National Technical University of Athens
School of Electrical and Computer Engineering

Computer Science Division
Computing Systems Laboratory

Performance evaluation of social networking services using a spatio-temporal and textual Big Data generator

DIPLOMA PROJECT

THALEIA-DIMITRA DOUDALI

Supervisor : Nectarios Koziris
Professor NTUA

Athens, July 2015

National Technical University of Athens
School of Electrical and Computer Engineering

Computer Science Division
Computing Systems Laboratory

Performance evaluation of social networking services using a spatio-temporal and textual Big Data generator

DIPLOMA PROJECT

THALEIA-DIMITRA DOUDALI

Supervisor : Nectarios Koziris
Professor NTUA

Approved by the examining committee on the July 20, 2015.

.....	.....	.....
Nectarios Koziris Professor NTUA	Nikolaos Papaspyrou Associate Professor NTUA	Dimitrios Tsoumakos Assistant Professor Ionian University

Athens, July 2015

Thaleia-Dimitra Doudali
Electrical and Computer Engineer

Abstract

Nowadays, in the era of Big Data, the amount of social media data, that is being produced daily, increases significantly. The storage and analysis of such data cannot be achieved any more with traditional means and methods. Consequently, social networking services resort in using distributed systems and techniques, in order to store and manage effectively the huge amount of the data they own. Usually, the evaluation of such services results through the ease of use and satisfactory performance. However, deep understanding of how data is stored and managed cannot be reached, due to the lack of access to these data which is imposed by privacy restrictions. In this way, it is not possible to evaluate properly such social networking services. Therefore, goal of the current diploma thesis is to design and implement a generator of realistic spatio-temporal and textual data, that will be similar to real social media data.

The generator uses as source data, real points of interest and reviews for these points, extracted by a well-known travel service. Then, it creates realistic daily routes per user on the map, using the Google Directions API. These daily routes are available in the form of static maps, using the Google Static Maps API. Each daily route includes check-ins at points of interest, together with rating and review of the point, and gps traces indicating the route. Also, the generator functions with various input parameters, that differentiate the amount and structure of data produced. For example, such parameters can be the number of users created, the time period in which daily routes will be produced per user and the number and duration of daily check-ins.

The generator was executed daily for a significant amount of time in order to create a Big Data dataset of spatio-temporal and textual data. More specifically, the generator created 9464 users, 1586537 check-ins and 38800019 GPS traces, which sum up to 3 GB data. The dataset was stored in a distributed database system using a specific data storage model. Moreover, we implemented certain queries for these data, that are representative of queries imposed by the users of real social networking services. Finally, we created a workload of queries and executed them for different number of concurrent queries and different number of nodes of the distributed database system. In this way, we were able to perform a scalability testing to the system and evaluate the performance of distributed means of storage and processing of social media data used by many social networking services.

Keywords

Generator, Spatio-temporal data, Textual data, Big Data, Points of Interest, Daily routes, Google Directions API, Google Static Maps API, HBase, Scalability Testing, Distributed Systems

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays, the amount of social networking data that are being produced and consumed daily is huge and it is constantly increasing. Moreover, these data can be in the form of text, photographs, video, time and location. For example, the online social networking service Twitter has 302 million active users per month and manages 500 million tweets per day, according to official statistics [1]. The tweets include textual data and a timestamp (the time of publication) and may also contain image, video, and information about location. Traditional data storage and processing techniques are no longer sufficient for the huge amount of polymorphic social media data. Consequently, modern social networking services employ distributed systems and data management tools. For example, in the beginning Twitter [5] used to store tweets in traditional databases. However, as time passed and the amount of tweets increased significantly, Twitter had several issues in reading and writing data in these databases. Thus, Twitter created and used distributed storage tools in order to manage it's data. These tools gave notable boost in the performance of the service.

A large number of online services now use distributed data storage systems. For example, large online social media platforms such as Twitter and Yahoo!, use the distributed database system HBase, so as to manage their data [7]. Similarly, many online social networking services like Facebook and Twitter, use the distributed data storage and management system Hadoop [8]. Even though we can retrieve a list of social media platforms that use certain distributed data management tools, we cannot have access to the data that these platforms use, due to user's privacy restrictions. In this way, it is not possible to evaluate properly such social networking services, in terms of scalability and performance.

In the current diploma thesis, we want to bridge the gap considering the full understanding of the function of social networking services. Since we can not have access to the relevant data, we tried to create realistic social networking data, which are similar to real such data. Hence, we designed and implemented a generator of realistic spatiotemporal and textual data, similar to those found in well-known social networking services such as Facebook. The current diploma thesis is inspired and builds upon MoDisSENSE, which is 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.[19]

1.2 Thesis contribution

The main contributions of this work are the following:

1. Design and implementation of an open source[1] parameterized generator of spatio-temporal and textual social media data.
2. Creation of a large dataset of such complex realistic social media data using the generator.

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

3. Creation of the appropriate data model and utilization of HBase, a distributed data management NoSQL database, for the storage and management of these data.

4. Scalability testing of the HBase cluster for the specific data storage model of the dataset produced by the generator.

1.3 Chapter outline

In chapter 2 we present the theoretical background, which is important for someone to know, in order to understand concepts and terminologies used later on. More specifically, we analyze the tools used in the implementation of the generator, such as the database for the source data, Google's services for the creation and presentation of user's routes and a number of mathematical concepts for the input parameters. Also, we include the distributed storage and data management tools, which were used in order to store the dataset created by the generator.

In chapter 3 we describe in detail the design of the generator. We analyze the storage schema of the source data, we define the input parameters and present thoroughly the attributes, the decision and complete function of the generator.

In chapter 4 we analyze the creation of a Big Data dataset using the generator and the data storage model according to which the dataset is stored into an HBase cluster.

In chapter 5 we present the implementation of several queries over the available dataset, as well as the creation of a workload consisting of these queries. Also, we perform the scalability testing of the HBase cluster and present the corresponding results.

In chapter 6 we sum up the conclusions from the above evaluation of the distributed storage and processing tools of the social media data. Finally, we present related work and possible future work.

1.4 Brief generator description

The main function of the generator is to create realistic spatiotemporal and textual social media data. More specifically, the generator creates users of online social networking services, who visit many places during the day and leave a review and rating to the corresponding points of interest. Therefore, the generator produces user's daily routes during a specific perdiod of time.

The generator uses real points of interest as source data. These points where crawled from TripAdvisor, a well-known online travel service. The source data contain the geographical coordinates, the title and address of each point of interest, as well as a list of ratings and reviews for each point made by real TripAdvisor users. In order to store these points, we used the PostgreSQL database, so as to have access to functions of geographic data types, such as the calculation of distance, given by the extension PostGIS of PostgreSQL. The generator is parameterized by certain input variables, so as to create data that can vary in size and structure. For example, there are input parameters that define the number of users created, the period of time for which the generator will create daily routes and others that influence the daily check-ins to the points of interest.

The generator takes certain decisions, that refer to the daily routes created, using random factors. More specifically, the points of interest that a user visits are defined in a random way. The only restriction to this decision, is that the points must be near the user's home. The home of the user is defined in a random way and the distance from there is determined by an input parameter. The generator gives the ability to travel to it's users, so that they can check-in into places that are further away from their home. The location and duration of their trips are also decided in a random way. The generator allows only one visit per day to each point of interest per user. Every user that visits a point of interest leaves a random rating and review for that point. This rating and review is real as it is written by real users of TripAdvisor.

As far as the transition from one point of interest to the next one is concerned, the user walks in order to get to his destination. This is the reason why the check-ins should occur in points of interest

that are in walking distance between them. The distance between the visits is defined through an input parameter. The generator issues a request to Google's Directions API [2] and gets a response file that contains analytical directions and information of the route from the origin to the destination. The response file includes all the intermediates points on the map in an encoded representation. The generator decodes and transforms these points to GPS traces, that indicate the route. The response file also contains the duration of the route from the starting point to the destination. Using this information, the generator can calculate the time when the user will check-in to the next place. Each check-in and GPS trace is timestamped. The number of points that a user will visit during the day and the duration of the visits is determined in a random way influenced by certain input parameters. Finally, the generator uses the Google Static Maps API [4], in order to illustrate each user's daily routes into a static map.

As a result, the data created by the generator simulate real social media data, due to the usage of real points of interest as source data, real reviews and rating to these points, real routes as given by the Google Directions API, and the utilization of random factors in the decisions that the generator makes, while creating the daily routes.

Chapter 2

Background

2.1 Databases

A database [9] is an organized collection of data, which can be managed in an easy way. A database management system is a computer software application that interacts with the user and other applications, in order to create, access, manage and execute queries at the data stored in the database. Databases are used in a wide range of services, such as bank, university and communication platforms.

2.1.1 Relational databases

A relational database is a digital database whose organization is based on the relational model of data [13]. According to this model, data are organized in tables with rows and columns. Each table represents a relation variable, each row is an implementation of this variable and each column is an attribute of the variable. Tables can be associated with one-to-one, one-to-many and many-to-many relationships. More specifically, a relation is a set of table rows that have common attributes. Moreover, each row can be uniquely identified by a certain attribute, called primary key. When a primary key is common between two tables, then it becomes a foreign key for the second table. Finally, a database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.

