Spatio-Temporal Hotspot Computation on Apache Spark (GIS Cup)

Paras Mehta
Freie Universität Berlin
Germany
paras.mehta@fu-berlin.de

Christian Windolf
Freie Universität Berlin
Germany
christianwindolf@web.de

Agnès Voisard
Freie Universität Berlin
Germany
agnes.voisard@fu-berlin.de

ABSTRACT

Large quantities of mobility data are produced by people and vehicles daily. Mining and analysis of patterns, such as hotspots, in this data can serve to improve location-based services. However, due to the massive amount of information, efficient techniques are needed for processing it in distributed environments using frameworks, such as Apache Spark. In this work, within the scope of the GIS Cup 2016, we focus on the detection of statistically significant hotspots in large-scale spatio-temporal data using the Getis-Ord G_i^* statistic on top of the Spark framework. Using a uniform spatio-temporal partitioning, we find the most significant drop-off locations in taxi trip data for New York City based on the passenger count. We present a baseline and two variants of an optimized solution for the problem. Finally, we compare and demonstrate the performance of the proposed algorithms through an experimental evaluation.

CCS Concepts

•**Computing methodologies** → **MapReduce algorithms;**
•**Information systems** → **Spatial-temporal systems;**

Keywords

spatio-temporal, mapreduce, spark, hotspot analysis, spatial autocorrelation, getis-ord

1. INTRODUCTION

With the widespread use of GPS devices and smartphones, the amount of data about the movement of persons and vehicles has grown rapidly. Through the discovery and analysis of mobility patterns, this data can provide valuable insights into urban environments that can be used to improve infrastructure and services. One such pattern is a spatial hotspot, where objects within the hotspot tend to exhibit a higher degree of positive correlation as opposed to those outside [4]. Their detection has applications in several fields, such as urban planning, transportation management and criminology. The extent of correlation between objects within the hotspot can be measured using spatial statistics, for example, the Getis-Ord G_i^* statistic. G_i^* provides a local estimate of spatial autocorrelation. It can be used to identify neighborhoods containing regions that exhibit a higher level of similarity with each other than a global mean [3].

As the volume of mobility data grows, distributed computing frameworks, such as Apache Spark[1], are used to efficiently process it. Furthermore, extending existing techniques and developing new methods for spatial data mining and management on distributed frameworks is an active area of research [5][2][1]. Therefore, as part of the GIS Cup 2016[2], the goal of this work is to identify the most significant hotspots based on passenger count in a large collection of taxi trips from New York City by computing the G_i^* score on Spark. Each trip record in the dataset contains various attributes, such as pickup and drop-off times and locations, fare amount, passenger count, and distance traveled. For the computation of the statistic values, space is discretized using a uniform spatial grid and time is split into steps, thus creating three-dimensional spatio-temporal cells. The passenger count values are aggregated per spatio-temporal cell. The neighborhood of a cell is defined as its immediate neighbors that share an edge or vertex with the cell, with each carrying equal weight. As a result, the calculation of G_i^* is a computationally intensive task as each region not only requires its own attribute values, but also those of neighboring locations. This can slow down query performance because of the large amount of data that needs to be exchanged if neighboring cells are present on different cluster nodes.

Given this problem setting, we first propose a baseline approach for finding the top-k cells in the dataset based on passenger count. The baseline uses a simple self-join for finding the spatio-temporal neighbors of each cell in order to compute the scores. However, as expected, this approach performs poorly since the score of each cell depends on the attribute values of other cells which may reside on different nodes. Thus, we propose a new technique where cells are grouped together based on spatio-temporal adjacency. Moreover, some cells are duplicated across multiple groups. This ensures that all the neighbors of a cell are present in the same group as the cell, and hence on the same node. Based on this idea, we present two algorithms, the first containing two *MapReduce* jobs and the second consisting of one. We also propose and integrate other optimizations in

SIGSPATIAL'16, October 31-November 03, 2016, Burlingame, CA, USA

ISBN 978-1-4503-4589-7/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2996913.3004063

[1] http://spark.apache.org/

[2] http://sigspatial2016.sigspatial.org/giscup2016/

our solution to improve performance, including use of better serialization, lightweight data structures and integer encoding. We show that our approach outperforms the baseline by at least an order of magnitude and demonstrate its performance using experiments.

