#### Nashville Music Map

**Group 11**

**Shamita Nagalla**

**Shaon Borosha**

**Geoff Ren**

#### SECTION 1. Overview

While big music events that occur in Nashville are highly publicized and take place at well known venues, there are numerous smaller shows that many people do not hear about. These include live music at restaurants, pop up shows, concerts at smaller venues, etc. We will create a web application that acts as a calendar to display all of these music events and list their corresponding details. Music events are made up of Venues that events occur at and Performers whom perform at an event. Each of these can be rated by users so that people will have a general idea of how good an event will be.

Functions*:*

* Venues will be described by name, location, description, photos, and user ratings
* Performers will be described by name, description, photos and user ratings
* Events will be described by date, genre, cost, performer, and venue
* Users, performers, and venues will be able to have a conversation about each event
* Users can rate performers and venues once as well as judge other ratings with an upvote system
* Performers and venues will be able to respond to ratings both positive & negative
* Users will also be able to bookmark events and see a history of past attended events

To support all this the database will store a variety of information. To support venue and performer descriptions, venue and performer tables will store all the needed information. An event table will also store all necessary event information as well as contain a reference to the performer and venue the event will be taking place at.

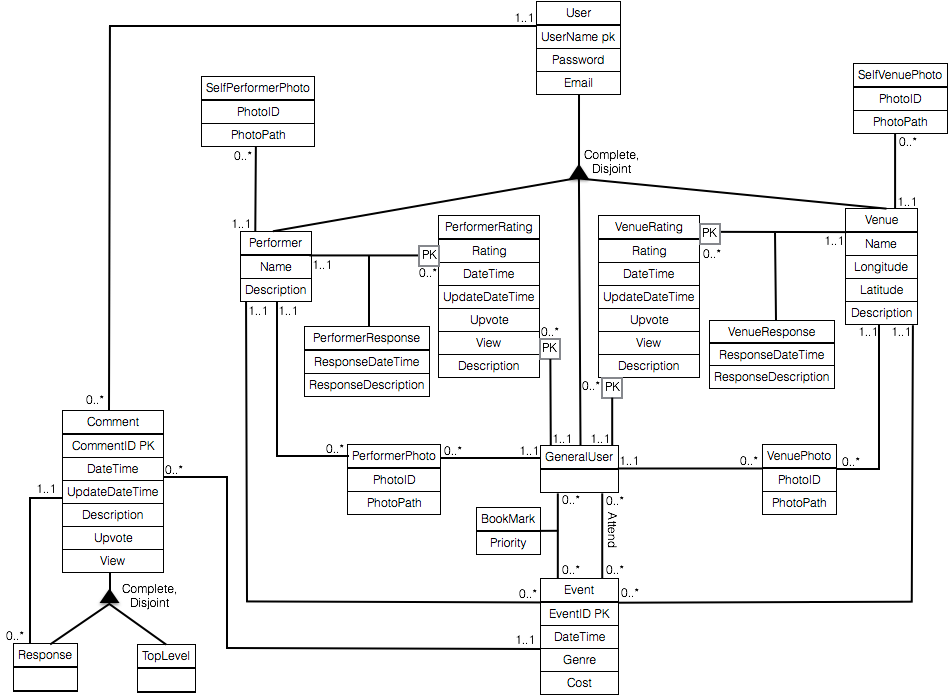
Four different photo tables exist in the database, two for performers and two for venues. The two types of photo tables are for photos uploaded by users and for photos uploaded by the performer or venue.

So that a User can only rate a performer or venue once, the performer and venue rating tables have a primary key containing the User performing the rating, and the entity being rated. The rating tables also contain information such as total number of upvotes and views to support functionality for users to judge other user’s ratings. Finally, the rating tables contain two columns that represent a performer or venue’s response to the rating. These can be null so that a performer or venue doesn’t need to respond to every rating.

Comments are split into two classes: top level and responses. The difference between these two is that responses contain a reference to the comment parent class. This allows responses to be responses to a top level or response. Top level is its own class to easily identify top level comments.

Performers, general users, and venues all belong to the same parent class which allows all of them to participate in comments. This makes sense because comments are only made in the scope of events which is supported by the 1..1 reference between comments and events.

#### SECTION 2. UML



#### SECTION 3. Assertions

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This assertion enforces disjointness between the Performer and GeneralUser tables.

This assertion is enforced by in-table checks in Performer and GeneralUser that make sure a User is either a Performer or GeneralUser. \*/

CREATE ASSERTION PerformerGeneralUserDisjoint(

CHECK (NOT EXISTS (SELECT \*

FROM Performer P, GeneralUser GU

WHERE P.UserName = GU.UserName))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This assertion enforces disjointness between the Performer and Venue tables.

This assertion is enforced by in-table checks in Performer and Venue that make sure a User is either a Performer or Venue. \*/

CREATE ASSERTION PerformerVenueDisjoint(

CHECK (NOT EXISTS (SELECT \*

FROM Performer P, Venue V

WHERE P.UserName = V.UserName))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This assertion enforces disjointness between the GeneralUser and Venue tables.

This assertion is enforced by in-table checks in GeneralUser and Venue that make sure a User is either a GeneralUser or Venue. \*/

CREATE ASSERTION GeneralUserVenueDisjoint(

CHECK (NOT EXISTS (SELECT \*

FROM GeneralUser GU, Venue V

WHERE GU.UserName = V.UserName))

);

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This assertion enforces disjointness between TopLevel and Response tables.

This assertion is enforced by in-table checks in TopLevel and Response that make sure a Comment is either a TopLevel or Response. \*/

CREATE ASSERTION TopLevelResponseDisjoint(

CHECK (NOT EXISTS (SELECT \*

FROM TopLevel TL, Response R

WHERE TL.CommentID = R.CommentID))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This assertion enforces completeness for User tables.

This assertion is enforced by an in-table check in User that makes sure a User has to exist in some child table, foreign key constraints in Performer, Venue, and GeneralUser, and triggers that will delete or update data in User if something in Performer, Venue, or GeneralUser are deleted or updated. \*/

CREATE ASSERTION UserCompleteness(

CHECK (NOT EXISTS (SELECT \*

FROM User U

WHERE U.UserName NOT IN (SELECT P.UserName

FROM Performer P))

OR (NOT EXISTS (SELECT \*

FROM User U

WHERE U.UserName NOT IN (SELECT V.UserName

FROM Venue V)))

OR (NOT EXISTS (SELECT \*

FROM User U

WHERE U.UserName NOT IN (SELECT GU.UserName

FROM GeneralUser GU))))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This assertion enforces completeness for Comment tables.

