TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
Chair of Network Software

Adaptive near real-time data processing of Estonian locations using Foursquare service

Bachelor thesis

Student: Dmitri Batõrjev
Student code: 112169IAPB
Supervisor: assistant Ago Luberg

Tallinn
2014

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

(*kuupäev*) (*allkiri*)

Peaaegu reaalajaline Eesti asukohtade kohanev andmetöötlus Foursquare teenuse abil

Annotatsioon

Antud töö eesmärk on tuvastada, kuidas muutub Eesti sündmuskohtade külastatavus aja jooksul. Saadud informatsiooni tulemusena saab analüüsida inimeste huvi muutumist erinevate kohtade vastu vastavalt hooajale, nädalapäevale, kellaajale jne. Samuti saab tuvastada, millistes kohtades ja mis ajal on toimunud (massi)üritused. Vastav informatsioon tõmmatakse alla Foursquare [1] teenuse abil.

Oma töös olen kokku puutunud järgmiste probleemidega: piiratud sündmuskohtade arv vastuses ja piiratud päringute arv Foursquare teenuse puhul.

Selle töö tulemus on infosüsteem, mis töötleb kogu Eesti andmeid ja mida võib kergesti seadistada teise riigi andmeid tõmbama.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 35 leheküljel, 6 peatükki, 8 joonist, 5 tabelit.

Abstract

The purpose of work done is continuously retrieving information about locations in Estonia using Foursquare service, to determine which places are popular and at what time.

The main problems I dealt with were: limited number of venues in response and limited number of requests can be made to Foursquare service per hour.

The result of work done is an infosystem that processes data about whole Estonia and can be easily configured to collect information about any other country.

The thesis is in English and contains 35 pages of text, 6 chapters, 8 figures and 5 tables.

Glossary of Terms and Abbreviations

PHP	***Hypertext Preprocessor*** [2] Programming language
MVC	***Model-View-Controller*** [3] Design pattern
Yii	***Yes It Is*** [4] PHP framework
JSON	***JavaScript Object Notation*** [5] Lightweight data-interchange format
JDBC	***Java Database Connectivity*** [6] *Java-based data access technology*
PostgreSQL	***Object-relational database management system*** [7]
REST	***Representational state transfer*** [8] *Software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system*
API	***Application Programming Interface*** [9] *API specifies how some software components should interact with each other*
AJAX	***Asynchronous JavaScript and XML*** [10] *With Ajax, Web applications can send data to, and retrieve data from, a server asynchronously (in the background) without interfering with the display and behavior of the existing page.*

Table of figures

Table of tables

Table of code snippets

Table of contents

1. Introduction

Foursquare is a social network, that allows users (about 50 millions at the moment [11]) to share their location with others. Every day Foursquare users „check in“ (announce their location information) at different venues located all over the world (about 1 million check-ins are done daily [11]). But what if we want to find out which venues are popular, when users mainly visit them and how does the popularity of some venue changes? As Foursquare does not offer a service to retrieve information about how the number of visits changes in time, we have to implement a system that collects information about venue visits in near real-time.

On the Internet I have found 2 almost similar projects – The Foursquare Time Machine [12] and 4sqmap [13]. The difference is that the first one shows only your personal checkins history with animation and the second one is just a graphical interface for every Foursquare API endpoint.

1.1 Background and problem

This work is a part of a project in cooperation with a company Positium. Positium has information about mobile phone positions in Estonia. They have anonymous data about geo-coordinates along with the timestamp. They want to know why people gathered at certain areas – was there an event or was it a traffic jam etc. The positioning is not very accurate (the actual location of a mobile phone is within 200 meters of provided location), therefore they need some additional information to decide what is really happening.

Foursquare is a popular social network where people can announce their location. This leads to a hypothesis, that this information could be used to detect special events. For example, based on the history of a pub if there is a significant change in check-ins one evening it can be said that there is something different (more popular) happening (a popular concert). As within 200 meters of a city centre, there are probably many candidates where mobile phones can actually be, combined with social media information we can do more precise deductions. So, if there is a significant change in one of the venues within the mentioned radius it could be concluded that the change is due to an event at that venue.

Data available at Foursquare does not include any information about changes of visits. This work aims to gather information from the service regularly to have near real-time data stream about visits in Estonian venues. Foursquare has certain limits how many queries can be done to retrieve information which prevents doing continuous scraping (basically all the time a new information is downloaded). Based on the changes in visits in near history, more popular places need to be scraped more often than those with mainly static visitors count. The latter lowers the number of needed requests.

Another need for Foursquare data is in the project Sightsmap.com which provides information about locations all over the world. Sightsmap.com combines information from different sources and presents the data to the user in a clean and understandable way. One of the data sources used in this project is Foursquare. So far the whole data for the world was gathered once in a row and that information is used. Instead, the same automatic scraping system could be used for the whole world. More popular places would be updated more often, smaller venues not so often.

1.2 Goals

1. Find all places (venues) in Foursquare located in Estonia
2. Continuously request data and save it in database
3. Optimize number of requests due to limited requests per hour
4. Show different textual and visual reports

1.3 Methodology

All places located in Estonia are found using Foursquare API – the application sends requests, parses the response and saves places' information in the database (the application keeps saving new venues that come every day). To send requests continuously, code must be inside the infinite loop. To avoid memory leaks (so the application would not crash) all objects and resources in the application have to be properly closed (unset) in time. Optimizing the number of requests is needed because Foursquare API has rate limit (5000 requests per hour). Information about popular venues has to be retrieved more often than about unpopular ones, because a popular venue may get several new visitors during one hour whereas another venue

only gets one new visitor every week. Optimization is done using simple algorithm that calculates priority for each area (priority shows how often area's information must be requested). Different visual reports were made using third-party libraries like Google Maps API and Google Charts API.

