Finding All Co-occurrences from Spatiotemporal Data in an Efficient and Scalable Manner

Presented to the S. Daniel Abraham Honors Program

in Partial Fulfillment of the

Requirements for Completion of the Program

Stern College for Women

Yeshiva University

April 28, 2022

Lily Polonetsky

Mentor: Professor Alan Broder, Computer Science

Abstract:

People are constantly moving around, coming and going from one place to another. People interact with friends, coworkers, and strangers on a daily basis. This data - that is a person's location at a given time - has the potential to tell much about the person's social connections. This has broad-ranging applications: it has applications in fields such as epidemiology, criminology, and marketing. Previous work on computing co-occurrences from spatiotemporal data to infer social ties has been mostly mathematical and has not focused on achieving computationally efficient results. In addition, many of the approaches have relinquished finding every co-occurrence in order to compute things in an efficient and scalable manner. This work sets out to discuss a scalable and computationally efficient approach to compute all co-occurrences from a spatiotemporal dataset for the purpose of inferring social ties.

Introduction:

The nature of the data

Spatial temporal data - that is data in which each data point represents a unique instance of time in a specified location - can tell us much about people's movement and relationship with one another. If person A and person B both visit the same coffee shop around the same time, go to the same movie theater around the same time, and visit the same park around the same time, a logical conclusion would be that person A and person B know each other. Computer scientists and mathematicians discuss the number of co-occurrences as well as the nature and length of the co-occurrences that can lead to the logical deduction that the two people know each other, but that is beyond the scope of this paper.

The problem of inferring co-occurrences from spatiotemporal data is quite simple if the data set only contains a small amount of data; Each data point can be compared to every other data point, allowing the overlapping data points to be found. However, as the data grows larger, it becomes harder to find occurrences because each person and their various location and time instances must be compared to all the other people and their various location and time instances. This approach, called the brute force approach, would require comparing each data point to all the other data points, making it an $O(n^2)$ problem.

While finding co-occurrences in a small dataset is relatively simple if using the brute force method, a small dataset likely does not reflect reality- making the results meaningless. A small dataset likely does not reflect people's movements down to the minute. After all, a single person visits multiple places a day. Multiply that by the tens of thousands, if not the hundreds of thousands of people that live in the area that the dataset is focusing on, and a simple dataset can have hundreds of thousands of data points for just a single day. A small dataset is likely lacking enough data points to allow for all co-occurrences to be found.

Dealing with large datasets in an efficient manner is a relatively new problem. Up until 2005, processors in computers improved every year. Each year, computers were able to process many more instructions than the previous year. This allowed applications to work faster without making any changes to the code. Memory, too, has continually improved. The price for 1 terabyte of data has dropped approximately twofold every fourteen months. However, unlike processing power, memory has continued to improve past 2005. This has allowed more and more data to be collected and stored. However, the improvements in processing power required to

analyze and make computations has not kept up with the improvements in memory. This has left software developers with the challenge of analyzing the data that is being stored in the inexpensive memory with processors that cannot handle the amount of data that needs to be processed. In response, parallel processing as a programming model became popularized to allow the vast amounts of data that are being collected to be analyzed. It is in this context that the need for something like Spark emerges [1].

History of Spark

Over the course of 2003 to 2006, Google released three papers that detailed an approach they had been developing to deal with big data in a scalable and efficient manner. The first paper, titled *The Google File System*, detailed a storage system that was fault tolerant and spread the files across many computers. In storing the files across many computers within a cluster, Google pooled together the resources of each of these computers and created a scalable storage system that could handle vast amounts of data. In 2004, Google released a paper detailing MapReduce, a parallel programming model for working on data stored distributively. In addition to creating a framework for dealing with the data that is stored on the Google File System and BigTable, MapReduce also implemented a framework to speed up the data processing by pushing the computational work to where the data is residing, ensuring that less time is wasted bringing the data to the application to do the processing. In 2006, Google released a paper detailing BigTable, a distributed storage system for structured data. While these three papers did not discuss the implementation of these systems in detail and left much as proprietary to Google, these papers inspired the people at Yahoo and the open-source community to work on creating systems to deal with big data.

The Hadoop File System (HDFS) was inspired by Google's work with the Google File System, BigTable, and MapReduce. While the Hadoop File System was similar in nature to Google's work, the Hadoop File System was open-source, giving everyone use. It was donated to the Apache Software Foundation in 2006 and became a part of a framework of modules called Apache Hadoop. It incurred a large following; Its use became widespread and a large community of open-source developers helped maintain and grow it.

