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).

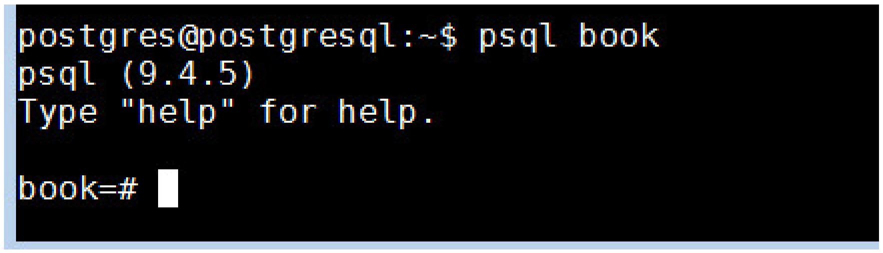
To get back to where we were at the end of part 1 of the lab:

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

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



```
postgres@postgresql:~$ psql book
psql (9.4.5)
Type "help" for help.

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:

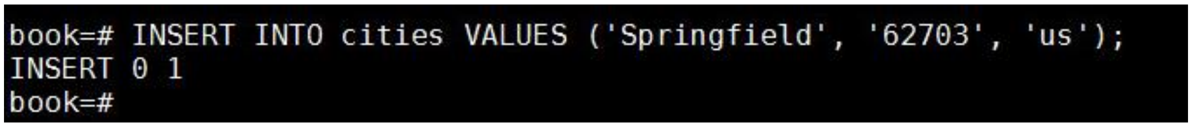
SELECT * FROM countries;

Aggregate Functions

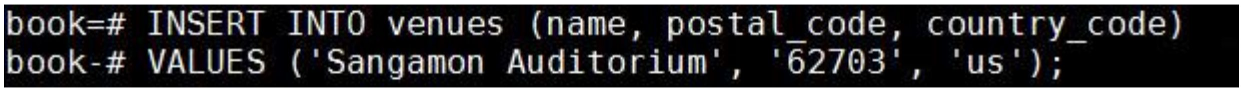
An aggregate query groups results from several rows by some common criteria. It can be as simple as counting the number of rows in a table or calculating the average of some numerical column. In order to explore this feature, we need insert some more data into our database.

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

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



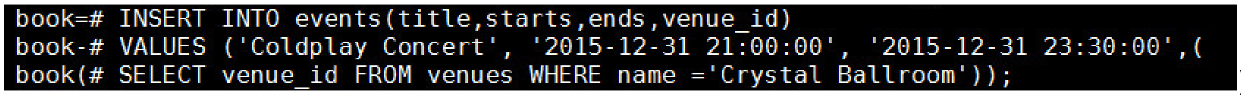
```
book=# INSERT INTO cities VALUES ('Springfield', '62703', 'us');
INSERT 0 1
book=#
```



```
book=# INSERT INTO venues (name, postal_code, country_code)
book-# VALUES ('Sangamon Auditorium', '62703', 'us');
```

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

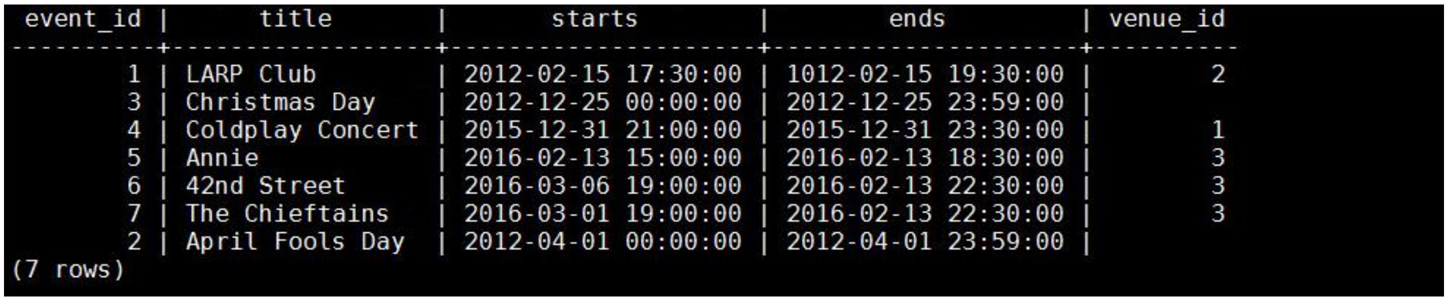
(Here's some SQL that will set your venue_id based on the name rather than trying to remember the number:



```
book=# INSERT INTO events(title,starts,ends,venue_id)
book-# VALUES ('Coldplay Concert', '2015-12-31 21:00:00', '2015-12-31 23:30:00',(
book(# SELECT venue_id FROM venues WHERE name ='Crystal Ballroom'));
```

)

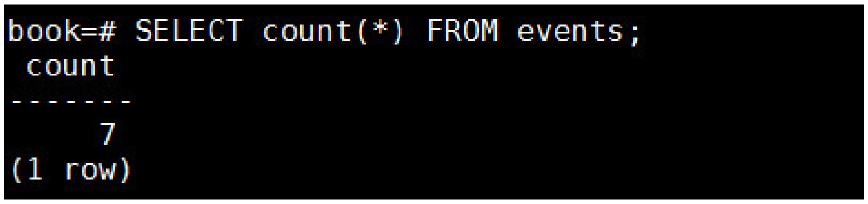
Your event table should have the following:



```
 event_id |      title       |       starts        |        ends         | venue_id
----------+------------------+---------------------+---------------------+----------
        1 | LARP Club        | 2012-02-15 17:30:00 | 1012-02-15 19:30:00 |        2
        3 | Christmas Day    | 2012-12-25 00:00:00 | 2012-12-25 23:59:00 |
        4 | Coldplay Concert | 2015-12-31 21:00:00 | 2015-12-31 23:30:00 |        1
        5 | Annie            | 2016-02-13 15:00:00 | 2016-02-13 18:30:00 |        3
        6 | 42nd Street      | 2016-03-06 19:00:00 | 2016-02-13 22:30:00 |        3
        7 | The Chieftains   | 2016-03-01 19:00:00 | 2016-02-13 22:30:00 |        3
        2 | April Fools Day  | 2012-04-01 00:00:00 | 2012-04-01 23:59:00 |
(7 rows)
```

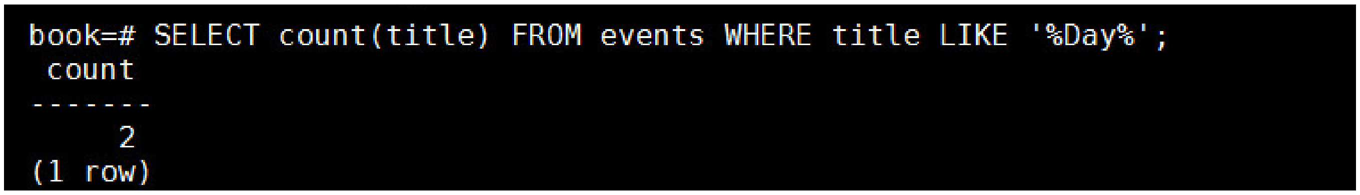
Aggregate Function: COUNT()

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:



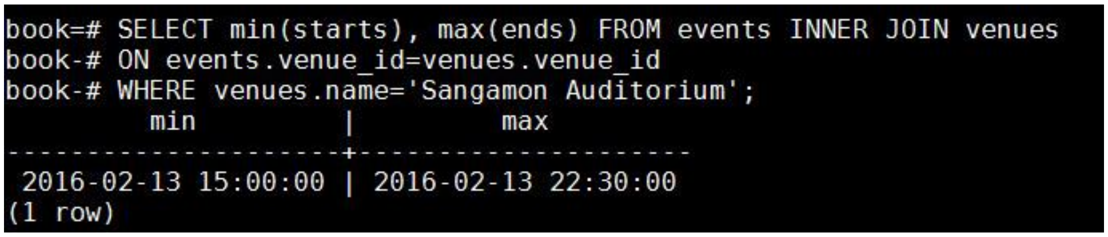
```
book=# SELECT count(*) FROM events;
 count
-------
     7
(1 row)
```

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.



```
book=# SELECT count(title) FROM events WHERE title LIKE '%Day%';
 count
-------
     2
(1 row)
```

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):



```
book=# SELECT min(starts), max(ends) FROM events INNER JOIN venues
book-# ON events.venue_id=venues.venue_id
book-# WHERE venues.name='Sangamon Auditorium';
         min         |         max
---------------------+---------------------
 2016-02-13 15:00:00 | 2016-02-13 22:30:00
(1 row)
```

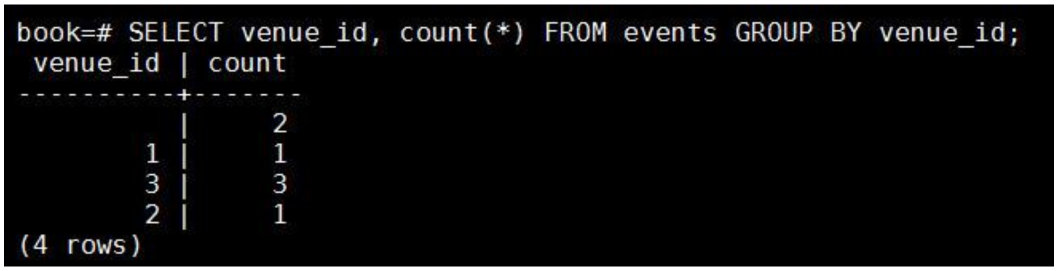
Aggregate functions are useful but limited on their own. If we wanted to count all events at each venue, we could write the following for each venue ID:

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.

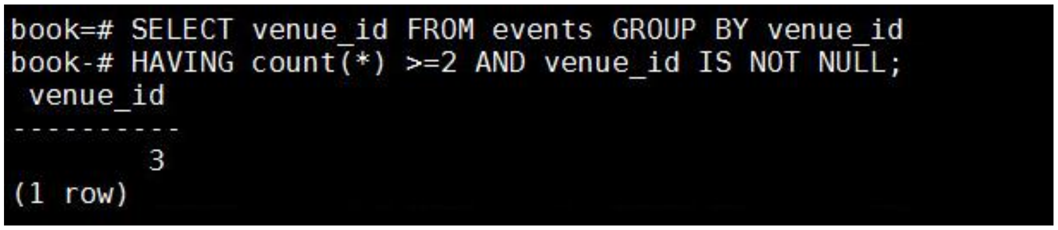
Aggregate Function: GROUP BY ()

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.



