



**NYU**

**TANDON SCHOOL  
OF ENGINEERING**

---

**CS 6083 - Principles of Database Systems (Fall 2018)**  
**Project Report – Phase 2**

**Done by:**

**Name: Kaustuk Kumar**

**Netid: kk3697**

**Name: Nikhil Narasimha Prasad**

**Netid: nnp267**

## **I. Introduction**

*Oingo* is a new mobile/web app that allows users to share useful information via their mobile devices/web browsers based on social, geographic, temporal, and keyword constraints. The main idea in oingo is that users can publish information in the form of short notes, and then link these notes to certain locations and certain times. Other users can then receive these notes based on their own location, the current time, and based on what type of messages they want to receive. This app has the potential to be a popular application. Here, we present the database backend of the app and its functionality.

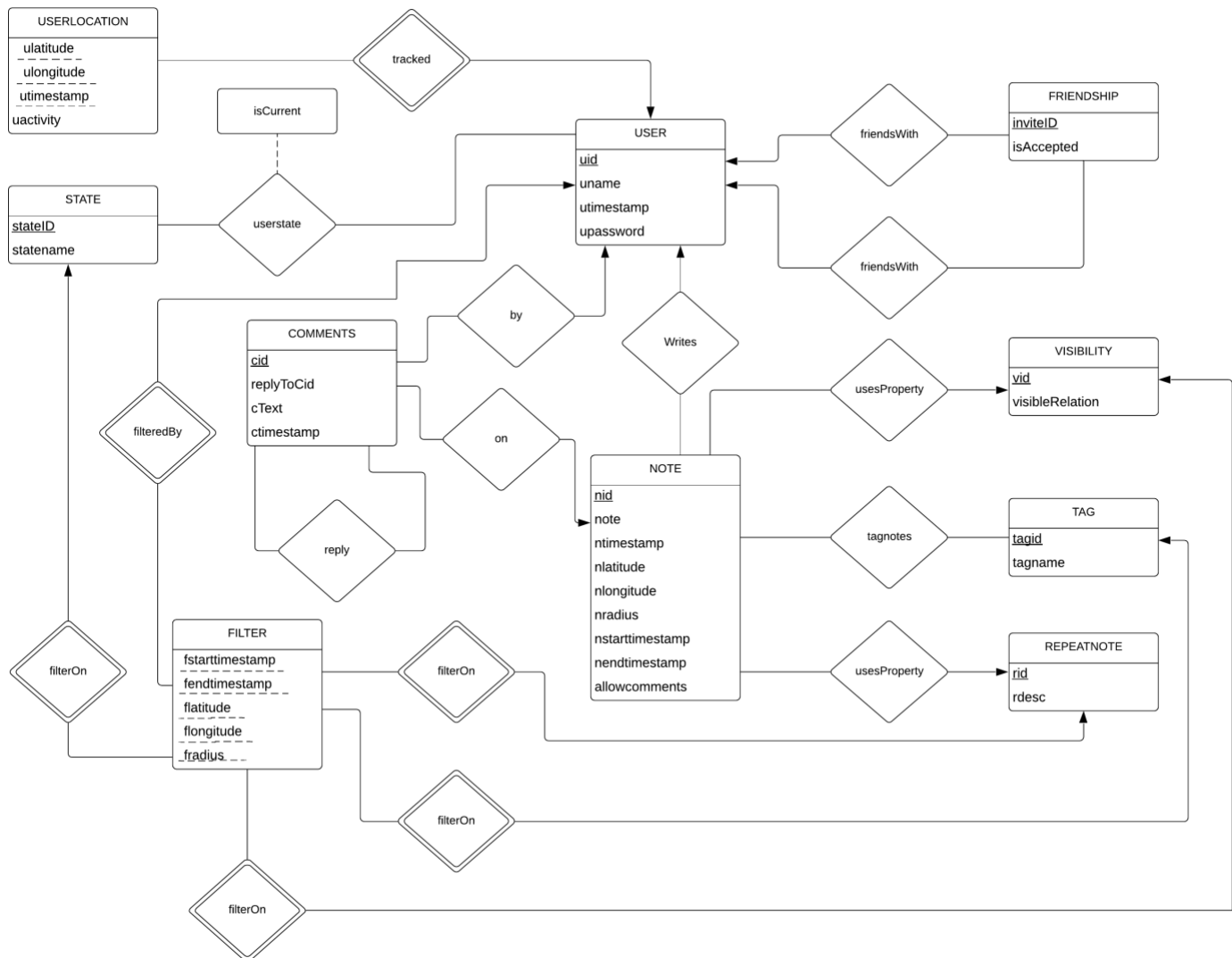
## **II. Entity – Relationship Diagram (ERD)**

In this section, we present the Entity – Relationship Diagram (ERD) for the app. The ERD represents the entities that are used in the app, along with their attributes, and their relationships. This provides the necessary information to derive a relational schema which can be implemented in the application.

The assumptions made while designing the ERD are as follows:

1. Every user is created with a default state.
2. Every time a user creates a new state, an entry will be made in the STATE and USERSTATE tables.
3. All user activity is logged in the USERLOCATION table.
4. A filter or a note's schedule can be repeated daily, weekly, bi-weekly, monthly, or yearly.
5. The unit of radius attribute in NOTE is in meters (m) and in FILTER is in kilometers (KM).

The ERD is shown below:



### Entity – Relationship Diagram

### III. Relational Schema

In this section, we present the relational schema that we have derived from the above ERD. This can be used to create the respective tables necessary for the app. The relational schema is as follows:

USER (uid, uname, uemail, utimestamp, upassword)

VISIBILITY (vid, visibleRelation)

REPEATNOTE (rid, rdesc)

NOTE (nid, uid, note, ntimestamp, nlatitude, nlongitude, nradius, nstarttimestamp, nendtimestamp, rid, vid, allowcomments)

FRIENDSHIP (inviteID, uid1, uid2, isAccepted)

TAG (tagid, tagname)

TAGNOTES (nid, tagid)

STATE (stateID, statename)

USERSTATE (uid, stateID, isCurrent)

USERLOCATION (uid, ulatitude, ulongitude, utimestamp, activity)

COMMENTS (cid, nid, uid, replyToCid, cText, ctimestamp)

FILTER (uid, stateid, tagID, fstarttimestamp, fendtimestamp, rid, flatitude, flongitude, fradius, vid)