2. PROBLEM DEFINITION

Journey A Journey $j \in \mathcal{J}$ represents a single trip and is defined as $j = \langle loc_s, t_s, loc_e, t_e, p \rangle$, where $loc_s = (lat_s, lon_s)$ and t_s are the starting location and time, $loc_e = (lat_e, lon_e)$ and t_e are the ending location and time, and p is the number of passengers, respectively.

The entire spatial region $\mathcal{R}$ is split into subregions of dimensions $l * l$ using a grid, whereas the entire temporal range $\mathcal{T}$ is divided into steps of duration d. We refer to each region with dimensions $l * l * d$ as a Spatio-Temporal Cell.

Spatio-Temporal Cell A Spatio-Temporal Cell, or simply, a Cell $c \in \mathcal{C}$ is defined as $c = \langle id, p, \phi \rangle$, where $id = \langle x_c, y_c, t_c \rangle$ is a unique identifier, p is the passenger count, and ϕ is the score of the cell, respectively.

We seek hotspots based on number of passengers dropped off inside each cell. Thus, the passenger count of a cell is set to the sum of passenger counts of all journeys with drop-off points inside it, i.e.,

$$c.p = \sum_{\substack{j \in \mathcal{J} \\ j.t_e \in c.\tau \\ j.loc_e \in c.r}} j.p,$$

where r and τ are the spatial region and the temporal interval covered by the cell, respectively.

Score The Score of a cell is equal to the value of the Getis-Ord G_i^* statistic. From [3], recall that G_i^* value for a cell c_i can be defined as:

$$G_i^\star = \frac{\sum_{j=1}^{n} w_{i,j} x_j - \bar{X} \sum_{j=1}^{n} w_{i,j}}{S \sqrt{\frac{[n \sum_{j=1}^{n} w_{i,j}^2 - (\sum_{j=1}^{n} w_{i,j})^2]}{n-1}}},$$

where $w_{i,j}$ is the weight of c_j w.r.t. c_i, x_i is the attribute value of c_i, $\bar{X}$ is the mean of x_i over all c_i, and n is the number of cells.

Here, the attribute value is the cell passenger count, i.e., $x_i = c_i.p$. From the above equation, it is evident that the score of a cell is dependent on the attribute values of itself, as well as those of other cells. By definition, in our problem setting, the size of the neighborhood of a cell is known in advance, namely 1 cell + 26 adjacent cells = 27 cells. Furthermore, each cell is weighted equally. Therefore,

$$w_{i,j} = \begin{cases} 1, & \text{if } adjacent(c_i, c_j) = true. \\ 0, & \text{otherwise.} \end{cases}$$

Top-k Hotspots Given a collection of journeys $\mathcal{J}$ lying inside a spatial range $\mathcal{R}$ and a temporal range $\mathcal{T}$ split into partitions of size $(l * l)$ and d, respectively, retrieve a set of k spatio-temporal cells $C \in \mathcal{C}$, such that,

$$c_i.\phi \geq c_j.\phi \quad \forall\ c_i \in C, c_j \in \mathcal{C} \setminus C.$$

3. METHODOLOGY

Spark is built on the *MapReduce* processing paradigm. The *Map* function is executed in parallel on each record, whereas the *Reduce* phase collects the records by key. Keeping this in mind, below we present a generic approach for our problem that serves as a foundation for explaining the baseline and optimized solutions.

3.1 Generic Approach

A generic approach for the *Top-k Hotspots* problem can be broken down into the following steps.