```
book=# SELECT venue_id, count(*) FROM events GROUP BY venue_id;
 venue_id | count
----------+-------
          |     2
        1 |     1
        3 |     3
        2 |     1
(4 rows)
```

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:



```
book=# SELECT venue_id FROM events GROUP BY venue_id
book-# HAVING count(*) >=2 AND venue_id IS NOT NULL;
 venue_id
----------
        3
(1 row)
```

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;

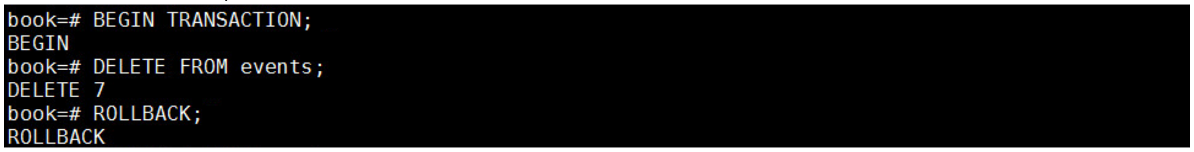
The results of both queries will be identical.

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).

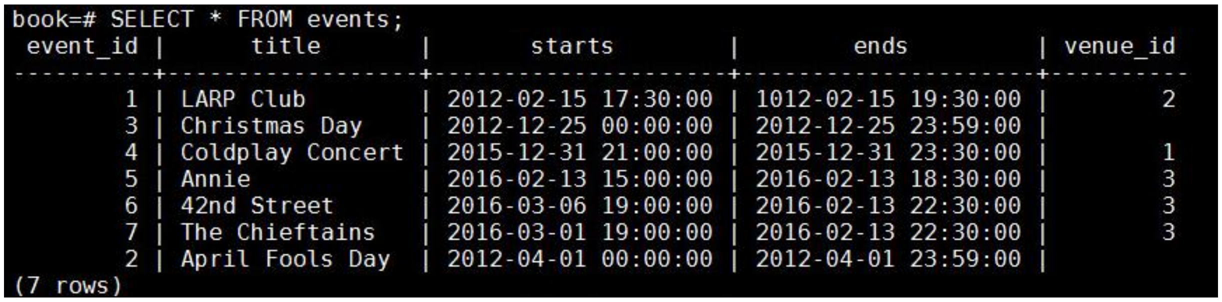
We can wrap any transaction within a BEGIN TRANSACTION block. To verify atomicity, we end the transaction with the ROLLBACK command.

```
BEGIN TRANSACTION;
DELETE FROM events;
ROLLBACK;
```



```
book=# BEGIN TRANSACTION;
BEGIN
book=# DELETE FROM events;
DELETE 7
book=# ROLLBACK;
ROLLBACK
```

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