2.1.2 ACID properties

ACID (Atomicity, Consistency, Isolation, Durability) [15] is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction.

Atomicity

Atomicity requires that each transaction be ”all or nothing”: if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible (”atomic”), and an aborted transaction does not happen.

Consistency

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.

Isolation

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially. Providing isolation is the main goal of concurrency control. Depending on concurrency control method (i.e. if it uses strict - as opposed to relaxed - serializability), the effects of an incomplete transaction might not even be visible to another transaction.

Durability

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

2.1.3 OLTP vs OLAP

OLTP

Online transaction processing, or OLTP, is a class of information systems that facilitate and manage transaction-oriented applications, typically for data entry and retrieval transaction processing. OLTP is characterized by a large number of short on-line transactions (INSERT, UPDATE, DELETE). OLTP must possess ACID qualities to maintain data integrity and to ensure that transactions are correctly executed. The main emphasis for OLTP systems is put on very fast query processing, maintaining data integrity in multi-access environments and an effectiveness measured by number of transactions per second. In OLTP database there is detailed and current data, and the schema used to store transactional databases is the entity model.

OLAP

Online analytical processing, or OLAP, is characterized by relatively low volume of transactions. Queries are often very complex and involve aggregations. For OLAP systems a response time is an effectiveness measure. OLAP applications are widely used by Data Mining techniques. In OLAP database there is aggregated, historical data, stored in multi-dimensional schemas.

2.2 PostgreSQL

PostgreSQL [21] is an open source object-relational database management system. It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video. Moreover, it has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC. It supports international character sets, multibyte character encodings, Unicode, and it is locale-aware for sorting, case-sensitivity, and formatting. In addition, it supports unlimited database size, rows and indexes per table, 32TB table size, 1.6TB row size and 1GB field size. PostgreSQL manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant membership in a role to another role, thus allowing the

member role use of privileges assigned to the role it is a member of. Finally, PostgreSQL comes with several extensions that add extra capabilities in its function and usage.

2.2.1 PostGIS

PostGIS [20] is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL. It defines new data types, functions, operators and indexes especially for geographic objects.

Geography type

The geography type provides native support for spatial features represented on geographic coordinates. A specific point on a map can be identified by it's geographic coordinates, stated in the form of (latitude, longitude). Geographic coordinates are spherical coordinates expressed in angular units (degrees). The basis for the PostGIS geographic type is a sphere. The shortest path between two points on the sphere is a great circle arc. That means that calculations on geographies (areas, distances, lengths, intersections, etc) must be calculated on the sphere, using more complicated mathematics. For more accurate measurements, the calculations must take the actual spheroidal shape of the world into account. There are several functions and operators that take as input or return as output a geography data type object. One very useful one, is the function ST_DWithin. This function returns true if the geographies are within the specified distance of one another. Units are in meters and measurement is defaulted to measure around spheroid.

2.2.2 Indexes

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

B-tree

PostgreSQL uses by default a B-tree index on a table column. B-tree [10] is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. More specifically, it is a balanced tree whose nodes contain pointers to a table's records and a number of keys. The keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a1 and a2. All values in the leftmost subtree will be less than a1, all values in the middle subtree will be between a1 and a2, and all values in the rightmost subtree will be greater than a2. Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data.

R-tree

R-trees [14] are tree data structures used for spatial access methods, such as indexing multi-dimensional information like geographical coordinates. The key idea of the data structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree; the ”R” in R-tree is for rectangle. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object; at higher levels the aggregation of an increasing number of objects. This can also be seen as an increasingly coarse approximation of the data set. R-tree's searching lgorithms use the bounding boxes to decide whether or not to search inside a

subtree. In this way, most of the nodes in the tree are never read during a search. Like B-trees, this makes R-trees suitable for large data sets and databases, where nodes can be paged to memory when needed, and the whole tree cannot be kept in main memory.

GiST

GiST [16] stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B-trees, R-trees and many other indexing schemes can be implemented in GiST. GiST tree nodes contain a pair of values in the form of (p, prt). The first value, p, is used as a search key and the second value, ptr, is a pointer to the data if the node is a leaf or a pointer to another node if the node is intermediate. The search key p represents an attribute that becomes true for all data that can be reached through the pointer ptr. Also, GiST index can handle any query predicate, as long as certain functions, that influence the behavior of the search keys, are implemented.

PostGIS can use GiST index at a table attribute. When a table column is a geography data type, then GiST will use an improved version of an R-tree index (R-tree-over-GiST scheme). PostgreSQL doesn't allow, at the most recent versions, the use of standard R-tree, because this type of index cannot handle attributes with size bigger than 8K and fail when a geography column is null.

2.3 HDFS

The Hadoop Distributed File System (HDFS) [22] is a distributed file system designed to run on commodity hardware. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance.

The NameNode is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself. Client applications talk to the NameNode whenever they wish to locate a file, or when they want to add/copy/move/delete a file. The NameNode responds the successful requests by returning a list of relevant DataNode servers where the data lives. A Datanode stores data in the HDFS. Client applications can talk directly to a DataNode, once the NameNode has provided the location of the data. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

2.4 HBase

HBase is an open source, non-relational (NoSQL), distributed database modeled after Google's BigTable [12]. It is developed as part of Apache Software Foundation's Apache Hadoop project and runs on top of HDFS.

2.4.1 Data model

HBase works best for sparse data sets, which are common in many Big Data use cases. Unlike relational database systems, HBase does not support a structured query language like SQL; in fact, HBase isn't a relational data store at all. HBase is a column-oriented database and follows a key/value data storage model. The main characteristics if the data model are the following:

- Table: HBase organizes data into tables as most database systems.

- Row: Within a table, data are stored according to its row. Rows are uniquely identified by a key. Row keys are treated as byte arrays, thus they do not have a specific data type.
- Column Family: Column families group data within a row, impacting their physical arrangement. They are stored together on disk, which is why HBase is referred to as a column-oriented data store. Column families must be defined up front, during table creation.
- Column Qualifier: Data within a column family are addressed via its column qualifier, or simply, column. Column qualifiers need not be consistent between rows. Like row keys, column qualifiers do not have a data type and are always treated as a byte array.
- Cell: A combination of row key, column family, and column qualifier uniquely identifies a cell. The data stored in a cell are referred to as that cell's value. Values also do not have a data type and are always treated as a byte array.
- Timestamp: Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If a timestamp is not specified during a write, the current timestamp is used. If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by HBase is configured for each column family. The default number of cell versions is three.

For example, a table in HBase can be the following:

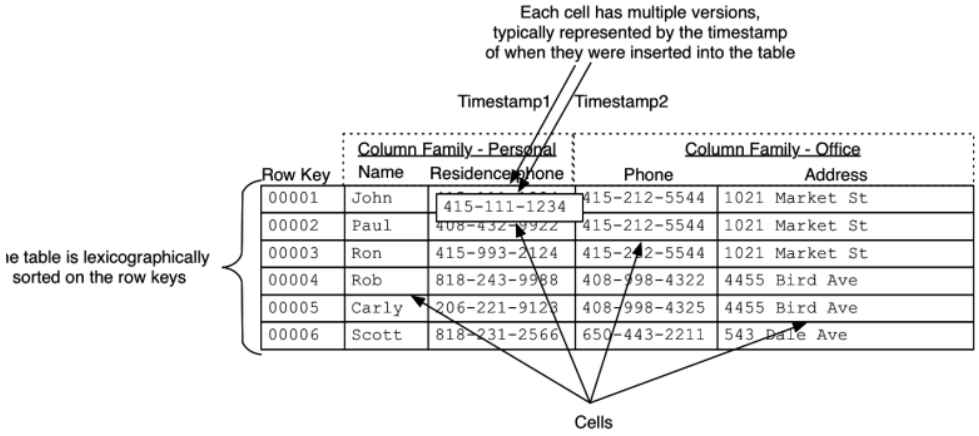

Figure 2.1: A table in HBase consisting of two column families

2.4.2 Architecture components

HBase has the following three major architecture components:

- Region server: Regions are parts of HBase tables, splitted accross the region servers of the distributed system and managed by a region server. Region server communicates with the client and handles data-related operations. Also, it handles read and write requests for all regions under it and decides the size of the regions.
- Master server: Master server assigns regions to the region servers and handles the load balancing of the regions across region servers. Moreover, it is responsible for schema changes and other metadata operations such as creation of tables and column families.
- Zookeeper: Zookeeper server is responsible to handle and synchronize the communication between the master and region servers, providing fault tolerance. More specifically, ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

2.4.3 Coprocessors

HBase coprocessors [6] enable distributed computation directly within the HBase server processes on the server's local data. The idea of HBase Coprocessors was inspired by Google's BigTable coprocessors. Coprocessors can be loaded globally on all tables and regions hosted by the region server, these are known as system coprocessors; or the administrator can specify which coprocessors should be loaded on all regions for a table on a per-table basis, these are known as table coprocessors. There are two categories of HBase coprocessors, the observer and the endpoint coprocessors. The idea behind observers is that we can insert user code by overriding upcall methods provided by the coprocessor framework. The callback functions are executed from core HBase code when certain events occur. Endpoints, on the other hand, are more powerful, resembling stored procedures. One can invoke an endpoint at any time from the client. The endpoint implementation will then be executed remotely at the target region or regions, and results from those executions will be returned to the client. In essence, coprocessors implement the idea of "move computation near the data", a method that google heavily utilizes to increase performance: it is much cheaper to transfer a small piece of code to be executed in parallel in various servers instead of moving data where the code is installed.