The primary keys of all tables are the underlined attributes. The foreign key relationships are as follows:

NOTE.uid references USER.uid

NOTE.rid references REPEATNOTE.rid

NOTE.vid references VISIBILITY.vid

FRIENDSHIP.uid1 references USER.uid

FRIENDSHIP.uid2 references USER.uid

TAGNOTES.nid references NOTE.nid

TAGNOTES.tagid references TAG.tagid

USERSTATE.uid references USER.uid

USERSTATE.stateID references STATE.stateID

USERLOCATION.uid references USER.uid

COMMENTS.nid references NOTE.nid

COMMENTS.uid references USER.uid

(FILTER.uid, FILTER.stateID) references (USERSTATE.uid, USERSTATE.stateID)

FILTER.tagID references TAG.tagid

FILTER.rid references REPEATNOTE.rid

FILTER.vid references VISIBILITY.vid

## IV. Data Tables

The sample data we have used for testing the schema and the SQL queries are as follows:

### USER

	uid	uname	uemail	utimestamp	upassword
▶	1	Adam	abc@abc.com	2018-11-29 21:16:53.630	qwerty
	2	Derek	def@def.com	2018-11-29 21:16:53.630	zxcvb
	3	Guan	ghi@ghi.com	2018-11-29 21:16:53.630	12345
	NULL	NULL	NULL	NULL	NULL

This table is used to store the user account information. When a user creates a new account, a new entry is created which stores their information.

### VISIBILITY

	vid	visibileRelation
▶	0	PRIVATE
	1	FRIENDS
	2	ALL
	NULL	NULL

This table defines the various visibility types (PRIVATE, FRIENDS, or ALL) for all notes and filters.

### REPEATNOTE

▶	0	NO_REPEAT
	1	REPEAT_DAILY
	2	REPEAT_WEEKLY
	3	REPEAT_BIWEEKLY
	4	REPEAT_MONTHLY
	5	REPEAT_YEARLY
	NULL	NULL

This table defines the various repeat options available to the user, for the schedules of all notes and filters.

## NOTE

nid	uid	note	ntimestamp	nlatitude	nlongitude	nradius	nstarttimestamp	nendtimestamp	rid	vid	allowcomments
▶ 1	1	The library is a calm place to finish DB project.	2018-10-12 06:06:34.063	40.72951100	-73.99646000	300	2018-10-12 20:00:00	2018-10-12 23:00:00	1	1	0
2	2	The food here is amazing!	2018-01-05 11:07:42.531	40.70239300	-73.98733000	300	2018-01-06 03:30:00	2018-01-06 15:30:00	1	2	1
3	3	This bank has horrible customer service	2018-07-10 06:23:36.694	40.60309400	-73.99386900	300	2018-07-10 07:39:41	2018-07-10 14:30:00	2	2	1
4	2	This restaurant is so posh!	2018-11-29 18:22:02.831	40.72394400	-73.99969400	300	2018-11-29 20:45:00	2018-11-29 23:30:00	1	2	1
5	3	The park is so beautiful at this time of the year!	2018-10-10 05:56:42.751	40.78461900	-73.96531500	300	2018-10-14 11:00:00	2018-10-14 17:00:00	5	2	1
6	1	The water is so calm this evening. It's the best!	2018-11-29 19:31:44.643	40.72950000	-73.99640000	3000	2018-11-29 19:32:45	2018-11-29 23:00:00	1	0	1
7	3	This street has the best places to eat!	2018-11-29 15:12:34.845	40.72951000	-73.99643000	3000	2018-11-29 16:00:00	2018-11-29 23:55:00	1	2	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

This table is used to store all the note details, including its temporal and spatial information.

## FRIENDSHIP

inviteID	uid1	uid2	isAccepted
▶ 1	1	2	1
NULL	NULL	NULL	NULL

This table is used to store the friendship details, identified by the inviteID attribute.

## TAG

tagid	tagname
▶ -1	no tag
1	#studying
2	#gradschool
3	#nyu
4	#library
5	#bar
6	#pub
7	#goodfood
8	#food
9	#ambiance
10	#park
11	#scenery
12	#snow
13	#beautiful
14	#bank
15	#customerservice
16	#restaurant
17	#posh
NULL	NULL

This table stores all the hashtags (#).

## STATE

	stateID	statename
▶	0	Default
	1	focused
	2	chilling
	3	at work
	NULL	NULL

This table stores the various states for the user. Every time a user creates a new state, a new entry is inserted into this table.

## TAGNOTES

	nid	tagid
▶	1	1
	1	2
	1	3
	1	4
	2	5
	2	6
	2	7
	4	7
	2	8
	4	8
	7	8
	4	9
	6	9
	5	10
	6	10
	5	11
	6	11
	5	12
	4	13
	5	13
	3	14
	3	15
	4	16
	4	17
	NULL	NULL

This table stores the associations between notes and the hashtags (#).

#### USERSTATE

	uid	stateID	isCurrent
▶	1	0	0
	1	1	1
	2	0	0
	2	2	1
	3	0	1
	3	2	0
	3	3	0
	NULL	NULL	NULL

This table stores the association between the state and the user. The isCurrent field specifies the current state of the user.

#### USERLOCATION

	uid	ulatitude	ulongitude	utimestamp	activity
▶	1	40.65937600	-74.00460200	2018-10-11 09:59:47.131	became friends with uid 2
	1	40.72950000	-73.99640000	2018-11-29 19:31:44.643	nid 6 created
	1	40.72951100	-73.99646000	2018-10-11 04:21:26.508	friend request sent to uid 2
	1	40.72951100	-73.99646000	2018-10-12 07:06:34.063	nid 1 created
	1	40.72953000	-73.99654000	2018-11-29 21:30:00.000	searched for notes
	1	40.74881700	-73.98542800	2018-10-12 08:03:22.931	cid 2 created
	1	40.76204400	-73.97609400	2018-10-09 18:40:10.919	uid 1 created
	2	40.70239300	-73.98733000	2018-01-05 11:07:42.531	nid 2 created
	2	40.71297400	-74.01339700	2018-01-02 03:05:03.767	uid 2 created
	2	40.72390000	-73.99960000	2018-11-29 21:45:30.290	searched for notes
	2	40.72394400	-73.99969400	2018-07-10 05:39:42.663	nid 4 created
	2	40.75874000	-73.97867400	2018-10-12 07:27:23.057	cid 1 created
	3	40.60309400	-73.99386900	2018-07-10 06:23:36.694	nid 3 created
	3	40.66553500	-73.96974900	2018-07-10 12:16:41.438	cid 3 created
	3	40.68940800	-74.04446800	2018-07-06 13:36:53.213	uid 3 created
	3	40.72951000	-73.99643000	2018-11-29 15:12:34.845	nid 7 created
	3	40.78461900	-73.96531500	2018-10-10 05:56:42.751	nid 5 created
	NULL	NULL	NULL	NULL	NULL

This table acts as a log entry, that stores the user's location periodically, and on performing an activity.