```
book=# SELECT * FROM events;
 event_id |      title       |       starts        |        ends         | venue_id
----------+------------------+---------------------+---------------------+----------
        1 | LARP Club        | 2012-02-15 17:30:00 | 1012-02-15 19:30:00 |        2
        3 | Christmas Day    | 2012-12-25 00:00:00 | 2012-12-25 23:59:00 |
        4 | Coldplay Concert | 2015-12-31 21:00:00 | 2015-12-31 23:30:00 |        1
        5 | Annie            | 2016-02-13 15:00:00 | 2016-02-13 18:30:00 |        3
        6 | 42nd Street      | 2016-03-06 19:00:00 | 2016-02-13 22:30:00 |        3
        7 | The Chieftains   | 2016-03-01 19:00:00 | 2016-02-13 22:30:00 |        3
        2 | April Fools Day  | 2012-04-01 00:00:00 | 2012-04-01 23:59:00 |
(7 rows)
```

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?

i. Reusability
    a. Avoid rewriting subqueries and improve readability.
    b. 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.

ii. Separation of duties
    a. You don't trust non-DBA's to write queries.

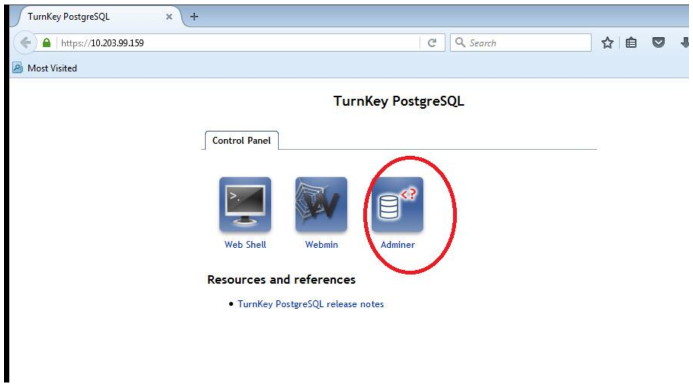
iii. Data integrity
    a. Use triggers or constraints to prevent bad data from entering.
    b. Run several interdependent queries in a transaction in a single stored procedure.

iv. Event handling
    a. Log changes.
    b. Notify other systems of new data.

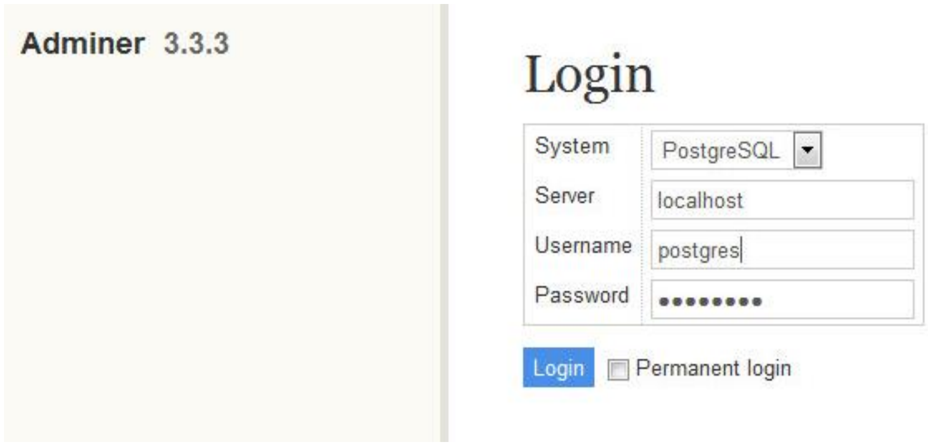
Why NOT use Stored Procedures?

1. Views may be all you need.
    a. An object-relational mapper (ORM) can help write queries safely.
    b. Difficult to version control stored procedures.
    c. Software rollouts may require more db changes.
    d. 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



System	PostgreSQL
Server	localhost
Username	postgres
Password	••••••••

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:

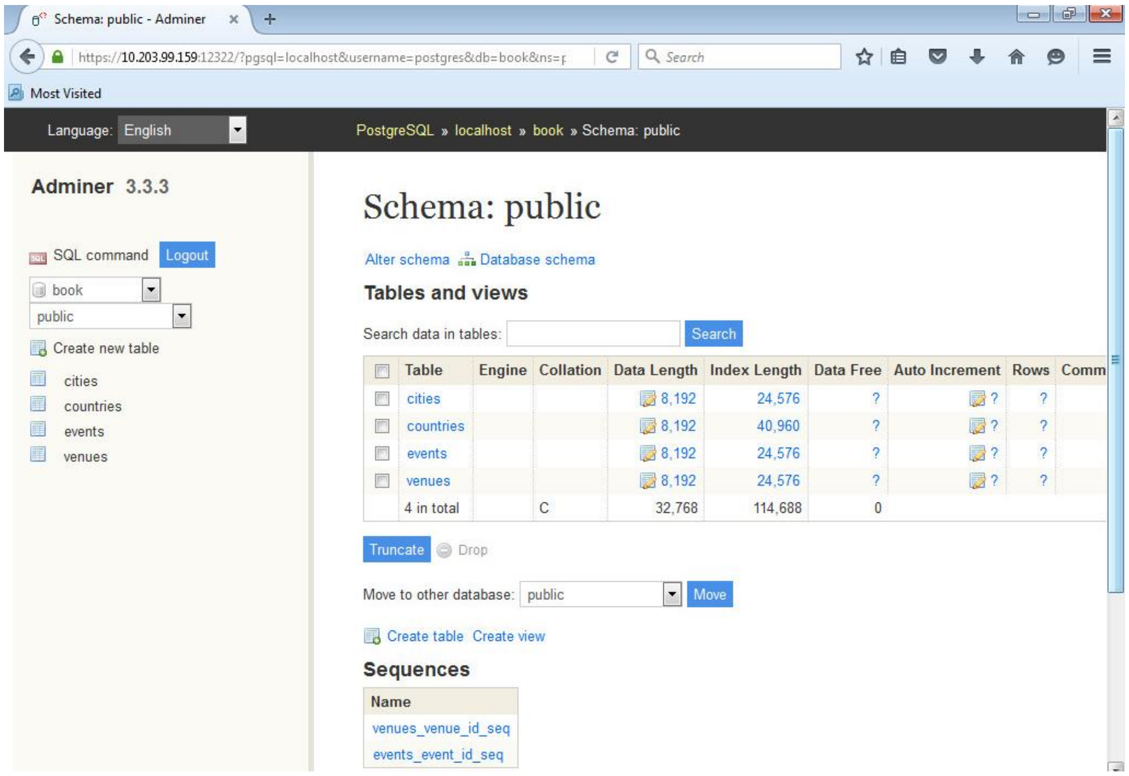


	Table	Engine	Collation	Data Length	Index Length	Data Free	Auto Increment	Rows	Comm
☐	cities			8,192	24,576	?	?	?	
☐	countries			8,192	40,960	?	?	?	
☐	events			8,192	24,576	?	?	?	
☐	venues			8,192	24,576	?	?	?	
	4 in total		C	32,768	114,688	0			

Name
venues_venue_id_seq
events_event_id_seq

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:

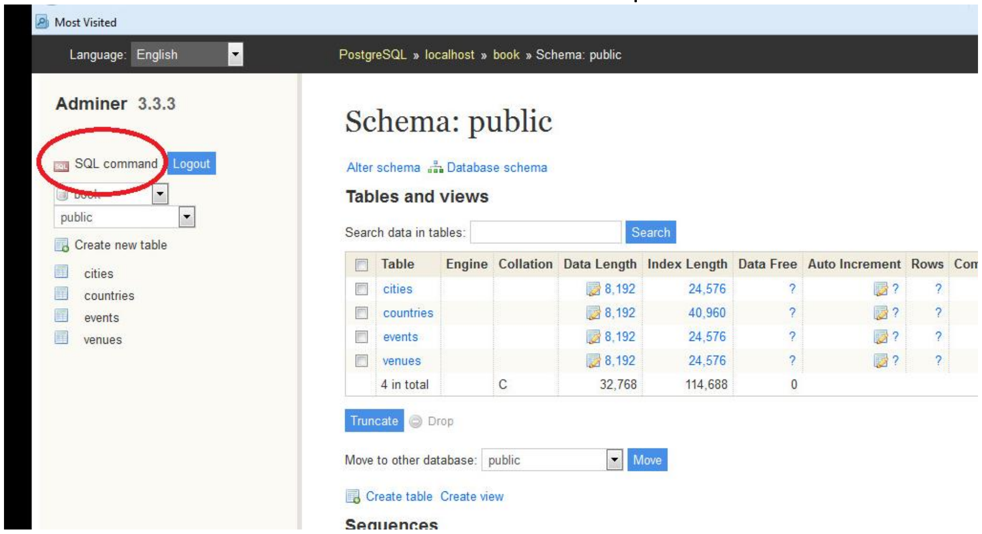
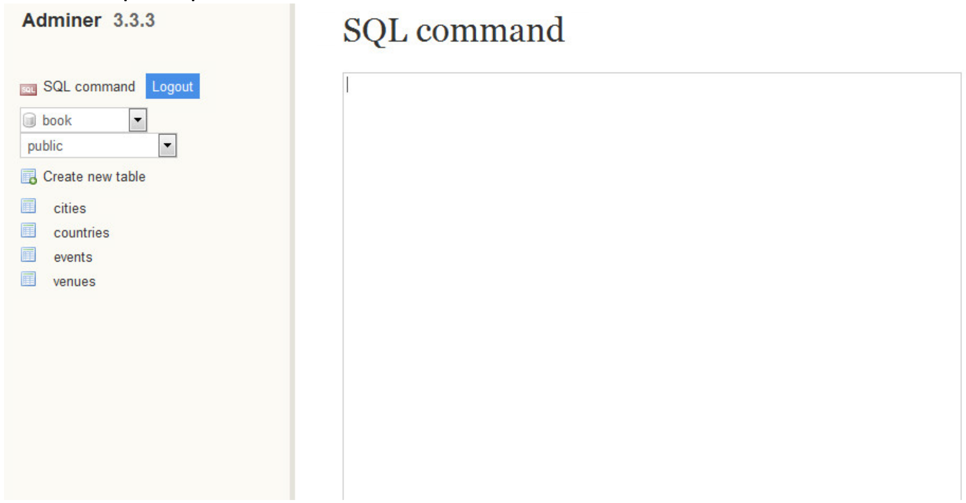
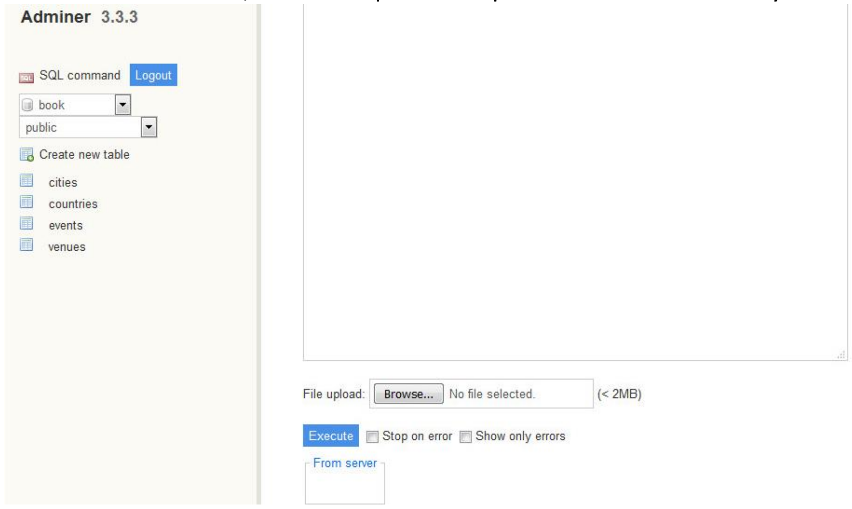


	Table	Engine	Collation	Data Length	Index Length	Data Free	Auto Increment	Rows	Com
☐	cities			8,192	24,576	?	?	?	
☐	countries			8,192	40,960	?	?	?	
☐	events			8,192	24,576	?	?	?	
☐	venues			8,192	24,576	?	?	?	
	4 in total		C	32,768	114,688	0			

This will open up a textbox in which we can enter SQL code:



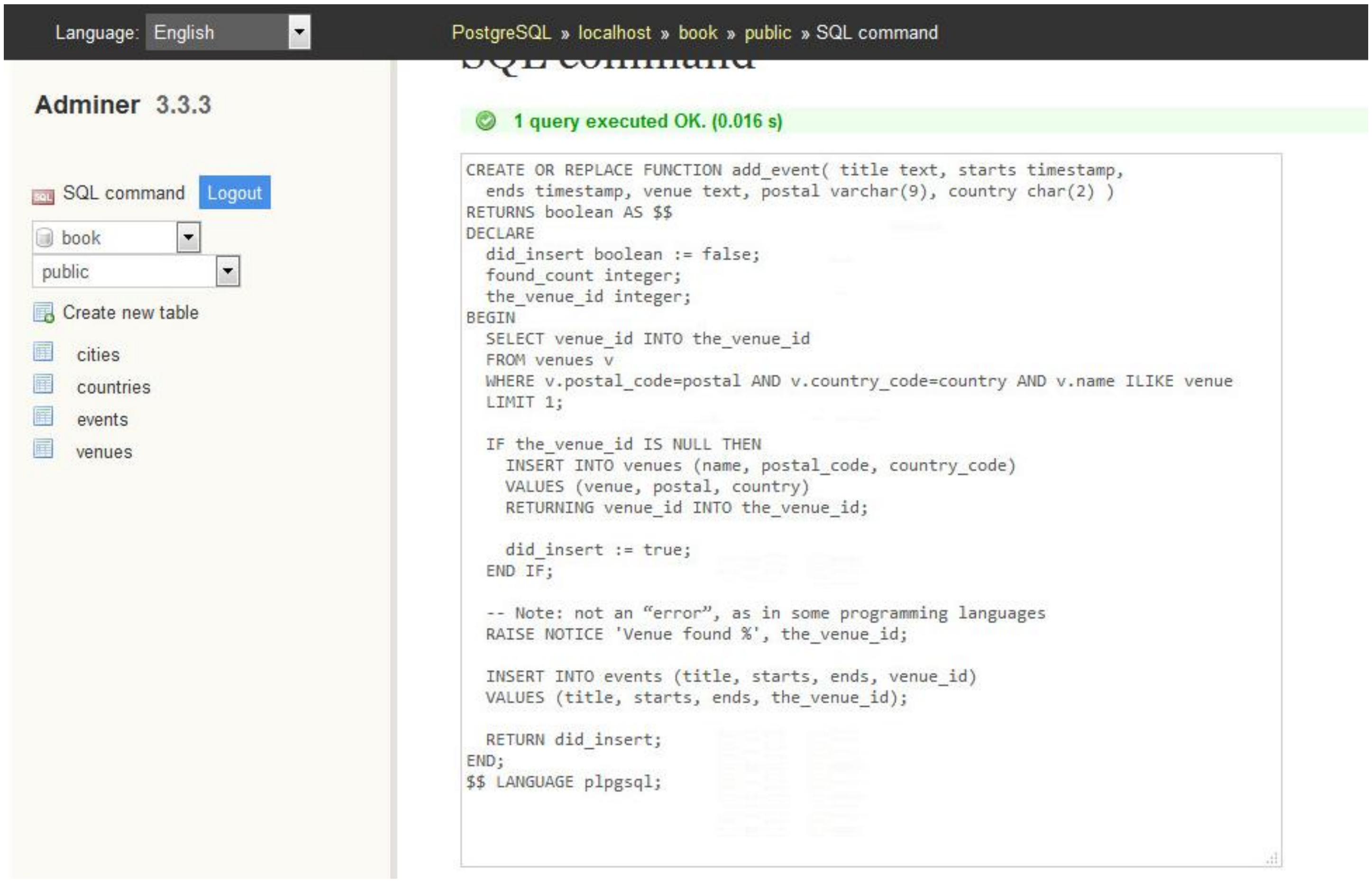
Scroll to the bottom, there is a place to upload saved code from your FE machine.



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.)

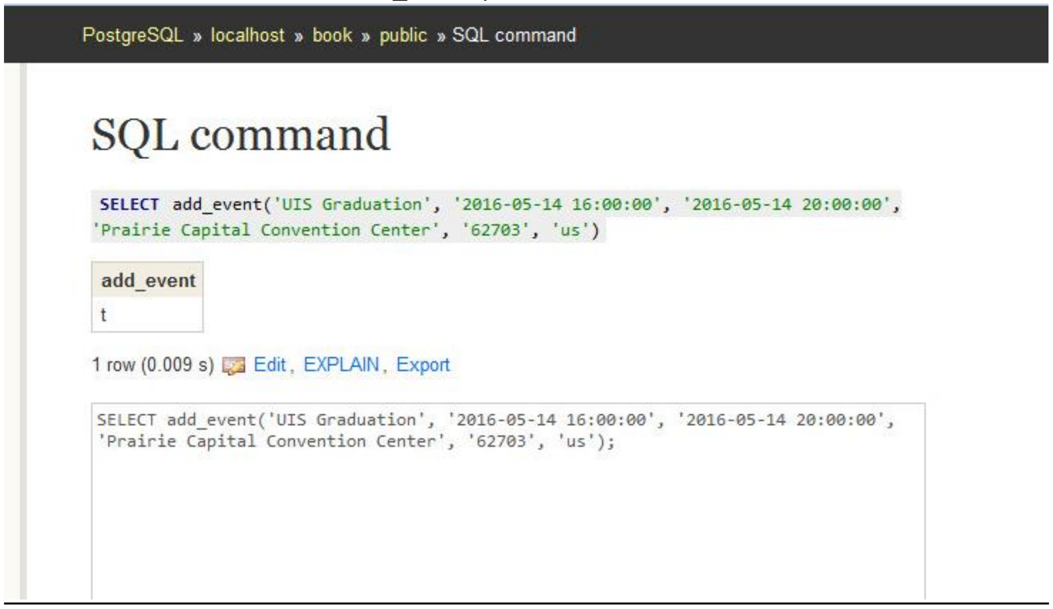


