An efficient map-reduce algorithm for spatio-temporal analysis using Spark (GIS Cup)

Salles Viana Gomes Magalhães
Univ. Federal de Viçosa
Viçosa, MG, Brazil
salles@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Inst.
Troy, NY, USA
mail@wrfranklin.org

Wenli Li
Rensselaer Polytechnic Inst.
Troy, NY, USA
liw9@rpi.edu

Marcus Vinicius Alvim Andrade
Univ. Federal de Viçosa
Viçosa, MG, Brazil
marcus@ufv.br

ABSTRACT

We present an efficient parallel algorithm for performing spatio-temporal analysis in a Spark cluster. It divides the spatio-temporal cube into partitions and uses these partitions as the elements of the Spark Resilient Distributed Datasets (RDDs). As a result, data presents a better locality and the overheads are smaller than the overheads observed when single cells are used as elements of the RDDs. When used to find hotspots in the NYC Yellowcab data, our algorithm is up to 52 times faster than algorithms using single cells RDDs.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations

General Terms

Algorithms, Experimentation, Performance

Keywords

Big data, spatio-temporal analysis, distributed processing

1. INTRODUCTION

The availability of big datasets due to advances in data collecting techniques allows the computation of several kinds of spatio-temporal analysis, with applications in crime avoidance, marketing, traffic modeling, etc. However, while the amount of information increases (and, consequently, the quality of the analysis improves) the processing of the data becomes harder. Thus, it is important to develop techniques to efficiently process information.

An example of big dataset is the New York City Taxi and Limousine Commission Yellow Cab trip data [5], containing information about taxi trips in New York City (NYC). Each record in this dataset represents one taxi trip in NYC and contains information such as the length of the trip, the number of passengers, and the pickup and drop-off locations and times.

The public availability of this dataset, its size (more than one billion records), and the importance of NYC public transportation make the Yellow Cab trip data particularly interesting for analysis. For example, Salnikov et al. [7] presented a study where this dataset is analyzed and the costs of taxi trips are compared with the estimated costs of similar Uber trips. Ferreira et al. [3] presented a model that allows users to visually analyze the NYC taxi trip dataset.

In the ACM GISCUP 2016 programming competition [9], the organizers proposed the development of efficient algorithms for performing hot-spot analysis on the NYC taxi trip data. The objective is to develop a map-reduce algorithm that can run on a Spark cluster to quickly (and accurately) find the 50 most statistically significant hot-spots (in both time and space) according to the G_i^* statistic[6].

2. THE PROBLEM

The objective is to process the NYC taxi records, to aggregate the drop-offs into cells according to their time and geographic coordinates. One drop-off may involve several passengers. Define an *event* to be one passenger being dropped off. Since the data may be noisy, the events are filtered to remove events outside a time bound or outside a bounding-box encompassing the five New York City boroughs.

The filtering bounding-box defines a spatio-temporal cube, which is partitioned into cells with given dimensions (the spatial dimensions are given in degrees and the temporal is given in days). The events are then aggregated into these cells.

After the aggregation, each spatio-temporal cell i should store the count c_i of the number of events in that cell. Based on the counts stored in each cell and on statistics such as the average number of events per cell, the G_i^* statistic (a z-

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

DOI: http://dx.doi.org/10.1145/2996913.3004062

score) of each cell i is computed using Equation 1. The cells with highest G_i^* scores will represent the hot-spots while the cells with lowest scores will be the cold-spots.

$$G_i^* = \frac{\sum_{j=1}^{n} w_{i,j} c_j - \bar{c} \sum_{j=1}^{n} w_{i,j}}{S \sqrt{\frac{n \sum_{j=1}^{n} w_{i,j}^2 - (\sum_{j=1}^{n} w_{i,j})^2}{n-1}}} \quad (1)$$

$$\bar{c} = \frac{\sum_{i=1}^{n} c_i}{n} \qquad \text{and} \qquad S = \sqrt{\frac{\sum_{i=1}^{n} c_i^2}{n} - \bar{c}^2} \quad (2)$$

In the above equations, n is the total number of cells in the area of analysis and $w_{i,j}$ is the weight between two cells i and j. The analysis proposed by the GISCUP competition organizers [9] sets $w_{i,j}$ to a constant, say $w_{i,j} = 1$, when the cells are neighbors in the L^∞-norm, and 0 otherwise. Two cells are considered neighbors iff the maximum distance between i and j in any of the 3 coordinates (longitude, latitude and time) ≤ 1. A cell is also its own neighbor.

