Spatial data querying and mining Demo

# FRAMEWORK

## 

## The Server Side Components

**Core Components**

* **Crawler & Data Integrator**:

1) Keep downloading tweets, foursquare check-ins and Instagram posts;

2) Download all tweets/check-ins for one user.

3) Extract POIs from Foursquare and Instagram;

4) Load check-ins from historical data to database;

5) Provide interface to check existence of users and to add new users;

* **Indexing Structures**:

1) Build IR-tree from static data;

2) Periodically update the index;

3) Inverted file / Database to store user login information

* **User Behavior Model & Model Combiner**:

1) Re-train the two models periodically;

2) Learn the weight between two models according to the text quality.

**Services**

* **Publish/Subscribe**:

Subscribe tweets for a specified POI.

* **POI & Region Querying**:

1) Selecting keywords from a set of words (tweets, POI description)

2) Retrieving POIs which contain a set of keywords

3) Retrieving diversified regions according to a set of keywords

* **Automatic User Profiling**:

Show user interests on topics, categories, aspects and personalized regions

* **Personalized Recommendation**:

1) POI recommendation (given any of time, category, location, aspect)

2) Show top aspects of the POI that may be satisfied by the user

* **Business Aspect Analysis**:

Show top aspects that is satisfied or dissatisfied in a user specified region and category.

## The Browser Side Components

The layout of the browser side is shown in the following figure.

****

**User Interface:**

* **Login Module** (login/sign up with twitter): interact with **Database and Crawlers** to implement login/sign up.
* **User Profile Panel**:

1) list user interests on 3 aspects in each of the 3 categories (interact with **Automatic User Profiling**);

2) click a button to show the user’s 3 personal regions on the map (the regions are ovals parameterized by Gaussian distributions) (interact with **Automatic User Profiling**);

3) shows the user’s subscription on POIs (interact with **Publish/Subscribe**);

* **Map UI**:

1) **Recommendation**: Select aspect, category, interact with (**Personalized Recommendation**) and show the recommended top 10 POIs on the map, and a list the top 10 POI names together with the top 3 aspects that satisfied the user’s interests.

2) **Aspect Analysis**: Draw a rectangle region and select a category, return a list of top 3 aspects that is positive/negative in the region of the specified category (interact with **Business Aspect Analysis**).

3) **POI Querying**:On the map, and input several keywords in the search box, show the most related POIs in the map.

4) **Diversified Region Querying**: Select a category, and input a region size (e.g., in square meters),

