A Lost and Found database for college students and faculty

Lost and Found

Database Design Document

Hannah Youssef

CMPT 308N-111
Database Management
Dr. Alan Labouseur
4 December 2017

Table of Contents

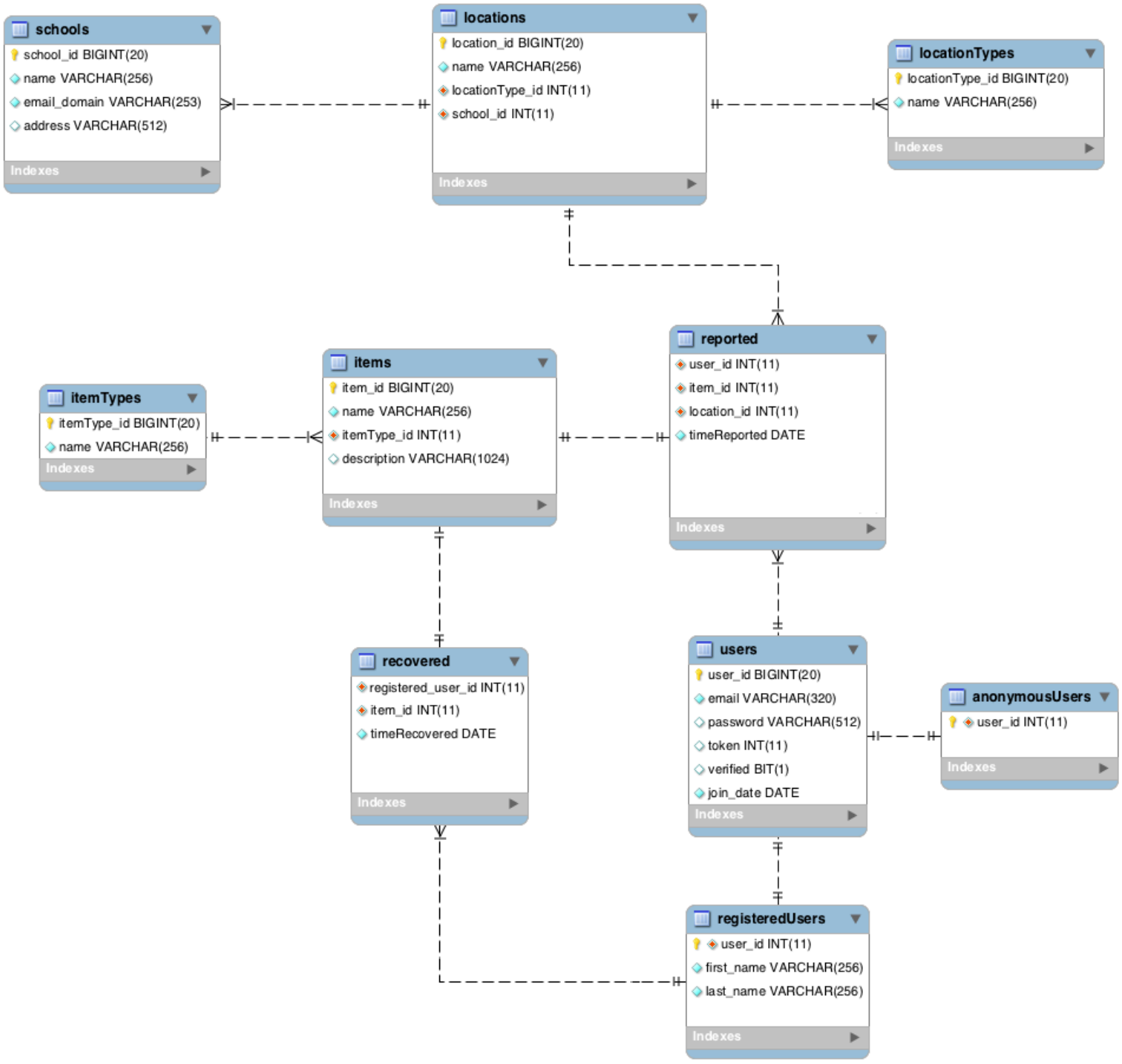
Executive Summary

This document presents an overview for the design and implementation of an item lost and found database. The system is designed with the intention of providing an easy way for college students to report and recover lost items. This document begins with an ER diagram that shows the overview of the database design. Each table's implementation and purpose are then covered in detail. The application that will use this database will use stored procedures to access data, rather than directly using complex query statements. These, as well as other useful features such as triggers and views, will be covered as well. A large system with many users connecting from many universities must make sure to provide the minimum access level to users. The details of this will be covered in the security section. Lastly, known problems and future plans will be discussed.

