Spaten: a Spatio-temporal and Textual Big Data Generator

Thaleia Dimitra Doudali *
Georgia Institute of Technology
thdoudali@gatech.edu

Ioannis Konstantinou and Nectarios Koziris
CSLAB, National Technical University of Athens
{ikons,nkoziris}@cslab.ece.ntua.gr

***Abstract*—Social networking users have the ability to check into Points of Interest (POIs) and associate location with their posts or tweets, leading to the creation of *Geo-Social Networks* (GeoSNs). There are many systems that aim to efficiently store and analyze plain and socially enhanced spatio-temporal data. A proper evaluation of these systems should be done using real data from popular GeoSNs, such as Foursquare, Facebook, etc. However, privacy restrictions prohibit the access to such real data in a large scale. Therefore, evaluations are done using real or synthetic data sets that include either only spatio-textual data (e.g. tweets) or plain spatial data (e.g. GPS traces) that are not socially enhanced. In this paper, we present Spaten, an open-source configurable spatio-temporal and textual data set generator, that extracts GPS traces from realistic routes utilizing Google Maps API, combines them with real POIs and relevant user comments crawled from TripAdvisor and makes the data available for further analysis. The injection of social properties extracted by existing Twitter graphs to the generated data along with further parameterization leads to realistic GeoSN data sets. We create and publicly offer GB-size datasets with millions of check-ins and GPS traces. As a proof of concept, we loaded the generated data into a Big Data enabled NoSQL system, and we evaluated its scalability by performing queries typically found in social networking sites. We hope that Spaten can provide the research community with the ability to generate realistic GeoSN data in a large scale, so as to properly evaluate their work.**

I. Introduction

Recently, the merge of social networking and location-based services has led to the creation of *Geo-Social Networks* (GeoSNs). Users of social media can check into locations of interest (POIs) or attach their current location to their posts. In this way, the corresponding GeoSN data contain information about the user (social aspect), the location (spatial aspect), the time (temporal aspect) and the actual post (textual aspect). The proliferation of spatial data in social networking services has motivated the development or extension of systems that aim to efficiently store and process this type of data. For example, PostgreSQL -a traditional relational database- has incorporated PostGIS, a spatial database extender, in order to allow location queries to be run in SQL. Also, Hadoop -a framework for distributed storage and processing of large data sets- is aware of spatial data through the SpatialHadoop extension.

In all such works related to spatial data management we are interested in identifying the data sets, that the authors use, in order to evaluate their contributions. We observe that spatial data sets can be either real or synthetic. For example, SpatialHadoop [6] uses three real spatial data sets -TIGER, OSM and NASA- and one synthetic -SYNTH-. Next, when it comes to spatio-temporal and textual data, there is a large use of real data sets extracted from Twitter, due to its convenient free API. For example, the authors of AQWA [2], which is an adaptive query-workload-aware spatial data partitioning mechanism, crawled tweets over a couple of months. The tweets include an identifier, a timestamp, the location in geographical coordinates and text. Likewise, the authors of a location-aware publish/subscribe system [8] used a Twitter data set and the authors of a location-based recommendation system [4] used a real data set extracted from Foursquare containing similar properties.

In Twitter, there is also an ability to crawl a graph of followers in the form of user identifiers instead of real user information. The combination of tweets and the follower's graph can lead to the creation of a real GeoSN data set. However, authors have limited time to crawl such data, therefore the extracted data sets are small. This is the reason why authors use synthetic data sets, which can be of a bigger scale, as the authors of a Geo-Social query processing framework [3] did. Another real GeoSN data set is Gowalla [5] and is used by the authors that introduced the problem of geo-social skyline queries [7]. Additionally, confidentiality and privacy concerns prohibit access to user information of social networking services. This restriction leads to attempts to anonymize GeoSN data sets as authors in [9] analyze. Thus, there is no ability to use a real large-scale GeoSN data set. Also, the use of synthetic GeoSN data sets created by an arbitrary combination of spatial and social data can favor some experiments instead of others, as the authors of [7] state. As a conclusion, there is an obvious need for real or at least realistic GeoSN data sets in a large scale.

