Mohamed Eid

Professor Alan Labouseur

CMPT 308

2 December 2013

The Tacker Database System

About Tacker.

Tacker is a simple social application for sharing one's current location with friends. It is essentially a "Snapchat" for sharing geo-tagged maps pinpointing where an individual is; alternatively one might call it a social version of "Find My Friends." The application also provides a built in messaging service with the ability to create a new conversation for each location share.

How does it work?

Users send their friends requests asking to share their location. A push notification is sent to the requested user so that he or she may be alerted of the request. If the requested user chooses to accept the request, a tracker containing that user's coordinates is created. The requesting user is then able to see where the requested user is and can get directions to that user.

How will the application be used?

Tacker will be used as a general tracking application to find friends and directions to their coordinates. The hope is to replace the routine of sending text messages containing only text through SMS to find out where a friend is. Whether two individuals are in a city, a concert, or a crowded festival, Tacker will always be better option than SMS because it provides a visual of where the other person is and removes possibly human error when it comes to providing the location.

Database Creations:

These are the SQL queries to create the Tacker databases. There is one database for each environment.

```
-- create databases
CREATE DATABASE test_tacker;
CREATE DATABASE development_tacker;
CREATE DATABASE production_tacker;
```

Access Rights:

Different users with different access to the three databases are created with permissions. Having three different users with limited permissions is better than having one super user in case one gets access to the password that ought to be encrypted and separate from the test and development environments.

```
-- Create the user.
CREATE ROLE test_postgres WITH LOGIN PASSWORD 'some_password';
```

```
CREATE ROLE development_postgres WITH LOGIN PASSWORD 'some_password';
CREATE ROLE production_postgres WITH LOGIN PASSWORD 'some_password';
```

```
-- prevent all users from being able to use the database, unless they have been specifically granted permission.
REVOKE ALL PRIVILEGES ON DATABASE production_tacker FROM PUBLIC;
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA public FROM PUBLIC;
REVOKE ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public FROM PUBLIC;

-- grant user connect permissions
GRANT CONNECT ON DATABASE test_tacker TO test_postgres;
GRANT CONNECT ON DATABASE development_tacker TO development_postgres;
GRANT CONNECT ON DATABASE production_tacker TO production_postgres;
```

The Design **(variables sent by the API are written as #{var}):**

The database design is simple. Tacker is currently in development using Postrges 8.4.18 as the database management system. There are three databases including a test database (test_tacker), a development database (development_tacker), and a production database (production_tacker). There are sixteen database tables.

Users.

Everything revolves around the user. With no user, there is obviously no Tacker. A user record contains a unique id as the primary key along with a couple other unique columns including “email and” “phone_number.”

```
-- users table
CREATE TABLE users (
  id integer,
  email char(60) UNIQUE,
  is_private boolean DEFAULT = false,
  phone_number char(13) UNIQUE,
  username char(20) UNIQUE,
  reset_token char(200) NOT NULL,
  salt char(200) NOT NULL,
  created_at date,
  updated_at date,
  PRIMARY KEY (id)
);
CREATE INDEX users_index
ON users (id);
```

A simple users view is generated for the first version of API. This is a simple virtual table for seeing users.

```
CREATE VIEW v1_users_view AS
```

```
SELECT id, is_private, username
FROM users;
```

Sample users data:

id	email	Is_private	Phone_number	username	reset_token	salt	Created_at	Updated_at
1	mohamedkeid@gmail.com	false	1111111111	mo	37f2359gh	Fd3kj	10-10-13	10-10-13
2	test@gmail.com	false	2222222222	test	Jgfd8787g	7667S	10-10-13	10-10-13
3	something@gmail.com	true	3333333333	something	Jgkfsgjfgd	H883g	10-10-13	10-10-13

Email is stored because it provides the application an address to send “Password Reset” emails if the user ever forgets his or her password. Email is also stored to allow a user to sign in if he or she were to forget his or her username.

Phone numbers are stored because it provides a way for the application to suggest potential friends. The contacts of a user are sent to the Tacker server where it then searches through the database for users with that phone number.

```
-- suggest friends
SELECT *
FROM users
WHERE phone_number
IN #{phone_number_array}
AND id NOT IN
(
```

```
SELECT id
FROM users
JOIN friendships
ON users.id = friendships.followed_user_id
WHERE  friendships.following_user_id = #{current_user_id}
);
```

Friendships.