## COMMENTS

	cid	nid	uid	replyToCid	cText	ctimestamp
▶	1	1	2	NULL	Haha yeah it is! I did my project there too!	2018-10-12 11:27:23.057
	2	1	1	1	I can see why. It's a quiet place with the books t...	2018-10-12 12:03:22.931
	3	4	3	NULL	Nice! I'll definitely visit this place!	2018-07-10 16:16:41.438
	NULL	NULL	NULL	NULL	NULL	NULL

This table stores the comments made by users on notes and other comments.

## FILTER

	uid	stateid	tagID	fstarttimestamp	fendtimestamp	rid	flatitude	flongitude	fradius	vid
▶	1	1	-1	0000-00-00 00:00:00	0000-00-00 00:00:00	0	40.78000000	-73.69600000	5.000000	2
	2	0	-1	0000-00-00 00:00:00	0000-00-00 00:00:00	0	-99.00000000	-999.00000000	0.000000	1
	3	0	-1	0000-00-00 00:00:00	0000-00-00 00:00:00	0	40.72900000	-73.99600000	0.200000	2
	3	2	-1	2018-11-29 19:00:00	2018-11-29 22:00:00	1	40.72300000	-73.99900000	0.500000	2
	2	2	7	2018-11-29 18:35:08	2018-11-29 22:00:00	1	-99.00000000	-999.00000000	0.000000	2
	2	2	9	2018-11-29 21:00:00	2018-11-29 23:00:00	1	-99.00000000	-999.00000000	0.000000	2
	1	0	12	2018-10-12 12:00:00	2018-10-12 16:00:00	1	40.78400000	-73.96500000	0.500000	2
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

This table stores the filter information created by each user. This is used to specify the notes the user is interested in viewing.

## V. SQL Queries

In this section, we test the schema and the populated data by executing the given SQL queries. All queries were executed on MySQL Workbench, running MySQL Server Version 5.7. These queries were discussed in the report for phase 1 as well. We have decided to include this section once more for a complete documentation.

### Query 1:

```
INSERT INTO `user` VALUES (1,'Adam','abc@abc.com','2018-11-30 02:16:53.630','qwerty');
```

### Query 2:


```
INSERT INTO `note` VALUES (4,2,'This restaurant is so posh!','2018-11-29 23:22:02.831',40.72394400,-73.99969400,300,'2018-11-30 01:45:00','2018-11-30 04:30:00',1,2,1);
INSERT INTO `tag` VALUES (7,'#goodfood'),(8,'#food'),(9,'#ambiance'),(13,'#beautiful'),(16,'#restaurant'),(17,'#posh');
INSERT INTO `tagnotes` VALUES (4,7),(4,8),(4,9),(4,13),(4,16),(4,17);
```

The tags can be seen in the INSERT INTO 'tag' statement. The temporal constraints are the timestamp values defined in the first INSERT statement. Similarly, the spatial constraints are the latitude and longitude values (in decimal format) defined in the first INSERT statement.

### Query 3:

```
1 • select uid2 as Friends
2   from friendship
3   where uid1 = 2
4   union
5   select uid1 as Friends
6   from friendship
7   where uid2 = 2;
```

<

Result Grid  Filter Rows:


	Friends
▶ 1	

As defined in the test table FRIENDSHIP, user 1 (Adam) and user 2 (Derek) are friends. Hence, when the above query is executed, the output is as expected.

### Query 4:

```
1 • call GETNOTES(2);
```

<

Result Grid  Filter Rows:

	nid	note
▶ 4		This restaurant is so posh!

```

CREATE DEFINER='root'@'localhost' PROCEDURE `GETNOTES`(IN inputuserid INT)
BEGIN
    select nid, note
from (select note.uid, nid, note, tagid, nlatitude, nlongitude, nradius, nstarttimestamp, nendtimestamp, rid, user.uid as visibletouser
      from note natural join tagnotes, user
      where CASE vid
        WHEN 0
        Then user.uid = note.uid
        WHEN 1
        THEN user.uid in (select uid2 as Friends
                        from friendship
                        where uid1 = note.uid
                        union
                        select uid1 as Friends
                        from friendship
                        where uid2 = note.uid
                        union
                        select uid from user
                        where uid = user.uid)
        ELSE user.uid in (select uid from user)
      END) notetag, (select filter.uid, tagid, fstarttimestamp, fendtimestamp, flatitude, flongitude, ulatitude, ulongitude, uloc.utimestamp, rid, user.uid as visiblefromuser
                    from filter, user, (select uid, ulatitude, ulongitude, utimestamp
                    from userlocation u1
                    where utimestamp = (select max(utimestamp)
                                      from userlocation u2
                                      where u1.uid = u2.uid)) uloc
                    where stateid = (select stateid from userstate where uid = filter.uid and iscurrent = 1) and uloc.uid = filter.uid and
                    CASE vid
        WHEN 0
        Then user.uid = filter.uid
        WHEN 1
        Then user.uid in (select uid2 as Friends
                        from friendship
                        where uid1 = filter.uid
                        union
                        select uid1 as Friends
                        from friendship
                        where uid2 = filter.uid
                        union
                        select uid from user
                        where uid = user.uid)
        ELSE user.uid in (select uid from user)
      END) filtertag
where filtertag.uid = inputuserid and CASE filtertag.tagid
  WHEN '-1' THEN notetag.tagid != ''
  ELSE notetag.tagid = filtertag.tagid
END
and CASE
  WHEN (filtertag.fstarttimestamp != '0000-00-00 00:00:00' and filtertag.fendtimestamp != '0000-00-00 00:00:00')
  THEN
    intime(filtertag.utimestamp, filtertag.fstarttimestamp, filtertag.fendtimestamp, filtertag.rid) = 1
  WHEN (filtertag.fstarttimestamp != '0000-00-00 00:00:00' and filtertag.fendtimestamp = '0000-00-00 00:00:00')
  THEN
    intime(filtertag.utimestamp, filtertag.fstarttimestamp, '', filtertag.rid) = 1
  ELSE filtertag.utimestamp != ''
END
and intime(filtertag.utimestamp, notetag.nstarttimestamp, notetag.nendtimestamp, notetag.rid) = 1
and notetag.visibletouser = filtertag.uid and filtertag.visiblefromuser = notetag.uid
and haversine(filtertag.ulatitude, filtertag.ulongitude, notetag.nlatitude, notetag.nlongitude) <= notetag.nradius
and CASE
  WHEN (filtertag.flatitude != -99.00000000 and filtertag.flongitude != -999.00000000)
  Then (haversine(filtertag.ulatitude, filtertag.ulongitude, filtertag.flatitude, filtertag.flongitude)/1000) <= notetag.nradius
  ELSE
    filtertag.ulatitude != ''
  END
group by nid, note;
END ;;

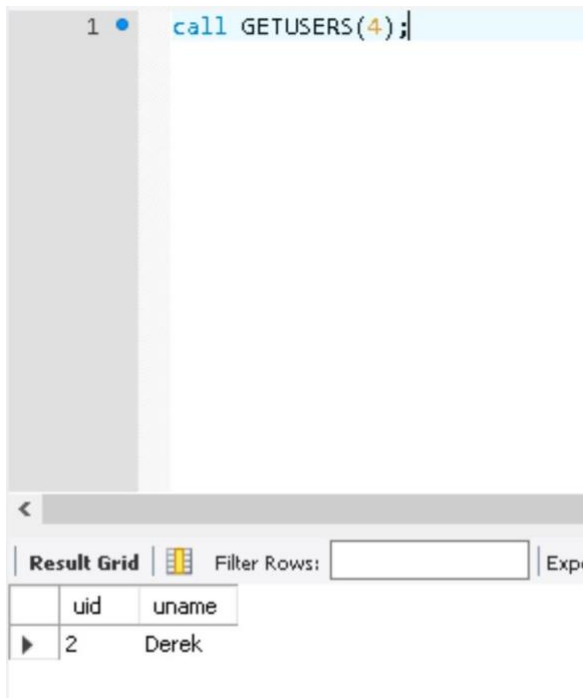
```