# System Architecture

## 4-Layer System

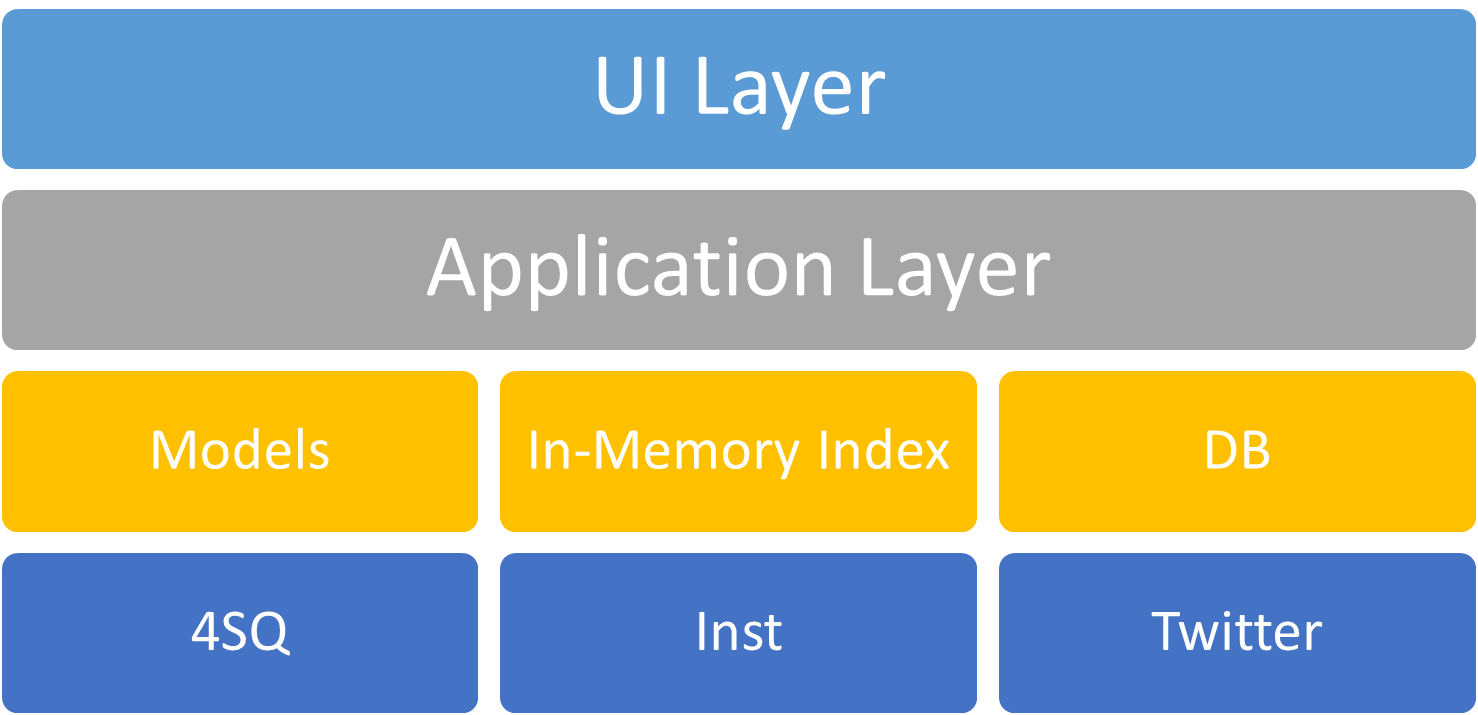
Our system processes the data (Twitter, 4SQ and Instagram) in a 4-layer architecture. The four layers have the following functionalities, respectively.