While Hadoop helps developers deal with big data in a scalable and efficient manner, it has many shortcomings. Firstly, it has a steep learning curve. The initial code to set up the job and get it running is verbose, often leading to mistakes in the code which causes the jobs to fail. Hadoop MapReduce only works for general batch processing so writing code involving machine learning or streaming requires adding in other systems, learning a whole other API, and custom cluster configurations. Secondly, Hadoop jobs with many pairs of MapReduce tasks could take days on end to complete. The result from each pair of tasks is written to the local disk, and only then does the program move on to the next pair of tasks. The disk input and output work that is done in between each pair of tasks adds a lot of time to the job. Really, most of the time, the computations are about to be changed by the next pair of tasks, making the input and output work futile. While Hadoop is efficient for some use cases, it falls short in its complexity, making it more difficult to expand its use to a variety of processing jobs and leading to a smaller following of developers.

Recognizing the need for a simpler - yet more flexible and versatile framework - researchers at what was then the RAD Lab, which then became the AMPLab and is now known

as RISELab, worked to create Spark. The idea behind Spark was to create a unified system to process and perform computations with large amounts of data in a parallel manner. Spark was created to be highly fault tolerant, ensuring that the system continues to operate even on failure, and to be embarrassingly parallel, meaning that the tasks in the job are broken up in such a way that they can be performed at the same time. The implementation and design of Spark makes it significantly faster - upwards of ten to twenty times - than Hadoop jobs.

The design of Spark is centered around four design principles: speed, ease of use, modularity, and extensibility. Spark's emphasis on speed is seen both in its use of the hardware and the software. Spark takes advantage of the CPU and memory it is given access to and utilizes the Unix system's support for multi-threading and parallel processing. In addition to taking advantage of the hardware, Spark also builds a directed acyclic graph to determine the quickest and best way to parallelize the jobs and uses a physical execution engine to generate compact code. For ease of use, Spark uses an abstracted model called Resilient Distributed Dataset, or RDD for short, on which all actions and transformations are done. Spark is modular in that it supports numerous programming languages and offers a unified and well-documented API under which numerous types of workloads can be done. Spark is extensible in that it allows use of many different storage sources. Data can be read into memory from Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, Apache Kafka, Kinesis, Azure Storage and Amazon S3 among others [2].

Applications

Inferring social ties from spatiotemporal co-occurrences has wide-reaching applications. For example, marketers would benefit from knowing about a person's connections and the places they frequent in order to better target their audience. If two people continually shop or dine together, a marketer might want to send a buy one get one (BOGO) coupon to them in order to encourage them to return to the same store or to try out a new store. Similarly, if a marketer knows that a person likes a specific restaurant, the marketer should send an advertisement or coupon for that restaurant to the person's associates. Seemingly, the person's associates are more likely to be interested in the restaurant.

Inferring social ties from spatiotemporal co-occurrences can also have applications in the criminal and epidemiological fields. If one person is known to be a part of a larger group of terrorists or criminals, it would be helpful to figure out who they frequently meet up with in order to determine who might be a part of their gang or network. In addition to adding new names to the suspect list, being able to determine who the person meets up with can guide detectives and agents on who they should be allocating more personnel and resources on to further investigate. This can be crucial when time is of the essence or resources are scarce. In the case of epidemiology, being able to trace a person's close contacts in a time efficient manner can be crucial when a disease is spreading rampantly. It can help the researchers see how fast the disease is spreading and notify the person's close contacts.

Previous Work:

Elisheva Muskat in *Computational Efficiency in Inferring Social Ties from Spatiotemporal Data* provides a literature survey from the past two decades of prominent

methods of computing co-occurrences from spatiotemporal data [3]. In her analysis, she discusses the computational efficiency and drawbacks of each approach. She first divides the different approaches that researchers have taken to extract co-occurrences from spatiotemporal data into four categories: the brute force method, partitioning the surface of the Earth into grid-like cells, using discrete check-in locations, and making use of k-d trees. After comparing the different articles, she concludes with suggestions for future research in the space. Muskat suggests that more work should be done to empirically measure the different methods, most notably and relevant to this research, how to deal with the problem of co-occurrences being lost in the surface partitioning method.

We briefly summarize here these four approaches, with particular emphasis on their relevance to the research described in this paper.

Approach 1: the brute force method

The problem with the brute force method is that it relies only on computing power. This makes the method unsuitable for large datasets as it is not scalable. The brute force method relies solely on computing power due to the nature of the algorithm. The brute force method starts by taking the first data point and comparing it to every other data point in the dataset. After finding all the co-occurrences for the first data point, it then takes the second data point and again looks through the dataset, comparing it to every other data point. The algorithm continues until every data point has been compared to all the other data points. As the data grows, the time complexity increases exponentially. While the brute force can be inefficient and scale exponentially, there are ways of minimizing the exponential growth. For example, Njoo et al. compared the

time-sorted data points of one user to the time-sorted data points of another user and stopped the comparisons once a user's data points were exhausted or the last time-stamp of a user surpasses the window of time allowed from the closest time-stamp of the other user. In doing so, some comparisons between data points were able to be skipped. In addition, the code was parallelized using MapReduce, minimizing the work done by a factor of *k*, where *k* represents the number of cores used [4]. While this brute force approach will have the worst possible performance, it is guaranteed to comprehensively guarantee the identification of all co-occurrences.