1.4 Thesis overview

The given thesis has 4 sections: „System overview", „Database structure", „Data collector application" and „Web application".

„System overview" section describes the infosystem's structure, how components are connected between each other and what programming languages and technologies are used.

„Database structure" section contains information about the database tables and views, tables' columns description and table relationships.

„Data collector application" section covers implementation of the main application – how requests are being sent, how response is being parsed and how area division and priority calculation work.

„Web application" section describes web application's pages – what information can be found on them, how data is being calculated and using what libraries different reports are being created.

2. System overview

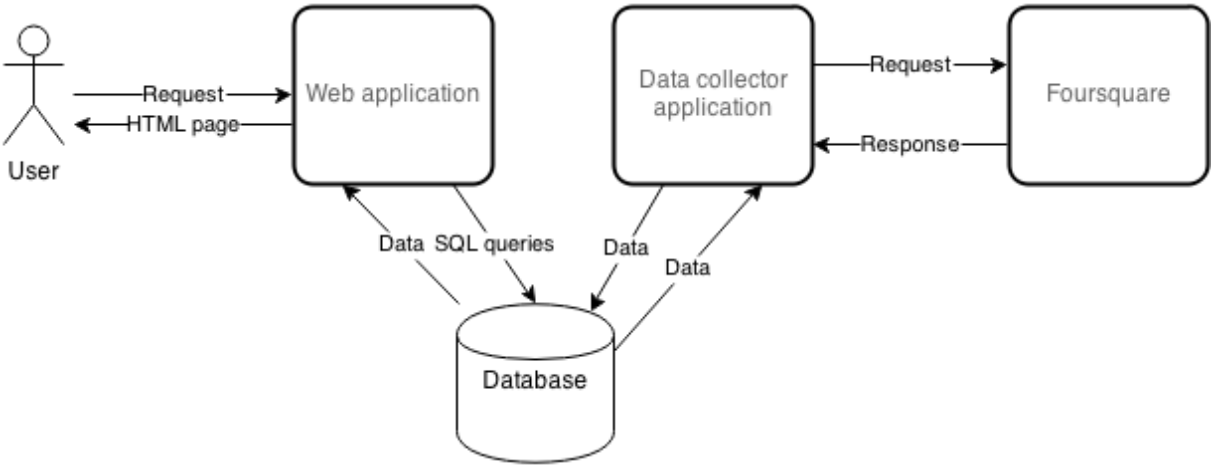

Figure 1 - System structure.

As we can see in the Figure 1 – Data collector application communicates with Foursquare server and saves data in the database. Web application uses the same database as Data collector application does to retrieve data about venues, does calculations and shows result to user.

2.1 Foursquare

To start our work and begin sending requests to Foursquare API [1], we have to create an account on Foursquare, go to the developers page and register our application, then Foursquare will generate *CLIENT ID* and *CLIENT SECRET* keys, which are used for authentication - they have to be sent with other request parameters.

Foursquare has REST [8] architectural style, so all the parameters are sent via HTTP [14]. In our case parameters are being sent using GET [14] method. As a result Foursquare returns data in JSON [5] format.

2.2 Data collector application

The application is implemented in Java programming language. The application sends requests to Foursquare, parses the responses, saves the data in the database and, when it is needed, divides areas and calculates area's priority. To communicate with database, java

application must have JDBC driver, which can be downloaded from official site of the database management system. Also Data collector application has JSON parser as third-party library (GSON [15]). These are the only third-party libraries that are used in the application.

2.3 Database

For this project PostgreSQL is used as a database management system. It is a free, open-source and cross-platform system with a great community.

The database has 5 tables: *area*, *venue*, *venue_stat*, *venue_category* and *venue_venue_category*, and 1 view: *venues*. For more information see section 3. Database structure.

2.4 Web application

Web application is built using PHP language and Yii (version 2) framework, that has MVC arhitecture. I decided to use Yii because it has great community and has a lot of tools, that help to develop the applications faster. I also used Bootstrap [16] front-end framework, which allows to create beautiful GUI [17] very fast, even if you have a little or no experience at all, because it has good tutorials and a lot of examples. As the application has some interactive elements and AJAX data loading I used jQuery [18] library, which has a lot of JavaScript [19] helper functions.

As the web application displays not only textual information, but also visual information like maps and graphs, I decided to use external services – Google Maps API [20] and Google Charts API [21].