Relationships between users are the glue to everything. Without the friendships table, users would not be able to have friends and thus Tacker would no longer be a social application.

```
-- friendships table
CREATE TABLE friendships (
  following_user_id integer NOT NULL,
  followed_user_id integer NOT NULL,
  following_user_name char(40) ,
  created_at date,
  PRIMARY KEY (following_user_id, followed_user_id),
  FOREIGN KEY (following_user_id) REFERENCES users(id),
  FOREIGN KEY (followed_user_id) REFERENCES users(id)
);
CREATE INDEX friendships_index
ON friendships (following_user_id, followed_user_id);
```

A simple friendships view is generated for the first version of API. This is a simple virtual table for seeing friendships.

```
CREATE VIEW v1_friendships_view AS
SELECT friending_user_id, friended_user_id, friended_user_name
FROM friendships;
```

Sample friendships data:

following_user_id	followed_user_id	followed_user_name	created_at
1	2	Test	10-10-13
2	1	Mo	10-10-13
1	3	NULL	10-10-13

The application never selects friended users from the database directly. Instead, it retrieves the friendships of a user and has each friendship record have a “has_one” with a friended user. This is not done in a view however because we cannot index nested views. It also gives the application context because we have the is_approved attribute available, indicating whether or not the friendship has been accepted by a private user.

```
-- select friend from friendship table
SELECT *
FROM users
WHERE id = #{followed_user_id}
```

```
LIMIT = 1;
```

Blocked Users.

One can block users in Tacker, preventing them from being able to send you requests. This is done with another simple table called the blocked_users table which includes a column called blocked_user_id which identifies the blocked user and blocking_user_id which identifies the blocking user.

```
-- blocked_users table
CREATE TABLE blocked_users (
  blocked_user_id integer NOT NULL,
  blocking_user_id integer NOT NULL,
  created_at date,
  PRIMARY KEY (blocked_user_id, blocking_user_id),
  FOREIGN KEY (blocked_user_id) REFERENCES users(id),
  FOREIGN KEY (blocking_user_id) REFERENCES users(id)
);
CREATE INDEX blocked_users_index
ON blocked_users (blocked_user_id, blocking_user_id);
```

Sample blocked users data:

blocking_user_id	blocked_user_id	created_at
2	3	10-10-13

Retrieving the blocked users of a user is done like so:

```
-- select user blocked users
```

```
SELECT *
FROM blocked_users
LEFT INNER JOIN users
ON users.id = blocked_users.blocking_user_id
WHERE blocking_user_id = #{blocking_user_id};
```

Just like with friendships, Tacker never selects blocked users from the database directly. It retrieves a list of blocked user records belonging to the user and has a “has_one” with a linked blocked user. Once again, this is done because we cannot index nested views. It also gives context as we have the blocked_user record’s id. This is convenient for the API if the user wants to unblock a user because we it does not require it to re-query the database for that record.

```
-- select blocked user from blocked users table
SELECT *
FROM users
WHERE id = #{blocked_user_id}
LIMIT = 1;
```

A simple blocked_users view is generated for the first version of API. This is a simple virtual table for seeing blocked users.

```
CREATE VIEW v1_blocked_users_view AS
SELECT blocking_user_id, blocked_user_id
```

```
FROM blocked_users;
```

Requests.

Sending requests is the driving force between the interactivity of Tacker. It is a way for users to create friendships with other users and to share locations. When a request is created, a push notification is also sent to the requested user creating real world interactivity.

```
-- requests table
CREATE TABLE requests (
  requesting_user_id integer NOT NULL,
  requested_user_id integer NOT NULL,
  type char(15),
  latitude float,
  longitude float,
  created_at date,
  PRIMARY KEY (requesting_user_id, requestes_user_id),
  FOREIGN KEY (requesting_user_id) REFERENCES users(id),
  FOREIGN KEY (requested_user_id) REFERENCES users(id)
);
CREATE INDEX requests_index
ON requests (requesting_user_id, requested_user_id);
```

A simple requests view is generated for the first version of API. This is a simple virtual table for seeing requests.

```
CREATE VIEW v1_requests_view AS
SELECT requesting_user_id, requested_user_id, latitude, longitude
FROM requests;
```

Sample requests data:

requesting_user_id	requested_user_id	type	latitude	longitude	created_at
1	3	friendship	NULL	NULL	10-10-13
1	2	tracker	0.0	0.0	10-10-13

Retrieving the requests of a user is done like so:

```
-- select user requests
SELECT *
FROM requests
LEFT INNER JOIN users
ON users.id = requests.requesting_user_id
WHERE requesting_user_id = #{requesting_user_id};
```

Trackers.

