Marist College's James A. Cannavino Library Tech Help Support Database Proposal

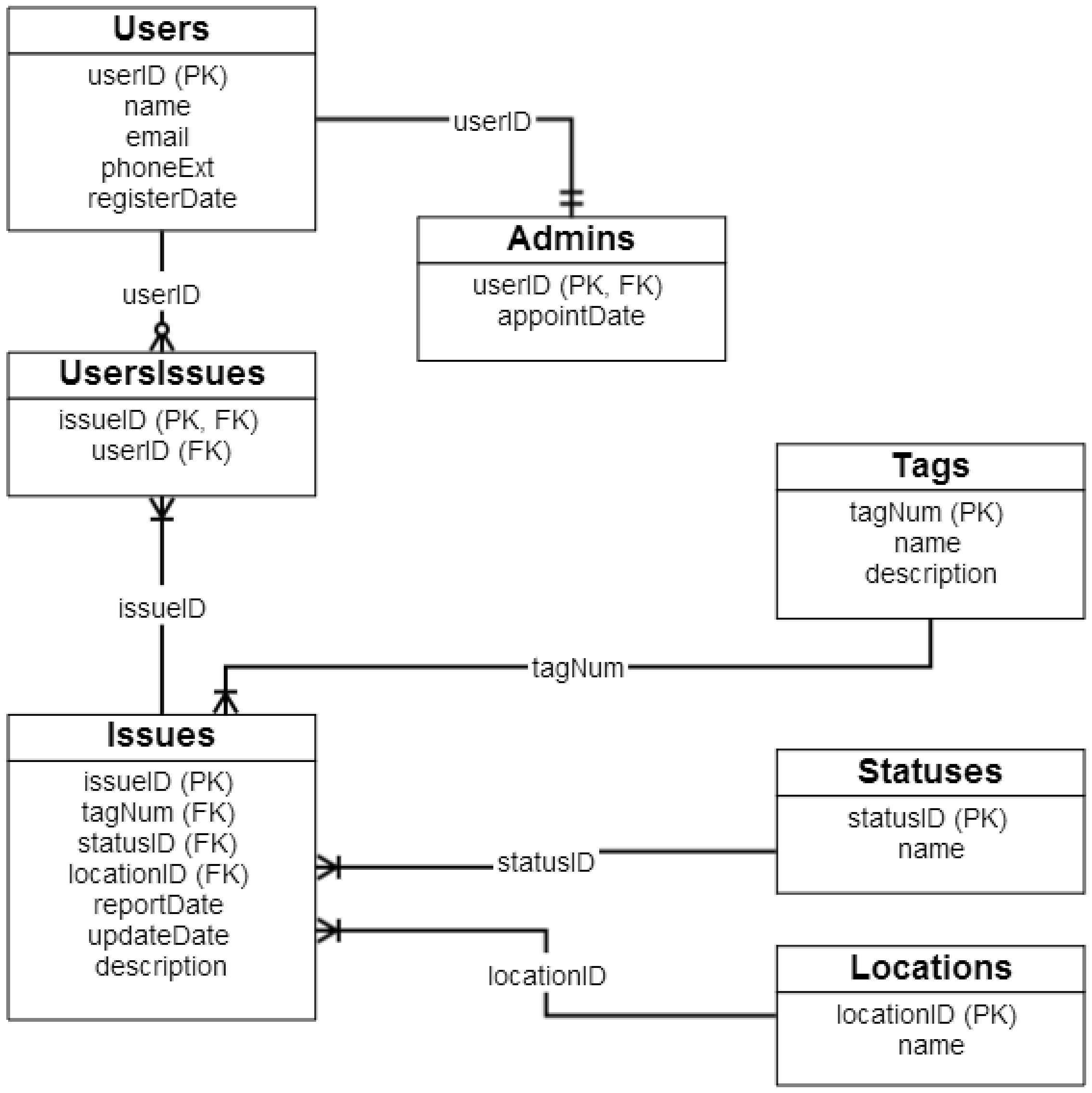
Weon Yuan
CMPT 308 – Database Management
11/29/13

Table of Contents

Executive Summary

This document serves as a database proposal for the Marist College's James A. Cannavino Library in Poughkeepsie, NY. The document will provide the design outlines of the database and its implementations, which will serve as a convenient place to store all tech-related issues pertaining to the James A. Cannavino Library. To start, an entity-relationship (ER) diagram will be shown to illustrate the proposed database, followed by the details of each table that are represented in the ER diagram. Every table details will provide a short explanation to the intentions of creating the table, the SQL statement used to create the table, and some sample data that can be represented in a real-life situation. By doing this, it provides a better idea of how the database will be implemented and will be ensured that it meets the requirements of the James A. Cannavino Library. In order to provide convenience, several stored procedures have been created in the database that allows complicated queries, or rather long lines of code, to be run usually through one line of code. Like with many good databases, data integrity is important. As a result, triggers have been programmed to the database, which will be explained further on. Reports will also be shown to demonstrate how meaningful data be by providing sample queries. The implementation will be covered more in depth towards the end of this proposal, and improvements and enhancements will be covered in details as well.