3. Database structure

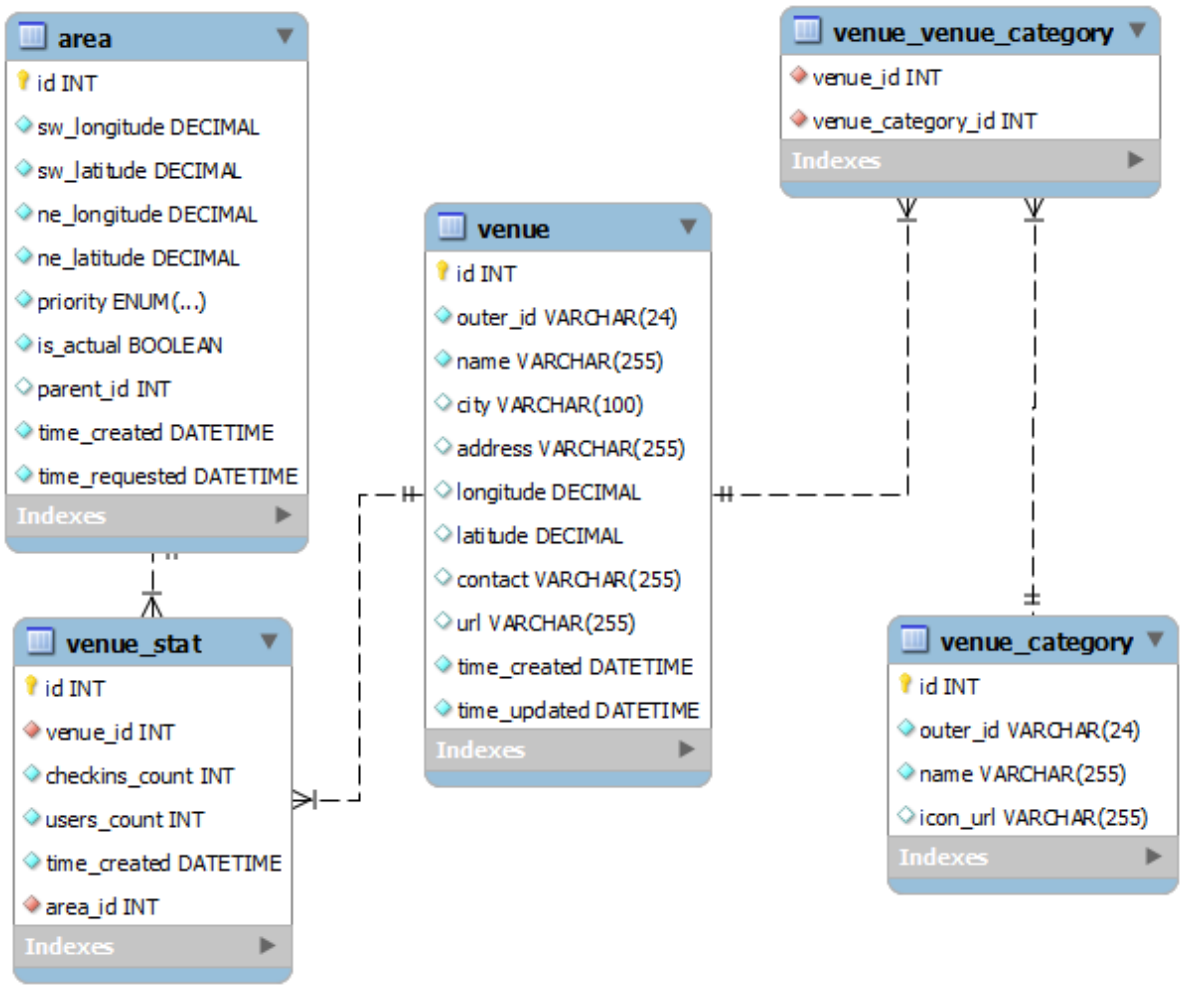

Figure 2 - Database structure and relationships.

3.1 Area table

Area object represents a „rectangle“ on a map. It has south-west and north-east coordinates (corners of the „rectangle“). When an area record is not needed anymore it is not being deleted from table, but *is_actual* flag is being set to false. When an area object is divided, children objects get a reference to the parent (*parent_id* column).

Table 1 - Area table columns description.

Column name	Description
id	Unique identifier

sw_longitude	South-west longitude of area
sw_latitude	South-west latitude of area
ne_longitude	North-east longitude of area
ne_latitude	North-east latitude of area
priority	Priority, that defines how often area's information must be requested. Possible values: Ignore, Everytime, Once an hour, Once a day, Once a week
is_actual	True if area's information can be requested, false otherwise.
parent_id	Id of parent area.
time_created	Time when area was created
time_requested	Last time when area's information was requested

3.2 Venue table

Venue table contains information about all the venues located in Estonia.

Table 2 - Venue table columns description.

Column name	**Description**
id	Unique identifier
outer_id	Unique identifier returned by Foursquare
name	Name of venue
city	Name of city, where venue is located
address	Venue's address
longitude	Venue's longitude
latitude	Venue's latitude
contact	Telephone number
url	Url address
time_created	Time when venue record was created in the database
time_updated	Last time when venue information was updated

3.3 Venue_category table

Venue category represents possible types of the venues.

Table 3 - Venue_category table columns description.

Column name	Description
id	Unique identifier
outer_id	Unique identifier returned by Foursquare
name	Name of category
icon_url	Url address of category icon

3.4 Venue_stat table

This table contains records about how number of checkins and users of some venue have changed over time.

Table 4 - Venue_stat table columns description.

Column name	Description
id	Unique identifier
venue_id	Venue id
checkins_count	How many times users „checked in“ the venue
users_count	How many unique users have „checked in“ the venue
time_created	Time when record was created
area_id	Area id

3.5 Venue_venue_category table

This table is used as connection between *venue* and *venue_category* tables, representing many-to-many relationship [22].

3.6 Venues view

Venues view is created to decrease amount of code in Web application. The following query is used to create the view:

Code snippet 1 – SQL query. Venues view.

```
CREATE OR REPLACE VIEW venues
AS SELECT
v.*, vs.area_id, vs.checkins_count, vs.users_count, vs.time_created AS
last_change
FROM venue AS v
INNER JOIN (
        SELECT DISTINCT ON (venue_id) *
        FROM venue_stat
        ORDER BY venue_id, time_created DESC
) AS vs ON vs.venue_id = v.id;
```

4. Data collector application

4.1 Request

The main purpose of the application is to send requests to Foursquare continuously, so all the code is inside the infinite loop. To avoid problems with memory leaks, every resource or object has to be closed or unset when it is not needed anymore, so Java Garbage Collector could free memory that is used by resource or object [23]. Within the loop the application retrieves area records from the database, generates request for each area and sends them to Foursquare server.

Not every area is requested in every iteration – it depends on which priority the area has (see section 4.4 Priority and request number optimization). Also due to encountered problem with limited number of requests it has been decided to set timeout, so each loop takes 30 minutes. For example, if 2000 areas were requested in 20 minutes, the application will wait another 10 minutes before starting a new cycle.

Requests are sent to the following endpoint of the Foursqure service: https://api.foursquare.com/v2/venues/search

Using the parameters presented in Table 5 and described in more detail in API documentation [24]:

Table 5 - Foursquare venues/search endpoint request parameters.

