# CSC570 NoSQL Databases

# Lab 1, Part 2: Relational Database Using PostgreSQL

## **Part 2: Advanced Queries, Code, Rules**

The tasks we want to complete in this section are to:

* Group similar values,
* Execute code on the server (stored procedures)
* Create custom interfaces using views and rules.

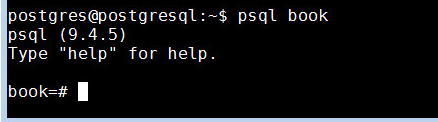
These are some of the features that make Relational Database so flexible in the ways it can query data. Stored procedures can help to move some business intelligence from the main application to the database- the idea is to have enough rules in the database that multiple applications can connect to it safely . Rules, and the ability to re-write them, enable other applications, some of which might be incompatible with the main business application, to use the business data. The database designer’s task is to balance control and openness ( which one depends on the situation).

To get back to where we were at the end of part 1 of the lab:

1. Start your Back-end and Front-end VMs
2. On your FE machine, Double click on the “Turnkey PostgreSQL” Firefox Shortcut on your desktop.
3. Select the “Web Shell” interface. The login for your postgres user is:

Postgresql login: postgres

Password: p@ssw0rd

1. To drop into the Postgres command-line shell and connect to our created DB enter: psql book

All of your prior work should still be present . You can quickly check by seeing if you country table is present by querying the book database with:

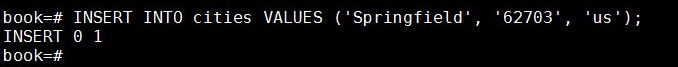
SELECT \* FROM countries;

## Aggregate Functions

An aggregate query groups results from several rows by some common criteria. It can be as simple as counting the number of rows in a table or calculating the average of some numerical column. In order to explore this feature, we need insert some more data into our database.

Let’s enter Sangamon Auditorium as a venue along with several upcoming events.

* 1. Insert Springfield into the *cities* table and Sangamon Auditorium into the *venues* table:



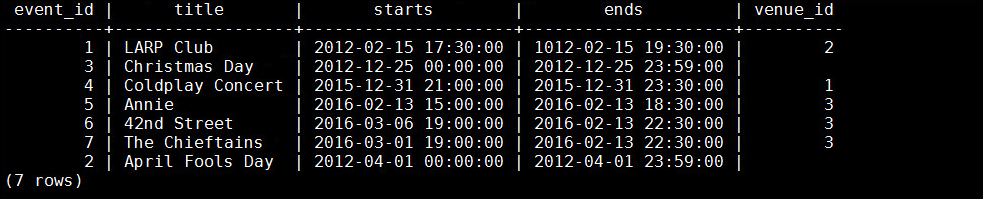


* 1. Add the events to the *events* table:

(Here’s some SQL that will set your venue\_id based on the name rather than trying to remember the number:

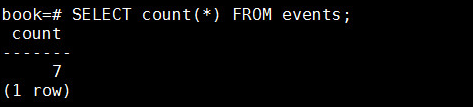
)

Your event table should have the following:

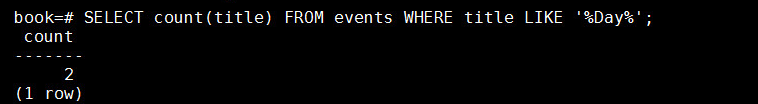


#### Aggregate Function: COUNT()

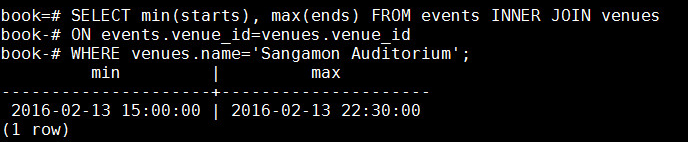
The simplest aggregate function is count(), which returns the integer count of the occurrences of something . Counting the total number of events in our events table should return 7:



A more useful count would be all event titles that contain the string “Day” (note: % is a wildcard on LIKE searches), you should receive a value of 2.



Or, to get the first start time and last end time of all events at the Sangamon Auditorium, use min() (return the smallest value) and max() (return the largest value):



Aggregate functions are useful but limited on their own. If we wanted to count all events at each venue, we could write the following for each venue ID:

SELECT count(\*) FROM events WHERE venue\_id = 1;

SELECT count(\*) FROM events WHERE venue\_id = 2;

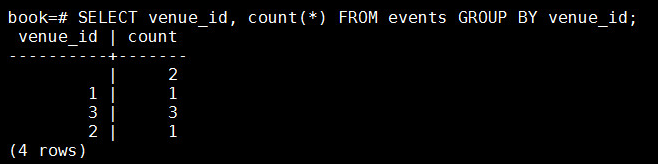
SELECT count(\*) FROM events WHERE venue\_id = 3;

SELECT count(\*) FROM events WHERE venue\_id IS NULL;

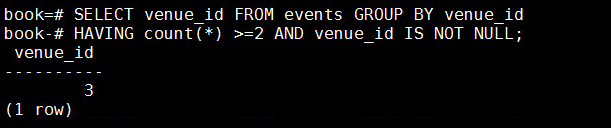
This would be tedious and would become unmanageable as the number of venues grows. Thankfully, there is the GROUP BY aggregate.