In this paper, we build Spaten[1], a configurable generator that can produce large amounts of spatio-temporal and textual data based on realistic user behavior. Spaten extracts GPS traces from realistic routes utilizing Google Maps API, creates user check-ins at real POIs associating relevant user comments crawled from TripAdvisor and makes the data available in a format that enables further analysis. The combination of these data with a social network user graph can lead to the

* Work done while the author was with CSLAB, NTUA

[1] Code available at https://github.com/Thaleia-DimitraDoudali/Spaten

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

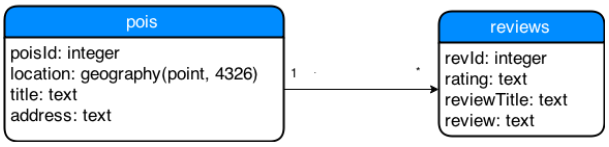

Fig. 1. Database schema for source data

creation of a realistic GeoSN data set in a large scale. The main contributions of our work are the following:

- Design and implementation of an open source configurable generator of spatio-temporal and textual data.
- Creation of Spaten-Dataset[2], a realistic GeoSN data set consisting of 3GB of data generated by Spaten and a 14GB Twitter graph.
- Insertion of Spaten-Dataset into an HBase cluster, a distributed NoSQL database.
- Scalability testing of the HBase cluster using a workload of customized queries.

This work is inspired and builds upon MoDisSENSE [10], a distributed platform that provides personalized search for points of interest and trending events based on the user's social graph by combining spatio-textual user generated data.

II. Generator

A. Source Data

Spaten uses real Points of Interest (POIs) as a source of data. These locations where extracted from the online travel service TripAdvisor, using a generic html crawler. The POIs are identified on the map by their geographical coordinates, as well as their address. Also, they come with ratings and reviews by real TripAdvisor users. The number of such available points is 136,409, as they were extracted by a 13GB response json file from the crawler. The points were stored in PostgreSQL, so as to take advantage of the spatial database extender, PostGIS. Figure 1 depicts the database schema that was used, which contains two tables, one for the points of interest and one for the reviews and ratings of these points.

After storing all POIs into the database, we created an B-tree index to the identifying attributes of the two tables. Additionally, we assigned a PostGIS spatial index called GiST, to the attibute location of the table pois. As location has a geographic data type, GiST implements an improved R-tree spatial index. In this way, searching for POIs according to their location or identifying number, can be done fast and efficiently.

B. Design attributes

Spaten creates users of Geo-Social Networks, who check into many places during the day and leave a review and rating to the corresponding points of interest, for a given period of time. The location and time of the check-in represent the spatio-temporal aspect of the generator and the review and rating, that is associated with each check-in, correspond to the textual aspect. The decisions that the generator makes, such as which and how many locations the user will visit a specific

[2]Dataset available at http://research.cslab.ece.ntua.gr/datasets/ikons/Spaten/

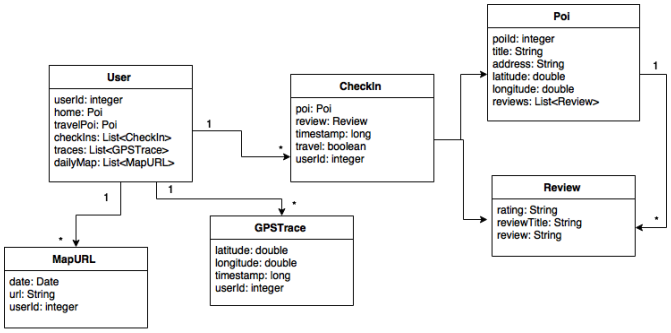

Fig. 2. Class diagram of generator's attributes

day, or the duration of each visit, are made using configurable random factors. The path from one POI to the next one is extracted using Google Maps Directions API. Spaten, is able to construct the GPS traces that represent the route from one POI to the next one. Finally, the generator uses the Google Static Maps API, in order to illustrate each user's daily routes into a static map. Figure 2 contains the class diagram that shows the main attributes of the generator, and their relationships. Spaten is configurable with a set of input parameters that are mentioned in the appropriate context later in this section. Next follows a more detailed description of the decisions that Spaten makes and the attributes that shape its design.