This assertion is enforced by an in-table check in Comment that makes sure a Comment has to exist in some child table, foreign key constraints in Response and TopLevel, and triggers that will delete or update data in Comment if something in TopLevel or Response are deleted or updated. \*/

CREATE ASSERTION CommentCompleteness(

CHECK (NOT EXISTS (SELECT \*

FROM Comment C

WHERE C.CommentID NOT IN (SELECT T.CommentID

FROM TopLevel T))

OR (NOT EXISTS (SELECT \*

FROM Comment C

WHERE C.CommentID NOT IN (SELECT R.CommentID

FROM Response R))))

);

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This assertion enforces that users can only rate a performer if the user has attended an event with the same performer. This assertion is enforced foreign key constraints in Attend, Event, and PerformerRating and triggers that will delete data in PerformerRating after something in Attend is deleted. \*/

CREATE ASSERTION PerformerRatingAttendEvent(

CHECK (NOT EXISTS (SELECT \*

FROM PerformerRating P

WHERE UserName NOT IN (SELECT UserName

FROM Attend A, Event E

WHERE A.EventID = E.EventID AND

E.Performer = P.Performer)))

);

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This assertion enforces that users can only rate a venue if the user has attended an event with the same venue. This assertion is enforced foreign key constraints in Attend, Event, and VenueRating and triggers that will delete data in VenueRating after something in Attend is deleted. \*/

CREATE ASSERTION VenueRatingAttendEvent(

CHECK (NOT EXISTS (SELECT \*

FROM VenueRating V

WHERE UserName NOT IN (SELECT UserName

FROM Attend A, Event E

WHERE A.EventID = E.EventID AND

E.Venue = V.Venue)))

);

/\*PRIMARY AUTHOR: SHAMITA NAGALLA

SECONDARY REVIEWERS: SHAON BOROSHA, GEOFF REN

This assertion enforces that events bookmarked by users have not already passed. This is implemented by an in table check. \*/

CREATE ASSERTION BookMarkPassedEvents(

CHECK (NOT EXISTS (SELECT \*

FROM BookMark B, Event E

WHERE B.EventID = E.EventID AND E.DateTime > GETDATE())

);

#### SECTION 4. Tables, Triggers, Inserts, Deletes, Updates, Views and Queries

#### /\* Group 11: Shaon Borosha, Shamita Nagalla, Geoff Ren \*/

##### /\* DROPS \*/

/\* Delete tables if they already exist \*/

DROP TABLE IF EXISTS User;

DROP TABLE IF EXISTS Performer;

DROP TABLE IF EXISTS GeneralUser;

DROP TABLE IF EXISTS Venue;

DROP TABLE IF EXISTS SelfPerformerPhoto;

DROP TABLE IF EXISTS SelfVenuePhoto;

DROP TABLE IF EXISTS PerformerPhoto;

DROP TABLE IF EXISTS VenuePhoto;

DROP TABLE IF EXISTS PerformerRating;

DROP TABLE IF EXISTS VenueRating;

DROP TABLE IF EXISTS BookMark;

DROP TABLE IF EXISTS Attend;

DROP TABLE IF EXISTS Event;

DROP TABLE IF EXISTS Comment;

DROP TABLE IF EXISTS TopLevel;

DROP TABLE IF EXISTS Response;

/\* Delete triggers if they already exist \*/

DROP TRIGGER IF EXISTS DeletePerformerDeleteUser;

DROP TRIGGER IF EXISTS DeleteGeneralUserDeleteUser;

DROP TRIGGER IF EXISTS DeleteVenueDeleteUser;

DROP TRIGGER IF EXISTS UpdatePerformerUpdateUser;

DROP TRIGGER IF EXISTS UpdateGeneralUserUpdateUser;

DROP TRIGGER IF EXISTS UpdateVenueUpdateUser;

DROP TRIGGER IF EXISTS DeleteTopLevelDeleteComment;

DROP TRIGGER IF EXISTS DeleteResponseDeleteComment;

DROP TRIGGER IF EXISTS UpdateTopLevelUpdateComment;

DROP TRIGGER IF EXISTS UpdateResponseUpdateComment;

DROP TRIGGER IF EXISTS DeleteAttendDeleteVenueRating;

DROP TRIGGER IF EXISTS DeleteAttendDeletePerformerRating;

DROP TRIGGER IF EXISTS DeleteAttendedEvents;

/\* Delete views if they already exist \*/

DROP VIEW IF EXISTS VenueAnalyticView;

DROP VIEW IF EXISTS PerformerAnalyticView;

DROP VIEW IF EXISTS UserPreferences;

DROP VIEW IF EXISTS VenueInformation;

##### /\* TABLES \*/

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about the user. \*/

CREATE TABLE User (

UserName VARCHAR(32),

Password CHARACTER(60) NOT NULL, /\* using bcrypt for encryption so built-in salt \*/

Email VARCHAR(255) NOT NULL UNIQUE,

PRIMARY KEY(UserName),

CHECK (EXISTS (SELECT \*

FROM Performer P, GeneralUser G, Venue V

WHERE UserName = P.UserName OR UserName = G.UserName

OR UserName = V.UserName))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about the general user. \*/

CREATE TABLE GeneralUser (

UserName VARCHAR(32),

PRIMARY KEY(UserName),

FOREIGN KEY(UserName) REFERENCES User(UserName) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK (NOT EXISTS (SELECT \*

FROM Performer P

WHERE UserName = P.UserName)),

CHECK (NOT EXISTS (SELECT \*

FROM Venue V

WHERE UserName = V.UserName))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about performers. \*/

CREATE TABLE Performer (

UserName VARCHAR(32),

Name VARCHAR(255) NOT NULL,

Description TEXT,

PRIMARY KEY(UserName),

FOREIGN KEY(UserName) REFERENCES User(UserName) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK (NOT EXISTS (SELECT \*

FROM GeneralUser GU

WHERE UserName = GU.UserName)),

CHECK (NOT EXISTS (SELECT \*

FROM Venue V

WHERE UserName = V.UserName))

);

/\* PRIMARY AUTHORS: SHAON BOROSHA, GEOFF REN

SECONDARY REVIEWERS: SHAMITA NAGALLA

This table holds information about a venue. \*/

CREATE TABLE Venue (

UserName VARCHAR(32),

Name VARCHAR(255) NOT NULL,

Longitude DECIMAL(9,6) NOT NULL,

Latitude DECIMAL(9,6) NOT NULL,

Description TEXT,

PRIMARY KEY(UserName),

FOREIGN KEY(UserName) REFERENCES USER(UserName) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK (NOT EXISTS (SELECT \*

FROM Performer P

WHERE UserName = P.UserName)),

CHECK (NOT EXISTS (SELECT \*

FROM GeneralUser GU

WHERE UserName = GU.UserName))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about photos of performers uploaded by the performer. \*/

CREATE TABLE SelfPerformerPhoto(

PhotoID INT AUTOINCREMENT,

Performer VARCHAR(32) NOT NULL,

PhotoPath VARCHAR(255) NOT NULL UNIQUE,

PRIMARY KEY(PhotoID),

FOREIGN KEY(Performer) REFERENCES Performer(UserName) ON DELETE

CASCADE ON UPDATE CASCADE

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about photos of venues uploaded by the venue. \*/

CREATE TABLE SelfVenuePhoto(

PhotoID INT AUTOINCREMENT,

Venue VARCHAR(32) NOT NULL,

PhotoPath VARCHAR(255) NOT NULL UNIQUE,

PRIMARY KEY(PhotoID),

FOREIGN KEY(Venue) REFERENCES Venue(UserName) ON DELETE

CASCADE ON UPDATE CASCADE

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about photos of performers uploaded by a general user. \*/

CREATE TABLE PerformerPhoto(

PhotoID INT AUTOINCREMENT,

UserName VARCHAR(32) NOT NULL,

Performer VARCHAR(32) NOT NULL,

PhotoPath VARCHAR(255) NOT NULL UNIQUE,

PRIMARY KEY(PhotoID),

FOREIGN KEY(UserName) REFERENCES GeneralUser(UserName) ON DELETE

CASCADE ON UPDATE CASCADE,

FOREIGN KEY(Performer) REFERENCES Performer(UserName) ON DELETE

CASCADE ON UPDATE CASCADE

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about photos of venues uploaded by a general user. \*/