System Objectives and Overview

The goal of this item lost and found system is to create a place where college students can post items that they've found with the intention of getting into contact with and returning the item to the original owner. Users can post items that they've found anonymously or with their real name; this encourages people who may have stolen something and then regretted it or who simply want to keep their name out of the scenario to report items. Users with valid college emails can then search for posted items at their school. Once they find an item that they believe belongs to them, they can contact the original poster and try to arrange for the item to be returned.

ER Diagram


schools
school_id BIGINT(20)
name VARCHAR(256)
email_domain VARCHAR(253)
address VARCHAR(512)
Indexes
locations
location_id BIGINT(20)
name VARCHAR(256)
locationType_id INT(11)
school_id INT(11)
Indexes
locationTypes
locationType_id BIGINT(20)
name VARCHAR(256)
Indexes
reported
user_id INT(11)
item_id INT(11)
location_id INT(11)
timeReported DATE
Indexes
items
item_id BIGINT(20)
name VARCHAR(256)
itemType_id INT(11)
description VARCHAR(1024)
Indexes
itemTypes
itemType_id BIGINT(20)
name VARCHAR(256)
Indexes
recovered
registered_user_id INT(11)
item_id INT(11)
timeRecovered DATE
Indexes
users
user_id BIGINT(20)
email VARCHAR(320)
password VARCHAR(512)
token INT(11)
verified BIT(1)
join_date DATE
Indexes
anonymousUsers
user_id INT(11)
Indexes
registeredUsers
user_id INT(11)
first_name VARCHAR(256)
last_name VARCHAR(256)
Indexes

Functional Dependencies

users
user_id -> email

user_id -> password

user_id -> token

user_id -> verified

user_id -> join_date

user_id -> first_name

user_id -> last_name

schools
school_id -> name

school_id -> email_domain

school_id -> address

location_types
locationType_id -> name

locations
locations -> name

locations -> locationType_id

locations -> school_id

itemTypes
itemType_id -> name

items
item_id -> name

item_id -> itemType_id

item_id -> description

reported
item_id -> user_id

item_id -> timeReported

item_id -> location_id

recovered
item_id -> registered_user_id

item_id -> timeRecovered

Entities

Users

The users entity stores information about a user of the lost and found system. It includes the user_id, email, password, token (for initial account verification), join_date, and a true or false column called “verified” which is set to true once the user has successfully verified their account after initial creation. Most importantly, the users entity is the base entity for the anonymousUsers and registeredUsers entities.

```
CREATE TABLE users (
user_id SERIAL PRIMARY KEY,
email varchar(320) NOT NULL,
password varchar(512),
token int,
verified boolean default '0',
join_date date NOT NULL DEFAULT CURRENT_DATE;
);
```

Sample Data:

user_id	email	password	token	join_date	verified
1	Hannah.Youssef1@marist.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
2	Alan.Labouseur@marist.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
3	stevejobs@mit.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
4	stevewozniak@harvard.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
5	Zimri.Mayfield@stanford.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
6	beyonceknowles@bc.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
7	jayz@bu.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
8	donaldtrump@yale.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
9	Justin.Bieber@nyu.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
10	John.Smith@bc.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
11	Frankie.Fox@marist.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
12	billgates@harvard.edu	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
13	appleluvr33@gmail.com	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
14	test.post@please.ignore.com	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
15	admin111111111111@gmail.com	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
16	AlanIsTheBestProfessorEver@lol.com	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
17	email@yahoo.com	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X
18	hannah@aol.com	5f4dcc3b5aa765d61d8327deb882cf99	NULL	2017-12-04	X