Parameter name	Description	Example
intent	Intent of the search that tells Foursquare how to perform the search. For our purposes we use „browse“ value, meaning that Foursquare will return venues located in given area.	browse
sw	South-west coordinates (longitude, latitude) divided with comma	59.384900,24.627936
ne	North-east coordinates (longitude, latitude) divided with comma	59.479882,24.919760

limit	How many rows in result (max. 50)	50
client_id	Application's id generated by Foursquare	
client_secret	Application's secret key generated by Foursquare	
v	Version of our product/application	20140320

As an example we will have the following request:

https://api.foursquare.com/v2/venues/search?intent=browse&sw=59.384900,24.627936&ne=59.479882,24.919760&limit=50&client_id=xxx&client_secret=xxx&v=20140320

4.2 Response

If the request is valid and no errors occured, Foursquare returns a list of venues [25] in JSON format. The response contains information about venues, venues' statistics (checkins and users count) and venues' categories.

The application parses the response, validates the country and saves venue's information (venue's general data and venue's categories). If the venue record already exists and the last time its information was updated in the database was more than a week ago, then venue information will be updated. The checkins and users count will be saved to the database only if the checkins count has changed (without this filter we would have about 3 000 000 new records saved to the database every day).

In case of a request error (the request contains wrong information or the rate limit is exceeded) Foursquare returns an error code and an error message, so we can make the application to determine and resolve the problem depending on the error code. At the moment the application handles the following error codes:

- geocode_too_big – the area is too big. In this case the application divides the area into half (see section 4.3 Area dividing) and continues sending requests.
- rate_limit_exceeded – the request limit is exceeded. In this case the application waits 5 minutes before making the next request.

In case the error code is unknown or it is some other type of an error (network or server fault), the application waits 5 seconds before making the next request. As the application logs everything into the log file, we can later view what errors have occured and complete the application, so it could handle new types of errors in the future.

4.3 Area dividing

As Foursquare returns maximum 50 venues in the response or returns an error if the area is too big (the maximum supported area is currently 10,000 square kilometers [24]), the application should be able to divide the area. In case of response with 50 venues the application divides the area until the area's venue count will be less than 50, which means that there are no other venues located in the area in addition to those listed in the response. In case of a big area, the application divides area until the size of the area will be accetable for the service.

At the begining the *area* table had only one record, that represented area information of whole Estonia. This initial record is needed, so that the application knows where to start searching. The record is inserted into database using the following query:

Code snippet 2 - SQL query. Initial area record.

```
INSERT INTO area (id, sw_longitude, sw_latitude, ne_longitude, ne_latitude,
priority, is_actual, parent_id, time_created)
VALUES (1, 21.20775500000000, 57.34200000000000, 29.12889700000000,
59.67112600000000, 1, TRUE, NULL, NOW());
```

Result of insertion we can see in the Figure 3.

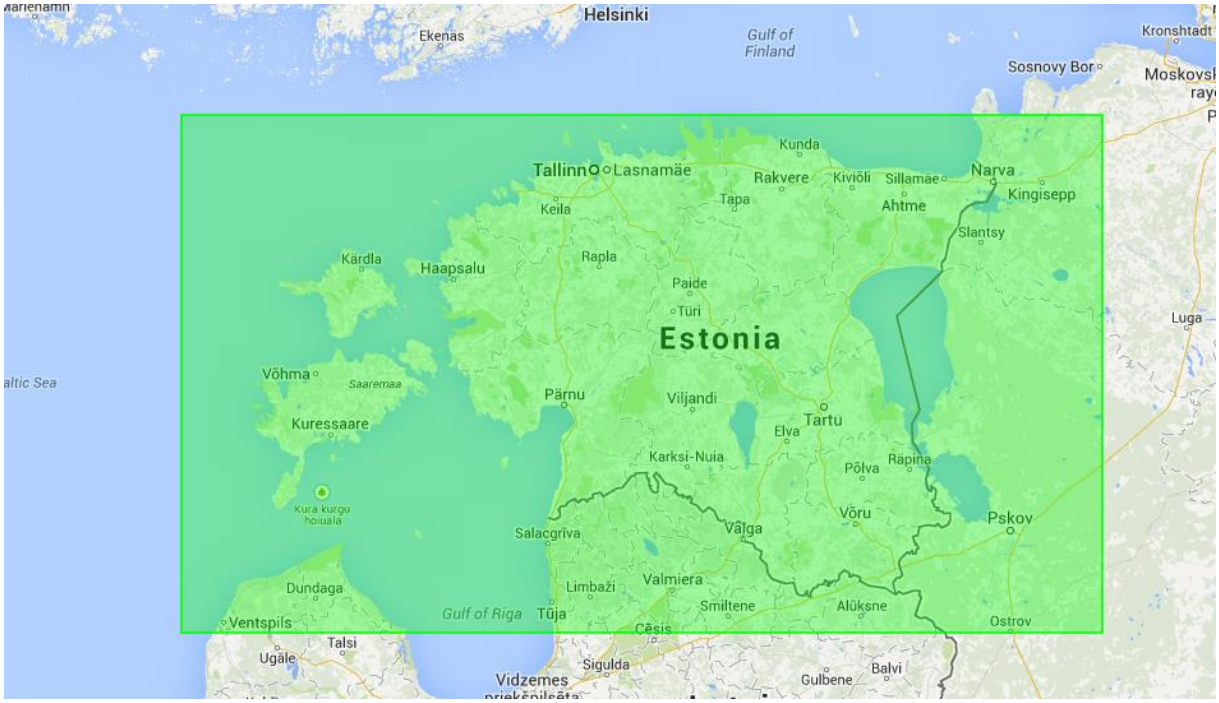

Figure 3 – Map of areas - begining state.