The Endpoint is an interface for dynamic RPC extension. The endpoint implementation is installed on the server side and can then be invoked with an HBase RPC. The client library provides convenience methods for invoking such dynamic interfaces. The application code client side performs a batch call. This initiates parallel RPC invocations of the registered dynamic protocol on every target table region. The results of those invocations are returned as they become available. The client library manages this parallel communication on behalf of the application, messy details such as dealing with retries and errors, until all results are returned (or in the event of an unrecoverable error). Then the client library rolls up the responses into a Map and hands it over to the application. If an unrecoverable error occurs, then an exception will be thrown for the application code to catch and take action.

2.5 Google Maps Directions API

The Google Directions API [2] is a service that calculates directions between locations using an HTTP request. A Directions API request takes the following form:

http://maps.googleapis.com/maps/api/directions/output?parameters

where output determines the type of file that will contain the answer to the HTTP request. The type of the output file can be either json of xml. The field parameters defines the parameters and their values that will come along with the HTTP request. Certain parameters are required while others are optional. As is standard in URLs, all parameters are separated using the ampersand (&) character. Required parameters are the origin and destination locations between which we request directions through the Directions API. These locations can be in the form of textual geographical coordinates as a value pair of latitude and longitude. Location can also be inserted as the address of the specific point. In this case, Directions service will geocode the string and convert it to a latitude/longitude coordinate in order to calculate directions. Other optional parameters, that can be defined, are the mode of transportation, which can be driving, walking, bicycling, or transit. For example, a complete HTTP request to the 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

2.5.1 JSON response file

JSON (JavaScript Object Notation) file format, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. Directions responses in json type format contain the following fields:

Status

The status field within the Directions response object contains the status of the request, and may contain debugging information to help you track down why the Directions service failed. The status field may contain the following values:

- OK, the response contains a valid result.
- NOT_FOUND, at least one of the locations specified in the request's origin, destination, or waypoints could not be geocoded.
- ZERO_RESULTS, no route could be found between the origin and destination.
- INVALID_REQUEST, the provided request was invalid. Common causes of this status include an invalid parameter or parameter value.
- REQUEST_DENIED, the service denied use of the directions service by your application.
- UNKNOWN_ERROR, a directions request could not be processed due to a server error.
- OVER_QUERY_LIMIT, the service has received too many requests from your application within the allowed time period. More specifically, users of the free Directions API are able to perform at most 2 HTTP request per second and 2500 HTTP requests per day to the API.

For example:

```
"status" : "OK"
```

Routes

Routes field is a JSON array, whose elements represent a different route between the origin and destination locations. Each route can contain the following fields:

- summary: a short textual description for the route, suitable for naming and disambiguating the route from alternatives.
- legs[]: an array which contains information about a leg (i.e. a part) of the route, between two locations within the given route. If there are no waypoints defined, the array will include one element, which will be the total route.
- waypoint_order: an array indicating the order of any waypoints in the calculated route.
- overview_polyline: a single points object that holds an encoded polyline representation of the route. This polyline is an approximate (smoothed) path of the resulting directions.
- bounds: the viewport bounding box of the overview_polyline.
- copyrights: the copyrights text to be displayed for this route.
- warnings[]: an array of warnings to be displayed when showing these directions.

For example:

```
   "bounds" : {
      "northeast" : {
         "lat" : 37.97818609999999,
         "lng" : 23.7762659
      },
      "southwest" : {
         "lat" : 37.9761428,
```

```
      "lng" : 23.7689148
   }
},
"copyrights" : "Map data ©2015 Google",
"legs" : [
   ... ]
"overview_polyline" : {
   "points" : "{exfFuxbpCg@fCkBpK}B~MkB|KIf@QG"
},
"summary" : Λεωφ". ΣτρατάρχουΑλεξάνδρουΠαπάγου  ",
"warnings" : [
   "Walking directions are in beta.    Use caution – This route may be missing
       sidewalks or pedestrian paths."
],
"waypoint_order" : []
```

legs[]

Each element in the legs array specifies a single leg of the journey from the origin to the destination in the calculated route. For routes that contain no waypoints, the route will consist of a single "leg," but for routes that define one or more waypoints, the route will consist of one or more legs, corresponding to the specific legs of the journey.

Each leg within the legs field(s) may contain the following fields:

- steps[]: an array of steps denoting information about each separate step of the leg of the journey.
- distance: the total distance covered by this leg, where value field contains the distance in meters and text field contains a human-readable representation of the distance, displayed in units as used at the origin
- duration: the total duration of this leg, where value field contains the duration in seconds and text field contains a human-readable representation of the duration.
- start_location: contains the latitude/longitude coordinates of the origin of this leg. Because the Directions API calculates directions between locations by using the nearest transportation option (usually a road) at the start and end points, start_location may be different than the provided origin of this leg if, for example, a road is not near the origin.
- start_address: contains the human-readable address (typically a street address) reflecting the start_location of this leg.
- end_location: contains the latitude/longitude coordinates of the given destination of this leg. Because the Directions API calculates directions between locations by using the nearest transportation option (usually a road) at the start and end points, end_location may be different than the provided destination of this leg if, for example, a road is not near the destination.
- end_address: contains the human-readable address (typically a street address) reflecting the end_location of this leg.

For example:

```
"distance" : {
    "text" : "0.7 km",
    "value" : 690
 },
 "duration" : {
    "text" : "7 mins",
    "value" : 434
 },
```

```
   ”end_address” : ”Iasona Maratou 509, Zografou 157 73, Greece”,
   ”end_location” : {
      ”lat” : 37.97818609999999,
      ”lng” : 23.7689477
   },
   ”start_address” : ”Papagou 147, Zografou 157 73, Greece”,
   ”start_location” : {
      ”lat” : 37.9761428,
      ”lng” : 23.7762659
   },
   ”steps” : [
        .... ]
```


steps[]

Each element in the steps array defines a single step of the calculated directions. A step is the most atomic unit of a direction's route, containing a single step describing a specific, single instruction on the journey. The step not only describes the instruction but also contains distance and duration information relating to how this step relates to the following step. Each step within the steps field(s) may contain the following fields:

- html_instructions: formatted instructions for this step, presented as an HTML text string.
- distance: the distance covered by this step until the next step.
- duration: the time required to perform the step, until the next step.
- start_location: the geographical coordinated of the location of the starting point of this step.
- end_location: the geographical coordinated of the location of the last point of this step.
- polyline: a single points object that holds an encoded polyline representation of the step. This polyline is an approximate (smoothed) path of the step.
- travel_mode: the mode of transportation, as it is defined in the mode parameter of the HTTP request.

For example:


```
   ”distance” : {
      ”text” : ”0.7 km”,
      ”value” : 680
   },
   ”duration” : {
      ”text” : ”7 mins”,
      ”value” : 427
   },
   ”end_location” : {
      ”lat” : 37.9781038,
      ”lng” : 23.7689148
   },
   ”html_instructions” : ”Head \u003cb\u003ewest\u003c/b\u003e on \u003cb\Λεωφu003e.
        ΣτρατάρχουΑλεξάνδρουΠαπάγου  \u003c/b\u003e toward \u003cb\Γρu003e. Κουσίδη\u003c/b\
        u003e”,
   ”polyline” : {
      ”points” : ”{exfFuxbpCQv@ETCLKj@CPYxAi@|Cc@fCm@lDg@pCCNc@nC_AnFCHYhBMx@If@”
   },
   ”start_location” : {
      ”lat” : 37.9761428,
      ”lng” : 23.7762659
   },
   ”travel_mode” : ”WALKING”
```

2.5.2 Encoded Polyline Algorithm Format

The JSON response file contains an encoded polyline as a representation of the route. This representation is the result of encoding the geographical coordinates of the intermediate points of the route. Google Maps API follows a specific encoding algorithm [3] of the latitude and longitude of any point. The steps for encoding such a signed value indicating the latitude or longitude are specified below:

1. Multiply the initial signed value by 1e5, rounding the result.
2. Convert the result to it's binary equivalent value, using it's two complement.
3. Left-shift the binary value one bit.
4. If the original decimal value is negative, invert this encoding.
5. Break the binary value out into 5-bit chunks (starting from the right hand side.
6. Place the 5-bit chunks into reverse order.
7. OR each value with 0x20 if another bit chunk follows.
8. Convert each value to decimal.
9. Add 63 to each value.
10. Convert each value to its ASCII equivalent.