```
CREATE OR REPLACE FUNCTION add_event( title text, starts timestamp,
  ends timestamp, venue text, postal varchar(9), country char(2) )
RETURNS boolean AS $$
DECLARE
  did_insert boolean := false;
  found_count integer;
  the_venue_id integer;
BEGIN
  SELECT venue_id INTO the_venue_id
  FROM venues 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;

  -- Note: not an “error”, as in some programming languages
  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;
```

How can we check if our stored procedure works? Let’s use it to add a new event in a new venue: UIS’ 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:



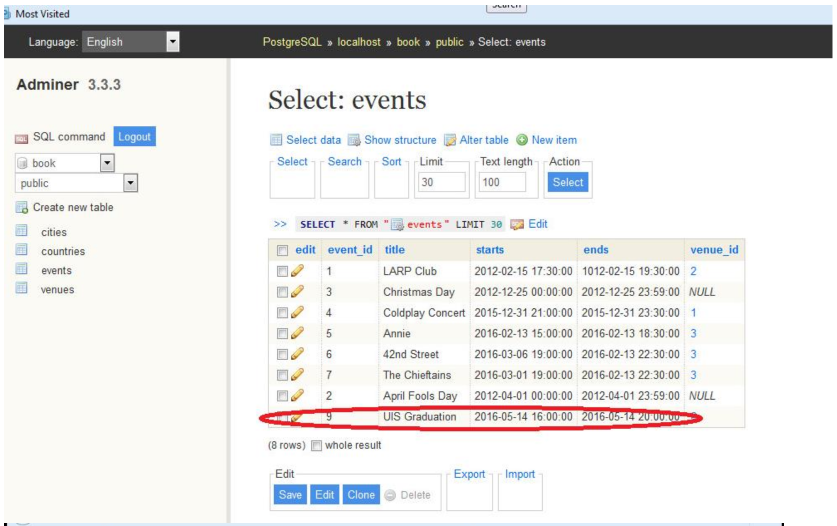
```
SELECT add_event('UIS Graduation', '2016-05-14 16:00:00', '2016-05-14 20:00:00',
'Prairie Capital Convention Center', '62703', 'us')
```

add_event
t

1 row (0.009 s) Edit, EXPLAIN, Export

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

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



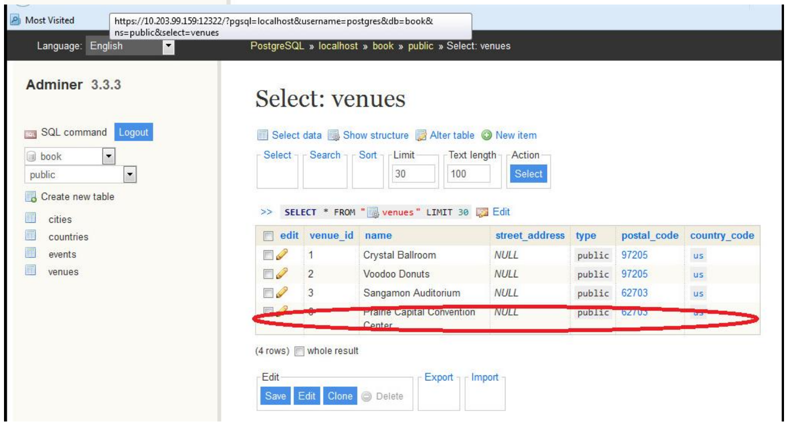
PostgreSQL » localhost » book » public » Select: events

Select: events