**Logic Layer:** basic logical functionalities includes user login, query validation, user interface

**Application Layer:** spatial querying and recommendation algorithms

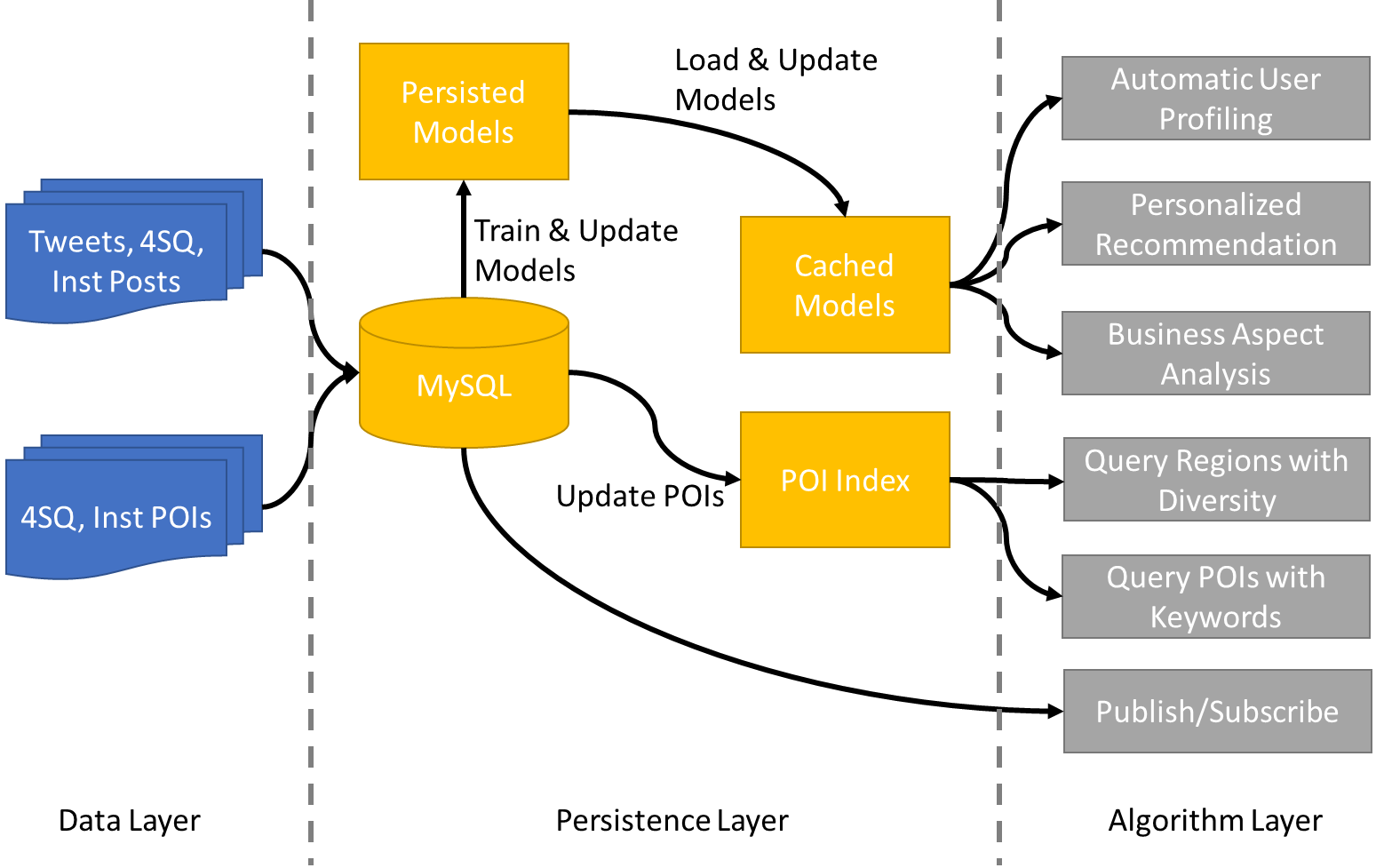
**Persistence Layer:** database and in-memory index maintenance

**Data Layer:** continuous crawling and parsing data



## Data Flow

The following figure shows the data flow of our system.



**From Data Layer to Persistence Layer:**

We crawl tweets, 4SQ check-ins, Instagram posts, and 4SQ and Instagram POIs, parse the results and store them in MySQL database (may develop in a Linux Server).

**Inside Persistence Layer:**

* From MySQL database, we periodically retrieve 4SQ check-ins and Instagram posts to re-train and update our user behavior models (W4 and SAR). The retrained model is stored in binary files.
* The POI Index read POI data from MySQL, and update the index when we find new POIs.
* We cached the user behavior models in memory to make fast recommendation and analysis. This treatment makes the model training and online analysis disjoint in real time. The benefit is that we can pre-train and update the model without interrupting online recommendation. In addition, when the system crashes, we can restore the cached model quickly by loading the models in binary files persisted in the hard disk.

**From Persistence Layer to Algorithm Layer:**

* From MySQL database, the publish / subscribe function retrieves related historical tweets once a user subscribes a POI.
* From POI Index, spatial key word querying functionalities access the POI index to answer spatial queries.

From cached models, user profiling, recommendation and business analysis can accomplish by retrieving related data.