Following these steps in reverse we can decode an encoded polyline into the appropriate list of geographical coordinates and extract the GPS traces that define the route.

2.6 Google Maps Static Maps API

The Google Static Map service [4] creates a map based on URL parameters sent through a standard HTTP request and returns the map as an image that can be displayed on a web page, or accessed through a URL. A simple HTTP request to the Google Static Maps API takes the following form:

https://maps.googleapis.com/maps/api/staticmap?parameters

The Static Maps API defines map images using the following URL parameters:

Location Parameters

- center: defines the center of the map, equidistant from all edges of the map. This parameter takes a location as either a comma-separated latitude,longitude pair or a string address identifying a unique location on the face of the earth.
- zoom: defines the zoom level of the map, which determines the magnification level of the map.

Map Parameters

- size: the rectangular dimensions of the map image. This parameter takes a string of the form {horizontal_value}x{vertical_value}.
- scale: affects the number of pixels that are returned. scale=2 returns twice as many pixels as scale=1 while retaining the same coverage area and level of detail.
- format: the format of the resulting image. Possible formats include PNG, GIF and JPEG types.

- maptype: the type of map to construct. There are several possible maptype values, including roadmap, satellite, hybrid, and terrain.

- language: the language to use for display of labels on map tiles.

Feature Parameters

- markers: one or more markers to attach to the image at specified locations. This parameter takes a single marker definition with parameters separated by the pipe character (|).

- path: a single path of two or more connected points to overlay on the image at specified locations. This parameter takes a string of point definitions separated by the pipe character (|). In addition, the encoded polyline representation can be used instead of multiple points, as long as the prefix enc: is used in the value of the parameter path.

- visible: one or more locations that should remain visible on the map, though no markers or other indicators will be displayed.

For example, a HTTP request at Google Static Maps API can be the following:

https://maps.googleapis.com/maps/api/staticmap?&size=1000x1000&markers=label:A|17.7466,-64.703&markers=label:C|17.7453,-64.7019&path=color:blue|enc:gcikBvh|iKFJ??nAaAFGjA}@rAiAJK

2.7 Mathematical Background

2.7.1 Random Variables

A random variable X is a measurable function from the set of possible outcomes Ω, X: Ω→A or X: Ω→R, where A is a subset of the set of real numbers. If A is a set of discrete values, such as the set of integer numbers, then the random variable is discrete. If A is a set of infinite possible values, then the random variable is continuous. In general, a random variable is a variable whose value is subject to variations due to chance. A random variable can take on a set of possible different values (similarly to other mathematical variables), each with an associated probability if the variable is discrete, or with a probability density function if the variable is continuous. The function of the random variable is called probability distribution.

2.7.2 Uniform Distribution

The discrete uniform distribution is a symmetric probability distribution whereby a finite number of values are equally likely to be observed; every one of n values has equal probability 1/n.

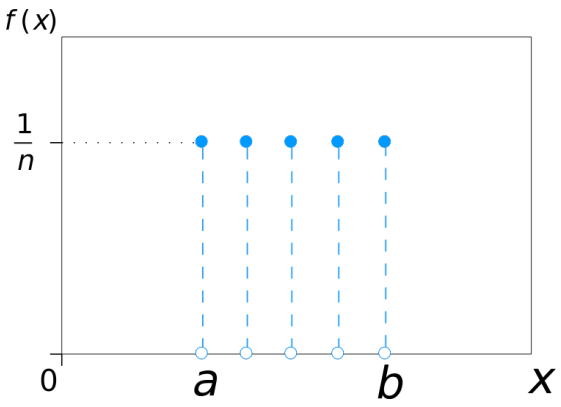

Figure 2.2: Discrete uniform probability mass function

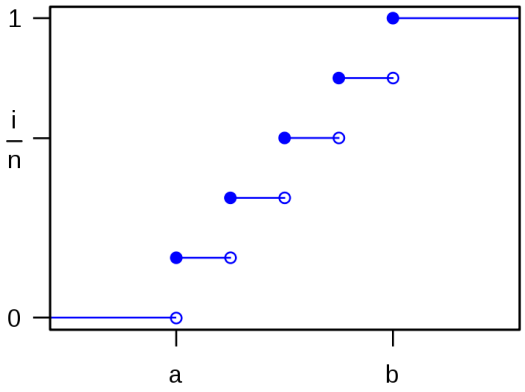

Figure 2.3: Discrete uniform cumulative distribution function

2.7.3 Normal Distribution

Normal distribution, known also as Gauss distribution, refers to continuous random variables. The normal distribution is remarkably useful because of the central limit theorem. In its most general form, it states that averages of random variables drawn from independent distributions are normally distributed. The probability density of the normal distribution is:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

where μ is the mean of the distribution and σ is the standard deviation of the distribution.

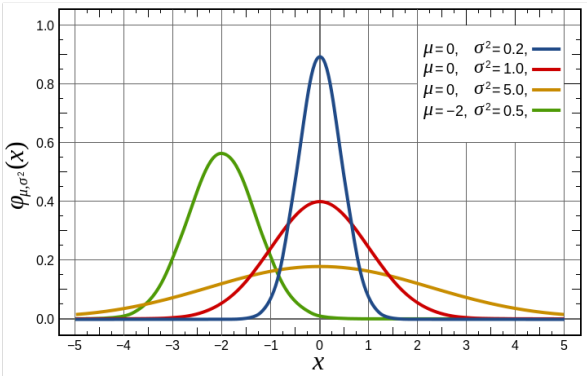

Figure 2.4: Probability density function

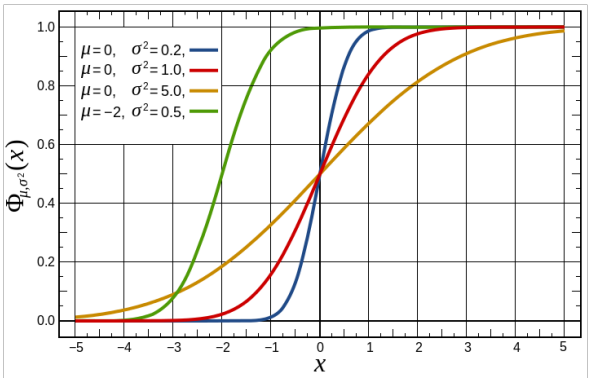

Figure 2.5: Cumulative distribution function

According to an empirical rule, about 68% of values drawn from a normal distribution are within one standard deviation σ away from the mean; about 95% of the values lie within two standard deviations; and about 99.7% are within three standard deviations. This fact is known as the 68-95-99.7 rule, or the 3-sigma rule.

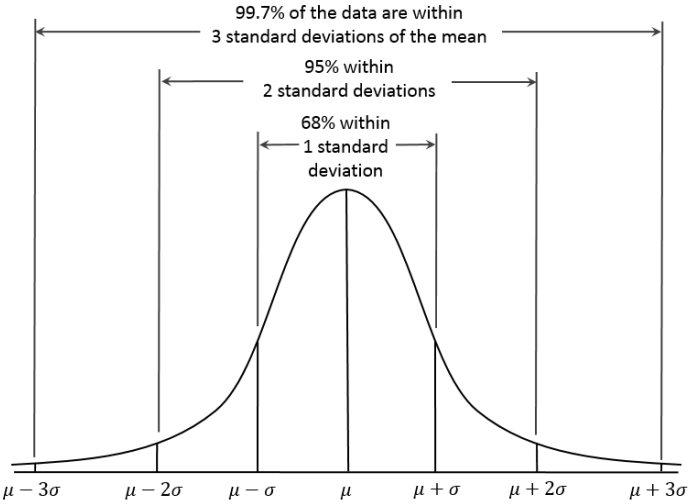

Figure 2.6: 3-sigma empirical rule for the normal distribution

2.7.4 Bernoulli Distribution

The Bernoulli distribution is the probability distribution of a random variable which takes value 1 with success probability p and value 0 with failure probability q=1-p. It can be used, for example, to represent the toss of a (not necessarily fair) coin, where ”1” is defined to mean ”heads” and ”0” is defined to mean ”tails” (or vice versa).

If X is a random variable with this distribution, we have:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The experiment is called fair if p=0.5 and can represent the toss of a fair coin.

Chapter 3

Generator

3.1 Storage of source data

The generator of spatio-temporal and textual data uses real point of interests on the map as a source of data. These data where extracted from the online travel service TripAdvisor, using an already implemented crawler [17]. These points of interest 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 136409, as they were extracted by a 13GB response json file from the crawler. The points where stored in PostgreSQL. The database schema used, contains two tables, one for the points of interest and one for the reviews and ratings of these points and is the following:

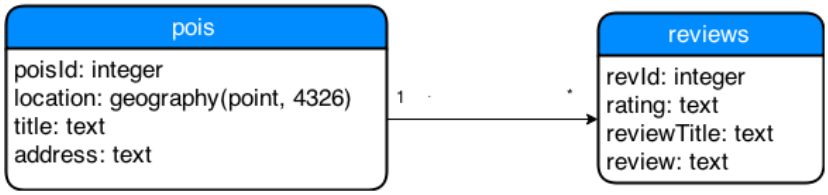

Figure 3.1: Database schema for source data

More specifically, the table for the points of interest has the following attributes:

- poisId: identifying number for the point of interest, primary key.
- location: geographical coordinates of the point. Usage of the geographic data type introduced by the extension PostGIS of PostgreSQL.
- title: the name of the point of interest.
- address: the address of the point on the map.