ER diagram


Users
userID (PK)
name
email
phoneExt
registerDate
userID
Admins
userID (PK, FK)
appointDate
userID
UsersIssues
issueID (PK, FK)
userID (FK)
issueID
Tags
tagNum (PK)
name
description
tagNum
Issues
issueID (PK)
tagNum (FK)
statusID (FK)
locationID (FK)
reportDate
updateDate
description
statusID
Statuses
statusID (PK)
name
locationID
Locations
locationID (PK)
name

Tables

Users table

This table contains a list of users in the database, containing their basic information such as name, email, and phone extension number. There is a constraint that checks for a valid marist.edu email address (since this is designed for the Marist College's library) and another constraint that checks for a valid phone extension.

```
create table if not exists Users (
  userID       serial,
  name         varchar(150) not null,
  email        varchar(256) not null,
  phoneExt     char(5)      not null,
  registerDate timestamp    not null,
 constraint validEmail check(email    like '%@marist.edu'),
 constraint validExt   check(phoneExt like 'x%'),
 primary key (userID)
);
```

Functional dependencies

<u>userID</u> → name, email, phoneExt, registerDate

Sample data

userID	name	email	phoneExt	registerDate
1	Weon Yuan	weon.yuan1@marist.edu	x2017	2013-12-01 17:30:20.109
2	John Ansley	john.ansley@marist.edu	x5217	2013-12-01 17:30:20.109
3	Lori Burns	lori.burns1@marist.edu	x3292	2013-12-01 17:30:20.109
4	Margaret Cirillo	margaret.cirillo1@marist.edu	x3292	2013-12-01 17:30:20.109
5	Elizabeth Clarke	elizabeth.clarke2@marist.edu	x2733	2013-12-01 17:30:20.109
6	Marta Cwik	marta.cwik@marist.edu	x2408	2013-12-01 17:30:20.109
7	Nancy Decker	nancy.decker@marist.edu	x3199	2013-12-01 17:30:20.109

Tables

Admins table

This table contains the list of users that are assigned as active admins.

```
create table if not exists Admins (
  userID      int       not null,
  appointDate timestamp not null,
 primary key (userID),
 foreign key (userID) references Users(userID)
);
```

Functional dependencies

userID → appointDate

Sample data

userID	appointDate
1	2013-12-01 17:30:20.109
8	2013-12-01 17:30:20.109

Tables

Tags table

This table contains a list of items that have been assigned a 5-digit tag number. Usually, the tag number is assigned by the Marist College IT department to computers, computer accessories, and electronics devices, but when certain items are not applicable to those categories, a generic tag number is provided – such as 99999 for web services. Since, the tag number is unique it serves as the primary key for this table rather than using an auto-incrementing number.

```
create table if not exists Tags (
  tagNum      char(5)     not null  unique,
  name        varchar(50) not null,
  description text        not null,
 primary key (tagNum)
);
```

Functional dependencies

<u>tagNum</u> → name, description

Sample data

tagNum	name	description
20213	EPSON Perfection V700	Scanner
23359	Lenovo Thinkpad T440	Computer notebook
44241	HP Laserjet 1100	Laser printer
45020	Lenovo Thinkpad T530	Computer notebook; used as a rental
67802	Logitech C320 Webcam	Webcam
99999	Web services	Any web services or websites hosted by the Marist Library servers
00000	Misc.	Item(s) that has no tag number

Tables

Statuses table

This table contains the list of possible statuses that can be assigned to reported issues.

```
create table if not exists Statuses (
  statusID serial,
  name     varchar(50) not null,
 primary key (statusID)
);
```

Functional dependencies

statusID → name

Sample data

statusID	name
1	Open
2	Closed
3	Pending
4	Forwarded to Help Desk

Tables

Locations table

This contains a list of possible locations, used to identify where the issue has taken place.

```
create table if not exists Locations (
  locationID serial,
  name       varchar(50) not null,
 primary key (locationID)
);
```