Cell Creation To compute the cells from the raw data with *MapReduce* is straightforward. In the *map* function, records are read and parsed to create journeys. Each journey is transformed into a tuple containing the cell identifier and the number of passengers that have been dropped off there. The *reduce* phase builds cells by gathering tuples by their cell coordinates and summing up their passenger counts.

Neighborhood Retrieval The implementation of the actual G_i^* computation on a distributed system is more challenging. As discussed earlier, it is necessary for each cell to retrieve its adjoining cells. This can be done via a self-join operation. However, a naïve self-join can result in each node querying for cells simultaneously, which may be located on different nodes. This would increase the network overhead and reduce query performance.

Score Computation Once the values needed to deduce the score of a cell have been gathered, the actual calculation of $G^\star$ becomes trivial. This is done in a single *map* execution and can thus be completed quickly in parallel on each node.

In the following, we present the baseline algorithm `BSL`, as well as an optimized solution `STG` consisting of two variants, namely `STG2` and `STG1`.

3.2 Algorithm BSL

A naïve approach for the *Top-k Hotspots* problem proceeds as follows. Firstly, trip records are read and converted into journeys, that are further aggregated into cells. Next, the neighborhood for each cell is found by doing a self-join on the cell identifier $id = \langle x_c, y_c, t_c \rangle$ to find all cells within a distance of one cell. Finally, the scores for each cell are computed and top-k ones are returned as result. This approach, called `BSL`, serves as our baseline for evaluation.

3.3 Spatio-Temporal Group (STG) Approach

We first describe the Spatio-Temporal Group (`STG`) method that groups cells that belong to the same neighborhood to improve query times. We then discuss two algorithms that use this technique, the first consisting of two *reduce* phases and the second of just one.

The *Cell Creation* and *Score Computation* phases in the `STG` method are the same as those of the generic approach. In the *Neighborhood Retrieval* stage, `STG` makes use of the following optimization.

Neighborhood Retrieval Optimization One way of lowering the time required for finding neighboring cells is to guarantee that the neighbors reside on the same node as the cell itself. In our scenario, since the neighborhood is known in advance, this technique has the potential of reducing query times by minimizing the amount of inter-node data exchange and allowing the score computation to be executed in parallel on each node.

For this, we group cells that are adjacent in space and time into cubes of $s * s * s$ cells, where cells in the same group are guaranteed to be on the same node. However, this has an important limitation. The cells that form the surface of the cube (group) might have their neighbors on a different node. To address this, we replicate the surface

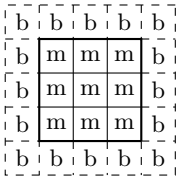

Figure 1: Arrangement of cells in a group.

cells across multiple groups. This is equivalent to creating a buffer of thickness equal to one cell around each group. Thus, although each cell is a member of one and only one group, it can be duplicated across up to 2^n groups, where n is the number of dimensions (three in our case). This strategy trades off higher memory consumption in favor of lower network traffic. The idea of *member* cells and $buffer$ cells is illustrated in Figure 1, where m and b in the Figure denote member and buffer cells, respectively.

More formally, a *Spatio-Temporal Group*, or simply, a *Group* g is defined as $g = \langle id, C_g \rangle$, where $id = \langle x_g, y_g, t_g \rangle$ is a unique identifier and $C_g = \{c_1, c_2, \cdots\}$ is the collection of cells belonging to the group.

General Optimizations `STG` also incorporates some general improvements listed below to speed up query execution.

- *Cell Identifier Encoding.* The identifier of each cell consists of three integer values, one for each dimension. In order to reduce memory consumption, we used a simple bit-shifting based scheme for encoding the three values into a single integer, thus potentially reducing the memory footprint of the identifier by a factor of three. Simply stated, the first one-third of the bits are occupied by the first dimension, the second by the second dimension, and the remaining by the third.

- *Object Serialization.* The serialization and deserialization of objects, including those of Journey, Cell and other classes, is a significant overhead in terms of bandwidth and query time. Thus, in place of the default serializer, we chose the faster and more compact serializer offered by the *Kryo*[3] library.