# Implementation Details

## Database Schema

**The field with underscore is primary key.**

**tweet** (***id*** BigInt not null, ***user\_id*** BigInt not null, ***text*** Text not null, ***time*** DateTime not null, ***lat*** Float, ***lon*** Float)

Each tweet contains a 64-bit integer ID, user ID, text, timestamp and coordinates (the coordinates can be null)

**Index:** clustered index on “time”, non-clustered index on “user\_id”

**4sq\_checkin** (***id*** VarChar(24) not null, ***user\_id*** VarChar(24) not null, ***text*** Text not null, ***time*** DateTime not null, ***poi\_id*** BigInt)

Each foursquare check-in contains a 24-character length ID, user ID, text, timestamp and POI (foreign key that is associated to the “id” in table “POI”)

**Index:** clustered index on “time”, non-clustered index on “user\_id”, “twitter\_id”

**4sq\_twitter\_user\_map** (***4sq\_id*** VarChar(24) not null, ***twitter\_id*** BigInt)

Each user map from 4SQ to Twitter contains a 24-character length 4SQ ID and a 64-bit integer Twitter ID.

**Index:** non-clustered index on “twitter\_id”

**inst\_post** (***id*** BigInt not null, ***user\_id*** BigInt not null, ***text*** Text not null, ***time*** Datatime not null, ***poi\_id*** BigInt)

Each Instagram post contains a 64-bit integer ID, user ID, text, timestamp and POI (foreign key that is associated to the “id” in table “POI”, ***twitter\_id*** BigInt not null)

**Index:** clustered index on “time”, non-clustered index on “user\_id”, “twitter\_id”

**inst\_twitter\_user\_map** (***inst\_id*** BigInt not null, ***twitter\_id*** BigInt)

Each user map from 4SQ to Twitter contains a 24-character length 4SQ ID and a 64-bit integer Twitter ID.

**Index:** non-clustered index on “twitter\_id”

**poi** (***id*** BigInt not null auto increment, ***name*** VarChar(150), ***category*** Text, ***lat*** Float, ***lon*** Float, ***4sq\_id*** VarChar(24), ***inst\_id*** BigInt)

Each POI contains a 64-bit integer ID, name, category IDs (**separated by “|”**), coordinates, and the ID from the 4SQ and Instagram. If a POI can be found in both sources (4SQ and Instagram), we have ids from both sources.

**4sq\_tips** (***id*** BigInt not null auto increment, ***user\_id*** VarChar(24) not null, ***text*** Text not null, ***time*** DateTime not null, ***4sq\_id*** VarChar(24))

Each tip contains a system auto-generated id, the 4sq user ID who post the tip, the content of the tip, the creation time of the tip and the 4sq ID of the tip.

**category** (***id*** Int not null auto increment, ***name*** VarChar(150), ***parent\_id*** Int)

Each cateory contains an integer ID, name, and the ID of its parent category.

**user\_subscription** (***user\_id*** BigInt not null, ***poi\_id*** BigInt)

Each subscription contains a 64-bit integer user ID and POI ID.

Note that, the POI id is assigned automatically. Each time we insert a 4SQ check-in or Instagram post, we first check if table POI contains the POI in the check-in or post. If yes, retrieve the id of the POI to fill in the poi\_id field. If no, create a new POI tuple in table **poi**, then use the new POI id to fill in the poi\_id field. To speed up the query of existing POIs, we build a non-clustered index for the field “4sq\_id” and “inst\_id” in table **poi**.

**We will host the database in machine 155.69.149.161. The name of the database is SQM (spatial query and mining).** The account used for accessing the database will be sent via email to the contributor who will use the database after the database is created.

## Modules & Interfaces

This section describes the modules and the interfaces **provided** by each module.