A point of interest can have many reviews, issued by different users, thus the two tables are associated by 1-to-many. The reviews table contains the following attributes:

- revId: the identifying number of the point of interest that the review refers to, foreign key.
- rating: rating of the point in a scale of 1 to 5.
- reviewTitle: the title of the review.
- review: the text of the review.

After storing all points of interest into the database, we create an B-tree index to the attribute poisId of the table pois and to the attribute revId of the table reviews, as well. Additionally, we create a GiST index to the attibute location of the table pois. As location has a geographic data type, GiST will implement an improved R-tree index, as described in section 2.2.2. In this way, searching points of interest according to their location or identifying number, can be done fast and efficiently.

3.2 Generator attributes

The following class diagram indicates the main attributes used in the description of the generator's design.

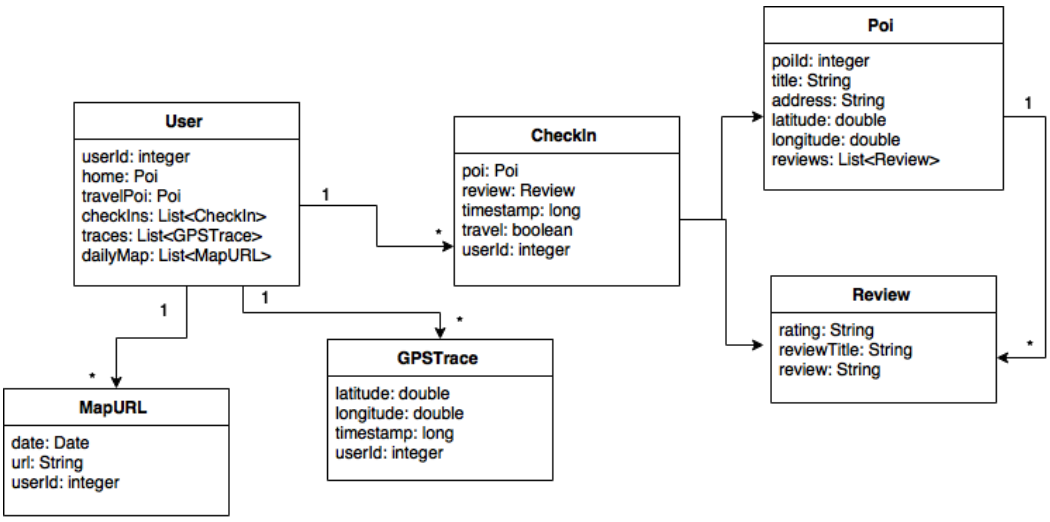

Figure 3.2: Class diagram of generator's attributes

3.2.1 User

The user produced by the generator has the following fields:

- userId: user's identifier.
- home: user's home defined as Poi.
- travelPoi: the central location of the current trip, defined as Poi.
- checkIns: list of user's check-ins during the days for which the generator produced data.
- traces: list of GPS traces indicating the user's daily routes during the days for which the generator produced data.
- dailyMap: list of URLs to static maps images showing the daily user routes.

3.2.2 Check-in

A check-in to a point of interest (POI) has the following fields:

- poi: the point of interest (POI) where the check-in was made.
- review: the review that the user made for the specfic POI on the check-in.
- timestamp: the time when the check-in was made, in representation of type long.
- travel: boolean value indicating whether the check-in was made during a user's trip.
- userId: the user's identifier.

3.2.3 GPS trace

A user's route consists of multiple GPS traces, which illustrate the path that the user followed. A GPS trace contains the following information:

- (latitude, longitude): the geographical coordinates of the point on the map where the GPS trace was taken.
- timestamp: the time when the GPS trace was taken.
- userId: user's identifier whose route contains the specific GPS trace.

3.2.4 Static route map - MapURL

A user's daily route is depicted on a static map, using Google Static Maps API. The map shows the points of interest that the user visited using markers, and the exact path from one POI to another using blue continuous lines. The image of the map can be accessed through a respective URL.

- url: the URL that directs to the image of the map.
- date: the date when the user walked the route showed on the map.
- userId: the identifier of the user who walked the specific route that date.

3.2.5 Point of interest - Poi

A point of interest has the following attributes:

- poiId: point's identifier.
- title: point's name.
- address: point's address.
- (latitude, longitude): point's geographical coordinates.
- reviews: a list with the available reviews for the specific point.

3.2.6 Review

A review to a point of interest has the following fields:

- rating: rating of the POI in scale 1 to 5.
- reviewTitle: the title of the review.
- review: the text of the review.

3.3 Input parameters

The generator takes as input the next parameters:

- userIdStart: identifier of the first user for whom the generator will create daily routes.
- userIdEnd: Respectively, the identifier of the last user created.
- chkNumMean: The mean of the number of daily check-ins to points of interest, which will follow a normal distribution.

- chkNumStDev: The standard deviation of the normal distribution that the number of daily check-ins follow.
- chkDurMean: The mean of the duration that each visit will last, which will follow a normal distribution.
- chkDurStDev: The standard deviation of the normal distribuition of the duration of user's visits.
- dist: The maximum distance in meters in which a user can walk from one point of interest to the next one.
- maxDist: The maximum distance in meters from a user's home, in which a user can walk every day.
- startTime: The time when the first check-in of the day will occur.
- endTime: The time when the last check-in of the day will occur.
- startDate: The first day starting from which the generator will create daily routes.
- endDate: Respectively, the last day that the generator will create daily routes.
- outCheckIns: The output file storing the daily check-ins of all users created.
- outTraces: The output files storing the GPS traces of the daily routes of all users created.
- outMaps: The output file storing the URLs for the daily maps depicting the daily routes of all users created.

3.4 Generator's configuration parameters

The generator's main function is to create user's daily routes, which will contain information simulating real social media data. These daily routes consist of visits (check-ins) to points of interest and paths that the user followed from one point to the next one. The generator takes as input the values of the corresponding parameters and based on the stored source data, creates daily routes for (userIdEnd - userIdStart + 1) users between the start and end dates. The decisions that the generator makes, such as which and how many points the user will visit a specific day, or the duration of each visit, are made using random factors. In this way, the produced data can be as realistic as possible. The generator stores the created data in three files, one for all the user check-ins (outCheckIns), one for the all the GPS traces (outTraces) and one for the all the daily maps (outMaps).

Briefly, the daily routes that the generator produces are designed in such a way that allows each user to visit places by walking from one point to the other. Each day, the user leaves his home and visits certain places that are in walking distance. The number of visits per day is decided in a random way. Moreover, the user stays for a couple of hours, also defined in a random way, at a point of interest and leaves a rating and review for that specific POI. Moving on, he walks to the next point, following the path that Google Directions API provides, and checks in to the next place. The generator enables a user to travel, following the same daily route strategy in the location of his trip.

3.4.1 Home

The location of a user's home is the center around which he walks every day. The points of interest that a user visits are selected in a random way, thus there can exist routes that don't make sense. For example, a user can walk one day in Greece, the next day in France and the other day again back in Greece. 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 choice of this point is done using a generator of random numbers which follow a uniform distribution. The range of the distribution are the total number of source POIs stored in PostgreSQL.

3.4.2 Trip

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 number that follow a uniform distribution. The range of the distribution is the number of available source POIs.

As far as the duration of each trip is concerned, each user can travel for a time interval that equals the 10% of the total time interval for which the generator 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.

3.4.3 Check-in POI

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 whio follow a uniform distribution. The range of this distribution is the number of available POIs who 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 donw 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 in the current 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.

3.4.4 Check-in 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.

3.4.5 Path between check-ins

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, he 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 paramater mode into walking, because 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, as it was described in section 2.4.2. 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, representating 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.

3.4.6 Timestamp

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 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 splitted 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.

3.4.7 Static map

The generator 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 (outMaps), so that the image of the map can be accessible by the users of the generator.