- *Lightweight Data Types.* In our implementation, custom classes were replaced with primitive data types and data structures offered by Scala to ensure low memory footprint and access times. This includes the use of the *fastutil*[4] library for the implementation of a Hashmap.

In the following, we present two concrete algorithms based on the general `STG` approach.

Algorithm STG2

The `STG2` algorithm consists of two *MapReduce* phases: the first one for the creation of cells and the second for the creation of groups. Aggregation of data into cells is straightforward. First, the records are parsed to create journeys and then grouped by key during *Reduce*. The procedure for creating groups is more complicated. During the map phase, the groups that a cell lies in are found, including the one of which the cell is a member and those where the cell lies in the buffer. Algorithm 1 shows the pseudocode for this routine, whereas Algorithm 2 depicts the second *Map* procedure which uses this routine. The reduce stage builds the groups by collecting all the cells for each group identifier.

[3]https://github.com/EsotericSoftware/kryo

[4]http://fastutil.di.unimi.it/

Algorithm 1: `GetGroupsForCell` routine

```
Data: cell c, group size s
Result: ids of groups the cell lies in G_c
g_m ← ⟨ c.id.x_c / s , c.id.y_c / s , c.t_c / s ⟩   ▷ group where c is member
G_c ← {g_m}
foreach i ∈ {0, 1, 2} do
    G_b ← ∅
    if (c[i] mod s) = 0 then
        foreach g ∈ G_c do
            g_b ← g
            g_b ← g_m[i] − 1
            G_b ← G_b ∪ {g_b}
        end
    end
    if (c[i] mod s) = s − 1 then
        foreach g ∈ G_c do
            g_b ← g
            g_b[i] ← g_m[i] + 1
            G_b ← G_b ∪ {g_b}
        end
    end
    G_c ← G_c ∪ G_b
end
return G_c
```

Algorithm 2: Second *Map* of `STG2`

```
Data: cell c, group size s
G_c ← GetGroupsForCell(c,s)
foreach g ∈ G_c do
    Emit(g,c)
end
```

Algorithm 3: *Map* routine of `STG1`

```
Data: record rec, group size s, grid size l, time step d
j ← ParseRecord(rec)                    ▷ create journey from record
c ← ⟨ ⟨ j.loc_e.x / l , j.loc_e.y / l , j.t_e / d ⟩ , j.p ⟩   ▷ create cell with j
G_c ← GetGroupsForCell(c,s)
foreach g ∈ G_c do
    Emit(g,{c})
end
```

Algorithm STG1

A potential limitation of the `STG2` algorithm is that it executes two separate *MapReduce* tasks for cell and group creation. Hence, there are two different parts of execution where data has to be shuffled between nodes. The algorithm `STG1` attempts to reduce the amount of data shuffled by merging the two *MapReduce* stages into one by mapping journeys to groups in the same *map* function. This is shown in Algorithm 3.

Table 1: Details of the evaluation datasets.

Dataset	Size	# of Trips
Feb 2015	1.8GB	12,450,521
2015	21.3GB	146,112,989

Table 2: Execution times using Feb 2015 dataset.

Algorithm	Runtime
BSL	8.9 hours
STG2	50.3 seconds
STG1	28 minutes

Table 3: Input parameters for experiments.

Parameter	Values
Grid size l (degrees)	0.0005 0.00075 **0.001** 0.005 0.01
Time Step d (days)	1 3 **7** 10 14

4. EXPERIMENTAL EVALUATION

For the evaluation, we ran experiments on a cluster of 8 machines, each with 8 GB of RAM and 4 CPUs, with Spark v1.6.1 and HDFS v2.6.4. We used taxi trips during 2015 from the New York City Yellow Cab dataset[5] as the evaluation dataset. A subset of this data from February 2015 was used to compare the performances of BSL, STG2 and STG1. The datasets were stored as HDFS files on the cluster. Their characteristics are shown in Table 1. The values of s and k are set to 25 and 50, respectively.