Functional dependencies

locationID → name

Sample data

locationID	name
1	1st floor
2	2nd floor
3	3rd floor
4	Online

Tables

Issues table

This table contains the list of technical issues or problems that have been reported by users.

```
create table if not exists Issues (
  issueID     serial,
  tagNum      char(5)   not null,
  statusID    int       not null,
  locationID  int       not null,
  reportDate  timestamp not null,
  updateDate  timestamp not null,
  description text      not null,
 primary key (issueID),
 foreign key (tagNum)     references Tags(tagNum),
 foreign key (statusID)   references Statuses(statusID),
 foreign key (locationID) references Locations(locationID)
);
```

Functional dependencies

issueID → tagNum, statusID, locationID, reportDate, updateDate, description

Sample data

issueID	tagNum	statusID	locationID	reportDate	updateDate	description
1	15482	2	1	2013-11-25 14:31:54	2013-11-25 16:32:43	Computer does not turn on.
2	44241	3	2	2013-11-28 10:19:16	2013-11-30 18:02:22	Printer leaves streaking on papers.
3	15482	4	1	2013-11-29 11:14:12	2013-12-01 19:25:32	Computer fails to recognize any library printers; will not install the printer drivers.
4	99999	1	4	2013-11-30 12:05:03	2013-11-30 12:05:03	I cannot login with my Marist account!
5	00000	2	3	2013-12-01 20:17:11	2013-12-01 21:12:00	Cannot boot; Rented Lenovo T440 laptop.

Tables

UserIssues table

This table links the issues from the Issues table to the users who have made the report. Compared to the other tables in the database, this table takes no functional dependencies (since creating a dependency between Users table and Issues table will cause a M:M relationship).

```
create table if not exists UsersIssues (
  issueID int not null,
  userID  int not null,
 primary key (issueID),
 foreign key (issueID) references Issues(issueID),
 foreign key (userID)  references Users(userID)
);
```

Functional dependencies

None

Sample data

issueID	userID
1	3
2	1
3	8
4	2
5	8

Views

OngoingIssues

This view displays results of all issues that have not been closed yet. Any issue with the “Open,” “In Process,” or “Forwarded to Help Desk” status will be shown on the query result.

```
create view OngoingIssues as
select i.updateDate as “Date”,
       i.tagNum as "Tag Number",
       t.name as "Tag Name",
       s.name as "Status",
       i.description as "Issue Description"
from Issues i, Tags t, Statuses s, Locations l
where i.tagNum = t.tagNum
  and i.statusID = s.statusID
  and i.locationID = l.locationID
  and i.statusID != 2;
```

ClosedIssues

This view displays results of all issues that have been closed, as indicated by its status.

```
create view ClosedIssues as
select i.updateDate as “Date”,
       i.tagNum as "Tag Number",
       t.name as "Tag Name",
       s.name as "Status",
       i.description as "Issue Description"
from Issues i, Tags t, Statuses s, Locations l
where i.tagNum = t.tagNum
  and i.statusID = s.statusID
  and i.locationID = l.locationID
and i.statusID = 2;
```

Stored Procedures

IssuesList

It is essential into accessing the list of issues, especially for admins so they can keep a better track of ongoing issues and ensure they are being handled in a timely manner. As a solution, admins can use the IssuesList view to quickly gather a list of issues in a table.

```
create or replace function IssuesList(status varchar(50))
returns table(tagNum char(5),
         tagName varchar(50),
         statusName varchar(50),
         updateDate timestamp,
         description text) as $$
begin
 return query
  select i.tagNum as "Tag Number",
         t.name as "Tag Name",
         s.name as "Status",
         i.updateDate as "Date",
         i.description as "Issue Description"
  from Issues i, Tags t, Statuses s, Locations l
  where i.tagNum = t.tagNum
    and i.statusID = s.statusID
    and i.locationID = l.locationID
    and s.name = status;
end;
$$ language plpgsql;


select IssuesList('Forwarded to Help Desk');
```

Stored Procedures

IssuesCount

Another stored procedure that is implemented to the database, is a counter function that shows the number of issues that have been reported to the database – categorized by the different statuses. This can be important, especially when admins can keep track of issues that have been successfully resolved and issues that are still ongoing. For instance, an admin can use this procedure to look up the number of issues that are still ongoing, then use the IssuesList procedure (from the last page) that provides a more detailed description of the issues.

```
create or replace function IssuesCount()
returns table(statusName varchar(50),
           amount bigint) as $$
begin
 return query
  select s.name as "Status",
         count(*) as "Reported Amount"
  from Issues i, Statuses s
  where i.statusID = s.statusID
  group by s.name;
end;
$$ language plpgsql;


select IssuesCount();
```