Here, we are looking for the notes that can be viewed by user 2 (Derek). The user's current location and current timestamp is stored in the USERLOCATION table and the user's current state is present in the USERSTATE table. When the user is within the radius of their filter and the radius of the note coincides with this, only then can the user be able to view that note. We

have created a procedure called GETNOTES (that contains the actual SQL query), which takes the uid as input, and retrieves the corresponding noteID and the note.

### **Query 5:**

Here, we define a procedure GETUSERS (that contains the actual SQL query), which takes the noteID as input, and retrieves all the users who can view this note depending on their filters and their last recorded location.



The procedure GETUSERS is shown below:

```

CREATE DEFINER='root'@'localhost' PROCEDURE `GETUSERS`(IN inputnoteid INT)
BEGIN
select filtertag.uid, uname
from (select note.uid, nid, note, tagid, nlatitude, nlongitude, nradius, nstarttimestamp, nendtimestamp, rid, user.uid as visibletouser
      from note natural join tagnotes, user
      where CASE vid
        WHEN 0
        Then user.uid = note.uid
        WHEN 1
        THEN user.uid in (select uid2 as Friends
                        from friendship
                        where uid1 = note.uid
                        union
                        select uid1 as Friends
                        from friendship
                        where uid2 = note.uid
                        union
                        select uid as Friends
                        from user where uid = user.uid)
        ELSE user.uid in (select uid from user)
      END) notetag, (select filter.uid, tagid, fstarttimestamp, fendtimestamp, flatitude, flongitude, ulatitude, ulongitude, uloc.utimestamp, rid, user.uid as visiblefromuser
                    from filter, user, (select uid, ulatitude, ulongitude, utimestamp
                                      from userlocation ul
                                      where utimestamp = (select max(utimestamp)
                                                         from userlocation u2
                                                         where u1.uid = u2.uid)) uloc
                    where stateid = (select stateid from userstate where uid = filter.uid and iscurrent = 1) and uloc.uid = filter.uid and
                    CASE vid
                    WHEN 0
                    Then user.uid = filter.uid
                    WHEN 1
                    Then user.uid in (select uid2 as Friends
                                      from friendship
                                      where uid1 = filter.uid
                                      union
                                      select uid1 as Friends
                                      from friendship
                                      where uid2 = filter.uid
                                      union
                                      select uid from user
                                      where uid = user.uid)
                    ELSE user.uid in (select uid from user)
                    END) filtertag, user
where notetag.nid = inputnoteid and CASE filtertag.tagid
  WHEN '-1' THEN notetag.tagid != ''
  ELSE notetag.tagid = filtertag.tagid
END
and CASE
  WHEN (filtertag.fstarttimestamp != '0000-00-00 00:00:00' and filtertag.fendtimestamp != '0000-00-00 00:00:00')
  THEN
    intime(filtertag.utimestamp, filtertag.fstarttimestamp, filtertag.fendtimestamp, filtertag.rid) = 1
  WHEN (filtertag.fstarttimestamp != '0000-00-00 00:00:00' and filtertag.fendtimestamp = '0000-00-00 00:00:00')
  THEN
    intime(filtertag.utimestamp, filtertag.fstarttimestamp, '', filtertag.rid) = 1
  ELSE filtertag.utimestamp != ''
END
and intime(filtertag.utimestamp, notetag.nstarttimestamp, notetag.nendtimestamp, notetag.rid) = 1
and notetag.visibletouser = filtertag.uid and filtertag.visiblefromuser = notetag.uid
and haversine(filtertag.ulatitude, filtertag.ulongitude, notetag.nlatitude, notetag.nlongitude) <= notetag.nradius
and CASE
  WHEN (filtertag.flatitude != -99.00000000 and filtertag.flongitude != -999.00000000)
  Then (haversine(filtertag.ulatitude, filtertag.ulongitude, filtertag.flatitude, filtertag.flongitude)/1000) <= notetag.nradius
  ELSE
    filtertag.ulatitude != ''
END
and filtertag.uid = user.uid
group by filtertag.uid;
-END ;;

```

## Query 6:

All the notes that can be viewed by user 1 (Adam) are shown below.

```
1 • call GETNOTES(1);
```

Result Grid		Filter Rows:	Export:
nid	note		
1	The library is a calm place to finish DB project.		
6	The water is so calm this evening. It's the best!		
7	This street has the best places to eat!		