```
SELECT * FROM "events" LIMIT 30
```

edit	event_id	title	starts	ends	venue_id
	1	LARP Club	2012-02-15 17:30:00	1012-02-15 19:30:00	2
	3	Christmas Day	2012-12-25 00:00:00	2012-12-25 23:59:00	NULL
	4	Coldplay Concert	2015-12-31 21:00:00	2015-12-31 23:30:00	1
	5	Annie	2016-02-13 15:00:00	2016-02-13 18:30:00	3
	6	42nd Street	2016-03-06 19:00:00	2016-02-13 22:30:00	3
	7	The Chieftains	2016-03-01 19:00:00	2016-02-13 22:30:00	3
	2	April Fools Day	2012-04-01 00:00:00	2012-04-01 23:59:00	NULL
	9	UIS Graduation	2016-05-14 16:00:00	2016-05-14 20:00:00	[illegible]

(8 rows)



https://10.203.99.159:12322/?pgsql=localhost&username=postgres&db=book&ns=public&select=venues

PostgreSQL » localhost » book » public » Select: venues

Select: venues

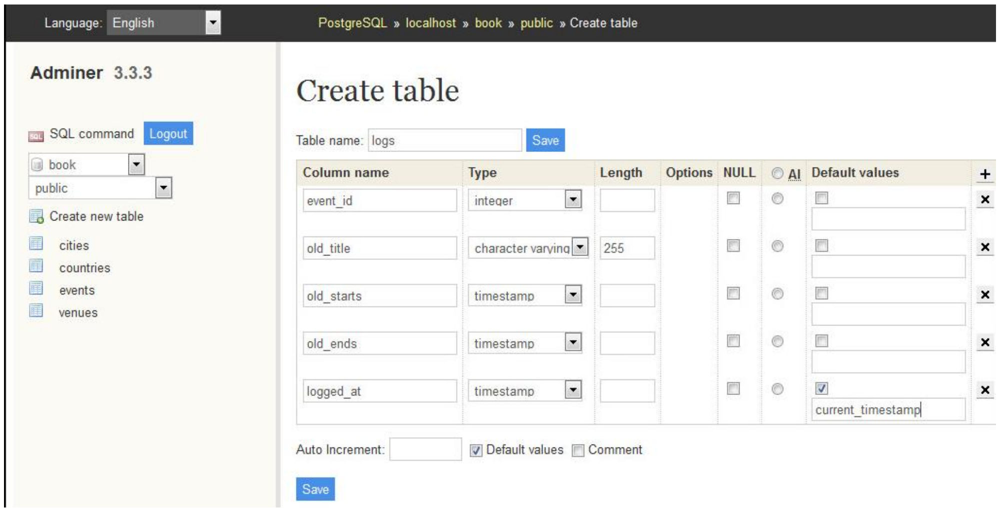
```
SELECT * FROM "venues" LIMIT 30
```

edit	venue_id	name	street_address	type	postal_code	country_code
	1	Crystal Ballroom	NULL	public	97205	us
	2	Voodoo Donuts	NULL	public	97205	us
	3	Sangamon Auditorium	NULL	public	62703	us
	[illegible]	Prairie Capital Convention Center	NULL	public	62703	us

(4 rows)

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:



PostgreSQL » localhost » book » public » Create table

Create table

Table name: logs

Column name	Type	Length	Options	NULL	AI	Default values
event_id	integer					
old_title	character varying	255				
old_starts	timestamp					
old_ends	timestamp					
logged_at	timestamp					✓ current_timestamp

Auto Increment: ✓ Default values ☐ Comment

If you are having trouble reading the textboxes above, I have 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:

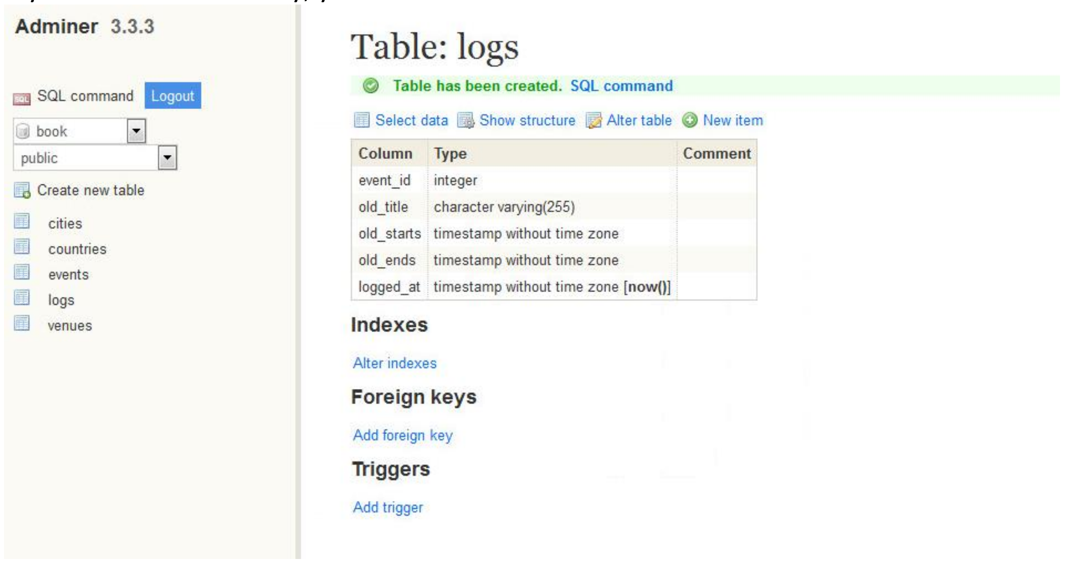


Table: logs

Table has been created. SQL command

Column	Type	Comment
event_id	integer	
old_title	character varying(255)	
old_starts	timestamp without time zone	
old_ends	timestamp without time zone	
logged_at	timestamp without time zone [now()]	

Indexes

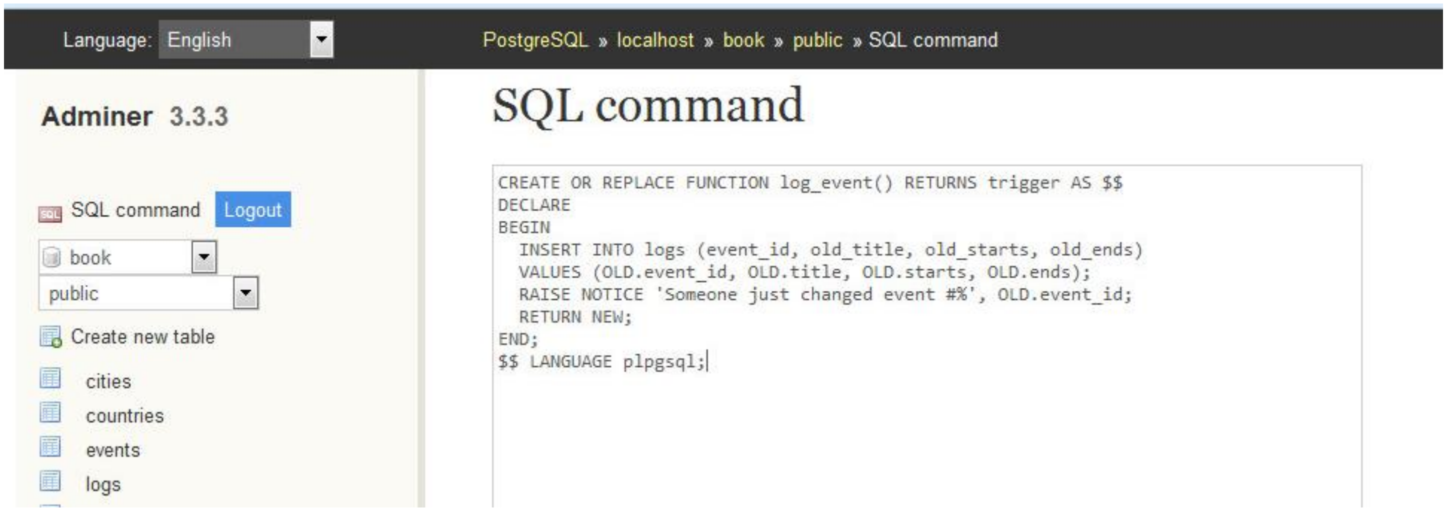
Foreign keys

Triggers

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:

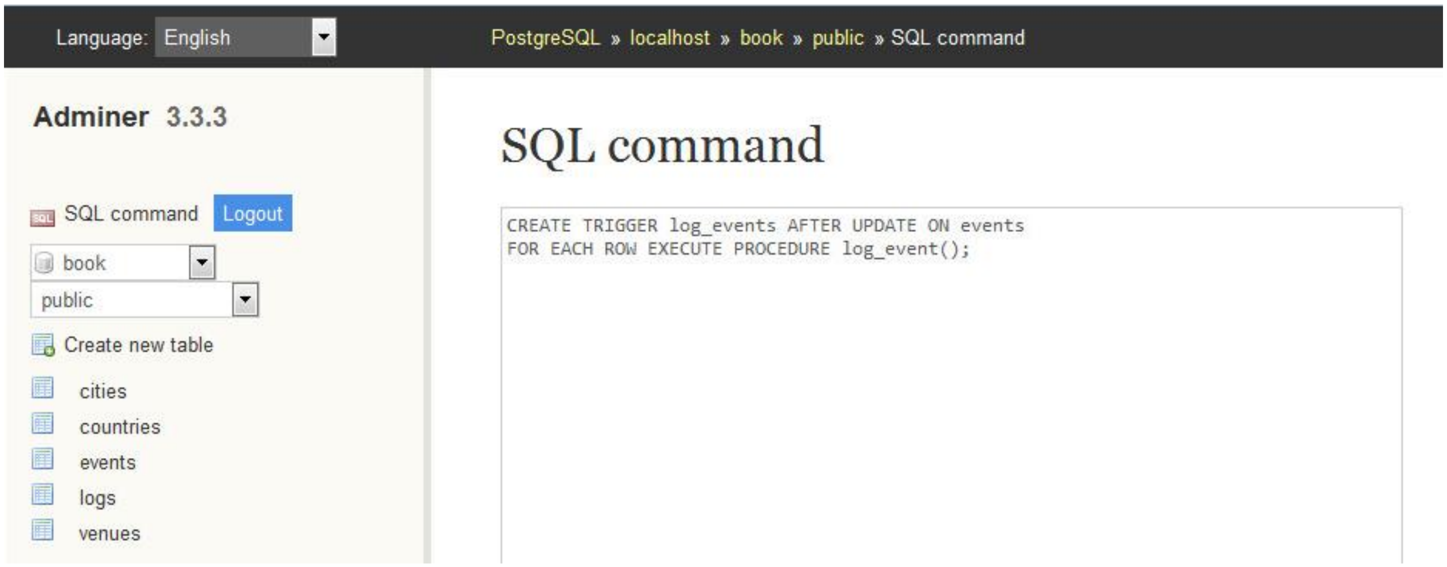


SQL command

```
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;
```

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();
```