For example, a request to Google Static Maps API, as it is created by the generator, can be the following:

```
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
```

The map that corresponds to the above URL is:

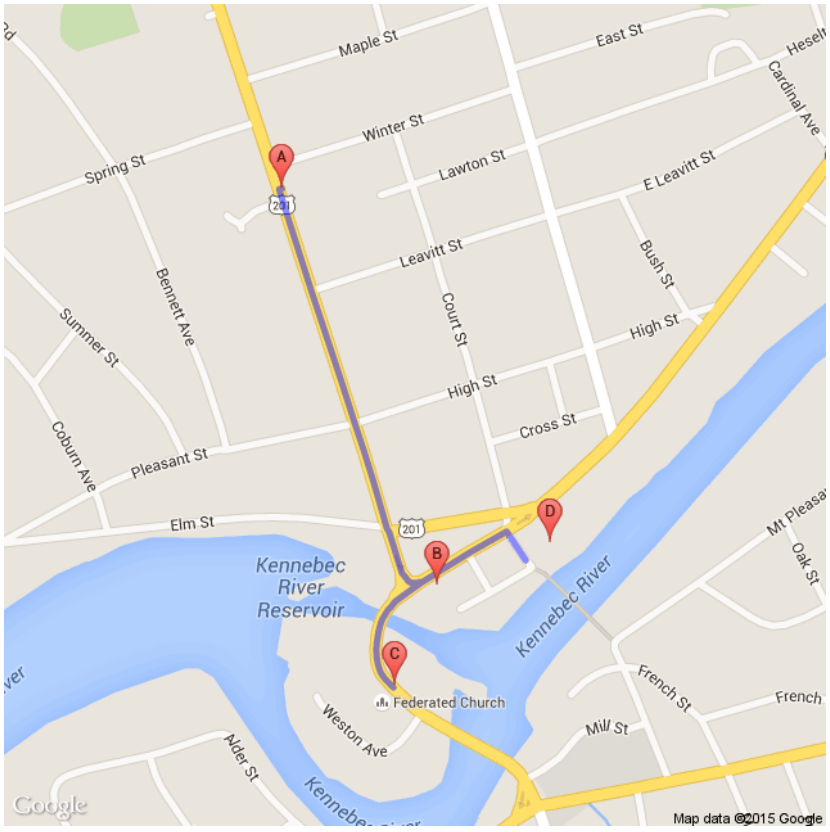

Figure 3.3: Example of static map image

Chapter 4

Dataset

4.1 Generator input parameters

We were able to create a large dataset using the generator. More specifically, we set the input parameters of the generator to the following values:

chkNumMean = 5 chkNumStDev = 2

The number of daily check-ins will be random and 95% of these random values will be between 1 and 9, according to the 3-sigma empirical rule for the normal distribution.

chkDurMean = 2 chkDurStDev = 0.1

The duration of each visit will be random and 95% of these random values will be between 1.8 and 2.2 hours.

maxDist = 50000.0 dist = 500.0

Each user will be able to visit places that are in 50000 meters radius from his home or travel location. Also, he is allowed to walk in a 500 meters radius from the one place he visits to the next one.

startTime = 9 endTime = 23

Each user will visit the first place of the day at 9 am. Moreover, the last daily check-in should take place no later than 11 pm.

startDate = 01-01-2015 endDate = 03-01-2015

The generator will produce daily routes for the time period between the 1st of January 2015 and the 1st of March 2015.

Therefore, the generator run with the specific input parameters in order to create a large dataset of users and their respective daily routes.

4.2 Generator deployment setup

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 input parameters. 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 cluster used recourses located at the Computing Systems Laboratory at NTUA. 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.

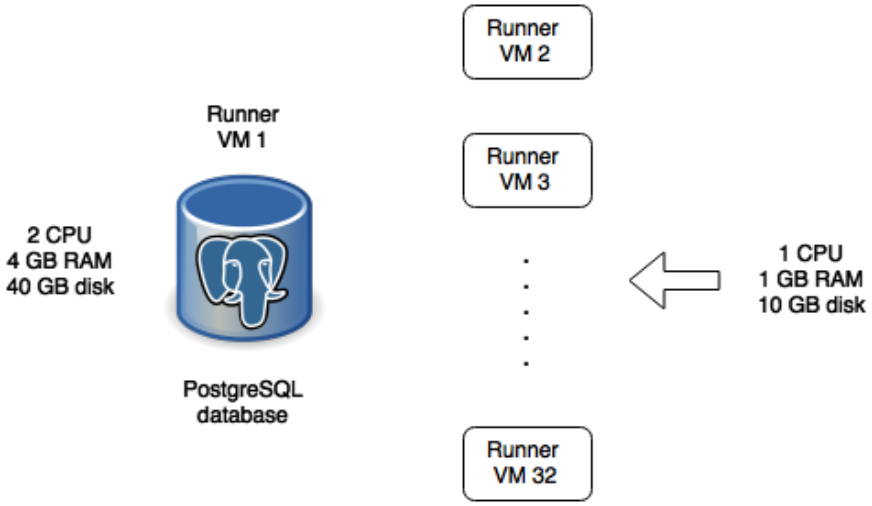

Figure 4.1: Generator execution architecture

In this way, we were able to run the generator on each VM collecting 448 users per day, creating 14 users per VM with the specified input parameters. 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. More specifically, the generator created the following dataset:

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

Table 4.1: Dataset produced by the generator

4.3 Dataset storage model

The dataset created by the generator consists of 641 MB data of check-ins and 2.4 GB data of GPS traces. 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. The overall dataset of check-ins, GPS traces and friends was stored into an HBase distributed database using the following data model.

4.3.1 Friends Table

The data about friends consist of user ids. For example, user no.1 has friends the users no.2, no.3 etc. This information is parsed from the friend graph and inserted into HBase in byte form using the following schema:

- Row: each row holds all the users that are friends of the user whose id is the key of the row.
- Column Family: there is one column family 'friends' including all the friends of one user.

- Column Qualifier: the qualifier that represents a single friend is the friend's user id.

Row	Friends		
Key	user id #1	user id #2	user id #3
1	145	2901	1204
2	3423	1023	965

Table 4.2: Example of HBase Table 'friends'

4.3.2 Check-ins Table

The user's check-ins are parsed from the corresponding generator's output file and inserted to HBase in byte form using the following schem:

- Row: each row holds all the check-ins of the user whose user id is the row key.
- Column Family: there is one column family 'checkIns' which holds all the check-ins of each user.
- Column Qualifier: the qualifier that represents a single user check-in is its timestamp.

Row	checkIns		
Key	timestamp #1	timestamp #2	timestamp #3
1	1420117200700	1420124471700	1420189200700

Table 4.3: Example of HBase Table 'check-ins'

4.3.3 GPS traces Table

The user's GPS traces are parsed from the corresponding generator's output file and inserted to HBase in byte form using the following schem:

- Row: each row holds all the GPS traces of the user whose user id is the row key.
- Column Family: there is one column family 'gpsTraces' which holds all the GPS traces of each user.
- Column Qualifier: the qualifier that represents a single user GPS trace is a string of the geographical coordinates combined with the timestamp of the GPS trace.

Row	gpsTraces	
Key	qualifier #1	qualifier #2
1	"(25.694, 79.836) 1420117200700"	"(25.69274, 79.8394) 1420124471700"

Table 4.4: Example of HBase Table 'gps-traces'

4.4 Dataset storage architecture

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. The cluster is accomodated on a OpenStack managed cloud service located at the Computing Systems Laboratory in NTUA.

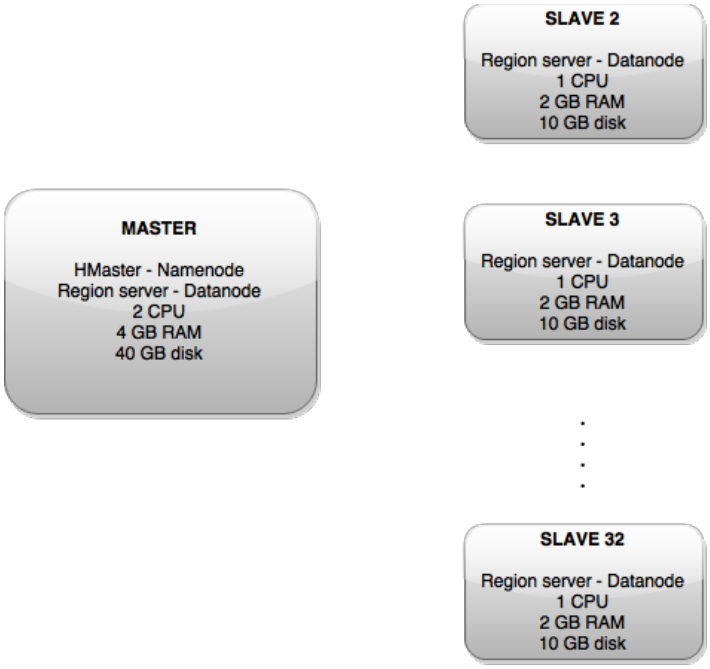

Figure 4.2: HBase cluster architecture

Chapter 5

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. In this way the computation of intermediate results and other complex calculations is transfered to the region servers that contain the respective data, decongesting the client from a heavy computational load.

5.1 Query implementation

We implemented the following queries over the data of tables 'friends' and 'check-ins' using HBase coprocessors.

5.1.1 Most visited POI query

This query contains the following question to the available data:

"Get the most visited points of interest of a certain user's friends"

The query is implemented in the following steps:

1. The client calls the coprocessor that returns the friends of the specified user. The coprocessor runs on the region server that contains the row of the table 'friends' that has as key the user id of the desired user. The client receives back a list of user id's that represent his friends.

2. The client splits the list of friends into sections according to the initial split of the 'check-ins' table into 32 regions. In this way, each splitted friends list will contain row keys that belong to only one region server.