We have used FULLTEXT search to filter the notes based on keywords. The procedure for this is GETNOTESKEYWORDS, which takes the uid and keywords as input, and display the filtered notes.

```
1 • call GETNOTESKEYWORDS(1, 'calm');
```

Result Grid		Filter Rows:	Export:	Wrap C
nid	note			
1	The library is a calm place to finish DB project.			
6	The water is so calm this evening. It's the best!			



```

CREATE DEFINER='root'@'localhost' PROCEDURE `GETNOTESKEYWORDS`(IN inputuserid INT, IN keywords VARCHAR(45))
BEGIN
    select nid, note
from (select note.uid, nid, note, tagid, nlatitude, nlongitude, nradius, nstarttimestamp, nendtimestamp, rid, user.uid as visibletouser
      from note natural join tagnotes, user
      where CASE vid
      WHEN 0
      Then user.uid = note.uid
      WHEN 1
      THEN user.uid in (select uid2 as Friends
                        from friendship
                        where uid1 = note.uid
                        union
                        select uid1 as Friends
                        from friendship
                        where uid2 = note.uid
                        union
                        select uid from user
                        where uid = user.uid)
      ELSE user.uid in (select uid from user)
      END) notetag, (select filter.uid, tagid, fstarttimestamp, fendtimestamp, flatitude, flongitude, ulatitude, ulongitude, uloc.utimestamp, rid, user.uid as visiblefromuser
                    from filter, user, (select uid, ulatitude, ulongitude, utimestamp
                    from userlocation u1
                    where utimestamp = (select max(utimestamp)
                                         from userlocation u2
                                         where u1.uid = u2.uid)) uloc
                    where stateid = (select stateid from userstate where uid = filter.uid and iscurrent = 1) and uloc.uid = filter.uid and
                    CASE vid
                    WHEN 0
                    Then user.uid = filter.uid
                    WHEN 1
                    Then user.uid in (select uid2 as Friends
                                      from friendship
                                      where uid1 = filter.uid
                                      union
                                      select uid1 as Friends
                                      from friendship
                                      where uid2 = filter.uid
                                      union
                                      select uid from user
                                      where uid = user.uid)
                    ELSE user.uid in (select uid from user)
                    END) filtertag
where filtertag.uid = inputuserid and CASE filtertag.tagid
    WHEN '-1' THEN notetag.tagid != ''
    ELSE notetag.tagid = filtertag.tagid
    END
and CASE
    WHEN (filtertag.fstarttimestamp != '0000-00-00 00:00:00' and filtertag.fendtimestamp != '0000-00-00 00:00:00')
    THEN
        intime(filtertag.utimestamp, filtertag.fstarttimestamp, filtertag.fendtimestamp, filtertag.rid) = 1
    WHEN (filtertag.fstarttimestamp != '0000-00-00 00:00:00' and filtertag.fendtimestamp = '0000-00-00 00:00:00')
    THEN
        intime(filtertag.utimestamp, filtertag.fstarttimestamp, '', filtertag.rid) = 1
    ELSE filtertag.utimestamp != ''
    END
and intime(filtertag.utimestamp, notetag.nstarttimestamp, notetag.nendtimestamp, notetag.rid) = 1
and notetag.visibletouser = filtertag.uid and filtertag.visiblefromuser = notetag.uid
and haversine(filtertag.ulatitude, filtertag.ulongitude, notetag.nlatitude, notetag.nlongitude) <= notetag.nradius
and CASE
    WHEN (filtertag.flatitude != -99.00000000 and filtertag.flongitude != -999.00000000)
    Then (haversine(filtertag.ulatitude, filtertag.ulongitude, filtertag.flatitude, filtertag.flongitude)/1000) <= notetag.nradius
    ELSE
        filtertag.ulatitude != ''
    END
and MATCH(note) AGAINST(keywords IN NATURAL LANGUAGE MODE)
group by nid, note;

END ;;

```

The functions that we have used in the above queries are:

1. INTIME: Checks if a specific timestamp is within a certain time frame.

```
CREATE DEFINER='root'@'localhost' FUNCTION `intime`(utime timestamp, starttime timestamp, endtime timestamp, rid int) RETURNS int(11)
BEGIN
  DECLARE RET INT DEFAULT 0;
  Select CASE
    WHEN (starttime != '' and endtime != '')
    THEN
      CASE rid
        WHEN 0
        THEN
          CASE
            WHEN(utime >= starttime and utime <= endtime)
            THEN 1
            ELSE 0
          END
        WHEN 1
        THEN
          CASE
            WHEN((time(utime) Between time(starttime) and time(endtime)) and utime >= starttime)
            THEN 1
            ELSE 0
          END
        WHEN 2
        THEN
          CASE
            WHEN((time(utime) Between time(starttime) and time(endtime)) and DAYOFWEEK(utime) = DAYOFWEEK(starttime))
            THEN 1
            ELSE 0
          END
        WHEN 3
        THEN
          CASE
            WHEN((time(utime) Between time(starttime) and time(endtime)) and DAYOFWEEK(utime) = DAYOFWEEK(starttime)
              and (((WEEK(utime) - WEEK(starttime)) % 2) = 0) and utime >= starttime)
            THEN 1
            ELSE 0
          END
        WHEN 4
        THEN
          CASE
            WHEN((time(utime) Between time(starttime) and time(endtime)) and DAY(utime) = DAY(starttime)
              and MONTH(utime) >= MONTH(starttime))
            THEN 1
            ELSE 0
          END
        WHEN 5
        THEN
          CASE
            WHEN((time(utime) Between time(starttime) and time(endtime))
              and MONTH(utime) = MONTH(starttime) and DAY(utime) = DAY(starttime)
              and YEAR(utime) >= YEAR(starttime))
            THEN 1
            ELSE 0
          END
        END
      WHEN (starttime != '' and endtime = '')
      THEN
        CASE rid
          WHEN 0
          THEN
            CASE
              WHEN(utime = starttime)
              THEN 1
              ELSE 0
            END
          WHEN 1
          THEN
            CASE
              WHEN((time(utime) = time(starttime)) and utime >= starttime)
              THEN 1
              ELSE 0
            END
          WHEN 2
          THEN
            CASE
              WHEN((time(utime) = time(starttime)) and DAYOFWEEK(utime) = DAYOFWEEK(starttime))
              THEN 1
              ELSE 0
            END
          WHEN 3
          THEN
            CASE
              WHEN((time(utime) = time(starttime)) and DAYOFWEEK(utime) = DAYOFWEEK(starttime)
                and (((WEEK(utime) - WEEK(starttime)) % 2) = 0) and utime >= starttime)
              THEN 1
              ELSE 0
            END
          WHEN 4
          THEN
            CASE
              WHEN((time(utime) = time(starttime)) and DAY(utime) = DAY(starttime)
                and MONTH(utime) >= MONTH(starttime))
              THEN 1
              ELSE 0
            END
          WHEN 5
          THEN
            CASE
              WHEN((time(utime) = time(starttime))
                and MONTH(utime) = MONTH(starttime) and DAY(utime) = DAY(starttime)
                and YEAR(utime) >= YEAR(starttime))
              THEN 1
              ELSE 0
            END
          END
        ELSE 0
      END as ifexists into ret;
  Return ret;
END ;;
```