#### Aggregate Function: GROUP BY ()

GROUP BY is a shortcut for running the previous queries all at once. With GROUP BY, you tell Postgres to place the rows into groups and then perform some aggregate function (such as count()) on those groups.



GROUP BY can be made more flexible by using a filter. The GROUP BY condition has its own filter keyword: HAVING. HAVING is like the WHERE clause, except it can filter by aggregate functions (whereas WHERE cannot). The following query SELECTs the most popular venues, those with two or more events:



You can use GROUP BY without any aggregate functions. If you call SELECT...FROM...GROUP BY on one column, you get all unique values.

SELECT venue\_id FROM events GROUP BY venue\_id;

This kind of grouping is so common that SQL has a shortcut in the DISTINCTkeyword.

SELECT DISTINCT venue\_id FROM events;

The results of both queries will be identical.

## Transactions

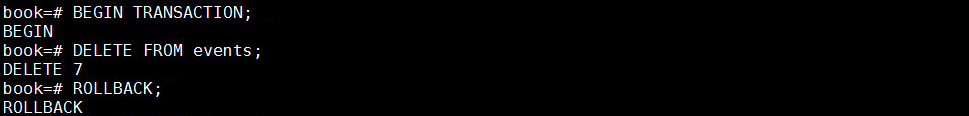
Transactions are the bulwark of relational database consistency. Transactions ensure that every command of a set is executed. If anything fails along the way, all of the commands are rolled back like they never happened. PostgreSQL transactions follow ACID compliance, which stands for Atomic (all ops succeed or none do), Consistent (the data will always be in a good state—no inconsistent states), Isolated (transactions don’t interfere with each other), and Durable (a committed transaction is safe, even after a server crash).

We can wrap any transaction within a BEGIN TRANSACTION block. To verify atomicity, we end the transaction with the ROLLBACK command.

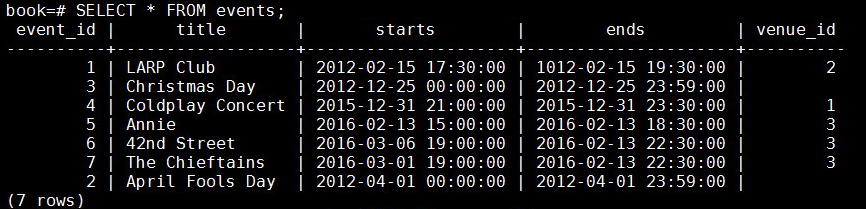
BEGIN TRANSACTION;

DELETE FROM events;

ROLLBACK;



The events all remain. If we run SELECT \* FROM events; all of our data is still there



Transactions are useful when you’re modifying two tables that you don’t want out of sync. The classic example is a debit/credit system for a bank, where money is moved from one account to another:

BEGIN TRANSACTION;

UPDATE account SET total=total+5000.0 WHERE account\_id=1337;

UPDATE account SET total=total-5000.0 WHERE account\_id=45887;

END;

If something happened between the two updates, this bank would lose $5000.00. But, when wrapped in a transaction block, the initial update is rolled back, even if the server crashes.

## Stored Procedures and Triggers

All of the commands we’ve used so far have been declarative, but sometimes you may need to run some code. As a Database Designer, you would now have to make a decision: execute code on the client side or execute code on the database side (Server side).

Stored procedures can offer large performance advantages but at a large architectural costs. You may avoid streaming thousands of rows to a client application, but now you have bound your application code to this database. The decision whether or not to stored procedures is an important one.

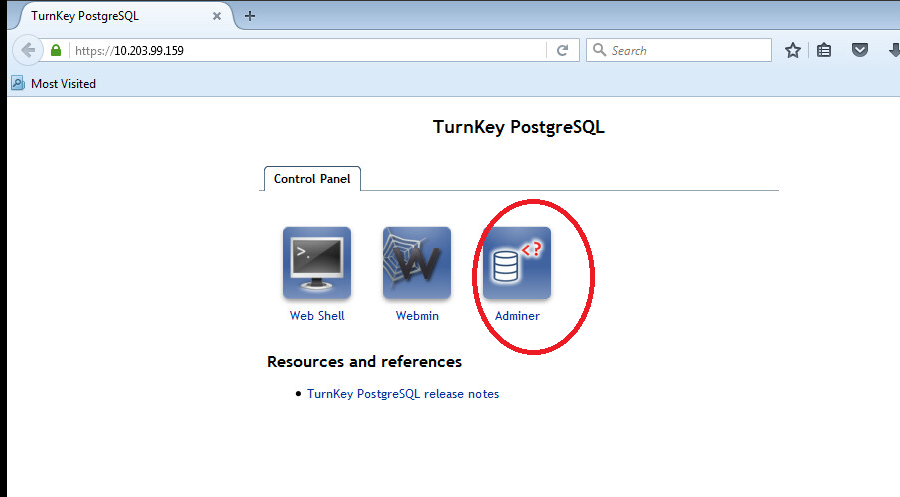
Why use Stored Procedures?