After some time we had the result presented in the Figure 4.

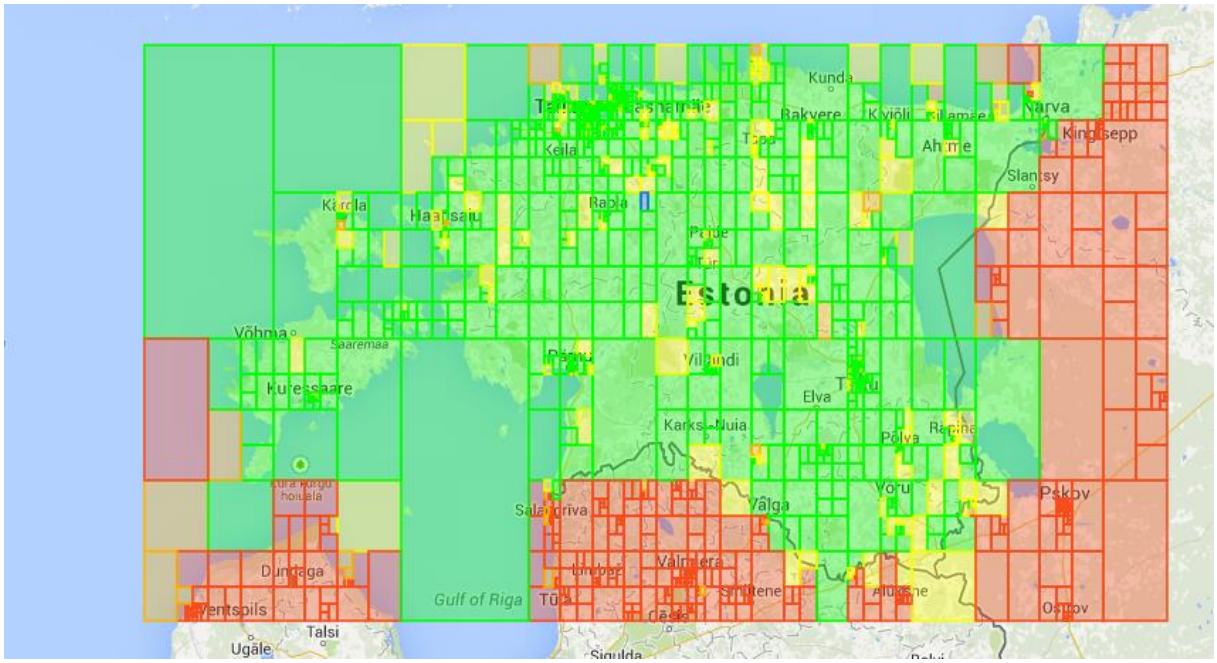

Figure 4 – Map of areas – next state.

Legend information can be found in section 5.5 Area map.

4.4 Priority and request number optimiazation

Due to the limited number of requests allowed (maximum 5 000 per hour [26]), the application has to optimize the number of requests as much as possible, so each area has a priority, which determines how often the area has to be searched. The following priorities are used today: „ignore“ (area is not being requested), „every time“ (every loop), „once an hour“, „once a day“ and „once a week“.

When an area object is created, its priority is „every time“ by default. If there is no venues located in a specific area, its priority becomes „once a week“ – the application will request the area in one week to see if any venues have appeared there. If there are less than 50 venues in the response and all these venues are not Estonian, the area’s priority becomes „ignore“, it could mean that the area is located outside of Estonia. Areas with the „ignore“ priority are not being requested and their priority can be changed only manually. After each Foursquare response, the application counts the total number of checkins (of all venues located in a specific area) and saves this number in memory. After an area has been requested 5 times, the application recalculates its priority and resets the request counter.

If the total number of checkins inside some area has not been changed during last 5 requests, this area becomes a lower priority area. If the number of chechins has been changed every request, then the area becomes higher priority area. Otherwise the priority remains the same as it was.

Example:

Area’s priority is „every time“ and after 5 requests the area has the same number of checkins. It means that the area priority has to be changed due to a rare change of checkins count. As the result, the priority recalculation method will return the „once an hour“ priority.

As the application must consider areas priority, the following query is used to retrieve the areas, which have to be searched, from the database:

Code snippet 3 - SQL query. Retrieving areas that have to be requested considering areas' priorities.

```
SELECT *
FROM area
WHERE is_actual = true
AND priority != 0
AND (
    time_requested <=
        CASE WHEN priority = 2 THEN LOCALTIMESTAMP - INTERVAL '1 HOUR'
            WHEN priority = 3 THEN LOCALTIMESTAMP - INTERVAL '1 DAY'
            WHEN priority = 4 THEN LOCALTIMESTAMP - INTERVAL '1 WEEK'
            ELSE LOCALTIMESTAMP
        END
    OR time_requested IS NULL
)
```

Priorities: 2 – „once an hour“, 3 – „once a day“, 4 – „once a week“

5. Web application

The Web application is used to display textual and visual reports. It allows the user to retrieve needed information from the database, analyze it and check that the Data collector application is working properly.