CREATE TABLE VenuePhoto(

PhotoID INT AUTOINCREMENT,

UserName VARCHAR(32) NOT NULL,

Venue VARCHAR(32) NOT NULL,

PhotoPath VARCHAR(255) NOT NULL UNIQUE,

PRIMARY KEY(PhotoID),

FOREIGN KEY(UserName) REFERENCES GeneralUser(UserName) ON DELETE

CASCADE ON UPDATE CASCADE,

FOREIGN KEY(Venue) REFERENCES Venue(UserName) ON DELETE

CASCADE ON UPDATE CASCADE

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds ratings corresponding to a performer. \*/

CREATE TABLE PerformerRating (

UserName VARCHAR(32),

Performer VARCHAR(32),

Rating TINYINT NOT NULL,

DateTime DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,

UpdateDateTime DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,

Upvote INT NOT NULL DEFAULT 0,

View INT NOT NULL DEFAULT 0,

Description TEXT,

ResponseDateTime DATETIME,

ResponseDescription TEXT,

PRIMARY KEY(UserName, Performer),

FOREIGN KEY(UserName) REFERENCES GeneralUser(UserName) ON DELETE CASCADE ON UPDATE CASCADE,

FOREIGN KEY(Performer) REFERENCES Performer(UserName) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK(Rating >= 0 AND Rating <= 5)

);

/\* PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: GEOFF REN, SHAON BOROSHA

This table holds ratings corresponding to a venue. \*/

CREATE TABLE VenueRating (

UserName VARCHAR(32),

Venue VARCHAR(32),

Rating TINYINT NOT NULL,

DateTime DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,

UpdateDateTime DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,

Upvote INT NOT NULL DEFAULT 0,

View INT NOT NULL DEFAULT 0,

Description TEXT,

ResponseDateTime DATETIME,

ResponseDescription TEXT,

PRIMARY KEY (UserName, Venue),

FOREIGN KEY(UserName) REFERENCES GeneralUser(UserName) ON DELETE

CASCADE ON UPDATE CASCADE,

FOREIGN KEY(Venue) REFERENCES Venue(UserName) ON DELETE CASCADE

ON UPDATE CASCADE,

CHECK (Rating >= 0 AND Rating <= 5)

);

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This table holds information about an event. \*/

CREATE TABLE Event (

EventID INT AUTOINCREMENT,

Performer VARCHAR(32) NOT NULL,

Venue VARCHAR(32) NOT NULL,

DateTime DATETIME NOT NULL,

Genre VARCHAR(255),

Cost INT, /\* store cost in cents \*/

PRIMARY KEY(EventID),

FOREIGN KEY(Performer) REFERENCES Performer(UserName) ON DELETE

CASCADE ON UPDATE CASCADE,

FOREIGN KEY(Venue) REFERENCES Venue(UserName) ON DELETE CASCADE

ON UPDATE CASCADE

);

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This table holds information about events attended by users. \*/

CREATE TABLE Attend (

UserName VARCHAR(32),

EventID INT,

PRIMARY KEY(UserName, EventID),

FOREIGN KEY(UserName) REFERENCES GeneralUser(UserName) ON DELETE CASCADE ON UPDATE CASCADE,

FOREIGN KEY(EventID) REFERENCES Event(EventID) ON DELETE

CASCADE ON UPDATE CASCADE

);

/\* PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: GEOFF REN, SHAON BOROSHA

This table holds information about events bookmarked by users. \*/

CREATE TABLE BookMark (

UserName VARCHAR(32),

EventID INT,

Priority INT NOT NULL DEFAULT 0,

PRIMARY KEY (UserName, EventID),

FOREIGN KEY(UserName) REFERENCES GeneralUser(UserName) ON DELETE

CASCADE ON UPDATE CASCADE,

FOREIGN KEY (EventID) REFERENCES Event(EventID) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK (Priority >= 0 AND Priority <= 10)

CHECK (NOT EXISTS (SELECT \*

FROM Event E

WHERE EventID = E.EventID AND

E.DateTime > CURRENT\_TIMESTAMP)

);

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: SHAMITA NAGALLA, GEOFF REN

This table holds information about a general comment. \*/

CREATE TABLE Comment (

CommentID INT AUTOINCREMENT,

UserName VARCHAR(32) NOT NULL,

EventID INT NOT NULL,

DateTime DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,

UpdateDateTime DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,

Description TEXT NOT NULL,

Upvote INT NOT NULL DEFAULT 0,

View INT NOT NULL DEFAULT 0,

PRIMARY KEY(CommentID),

FOREIGN KEY(UserName) REFERENCES User(UserName) ON DELETE CASCADE ON UPDATE CASCADE,

FOREIGN KEY(EventID) REFERENCES Event(EventID) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK (EXISTS (SELECT \*

FROM TopLevel T, Response R

WHERE CommentID = T.CommentID OR

CommentID = R.CommentID))

);

/\* PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: GEOFF REN, SHAON BOROSHA

This table holds information about top level comments. \*/

CREATE TABLE TopLevel(

CommentID INT,

PRIMARY KEY(CommentID),

FOREIGN KEY(CommentID) REFERENCES Comment(CommentID) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK (NOT EXISTS (SELECT \* FROM Response R

WHERE CommentID = R.CommentID))

);

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This table holds information about users who bookmarked an event. \*/

CREATE TABLE Response(

CommentID INT,

RespondeeID INT NOT NULL,

PRIMARY KEY(CommentID),

FOREIGN KEY(CommentID) REFERENCES Comment(CommentID) ON DELETE CASCADE ON UPDATE CASCADE,

\*/ If the comment the response is responding to is deleted, delete the entire chain \*/

FOREIGN KEY(RespondeeID) REFERENCES Comment(CommentID) ON DELETE CASCADE ON UPDATE CASCADE,

CHECK (NOT EXISTS (SELECT \* FROM TopLevel T

WHERE CommentID = T.CommentID))

);

##### /\* TRIGGERS \*/

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will delete a User if the User is deleted from Performer. \*/

CREATE TRIGGER DeletePerformerDeleteUser

AFTER DELETE ON Performer

BEGIN

DELETE FROM User

WHERE UserName = OLD.UserName;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will delete a User if the User is deleted from GeneralUser. \*/

CREATE TRIGGER DeleteGeneralUserDeleteUser

AFTER DELETE ON GeneralUser

BEGIN

DELETE FROM User

WHERE UserName = OLD.UserName;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will delete a User if the User is deleted from Venue. \*/

CREATE TRIGGER DeleteVenueDeleteUser

AFTER DELETE ON Venue

BEGIN

DELETE FROM User

WHERE UserName = OLD.UserName;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will update a User if the User is updated from Performer. \*/

CREATE TRIGGER UpdatePerformerUpdateUser

AFTER UPDATE ON Performer

BEGIN

UPDATE User

SET UserName = NEW.UserName;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will update a User if the User is updated from GeneralUser. \*/

CREATE TRIGGER UpdateGeneralUserUpdateUser

AFTER UPDATE ON GeneralUser

BEGIN

UPDATE User

SET UserName = NEW.UserName;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will update a User if the User is updated from Venue. \*/

CREATE TRIGGER UpdateVenueUpdateUser

AFTER UPDATE ON Venue

BEGIN

UPDATE User

SET UserName = NEW.UserName;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will delete a Comment if the Comment is deleted from TopLevel. \*/