1. Reusability
   1. Avoid rewriting subqueries and improve readability.
   2. If you can't store a query in a library that all the applications can access, you can put that query in a stored procedure.
2. Separation of duties
   1. You don't trust non-DBA's to write queries.
3. Data integrity
   1. Use triggers or constraints to prevent bad data from entering.
   2. Run several interdependent queries in a transaction in a single stored procedure.
4. Event handling
   1. Log changes.
   2. Notify other systems of new data.

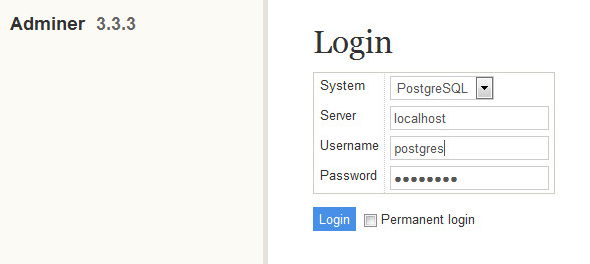
Why NOT use Stored Procedures?

1. Views may be all you need.
   1. An object-relational mapper (ORM) can helpwrite queries safely.
   2. Difficult to version control stored procedures.
   3. Software rollouts may require more db changes.
   4. Could slow software development process

To look at the power of using stored procedures, let’s use a Database Admin tool. Adminer has been installed on your machine and is available from your Mozilla shortcut:



You login with Username: postgres and Password: p@ssw0rd



You should now see all of the databases installed on our server, we want to select our “book”database; double-click on it to see our existing tables:



Admin tools, like Adminer, as typically used by DBAs to manage databases. (We could have done all of our Database CRUD from a tool, but each tool is slightly different . Working from a command line is constant for all installations.

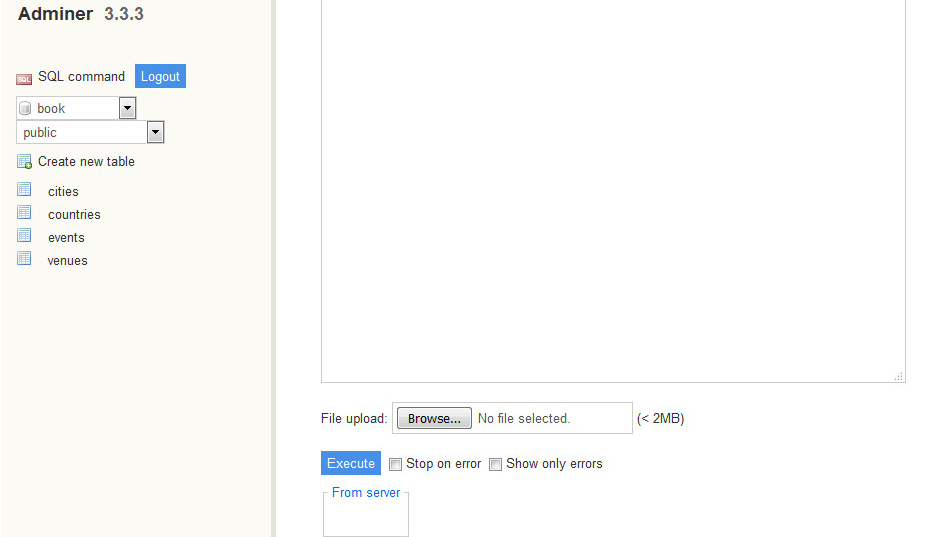
Click on the “SQL command” button in the left-hand pane:



This will open up an text box in which we can enter SQL code:



Scroll to the bottom, there is a place to upload saved code from your FE machine.



Now let’s create a stored procedure (or FUNCTION) that simplifies INSERTing a new event at a venue without needing the venue\_id. Further, if the user tries to enter an event for a venue that doesn’t exist, the procedure will create it first and reference it in the new event. Also, we’ll return a Boolean indicating whether a new venue was added, as a check to let the user know a new venue has been created (and correct the input if that’s not what they wanted).