Anonymous Users

The anonymousUsers entity stores information for users of the lost and found system that chooses to remain anonymous. It inherits all of the attributes from the users base entity. No further attributes are added to the anonymousUsers entity. Anonymous users can post items but cannot recover items. Additionally, anonymous users do not need a college email. This allows, for example, a bar tender to report a lost item despite not attending the college where the item likely originated.

```
CREATE TABLE anonymousUsers (
     user_id int REFERENCES users(user_id) PRIMARY KEY
);
```

Sample Data:

user_id
7
8
9
14
15
2

Registered Users

The registeredUsers entity stores information about users who choose to not be anonymous – in other words, fully registered users with all attributes stored in the system. Like the anonymousUsers entity, all attributes of the users base table are inherited by registeredUsers. However, registeredUsers also has additional attributes. These include first_name and last_name that will be publicly visible to other users when this user posts an item or recovers an item. Registered users must use a college email.

```
CREATE TABLE registeredUsers (
     user_id int REFERENCES users(user_id) PRIMARY KEY,
     first_name varchar(256) NOT NULL CONSTRAINT non-empty
CHECK(length(first_name)>0),
     last_name varchar(256) NOT NULL CONSTRAINT non-empty
CHECK(length(last_name)>0)
);
```

Sample Data:

user_id	first_name	last_name
1	Hannah	Youssef
2	Alan	Labouseur
3	Steve	Jobs
4	Steve	Wozniak
5	Zimri	Mayfield
6	Beyonce	Knowles
7	Jay	Z
8	Donald	Trump
9	Justin	Bieber
10	John	Smith
11	Frankie	Fox
12	Bill	Gates

Schools

The schools entity stores information about schools (colleges and universities) that are current part of the lost and found system. Basic information about each school is stored, including name, email_domain, and address. Special attention should be brought to email_domain. The purpose of this attribute is to match students who sign up with their campus email address to a specific school. For example, if I sign up with Hannah.Youssef1@marist.edu, then I can only view and post lost items that are associated with the school with email_domain 'marist.edu.' This prevents someone from UCONN accessing Marist's lost and found service, and vice versa.

```
CREATE TABLE schools (
      school_id SERIAL PRIMARY KEY,
      name varchar(256),
      email_domain varchar(253) NOT NULL,
      address varchar(512)
);
```

Sample Data:

school_id	name	email_domain	address
1	Marist College	marist.edu	Poughkeepsie NY 12601
2	Boston College	bc.edu	Chesnut Hill, MA 02467
3	Boston University	bu.edu	Boston MA 02215
4	Stanford University	stanford.edu	Stanford CA 94305
5	MIT	mit.edu	Cambridge MA 02139
6	UCONN	uconn.edu	Storrs CT 06269
7	UPENN	upenn.edu	Philadelphia PA 19104
8	Harvard	harvard.edu	Cambridge MA 02138
9	Yale	yale.edu	New Haven CT 06520
10	NYU	nyu.edu	New York NY 10003

Location Types

The locationTypes entity stores the locationType_id and name for each type of location.

These values are references exclusively by the locations entity.

```
CREATE TABLE locationTypes (
     locationType_id SERIAL PRIMARY KEY,
     name varchar(256) NOT NULL CONSTRAINT non-empty CHECK(length(name)>0)
);
```

Sample Data:

locationtype_id	name
1	Dining Hall/Area
2	Academic Building
3	Outdoors
4	Taxi
5	Bus
6	Residence Hall
7	Gym
8	Library
9	Bar / Club
10	Party

Locations

The locations entity stores the various locations that can be associated with a school. Locations are used to provide a more precise origin when posting where a lost item was originally discovered. It can also be used to narrow the search results when searching for a lost item. Locations can only belong to one school, but one school can have many locations, both on and off campus, associated with it. Each location also has a location type, referenced from the locationTypes entity table.

```
CREATE TABLE locations (
      location_id SERIAL PRIMARY KEY,
      name varchar(256) NOT NULL CONSTRAINT non-empty CHECK(length(name)>0),
      locationType_id int REFERENCES locationTypes(locationType_id),
      school_id int REFERENCES schools(school_id)
);
```