Approach 2: grouping the data points based on their location

In this approach, only the data in each group would need to be compared, eliminating many unnecessary comparisons. The data in each group could then be compared using the brute-force method. While the computational efficiency of this approach is still $O(n^2)$, the n in the equation represents the size of each group, as opposed to the original brute force method discussed where the n in $O(n^2)$ represented the size of the entire dataset. Alternatively, the data in each group could be sorted on time. Then, the data in each group could be compared using the sliding-window approach. The sliding-window approach takes advantage of the fact that the data is sorted on time in order to minimize the work done. It does this by creating a window of the size of the time criteria for a temporal co-occurrence. As each data point is scanned, the program checks if it is within the time range for a co-occurrence from the data at the beginning of the window. If it is, it is added to the window. If it is not, the window continually shifts over until the earliest data point in the window is considered to temporally co-occur with the current data point. The process continues until all the data points have been scanned. The sliding window approach is computationally efficient; It is $O(n * log(n))$, assuming the sorting takes $O(n * log(n))$

time. The time complexity is not affected by the time required to check each data point since each data point only needs to be looked at once.

All the articles written about this approach had similar approaches to splitting the Earth into cells. The articles differed, however, on what method they used to determine if data points within a cell spatially co-occur. Cranshaw et al. does not seem to specify what method was used [5], while Cheng et al. specifies that within each cell, the data points were split up into groups of users and each user's check-ins were compared to all the other users' check-ins within that cell. While different than the brute force method, the computational efficiency of Cheng et al.'s approach is still $O(n^2)$, where depending on the method used to compare the data point's times, n is either the number of unique users in the cell or the number of check-ins in the cell [6]. The last approach, seen in the article by Crandall et al., makes use of the sliding-window approach in each cell in order to find which data points temporally co-occur [7].

While splitting the data points into groups based on their spatial distribution improves the computational efficiency, many potential co-occurrences are lost. The traditional implementation of this approach is to partition the Earth into cells the size of the criteria for a spatial co-occurrence. Once partitioned, data points are only compared if they are in the same cell. However, data points in the right quadrant of a cell can spatially co-occur with data points in the left quadrant of the cell to the right of it. This creates many missed co-occurrences. While splitting the Earth into equal-sized cells and only comparing data points within a cell can be computationally efficient, the approach misses many co-occurrences.

Approach 3: using data with discretized check-in locations

In this approach, the complexity involved with discerning if data spatially co-occurs is simplified since the spatial component of the data is discretized. The only spatial question that needs to be asked when comparing data points is if they have the same check-in location or not. There are many different scenarios where data would have a discrete check-in location: cell phone towers, WiFi APs, places where people are asked to swipe an identification card, among others. Given that two people spatially co-occur, there are three ways to identify if they also temporally co-occur: the fixed time slicing approach, sliding window approach, and clustering approach. The fixed time slicing approach involves sorting the list of occurrences by time and then separating it into groups by the time interval. All occurrences in the same group are considered to temporally co-occur. The downside of this approach is that there can be a loss of co-occurrences if one person is towards the end of a time slot and one person is towards the start of a time slot. The sliding window approach is the same approach discussed earlier. The clustering approach involves considering all check-ins that occur at the same event to be co-occurrences. For example, if a significant amount of people connect to a WiFi AP for a long enough duration of time, consider it to be an event and consider everyone to have co-occurred. The advantage to this approach is that there is no set amount of time that needs to determine what makes a co-occurrence. The researchers do not have to figure out what is a meaningful amount of time, instead relying on what would indicate an event.

The various articles on this approach use varying data sources: such as discrete check-in locations for college students to check in at [8], cell-phone towers [9], WiFi Access Points [10], and discretized space with a large population of birds and feeders [11]. Each of the articles

discussed a different algorithmic approach to finding co-occurrences. The algorithms used range from a sliding-window on what may or may not be sorted data (if unsorted, it becomes an $O(n^2)$ algorithm) [8,9], a clustering algorithm (with an $O(n)$ runtime) [10], and the Gaussian mixture model (with an $O(n)$ runtime) [11].

While the discrete check-in location approach has a better time complexity than the other approaches mentioned thus far, the approach's need for discrete locations minimizes the types of data that can be analyzed with the algorithm and does not allow for fine-tuning spatial boundaries. While the approach is convenient for places where locations are discrete such as WiFi access points or cell phone towers, the approach does not work for analyzing co-occurrences from data that makes use of the geographic coordinate system or other situations where locations cannot be discretized. Furthermore, the approach does not allow for flexibility in redefining locations upon further analysis. In highly dense and large locations - like a stadium or a transit terminal - defining a location by an address may help narrow the data. However, it will not necessarily assist in inferring social ties from co-occurrences since many co-occurrences will be meaningless and a product of mere chance.