1) Home: The location of a user's home is the center around which he walks every day. Since the points of interest that a user visits are selected in a random way, special handling is required to avoid routes that don't make sense. For example, a route where a user walks one day in Greece, the next day in France and the other day again back in Greece is not realistic. This is the reason why the location of a user's home is specified, so that a user can walk in a sensible distance from his home every day. More specifically, a user can walk in a range of *maxDist* meters from his home, as those are defined by the according input parameter. In the implementation of the generator, the home of a user is defined as the point where his first ever visit is made on *startDate* -the input parameter that defines the first day of generated daily routes. The choice of the home location is done using a generator of random numbers which follow a uniform distribution. The range of the distribution is the total number of source POIs stored in PostgreSQL.

2) Travel: A user created by the generator is capable to travel, so as to be able to visit places that are further than *maxDist* meters from his home. In this way, routes one day in Greece and the next one in France make sense. However, a central point around which the user will walk during his trip has to be defined. In the implementation of the generator, this point is chosen to be the first ever point that the user visits during the current trip. The choice of that point is done using a generator of random numbers that follow a uniform distribution. The range of the distribution is the number of available source POIs.

As far as the duration of the trip is concerned, each user can travel for a time interval that corresponds to 10% of the total

time interval for which Spaten produces daily routes. The days of the time interval can be spread out to multiple trips. The duration of each trip is defined using a generator of random numbers which follow a normal distribution. The mean of this distribution is declared to be 5 and the standard deviation is 2. In this way, according to the 3-sigma empirical rule for the normal distribution, 95% of the random trip durations will be between 1 and 9 days, which is reasonable for short and long trips. To sum up, if the user is about to travel the next days, the duration of the trip is defined in a random way and if the duration of the trip doesn't exceed the available travel days, the prospective trip begins. If the duration of the trip exceeds the available days, then the trip is calculated to last for the available days.

Finally, the decision whether the user will begin a trip or not is made in a random way, using a generator of random numbers that follow the Bernoulli distribution. Thus, the generator decides every day whether or not to start a trip for the current user. This decision ressembles a fair coin toss. Therefore, if the generator decides to start a trip for the current user, then it decides, in a random way as well, the duration and the location of the trip. Each daily route during the trip has to be in *maxDist* range from the trip's central location.

3) Check-in: The number of daily check-ins is defined using a generator of random numbers that follow a normal distribution with a mean value determined by the input parameter *chkNumMean* and standard deviation determined by the input parameter *chkNumStDev*. Therefore, every day the generator picks a different number of daily check-ins, which according to the 3-sigma empirical rule for the normal distribution, will most probably be around the mean value.

The locations of check-ins are determined in a random way by the generator. More specifically, the choice of the point of interest, where the first check-in of the day will take place, is made using a generator of random number which follows a uniform distribution. The range of this distribution is the number of available POIs that are located in *maxDist* range from the user's home. In this way, the generator will pick a random POI between those who are in walking distance from the user's home. If the user travels, then the generator will choose a random POI between those who are in *maxDist* range for the trip's central location. In order to find those points and select them from the PostgreSQL database, the function ST_DWithin function is used, which is available through the PostGIS extension. This function returns true for the points which are in the desired distance from the user's home, calculating the distance using the geographical coordinates of the points. The search of these points in the PostgreSQL table is done efficiently, due to the GiST index. The generator will assemble all the points that are in the desired range, and choose a random one as the first point that the user visits that day.

Using the same strategy, the generator chooses all next points of interest that the user will visit the specific day. However, the next points visited should be in a smaller distance from the first POI visited. This distance is defined by the input parameter *dist*. Also, a user is not allowed to visit the same place twice during the day, so every next check-in should be in a POI not visited that specific day.

4) Review: The generator assigns a rating and review to every user's check-in. More specifically, every source point of interest as it is stored in PostgreSQL database, contains certain reviews for the specific point. The generator chooses randomly a review amongst the available for the POI, using a generator of random numbers that follow a uniform distribution. The range of the distribution is the number of available reviews for the specific POI.