Sample Data:

location_id	name	locationtype_id	school_id
1	Cabaret	1	1
2	Dining Hall	1	1
3	Donnelly Hall	2	1
4	Engineering Building	8	6
5	The Library	2	6
6	BC Gym	7	2
7	Upper Housing A	6	2
8	Crazy party on 4th street	10	10
9	Near campus entrance	3	10
10	BU Bus 4	5	3
11	MIT Famous Bar	9	5

Item Types

The itemTypes entity stores the itemType_id and name for each type of item. These values are references exclusively by the items entity.

```
CREATE TABLE itemTypes (
     itemType_id SERIAL PRIMARY KEY,
     name varchar(256) NOT NULL CONSTRAINT non-empty CHECK(length(name)>0)
);
```

Sample Data:

itemtype_id	name
1	Phone
2	Laptop
3	MP3 Player
4	Portable Video Game System
5	Electronic Accessory
6	Jewelry
7	Book
8	Notebook
9	Credit/Debit Card
10	ID
11	Clothing
12	Wallet/Purse
13	Bag

Items

The items entity stores the attributes for an item that was posted by a user. Attributes that will help the original owner find the item are stored. These attributes include the item type, which is references from the itemTypes entity table, the name of the item, and a description of the item.

```
CREATE TABLE items (
      item_id SERIAL PRIMARY KEY,
      name varchar(256) NOT NULL CONSTRAINT non-empty CHECK(length(name)>0),
      itemType_id int REFERENCES itemTypes(itemType_id),
      description varchar(1024)
);
```

Sample Data:

item_id	name	itemtype_id	description
1	iPhone 5S	1	White. Crack on front.
2	Retina MacBook Pro	2	Blue case. Login screen says John with red striped background
3	iPod Shuffle	3	Green iPod Shuffle.
4	Nintendo 3DS	4	Black. Broken hinge. Pokemon Rainbow is in it.
5	Bose Headphones	5	Black and silver.
6	Earrings	6	Hoop silver earrings
7	Database Systems Book	7	Faded cover.
8	Database Systems Notebook	8	Red cover. Wide ruled paper. Many doodles, few notes.
9	Visa Card	9	Name on card reads Alan Labouseur.
10	Marist ID	10	CWID 200-80-908 and name Hannah Youssef
11	Plaid shirt	11	Long sleeves. Stain on back.
12	Brown leather wallet	12	Contains an ID and some one dollar bills.
13	Very Expensive Name Brand Bag	13	LV logo
14	Philosophy Notebook	8	Red pen was used throughout.
15	T shirt	11	Long sleeves. Black with brown stripes.
16	Black leather wallet	12	Contains some one hundred dollar bills.
17	Coach Bag	13	Brown with black Coach logos

Reported

The reported entity stores reports of lost items. Each report has a user_id associated with it (anonymous or registered) and a unique item_id (since the same item can't be part of more than one lost item report). If the same item were lost in the future, it would be stored as a completely new item. Additionally, the location where the item was found (referenced from the locations entity) is stored, as well as when the item was reported to the system.

```
CREATE TABLE reported (
     user_id int REFERENCES users(user_id),
     item_id int REFERENCES items(item_id) UNIQUE,
     location_id int REFERENCES locations(location_id),
     timeReported date NOT NULL DEFAULT CURRENT_DATE
);
```

Sample Data:

user_id	item_id	location_id	timereported
1	1	3	2017-12-04
2	2	1	2017-12-04
3	10	6	2017-12-04
4	8	6	2017-12-04
5	4	4	2017-12-04
6	5	5	2017-12-04
7	12	11	2017-12-04
8	3	5	2017-12-04
9	6	8	2017-12-04
10	7	9	2017-12-04
11	9	10	2017-12-04
12	13	2	2017-12-04