Let ν_i be the number of neighbors of cell i. Then $\nu_i = 27$ for cells in the interior of the cube, and 8, 12, or 18 for cells on the boundary. Let

$$\sigma_i = \sum_{j=1}^{n} w_{i,j} c_j$$

i.e., the number of events in a $3 \times 3 \times 3$ box centered on cell i. This is the only term of Equation 1 that depends on the number of events in cell i. The other terms depend only on statistics about the input data (such as average and standard deviation of the number of events in each cell), or on ν_i. Thus, the main challenge for computing G_i^* is computing σ_i, G_i^* can be computed as follows.

$$G_i^* = \frac{\sigma_i - \bar{c}\nu_i}{S \sqrt{\frac{n\nu_i - \nu_i^2}{n-1}}}$$

3. MAPREDUCE AND SPARK

MapReduce is a parallel programming model that has recently been used in architectures ranging from multi-core computers to GPUs and clusters. The concept has been obvious for decades; the Thinking Machines CM-2 Connection Machine implemented parallel reduction in 1990 [8].

Applications that can be modeled using MapReduce usually share a common pattern: they require a series of operations that can be performed independently (map), followed by an operation that combines the individual results (reduce) [2].

The main advantage of this model is that it abstracts and simplifies the development of parallel applications. As a result, it has been recently used to not only allow the development of efficient parallel algorithms but also to improve the productivity of the programmers, who do not need to directly deal with low-level details inherent of the different parallel programming architectures. For example, the development of parallel algorithms for CUDA-enabled GPUs can be simplified using the Thrust library [4], that uses several MapReduce concepts to abstract the development of these algorithms. Another advantage of this abstraction is the interoperability: algorithms implemented using Thrust can not only be compiled for GPUs but also for multi-core computers, two completely different architectures.

In fact because of its modest size, cloud-based techniques are not necessary to process this dataset. The 1,000,000,000 drop-off events in the complete 7 year dataset (of which this contest used only one year), if encoded into binary at 6 bytes per event, would fit into the 32GB main memory of a high-end multi-core laptop. Then the G_i^* scores could be calculated in parallel with Thrust using OpenMP as a backend. By far the slowest part of the process would be reading the 200GB of ASCII files to extract the drop-off data. The actual statistical computation would be relatively trivial. Nevertheless, a distributed file system would improve the performance. Also, for datasets too large to fit even into the 2TB main memory of a serious compute server, a distributed architecture like Apache Spark would be required.

"Apache Spark is a fast and general-purpose cluster computing system" [1]. It provides APIs in languages such as Java and Scala for the development of Map-Reduce applications. By presenting an efficient processing engine and performing in-memory computation, Spark's applications can usually present good performance and scalability.

The most important concept in Spark is the RDD - *Resilient Distributed Dataset*, a distributed data structure that abstracts a dataset stored across the Spark cluster. It can be created, for example, from an input dataset stored in HDFS (a distributed file system). Furthermore, RDDs can be created from transformations of other RDDs. For example, if a RDD A stores integers and a transformation (mapping) operation is performed such that each integer is squared, then the square operation will be applied in parallel to all elements from A and a new RDD B will be created such that the elements in B represents the elements in A squared.

RDDs also support actions, i.e., operations that return a single value (instead of a new RDD). Example of these operations are counting the number of elements in the dataset or performing a reduction operation (such as finding the maximum element).

Finally, it is possible to create RDDs of *key-value* pairs. These special datasets support some operations such as the *reduce by key* transformation that uses a reduction operation to aggregate values sharing the same key. The algorithms presented in this paper use several of these functions to efficiently compute the hot-spots.

4. OUR SOLUTIONS

4.1 Simple MapReduce solution

After many preliminary experiments testing assorted data structures and algorithms, we developed the following MapReduce algorithm using Java and Spark:

1. Read the input file from HDFS, compute the cell coordinates of each drop-off, and create a RDD containing (t, x, y) as key and *numPassenger* as value. (t, x, y) represents the cell coordinates of the drop-off and *numPassenger* is the number of passengers dropped off at that event.

2. Perform a reduction by key operation to create a RDD *counts*. In this RDD, for each key (coordinates (t, x, y)) the value will be the total number of events at that coordinate (*count*).

3. Compute statistics such as the average and standard deviation of the number of events in each cell using the previous RDD.

4. Create a new RDD by generating 27 versions of each cell $i = ((t, x, y), count)$, each with the same *count*, but the following various coordinates $((t + dt, x + dx, y + dy), count)$, for $dt = -1..1, dx = -1..1, dy = -1..1$.