The primary functionality of the application depends on trackers. Trackers contain the coordinates of a tracked user and are used to display this user on a map in the user interface. The trackers table includes a column called tracking_user_id which determines the tracking user, tracked_user_id which determines the tracked_user, latitude and longitude which contain the coordinates received from geo-tagged devices, and created_at and updated_at as time stamps.

```
-- trackers table
CREATE TABLE trackers (
```

```
  tracking_user_id integer NOT NULL,
  tracked_user_id integer NOT NULL,
  latitude float NOT NULL,
  longitude float NOT NULL,
  created_at date,
  updated_at date,
  PRIMARY KEY (tracking_user_id, tracked_user_id),
  FOREIGN KEY (tracking_user_id) REFERENCES users(id),
  FOREIGN KEY (tracked_user_id) REFERENCES users(id)
);
CREATE INDEX trackers_index
ON trackers (tracking_user_id, tracked_user_id);
```

tracking_user_id	tracked_user_id	latitude	longitude	created_at	updated_at
1	2	100.0	100.1	10-10-13	10-10-13
2	1	0.0	0.0	10-10-13	Jgfd8787g

A simple trackers view is generated for the first version of API. This is a simple virtual table for seeing trackers.

```
CREATE VIEW v1_trackers_view AS
SELECT tracking_user_id, tracked_user_id, latitude, longitude
```

```
FROM trackers;
```

Retrieving the trackers of a user is done like so:

```
-- select user trackers
SELECT *
FROM trackers
LEFT INNER JOIN users
ON users.id = trackers.tracking_user_id
WHERE tracking_user_id = #{tracking_user_id};
```

Push Apps.

Push notifications are essential in providing a better interactive experience between the user and the application. The API must know through where these notifications are to be sent. For instance a user might be using the Android version of Tacker so using Apple's push notification system and sending it to their relay servers would not make much sense. Tacker needs to know what application is being used so a push_apps table is necessary. This table contains important columns including a primary key called id, a name column determining the name of the push application, and three certificate columns that contain the path on the server where the SSL certificates are located.

```
-- push_apps
CREATE TABLE push_apps (
  id integer NOT NULL,
  name char(40) UNIQUE,
  test_certificate char(3000),
```

```
  development_certificate char(3000),
  production_certificate char(3000),
  created_at date,
  updated_at date,
  PRIMARY KEY (id),
);
CREATE INDEX push_apps_index
ON push_apps (id);
```

Push Devices.

The push_devices table is used for storing the identities and addresses of the physical devices used for sending push notifications. This could be any device capable of sending push including iOS devices and Android devices.

```
-- push_devices table
CREATE TABLE push_devices (
  id integer NOT NULL,
  push_app_id integer NOT NULL,
  updated_at date,
  PRIMARY KEY (id),
  FOREIGN KEY (push_app_id) REFERENCES push_devices(id),
);
CREATE INDEX push_notification_devices_index
ON push_notification_devices (push_notification_id, push_device_id);
```

A simple push_devices view is generated for the first version of API. This is a simple virtual table for seeing push devices

```
CREATE VIEW v1_push_devices_view AS
SELECT id, push_app_id, updated_at
FROM push_devices;
```

To determine which devices a user owns, a user_push_devices table is needed to establish that relationship in the database. This table includes the columns user_id (identifies the user), push_device_id (identifies the push_device), and created_at and updated_at as time stamps.

```
-- user_push_devices table
CREATE TABLE user_push_devices (
  user_id integer NOT NULL,
  push_device_id integer NOT NULL,
  created_at date,
  updated_at date,
  PRIMARY KEY (user_id, push_device_id),
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (push_device_id) REFERENCES push_devices(id)
);
CREATE INDEX user_push_devices_index
ON user_push_devices (user_id, push_device_id);
```

Push Notifications.