**Data Layer**

Module List:

|  |  |
| --- | --- |
| Module | Functionality |
| Twitter crawler | 1. Use streaming API to keep crawling tweets and store them into database. 2. If the tweet is a foursquare or Instagram check-in, crawl the foursquare / Instagram check-in by using foursquare / Instagram ID wrapped in the tweet. 3. For users who have signed up our system, keep crawling each user’s posts using REST API. |
| 4SQ check-in crawler | Keep crawling check-ins to database, and   1. Check if the database (table **poi**) contains the POI mentioned in the check-in, and insert the POI to database if no. To check the existence of a 4SQ POI, we follow two steps:    1. Check the 4SQ id in database, go to b) if not exists;    2. Use Instagram api to map the 4SQ id to an Instagram id, then check if the database contains that Instagram POI. 2. Check if the database (table **4sq\_twitter\_user\_map**) contains the map from the 4SQ user to Twitter account (includes the map from 4SQ user to null twitter user). If the table does not contain information about that 4SQ user, query the website **about.me** to check if the 4SQ user has associated Twitter account. If we can find an associated Twitter account, insert the mapping into the table; otherwise, insert a new mapping from the 4SQ user to null twitter user into the table.   Also, keep crawling check-ins for the users in our system if they have associated with their 4SQ accounts. |
| Instagram post crawler | Keep crawling posts to database, and   1. Check if the data contains the POI mentioned in the check-in, and insert the POI to database if no. 2. Check if the database (table **inst\_twitter\_user\_map**) contains the map from the Instagram user to Twitter account (includes the map from Instagram user to null twitter user). If the table does not contain information about that Instagram user, query the website **about.me** to check if the Instagram user has associated Twitter account. If we can find an associated Twitter account, insert the mapping into the table; otherwise, insert a new mapping from the Instagram user to null twitter user into the table.   Also, keep crawling check-ins for the users in our system if they have associated with their Instagram accounts. |
| Data Importer | Import tweets, 4SQ check-ins, Instagram posts, POIs, 4SQ POI categories and tips into database. We already have some tweets and a program to crawl all the POIs in a region, and a program to extract 4SQ check-ins and Instagram posts from the tweets. We need to incorporate these programs to import data to database. |

Interface List:

The twitter crawler module provides an interface to let the UI level directly notify the crawler to include new users.

We host the crawler in tomcat ([http://155.69.149.161:8080/twitter/](http://155.69.149.245:8080/twitter/)) as a servlet and if the UI request to add a new user, we add the user to our tracking list. Similar interfaces are adapted to Foursquare and Instagram.

|  |  |
| --- | --- |
| Module | Interface |
| Twitter crawler | **Query format:** [http://155.69.149.161:8080/twitter/newuser?user=USERID](http://155.69.149.245:8080/twitter/newuser?user=USERID)  **Parameters:**  user – the Twitter ID of the new user  **Return:** null |
| 4SQ check-in crawler | **Query format:** [http://155.69.149.161:8080/4sq/newuser?user=USERID](http://155.69.149.245:8080/4sq/newuser?user=USERID)  **Parameters:**  user – the Foursquare ID of the new user  **Return:** null |
| Instagram post crawler | **Query format:** [http://155.69.149.161:8080/inst/newuser?user=USERID](http://155.69.149.245:8080/inst/newuser?user=USERID)  **Parameters:**  user – the Instagram ID of the new user  **Return:** null |

**Persistence Layer**

Module List:

|  |  |
| --- | --- |
| Module | Functionality |
| POI index | Retrieve POIs from the database and build the in-memory index for POIs when first start. Check the database for new POIs periodically (e.g., once a day). This module hosts a local service for querying POIs. |
| W4 trainer | Train W4 on 4SQ check-ins and Instagram posts and persist to files (e.g., once a month) |
| SAR trainer | Train SAR on Instagram posts and persist to files (e.g., once a month) |
| Model loader | Load the persisted W4 and SAR models from files to memories. Reload the model every time the trainers finish training the new models |

We develop the **POI index** module with the following two phases.

**Phase I (For fast building the system):**

Each application builds and keeps their own index.

**Phase II (For optimizing the memory usage):**

Build a central index management service to support querying on shared index data among different applications (e.g., querying regions and POIs).

Interface List:

To make the process communication efficient, we implement the following modules with message queues.