Recovered

The recovered entity stores recovery records of lost items. Each recovery has a registered_user_id associated with it (from the registeredUsers entity since only registered users can recover items) and a unique item_id (since the same item can't be recovered twice). If the same item were recovered in the future, it would be stored as a completely new item by the reporter and would therefore carry that unique item_id throughout the recovery process. Lastly, the time that the item was recovered is stored in timeRecovered.

```
CREATE TABLE recovered (
     registered_user_id int REFERENCES registeredUsers(user_id),
     item_id int REFERENCES items(item_id) UNIQUE,
     timeRecovered date NOT NULL DEFAULT CURRENT_DATE
);
```

Sample Data:

user_id	item_id	timereported
1	3	2017-12-02
1	6	2017-12-03
1	8	2017-12-04
1	11	2017-12-02
3	14	2017-12-01
3	15	2017-12-03
4	16	2017-12-01
4	17	2017-12-02
4	18	2017-12-03
12	19	2017-12-01

Stored Procedures

This section demonstrates stored procedures that can be used to query the database in a much safer way than having an application directly query.

All currently lost items for a particular school reported by an anonymous user

Usage: SELECT anonymouslyReportedItemsForSchool('Marist College')

```
CREATE FUNCTION anonymouslyReportedItemsForSchool(schoolName VARCHAR(256))
RETURNS TABLE(itemName varchar(256), itemType varchar(256), reportedLocationName
varchar(256)) AS $$
BEGIN
     RETURN QUERY SELECT items.name, itemTypes.name, locations.name
                 FROM items, itemTypes, locations, reported,  anonymousUsers, schools
                 WHERE itemTypes.itemType_id = items.itemType_id
                 AND items.item_id = reported.item_id
                 AND reported.location_id = locations.location_id
                 AND reported.user_id = anonymousUsers.user_id
                 AND locations.school_id = schools.school_id
                 AND schools.name = schoolName;
END;
$$ LANGUAGE PLPGSQL;
```

Names of users who have reported and returned more than one item at a school

Usage: `SELECT goodReputationUsers()`

```
CREATE FUNCTION goodReputationUsers()
RETURNS TABLE(firstName VARCHAR(256),  lastName VARCHAR(256)) AS $$
BEGIN
     RETURN QUERY SELECT registeredUsers.first_name, registeredUsers.last_name
          FROM registeredUsers, (SELECT count(*), reported.user_id
                                 FROM reported, recovered
                                 WHERE reported.item_id = recovered.item_id
                                 AND reported.user_id IN (SELECT
                                     registeredUsers.user_id
                                                          FROM registeredUsers)
                                                          GROUP BY reported.user_id
                                                          HAVING count(*) > 1
                                                          ) AS goodUsers
          WHERE goodUsers.user_id = registeredUsers.user_id;
END;
$$ LANGUAGE PLPGSQL;
```

Get currently lost items

Usage: `SELECT currentlyLostItems()`

```
CREATE FUNCTION currentlyLostItems ()
RETURNS TABLE(id INT, name VARCHAR(256)) AS $$
BEGIN
     RETURN QUERY SELECT items.item_id,
                         items.name
          FROM items, (SELECT reported.item_id reported_item_id, recovered.item_id
     recovered_item_id
                          FROM reported LEFT OUTER JOIN recovered
                          ON recovered.item_id = reported.item_id
                          WHERE recovered.item_id IS NULL) AS lostItems
                          WHERE lostItems.reported_item_id = items.item_id;
END;
$$ LANGUAGE PLPGSQL;
```

Triggers

A trigger and function that automatically deletes a user from the base user table when the user is removed from anonymousUsers.

```
CREATE FUNCTION delete_anonymous_user() RETURNS TRIGGER AS $_$
     BEGIN
          DELETE FROM users WHERE users.user_id = OLD.user_id;
          RETURN OLD;
     END $_$ LANGUAGE PLPGSQL;
```

```
CREATE TRIGGER remove_anonoymous_user AFTER DELETE ON
anonymousUsers FOR EACH ROW EXECUTE PROCEDURE
delete_anonymous_user();
```

This can easily be tested with the following query:

```
DELETE FROM anonymousUsers WHERE anonymousUsers.user_id = 8;
```