Approach 4: using a k-d tree

In this approach, the check-ins are stored in a k-d tree. Muskat explains that a k-d tree is a k-dimensional data structure that operates like a binary tree. Each node in the k-d tree represents a check-in. The k-d tree allows for $O(log(n))$ time search and can be configured to partition itself into small subtrees so that even the denser areas within the tree can be traversed quickly. The k-d tree is specially suited for multidimensional data, which is great for spatiotemporal data.

This is because spatiotemporal data consists of three dimensions: latitude, longitude, and time. For any given data point, the time complexity for finding a co-occurrence in a k-d tree is $O(n * log(n))$.

Muskat mentions two articles that use a k-d tree in order to infer co-occurrences from spatiotemporal data. The first article, by Pham et al., is titled *EBM: an entropy-based model to infer social strength from spatiotemporal data*. In it, Pham et al. details their approach of using a k-d tree to store all the data points and then using the MapReduce framework to parallelize the process of finding co-occurrences [12]. In addition to noting how frequently two people co-occurred, Pham et al. factored in the diversity of the co-occurrences in order to ensure that there were no mistaken social ties that happened by chance or as a result of people's everyday routines. Njoo et al. in *Distinguishing Friends from Strangers in Location-Based Social Networks Using Co-Location* uses a similar approach as Pham et al. in creating a k-d tree to store all the data points and using the MapReduce framework to identify co-occurrences [13]. However, Njoo et al. extracts what they call "four key features" - namely diversity, popularity, stability, and duration - in order to identify which co-occurrences are more meaningful. In addition, Njoo et al. also considered the difference between weekday and weekend co-occurrences to see if it makes a co-occurrence more meaningful or not.

While the approach of identifying social ties from spatiotemporal co-occurrences using a k-d tree has a fairly good time complexity of $O(n * log(n))$ - with n being the number of check-ins in the dataset - this approach can lead to a loss of co-occurrences, and it does not allow for the entire process to be parallelized. Pham et al. and Njoo et al. 's approach leads to a loss of

co-occurrences because the data is passed to the mapper in subtrees. Each subtree is dealt with in separate mappers. If two co-occurring points occur in different subtrees, the co-occurrence will not be identified. In her analysis, Muskat notes that this problem may be partially solved by Pham et al. 's use of Shanon entropy in determining the partitioning of the subtrees. She argues that the use of Shanon entropy may allow for a better distribution of the data points within each subtree. However, Muskat notes that splitting up the subtrees amongst different mappers nonetheless can lead to a loss of co-occurrences. Another flaw in this approach is that the entire process is not able to take advantage of parallelization. The initial creation of the k-d tree needs to be done on the same core, and only then is the algorithm able to be parallelized.

How This Work is Different:

Analysis of previous approaches

This work sets out to improve upon past approaches in identifying co-occurrences from spatiotemporal data. Previous approaches have varied in time complexity, ability to be parallelized, in its ability to find all co-occurrences, and in its ability to apply to all different types of data. Each of the approaches lacks some of these features; None of the approaches are fully scalable, efficient, and comprehensive in retrieving all of the co-occurrences. While the concern for comprehensiveness can potentially be dismissed on the grounds that identifying most co-occurrences would be enough to identify social ties, the concern for scalability and efficiency cannot be dismissed if it is to be used on real-world datasets which are usually massive in size.

Of the four approaches discussed - the third approach - that uses data with discretized locations and analyzes it using a clustering algorithm has the best time complexity of $O(N)$, with

N being the number of data points in the dataset. However, as mentioned earlier, this approach has limited applicability since it is not always possible or logical to discretize the spatial data.

Following this approach, the next approach with the best time complexity is the second approach mentioned which was that of splitting the Earth into cells and analyzing the data that falls within each cell independently from the other cells. This has a time complexity of $O(n * log(n))$, where n is the number of data points in the most populated cell. However, that approach inevitably leads to a loss of co-occurrences since a data point on the periphery of a cell could co-occur with a data point on the periphery of an adjacent cell. As discussed in the previous paragraph, this concern could be dismissed on the grounds that the loss of some co-occurrences will likely not hinder identifying social ties amongst users. While that may be true some of the time, it is also feasible to argue that if two users co-occurred seven times but three of them were missed, the social tie would not be identified. In addition, in some cases that this data is used, it might be necessary to identify every single person that a person has co-occurred with.

Following the approach of splitting the Earth into cells and analyzing the data independently, the next approach that has the best time complexity is the fourth approach - using a k-d tree to store the data and use the MapReduce framework to identify co-occurrences. This approach has the time complexity $O(N * log(N))$, where N represents the number of data points in the dataset. While the process is sped up by using the MapReduce framework, the k-d tree is split into subtrees and fed to mappers which can cause a loss of co-occurrences. In addition, the initial creation of the k-d tree is not done over the MapReduce framework; All the initial data

must be dealt with on the same core. Should the data grow immensely, a single core may not be sufficient. The last approach - the brute force approach - is not scalable at $O(N^2)$ so it is not worth further exploration.