5.1 Main page

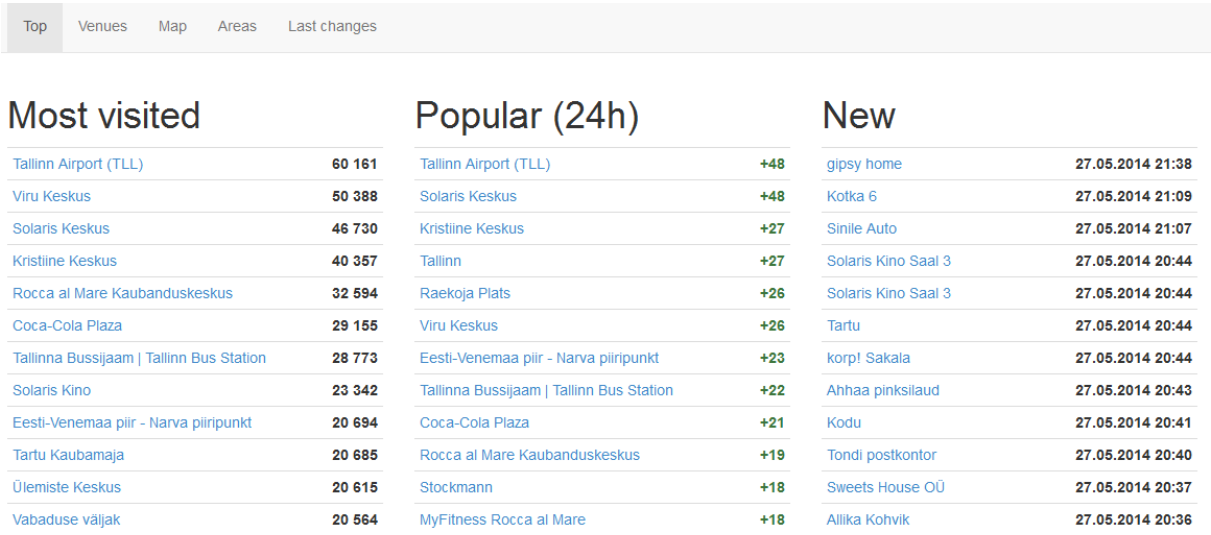

Figure 5 - Web application - main page.

The main page contains 3 tables: most visited venues, popular venues and new venues. These tables are loaded in parallel using AJAX request to reduce loading time. There is also used caching on the server side for a small period of time as table information does not change too often.

The „Most visited“ table shows top 50 venues which have the largest number of checkins. The following SQL query is used to retrieve the most visited venues:

Code snippet 4 - SQL query. Retrieving the most visited venues.

```
SELECT * FROM venues ORDER BY checkins_count DESC LIMIT 50;
```

The „Popular“ table shows 50 most popular venues for the last 24 hours. The following SQL query is used to retrieve needed information:

Code snippet 5 - SQL query. Retrieving the most popular venues.

```
SELECT
v.id, v.name, v.checkins_count - vs1.checkins_count AS checkins_count_diff
FROM venues AS v
INNER JOIN (
    SELECT DISTINCT ON (venue_id) *
    FROM venue_stat
    WHERE time_created < LOCALTIMESTAMP - INTERVAL '24 HOUR'
    ORDER BY venue_id, time_created DESC
) AS vs1 ON vs1.venue_id = v.id
ORDER BY checkins_count_diff DESC
LIMIT 50;
```

The „New“ table shows 50 recently saved venues in the database:

Code snippet 6 – SQL query. Retrieving 50 recently venues.

```
SELECT * FROM venue ORDER BY time_created DESC LIMIT 50;
```

5.2 Venue list

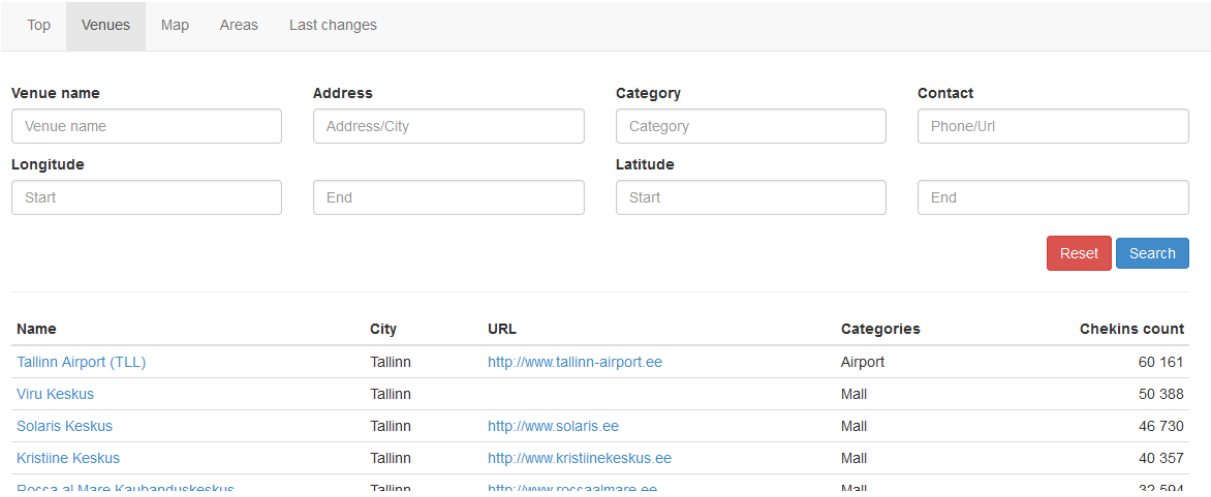

Figure 6 - Web application – venue list.

This page allows the user to search venues. User can search venue by its name, address, category, contact or coordinates. The category input field uses autocomplete (as the user types, the application makes suggestions and the user can select from a list of suggestions).

5.3 Venue view

The venue view contains different information related to the specific venue. The page consists of 4 sections: general information, map, statistics and history.

In the general information section the user can find information like venue name, address, phone number or URL address.

The next section is a map with a marker, that shows the location of the venue.

The statistics section contains calculated data like average daily checkins count and 2 charts: the first one displays the venue popularity depending on the day of week (Figure 7) and the second one displays the popularity depending on the hour of day (Figure 8).