2. Haversine: This is used to determine the distance between two lat long coordinates.

```
CREATE DEFINER='root'@'localhost' FUNCTION `haversine`(  
    lat1 FLOAT, lon1 FLOAT,  
    lat2 FLOAT, lon2 FLOAT  
    ) RETURNS float  
NO SQL  
DETERMINISTIC  
COMMENT 'Returns the distance in degrees on the Earth between two known points of latitude and longitude. To get miles, multiply by 3961, and km by 6373'  
BEGIN  
  
    RETURN 6371000 * (ACOS(  
        COS(RADIANS(lat1)) *  
        COS(RADIANS(lat2)) *  
        COS(RADIANS(lon2) - RADIANS(lon1)) +  
        SIN(RADIANS(lat1)) * SIN(RADIANS(lat2))  
    ));  
  
END ;;
```

## VI. Schema Changes for Phase 2

This section discusses the changes we made to the original database schema, to improve the working of the web application, and to make it more efficient. These changes include:

- Updated uid attribute in USER table to auto-increment.
- Updated nID attribute in NOTE table to auto-increment.
- Updated cid attribute in COMMENTS table to auto-increment.
- Updated inviteID attribute in FRIENDSHIP table to auto-increment.
- Updated tagid attribute in Tag table to auto-increment.
- Changed size of upassword attribute in USER table from 45 to 255 to allow insertion of hashed passwords.
- Implemented a trigger such that, when a new user registers a profile, their current state is set to “Default”. The trigger is shown below.

```
DELIMITER $$  
create trigger after_user_register  
    after insert on user  
    for each row begin  
        insert into userstate (uid, stateID, isCurrent) VALUES (NEW.uid, 0, 1);  
    END$$
```

## VII. Web-based User Interface Design

In this section, we discuss the technologies used to implement a web-based front-end for the above database, and the actual interface itself. The following technologies have been implemented:

- HTML5
- CSS
- JavaScript
- XML
- AJAX

The backend databases were developed using MySQL Workbench, running MySQL Server Version 5.7 and MySQL Server Version 8.0.13 (for certain functions). The query transactions were performed using PHP Version 5.6, and PHP Version 7.2.11. The PHP code can be run on any system with a modern web browser, and running Apache Web Server (XAMPP, or equivalent).

The maps interfaces were implemented using Google Maps API.

The first page displayed to the user is the **Login** page. Here, a returning user can enter their credentials, and on successful verification, will be taken to the **Home** page. New users can register by providing a username, email, and a password. PHP includes a function `password_hash()`, that takes a string (in this case, the user's password) and the hashing algorithm to use. In this case, we are using PHP's default algorithm - `password_default` - to hash the user's password before storing their credentials in the database.

On successful login, the user is directed to the **Home** page. Here, the user can see their credentials and their current state. The user can also see the markers of various notes published by other users, and their current location marker on the map interface. The user's current location is automatically tracked and displayed on the map interface. If desired, the current location can be changed manually by clicking on the "Manually Update Location" button, and then dragging their marker on the map to any desired location, and then clicking on "Update Location" button. They can also see all the notes posted by them.

# Login

Username

Adam

Password

●●●●●●

Login

Not yet a member? [Sign up](#)