CREATE TRIGGER DeleteTopLevelDeleteComment

AFTER DELETE ON TopLevel

BEGIN

DELETE FROM Comment

WHERE CommentID = OLD.CommentID;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This trigger will delete a Comment if the Comment is deleted from Response. \*/

CREATE TRIGGER DeleteResponseDeleteComment

AFTER DELETE ON Response

BEGIN

DELETE FROM Comment

WHERE CommentID = OLD.CommentID;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAMITA NAGALLA, SHAON BOROSHA

This trigger will update a Comment if the Comment is updated from TopLevel. \*/

CREATE TRIGGER UpdateTopLevelUpdateComment

AFTER UPDATE ON TopLevel

BEGIN

UPDATE Comment

SET CommentID = NEW.CommentID;

END;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAMITA NAGALLA, SHAON BOROSHA

This trigger will update a Comment if the Comment is updated from Response. \*/

CREATE TRIGGER UpdateResponseUpdateComment

AFTER UPDATE ON Response

BEGIN

UPDATE Comment

SET CommentID = NEW.CommentID;

END;

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This trigger will delete a venue rating by a user after the user’s attendance at an event with that venue is deleted. \*/

CREATE TRIGGER DeleteAttendDeleteVenueRating

AFTER DELETE ON Attend

WHEN (SELECT COUNT(\*)

FROM /\*venue from deleted attend event\*/

(SELECT E.Venue AS Venue

FROM Event E

WHERE OLD.EventID = E.EventID) X,

/\*venues from attended events by user\*/

(SELECT E.Venue as Venue

FROM Event E, Attend A

WHERE A.EventID = E.EventID AND

A.UserName = OLD.UserName) Y

WHERE Y.Venue = X.Venue) = 0

BEGIN

DELETE FROM VenueRating

WHERE UserName = OLD.UserName

AND Venue IN (SELECT E.Venue

FROM Event E

WHERE E.EventID = OLD.EventID)

END;

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: SHAMITA NAGALLA

This trigger will delete a performer rating by a user after the user’s attendance at an event with that performer is deleted. \*/

CREATE TRIGGER DeleteAttendDeletePerformerRating

AFTER DELETE ON Attend

WHEN (SELECT COUNT(\*)

FROM /\*venue from deleted attend event\*/

(SELECT E.Performer AS Performer

FROM Event E

WHERE OLD.EventID = E.EventID) X,

/\*venues from attended events by user\*/

(SELECT E.Performer as Performer

FROM Event E, Attend A

WHERE A.EventID = E.EventID AND

A.UserName = OLD.UserName) Y

WHERE Y.Performer = X.Performer) = 0

BEGIN

DELETE FROM PerformerRating

WHERE UserName = OLD.UserName

AND Performer IN (SELECT E.Performer

FROM Event E

WHERE E.EventID = OLD.EventID)

END;

/\*PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: GEOFF REN, SHAON BOROSHA

This triggers deletes bookmarked events once they have been attended. \*/

CREATE TRIGGER DeleteAttendedEvents

AFTER INSERT ON Attend

BEGIN

DELETE FROM BookMark

WHERE BookMark.EventID = New.EventID AND

BookMark.UserName = New.UserName

END;

##### /\* INSERTS \*/

PRAGMA foreign\_keys = OFF;

INSERT INTO User VALUES(‘GeoffRen’,

‘$2b$12$GPBNbk5H6EJOkJ5rdg3PW.AO7v/vYvHhsUZJhQco77.QPDW2oR6em’,

‘geoff.ren@vanderbilt.edu’);

INSERT INTO User VALUES(‘ShamitaNagalla’,

‘$2b$12$z9Y3VAJp0W4n5ZQlnOjPKOBUO/.cDsmuKAG.9DjwsZAH7WsPkGtR2’,

‘shamita.nagalla@vanderbilt.edu’);

INSERT INTO User VALUES(‘ShaonBorosha’,

‘$2b$12$zgmKEqYDtYCRM1sRFC8Od.p/F9Y/98xbnMsArUY7TA50/2OhIJPxq’,

‘shaon.borosha@vanderbilt.edu’);

INSERT INTO User VALUES(‘LebronJames’,

‘$2b$12$iF6a9EW.IJU2GTjf/tCaIeX8J6RxV3rCqBa9bcByXllzOfhbMRpRC’,

‘lebron.james@gmail.com’);

INSERT INTO User VALUES(‘CathyWilla’,

‘$2b$12$oVqcsci6ChvD8Us0Yc9p6utxw83cU13qwnSP1vFfquTxpKCSmgPza’,

‘cw@mail.gov’);

INSERT INTO User VALUES(‘Puckett’’s’,

‘$2b$12$muVk1ZlMMHAZEqOjTopYuuSQtJFPrvGeLcj9z8mhC/gVn8pgs6t42’,

‘pucketts.@restaurant.com’);

INSERT INTO User VALUES(‘ExitIn’,

‘$2b$12$w8fPKHHNMTuoIMpeE2BH5.c0DSxzR6U.z4oz4rgl4pSIiQVzdYcRe’,

‘exit@in.com’);

INSERT INTO User VALUES(‘Bluebird’,

‘$2b$12$tk5y1h2GqEJ5HttOLQOGWeNtLgRgWcPWhs5225k/UUS6BcWbY6S6m’,

‘blue.bird@nest.com’);

INSERT INTO User VALUES(‘TaylorSwift’,

‘$2b$12$z9Y3VAJp0W4n5ZQlnOjPKOBUO/.cDsmuKAG.9DjwsZAH7WsPkGtR2’,

‘shamita.nagalla@vanderbilt.edu’);

INSERT INTO User VALUES(‘KanyeWest’,

‘$2b$12$zgmKEqYDtYCRM1sRFC8Od.p/F9Y/98xbnMsArUY7TA50/2OhIJPxq’,

‘shaon.borosha@vanderbilt.edu’);

INSERT INTO User VALUES(‘RollingStones’,

‘$2b$12$tbbmeTdhH3y/dT7rPA6xr.PTDF90pG9HrziCtXhehj691gxbYajNq’,

‘rollingstones@music.com’);

INSERT INTO User VALUES(‘Metallica’,

‘$2b$12$EizVNDudVBTnVlz5sFcIzOHrfVeeGaB0xVxm.oECav1tGZRiNUWJa’

‘metallica@metallica.com’);

INSERT INTO User VALUES(‘Acme’,

‘$2b$12$FSclaEzha06QWo4aJ4QBweudirHo7oOg4C5x3CacRdGA/TbvAU5Qe’

‘acme@feed.seed’);

INSERT INTO GeneralUser VALUES(‘GeoffRen’);

INSERT INTO GeneralUser VALUES(‘ShamitaNagalla’);

INSERT INTO GeneralUser VALUES(‘ShaonBorosha’);

INSERT INTO GeneralUser VALUES(‘LebronJames’);

INSERT INTO GeneralUser VALUES(‘CathyWilla’);

INSERT INTO Performer VALUES(‘TaylorSwift’, ‘Taylor Swift’, ‘I am Taylor Swift and I like to hold concerts that you can’t afford to get into if you live in Nashville’);

INSERT INTO Performer VALUES(‘KanyeWest’, ‘Kanye West’, ‘I am Kanye West and I have a God complex’);

INSERT INTO Performer VALUES(‘RollingStones’, ‘Rolling Stones’, NULL);

INSERT INTO Performer VALUES(‘Metallica’, ‘Metallica’, NULL);