3. The client issues a call to the coprocessor that calculates the most visited POIs for every splitted friend list. More specifically, the client starts a new thread that is responsible for calling the coprocessor and getting the result back. In this way, the client issues parallel calls to the coprocessor and the calculations to the respective region servers are done simultaneously.

4. The coprocessor that runs on each region server gets from the 'check-ins' table the rows that include all the check-ins of the user friends that are assigned to this region. He then iterates over each row in order to store and count how many times the user's friend has visited each POI. The POIs are stored into a hash table for faster calculations. At the end, he iterates over the hash table in order to get the POI that the specific friend visited the most times.

5. Finally, the client merges the results that he got back from each coprocessor call and forms the final result, which contains the place and how many times each of his friends went to their most visited POI.

5.1.2 News Feed query

This query contains the following question to the available data:

”Get the check-ins of all the friends of a specific user for a certain day into chronological order”

The query is implemented in the following steps:

1. The client calls the coprocessor that returns the friends of the specified user. The coprocessor runs on the region server that contains the row of the table ’friends’ that has as key the user id of the desired user. The client receives back a list of user id’s that represent his friends.

2. The client splits the list of friends into sections according to the initial split of the ’check-ins’ table into 32 regions. In this way, each splitted friends list will contain row keys that belong to only one region server.

3. The client issues a call to the coprocessor that calculates the news feed for every splitted friend list. More specifically, the client starts a new thread that is responsible for calling the coprocessor and getting the result back. In this way, the client issues parallel calls to the coprocessor and the calculations to the respective region servers are done simultaneously.

4. The coprocessor that runs on each region server gets from the ’check-ins’ table the rows that have as key the friend’s ids. Moreover, there are certain columns that are selected from each row. These are the ones that are between two certain timestamps, as the columns have as qualifiers the timestamp of the check-in. The start timestamp is the one that corresponds to the UTC timestamp conversion of the start of the specific day at 12 am and the end timestamp corresponds to the end of the day at 12 am of the next day. In this way, the coprocessor returns only the check-ins of every friend that were made the intended day.

5. The client merges the the resuls that he got back from the multiple coprocessor calls into one list of user check-ins. Finally, the client sorts this list in order to present the check-ins of the user’s friends into chronological order.

5.1.3 Correlated Most Visited POI query

This query contains the following question to the available data:

”Get the number of times that a user’s friends have visited the user’s most visited POI”

The query is implemented in the following steps:

1. The client calls the coprocessor that returns the friends of the specified user. The coprocessor runs on the region server that contains the row of the table ’friends’ that has as key the user id of the desired user. The client receives back a list of user id’s that represent his friends.

2. The client splits the list of friends into sections according to the initial split of the ’check-ins’ table into 32 regions. In this way, each splitted friends list will contain row keys that belong to only one region server.

3. The client calls the most visited POI coprocessor in order to get the most visited POI of the specific user.

4. The client issues a call to the coprocessor that calculates the correlated most visited POI for every splitted friend list. More specifically, the client starts a new thread that is responsible for calling the coprocessor and getting the result back. In this way, the client issues parallel calls to the coprocessor and the calculations to the respective region servers are done simultaneously.

5. The coprocessor that runs on each region server gets from the 'check-ins' table the rows with the check-ins of all the friends that are on that region. Then he iterates over every check-in and checks whether it's location is the same with the location of the most visited POI of the original user. In this way it counts how many times each friend went to that place and returns that counter as a result.

6. The client merges the resuls that he got back from the multiple coprocessor calls and presents to the user which friends of him went to his most visited POI and how many times they visited that place.

5.2 Queries execution

5.2.1 Workload

Using the above queries we created a 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 using a generator of random numbers that follows a uniform distribution. The range of this distribution if the total number of available users, which is 9464.

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. corellated 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.

5.2.2 Workload generation setup

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 that was described in section 4.1.

The workload is executed in the following setup:

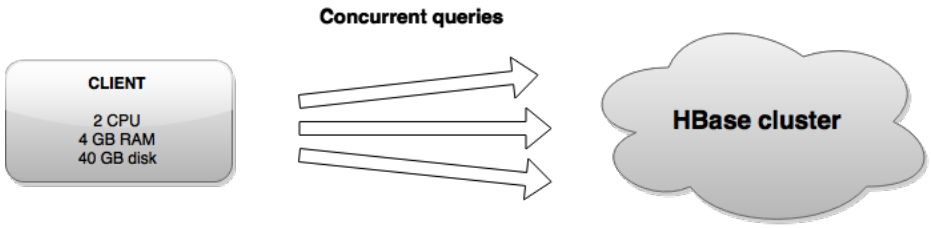

Figure 5.1: Query execution architecture

5.2.3 Scalability testing

Using the above architecture we performed a scalability testing, in order to evaluate how HBase handles the above workload for the specified dataset storage model in different cluster sizes. We calculated the latency and throughput of the system. More specifically, latency is the mean execution time of the queries. Throughput is the number of queries executed per second.

$$latency = \frac{\text{total execution time of all queries}}{\text{number of queries}} sec$$

$$throughput = \frac{\text{number of queries}}{\text{maximum query execution time}} queries/sec$$

We calculated the latency and throughput of the system for an increasing number of concurrent queries. We started with the HBase cluster having 32 nodes, as described previously. 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 and region servers to the desired number. Both HDFS and HBase offer commands in order to achieve the resize of the cluster by moving data and regions into the remaining nodes, preventing data loss and ensuring that the regions will be data balanced.

The scalability testing can be presented in the following tables and corresponding plots:

Queries no.	32 nodes	24 nodes	16 nodes	8 nodes	4 nodes
5	2.4	2.8	2.4	2.6	3
10	2.5	2.7	2.7	3.2	4.6
15	3.2	3.06	3.6	3.73	5
20	3.4	3.9	4.2	4	5.95
25	5	5.32	5.56	5.76	6.56
30	5.5	5.76	6.36	6.03	7.66
35	6.37	6.08	7.06	7.51	9.91
40	6.97	7.52	8.05	8.35	10.725
45	7.55	8.31	9.02	10	10.86
50	8.34	9.02	10.32	11.2	13.6
55	8.96	10.14	11.21	11.45	13.85
60	10.26	10.88	11.3	12.58	15.98

Table 5.1: Scalability testing - Latency

The 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. Therefore, the remaining HBase region servers have to handle more workload as opposed to when the cluster size was bigger. More specifically, since the data on the tables 'friends' and 'check-ins' are splitted into 32 regions in order to be equally distributed in the 32 nodes cluster, when we reduce the nodes, a region server can end up with more than one region from each table. In this way, the region server will have to handle more coprocessor calls as he has to server calculations for more data than before. Consequently, the execution time of the queries increases as there are fewer region servers and datanodes. Additionally, the latency increases as the client sends more concurrent queries. This happens due to the fact that as the number of concurrent queries elevate the servers cannot resolve them simultaneously and many of them have to wait in the servers 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.

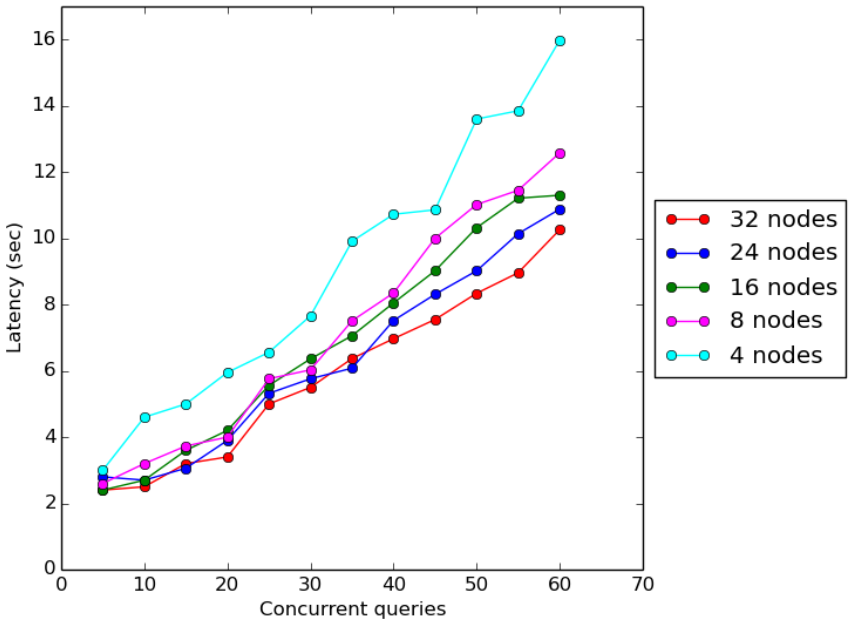

Figure 5.2: Scalability testing - Latency

Queries no.	32 nodes	24 nodes	16 nodes	8 nodes	4 nodes
5	1.66	1.66	1.66	1.66	1.25
10	3.33	2.5	2	1.8	1.66
15	3.75	3.45	3.2	2.9	2.54
20	4	3.9	3.5	3	2.5
25	4.46	3.77	3.46	3.47	2.77
30	5	4.28	3.75	3.33	2.72
35	5	4.375	3.9	3.78	2.6
40	5.2	4.44	4	3.63	2.85
45	5.425	4.5	4.09	3.75	2.8125
50	5.55	4.54	4.16	3.84	2.63
55	5.5	4.58	4.13	3.66	2.65
60	5.2	4.61	4	3.52	2.72

Table 5.2: Scalability testing - Throughput

Throughput is the number of queries executed per second. It is indicative of the performance of the system. For the current setup and workload, 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. Moreover, we can see that for a specific cluster size the throughput increases as the number of concurrent queries elevate. This indicates that the system can handle a bigger load without dropping in performance. However, there is a certain limit to the throughput that prevents it from increasing even more. In every real time system exists that limit and indicates that the system

cannot handle more queries per second for the specific cluster size. This limit is different according to the number of nodes, while a bigger cluster size can have bigger throughput. Finally, we can see that this limit in the throughput is not a fixed value for every cluster size. This is due to the fact that the workload is not preset and can variate according to the randomly selected user factor. Additionally, cache affects 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 datanode'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 that accomodates the VMs. However, those variations to the maximum value of the throughput are small and don't prevent us from seeing a stable limitation to the system's throughput.