**Fig: Login Page**

# Register

Username

Email

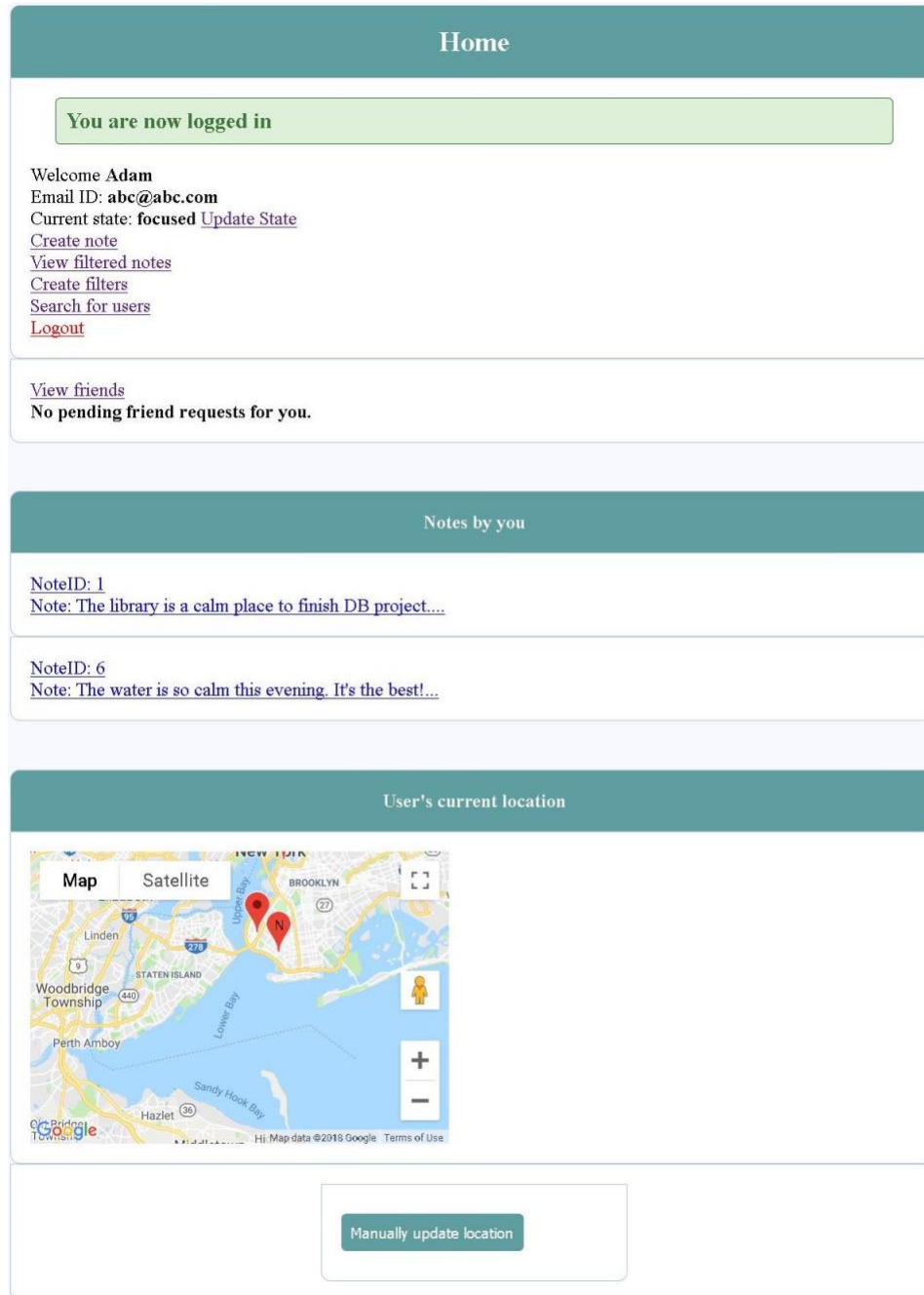
Password

Confirm Password

Register

Already a member? [Sign in](#)

**Fig: Registration Page**



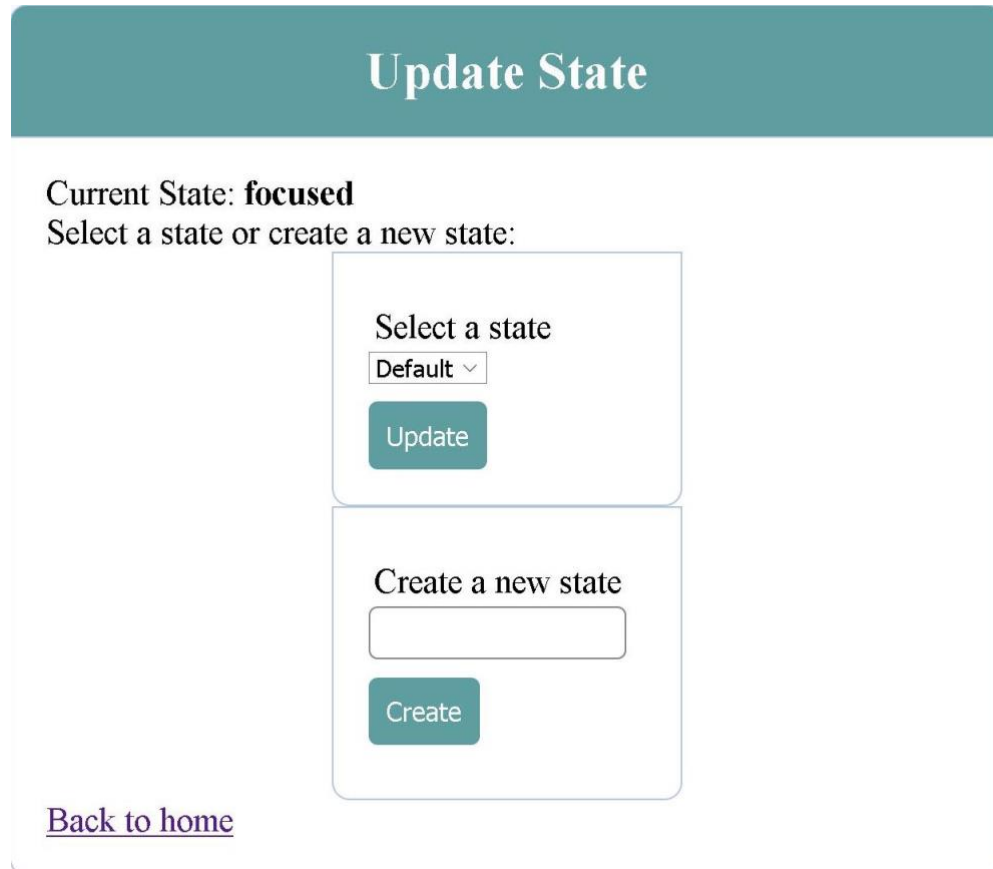
**Fig: Home Page**

Various links are provided for the user, using which they can navigate and perform actions throughout the app. The functionalities provided for the user are as follows.

- Update State

Every user is assigned a default state, “Default”, when their profile is initially created. The user can change their current state if desired, by selecting an existing state from the drop-

down menu, or by typing in a new state. This will be their current state, and will be used as a parameter in retrieving notes from other users.



The image shows a web form titled "Update State" in a teal header. Below the header, the text "Current State: **focused**" is displayed. Underneath, it says "Select a state or create a new state:". There are two main sections: the top one is "Select a state" with a dropdown menu showing "Default" and a teal "Update" button; the bottom one is "Create a new state" with a text input field and a teal "Create" button. At the bottom left, there is a purple link that says "Back to home".

**Fig: Update State**

- Create Note

Here, the user can create a note, and set various parameters for the note. These parameters include the following:

- Tags: Here, the user can include tags of their interest. Each tag must begin with the '#' character.
- Repeat Value: The user can choose the duration of time the note must be repeated, or choose not to repeat a note.
- Visibility: The user can choose who is allowed to view this particular note, or can choose to make the note private.
- Start Date: The user can select the date on when the note will be published on the app, via a calendar interface.
- Start Time: The user can select the time of the day the note will be published.
- End Time: The user can select the time of the day until when the note will be available on the app.
- Location: The user can select the location of the note via a map interface. The marker on the map can be moved to any location on the map.

- Radius: The user can specify the radius of interest, around the location on the note.
- Allow Comments: The user can decide whether other users can comment on this note or not.

