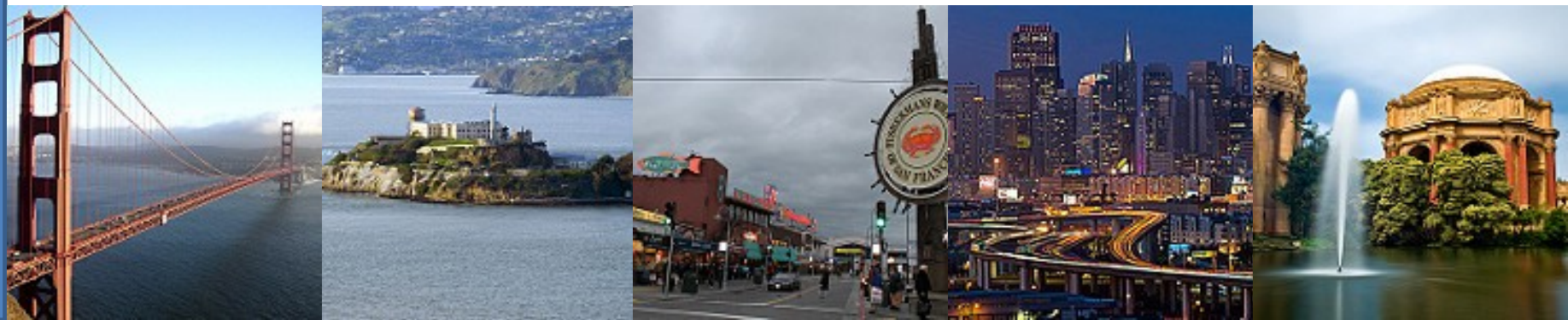
GISCUP - ACM SIGSPATIAL 2016

Rensselaer Polytechnic Institute
Universidade Federal de Viçosa

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

Prof. Dr. W Randolph Franklin, RPI
Salles Viana Gomes de Magalhães, PhD. Student
Wenli Li, PhD. Student
Prof. Dr. Marcus V. A. Andrade, UFV



NYC taxi trip dataset

- NYC Taxi and Limousine Commission (TLC)
- > 1 billion records
- Since 2009

- Several CSV files containing:
  - Drop-off lat/long/time
  - Pick-up lat/long/time
  - Number of passengers
  - Trip distance
  - Fare
  - Payment type
  - Tolls
  - Etc.

Source: http://www.nyc.gov

NYC taxi trip dataset

- Big amount of information → many possibilities of analysis

- Example:
  - What is the average price of trips from JFK to LGA?
  - Is the most used type of payment different for different neighborhoods/days/hours?
  - What is the most frequent destination from Penn Station?
  - Hotspots for full taxis?
- Some interesting observations (2015 dataset).
  - Most frequent fare: $7.80 (3,804,101
  - 195 trips cost more than $1,000 (noise?)
  - # trips costing (0,$1] : 35,893
  - Average # of passengers per trip: 1.6

Source: http://www.nyc.gov

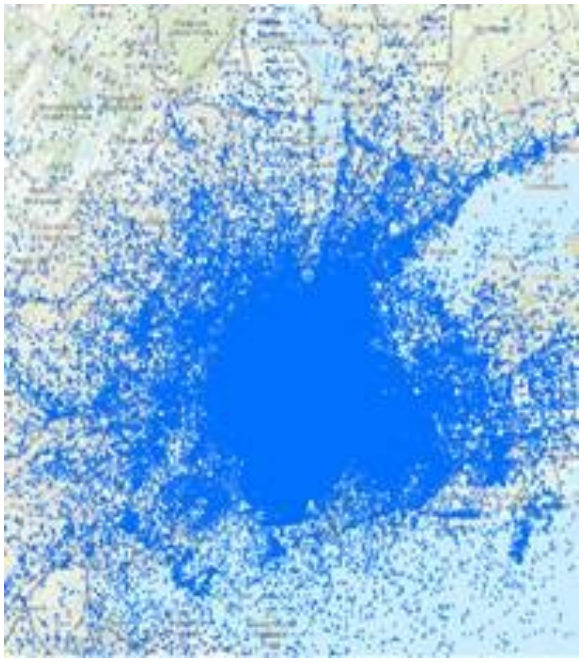
NYC taxi trip dataset

- Has some errors.