INSERT INTO Venue VALUES(‘Puckett’’s’, ‘Puckett’’s Grocery & Restaurant’, 36.163225, -86.780460, NULL);

INSERT INTO Venue VALUES(‘ExitIn’, ‘Exit/In’, 36.151384, -86.804395, ‘We are really close to campus!’);

INSERT INTO Venue VALUES(‘Bluebird’, ‘The Bluebird Cafe’, 36.102054, -86.816736, ‘Good luck getting tickets!’);

INSERT INTO Venue VALUES(Acme, ‘Acme Feed & Seed’, 36.161998, -86.774402, ‘We are pretty cool’);

INSERT INTO SelfPerformerPhoto VALUES(NULL, ‘TaylorSwift’, ‘c:\somepath\taylor1.jpg’);

INSERT INTO SelfPerformerPhoto VALUES(NULL, ‘TaylorSwift’, ‘c:\somepath\taylor2.jpg’);

INSERT INTO SelfPerformerPhoto VALUES(NULL, ‘KanyeWest’, ‘c:\somepath\kanye1.jpg’);

INSERT INTO SelfVenuePhoto VALUES(NULL, ‘ExitIn’, ‘c:\somepath\exit1.jpg’);

INSERT INTO SelfVenuePhoto VALUES(NULL, ‘ExitIn’, ‘c:\somepath\exit2.jpg’);

INSERT INTO SelfVenuePhoto VALUES(NULL, ‘Bluebird’, ‘c:\somepath\bluebird.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘GeoffRen’, ‘TaylorSwift’,

‘c:\somepath\geoffswift1.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘GeoffRen’, ‘TaylorSwift’,

‘c:\somepath\geoffswift2.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘GeoffRen’, ‘KanyeWest’,

‘c:\somepath\geoffwest1.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘ShamitaNagalla’, ‘KanyeWest’,

‘c:\somepath\sham1.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘ShamitaNagalla’, ‘KanyeWest’,

‘c:\somepath\sham2.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘ShamitaNagalla’, ‘KanyeWest’,

‘c:\somepath\sham3.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘ShamitaNagalla’, ‘KanyeWest’,

‘c:\somepath\sham4.jpg’);

INSERT INTO PerformerPhoto VALUES(NULL, ‘ShaonBorosha’, ‘TaylorSwift’,

‘c:\somepath\shaon1.jpg’);

INSERT INTO VenuePhoto VALUES(NULL, ‘GeoffRen’, ‘ExitIn’,

‘c:\somepath\geoffexitin1.jpg’);

INSERT INTO VenuePhoto VALUES(NULL, ‘GeoffRen’, ‘ExitIn’,

‘c:\somepath\geoffexitin2.jpg’);

INSERT INTO VenuePhoto VALUES(NULL, ‘GeoffRen’, ‘Bluebird’,

‘c:\somepath\geoffbird1.jpg’);

INSERT INTO VenuePhoto VALUES(NULL, ‘ShamitaNagalla, ‘Bluebird’,

‘c:\somepath\shambird1.jpg’)

INSERT INTO VenuePhoto VALUES(NULL, ‘ShamitaNagalla, ‘Bluebird’,

‘c:\somepath\shambird2.jpg’)

INSERT INTO VenuePhoto VALUES(NULL, ‘ShamitaNagalla, ‘Bluebird’,

‘c:\somepath\shambird3.jpg’)

INSERT INTO VenuePhoto VALUES(NULL, ‘ShamitaNagalla, ‘Bluebird’,

‘c:\somepath\shambird4.jpg’)

INSERT INTO VenuePhoto VALUES(NULL, ‘ShaonBorosha’, ‘ExitIn’,

‘c:\somepath\shaonexit1.jpg’);

INSERT INTO PerformerRating VALUES(‘GeoffRen’, ‘TaylorSwift’, 5, ‘2017-01-01 20:00:00’,

‘2017-01-01 20:00:00’, 500, 1000, ‘Cool stuff’, ‘2017-02-01 11:00:00’, ‘Thanks!’);

INSERT INTO PerformerRating VALUES(‘GeoffRen’, ’KanyeWest’, 4, ‘2017-05-01 20:00:00’,

‘2017-05-01 20:00:00’, 4, 25,’Good stuff’, NULL, NULL);

INSERT INTO PerformerRating VALUES(‘GeoffRen’, ‘Metallica’, 3, ‘2010-01-01 20:00:00’,

‘2010-01-01 20:00:00’, 25, 677, ‘Fine stuff’, ‘2011-02-01 11:00:00’, ‘Yea’);

INSERT INTO PerformerRating VALUES(‘ShamitaNagalla’, ‘TaylorSwift’, 5, ‘2017-02-01

20:00:00’, ‘2017-04-01 20:00:00’, 43, 142, ‘Cool’, ‘2017-05-01 11:00:00’, ‘Thanks!’);

INSERT INTO PerformerRating VALUES(‘ShamitaNagalla’, ’KanyeWest’, 4, ‘2016-05-01

20:00:00’, ‘2016-05-01 20:00:00’, 4, 55,’Good stuff’, ‘2016-06-01 20:00:00’, ‘Thanks!’);

INSERT INTO PerformerRating VALUES(‘ShamitaNagalla’, ‘Metallica’, 4, ‘2010-02-01 20:00:00’, ‘2010-02-01 20:00:00’, 0, 0, ‘Fine stuff’, ‘2011-03-01 11:00:00’, ‘Yea’);

INSERT INTO PerformerRating VALUES(‘ShaonBorosha’, ‘TaylorSwift’, 5, ‘2017-03-01 20:00:00’,

‘2017-03-01 20:00:00’, 0, 5, ‘Nice’, ‘2017-04-01 11:00:00’, ‘Thanks!’);

INSERT INTO PerformerRating VALUES(‘ShaonBorosha’, ’KanyeWest’, 3, ‘2016-09-01 20:00:00’,

‘2016-09-01 20:00:00’, 6, 56,’Good stuff’, ‘2016-10-01 20:00:00’, ‘Thanks!’);

INSERT INTO PerformerRating VALUES(‘LebronJames’, ‘TaylorSwift’, 5, ‘2017-07-01 20:00:00’,

‘2017-07-01 20:00:00’, 0, 123, ‘Great’, ‘2017-09-01 11:00:00’, ‘Thanks!’);

INSERT INTO VenueRating VALUES(‘GeoffRen’, ‘ExitIn’, 5, ‘2017-01-01 20:00:00’, ‘2017-01-01

20:00:00’, 123, 3141, ‘Cool’, ‘2017-02-01 20:00:00’, ‘Thanks!’);

INSERT INTO VenueRating VALUES(‘GeoffRen’, ‘Bluebird’, 2, ‘2016-01-01 20:00:00’,

‘2016-01-01 20:00:00’, 123, 3141, ‘Fine’, ‘2016-02-01 20:00:00’, ‘Aww’);

INSERT INTO VenueRating VALUES(‘GeoffRen’, ‘Acme’, 4, ‘2013-01-01 20:00:00’, ‘2013-01-01

20:00:00’, 123, 3141, ‘Fine’, NULL, NULL);

INSERT INTO VenueRating VALUES(‘ShamitaNagalla’, ‘ExitIn’, 4, ‘2017-02-01 20:00:00’,

‘2017-02-01 20:00:00’, 123, 3141, ‘Cool’, ‘2017-03-01 20:00:00’, ‘Thanks!’);