CREATE OR REPLACE FUNCTION add\_event( title text, starts timestamp,

ends timestamp, venue text, postal varchar(9), country char(2) )

RETURNS boolean AS $$

DECLARE

did\_insert boolean := false;

found\_count integer;

the\_venue\_id integer;

BEGIN

SELECT venue\_id INTO the\_venue\_id

FROM venues v

WHERE v.postal\_code=postal AND v.country\_code=country AND v.name ILIKE venue

LIMIT 1;

IF the\_venue\_id IS NULL THEN

INSERT INTO venues (name, postal\_code, country\_code)

VALUES (venue, postal, country)

RETURNING venue\_id INTO the\_venue\_id;

did\_insert := true;

END IF;

RAISE NOTICE 'Venue found %', the\_venue\_id;

INSERT INTO events (title, starts, ends, venue\_id)

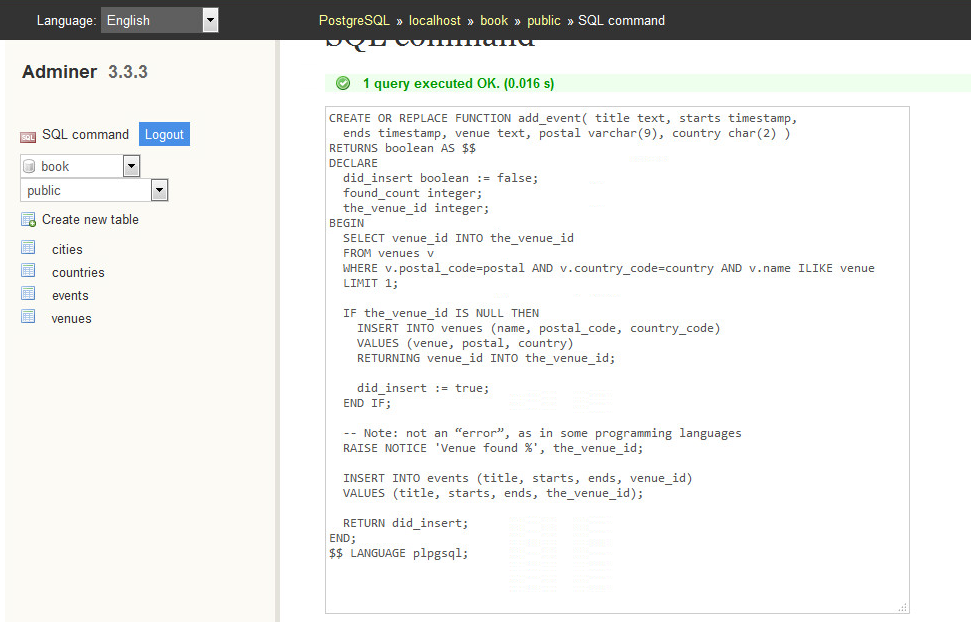
VALUES (title, starts, ends, the\_venue\_id);

RETURN did\_insert;

END;

$$ LANGUAGE plpgsql;

You can type it in the SQL command box in Adminer or download the .sql file from Blackboard onto your Front-end machine. You can then copy from the .sql file into the command window or upload it to Execute. (If you upload it, you will not see the code, it will only tell you that it successfully executed.)



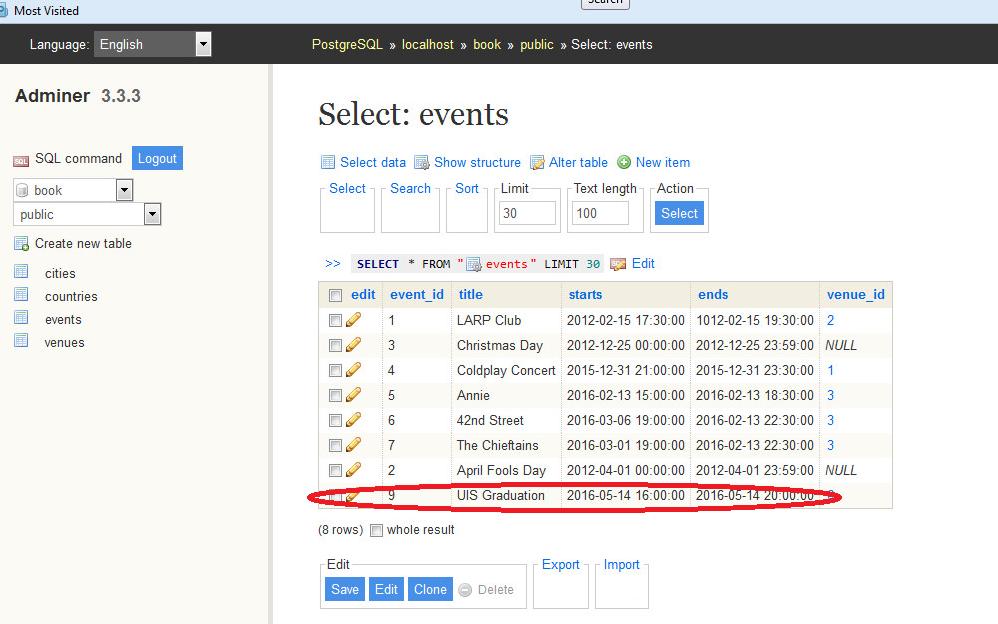
How can we check if our stored procedure works? Let’s use it to add a new event in a new venue: UIS’s May Graduation at the Prairie Capital Convention Center:

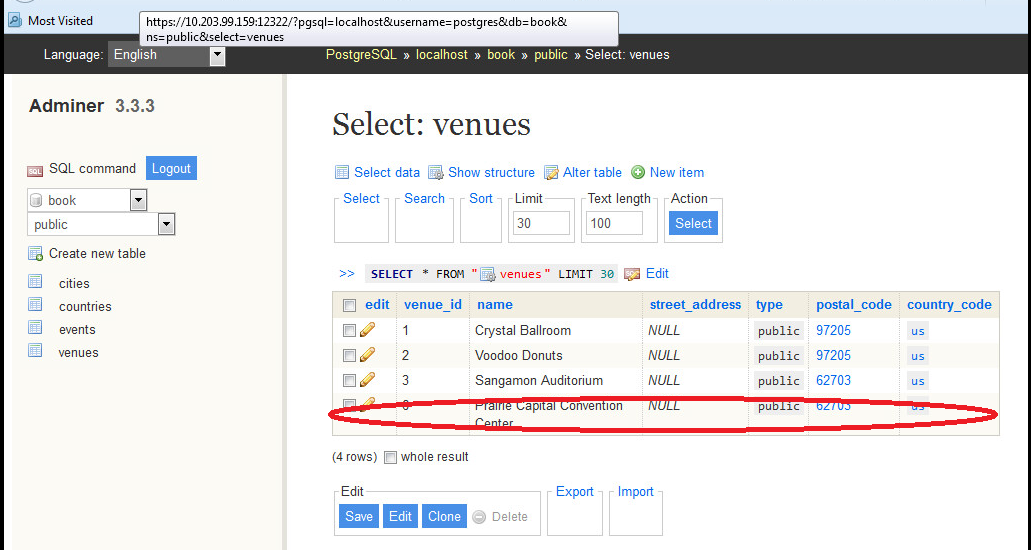
SELECT add\_event('UIS ,, Graduation', '2016-05-14 16:00:00', '2016-05-14 20:00:00', 'Prairie Capital Convention Center', '62703', 'us');

Adminer executes the saved add\_event procedure :

****

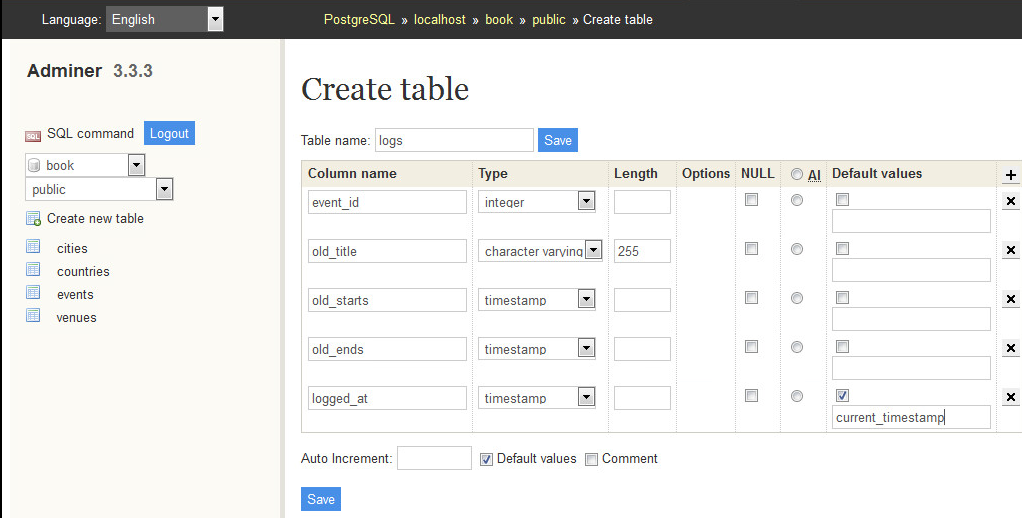
We can examine our tables in adminer to see that the new event and the new venue have been added:





The power of a stored procedure is when they are combined with triggers. Triggers automatically launch stored procedures when some event happens, like an insert or update. They allow the database to automatically enforce some required behavior in response to changing data.