The actual push notification exists in a table called push_notification. The table includes a primary key column named id and an alert column which includes the message that is to be sent. It also has a badge column that includes the integer to be displayed on the device indicating the number of notifications. The sound attribute specifies whether or not the push notification will play a sound when it is received on the device. The custom properties attribute includes any additional properties that the notification has. For instance a notification might run a specific method in the application when opened. It also has a created_at and sent_at time stamp.

```
-- push_notification
CREATE TABLE push_notifications (
  id integer,
  alert char(140),
  badge integer,
  sound char(200),
  custom_properties char(200),
  created_at date,
 sent_at date,
  PRIMARY KEY (id),
);
CREATE INDEX push_notifications_index
ON push_notifications (id);
```

id	alert	badge	sound	custom_properties	created_at	sent_at
1	Mo wants to be your	2	true	NULL	10-10-13	10-10-13

	friend.					
2	Test accepted your friend request.	3	true	NULL	10-10-13	10-10-13
3	Mo wants to share current locations.	1	true	NULL	10-10-13	10-10-13

A simple push_notifications view is generated for the first version of API. This is a simple virtual table for seeing push notifications.

```
CREATE VIEW v1_push_notifications_view AS
SELECT id, alert, badge, sound,  custom_properties, created_at, sent_at
FROM push_notifications;
```

Tacker needs to know which device a push notification is to be sent to and that is established in the push_notification_devices table. The table has a push_notification_id that specifies the push notification and a push_device_id that specifies the push device.

```
-- push_notification_devices table
CREATE TABLE push_notification_devices (
  push_notification_id integer NOT NULL,
  push_device_id integer NOT NULL,
  PRIMARY KEY (push_notification_id, push_device_id),
  FOREIGN KEY (push_notification_id) REFERENCES push_notifications(id),
  FOREIGN KEY (push_device_id) REFERENCES push_devices(id)
);
CREATE INDEX push_notification_devices_index
```

```
ON push_notification_devices (push_notification_id, push_device_id);
```

Another simple table called user_push_notifications exists to establish the “hash many” relationship between a user and his or her push notifications. The table has a user_id column that specifies the user and a push_notification_id which specifies the push notification.

```
-- user_push_notifications table
CREATE TABLE user_push_notifications (
  user_id integer NOT NULL,
  push_notification_id integer NOT NULL,
  PRIMARY KEY (user_id, push_notification_id),
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (push_notification_id) REFERENCES push_notifications(id)
);
CREATE INDEX user_push_notifications_index
ON user_push_notifications (user_id, push_notification_id);
```

Conversations.

When users share their locations, they can then have a conversation about that share through Tacker’s built in messaging system as if it were conversing through SMS. Each tracker (shared location) has it’s own conversation record. Since there would be two trackers at max (each pointing at the other user), a conversation has two tracker_conversations and two conversations through tracker_conversations at most.

```
-- conversations table
```

```
CREATE TABLE conversations (
  id integer NOT NULL,
  created_at date,
  updated_at date,
  PRIMARY KEY (id)
);
CREATE INDEX conversations_index
ON conversations (id);
```

The tracker_conversations table defines the relationship between conversations and trackers through two columns, tracker_id (indiciating the tracker) and conversation_id (indicating the conversation).

```
-- tracker_conversations table
CREATE TABLE tracker_conversations (
  tracker_id integer NOT NULL,
  conversation_id integer NOT NULL,
  PRIMARY KEY (tracker_id, conversation_id),
  FOREIGN KEY (tracker_id) REFERENCES trackers(id),
  FOREIGN KEY (conversation_id) REFERENCES conversations(id)
);
CREATE INDEX tracker_conversations_index
ON tracker_conversations (tracker_id, conversation_id);
```

Retrieving the conversation of a tracker is done like so:

```
-- select tracker conversation
```

```
SELECT *
FROM conversations
LEFT INNER JOIN tracker_conversations
ON conversation.id = tracker_conversations.conversation_id
WHERE tracker_id = #{tracker_conversations.tracker_id}
LIMIT = 1;
```

Messages.