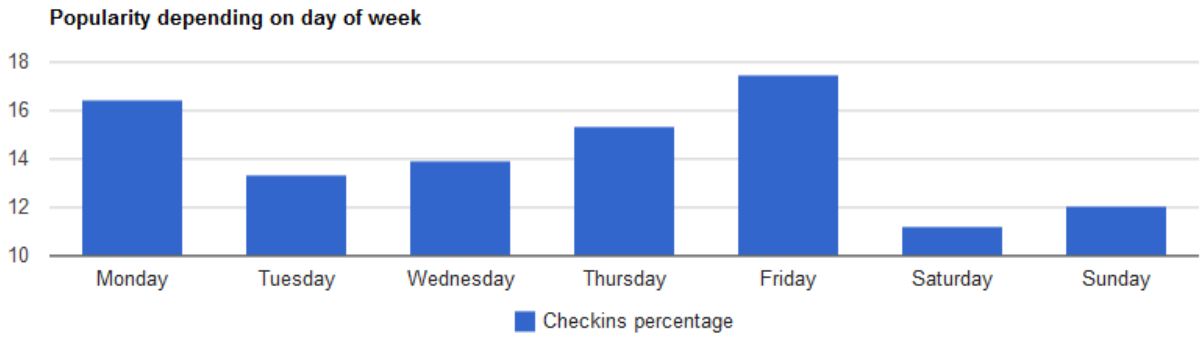

Figure 7 - Chart. Venue popularity depending on day of week.

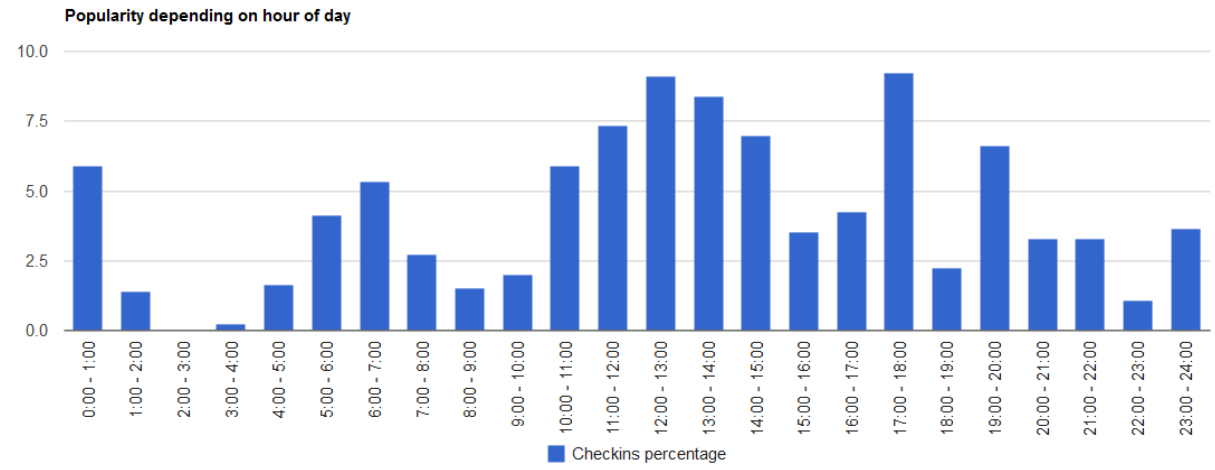

Figure 8 - Chart. Venue popularity depending on hour of day.

And finally, the history section displays all the venue checkins count changes that are stored in the database.

5.4 Venue map

This page shows an interactive map (Google Map) of venues. It shows up to 1500 venues at time. The user can zoom in and zoom out and drag the map in different directions – the application will asynchronously load (using AJAX) and display the venues depending on the current position of the map.

5.5 Area map

This page shows an interactive map that represents how the area of Estonia is divided and what priority every area has. The map has the following legend:

- Green rectangle – an area with the „every time“ priority
- Blue rectangle – an area with the „once an hour“ priority
- Yellow rectangle – an area with the „once a day“ priority
- Orange rectangle – an area with the „once a week“ priority
- Red rectangle – an area with the „ignore“ priority

5.6 Last changes

This page shows a list of the last 200 venues that had checkins changes sorted by the change time in descending order. This page can be used to check if the Data collector application is working properly.

6. Kokkuvõte

Käesoleva töö põhieesmärk on tuvastada, kuidas muutub Eesti sündmuskohtade külastatavus aja jooksul. Saadud informatsiooni tulemusena saab analüüsida inimeste huvi muutumist erinevate kohtade vastu vastavalt hooajale, nädalapäevale, kellaajale jne. Samuti saab tuvastada, millistes kohtades ja mis ajal on toimunud (massi)üritused. Vastav informatsioon tõmmatakse alla Foursquare [1] teenuse abil.

Selle töö põhitulemus on infosüsteem, mis töötleb kogu Eesti andmeid ja mida võib kergesti seadistada teise riigi andmeid tõmbama.

Summary

The main purpose of this work is to continuously retrieve information about the locations in Estonia using Foursquare service, to determine which places are popular and at what time.

The result of the work done is an infosystem that processes data about the whole Estonia and can be easily configured to collect information about any other country. Another result is a web system where all the gathered information can be viewed and analyzed.

Future plans

The implemented application is a starting point for the analysis needed. There are several additional features which will help to have more in-depth conclusions. The working application is a clear mark that this kind of methodology can be successfully applied to gather information about visitors in the whole country. Some of the plans for the future:

- To use the built system to collect information about other countries.
- Merge location history data with positioning data provided by Positium. Some automation for that can be done – the user can upload positioning data and the system automatically aligns the locations and positionings.
- To connect system with weather, event and tourism services to find a reason for some increase or decrease of number of visitors .
- To make the reports better. Today the application generates a few types of reports and they may not contain all the information needed.
- To upgrate request optimization logic. As number of requested countries will increase and these contries may be bigger than Estonia or have much more venues than Estonia does, 5000 requests per hour may not be enough to parse whole country in the same manner presented in this work.