4.1 Comparison of Algorithms

We start by comparing the three proposed algorithms using the February 2015 subset. It is evident from the results shown in Table 2 that STG2 outperforms the other two by a wide margin. The poor performance of BSL is caused by heavy network load due to the distributed environment as it does not include any spatio-temporal ordering.

The poor performance of STG1 in comparison to STG2 can be attributed to the following. In case of STG2, the first *MapReduce* job already reduces the amount of data drastically by aggregating trips into cells. Moreover, as the input data is ordered by time, it is likely that journeys belonging to one cell end up on the same node so that cells can be created without shuffling much data. This is not the case for STG1. As groups span longer in duration than cells and since raw trip information has to be shuffled between nodes to create groups, the performance degrades when the groups are built in a single *Reduce* phase.

Due to the poor performance of BSL and STG1 in comparison with STG2, they are omitted from the remainder of the experiments.

4.2 Varying Input Parameters

We also conducted experiments to understand the behavior of STG2 in different settings. The parameter values used are shown in Table 3, with the default values highlighted in bold font.

4.2.1 Grid Size

The results for changing the spatial grid size are shown in Figure 2 (left). As is evident from the plot, the runtime remains largely stable except for low values of grid size where it tends to increase. This is because the number of cells increase for smaller grid sizes and hence the processing time is higher.

[5]http://nyc.gov/html/tlc/html/about/trip_record_data.shtml

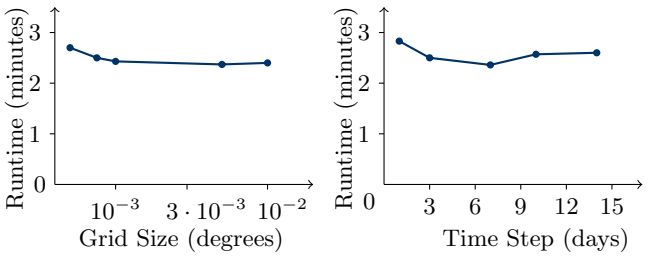

Figure 2: Execution Time vs. Grid Size and Time Step.

4.2.2 Time Step

Figure 2 (right) depicts the behavior of STG2 with a varying time step. As the time step goes from $d = 1$ to $d = 7$ days, the execution time decreases because the number of cells goes down. However, after this, the trend reverses and the runtime increases. This is because the number of cells becomes so low that the groups are only partially filled. Increasing the time step further reduces the execution time because the effect of decrease in the number of cells becomes more dominant.

5. CONCLUSIONS

Spark offers a transparent and efficient way of executing computations on a cluster. However, as our experiments demonstrate, network traffic in distributed settings can have a significant impact on performance. Thus, application-level optimizations are crucial. Certain third party libraries, as well as the Java Standard Library, are either not thread-safe or sacrifice low memory usage in favor of supporting diverse use cases. Thus, re-implementation of generic routines tailored to the problem at hand should be considered. Finally, the importance of understanding the characteristics of the problem and the experimental data cannot be overstated.

Acknowledgments

We would like to thank Dimitris Skoutas for providing us the cluster for conducting the evaluation.

6. REFERENCES

[1] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *International Conference on Scientific and Statistical Database Management*, pages 302–319. Springer, 2009.

[2] Y. Liu, K. Wu, S. Wang, Y. Zhao, and Q. Huang. A mapreduce approach to g i*(d) spatial statistic. In *Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems*, pages 11–18. ACM, 2010.

[3] J. K. Ord and A. Getis. Local spatial autocorrelation statistics: distributional issues and an application. *Geographical analysis*, 27(4):286–306, 1995.

[4] S. Shekhar, M. R. Evans, J. M. Kang, and P. Mohan. Identifying patterns in spatial information: A survey of methods. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(3):193–214, 2011.

[5] R. T. Whitman, M. B. Park, S. M. Ambrose, and E. G. Hoel. Spatial indexing and analytics on hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 73–82. ACM, 2014.