Views

CurrentReporters

This view displays information about current reporters of lost items.

```
CREATE VIEW CurrentReporters AS
     SELECT lostItems.name AS Item_Name,
             users.email AS Reporter_Contact
             FROM currentlyLostItems() AS lostItems, users, reported
             WHERE lostItems.id = reported.item_id
             AND reported.user_id = users.user_id;
```

OldUnverifiedUsers

This view displays users who are unverified and have been unverified for over 1 month.

```
CREATE VIEW OldUnverifiedUsers AS
     SELECT users.*
     FROM users
     WHERE users.verified = '0'
     AND users.join_date < now() - INTERVAL '1 month';
```

MostCommonLostLocation

This view displays the information about the most common locations.

```
CREATE VIEW MostCommonLostLocation AS
     SELECT count(*), locations.name
     FROM reported, locations
     WHERE reported.location_id = locations.location_id
     GROUP BY locations.name
     ORDER BY count(*) DESC;
```

Security

In an attempt to create a secure environment for both current plans and future plans, different parts of the database and different interactions with the database are limited through the use of multiple users and access controls.

1. Database Administrator: The DBA has access to the database to allow him or her to do read, write, and modify actions like SELECT, INSERT, DELETE, and UPDATE. For example, the DBA can insert a new user, or update the token field in an existing user.

```
CREATE ROLE admin

GRANT SELECT, INSERT, UPDATE, DELETE
ON ALL TABLES IN SCHEMA PUBLIC
TO admin;
```

2. The second user would be a user in the company that would simply like to generate reports. This user has read only access to the database. This user is not expected to be tech savvy and is therefore given extremely limited privileges.

```
CREATE ROLE viewer

GRANT SELECT
ON ALL TABLES IN SCHEMA PUBLIC
TO viewer;
```

Reports

Locations in Schools

```
SELECT schools.name, locations.name
FROM schools, locations
WHERE schools.school_id = locations.school_id;
```

All Lost Items Current and Past

```
SELECT schools.name, items.name
FROM schools, items, locations, reported
WHERE reported.item_id = items.item_id
AND reported.location_id = locations.location_id
AND locations.school_id = schools.school_id;
```

Most Common Lost Item Type

```
SELECT count(*), itemTypes.name
FROM itemTypes, items, reported
WHERE reported.item_id = items.item_id
AND items.itemType_id = itemTypes.itemType_id
GROUP BY itemTypes.name
ORDER BY count(*) DESC;
```

Known Issues

1. Currently, the system does not have a messaging feature built in. This means that, while connecting with a registered user is trivial since their email and name is public, connecting to an anonymous user would require an email alias to be generated at runtime to allow another user to contact them without divulging their identity. The mail would then be sent through a form on the site. Most mailers allow for a queue, so messages, rather than being stored in a table in the database until they're sent out, would be stored in the mailer's own queue. This kind of setup would make it difficult to see what messages a user has sent in the past. In the future, it would be deal if all messaging was done via the site, rather than going through email.

2. In a crowded city like Boston, multiple schools might claim that the same off campus location is associated with their school. This might seem like a problem at first, but I believe it's best for the system to behave this way. This allows schools to claim association with whatever place they want, and lost items for a particular school will only show up for users at that school, even if they were lost in a "shared" area. This does, however, pose a problem for anonymous users who want to report a lost item they found at a location. The user would have to post to all potential schools that would claim association with that establishment for the posting to be effective.

Future Plans

1. As one might assume from the “Known Issues” section, the next major plan would be to implement an in-house messaging system to get users to communicate in order to get their lost possession back. This would remove all of the issues that come with using email as a middleman (though it is extremely convenient).

2. The next major plan is to have a reward system. Users who successfully return items could get discounts at local businesses or some other kind of reward. The goal is to encourage people to return the item rather than keep it for themselves. One would then naturally question whether a discount at a local business is enough incentive to return a $2,500 top of the line Retina MacBook Pro.

3. Another possibility is to make it some kind of competition among schools and maybe individual users. People can get certain badges like “Most likely to lose something” for users who frequently lose things. Another possible badge would be “Most trustworthy user” for the user who successfully returns the most items to original owners. Of course, one would need to figure out a way to verify that items are actually being returned and that it’s not just two friends trying to get points by making up fake lost and found scenarios.