Conversations would be useless without messages. Messages have a primary id, a text column containing the message the user created, and a created_at column so the receiving user may see when the message was made.

```
-- messages table
CREATE TABLE messages (
  id integer NOT NULL,
  text char(140) NOT NULL,
  created_at date,
  PRIMARY KEY (id)
);
CREATE INDEX messages_index
ON messages (id);
```

The has_many relationship of a conversation and its messages is determined through the conversation_messages table. The table includes two columns including the

conversation_id (determines the conversation) and the message_id (determines the message).

```
-- conversation_messages table
CREATE TABLE conversation_messages (
  conversation_id integer NOT NULL,
  message_id integer NOT NULL,
  PRIMARY KEY (conversation_id, message_id),
  FOREIGN KEY (conversation_id) REFERENCES conversations(id),
  FOREIGN KEY (message_id) REFERENCES messages(id)
);
CREATE INDEX conversation_messages_index
ON conversation_messages (conversation_id, messages_id);
```

Just like the has_many relationship of a conversation and its messages, the has_many relationship of a user and his or her messages is determined through a table. This similar table is called the user_messages table and it contains two columns that include the user_id (determines the user) and the message_id (determines the message).

```
-- user_messages table
CREATE TABLE user_messages (
  user_id integer NOT NULL,
  message_id integer NOT NULL,
  created_at date,
  PRIMARY KEY (user_id, message_id),
  FOREIGN KEY (user_id) REFERENCES users(id),
```

```
  FOREIGN KEY (message_id) REFERENCES messages(id)
);
CREATE INDEX user_messages_index
ON user_messages (user_id, messages_id);
```

A simple messages view is generated for the first version of API. This is a simple virtual table for seeing messages.

```
CREATE VIEW v1_messages_view AS
SELECT id, text, created_at
FROM messages;
```

user_id	text	created_At
1	hey	10-10-13
1	Are you still there?	10-10-13
2	yea	10-10-13

Getting the signed in user:

Retrieving the current user (the user who is signed into the application) is done through a procedure. This is because it is a query that will be executed often in Tacker. The following SQL statement creates this procedure.

```
-- get current user
CREATE PROCEDURE get_current_user_procedure
   @login integer = #{user_id}
AS
```

```
SELECT * from users
WHERE id = login;
```

Triggers:

Since everything revolves around the user, many records are dependent on a single user record. If a user were to delete his or her account and delete the record from the table, many potential problems arise if the depending records are not destroyed too.

```
CREATE TRIGGER user_destroy_trigger
ON users
FOR DELETE
DELETE FROM friendships
WHERE friending_user_id = #{deleted_id}
OR friended_user_id = #{deleted_id}
DELETE FROM requests
WHERE requesting_user_id = #{deleted_id}
OR requested_user_id = #{deleted_id}
DELETE FROM trackers
WHERE tracking_user_id = #{deleted_id}
OR tracked_user_id = #{deleted_id}
DELETE FROM blocked_users
WHERE blocking_user_id = #{deleted_id}
OR blocked_user_id = #{deleted_id};
```