Let’s create a new stored procedure that logs whenever an event is updated (we want a record of anytime someone alters an existing event ). First, create a logs table to store event changes. A primary key isn’t necessary here, since it’s just a log. Click on the “Create new Table” button in the left-hand pane and enter the following in the textboxes:



If you are having trouble reading the textboxes above, I have just entered the options below:

CREATE TABLE logs (

event\_id integer,

old\_title varchar(255),

old\_starts timestamp,

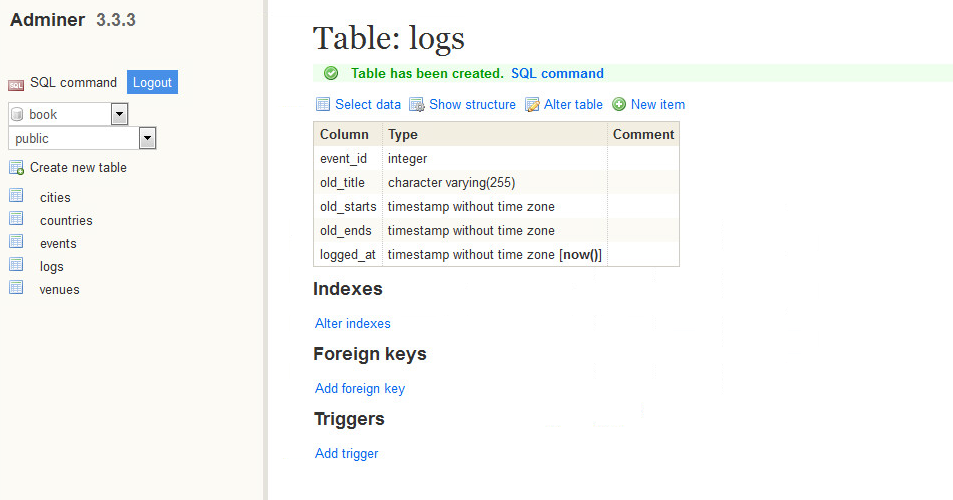
old\_ends timestamp,

logged\_at timestamp DEFAULT current\_timestamp

);

You could also enter the above code in the SQL command window and it will create the table.

If you created it correctly, you should see:



Next, we build a function to insert old data into the log. The OLD variable represents the row about to be changed (NEW represents an incoming row. It will then Output a notice to the console with the event\_id before returning.

**CREATE OR REPLACE** FUNCTION log\_event() RETURNS **trigger AS** $$

DECLARE

**BEGIN**

**INSERT INTO** logs (event\_id, old\_title, old\_starts, old\_ends)

**VALUES** (OLD.event\_id, OLD.title, OLD.starts, OLD.ends);

RAISE NOTICE *'Someone just changed event #%'*, OLD.event\_id;

RETURN NEW;

**END**;

$$ LANGUAGE plpgsql;

Type this into your SQL command window and execute:



Finally, we create our trigger to log changes after any row is updated by entering and executing the following SQL:

CREATE TRIGGER log\_events

AFTER UPDATE ON events

FOR EACH ROW EXECUTE PROCEDURE log\_event();

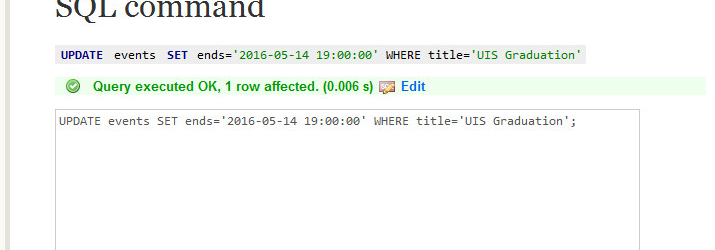


Now let’s test it by updating the time Graduation will end:

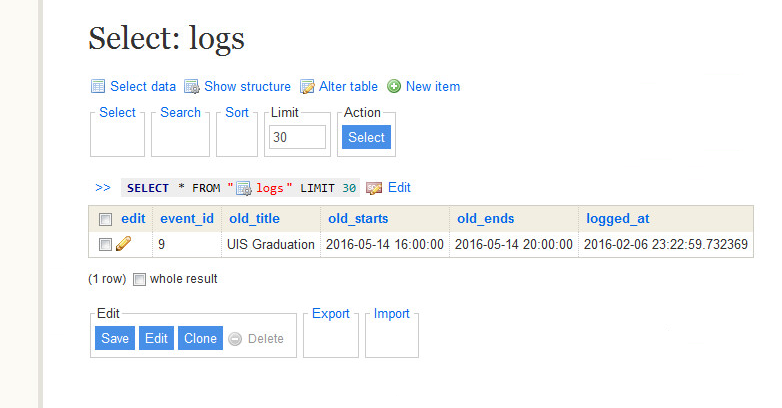
UPDATE events

SET ends='20162-05-14 19:00:00'