An alternative way is to communicate with sockets which may have higher communication cost. The following interfaces of message queues are based on Posix IPC in Linux. Please refer to <http://www.tldp.org/LDP/lpg/node21.html> for more details. The message queue in Posix IPC uses a key to specify which queue we are using and use “mtype” to specify the type of the message.

Note that all the messages sent should have a header to specify whether the current block is the last one of the content, and the number of the bytes as described below:

**Header** (*5 bytes*): is the last block? (the first byte), number of bytes in the block excludes the header (4 bytes)

|  |  |
| --- | --- |
| Module | Interface |
| POI index | **Request message**: (key: 101, mtype: 1)  Header (5 bytes)  Body (max 4056 bytes a block): the content of the request, e.g., the categories of a POI.  **Response message**: (key: 101, mtype: 2)  Header (5 bytes)  Body (max 4056 bytes a block): the response of the request.  **Request format:**  [type]\t[arg1]=value1\t[arg2]=value2\t…  Types:  r – query POIs within a specified region,  **parameters**:  sw -- southwest point (e.g., sw=1.4,-79)  ne -- northeast point (e.g., ne=1.5, -70)  **return**:  poi list  pc – query the categories of a specified POI, arguments:  **parameters**:  pid – poi id (e.g., id=123)  **return**:  category list  cd – query the descendent categories of a given category  **parameters**:  cid – category id (e.g., id=123)  r – recursively retrieving all the descendants. r=0 returns the direct children, while r=1 returns all the descendants.  **return**:  category list  **Response format:**  **POI list**: [poi1]\n[poi2]\n…  **POI**: [id]\t[name]\t[category list]\t[latitude]\t[longitude]\t[4sq\_id]\t[inst\_id]  **Category list**: [category\_id1, category\_name1]|[category\_id2, category\_name2]|… |

**Application Layer**

Module List:

|  |  |
| --- | --- |
| Module | Functionality |
| Query POIs | Retrieve POIs with specified keywords. |
| Query Diversified Regions | Retrieve most diversified region (in categories, e.g., Chinese restaurant, French, etc.) with a given high category (e.g., restaurant) |
| Publish/Subscribe | Push new tweets to the users for the subscribed POIs |
| Automatic User Profiling | Load the aspects & topics that a user may prefer in word distributions |
| Personalized Recommendation | Recommend top (10) POIs to users with user specified requirements (keyword, aspect or category) |
| Business Aspect Analysis | Rank top three positive and negative aspects that are in the user specified region and category. |

All of the above modules are implemented as http services that receive http requests from the UI layer and send response to the UI layer. Note that the programming language of the modules are different, so we deploy them into different services. We assign each module a port number for communication:

|  |  |
| --- | --- |
| Module | TCP Port number |
| Query POIs | 8001 |
| Query Diversified Regions | 8002 |
| Publish/Subscribe | 8080 |
| Automatic User Profiling | 8080 |
| Personalized Recommendation | 8080 |
| Business Aspect Analysis | 8080 |

The port number of 8080 is assigned for tomcat, i.e., and the four applications developed in Java are deployed in tomcat and use the following virtual folders and prefix:

|  |  |
| --- | --- |
| Module | Virtual folders |
| Publish/Subscribe | /subscribe/ |
| Automatic User Profiling | /behavior/ |
| Personalized Recommendation | /behavior/ |
| Business Aspect Analysis | /behavior/ |