INSERT INTO VenueRating VALUES(‘ShamitaNagalla’, ‘Acme’, 3, ‘2013-01-01 20:00:00’,

‘2013-01-01 20:00:00’, 123, 3141, ‘Fine’, NULL, NULL);

INSERT INTO VenueRating VALUES(‘ShamitaNagalla’, ‘Bluebird’, 4, ‘2014-01-01 20:00:00’,

‘2014-01-01 20:00:00’, 123, 3141, ‘Fine’, NULL, NULL);

INSERT INTO VenueRating VALUES(‘ShaonBorosha’, ‘ExitIn’, 5, ‘2017-03-01 20:00:00’,

‘2017-03-01 20:00:00’, 123, 3141, ‘Cool’, ‘2017-04-01 20:00:00’, ‘Thanks!’);

INSERT INTO VenueRating VALUES(‘ShaonBorosha’, ‘Bluebird’, 3, ‘2014-02-01 20:00:00’,

‘2014-02-01 20:00:00’, 123, 3141, ‘Fine’, ‘2014-03-01 20:00:00’, ‘Good enough’);

INSERT INTO VenueRating VALUES(‘LebronJames’, ‘ExitIn’, 5, ‘2017-04-01 20:00:00’,

‘2017-04-01 20:00:00’, 123, 3141, ‘Cool’, ‘2017-05-01 20:00:00’, ‘Thanks!’);

INSERT INTO Event VALUES(NULL, ‘TaylorSwift’, ‘ExitIn’, ‘2009-01-01 20:00:00’, ‘Country’,

100000);

INSERT INTO Event VALUES(NULL, ‘TaylorSwift’, ‘Bluebird’, ‘2009-02-01 20:00:00’, ‘Country’,

100000);

INSERT INTO Event VALUES(NULL, ‘TaylorSwift’, ‘Acme’, ‘2009-03-01 20:00:00’, ‘Country’,

100000);

INSERT INTO Event VALUES(NULL, ‘KanyeWest’, ‘ExitIn’, ‘2008-01-01 20:00:00’, ‘Hip-Hop’,

1000);

INSERT INTO Event VALUES(NULL, ‘KanyeWest’, ‘Bluebird’, ‘2008-02-01 20:00:00’, ‘Hip-Hop’,

1000);

INSERT INTO Event VALUES(NULL, ‘Metallica’, ‘ExitIn’, ‘2007-01-01 20:00:00’, ‘Metal’, 100);

INSERT INTO Event VALUES(NULL, ‘Metallica’, ‘ExitIn’, ‘2020-01-01 20:00:00’, ‘Metal’, 100);

INSERT INTO Event VALUES(NULL, ‘Metallica’, ‘ExitIn’, ‘2020-01-02 20:00:00’, ‘Metal’, 100);

INSERT INTO Event VALUES(NULL, ‘TaylorSwift’, ‘Acme’, ‘2020-03-01 20:00:00’, ‘Country’,

100000);

INSERT INTO Attend VALUES(‘GeoffRen’, 1);

INSERT INTO Attend VALUES(‘GeoffRen’, 2);

INSERT INTO Attend VALUES(‘GeoffRen’, 3);

INSERT INTO Attend VALUES(‘GeoffRen’, 4);

INSERT INTO Attend VALUES(‘GeoffRen’, 5);

INSERT INTO Attend VALUES(‘GeoffRen’, 6);

INSERT INTO Attend VALUES(‘ShamitaNagalla’, 3);

INSERT INTO Attend VALUES(‘ShamitaNagalla’, 6);

INSERT INTO Attend VALUES(‘ShamitaNagalla’, 5);

INSERT INTO Attend VALUES(‘ShaonBorosha’, 2);

INSERT INTO Attend VALUES(‘ShaonBorosha’, 4);

INSERT INTO Attend VALUES(‘LebronJames’, 1);

INSERT INTO Attend VALUES(‘LebronJames’, 2);

INSERT INTO Attend VALUES(‘LebronJames’, 3);

INSERT INTO Attend VALUES(‘LebronJames’, 6);

INSERT INTO BookMark VALUES(‘GeoffRen’, 7, 10);

INSERT INTO BookMark VALUES(‘GeoffRen’, 8, 9);

INSERT INTO BookMark VALUES(‘GeoffRen’, 9, 10);

INSERT INTO BookMark VALUES(‘ShamitaNagalla’, 7, 5);

INSERT INTO BookMark VALUES(‘ShamitaNagalla’, 8, 6);

INSERT INTO BookMark VALUES(‘ShaonBorosha’, 7, 1);

INSERT INTO Comment VALUES(NULL, ‘GeoffRen’, 1, ‘2009-01-02 20:00:00’, ‘2009-01-02

20:00:00’, ‘Toplevel comment for Event 1’, 500, 1000);

INSERT INTO Comment VALUES(NULL, ‘GeoffRen’, 1, ‘2009-01-03 20:00:00’, ‘2009-02-03

20:00:00’, ‘Toplevel comment for Event 1’, 0, 120);

INSERT INTO Comment VALUES(NULL, ‘GeoffRen’, 1, ‘2009-01-03 20:00:00’, ‘2009-03-03

20:00:00’, ‘Toplevel comment for Event 1’, 35, 544);

INSERT INTO Comment VALUES(NULL, ‘GeoffRen’, 1, ‘2009-01-03 20:00:00’, ‘2009-04-03

20:00:00’, Response comment for Comment 1’, 0, 4);

INSERT INTO Comment VALUES(NULL, ‘GeoffRen’, 3, ‘2009-01-03 20:00:00’, ‘2009-05-03

20:00:00’, ‘Toplevel comment for Event 3’, 12, 1212);

INSERT INTO Comment VALUES(NULL, ‘ShamitaNagalla’, 2, ‘2009-01-03 20:00:00’,

‘2009-01-03 20:00:00’, ‘Toplevel comment for Event 2’, 23, 142);

INSERT INTO Comment VALUES(NULL, ‘ShamitaNagalla’, 3, ‘2009-01-03 20:00:00’,

‘2009-01-03 20:00:00’, ‘Toplevel comment for Event 3’, 2, 244);

INSERT INTO Comment VALUES(NULL, ‘ShamitaNagalla’, 4, ‘2009-01-03 20:00:00’,

‘2009-01-03 20:00:00’, ‘Toplevel comment for Event 4’, 345, 2555);

INSERT INTO Comment VALUES(NULL, ‘ShaonBorosha’, 2, ‘2009-01-04 20:00:00’,

‘2009-01-03 20:00:00’, Response comment for Event 2’, 1, 23);

INSERT INTO Comment VALUES(NULL, ‘ShaonBorosha’, 2, ‘2009-01-05 20:00:00’,

‘2009-01-03 20:00:00’, Response comment for Event 2’, 4, 22);

INSERT INTO Comment VALUES(NULL, ‘ShaonBorosha’, 2, ‘2009-01-06 20:00:00’,

‘2009-01-03 20:00:00’, Response comment for Event 2’, 1, 5);

INSERT INTO Comment VALUES(NULL, ‘ShaonBorosha’, 2, ‘2009-01-07 20:00:00’,

‘2009-01-03 20:00:00’, Response comment for Event 2’, 0, 23);

INSERT INTO TopLevel VALUES(1);

INSERT INTO TopLevel VALUES(2);

INSERT INTO TopLevel VALUES(3);

INSERT INTO TopLevel VALUES(5);

INSERT INTO TopLevel VALUES(6);