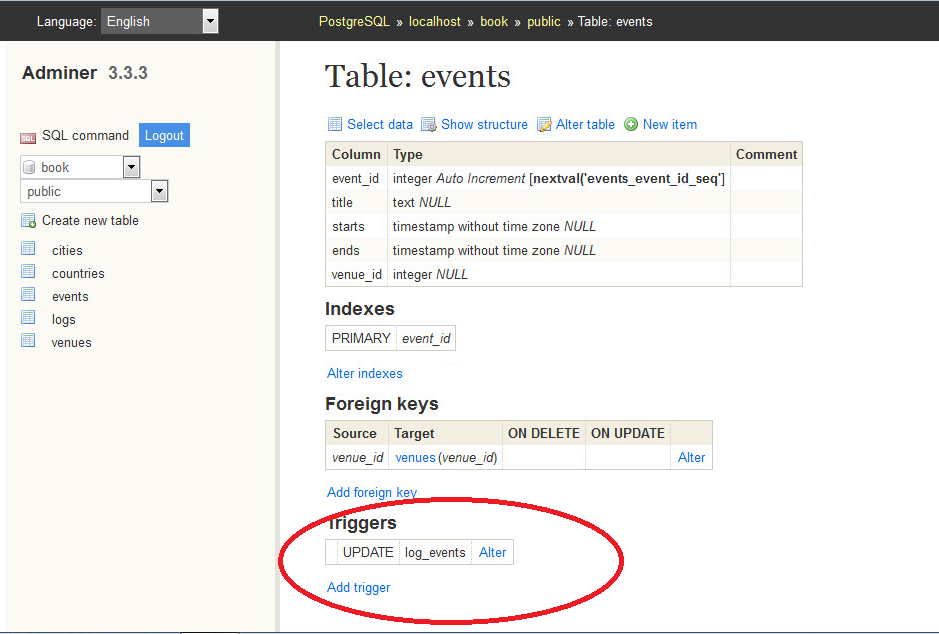
WHERE title='UIS Graduation';



We can see this change in our *logs* table:



Triggers can also be created before updates and before or after inserts. Triggers are tied to the table whose Update or Insert causes its executeion. In our example it is a change on *events*. Adminer stores the information for the trigger with its triggering table:



## Views and Rules

Relational Database allows for the creation of very complex queries. The results are often a subset or union of data that we would like to treat just like any other table. VIEWs allow us to dothat. Unlike stored procedures, these aren’t functions being executed but rather aliased queries. In our current database, all holidays contain the word Day and have no venue. Creating a view on just holidays is as simple as writing a query and prefixing it with CREATE VIEW *some\_view\_name* AS…

CREATE VIEW holidays AS

SELECT event\_id AS holiday\_id, title AS name, starts AS date

FROM events

WHERE title LIKE '%Day%' AND venue\_id IS NULL;



Let’s test it by querying the new holiday view:

SELECT name, to\_char(date, 'Month DD, YYYY') AS date

FROM holidays

WHERE date <= '2012-12-25';



Views are powerful tools for opening up complex queried data in a simple way. If you want to add a new column to the view, it will have to come from the underlying table. Let’s alter the events table to have an array of associated

colors.

ALTER TABLE events

ADD colors text ARRAY;



Since holidays are to have colors associated with them, let’s update the VIEW query to contain the colors array.

CREATE OR REPLACE VIEW holidays AS

SELECT event\_id AS holiday\_id, title AS name, starts AS date, colors

FROM events

WHERE title LIKE '%Day%' AND venue\_id IS NULL;



Now we can add color strings to the holiday of choice.

Unfortunately, we cannot update a view directly. For this we need a RULE. A RULE is a description of how to alter the parsed query tree. Every time Postgres runs an SQL statement, it parses the statement into a query tree (generally called an abstract syntax tree). So, to allow updates against our holidays view, we need to craft a RULE that tells Postgres what to do with an UPDATE. Our rule will capture updates to the holidays view and instead run the update on events, pulling values from the pseudorelations NEW and OLD. NEW functionally acts as the relation containing the values we’re setting, while OLD contains the values we query by.

CREATE RULE update\_holidays AS ON UPDATE TO holidays DO INSTEAD

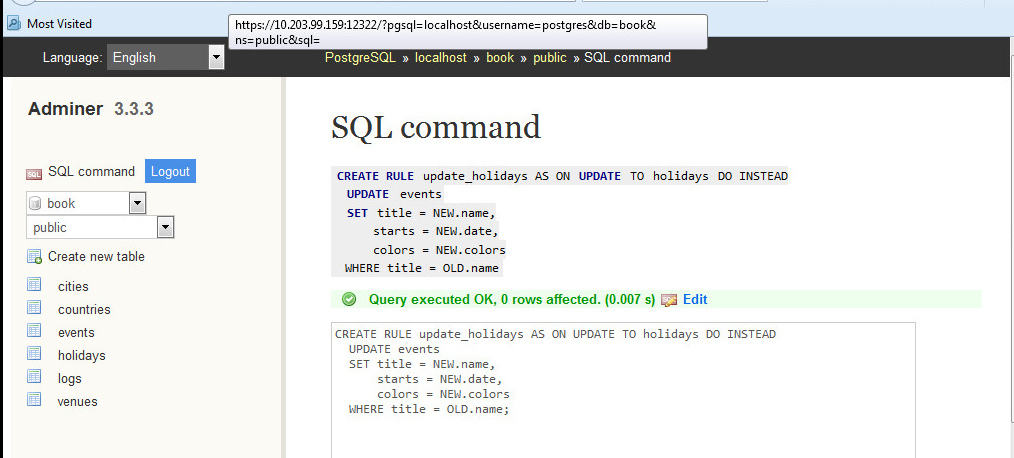
UPDATE events

SET title = NEW.name,

starts = NEW.date,

colors = NEW.colors

WHERE title = OLD.name;