References

[1] „Foursquare for developers“ [WWW]. Available: https://developer.foursquare.com/.

[2] „PHP“ [WWW]. Available: http://www.php.net/.

[3] „MVC“ [WWW]. Available: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller.

[4] „Yii framework“ [WWW]. Available: http://www.yiiframework.com/.

[5] „JSON“ [WWW]. Available: http://www.json.org/.

[6] „JDBC“ [WWW]. Available: http://www.oracle.com/technetwork/java/javase/jdbc/index.html.

[7] „PostgreSQL“ [WWW]. Available: http://www.postgresql.org/.

[8] „REST“ [WWW]. Available: http://en.wikipedia.org/wiki/Representational_state_transfer.

[9] „API“ [WWW]. Available: http://en.wikipedia.org/wiki/API.

[10] „Ajax“ [WWW]. Available: http://en.wikipedia.org/wiki/Ajax_%28programming%29.

[11] „Foursquare. About“ [WWW]. Available: https://foursquare.com/about.

[12] „The Foursquare Time Machine“ [WWW]. Available: https://foursquare.com/timemachine.

[13] „Foursquare maps and statistics“ [WWW]. Available: http://www.4sqmap.com/.

[14] „HTTP“ [WWW]. Available: http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

[15] „GSON“ [WWW]. Available: https://code.google.com/p/google-gson/.

[16] „Twitter Bootstrap“ [WWW]. Available: http://getbootstrap.com/.

[17] „GUI“ [WWW]. Available: http://en.wikipedia.org/wiki/Graphical_user_interface.

[18] „jQuery“ [WWW]. Available: http://jquery.com/.

[19] „JavaScript“ [WWW]. Available: http://en.wikipedia.org/wiki/JavaScript.

[20] „Google Maps API“ [WWW]. Available: https://developers.google.com/maps/.

[21] „Google Charts API“ [WWW]. Available: https://developers.google.com/chart/.

[22] „Many-to-many relationship“ [WWW]. Available: http://en.wikipedia.org/wiki/Many-to-many_%28data_model%29.

[23] „Garbage collection“ [WWW]. Available: http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29.

[24] „Foursqure API - venues/search endpoint“ [WWW]. Available: https://developer.foursquare.com/docs/venues/search.

[25] „Foursquare API - venue response“ [WWW]. Available: https://developer.foursquare.com/docs/responses/venue.

[26] „Foursquare API - rate limits“ [WWW]. Available: https://developer.foursquare.com/overview/ratelimits.

Appendix 1

Web application address

The Web application can be found on

http://dijkstra.cs.ttu.ee/~t112169/foursquare/web/index.php?r=venue/index

Area division

Code snippet 7 - Java code. Area division method.

```
/**
 * Divide area in half
 * @param area
 * @return two areas
 */
public static Area[] divideArea(Area area) {
    Area[] areas = new Area[2];
    double newLon = area.getNeLongitude() + (area.getSwLongitude() -
area.getNeLongitude()) / 2;
    double newLat = area.getSwLatitude() + (area.getNeLatitude() -
area.getSwLatitude()) / 2;

    areas[0] = new Area();
    areas[0].setActual(true);
    areas[0].setNeLatitude(area.getNeLatitude());
    areas[0].setNeLongitude(area.getNeLongitude());
    areas[0].setParentId(area.getId());
    areas[0].setPriority(area.getPriority());
    areas[0].setSwLatitude(area.getSwLatitude());

    areas[1] = new Area();
    areas[1].setActual(true);
    areas[1].setNeLatitude(area.getNeLatitude());
    areas[1].setParentId(area.getId());
    areas[1].setPriority(area.getPriority());
    areas[1].setSwLatitude(area.getSwLatitude());
    areas[1].setSwLongitude(area.getSwLongitude());

    if (Math.abs(area.getSwLongitude() - area.getNeLongitude()) >
Math.abs(area.getNeLatitude() - area.getSwLatitude())) {
        // dividing vertically
        areas[0].setSwLongitude(newLon);
        areas[1].setNeLongitude(newLon);
        areas[0].setSwLatitude(area.getSwLatitude());
        areas[1].setNeLatitude(area.getNeLatitude());
    } else {
        // dividing horizontally
        areas[0].setSwLatitude(newLat);
        areas[1].setNeLatitude(newLat);
        areas[0].setSwLongitude(area.getSwLongitude());
        areas[1].setNeLongitude(area.getNeLongitude());
    }

    return areas;
}
```

Code snippet 8 - Java code. Priority recalculation method.

```java
/**
 * Recalculates priority depending on area last stats (checkins count)
 * @param currentPriority area's current priority
 * @param checkinsCountArr last stats
 * @return recalculated priority
 */
public static Priority recalculatePriority(Priority currentPriority,
Integer[] checkinsCountArr) {
    boolean allAreDifferent = true; // all checkins counts are different
    boolean allAreIdentical = true; // all checkins counts are same

    for (int i = 1; i < checkinsCountArr.length; i++) {
        if (checkinsCountArr[i] == checkinsCountArr[i-1]) {
            allAreDifferent = false;
        } else {
            allAreIdentical = false;
        }
    }

    if (allAreDifferent) {
        switch(currentPriority) {
            case ONCE_AN_HOUR: return Priority.EVERYTIME;
            case ONCE_A_DAY: return Priority.ONCE_AN_HOUR;
            case ONCE_A_WEEK: return Priority.ONCE_A_DAY;
            default: break;
        }
    } else if (allAreIdentical) {
        switch(currentPriority) {
            case EVERYTIME: return Priority.ONCE_AN_HOUR;
            case ONCE_AN_HOUR: return Priority.ONCE_A_DAY;
            case ONCE_A_DAY: return Priority.ONCE_A_WEEK;
            default: break;
        }
    }

    return currentPriority;
}
```