INSERT INTO TopLevel VALUES(7);

INSERT INTO TopLevel VALUES(8);

INSERT INTO Response VALUES(4, 1);

INSERT INTO Response VALUES(9, 6);

INSERT INTO Response VALUES(10, 9);

INSERT INTO Response VALUES(11, 9);

INSERT INTO Response VALUES(12, 10);

PRAGMA foreign\_keys = ON;

##### /\* VIEWS \*/

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAMITA NAGALLA, SHAON BOROSHA

This view displays various analytics for each venue. These analytics include: the number of times the venue was rated, the average rating for the venue, the number of events the venue has appeared in, the average cost of events at the venue, the total number of comments events hosted at the venue have, the average number of comments per event hosted at the venue have, the total number of users who have attended an event hosted at the venue, and the average number of users per event hosted at this venue. \*/

CREATE VIEW VenueAnalyticView(UserName, Venue, NumRatings, AvgRating, NumEvents AvgCost, NumEventComments, AvgEventComments,

NumAttendees, AvgAttendees )

AS SELECT V.UserName, V.Name, Rating.NumRatings, Rating.AvgRating, Event.NumEvents, Event.AvgCost, Comment.NumEventComments, Comment.AvgEventComments, Attend.NumAttendees, Attend.AvgAttendees

FROM /\* Finds the number of ratings and average rating for each Venue. \*/

(SELECT R.Venue AS UserName, COUNT(\*) AS NumRatings,

AVG(R.Rating) AS AvgRating

FROM VenueRating R

GROUP BY R.Venue) Rating,

/\* Finds the number of events and average cost of events each Venue has been a part of. \*/

(SELECT E.Venue AS UserName, COUNT(\*) AS NumEvents ,

AVG(E.Cost) AS AvgCost

FROM Event E

GROUP BY E.Venue) Event,

/\* Finds the total and average number of comments of events each Venue has hosted. \*/

(SELECT tmp.UserName AS UserName, SUM(tmp.NumComments) AS

NumEventComments, AVG(tmp.NumComments) AS AvgEventComments

FROM (SELECT E.Venue AS UserName, COUNT(\*) AS NumComments

FROM Event E, Comment C

WHERE E.EventID = C.EventID

GROUP BY E.Venue, E.EventID) tmp

GROUP BY tmp.UserName) Comment,

/\* Finds the number and average number of attendees of events each Venue has hosted. \*/

(SELECT tmp.UserName AS UserName, SUM(tmp.NumAttendees) AS NumAttendees, AVG(tmp.NumAttendees) AS AvgAttendees

FROM (SELECT E.Venue AS UserName, COUNT(\*) AS NumAttendees

FROM Event E, Attend A

WHERE E.EventID = A.EventID

GROUP BY E.Venue, E.EventID) tmp

GROUP BY tmp.UserName) Attend,

Venue V

WHERE V.UserName = Rating.UserName AND V.UserName = Event.UserName AND V.UserName = Comment.UserName AND V.UserName = Attend.UserName;

/\* PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAMITA NAGALLA, SHAON BOROSHA

This view displays various analytics for each performer. These analytics include: the number of times the performer was rated, the average rating for the performer, the number of events the performer has appeared in, the average cost of events with the performer, the total number of comments events with the performer have, the average number of comments per event the performer appears in have, the total number of users who have attended an event performed by the performer, and the average number of users per event performed by the performer. \*/

CREATE VIEW PerformerAnalyticView(UserName, Performer, NumRatings, AvgRating,

NumEvents, AvgCost, NumEventComments, AvgEventComments, NumAttendees, AvgAttendees)

AS SELECT P.UserName, P.Name, Rating.NumRatings, Rating.AvgRating, Event.NumEvents, Event.AvgCost, Comment.NumEventComments, Comment.AvgEventComments, Attend.NumAttendees, Attend.AvgAttendees

FROM /\* Finds the number of ratings and average rating for each Performer. \*/

(SELECT R.Performer AS UserName, COUNT(\*) AS NumRatings, AVG(R.Rating) AS AvgRating

FROM PerformerRating R

GROUP BY R.Performer) Rating,

/\* Finds the number of events and average cost of events for each Performer. \*/

(SELECT E.Performer AS UserName, COUNT(\*) AS NumEvents ,

AVG(E.Cost) AS AvgCost

FROM Event E

GROUP BY E.Performer) Event,

/\* Finds the total and average number of comments of events for each Performer. \*/

(SELECT tmp.UserName AS UserName, SUM(tmp.NumComments) AS

NumEventComments, AVG(tmp.NumComments) AS AvgEventComments

FROM (SELECT E.Performer AS UserName, COUNT(\*) AS NumComments

FROM Event E, Comment C

WHERE E.EventID = C.EventID

GROUP BY E.Performer, E.EventID) tmp

GROUP BY tmp.UserName) Comment,

/\* Finds the number and average number of attendees of events for each Performer. \*/

(SELECT tmp.UserName AS UserName, SUM(tmp.NumAttendees) AS NumAttendees, AVG(tmp.NumAttendees) AS AvgAttendees

FROM (SELECT E.Performer AS UserName, COUNT(\*) AS NumAttendees

FROM Event E, Attend A

WHERE E.EventID = A.EventID

GROUP BY E.Performer, E.EventID) tmp

GROUP BY tmp.UserName) Attend,

Performer P

WHERE P.UserName = Rating.UserName AND P.UserName = Event.UserName AND P.UserName = Comment.UserName AND P.UserName = Attend.UserName;

/\* PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This view pairs venue information with information about overall attendance at venues; allows emails to be sent out about venues and hides passwords for security reasons. \*/

CREATE VIEW VenueInformation(UserName, Email, Attendance)

SELECT U.UserName, U.Email, E.EventID, COUNT(\*)

FROM User U, Venue V, Attend A, Event E

WHERE U.Username = V.UserName

AND A.EventID = E.EventID

AND V.UserName = E.Venue

GROUP BY U.UserName, U.Email, E.EventID;

/\* PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: GEOFF REN, SHAON BOROSHA

This view pairs user information with attendance to show a user’s highly frequented venues (more than 3 times); allows promotional emails to be sent out. Hides passwords for security reasons.

\*/

CREATE VIEW UserPreferences(UserName, Email, Venue, Visits) AS

SELECT U.UserName, U.Email, E.Venue, COUNT(\*)

FROM User U, GeneralUser G, Attend A, Event E

WHERE U.UserName = G.UserName AND G.UserName = A.UserName AND E.EventID =

A.EventID

GROUP BY U.UserName, U.Email, E.Venue

HAVING COUNT(\*) > 3;

##### /\* QUERIES \*/

/\* 1. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query returns performers ordered by their average rating filtered by ratings >= 4 \*/

SELECT R.Performer, AVG(R.Rating)  
FROM PerformerRating R

GROUP BY R.Performer

HAVING AVG(R.Rating) >= 4

ORDER BY AVG(R.Rating);

/\* 2. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query returns all events that have yet to happen. \*/

SELECT \*

FROM Event E

WHERE E.DateTime > CURRENT\_TIMESTAMP;

/\* 3. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query returns all performer ratings that have been responded to by the performer. \*/

SELECT \*

FROM PerformerRating R

WHERE R.ResponseDescription IS NOT NULL;

/\* 4. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query returns all venue ratings that have been responded to by the venue. \*/

SELECT \*

FROM VenueRating R

WHERE R.ResponseDescription IS NOT NULL;