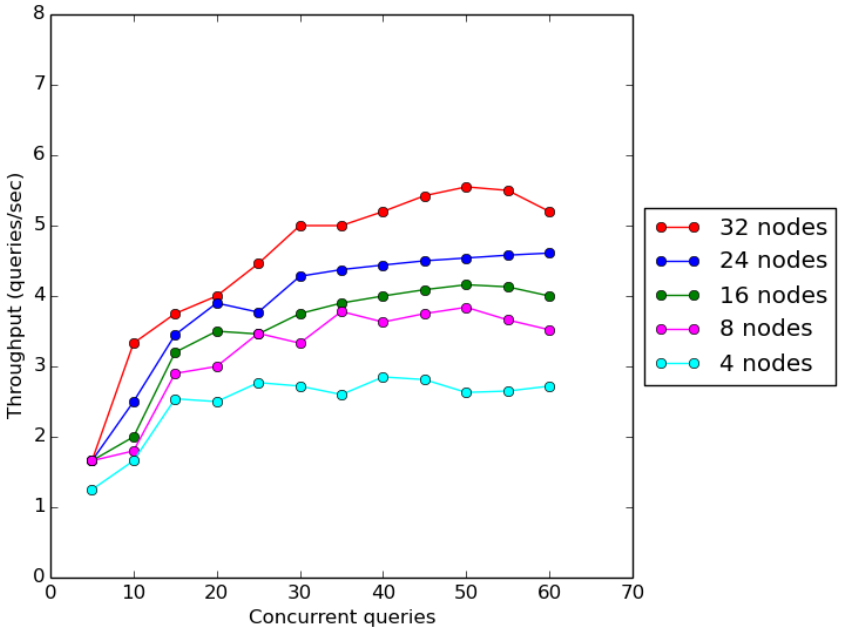

Figure 5.3: Scalability testing - Throughput

In conclusion, the HBase cluster that contains the social networking data produced by the generator and stored with the data model described previously, is scalable as it has the expected behavior of common scalable systems. The construction and execution of the workload that contains the three types of queries, allows as to test the scalability of the HBase cluster by calculating the latency and throughput.

Chapter 6

Conclusion

6.1 Summary

In this diploma thesis we were able to create a generator of spatio-temporal and textual social media data. We used real points of interest and ratings and reviews for these points as those were crawled by real users of the online travel service TripAdvisor. The utilization of these real points and the directions obtained from Google Directions API resulted in the creation of realistic routes, check-ins and GPS traces. Also, the fact that the generator uses random factors in order to take decisions about the place, duration and number of daily visits enhances the realistic aspect of the generated data. Additionally, most of these factors are instantiated by the corresponding input parameters, so that the generator can produce datasets with characteristics that vary. Therefore, we were able to create users that resemble real users found in well-known online social networking services.

The daily execution of the generator for a significant period of time resulted in the assemblance of a Big Data dataset of users with daily routes and check-ins to points of interest. More specifically, the generator created 9464 users, 1586537 check-ins and 38800019 GPS traces, which sum up to 3 GB data. In this way, we were able to re-create realistic social media data that we couldn't have access to due to privacy restrictions imposed by current social networking services.

With these data available, we were able to evaluate the distributed storage system that HBase provides and is used by a significant number of such social networking services. For this reason, we stored the created dataset into an HBase cluster using an appropriate data model and by ensuring that the cluster will be data balanced. Then, we implemented several queries over the available data that are common to such services. The queries were implemented using HBase coprocessors, which move the computation of intermediate results and other complex calculations to the region servers that contain the respective data, decongesting the client from a heavy computational load. We, then, created a workload with these types of queries in order to test the scalability of the HBase cluster. We measured the latency and throughput of the HBase cluster while changing the number of concurrent queries and cluster size. We came to the conclusion that the HBase cluster which contains the social networking data produced by the generator and stored with the specific data model, is scalable as it has the expected behavior of common scalable systems. Therefore, HBase provides indeed good performance and storage for Big Data social media services.

6.2 Related Work

There are already generators that produce social media data. One such generator is DATAGEN, which is a social network data generator, provided by LDBC-SNB (Linked Data Benchmark Council - Social Network Benchmark [11]). The LDBC-SNB aims to provide a realistic workload, consisting of a social network and a set of queries. DATAGEN is a synthetic data generator responsible for generating datasets for the three LDBC-SNB workloads: the Interactive, the Business Intelligence and the Analytical. DATAGEN mimics features of those found in real social network. They create a synthetic dataset due to the fact that it is difficult to find data with all the scaling characteristics their benchmark requires and collecting real data can be expensive or simply not possible due to privacy

concerns.

Also, there is LFR-Benchmark generator [18] which provides the network structure of a social network, being able to mimic some of the properties of real social media. The LFR generator produces social network graphs with power-law degree distribution and with implanted communities within the network.

6.3 Future Work

There are several ways in which the current diploma thesis could be extended. As fas the generator is concerned, the produced data could be enhanced by also adding more textual data such as Facebook posts or Twitter tweets. Also, the generator could use a photo and video pool in order to simulate the upload of photos and videos, which is a basic functionality of most social networking services. As far as the generator execution is concerned, the generator could be instantiated with other values in order to create a more polymorphic dataset. Also, the usage of Google Maps API for Work enables 100000 directions requests per 24 hour period, which is significantly more than the limit of 2500 daily requests for the users of the free API. Therefore, the generator can be used more efficiently for users subscribed to the Google Work services.

Moreover, regarding the implemented queries, there are many more that can be implemented. Also, there could be queries over the data of GPS traces which are created by the generator. In this way, the created workload can be more efficient into the scalability testing of the HBase cluster. Finally, other distributed storage and management systems could be evaluated in a similar way, such as Hadoop.

Bibliography

[1] Twitter statistics. https://about.twitter.com/company. [Online, accessed June 2015].

[2] Google directions api. https://developers.google.com/maps/documentation/directions/. [Online, accessed June 2015].

[3] Encoded polyline algorithm format. https://developers.google.com/maps/documentation/utilities/polylinealgorithm. [Online, accessed June 2015].

[4] Google maps static maps api. https://developers.google.com/maps/documentation/staticmaps/. [Online, accessed June 2015].

[5] Twitter re-architect. https://blog.twitter.com/2013/new-tweets-per-second-record-and-how. [Online, accessed June 2015].

[6] Apache hbase coprocessors. https://hbase.apache.org/book.html#cp. [Online, accessed June 2015].

[7] Institutions using hbase. http://wiki.apache.org/hadoop/Hbase/PoweredBy. [Online, accessed June 2015].

[8] Institutions using hadoop. https://wiki.apache.org/hadoop/PoweredBy. [Online, accessed June 2015].

[9] Databases - wikipedia. https://en.wikipedia.org/wiki/Database. [Online, accessed June 2015].

[10] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[11] Peter Boncz. Ldbc: Benchmarks for graph and rdf data management. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, IDEAS '13, pages 1–2, New York, NY, USA, 2013. ACM.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[13] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[14] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[15] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[16] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[17] Ioannis Ioannou. Ανάλυση Συναισθήματος Σε Μεγάλο Όγκο Δεδομένων Κειμένου Με Χρήση Κατανεμημένων Τεχνικών Μηχανικής Εκμάθησης, July 2014. Diploma Thesis, National Technical University of Athens.

[18] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 78(4), 2008.

[19] Ioannis Mytilinis, Ioannis Giannakopoulos, Ioannis Konstantinou, Katerina Doka, Dimitrios Tsitsigkos, Manolis Terrovitis, Lampros Giampouras, and Nectarios Koziris. Modissense: A distributed spatio-temporal and textual processing platform for social networking services. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 895–900, New York, NY, USA, 2015. ACM.

[20] Regina Obe and Leo Hsu. *PostGIS in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[21] Regina Obe and Leo Hsu. *PostgreSQL: Up and Running*. O'Reilly Media, Inc., 2012.

[22] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.