After this analysis it becomes apparent that none of the approaches are computationally efficient and scalable, while also accounting for *every* co-occurrence. In addition, most research has been mathematical in nature. This work sets out to combine some of the approaches in order to reach a computationally efficient and scalable approach that finds every co-occurrence.

Summary of the analysis of previous approaches

Approach	**Big O**	**Parallelization**	**Ability to find all co-occurrences**
1 - brute force	$O(N^2)$, where N represents the number of data points in the dataset.	If the data was split up amongst cores, the comparisons could be partially parallelized.	Can find all co-occurrences.
2 - grouping the data points based on their location	$O(n * log(n))$, where n is the number of data points in the most populated cell. This assumes the data in each cell is sorted with an $O(n * log(n))$ sort and co-occurrences in each cell are found using the sliding window approach.	The processing of each individual cell could be done independently, allowing for parallelization.	Data points on the periphery of a cell could potentially co-occur with a data point in an adjacent cell but will not be detected.
3 - using data with discretized check-in locations	$O(N)$, with N being the number of data points in the dataset. This assumes a clustering approach is used.	Could be parallelized.	Will find all co-occurrences that the clustering algorithm defines as a co-occurrence.

4 - using a k-d tree to store the data and use the MapReduce framework to identify co-occurrences	$O(N * log(N))$, where N represents the number of data points in the dataset.	Partial parallelization is possible. The k-d tree would need to be created and then the processing of the sub-trees could be parallelized.	In feeding subtrees of the k-d tree to different mappers, data points that could have co-occurred but were in different subtrees will not be found

Why this work is important

The work to be discussed is significant because there are certain scenarios, like terrorism or epidemiological tracing, where exactness is necessary. It would not be sufficient to have found most of the co-occurrences when trying to uncover a ring of terrorist or criminals. It would be critical to find every single person in the organization. In addition, it is important to have an approach that is scalable and computationally efficient. The magnitude of data being collected is only growing as memory becomes cheaper and more devices are created to track different metrics. The sheer quantity of data is only growing by the day so having an approach that can deal with the magnitude of data is critical. As this work will set out to show, the problem can be solved in a scalable, computationally efficient, and memory efficient way using Spark. All the data can exist on separate cores; Never does all of the data need to be together.

This Work's Approach:

Data used

The dataset used in the testing of the algorithm is the dataset from the Stanford Network Analysis Project which has data from Gowalla, a location based social networking website where users share their locations by checking-in[1]. The data has 196,591 nodes, 950,327 edges, and

[1] https://snap.stanford.edu/data/loc-gowalla.html

6,442,890 check-ins. Each check-in contains a user identifier, the check-in time, the latitude and longitude of where the user checked in, and the location id of where the user checked in.

The program

The program requires the following inputs: a file that contains the check-ins, the distance from which two check-ins can be apart and still be considered co-occurrences, and the length of time that two check-ins can be apart and still be considered co-occurrences. The format of the file is assumed by the program to have each check-in on its own line and for each check-in to have a user identifier first, then the check-in time, then the latitude, and then the longitude. The information about each check-in is assumed to be separated by commas.

The program assigns each data point to cubes. The cubes are best imagined as existing within a three-dimensional space. The width, length, and height of the cube correspond to longitude, latitude, and time. Thus, each cube represents a certain range of space and time. Each cube will be the same size; Each cube's width, length, and height will depend on the criteria that was defined for a co-occurrence. The width and length of the cube will be double the criteria for a spatial co-occurrence and the height of the cube will be double the criteria for a temporal co-occurrence. For example, if the criteria for a spatial co-occurrence is 1 degree, then the width and length of each cube will be 2. The same logic applies for the temporal criteria. For example, assuming that the criteria for a spatial co-occurrence is 1 and the criteria for a temporal co-occurrence is 540 seconds (9 minutes), all data points that have a longitude of 100 to 102, latitude of 100 to 102, and a time between 8/1/2009 5:12pm to 8/1/2009 5:30pm would be assigned to a single cube.

The assignment of each data point to a cube is best demonstrated through an example. Suppose the spatial criteria for a co-occurrence was .5 degrees and the temporal criteria for a co-occurrence was 300 seconds (5 minutes). That means that each cube will have a width of 1, a length of 1, and a height of 600. In the code, this information would be stored in the variable *spaceWindow* and the variable *timeWindow*. The variable *spaceWindow* corresponds to the width and length of each cube - representing the spatial criteria - and the variable *timeWindow* corresponds to the height of each cube- representing the temporal criteria. Prior to calculating what cube the data point belongs to, the program converts the timestamp given to epoch time. Following that, the program uses the equation $floor(latitude/spaceWindow)$ to calculate the x-coordinate of the cube, the equation $floor(longitude/spaceWindow)$ to calculate the y-coordinate of the cube, and the equation $floor(time/timeWindow)$ to calculate the z-coordinate of the cube. For example, if we had the data point in which the user had checked in at -40.74°, -73.97° on 07/16/2009 at 6:13:00am, the time would first be converted to epoch time. The data point would thus become -40.74°, -73.97°, 1,247,724,780. Following that, the program would use the equations to convert the check-in data to coordinates. The x-coordinate, representing the longitude, would be $floor(-\ 40.74/1)$, yielding -41. The y-coordinate, representing the latitude, would be $floor(-\ 73.97/1)$, yielding -74. The z-coordinate, representing time, would be $floor(1,247,724,780/600)$, yielding 2,079,541. Thus, the coordinates for the data point in which a user checked in at -40.74°, -73.97° on 07/16/2009 at 6:13:00am would be $(-\ 41, -\ 74,\ 2,079,541)$.