/\* 5. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query returns a sorted list of general users by how many events they’ve attended. \*/

SELECT U.UserName, U.Email, COUNT(\*)

FROM User U, GeneralUser G, Attend A

WHERE U.UserName = G.UserName AND G.UserName = A.UserName

GROUP BY U.UserName, U.Email

ORDER BY COUNT(\*) DESC;

/\* 6. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query performers who usually perform at very expensive events with their average rating. \*/

SELECT E.Performer, R.Avg, AVG(E.Cost)

FROM Event E

(SELECT R.Performer AS Performer, AVG(R.Rating) AS Avg

FROM PerformerRating R

GROUP BY R.Performer) R

WHERE E.Performer = R.Performer

GROUP BY E.Performer, R.Avg

HAVING AVG(E.Cost) > 500;

/\* 7. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query returns performers ordered by the percentage of ratings they respond to \*/

SELECT R.Performer, COUNT(\*) \* 1.0 / Responded.Total

FROM PerformerRating R

(SELECT R.Performer AS Performer, COUNT(\*) AS Total

FROM PerformerRating R

WHERE R.ResponseDescription IS NOT NULL

GROUP BY R.Performer) Responded

WHERE R.Performer = Responded.Performer

GROUP BY R.Performer

ORDER BY COUNT(\*) \* 1.0 / Responded.Total DESC

/\* 8. PRIMARY AUTHORS: GEOFF REN

SECONDARY REVIEWERS: SHAON BOROSHA, SHAMITA NAGALLA

This query will act as a username change. \*/

UPDATE User

SET UserName = ‘NewUserName’

WHERE UserName = ‘OldUserName’;

/\* 9. PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This query returns the average rating of all performers, as rated by other performers, with at least 10 ratings in order of highest to lowest average rating. \*/

SELECT PR.Performer AS Rating, AVG(PR.Rating)

FROM PerformerRating PR

GROUPBY PR.Performer

HAVING COUNT(PR.Rating)>=10

ORDER BY AVG(PR.Rating);

/\* 10. PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This query returns all events with any comments made by users that attended the event. \*/

SELECT A.EventID

FROM Attend A, Comment C

WHERE A.UserName = C.UserName;

/\* 11. PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: SHAON BOROSHA, GEOFF REN

This query returns how many events a user has attended. \*/

SELECT UserName, COUNT(\*)

FROM Attend A

GROUP BY UserName

/\* 12. PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: SHAON BOROSHA, GEOFF REN

This query returns how many people attended each event. \*/

SELECT E.EventID, COUNT(\*)

FROM Event E, Attend A

WHERE A.EventID = E.EventID

GROUP BY A.EventID

/\* 13. PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: GEOFF REN, SHAON BOROSHA

This query returns top level comments for an event ordered by most upvotes, then date. \*/

SELECT \*

FROM Comment C, TopLevel TL

WHERE C.CommentID = TL.CommentID AND

C.EventID = EVENT\_NUM

ORDER BY C.Upvote DESC, C.DateTime DESC

/\* 14. PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: SHAON BOROSHA, GEOFF REN

This query returns all the responses under a top level comment. \*/

SELECT \*

FROM Response R, Comment C

WHERE C.CommentID = R.CommentID AND R.RespondeeID = SPEC\_ID

ORDER BY C.DateTime DESC

/\* 15. PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: SHAON BOROHSA, GEOFF REN

This query returns top 5 performers for each venue. \*/

SELECT tmp.Venue, tmp.Performer, tmp.Rating

FROM (SELECT E.Performer AS Performer, E.Venue AS Venue, AVG(PR.Rating) AS

Rating

FROM PerformerRating PR, Event E

WHERE PR.Performer = E.Performer

GROUP BY E.Performer, E.Venue) tmp

ORDER BY Rating DESC

LIMIT 5

/\* 16. PRIMARY AUTHORS: SHAMITA NAGALLA

SECONDARY REVIEWERS: SHAON BOROHSA, GEOFF REN

This query returns events of a specified genre for a specified venue. \*/

SELECT \*

FROM Event E

WHERE E.Venue = SPEC\_VENUE AND E.Genre = SPEC\_GENRE;

/\* 17. PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This query returns events bookmarked more than 2 times. \*/

SELECT B.EventID

FROM BookMark B

GROUP BY B.EventID

HAVING COUNT(\*) > 2;

/\* 18. PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This query returns the average price of events at the venue with username Acme.\*/

SELECT AVG(E.Cost)

FROM Event E

WHERE E.Venue = ‘Acme’;

/\* 19. PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This query returns users that have attended events but not made any venue ratings or performer ratings. \*/

SELECT UNIQUE A.UserName

FROM Attend A

EXCEPT

SELECT VR.UserName

FROM VenueRating VR

EXCEPT

SELECT PR.UserName

FROM PerformerRating PR;

/\* 20. PRIMARY AUTHORS: SHAON BOROSHA

SECONDARY REVIEWERS: GEOFF REN, SHAMITA NAGALLA

This query returns the number of events a user has attended at one venue. \*/

SELECT A.UserName, E.Venue, COUNT(\*)

FROM Attend A, Event E

WHERE A.EventID = E.EventID

GROUP BY A.UserName, E.Venue;

#### 

##### /\* DROPS \*/

/\* Delete tables if they already exist \*/

DROP TABLE IF EXISTS User;

DROP TABLE IF EXISTS Performer;

DROP TABLE IF EXISTS GeneralUser;

DROP TABLE IF EXISTS Venue;

DROP TABLE IF EXISTS SelfPerformerPhoto;

DROP TABLE IF EXISTS SelfVenuePhoto;

DROP TABLE IF EXISTS PerformerPhoto;

DROP TABLE IF EXISTS VenuePhoto;

DROP TABLE IF EXISTS PerformerRating;

DROP TABLE IF EXISTS VenueRating;

DROP TABLE IF EXISTS BookMark;

DROP TABLE IF EXISTS Attend;

DROP TABLE IF EXISTS Event;

DROP TABLE IF EXISTS Comment;

DROP TABLE IF EXISTS TopLevel;

DROP TABLE IF EXISTS Response;

/\* Delete triggers if they already exist \*/

DROP TRIGGER IF EXISTS DeletePerformerDeleteUser;

DROP TRIGGER IF EXISTS DeleteGeneralUserDeleteUser;

DROP TRIGGER IF EXISTS DeleteVenueDeleteUser;

DROP TRIGGER IF EXISTS UpdatePerformerUpdateUser;

DROP TRIGGER IF EXISTS UpdateGeneralUserUpdateUser;

DROP TRIGGER IF EXISTS UpdateVenueUpdateUser;

DROP TRIGGER IF EXISTS DeleteTopLevelDeleteComment;

DROP TRIGGER IF EXISTS DeleteResponseDeleteComment;

DROP TRIGGER IF EXISTS UpdateTopLevelUpdateComment;

DROP TRIGGER IF EXISTS UpdateResponseUpdateComment;

DROP TRIGGER IF EXISTS DeleteAttendDeleteVenueRating;

DROP TRIGGER IF EXISTS DeleteAttendDeletePerformerRating;

DROP TRIGGER IF EXISTS DeleteAttendedEvents;

/\* Delete views if they already exist \*/

DROP VIEW IF EXISTS VenueAnalyticView;

DROP VIEW IF EXISTS PerformerAnalyticView;

DROP VIEW IF EXISTS UserPreferences;

DROP VIEW IF EXISTS VenueInformation;