Interface List:

|  |  |
| --- | --- |
| Module | Interface |
| Query POIs | **Query format:** [http://155.69.149.161:8001/poi?keyword=WORDLIST&ll=LATITUDE,LONGITUDE](http://155.69.149.245:8001/poi?keyword=WORDLIST&ll=LATITUDE,LONGITUDE)  **Parameters:**  keyword – a list of keywords, separated by “,”.  ll – a spatial point. We return the POIs that satisfy the keyword and are near the coordinates.  **Return format (json):**  {  “status”: OK/Error,  “pois”:  {  “count”: the number of returned POIs,  “list”  [  {  “name”: name of the POI,  “4sq\_id”: id of 4SQ,  “inst\_id” id of Instagram,  “categories”:  [  {  “cid”: category id  “cname”: category name  },  ],  “location”:  {  “lat”: latitude,  “lon”: longitude  }  },  …  ],  }  } |
| Query Diversified Regions | **Query format:** [http://155.69.149.161:8002/region?keyword=WORDLIST&size=REGIONSIZE](http://155.69.149.245:8002/region?keyword=WORDLIST&size=REGIONSIZE)  **Parameters:**  keyword – a list of keywords, separated by “,”.  size – the size of a region  **Return format (json):**  {  “status”: OK/Error,  “region”:  {  “southwest”:  {  “lat”: latitude,  “lon”: longitude  },  “northeast”:  {  “lat”: latitude,  “lon”: longitude  }  }  } |
| Publish/Subscribe | **Query format:** [http://155.69.149.161:8080/subscribe/posts?poi=POIID](http://155.69.149.245:8080/subscribe/posts?poi=POIID)  **Parameters:**  poi – the id of a POI  **Return format (json):**  {  “status”: OK/Error,  “posts”:  {  “count”: the number of return posts,  “list”:  [  {  “id”: the ID of the post,  “text”: the content of the post with quotes, (e.g., “hello world!”)  “time”: the time of the post with quotes, (e.g., “2015-10-20 14:20:30”)  “user\_id”: the id of the user who posted the post.  “location”:  {  “lat”: latitude,  “lon”: longitude  },  }, …  ]  }  } |
| Automatic User Profiling | **Query format:** [http://155.69.149.161:8080/behavior/profile?user=USERID](http://155.69.149.245:8080/behavior/profile?user=USERID)  **Parameters:**  user – the Twitter id of a user  **Return format (json):**  {  “status”: OK/Error,  “categories”:  {  “count”: the number of categories,  “list”:  [  {  “category\_id”: the id of the category,  “category\_name”: the name of the category,  “category\_score”: the rank score of the category,  “aspects”:  [  o “count”: the number of aspects,  “list”:  [{  “aspect\_id”: the id of the aspect,  “aspect\_score”: the preference score on the aspect,  “aspect\_name”: the name of the aspect,  “word\_count”: the number of words returned for an aspect,  “word\_list”:  [  {“word”: word, “pro”: probability of the word},  …  ]  }, …  ]  ]  }…  ]  },  “regions”:  {  “count”: the number of regions,  “list”:  [  {  “region\_id”: the ID of the region,  “region\_score”: the preference score on the region,  “center”:  {  “lat”: latitude of the center,  “lon”: longitude of the center  },  “variance”:  {  “var00”: the first element in the 2x2 co-variance matrix,  “var01”: the second element in the 2x2 co-variance matrix,  “var10”: the third element in the 2x2 co-variance matrix,  “var11”: the third element in the 2x2 co-variance matrix,  }  },…  ]  }  } |
| Personalized Recommendation | **Query format:** [http://155.69.149.161:8080/behavior/recommend?user=USERID&aspect=ASPECTID&category=CATEGORYID&keyword=KEYWORDLIST](http://155.69.149.245:8080/behavior/recommend?user=USERID&aspect=ASPECTID&category=CATEGORYID&keyword=KEYWORDLIST)  **Parameters:**  user – the Twitter id of a user  aspect – aspect IDs to specify the user interests in aspects, separated by “,”  category – category IDs to specify the user interests in categories, separated by “,”  keyword – keywords use to specify the topics, separated by “,”  **Return format (json):**  {  “status”: OK/Error,  “pois”:  {  “count”: the number of returned POIs,  “list”  [  {  “name”: name of the POI,  “4sq\_id”: id of 4SQ,  “inst\_id” id of Instagram,  “categories”:  [  {  “cid”: category id  “cname”: category name  },  ],  “location”:  {  “lat”: latitude,  “lon”: longitude  }  “aspects”:  {  “count”: the number of return aspects, (we return top 3 aspects, usually it is 3)  “list”:  [  {  “aspect\_id”: the ID of the aspect,  “aspect\_score”: the aspect score of the POI,  “aspect\_name”: the name of the aspect,  “word\_count”: the number of words returned for an aspect,  “word\_list”:  [  {“word”: word, “pro”: probability of the word},  …  ]  }, …  ]  }  },  …  ],  }  } |
| Business Aspect Analysis | **Query format:** [http://155.69.149.161:8080/behavior/baspect?sw=SOUTHWESTPOINT&ne=NORTHEASTPOINT&category=CATEGORYID](http://155.69.149.245:8080/behavior/baspect?sw=SOUTHWESTPOINT&ne=NORTHEASTPOINT&category=CATEGORYID)  **Parameters:**  sw – the south west point of the specified region, latitude and longitude are separated by “,”  ne – the north east point of the specified region  category – the category ID of the specified category  **Return format (json):**  {  “status”: OK/Error,  “aspects\_pos”:  {  “count”: the number of return aspects,  “list”:  [  {  “aspect\_id”: the id of the aspect,  “aspect\_score”: the score on the positive aspect,  “aspect\_name”: the name of the aspect,  “word\_count”: the number of words returned for an aspect,  “word\_list”:  [  {“word”: word, “pro”: probability of the word},  …  ]  }, …  ]  },  “aspects\_neg”:  {  “count”: the number of return aspects,  “list”:  [  {  “aspect\_id”: the id of the aspect,  “aspect\_score”: the score on the negative aspect,  “aspect\_name”: the name of the aspect,  “word\_count”: the number of words returned for an aspect,  “word\_list”:  [  {“word”: word, “pro”: probability of the word},  …  ]  }, …  ]  }  } |