In addition to assigning each data point to a cube, the program also considers which data points are near the edge of the cube. Any data points that are on the periphery of the cube could potentially co-occur with data points in the adjacent cube. To ensure that all potential co-occurrences are found, the program checks the location of each data point relative to the cube to see which section of the cube it falls into. There are seven different areas of the cube that will require the data to be duplicated to an adjacent cube.

The areas of the cube that need to be checked to see if they require duplication are: the front, right, bottom, front right, front bottom, bottom right, and bottom right front. The front of the cube - more technically speaking - is considered to be from the *x*-coordinate that is at the middle of the cube until the end of the cube. To check if the data point is within that region, it checks if the longitude is greater than or equal to $2 * spaceCrit * curCubeX + spaceCrit$, where *spaceCrit* represents the criteria for a spatial co-occurrence and *curCubeX* represents the top left corner of the cube that the data point is initially assigned to.
$2 * spaceCrit * curCubeX$ calculates the longitudinal value that is at the top left corner of the cube. Adding *spaceCrit* gives the longitudinal value that would be found in the middle of the cube. The right and bottom of the cube would be calculated similarly, but with the *y*-axis and *z*-axis, respectively. The front right of the cube is considered to be the space that overlaps with the front and right area of the cube. The same can be said for the front bottom, bottom right, and bottom right front.

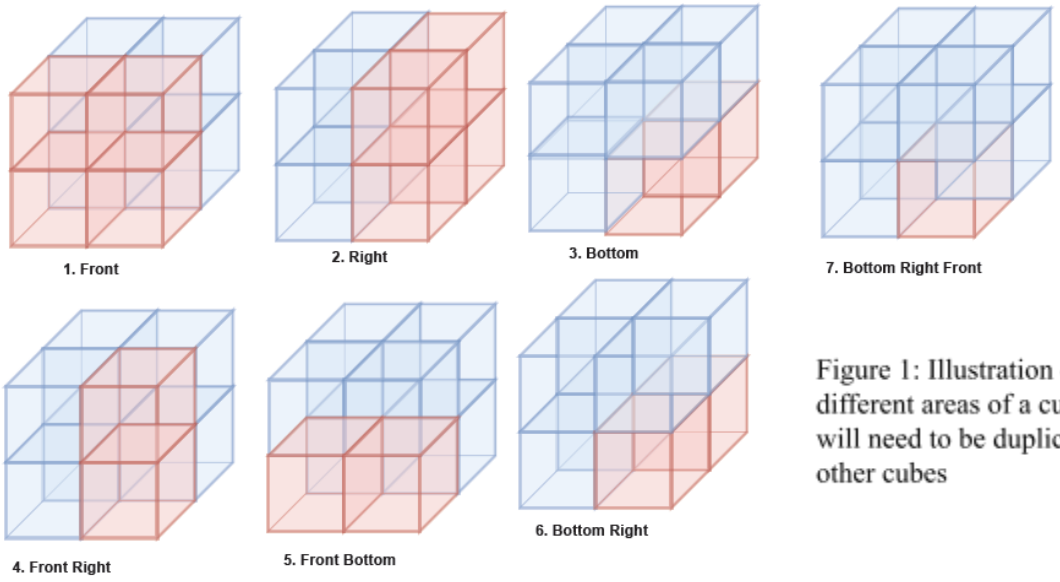


Figure 1: Illustration of the different areas of a cube that will need to be duplicated into other cubes

If the data point does fall within one of the defined areas that require duplication, the program will duplicate the data point and send it in the direction that it was checking. For example, if the program deemed a data point it was checking to be in the front half of the cube, the program would duplicate that datapoint and send it to the cube in front of it. In duplicating the data in this manner, all of the data points that could potentially co-occur will be compared. Data points are not duplicated to any other cubes than those listed because the data will inevitably rendezvous in a different cube. For example, data in the left half of the cube is not duplicated to the cube to the left of the current cube. This is because the data in the right half of the left cube will be duplicated into the current cube. Then, the relevant data in the left side of the current cube will be compared to the relevant data that was duplicated in. In limiting the number of directions that data is duplicated, less work and memory is required. If n represents the number of data points in a given cube, the average number of duplicates created per data point is $\frac{19}{8}n$. The calculation is as follows: ½ of the cube needs to be duplicated in the front, right, or bottom direction, ¼ of the cube needs to be duplicated in the front right, bottom right, or

bottom front direction, and ⅛ of the cube needs to be duplicated to the cube that is in the front right bottom area of the cube, leading to $\frac{n}{2} * 3 + \frac{n}{4} * 3 + \frac{n}{8} = \frac{19}{8}n$.