Pickup time	Drop-off time	Dist. (miles)	Pickup long.	Pickup lat.	Fare (USD)	Tip (USD)
03/29/2015 00:**14:32**	03/29/2015 00:**28:41**	**1030**	0.0000	0.0000	**4800.21**	0
09/20/2015 21:50:26	09/20/2015 21:52:57	1.2	0.0000	0.0000	**8004.5**	**1200.8**
11/25/2015 08:24:50	11/25/2015 08:51:53	7.5	-73.8705	40.7736	**93977.3**	**0**
12/27/2015 03:45:54	12/27/2015 03:45:54	0	-73.9289	40.7061	**825998.61**	**0**
01/18/2015 19:24:15	01/18/2015 19:51:55	5.3	-74.0021	40.7395	22	**3950588.8**

GISCUP 2016

- Given the 2015 dataset → what are the top 50 spatio-temporal hotspots?
- Consider the number of passengers being dropped-off.
- Clip the dataset to eliminate noise; consider only the 5 boroughs – remove dropoffs in the Atlantic Ocean.
- Filter drop-offs that happened in 2015. (e.g. remove New Years' Eve)



Source: GISCUP 2016

Hot-spot analysis

- Filtering bounding-box (5 boroughs, only 2015) → spatio-temporal cube.
- Aggregation: cell i → compute c_i (# passengers dropped-off)
- We can optionally always assume that $c_i = 1$.
- Compute the Getis-Ord G_i^* statistic.
- It's a z-score measure of statistical significance.

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

$$\bar{c} = \frac{\sum_{i=1}^{n} c_i}{n} \qquad S = \sqrt{\frac{\sum_{i=1}^{n} c_i^2}{n} - \bar{c}^2}$$

Map-reduce history

- **Map-reduce: Functional Programming** (FP) concept.
- FP applies nested functions to sets of data..

- **Reduction:** component of high level languages at least since the APL language, proposed in 1957, operated on vectors and arrays.
- Implemented by IBM in 1965, widely used for some years.

- Thinking Machines CM-2 (Connection Machine 2) implemented hardware reduction in 1990.

- Google adopted map-reduce in 2004.

- Map-reduce is implemented well in parallel libraries like OpenMP and CUDA/Thrust.

Parallelism

- Serial processors have scarcely gotten faster in 5 years.
- It's a physics problem; must go parallel.
- Or use more efficient algorithms, or make a fundamental discovery.

- Communication dominates computation: keep the data close.

- 1st choice: large amounts of memory *(2TB workstations exist)*.
- Plus multicore Intel Xeon CPUs. *(I have a dual 14-core machine)*.

- 2nd choice: Nvidia GPUs
  - 1000s of slow CUDA cores. (20 CUDA cores = 1 Xeon core)..
  - Very complicated programming.

- 3rd choice: more distributed systems.

Hot-spot analysis

- $w_{i,j}$: 1 for neighbor cells, 0 for other pairs
- Interior cells: 26+1 neighbors; border cells: 8, 12, or 18.
- Let v_i be the number of neighbors of i.
- We can optionally always use v_i =27.
- Let **the sum of neighbors** be

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

- Computing that is by far the hardest step.
- Then

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

Computing the "sum of neighbors"

"cell" coordinates

- Simple map-reduce algorithm
  - First step:
    1. Read files from HDFS and create a pair RDD: ((t,x,y),drop_off)
    2. Reduce by key
    3. Now: {((t,x,y) , c)}
    4. Compute statistics
  - Second step (sum of neighbors):
    1. Each cell ((t,x,y) , c) → {($(t+d_t, x+d_x, y+d_y)$, c), $d_t, d_x, d_y = -1..1$}
    2. Reduce by key
    3. Now: {((t,x,y) , σ)}
  - Third step:
    1. Compute G_i^* using σ/statistics
    2. Get top cells
  - Implemented in Java+Spark

Optimizing the coordinate representation

Optimization:

- Java Tuple3<Integer,Integer,Integer> (generic class) internally uses three pointers to reference the integers.
- Big overhead for a class that simply stores three integers.
- Customized classes avoid pointers.
- Therefore we create a custom class with 3 integers

Optimizing the coordinate representation

Optimization:

- Represent "small coordinate tuple" as a single integer.
  - $(t,x,y) \rightarrow t \; x \; (1+MAX_y) \; x \; (1+MAX_x) + y \; x \; (1+MAX_x) + x$
  - Easily handles 2G cells (4G would be possible).
  - *Note that ARCGIS space-time cubes have same limitation.*
  - Saves memory and I/O.
  - Faster comparison/hashing.
  - We also implement the general and slow data structure in case the user wants over 2G cells.