## Create Note

Enter your note here

Tags

Select a Repeat  
None

Select a Visibility  
Private

Select a Start Date (YYYY-MM-DD)

Select a Start Time:  
-- : -- --

Select an End Time:  
-- : -- --

Map
Satellite

Select the note location by dragging the marker on the map to the desired location

Enter Radius (in meters):

Allow comments from other users?  
Yes

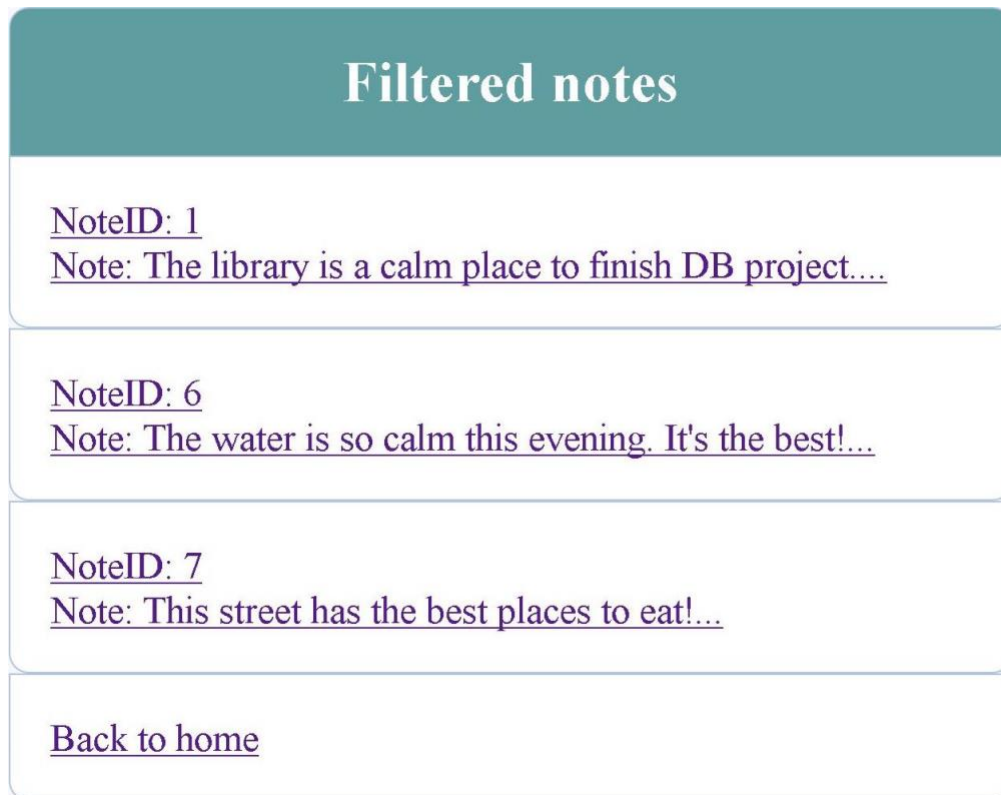
Create Note

[Back to Home](#)

**Fig: Create Note Page**

- View Filtered Notes

Here, the user can view notes posted by other users, based on the filters he has set, and the parameters set by the other user. If there are any notes to be viewed, the user can click on any of them to view all details of the note, including the note itself, any comments, and the location of the note on the map interface. Here, the user can comment on any note, or reply to a comment. The user cannot reply to a previous reply, however.



**Fig: Filtered Notes Page**

- Create Filters

The user can view all current filters set by them at the top of the page. If no current filters are set, a message will be displayed, prompting the user to create a new filter. These filters are used to filter user notes and display the resulting notes to the user. These filters include the following attributes: Tag, Repeat, Visibility, Start Date, Start Time, End Time, Location, Radius (in KM). The create filter page is shown below.

## Filters

Tag	State	Start Date	Start Time	End Time	Latitude	Longitude	Radius (in KM)	Repeat	Visibility
#snow	Default	2018-10-12	12:00:00	16:00:00	40.78400000	-73.96500000	0.500000	Daily	All
no tag	focused	0000-00-00	00:00:00	00:00:00	40.78000000	-73.69600000	5.000000	None	All

Create new filter

Select a Tag

no tag

Select a Repeat

None

Select a Visibility

Private

Select a Start Date

Select a Start Time:

-- : -- --

Select an End Time:

-- : -- --

Map

Satellite

Select the filter location by dragging the marker on the map to the desired location

Enter Radius (in km):

Create Filter

[Back to Home](#)

**Fig: Create Filters Page (with set filters)**

- Search for Users

The current user can search for other users using this option. If the searched user exists, the current user can visit their profile, and can send a friend request, if desired.





The 'Search for user' page features a teal header with the title 'Search for user'. Below the header is a white form area containing a 'Username' label, a text input field, a teal 'Search' button, and a purple link 'Back to Home'.

**Fig: User Search Page**



The 'User Profile' page has a teal header with the title 'User Profile'. The main content area is white and divided into three sections. The top section displays 'Welcome to Guan's profile', 'Email: ghi@ghi.com', and 'Current state: Default'. The middle section contains a teal 'Send Friend Request' button. The bottom section features a purple link 'Back to home'.

**Fig: User Profile Page after searching (with option to send friend request)**

- View Friends

If there are any pending friend requests for the current user, these will be displayed here, until the user accepts the requests. The user can also view their friends, and if they have any, can visit their profile and check their information.



The 'View Friends' page has a teal header with the title 'View Friends'. The main content area is white and divided into two sections. The top section displays the name 'Derek' in purple. The bottom section features a purple link 'Back to home'.

**Fig: View Friends Page**

## **VII. Security Features**

Every action performed by the user is kept track of and this information is logged in the **userlocation** table, along with other values such as the timestamp of their recent activities, and location information. This is very handy for debugging purposes, and for ensuring the system is running smoothly.

Every major database operation is performed using a prepared statement. Using prepared statements enable us to execute the same (or similar) SQL queries repeatedly with high efficiency. Comparing to executing SQL statements directly, prepared statements have three main advantages:

- Prepared statements reduce parsing time as the preparation on the query is done only once (although the statement is executed multiple times).
- Bound parameters minimize bandwidth to the server as you need to send only the parameters each time, and not the entire query.
- Prepared statements are very useful against SQL injections, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.

We have also implemented transactions in the design in order to enable multiple users to use the application simultaneously. This ensures that concurrency is maintained among all database operations without overlapping with each other.