5. Reduce this new RDD by key, creating an RDD *sumNeigh* that will contain, for each cell i, σ_i, the sum of the number of events in the neighbors of i.

6. Finally, from the statistics in step 3, compute the G_i^* score of each cell in the RDD *sumNeigh* and write the cells with highest scores to the output.

Several optimizations were performed to improve the algorithm's performance. First, a typical algorithm implemented in Java or Scala usually parses the input dataset by using the Spark function *map* to transform each line of the input into a parsed object. The drawback of this solution is that it is necessary to allocate one parser object for each input line. For performance, we used the *mapPartitions* function, that allows the re-use of the parser in all lines of a RDD partition. Furthermore, we implemented a custom parser specifically designed to parse the Taxi trip data. One of the advantages of this custom class is that it does not create wrapper Java objects (Integers or Doubles) such as the default java parser does. Our parser, instead, uses primitive types (such as ints and doubles) that are faster and uses less memory.

Second, custom classes were created to represent all the data required by our algorithm. For example, the coordinate of cells could be represented using a *Tuple3<Integer,Integer, Integer>* (available in the Spark API) but, instead, we created a custom class storing 3 primitive *ints*. The *Tuple3* object needs 3 pointers to reference the Integer wrappers it stores (the memory used by the pointers will usually be larger than the memory used to store the contents of the tuple), while our custom class does not need any pointer.

Third, if the size of the spatio-temporal cube is small enough, we optimized by representing each coordinate triple with one 4-byte integer, instead of with three integers. Since a single integer can represent 2 billion different numbers (ignoring negative numbers), if the grid resolution is, say, $36 \times 2000 \times 2000$ any cell in this grid can be uniquely represented using a single integer. This strategy not only saves space but also makes the algorithm faster. Indeed, comparing two cells keyed by an integer is faster than comparing two cells with keys represented by 3 coordinates.

Finally, several configuration parameters present in Spark were used to try to improve the performance. For example, the Kryo serializer [1] was employed to serialize objects faster than the Java default serializer. Furthermore, the hash shuffler [1] was employed to partition the data when information needs to be shuffled across the cluster.

4.2 Improved solution

Preliminary experiments showed that the bottlenecks of the above algorithm are the fourth and fifth steps, where a list with size equal to approximately 27 times the number of cells in the spatio-temporal cube is created and reduced by key.

We developed an improved version of the previous algorithm, which tries to accelerate this process by storing, as elements of the RDDs, a matrix representing a partition of the spatio-temporal cube. Figures 1, 2 and 3 illustrate this strategy for representing the number of events in each cell (shown in 2D for simplicity): in the algorithm described in Section 4.1 (Figure 2), each element of the RDD represents a cell of the spatio-temporal cube while in this new strategy (Figure 3) each element represents a partition of the cube.

This strategy has several advantages. First, it is more compact and has less overhead. Instead of explicitly storing the coordinates of each cell, only the coordinate of the partition of the cube is stored explicitly. Second, since each element of the RDD stores a partition of the cube (with several cells), the RDDs will have fewer elements (but each element will be larger).

Finally, and most importantly, σ_i can be computed faster. Since the neighbors of most of the cells are stored in the same element, a RDD representing *sumNeigh* can be created by simply processing the partitions of the spatio-temporal cube, and for each partition, adding (locally) the count in each cell to its 27 neighbors (most of the neighbors will be in the same matrix). The only special case that needs to be treated are the cells on the boundary of the partition. Since these cells will have some neighbors in other partitions, an aggregation operation will be necessary to add the count of these boundary cells to some of their neighbors.

5. EXPERIMENTS

We performed experiments with a Spark cluster created on Amazon EC2, using an instance of 25 *m2.2xlarge* nodes. Each node is composed of a 4 virtual CPUs, 34.2 GB of RAM and a 850 GB rotational hard drive.

In these experiments, the same dataset used to evaluate solutions in the GISCUP contest was processed. More specifically, the whole taxi data for 2015 (containing about 150 million records) was stored in HDFS, and our algorithms were employed to compute the hot-spots considering different spatial and temporal resolutions.

Table 1 presents the results (all times are in seconds). Column *Spat. size* and *Time size* are, respectively, the spatial (in degrees) and temporal (in days) size of the grid cells. E.g., a cell with width 0.0001 degrees represents an 8x11 meter rectangle.

Column *Simple* presents the time obtained by our simplest algorithm (described in section 4.1) using Spark's default configuration. The other algorithms are described by the combination of the following optimizations employed in the implementation:

- K: the Kryo serializer was employed instead of the standard Java serializer.