5) Path: The generator issues an http request to Google Directions API, in order to obtain information about the path that a user will follow in order to walk from one point of interest to the next one. Since the source data contain the geographical coordinates of every point of interest, the generator has access to the latitude and longitude of every available point. Thus, it sets as value of the origin parameter to the http request the geographical coordinates of the current point where the user checked in and as destination the coordinates of the next point of interest that the user will visit, as it was selected in a random way. Also, he specifies the parameter mode into walking, because in our approach the user walks from one POI to the next one. Finally, the response file of the request will be in json file format. For example, a request to Google Directions API can be the following:

http:
//maps.googleapis.com/maps/api/directions/json?origin=37.976159,
23.776274&destination=37.978180,23.768957&mode=walking

The json response file contains information about the path on the map, that the user will have to follow in order to get to his destination, as well as the duration of his walk to the destination. We extract from the json file, the field polyline from each step of the route. The polyline holds an encoded representation of the step's path. We decode each step's polyline using the reverse Encoded Polyline Algorithm Format [1]. In this way, we have access to a list of geographical coordinates indicating all the points on the map, that the user will walk through from the origin to the destination. These points will be stored as GPS traces, representing the user's route. For each path, the starting and ending points, which are the points of interest, will also be stored as GPS traces.

6) Time: As far as time is concerned, the generator defines that the first check-in of the day will happen at the time defined by the input parameter *startTime*. The duration of the visit to a point of interest is set using a generator of random numbers that follow a normal distribution. The mean value of the distribution is defined by the *chkDurMean* input parameter, and the standard deviation by the *chkDurStDev* parameter. The time when the next check-in will occur is set to be the time when the previous one happened, plus the duration of the previous visit and the duration of the walk from the previous point to the next one. The duration of the walk, is extracted from the field duration of the Google Directions json response file. If the time that the next check-in will happen exceeds the time defined by the input parameter endTime, then the next

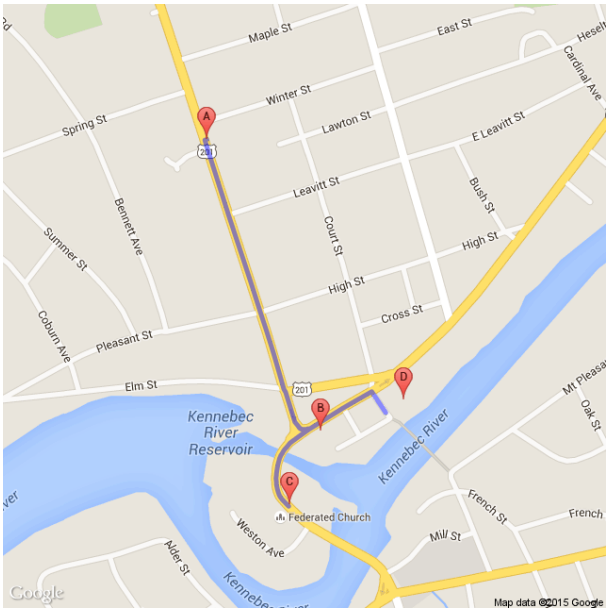

Fig. 3. Example of static map image

check-in won't occur, and the check-ins end at that time for that specific day. Each check-in is timestamped using the Unix Time Stamp, which is a long integer representation of the date and specific time (UTC timezone) of the event.

Concerning the timestamp of the GPS traces, they are calculated through the json response file. The duration of the route from the origin to the destination is split up by the number of points decoded from the polyline. Therefore, the timestamp of the first GPS trace of the path is set to be the time the visit at the origin ended plus the fraction of the divided time needed to get to that point on the map.

7) Static Map: Spaten uses a static map in order to depict the user's daily route, by issuing an http request to the Google Static Maps API. The points where a user checked in during the day are distinctly visible on the map with markers. These markers are also named using capital letters of the alphabet in order to show the order that the user visited them. The generator uses the stored polylines, as they were extracted by the json response file, in order to define the path that the map will indicate using a blue continuous line. The image of the map is accessible through the URL that forms the http request to the API. The generator stores the URLs created at the corresponding output file, so that the image of the map can be accessible by the users of the generator. Figure 3 shows the static map that corresponds to the following request to Google Static Maps API:

https://maps.googleapis.com/maps/api/staticmap?&size=1000x1000&markers=label:A|44.7698,-69.7215&markers=label:B|44.7651,-69.7189&markers=label:C|44.7639,-69.7196&markers=label:D|44.7656,-69.717&path=color:blue|enc:gbgpGjnphL@FZKtE_BxEeB‘Bk@z@[bA]tBo@HG~Ai@JMFG?A@A?A@C?C?E?C?EAE?I??M_@??L^??Tj@Vl@LNDFDDFDDDB@DBD@HBLBRBB?H@JAF?DAFAHCFCDCBCDEDGNU??OTEFEDCBEBGBIBG@E@G?K@IAC?SCMCICEAECCAEEGEEEEGMO??Wm@Uk@cAyCs@_C??bA_A