Computing the "sum of neighbors"

- Reading and parsing is the slowest step.
  - Use a custom parser (up to 2x speedup)
  - Recommendation: store the data in binary, not CSV.

- Spark configuration:

  - Kryo serializer
    - Convert between internal Java object and byte stream for file.
    - Many systems use it.
    - It trades security for speed, but that's ok here.
  - Hash shuffler
    - To get the data from the mapper to the reducer.
    - Various shufflers are available.

Computing the "sum of neighbors"

- Most important optimization:
  - create an RDD with 27 times # of cells
  - Each cell ((t,x,y) , c) → {($(t+d_t, x+d_x, y+d_y)$, c), $d_t, d_x, d_y = -1..1$}
  - Before: element type stored in the RDD is a cell
  - Now: each element is a partition of the space-time cube.

3	2	7	8	1	7
2	1	3	9	9	3
4	5	4	1	1	0
9	3	8	7	2	1
1	2	4	2	7	7
2	4	2	1	9	4

  - From:

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

  - To:

(0,0),

9	3	8
1	2	4
2	4	2

;(1,0),

7	2	1
2	7	7
1	9	4

;(0,1),

3	2	7
2	1	3
4	5	4

;(1,1),

8	1	7
9	9	3
1	1	0

Computing the "sum of neighbors"

- Improved solution:
  - More locality:
    - Sum of neighbors: *for* loop to add elements
    - Cells in boundary: special case (separate list and aggregate)
  - Fewer (but larger) elements:
    - Less overhead (implicit coordinates)

3	2	7	8	1	7
2	1	3	9	9	3
4	5	4	1	1	0
9	3	8	7	2	1
1	2	4	2	7	7
2	4	2	1	9	4

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

Failed idea: sample the data

Before settling on the previous algorithm, we tested several other ideas, but they were bad.

Failed idea: sample the data.

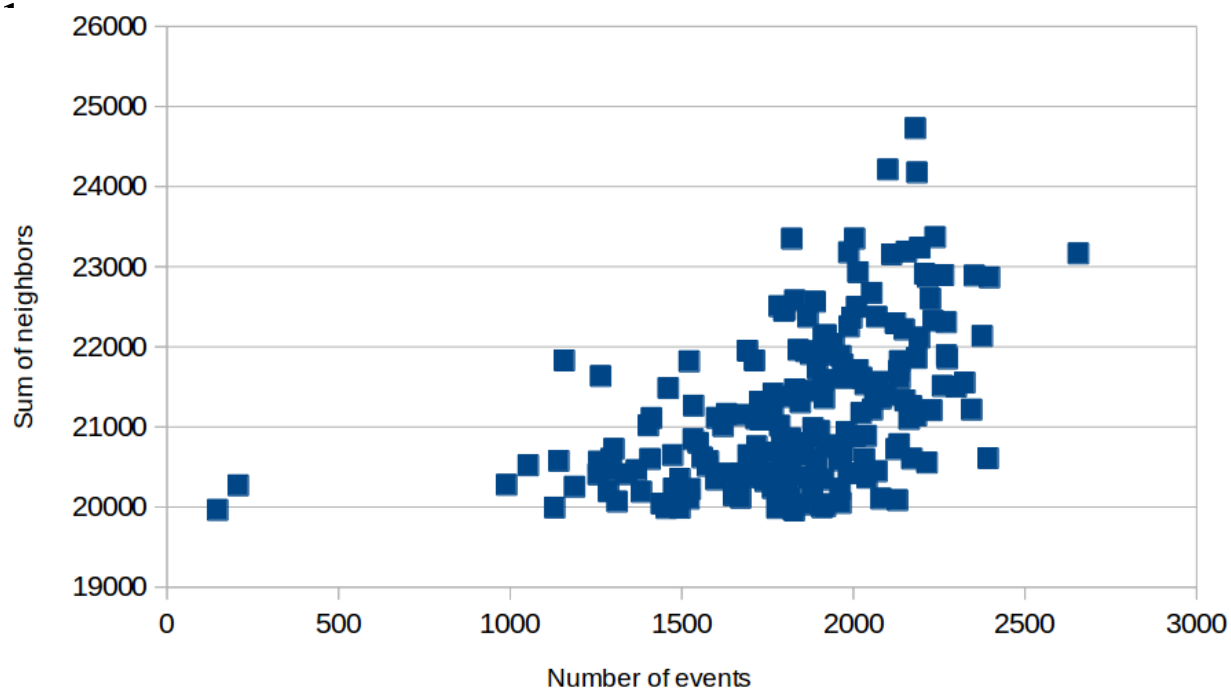
- Sampling large datasets is a common operation in statistics.
- Relative error falls with sqrt(sample size).
- So: Compute the hot spots for only some of the data.
- However: the list of hot spots changes.
- Fail.