**User Interface Layer**

Module List:

|  |  |
| --- | --- |
| Module | Functionality |
| Sign Up / Sign In | Sign up and sign in using a **Twitter** account. ***Call the interface provided by the data layer*** to add new users in the system. |
| Main Panel | Support queries, subscriptions in section “The Browser Side Components”, by calling the ***interfaces provided in application layer***. |

# Task distribution

The following table shows the distribution of the tasks, which are associated with the aforementioned. Please refer the section “Models & Interfaces” for more details.

|  |  |
| --- | --- |
| Contributor | Module to complete |
| Lisi Chen | **Publish/Subscribe (Application Layer)** |
| Kaiyu Feng | **Query Diversified Regions (Application Layer)** |
| Tao Guo | **Query POIs (Application Layer)** |
| Kaiqi Zhao | **User Behavior Model & Combiner**, **Automatic User Profiling**, **Personalized Recommendation** and **Business Aspect Analysis (Application Layer);**  **W4, SAR trainer (Persistent Layer)** |
| Yiding Liu | **User Panels and Map UI (UI Layer)** |
| Pengfei Li | **User Panels and Map UI (UI Layer)** |
| Pham Vu Tuan (URECA) | **User Panels and Map UI (UI Layer)** |
| Yimin Xu (URECA) | **Data Importer, Twitter Crawler (Data Layer)** |

**Low priority: 4SQ Check-in Crawler, Instagram Post Crawler (Data Layer)**

# Project Schedule

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Milestones | Nov 15, 2015 | Nov 30, 2015 | Dec 15, 2015 | Dec 31, 2015 |
| Finish building up **data layer** with existing data and start the streaming crawler |  |  |  |  |
| Finish building up **persistent layer** (phase I) and implementing **application layer** and **UI layer** |  |  |  |  |
| Integrate different layers and debug the system |  |  |  |  |
| Optimize memory usage of (phase II in persistent layer) the system |  |  |  |  |