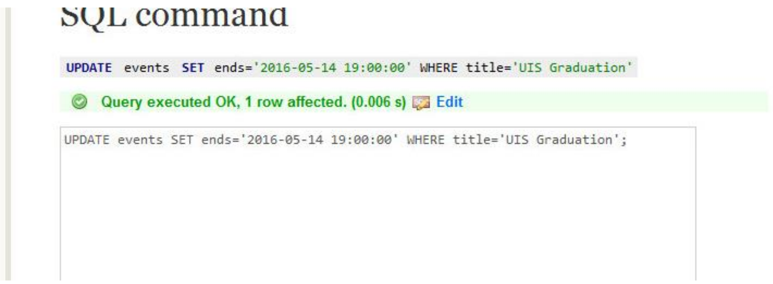


SQL command

```
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';
```



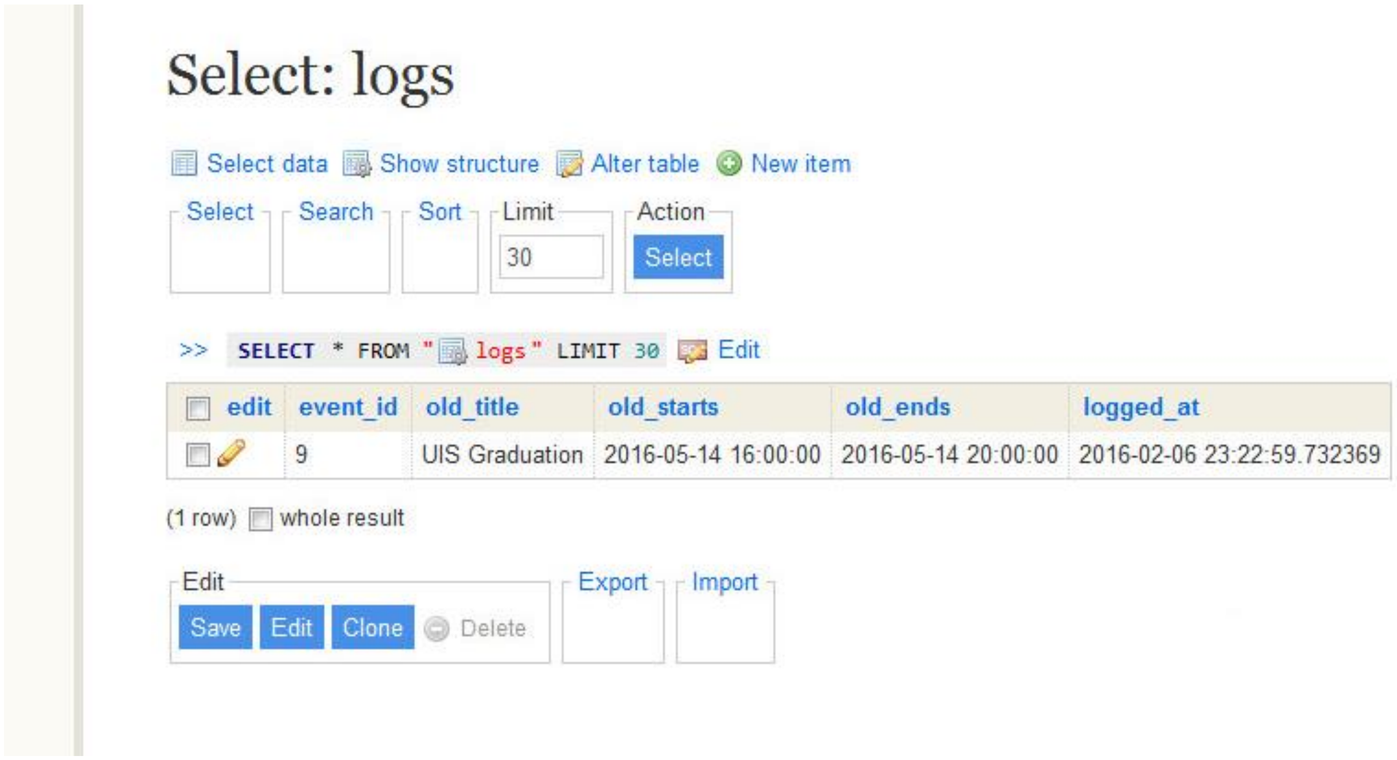
SQL command

```
UPDATE events SET ends='2016-05-14 19:00:00' WHERE title='UIS Graduation'
```

Query executed OK, 1 row affected. (0.006 s) Edit

```
UPDATE events SET ends='2016-05-14 19:00:00' WHERE title='UIS Graduation';
```

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



Select: logs

Limit: 30

```
>> SELECT * FROM "logs" LIMIT 30
```

edit	event_id	old_title	old_starts	old_ends	logged_at
	9	UIS Graduation	2016-05-14 16:00:00	2016-05-14 20:00:00	2016-02-06 23:22:59.732369

(1 row) whole result

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

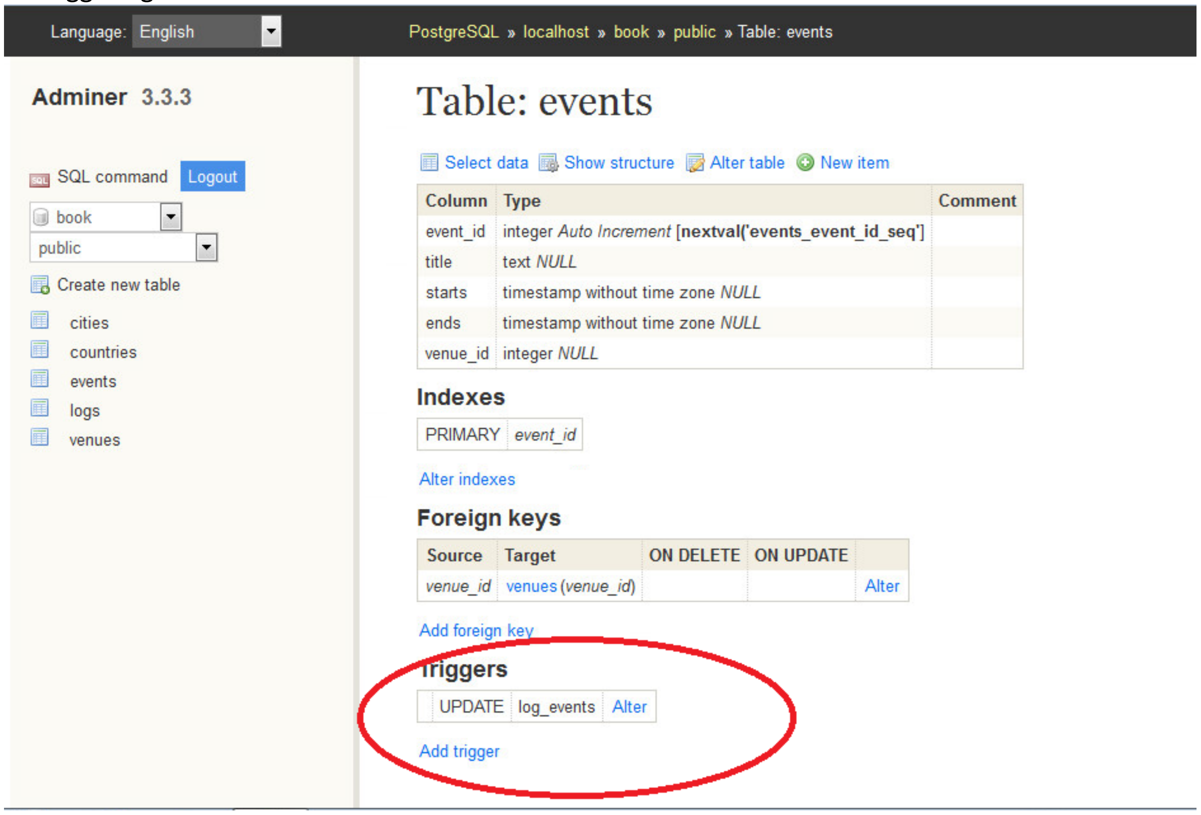


Table: events

Column	Type	Comment
event_id	integer *Auto Increment* [nextval('events_event_id_seq')]	
title	text *NULL*	
starts	timestamp without time zone *NULL*	
ends	timestamp without time zone *NULL*	
venue_id	integer *NULL*	

Indexes

PRIMARY	*event_id*

Foreign keys

Source	Target	ON DELETE	ON UPDATE	
venue_id	venues (*venue_id*)			Alter

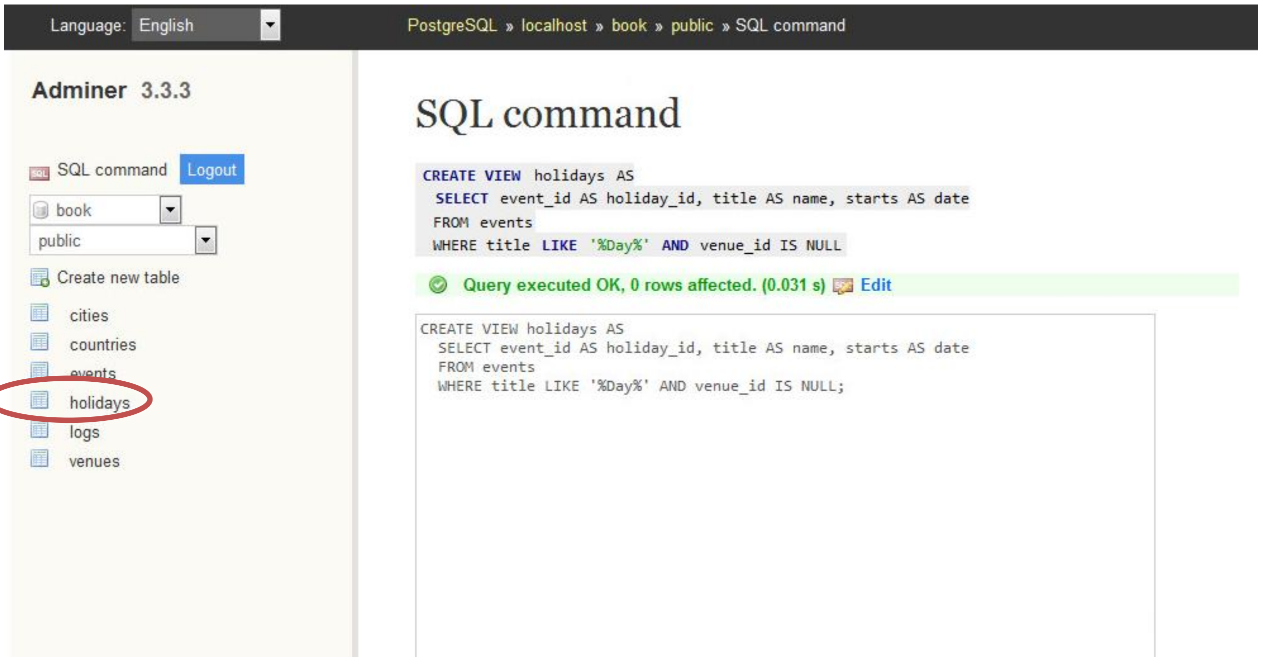
Triggers

UPDATE	log_events	Alter

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 do that. 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;
```



SQL command

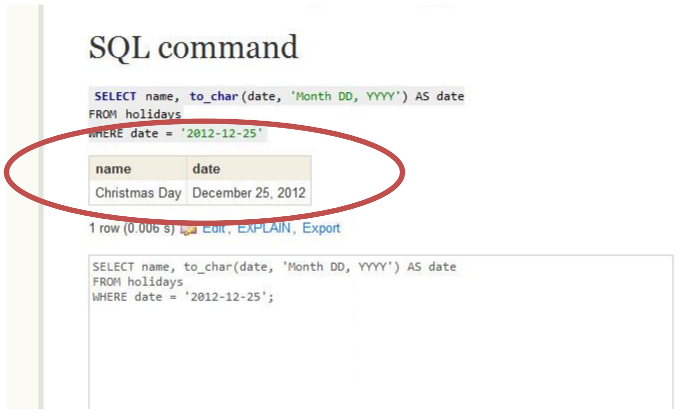
```
CREATE VIEW holidays AS
 SELECT event_id AS holiday_id, title AS name, starts AS date
 FROM events
 WHERE title LIKE '%Day%' AND venue_id IS NULL
```

Query executed OK, 0 rows affected. (0.031 s) Edit