Failed idea: allocate complete array of counts

- Failed idea: Instead of storing ((t,x,y), count) tuples, allocate a complete array to store the counts.
- However most of the array would be 0.
- This would take more space (= more time).

Failed idea: count neighbors for only hot cells

- Perhaps a cell with high G_i* will itself have many events.
- That is, regions of many events are several cells wide.
- So, count number of events for all cells.
- Pick the top 1000 cells.
- Count neighbors and compute G_i* for only them.
- However, this doesn't always work.
- Some cells are like a donut hole.
- Plot: 2015 top 50, res: 0.001°, 1 day.



Experiments

- Amazon EC2

- 25x (1+24) m2.2xlarge
  - 4 CPUs
  - 34.2 GB of RAM
  - 850 GB rotational hard drive

- 2015 dataset (stored in HDFS)

- Algorithms/improvements:
  - Simple algorithm
  - K: Kryo serializer
  - H: Hash shuffler
  - C: Compressed coordinates
  - CP: Cube partitioning

Experiments

Spat. size	Time size	Algorithm			
		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

Experiments

- 0.0001°→ ~ 8m x 11m
  - 0.0001°, 1 day→Bounding-box with: 4000 *x* 5500 *x* 365 - 8G cells!
  - ~146 million trips in 2015
  - ~144 million filtered events (bounding-box)
  - → 0.02 trips per cell, 0.03 drop-offs/cell
  - Hotspots:
    - long, lat, days from 1/1/2015, z-score, sum_neighbors
    - -73.9913,40.7501,352,1427.6,4207
    - -73.9913,40.7501,353,1413.1,4164
    - -73.9912,40.7502,114,1324.5,3903
    - -74.0001,40.7585,11,1323.1,3899
    - -73.9913,40.7501,114,1314.3,3873
    - -74.0000,40.7586,11,1311.9,3866
    - -73.9913,40.7502,352,1307.2,3852
    - -73.9912,40.7502,93,1306.8,3851
    - -73.9912,40.7502,353,1305.8,3848
    - -73.9913,40.7501,354,1303.8,3842

Conclusions

- Spark Map-Reduce:
  - Simple to implement
  - Can achieve good performance

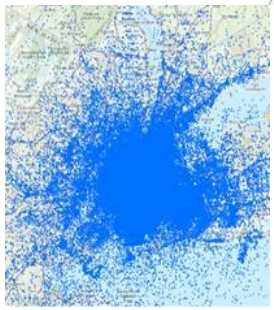
- Best optimizations:
  - Compact coordinates
  - Cube partitioning

  Up to 52x faster than simplest algorithm

Thank you!

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

3	2	7	8	1	7
2	1	3	9	9	3
4	5	4	1	1	0
9	3	8	7	2	1
1	2	4	2	7	7
2	4	2	1	9	4



Spat. size	Time size	Algorithm			
		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

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

(0,0),
9	3	8
1	2	4
2	4	2

;(1,0),
7	2	1
2	7	7
1	9	4

;(0,1),
3	2	7
2	1	3
4	5	4

;(1,1),
8	1	7
9	9	3
1	1	0

Acknowledgement:

Contact:
Salles Viana Gomes de Magalhães, vianas2@rpi.edu
W Randolph Franklin, mail@wrfranklin.org
Wenli Li, liw9@rpi.edu
Marcus V. A. Andrade, marcus@ufv.br