Users	Check-ins	GPS traces
9464	1586537	38800019
3GB	641 MB	2.4 GB

TABLE I
GENERATED DATA

III. DATASET

A. Configuration

We were able to create a large spatio-temporal and textual data set using Spaten. The generator was configured so that the number of daily check-ins would follow a normal distribution with a mean value of 5 and standard deviation of 2. The duration of each visit in hours also followed a normal distribution with a mean value of 2 and standard deviation of 0.1. Each user was be able to visit places that are in 50,000 meters radius from his home or travel location. Also, he was allowed to walk in a 500 meters radius from the one place he visits to the next one. Each user visited the first place of the day at 9 am. Moreover, the last daily check-in would take place no later than 11 pm. The generator produced daily routes for the time period between the 1st of January 2015 and the 1st of March 2015.

Respecting the request restrictions for the users of the free Google Directions API to 2500 requests per day to the API, the generator could run once a day creating 14 users at a time for the specific configuration. Therefore, we set up a cluster of 31 Virtual Machines (VMs) in order to be able to create a much bigger number of users per day. Each VM of the cluster had 1 CPU, 1 GB RAM and 10 GB disk. The PostgreSQL database, where the source data were stored, was set up in a different VM where the generator run as well. This specific VM had 2 CPU, 4 GB RAM and 40 GB disk in order to be able to store the source data. When the generator was running on the cluster VMs, a remote connection to the PostgreSQL database was established in order to gain access to the source data as well.

With this setup and using the Google Directions free API, we were able to run the generator on each VM collecting 448 users per day. At the end of the generator's run period, we were able to have 9464 users and their 2 months daily routes. The created dataset was available in the generator's two output files, one that stored the user's daily check-ins and another one having the user's total gps traces indicating the daily trajectories. Table 1 shows the number of generated data. We also had available a 14 GB friend graph which we adjusted in order to match the number of 9464 users created by the generator. All these data constitute Spaten-Dataset, a realistic GeoSN data set of a large scale which is available for download through Spaten's repository web page.

B. HBase data model

The overall dataset of check-ins, GPS traces and friends was stored into an HBase distributed database for a proof of concept scalability evaluation. HBase is a NoSQL database where tables consist of rows and columns. All columns of a table belong to a particular column family. We create one

table in order to store the friend graph. The data about friends consist of user ids. For example, user no.1 has friends the users no.2, no.3 etc. Each row holds all the users that are friends of the user whose id is the key of the row. There is one column family 'friends' including all the friends of one user and the column qualifier represents a single friend using his user id. Then, we create a table for the check-ins. Each row holds all the check-ins of the user whose user id is the row key. There is one column family 'checkIns' which holds all the check-ins of each user and the column qualifier represents a single user check-in through its timestamp. Finally, we create a table in order to store the GPS traces. Each row holds all the GPS traces of the user whose user id is the row key. There is one column family 'gpsTraces' which holds all the GPS traces of each user and the column qualifier represents a single user GPS trace through a string of the geographical coordinates combined with the timestamp of the GPS trace.

C. HBase cluster

The overall dataset was stored into an HBase distributed database. We used the version 0.94.27 of HBase and the version 1.2.1 of Hadoop in order to utilize the HDFS. The HBase was set up over HDFS on a cluster of 32 VMs, consisting of 1 master and and 32 region servers and 1 namenode and 32 datanodes. All different types of data were splitted into 32 parts, so that they are distributed equally into the region servers when the following tables were created. More specifically, when we created the table of 'friends' in HBase, we predefined the keys where the total table would be splitted into the region servers. Thus, the table was pre-splitted into 32 regions so that the data were equally divided into the region servers. The split into 32 parts was also done for the tables of 'check-ins' and 'gps-traces'. The master VM which contains the HBase master as well as the namenode has 2 CPU, 4GB RAM and 40 GB disk. The master is at the same time a region server and a datanode. The other 31 VMs holding the rest region servers and datanodes have 1 CPU, 2 GB RAM and 10 GB disk.