```
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';
```



SQL command

```
SELECT name, to_char(date, 'Month DD, YYYY') AS date
FROM holidays
WHERE date = '2012-12-25'
```

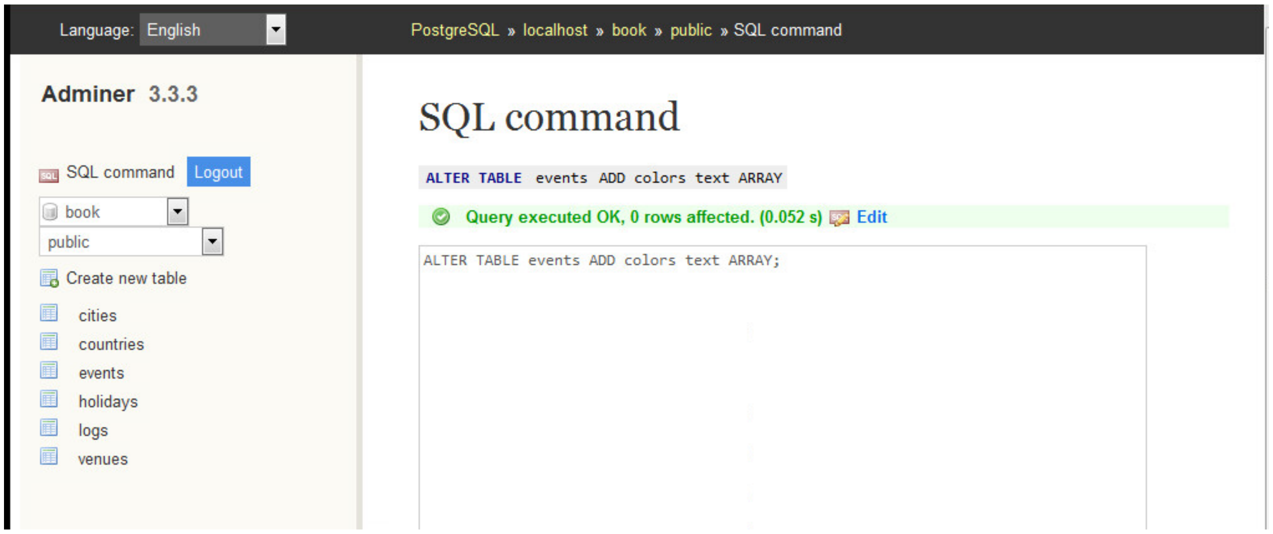
name	date
Christmas Day	December 25, 2012

1 row (0.006 s) Edit, EXPLAIN, Export

```
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;



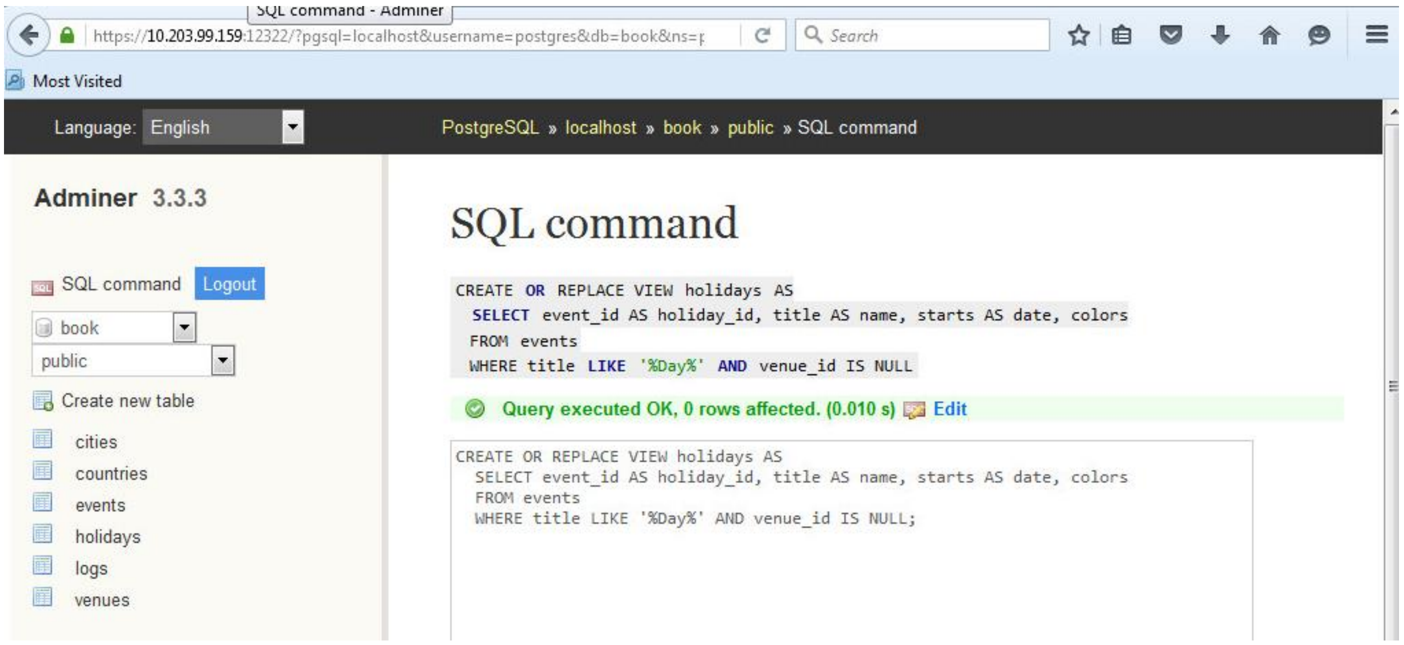
```
ALTER TABLE events ADD colors text ARRAY
```

Query executed OK, 0 rows affected. (0.052 s) Edit

```
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;



```
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
```

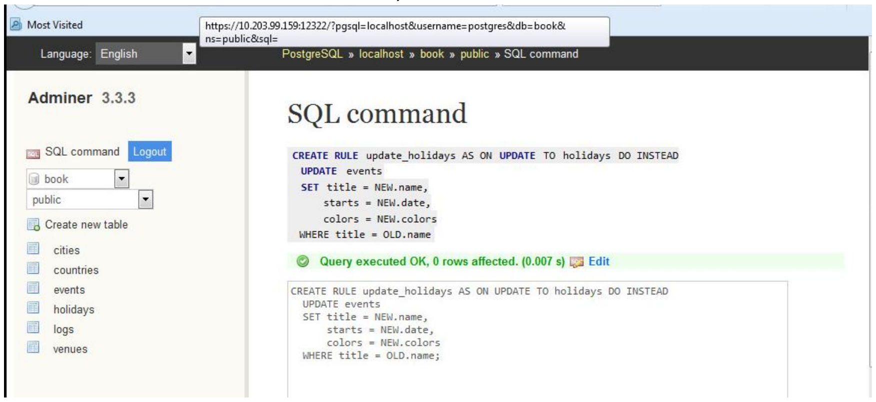
Query executed OK, 0 rows affected. (0.010 s) Edit

```
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;
```


Most Visited
https://10.203.99.159:12322/?pgsql=localhost&username=postgres&db=book&ns=public&sql=
Language: English
PostgreSQL » localhost » book » public » SQL command
Adminer 3.3.3
SQL command
Logout
book
public
Create new table
cities
countries
events
holidays
logs
venues

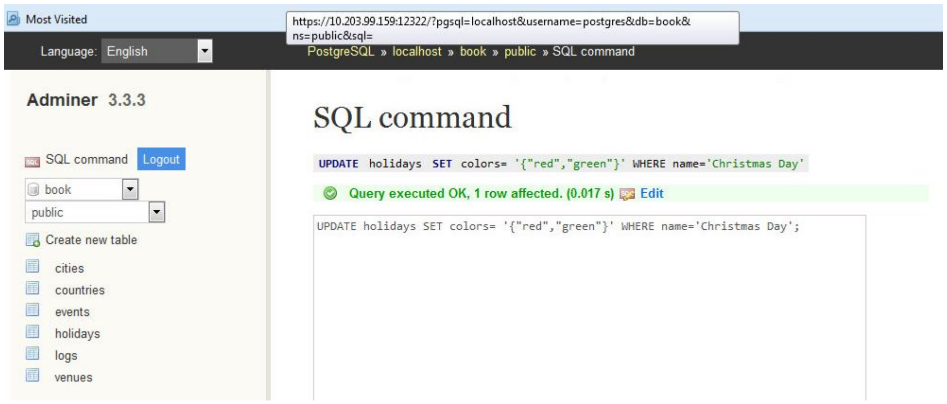
SQL command

