. This means that the first example from the two select statements below will fail while the latter one will succeed.

```
SELECT *
FROM t1 JOIN JOIN t2 ON intersects(t1.geom, t2.geom) = true

SELECT *
FROM t1 JOIN t2 ON t1.col = t2.col
WHERE intersects(t1.geom, t2.geom) = true
```

The semantic of the first select statement is that it will join those rows that are intersecting, while the latter select statement first joins the two tables and then starts to filter. It is a very inefficient way to do a join compared to the first example. The whole query language is documented in [2].

To be able to query the created data set for nodes and ways a table schema needs to be defined. It is defined as following:

```
CREATE TABLE t (
       type STRING,
       wayId INT,
       userId INT,
       username STRING,
       version INT,
       visible STRING,
       changeset STRING,
       tstamp STRING,
       tag MAP<STRING, STRING>,
       geom STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY '=';
```

where table name `t` is either `ways` or `nodes`. Each column is trivial and well-known from the SQL-92 standard except for the column `tag`, which is defined as an associated array. We delimit each key/value pair by a `','` and a key from a value by `'='`. Also, we need to specify how columns are delimited. this is by a tab-space. By default, each row is delimited by a newline or a carriage return.

4.1 User defined functions

In the two above select statements we make use of a function called `intersects`. This is a *user defined function* (UDF) [8]. There are three types of UDFs in Hive: regular UDFs that operate on a single row in a table and output a single row. User defined Aggregate functions (UDAFs) that work on multiple rows and output a single row. User defined table functions (UDTFs) that operate on a single row and produce multiple rows. We will in this project only experiment with regular UDFs.

A UDF is implemented in Java. It must satisfy two properties: UDF must be a subclass of `org.apache.hadoop.hive.ql.exec.UDF` and it must implement at least one `evaluate` method. The `evaluate()` method is not defined by an interface, which makes it possible to decide how many input parameters it takes and which data type it outputs.

In this project, three UDFs is implemented. These are:

- **intersects()** - reads two WKT objects and return true if they intersects,
- **contains()** - reads two WKT objects and return true if the first argument contains the second argument and,
- **distance()** - reads in two WKT objects and returns the minimum distance between the two objects.

To implement these functions JTS Topology Suite (JTS) [4] has been used. JTS is an api that implements a core set of spatial data operations such as the above mentioned, based on the the standard specification defined by the Open Geo Spatial Consortium.

5 experiment

The experiment is based on how expressive we can be in terms of querying spatial data with Hive. Since this experiment is performed on a single laptop it would not be meaningful to do benchmarking since Hadoop and Hive is meant for a large scale cluster.

The setup system for this experiment is a single laptop with an Intel Duo Core 2.00GHz, 3 GB RAM, 100 GB disk and with Ubuntu 12.04 as the operating system. Hadoop is set up to run in pseudo-distributed mode and with Hive on top of it with a local Derby database to store meta data.

Data is imported into HDFS as described in section 3. We have experimented with a data set for Bornholm from *open street map*. The data set is 32 MB in its normal form. In transformed form it takes up a total of 19 MB separated into two files: one data set for nodes and one data set for ways. This experiment will focus on how to well we can do join on spatial data. The most interesting thing is do queries on the data set containing ways since the nodes data set is just points that are not connected in any way.

The experiment performed is very much inspired by [9] in terms of how to join two tables. We make use of a standard join method where the condition statement is in the where clause. If we would like to express which ways intersect in HiveQL, it can be done in the following way:

```
SELECT a.wayId, b.wayId
FROM ways a JOIN ways b ON (a.type = b.type)
WHERE intersects(a.geom, b.geom) = true AND a.wayId <> b.wayId
```

This HiveQL code shows that we have to join by `type` which is the text string `way`. The output of this join is the Cartesian product and then a filtering is done. There are 9461 records in the ways data set. This means that a total of 89510521 records have to be tested for an intersect. Expressed in relational algebra, what we are trying to do is $\pi_{a.id,b.id}(way\ a \bowtie_{intersects} way\ b)$ but what we are doing is $\pi_{a.id,b.id}(\sigma_{intersects}(way\ a \bowtie_{type} way\ b))$.

Join is an expensive expression when we end up with The Cartesian product, but with Hadoop and Hive there are ways to speed this up. A cluster can be scaled out which gives more computing power. On a single laptop (that was not solely devoted to a MapReduce job) it took hours to join two tables and filter the output table. On a cluster of say 10 commodity machine fully devoted to this job, the time for the Hive job to finish for Bornholm can be reduced to few seconds. This postulat is based on how well hadoop scale from [1].

If one of the tables that are being joined are small enough it can be stored in Hadoops distributed cache, and the join can be performed in the map phase. Also if the data set is the whole world and not just Bornholm, a partitioning of the data into regions would save alot of time by only intersects region that are closed to each other and not the whole data set.

6 Conclusion

This project was meant as a know-how project to get some hands-on with technology such as Hive and JTS, how to use UDFs and how to work with spatial data. In this report the steps on how to be able to support spatial querying and spatial joins have been described.

Also, it has been described and shown in a simple local example how spatial data can by queried in the paradigm of large scale high performance systems. Joins have proven to be expensive since only equi-join is supported by Hive at the moment. Although, by scaling out, partitioning data and use Map-join if data set fits into main memory it is possible to optimize this type of query.

References

[1] Hadoop faq. `http://wiki.apache.org/hadoop/FAQ`. Accessed: 12/01/2013.

[2] Hive wiki. `https://cwiki.apache.org/confluence/display/Hive/Home`. Accessed: 12/01/2013.

[3] Iso/iec wd 13249-3 information technology - sql multimedia and application packages - part 3: Spatial 3rd edition. `http://jtc1sc32.org/doc/N1101-1150/32N1107-WD13249-3--spatial.pdf`. Accessed: 12/01/2013.

[4] Jts topology suite. `http://www.vividsolutions.com/jts/jtshome.htm`. Accessed: 12/01/2013.

[5] Osm xml. `http://wiki.openstreetmap.org/wiki/OSM_XML`. Accessed: 12/01/2013.

[6] A. Aji and F. Wang. High performance spatial query processing for large scale scientific data. In *Proceedings of the on SIGMOD/PODS 2012 PhD Symposium*, pages 9–14. ACM, 2012.

[7] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[8] T. White. *Hadoop: The definitive guide*. O'Reilly Media, 2012.

[9] A. Witayangkurn, T. Horanont, and R. Shibasaki. Performance comparisons of spatial data processing techniques for a large scale mobile phone dataset. In *Proceedings of the 3rd International Conference on Computing for Geospatial Research and Applications*, page 25. ACM, 2012.