IV. Experiments

A. Queries

After the insertion of all available data into the HBase cluster, we implemented several queries over the tables of 'friends' and 'check-ins'. These queries can be imposed to any social networking service that contains data about users that check in to several locations and have as friends other users of the service. As HBase is a NoSQL database and doesn't have a query execution language like SQL for example, the implementation of the queries was done using HBase coprocessors. HBase coprocessors enable distributed computation directly within the HBase server processes on the servers local data. In this way the computation of intermediate results and other complex calculations is transferred to the region servers that contain the respective data, alleviating the client from a heavy computational load. The following queries were implemented:

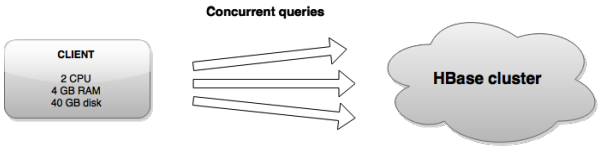

Fig. 4. Workload setup

- *Most visited POI*: Get the most visited points of interest of a certain user's friends.
- *News Feed*: Get the check-ins of all the friends of a specific user for a certain day into chronological order.
- *Correlated Most Visited POI*: Get the number of times that a user's friends have visited the user's most visited POI.

B. Workload

Using the above queries we created a custom workload in order to test the behavior of the HBase cluster to multiple requests. More specifically, the workload consists of the three different type of queries, since they all refer to the same HBase tables. Moreover, the workload takes as input the number of queries that will be executed. In addition, since all queries include the retrieval of the friends of one user, that user is chosen randomly. The user that corresponds to each query is chosen randomly from the total user population.

Then, according to the number of queries, all types of queries participate in the workload in a cyclic assignment. For example, if the client wants 5 queries to be executed then those will be 1. most visited POI, 2. news feed, 3. correlated most visited POI, 4. most visited POI, 5. news feed. Also, the queries are executed in parallel as different threads. In this way, the HBase cluster receives simultaneously the query requests. The total execution time of the workload will be the biggest execution time amongst the queries of the workload.

There is one client that according to the specified number of queries to be executed, creates the workload in the way described previously. This client is hosted on a VM with 2 CPU, 4 GB RAM and 40 GB disk. The client is responsible to receive the number of queries to be executed and create the workload in the way described previously. The queries arrive at the same time in the HBase cluster. Figure 4 depicts the workload setup.

C. Scalability Evaluation

Using the aforementioned setup we performed a scalability evaluation, in order to analyze how HBase handles the workload for the specified data storage model in different cluster sizes. We calculated the latency and throughput of the system. More specifically, latency is calculated as the mean execution time of the queries. Throughput is the number of queries executed per second. We measured the latency and throughput of the system for an increasing number of concurrent queries. We started with the HBase cluster having 32 nodes. Then, we resized the cluster to 24, 16, 8 and 4 nodes in order to observer the variations in the latency and throughput. The restructure of the cluster was achieved by decommissioning the datanodes

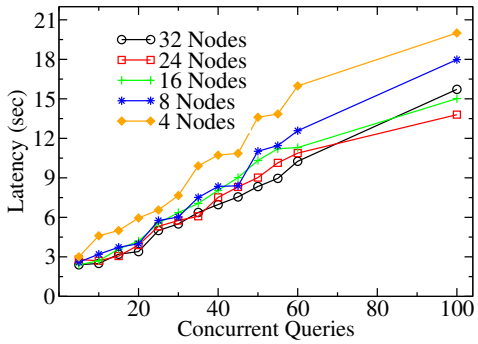

Fig. 5. Scalability Evaluation - Latency

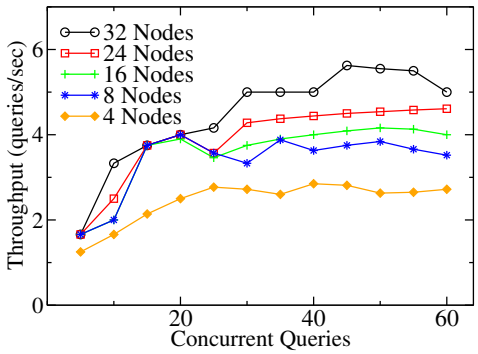

Fig. 6. Scalability Evaluation - Throughput

and region servers to the desired number. Both HDFS and HBase offer commands in order to achieve a cluster resize by moving data and regions into the remaining nodes, preventing data loss and ensuring that the regions will be data balanced.