```
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
```

Query executed OK, 0 rows affected. (0.007 s) Edit

```
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';


Most Visited
https://10.203.99.159:12322/?pgsql=localhost&username=postgres&db=book&ns=public&sql=
Language: English
PostgreSQL » localhost » book » public » SQL command
Adminer 3.3.3
SQL command
Logout
book
public
Create new table
cities
countries
events
holidays
logs
venues

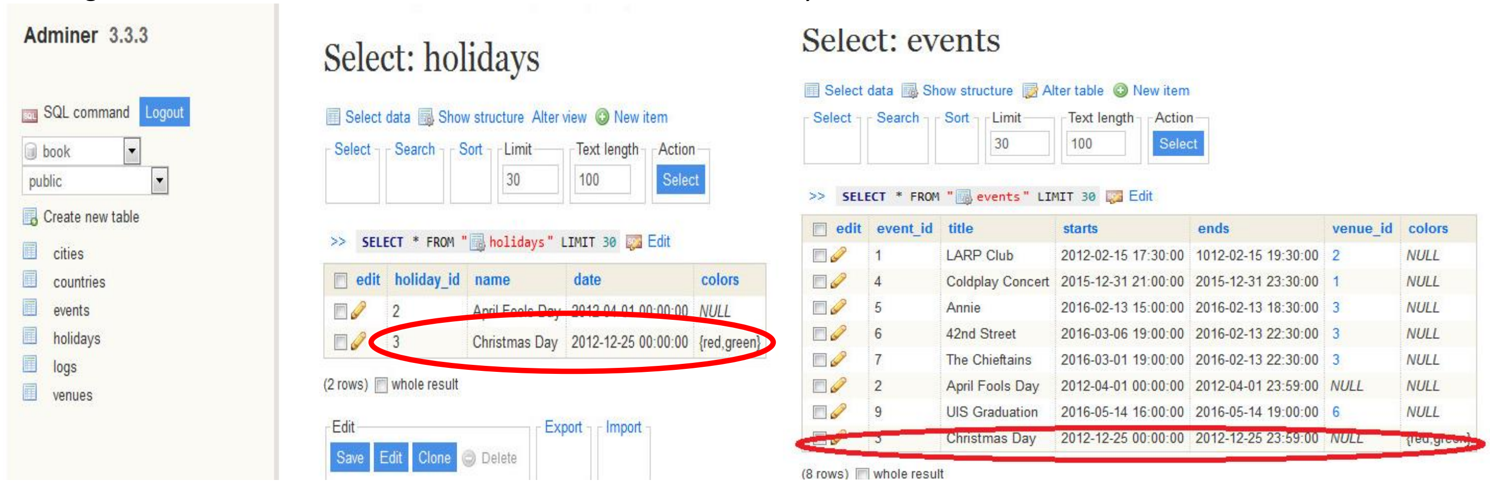
SQL command

```
UPDATE holidays SET colors= '{"red","green"}' WHERE name='Christmas Day'
```

Query executed OK, 1 row affected. (0.017 s) Edit

```
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*:



Select: holidays

Limit 30, Text length 100

```
>> SELECT * FROM "holidays" LIMIT 30 Edit
```

edit	holiday_id	name	date	colors
	2	April Fools Day	2012-04-01 00:00:00	NULL
	3	Christmas Day	2012-12-25 00:00:00	{red,green}

(2 rows) whole result

Select: events

Limit 30, Text length 100

```
>> SELECT * FROM "events" LIMIT 30 Edit
```

edit	event_id	title	starts	ends	venue_id	colors
	1	LARP Club	2012-02-15 17:30:00	1012-02-15 19:30:00	2	NULL
	4	Coldplay Concert	2015-12-31 21:00:00	2015-12-31 23:30:00	1	NULL
	5	Annie	2016-02-13 15:00:00	2016-02-13 18:30:00	3	NULL
	6	42nd Street	2016-03-06 19:00:00	2016-02-13 22:30:00	3	NULL
	7	The Chieftains	2016-03-01 19:00:00	2016-02-13 22:30:00	3	NULL
	2	April Fools Day	2012-04-01 00:00:00	2012-04-01 23:59:00	NULL	NULL
	9	UIS Graduation	2016-05-14 16:00:00	2016-05-14 19:00:00	6	NULL
	3	Christmas Day	2012-12-25 00:00:00	2012-12-25 23:59:00	NULL	{red,green}

(8 rows) whole result

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.

1. Views

   To submit your work, add this code at the bottom of the file for Week 1 (Lab 1 Part 1). The file should be named lab1.sql and uploaded to GitHub.

   Your code should be lightly commented. Use -- to add comments (double dashes) that explain each step listed below. This is good practice for when you share code with colleagues so that they can contribute to the project.

   a. Using the tables created in Lab 1 Part 1, insert 20 transactions. Three of these transactions need to have the actual_checkin_time after the scheduled_checkin_time. This will allow you to test the view you will be creating in the next steps. For example, a transaction where the scheduled_checkin_time is 2018-08-01 14:39:53 and the actual_checkin_time is 2018-08-02 14:39:53. Additionally, five of the transactions need to have a checkout_time after September 3 2018.

   b. Create a late checkins view of distinct items that were checked in late grouped by user_id, inventory_id, and description. This view should display the total number of late checkins per device per user. For example, if user1 checked in two items late, there should be two rows displayed for user1 and each row should include the total number of times that user returned that particular item late.

   c. Test the late checkins view by selecting and displaying all records from the view.

2. The tasks we want to complete in the second part of Lab 1 Part 2 are to:

   a. On our PostgreSQL VM, run the lab1.sql file from Step 1 to create the tables and data.
   b. Access our PostgreSQL databases from a Laravel PHP app. Laravel is a PHP framework.
   c. Execute a query from Laravel using Laravel's Eloquent Object-Relational Mapping (ORM).
   d. Display our query results on a webpage.

   Utilizing a database might be all that you need in the real world, but accessing that database from a programming language such as PHP will give you an advantage in the workplace. Knowing how to retrieve, manipulate and save data from a programming language is not a new feature of any programming language or database, so it will be helpful to understand how to set-up and perform some basic tasks, such as the ones we will cover now:

   1. Install Visual Studio code integrated development environment (IDE). You can use other PHP IDE's such as Atom.
   2. Map a drive to the Windows share on the container with the Laravel code
   3. Open up the Laravel project in the Visual Studio code IDE
   4. Modify the Laravel controller and view to execute the queries using the ORM relationships and display the data in the lab

Video Tutorials

Please review these video lectures:
Lab Setup
[9 min 33 sec]:
https://cdnapisec.kaltura.com/index.php/extwidget/preview/partner_id/1371761/uiconf_id/31473632/entry_id/1_n4lu2j8q/embed/dynamic

Access from Programming Language (PHP/Laravel Eloquent ORM)
[42 min 31 sec]:
https://cdnapisec.kaltura.com/index.php/extwidget/preview/partner_id/1371761/uiconf_id/31473632/entry_id/1_yx21kgc5/embed/dynamic

Please be sure to follow the steps in the Lab Setup video linked above to connect to your container. Use this table to determine which container is yours. You will log into the share with .\NetID for the username (.\tllos1 for example) and your UIN for the password.

Netid	Windows share	Url of the PHP application
agang2	\\10.64.3.56\agang2	https://csc570e.uis.edu:9444
bbala5	\\10.64.3.56\bbala5	https://csc570e.uis.edu:9445
bguti6	\\10.64.3.56\bguti6	https://csc570e.uis.edu:9446
brodr22	\\10.64.3.56\brodr22	https://csc570e.uis.edu:9447
chick7	\\10.64.3.56\chick7	https://csc570e.uis.edu:9448
eunsik2	\\10.64.3.56\eunsik2	https://csc570e.uis.edu:9449
jlund6	\\10.64.3.56\jlund6	https://csc570e.uis.edu:9450
jshei3	\\10.64.3.56\jshei3	https://csc570e.uis.edu:9451
mpavl3	\\10.64.3.56\mpavl3	https://csc570e.uis.edu:9452
rsayy2	\\10.64.3.56\rsayy2	https://csc570e.uis.edu:9453
sarya7	\\10.64.3.56\sarya7	https://csc570e.uis.edu:9454
skoch7	\\10.64.3.56\skoch7	https://csc570e.uis.edu:9455
szhen6	\\10.64.3.56\szhen6	https://csc570e.uis.edu:9456
zwold2	\\10.64.3.56\zwold2	https://csc570e.uis.edu:9457
smeka6	\\10.64.3.56\smeka6	https://csc570e.uis.edu:9458

Relationships will need to be set before writing the queries (hasMany, belongsTo, etc). For Step 1, you will need to create a relationship from the Users table to the Transactions table, from the Transactions table to the Users table, and from the Transactions table to the Inventory table. For Step 2, you will need to create a relationship from the Transactions table to the User table and from the Transactions table to the Inventory table.

Your results will be the view transaction.index, which already maps to the route of the root of the website. This view will display two tables with the data similar to what I showed in the video example for Status and Inventory (see video link above for Access from Programming Language).

1. Write a query to show all items that user1 has checked out with the corresponding checkout_time and display the results.

2. Write a query to show all users who checked out items before September 3 2018 and display the results, which should include the user first name, the item description, the checkout_time, and the status.