Reports

Show all admins

This query will show all active admins on the database.

```
select u.name, u.email, u.phoneExt
from users u
where u.userID in (
  select userID
  from userAdmins
);
```

Show the issues reported by an user

This query will show the issues a certain user has reported.

```
select i.updateDate, u.name, s.name, l.name, i.description
from UsersIssues ui inner join Users u  on ui.userID = u.userID
               inner join Issues i on ui.issueID = i.issueID
                    inner join Statuses s on i.statusID = s.statusID
                    inner join Locations l on i.locationID = l.locationID
where ui.userID = /* userID */ ;
```

Reports

Show the reported frequency of an item

This query will show the number of times this item has been reported having tech issues.

```
select t.tagNum, t.name, count(i.*)
from Tags t left outer join Issues i on t.tagNum = i.tagNum
where t.tagNum = /* tagNum */
group by t.name, t.tagNum;
```

Show the frequency of user submissions by those who have submitted at least once

This query will show the amount every user in the database has submitted a report at least once.

```
select distinct u.name, count(i.*)
from Users u, UsersIssues ui, Issues i
where ui.issueID = i.issueID
  and ui.userID = u.userID
group by u.name;
```

Security

normalUser role

A normalUser is a person (librarian or library-related staff) who has been granted permission to post any tech-related issues to the database. The normalUser can represent someone that works for the library, who can identify any valid technical problems of an item and is looking forward to reporting this issue. This role should have minimal knowledge of the base tables, least to say the role should not be able to modify or alter any data in the base tables – with the exception of updating their own personal information (e.g., phone extension). However, the user should be permitted to view all the reported issues in the database, and anything that is dependent on the Issues table.

```
create role normalUser;

revoke all privileges on Users from normalUser;
revoke all privileges on Admins from normalUser;
revoke all privileges on Tags from normalUser;
revoke all privileges on Statuses from normalUser;
revoke all privileges on Locations from normalUser;
revoke all privileges on Issues from normalUser;
revoke all privileges on UsersIssues from normalUser;

grant select, update on Users to normalUser;
grant select on Tags to normalUser;
grant select on Statuses to normalUser;
grant select on Locations to normalUser;
grant select, insert on Issues to normalUser;
grant select on UsersIssues to normalUser;
```

Security

Admin role

The Admin role is responsible for maintaining the database, and ensuring that every track is being processed and resolved in a timely manner. They are practically given privileges to select, insert, and update all tables in the database, including the privilege to delete other users from the Admins table – in case if any admins are no longer active or are not fulfilling their rightful duties in maintaining the database.

```
create role admin;

revoke all privileges on Users from admin;
revoke all privileges on Admins from admin;
revoke all privileges on Tags from admin;
revoke all privileges on Statuses from admin;
revoke all privileges on Locations from admin;
revoke all privileges on Issues from admin;
revoke all privileges on UsersIssues from admin;

grant select, insert, update on Users to admin;
grant select, insert, update, delete on Admins to admin;
grant select, insert, update on Tags to admin;
grant select, insert, update on Statuses to admin;
grant select, insert, update on Locations to admin;
grant select, insert, update on Issues to admin;
grant select, insert, update on UsersIssues to admin;
```

Implementation Notes

The majority of the tables in this databases are connected (or dependent) with another table, and as a result, the tables have to be created in the same sequence the tables appear in this document.

Known Issues

- The current design restricts an issue being handled by more than one admin
    - However, by having one admin it will maintain simplicity and consistency within the issue
        - Compared to having multiple admins handle an issue
        - This will also reduce the chance of having conflicted or redundant data
- What happens if the item, identified by a tag number, has a history of being moved to different locations?
    - This can be an issue if the item is moved/transferred from one room to a different room of the same floor
        - The Locations table should provide more choices to select rooms in the Library, instead of floors
- May be difficult to track down an item that does not have a tag number identification
    - Its issues history may be conjoined with other items that are not identified by a tag number
- Triggers are not included in this database proposal
    - The goal of this design is to be simplistic and consistent as possible
    - Adding triggers will make the design more complicated

Future Enhancements

- Reconfigure the Locations table to provide a specific selection of rooms and areas in the Library
    - Instead of providing a general selection to users, such as floors
- Closed issues should be transferred into a table containing a history of issues
- Automatically report Marist College’s Help Desk with any issues that have the status, “Forwarded to Help Desk”
    - Perhaps use a trigger to perform this task