As Figure 5 shows, latency increases as the number of nodes of the HBase cluster decreases. Latency is the mean execution time of the queries. Thus, it is expected that the latency will increase when there are fewer nodes in the cluster. In more details, when the cluster size reduces there are fewer servers to handle the read requests and calculations that accompany the query. Additionally, the latency increases as the client sends more concurrent queries. This happens due to the fact that as the number of concurrent queries elevates, the servers cannot resolve them simultaneously and many of them have to wait in the respective queues. Therefore, there are queries that have additional waiting time to their total execution time. This waiting time becomes bigger when there are fewer servers to handle the concurrent queries. On the same level, when there is a small number of concurrent queries on different cluster sizes, then the latency is approximately the same due to the fact that there is no added waiting time.

As far as throughput is concerned, according to Figure 6, we observe that the throughput increases as the size of the cluster becomes bigger. This is expected due to the fact that more servers can serve more concurrent queries. When the number of servers increases, the amount of work assigned to each server reduces, if the cluster is balanced. In our case, we ensure that the cluster is balanced as far as the data are concerned, when we pre-split the data into 32 regions. Also, HBase runs a balancer that keeps the regions equally distributed to the region servers. Therefore, when the cluster has 32 region servers, those have to run calculations over fewer data when the coprocessors are called, as opposed to when the cluster has fewer nodes.

As fas as scalability is concerned we observe that rise in resources and performance is almost linear, which shows that the system is not scalable in a satisfactory level. There are many reasons that can contribute to this result. Firstly, the workload is not predefined and can variate according to the randomly selected user factor. Additionally, cache misses affect tremendously the performance. When data for a query are retrieved from the server's cache, then the response time is noticeably less than getting the data from the data node's disk. Also, the workload is executed on a real time system, on a cluster of virtual machines whose performance can be affected by the rest users of the cloud service.

V. Conclusion

In this paper, we designed and built Spaten, a configurable generator of spatio-temporal and textual data. We combined the data created by Spaten with a social network graph and created Spaten-Dataset, a realistic Geo-Social network data set of a large scale. Then, we proposed a storage schema for Spaten-Dataset into a NoSQL distributed database, in order to enable further analysis. We implemented customized queries over those data and created a workload so as to test the scalability of the corresponding HBase cluster deployed in the cloud where we showed that our approach can scale almost linearly to the number of computing and storage resources.

Acknowledgment

This paper is supported by European Union's Horizon 2020 RIA programme under GA No 690588, project SELIS.

References

[1] Encoded polyline algorithm format. https://developers.google.com/maps/documentation/utilities/polylinealgorithm.

[2] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data. *PVLDB*, 8(13):2062–2073, 2015.

[3] N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A general framework for geo-social query processing. *Proc. VLDB Endow.*, 6(10):913–924, Aug. 2013.

[4] J. Bao, Y. Zheng, and M. F. Mokbel. Location-based and preference-aware recommendation using sparse geo-social networking data. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 199–208, New York, NY, USA, 2012. ACM.

[5] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. KDD '11, pages 1082–1090, New York, NY, USA, 2011. ACM.

[6] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1352–1363, 2015.

[7] T. Emrich, M. Franzke, N. Mamoulis, M. Renz, and A. Züfle. *Database Systems for Advanced Applications: 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part II*, chapter Geo-Social Skyline Queries, pages 77–91. Springer International Publishing, Cham, 2014.

[8] G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware publish/subscribe. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 802–810, New York, NY, USA, 2013. ACM.

[9] A. Masoumzadeh and J. Joshi. Anonymizing geo-social network datasets. In *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Security and Privacy in GIS and LBS*, SPRINGL '11, pages 25–32, New York, NY, USA, 2011. ACM.

[10] I. Mytilinis, I. Giannakopoulos, I. Konstantinou, K. Doka, D. Tsitsigkos, M. Terrovitis, L. Giampouras, and N. Koziris. Modissense: A distributed spatio-temporal and textual processing platform for social networking services. SIGMOD, pages 895–900, 2015.