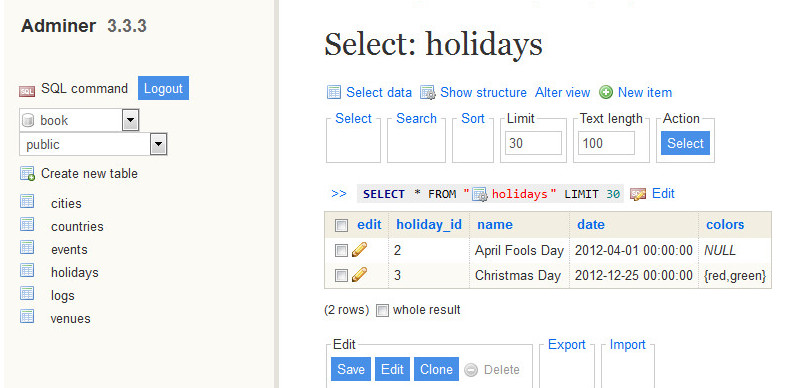
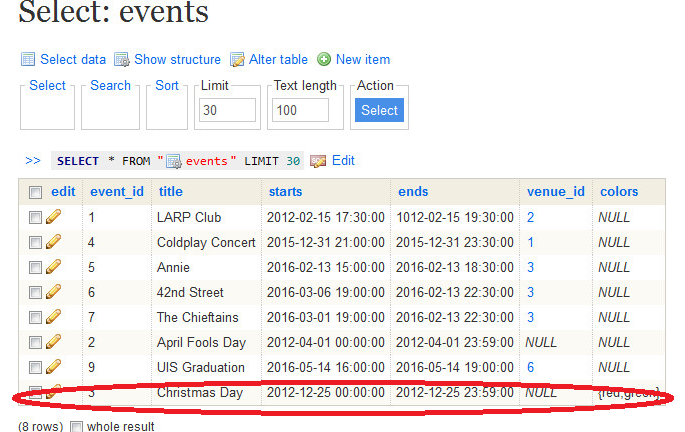


With this rule in place, now we can update holidays directly.

UPDATE holidays SET colors = '{"red","green"}' where name = 'Christmas Day';



Viewing the data in our tables, we can see it worked on both *holidays* and *events*:



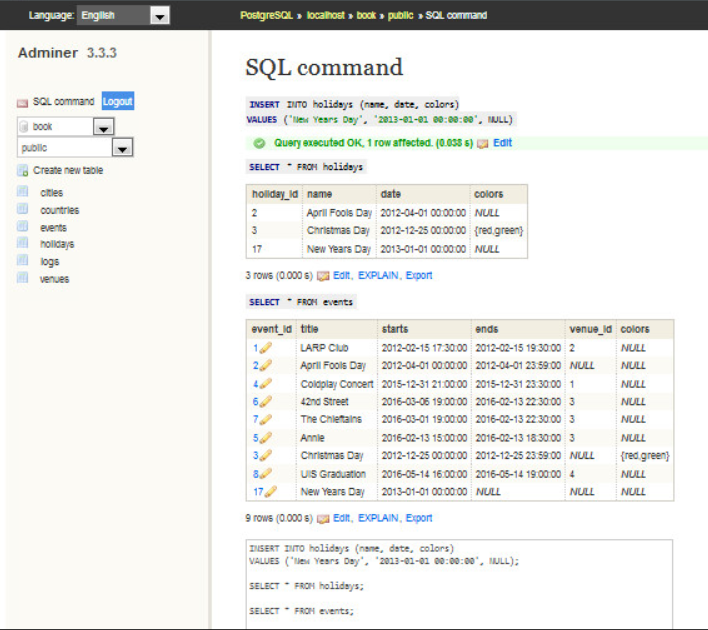
Work on your own to submit

This is a quick lesson on both relational database and the tools that are used by database admins to work with them. The following are some final exercises to let you try these advanced features on your own Take screen shots of your VM output (like in the previous lab steps)and insert these into this word document for submission.

1. Create a new rule, inset\_holidays, and insert New Years Day on 2013-01-01 into holidays. (Remember, it should also insert into the underlying table, events).

*HINT: CREATE RULE insert\_holidays AS ON INSERT TO holidays DO INSTEAD…*

**

**