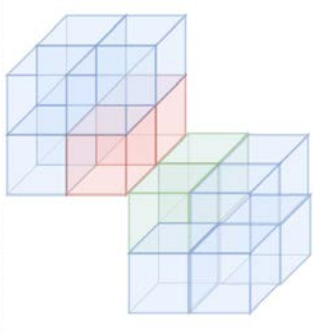


Figure 2: Illustration of the data in the bottom right area of the cube being duplicated into the cube that is to the bottom right of the cube

After each data point is assigned its respective cubes, all the data points that are in the same cube or were assigned to the same cube are gathered and sorted on time. In the context of Spark, this means that the assigned cube acts as the key and the data point acts as the value in the tuple. This results in a key-value tuple in a form similar to *(cube coordinates, data point)*. Spark ensures that all data points with the same cube coordinates are grouped together.Then, a sliding window approach will be used to compare all data points within the time-frame defined for a temporal co-occurrence. Within the sliding window, each data point will be looked at in relation to all the other data points. The data points will be compared to see if they fit the spatial criteria. If they are considered to spatially co-occur then the program will emit a tuple in the form (User A, Time A, User B, Time B). User A will always be the smaller number. This must hold true so that the program properly uniquifies all instances (via a set) and it does not count a single co-occurrence multiple times. The time will then be stripped from the tuple so the program will be left with tuples like this (User A, User B). The program then counts how many tuples User A and User B have and emits the pair of users and the number of times they co-occurred.

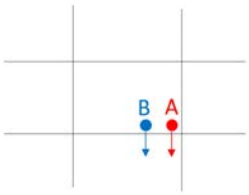


Figure 3: An illustration showing when two data points will be compared more than once. An extraneous pair may be emitted, but will be uniquified when the tuples are placed in sets

Potential issues

If the criteria for the spatial or temporal criteria is too large or too small, the program will not identify all co-occurrences or will identify meaningless co-occurrences. If the criteria for a co-occurrence is too large, then co-occurrences amongst users could be counted more than once. For example, if user A visits a coffee shop at 8:30am, 12:32pm, and 3:59pm and user B visits the same coffee shop at 8:46am, 12:56pm, and at 4:00pm and the criteria for a temporal co-occurrence is twelve hours, the program would count nine co-occurrences. In truth, user A and user B probably only co-occurred three times. Similarly, if the criteria for a spatial co-occurrence is too large in a city where the population is congested, there will be many meaningless co-occurrences generated. If the criterias provided are too small, co-occurrences that are meaningful will be missed. While this is a potential pitfall, the issue lies with the user of the program and is not an inherent flaw of the program.

Testing

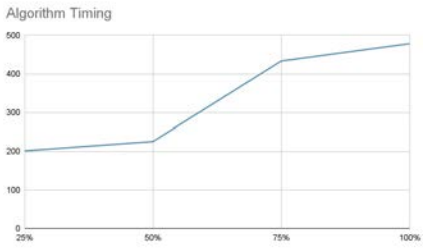
In order to ensure that all co-occurrences are accounted for and no false co-occurrences are reported by the program, testing of the program was done on a smaller dataset. The brute force approach of comparing every data point to every other data point was done and all of the co-occurrences and their counts were emitted and compared to the co-occurrences and their

counts produced by the program using the smaller dataset. There were no differences when comparing the output of the two programs.

The time performance of the program was tested on varying amounts of data in order to see how it scaled. The program execution was timed with the same criteria - .05 for the spatial criteria and 300 for the temporal criteria - on 25%, 50%, 75%, and 100% of the test data. When the program ran on 25% of the data set - which is 1,610,723 data points - the program executed in 201.3101 seconds. When the program ran on 50% of the data set - which is 3,221,446 data points - the program executed in 225.2517 seconds. When the program ran on 75% of the data set - which is 4,832,169 data points - the program executed in 433.4002 seconds. When the program ran on 100% of the data - which is 6,442,892 data points - the program executed in 478.2836 seconds.

In discussing the directions in which data is to be duplicated, this paper claimed that if n represents the number of data points in a given cube, the average number of duplicates created per data point is $\frac{19}{8}n$. This claim was tested on 50%, 75%, and 100% of the data. Each time a data point was duplicated into a cube other than its starting cube, that data point's count increased. The average number of duplications per data point was calculated for 50% of the data to be 2.383, for 75% of the data to be 2.381, and for 100% of the data to be 2.379. Based on this, the average number of duplications per data seems to be converging to 2.375 or $\frac{19}{8}$.