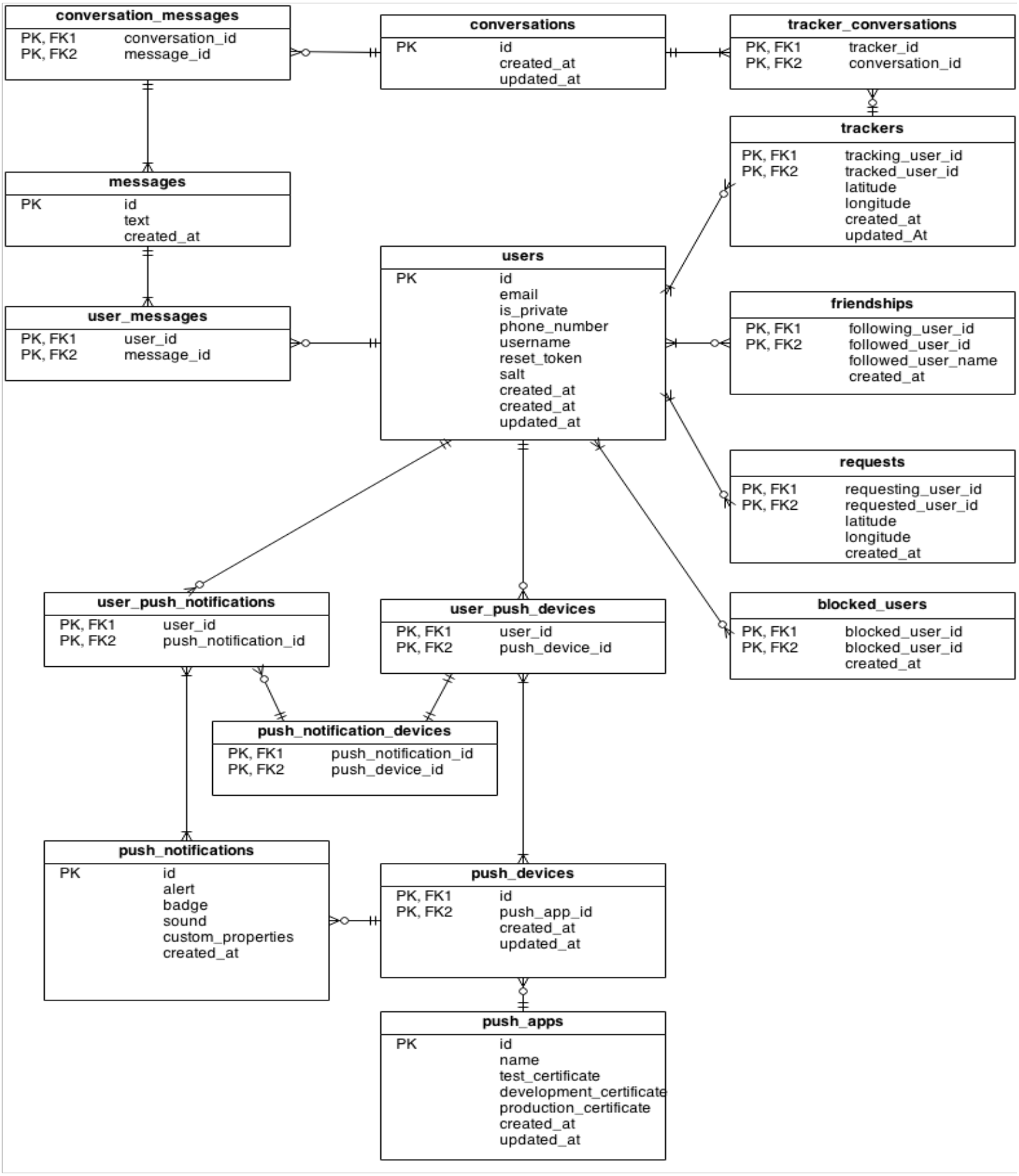
Here is a trigger that deletes all dependent friendship, requests, trackers, and blocked_user records if a user is deleted.

Known Problems:

The method in which the type of request is determined is an issue. A type column in the request table determines request types, which is problematic. A better alternative would be to create a request_types table and change the type column in the requests table to request_type_id. The request_types table would have to columns, an id column (the primary key) and name column.

Future Enhancements:

One possible enhancement would be to include self-updating tracker instead of a tracker that only takes a snapshot of a user's location. This would require a new request type so the previous issue regarding a request_types table would have to be resolved.


conversation_messages
PK, FK1 conversation_id
PK, FK2 message_id
conversations
PK id
created_at
updated_at
tracker_conversations
PK, FK1 tracker_id
PK, FK2 conversation_id
trackers
PK, FK1 tracking_user_id
PK, FK2 tracked_user_id
latitude
longitude
created_at
updated_At
messages
PK id
text
created_at
users
PK id
email
is_private
phone_number
username
reset_token
salt
created_at
created_at
updated_at
user_messages
PK, FK1 user_id
PK, FK2 message_id
friendships
PK, FK1 following_user_id
PK, FK2 followed_user_id
followed_user_name
created_at
requests
PK, FK1 requesting_user_id
PK, FK2 requested_user_id
latitude
longitude
created_at
user_push_notifications
PK, FK1 user_id
PK, FK2 push_notification_id
user_push_devices
PK, FK1 user_id
PK, FK2 push_device_id
blocked_users
PK, FK1 blocked_user_id
PK, FK2 blocked_user_id
created_at
push_notification_devices
PK, FK1 push_notification_id
PK, FK2 push_device_id
push_notifications
PK id
alert
badge
sound
custom_properties
created_at
push_devices
PK, FK1 id
PK, FK2 push_app_id
created_at
updated_at
push_apps
PK id
name
test_certificate
development_certificate
production_certificate
created_at
updated_at