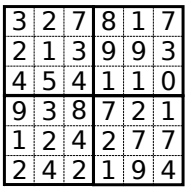

Figure 1: Spatio-temporal grid

(0,0),2;(1,0),4;(2,0),2;(3,0),1;(4,0),9;(5,0),4;(1,0),1; ...

Figure 2: Each element represents a cell

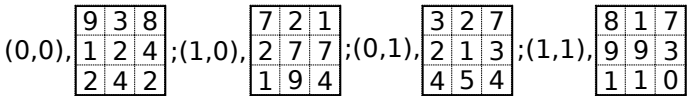

Figure 3: Each element represents a partition of the cube (matrix in 2D)

Table 1: Running-times (in seconds) for different combinations of techniques to accelerate the hot-spot analysis and for different resolutions of the spatio-temporal cube

Spat.	Time	Algorithm			
size	size	Simple	K+H	K+H+C	K+H+CP
0.01	30	18	20	19	21
0.01	7	18	20	19	19
0.01	1	26	20	20	19
0.001	30	20	20	19	25
0.001	7	27	23	22	22
0.001	1	37	27	24	24
0.0003	30	39	27	22	28
0.0003	7	65	42	25	25
0.0003	1	114	70	51	33
0.0001	30	84	61	30	25
0.0001	7	321	212	49	27
0.0001	1	675	455	-	33
0.00005	30	229	147	45	28
0.00005	7	1102	712	-	31
0.00005	1	1921	1318	-	37

- H: the Hash shuffler was employed instead of the default shuffler.
- C: the coordinates of the cells were coded together in a single integer. A dash shows when this was not possible because there were more than 2^{31} cells.
- CP: the spatio-temporal cube was partitioned and each element of the RDDs stored a partition of the cube (instead of a single cell). Based on preliminary experiments we always used partitions with 10^3 cells.

As can be seen, the first three algorithms in the Table 1 work well for lower resolution grids (since, in these situations, the RDDs contain few elements and, thus, I/O dominates the time). However, as the resolution increases the difference between the different versions of the algorithms increases and the more sophisticated algorithm that uses as elements of the RDDs, partitions, the spatio-temporal cubes becomes much faster than the other ones, being up to 52 times faster than the simplest algorithm.

Considering the algorithms that use single cells as the elements of the RDDs (first 3 algorithms on Table 1), in general, the fastest one is the one that codes the coordinates in single integers. This can be explained because this representation is more compact than the representation that stores the 3 coordinates in separate integers (and, therefore, uses 3 times more space to represent the coordinates than to store the integer representing the number of events in each cell). Besides being more compact, comparison and hashing operations can be performed faster on single integers than on triples of integers.

It is also interesting to observe that the simplest algorithm can be easily improved by simply changing some Spark configurations. Indeed, the use of a hash shuffler associated with the Kryo serializer reduced the processing time of the simple algorithm in up to 36%.

6. CONCLUSIONS

We have presented efficient map-reduce algorithms for performing hot-spot analysis on the New York Taxi dataset. Initially, we developed a simple map-reduce algorithm, and then employed a series of techniques to further improve this algorithm. As a result, we achieved a speedup of up to 52 times over the simplest algorithm for the highest resolution grid evaluated.

Acknowledgment

This research was partially supported by NSF grant IIS-1117277, by CAPES (Ciencia sem Fronteiras) and FAPEMIG.

7. REFERENCES

[1] Apache. Spark - lightning-fast cluster computing. https://spark.apache.org/ (accessed on Sep-2016).

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[3] N. Ferreira, J. Poco, H. T. Vo, J. Freire, and C. T. Silva. Visual exploration of big spatio-temporal urban data: A study of new york city taxi trips. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2149–2158, Dec. 2013.

[4] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[5] NYC Taxi and Limousine Commission. TLC trip record data. http://www.nyc.gov/html/tlc (accessed on Sep-2016).

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

[7] V. Salnikov, R. Lambiotte, A. Noulas, and C. Mascolo. Openstreetcab: Exploiting taxi mobility patterns in new york city to reduce commuter costs. *CoRR*, abs/1503.03021, 2015.

[8] Thinking Machines Corp. *Connection Machine Model CM-2 Technical Summary, version 6.0*, Nov. 1990. `http://people.csail.mit.edu/bradley/cm5docs/nov06/ConnectionMachineModelCM-2TechnicalSummary.pdf`.

[9] R. R. Vatsavai, E. Hoel, M. Werner, R. Whitman, and M. Park. GISCUP - ACM SIGSPATIAL CUP 2016. http://sigspatial2016.sigspatial.org/giscup2016/home (accessed on Sept–2016).