Percent of test data	**Number of lines**	**Time** (seconds)
100%	6,442,892	478.2836
75%	4,832,169	433.4002
50%	3,221,446	225.2517
25%	1,610,723	201.3101



Conclusion

This paper outlines an algorithm for computing all co-occurrences from spatiotemporal data in an efficient and scalable way while ensuring that no co-occurrences get lost. As discussed in the paper, no prior methods include all of the following characteristics: scalable and efficient so it can be used with large amounts of data, flexible in that it can be used with many different types of spatiotemporal data, and comprehensive in that it ensures that no co-occurrences are missed. As such, the contribution of this paper is an algorithm that has all of these characteristics.

The algorithm begins similarly to approach two discussed in the paper - grouping data points based on their location - by assigning each data point coordinates of a cube as a way to later on the group the data. In order to ensure that all co-occurrences are found, the algorithm also creates duplicates of data that are in areas of its assigned cube that are close enough to another cube to spatially co-occur with data in the adjacent cube and assigns that duplicated data point to the adjacent cube. Following that, all data points with the same assigned cube are gathered and sorted based on time. A sliding window approach is then used to efficiently determine which data points co-occurred. All of this work is fully parallelized using Spark.

Future work in this area could involve empirically testing all the different approaches on varying types of spatiotemporal data. More specifically to the work presented in this paper, this algorithm could be tested on varying types of spatiotemporal data and with a larger dataset than used in this paper. In addition, more work needs to be done on maximizing the efficiency of the algorithm. Potential avenues for maximizing the efficiency of the algorithm involve trying to

swap the Spark functions currently being used with more efficient ones and experimenting with the number of partitions and memory assigned. While this paper has contributed to the work on computing all co-occurrences of spatiotemporal data with the algorithm presented, there is still much room for improvement and for further strides to be made.

The code written in conjunction with this paper can be found at:

https://github.com/lilypolonetsky/SpatioTemporalCo-occurrence

Bibliography

[1] Chambers, Bill, and Matei Zaharia. “1. What Is Apache Spark?” *Spark: The Definitive Guide: Big Data Processing Made Simple*, O'Reilly, Beijing, 2018.

[2] Damji, Jules S., and Jules S. Damji. “1. Introduction to Apache Spark: A Unified Analytic Engine.” *Learning Spark: Lightning-Fast Data Analytics*, O'Reilly Media, Sebastopol, 2020.

[3] Muskat, Elisheva. Computational Efficiency in Inferring Social Ties from Spatiotemporal Data Presented to the S. Daniel Abraham Honors Program in Partial Fulfillment of the Requirements for Completion of the Program Stern College for Women Yeshiva University January 4, 2021. New York, NY. Mentor: Professor Alan Broder, Computer Science.

[4] Njoo, Gunarto Sindoro, et al. “Exploring Check-in Data to Infer Social Ties in Location Based Social Networks.” *Advances in Knowledge Discovery and Data Mining Lecture Notes in Computer Science*, 2017, pp. 460–471., doi:10.1007/978-3-319-57454-7_36.

[5] Cranshaw, Justin, et al. “Bridging the Gap between Physical Location and Online Social Networks.” *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, 2010, doi:10.1145/1864349.1864380.

[6] Cheng, Ran, et al. “Inferring Friendship from Check-in Data of Location-Based Social Networks.” *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, 2015, doi:10.1145/2808797.2808884.

[7] Crandall, D. J., et al. “Inferring Social Ties from Geographic Coincidences.” *Proceedings of the National Academy of Sciences*, vol. 107, no. 52, 2010, pp. 22436–22441.,doi:10.1073/pnas.1006155107.

[8] Xu, Jing-Ya, et al. “Finding College Student Social Networks by Mining the Records of Student ID Transactions.” *Symmetry*, vol. 11, no. 3, 2019, p. 307., doi:10.3390/sym11030307.

[9] Shi, Li, et al. “Geographical Impacts on Social Networks from Perspectives of Space and Place: an Empirical Study Using Mobile Phone Data.” *Journal of Geographical Systems*, vol. 18, no. 4, 2016, pp. 359–376., doi:10.1007/s10109-016-0236-8.

[10] Fang, Le, et al. *Event-Based Social Network Discovery (ESONED) Using WiFi Access Points*, 2016, www.semanticscholar.org/paper/Event-Based-Social-Network-Discovery-(ESONED)-Using-Fang-Guan/005cfab60b6f0dd2f3da94ce725a45775c3a09e6.

[11] Psorakis, Ioannis, et al. “Inferring Social Structure from Temporal Data.” *Behavioral Ecology and Sociobiology*, vol. 69, no. 5, 2015, pp. 857–866., doi:10.1007/s00265-015-1906-0.

[12] Pham, Huy, et al. “Ebm.” *Proceedings of the 2013 International Conference on Management of Data - SIGMOD '13*, 2013, doi:10.1145/2463676.2465301.

[13] Njoo, Gunarto Sindoro, et al. “Distinguishing Friends from Strangers in Location-Based Social Networks Using Co-Location.” *Pervasive and Mobile Computing*, vol. 50, 2018, pp. 114–123., doi:10.1016/j.pmcj.2018.09.001.