

Joshua Catoe

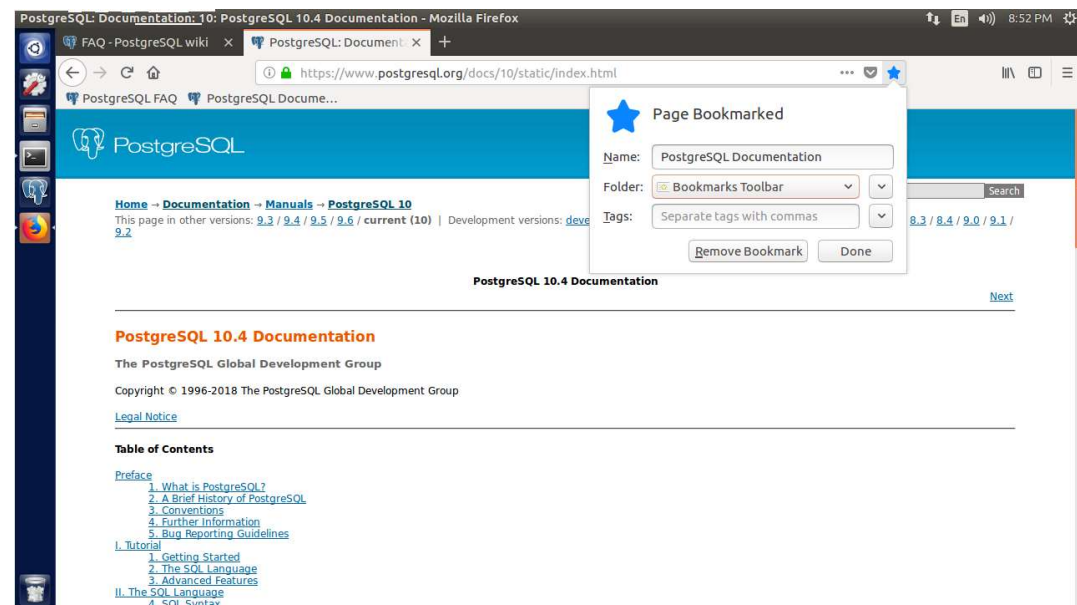
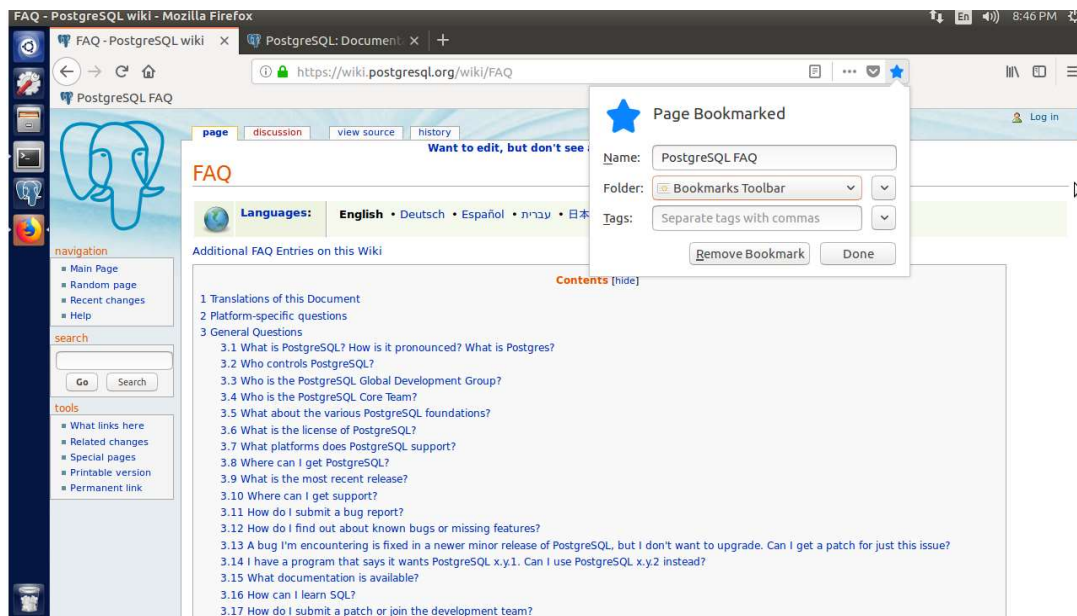
07/22/18

PostgreSQL

Day 1

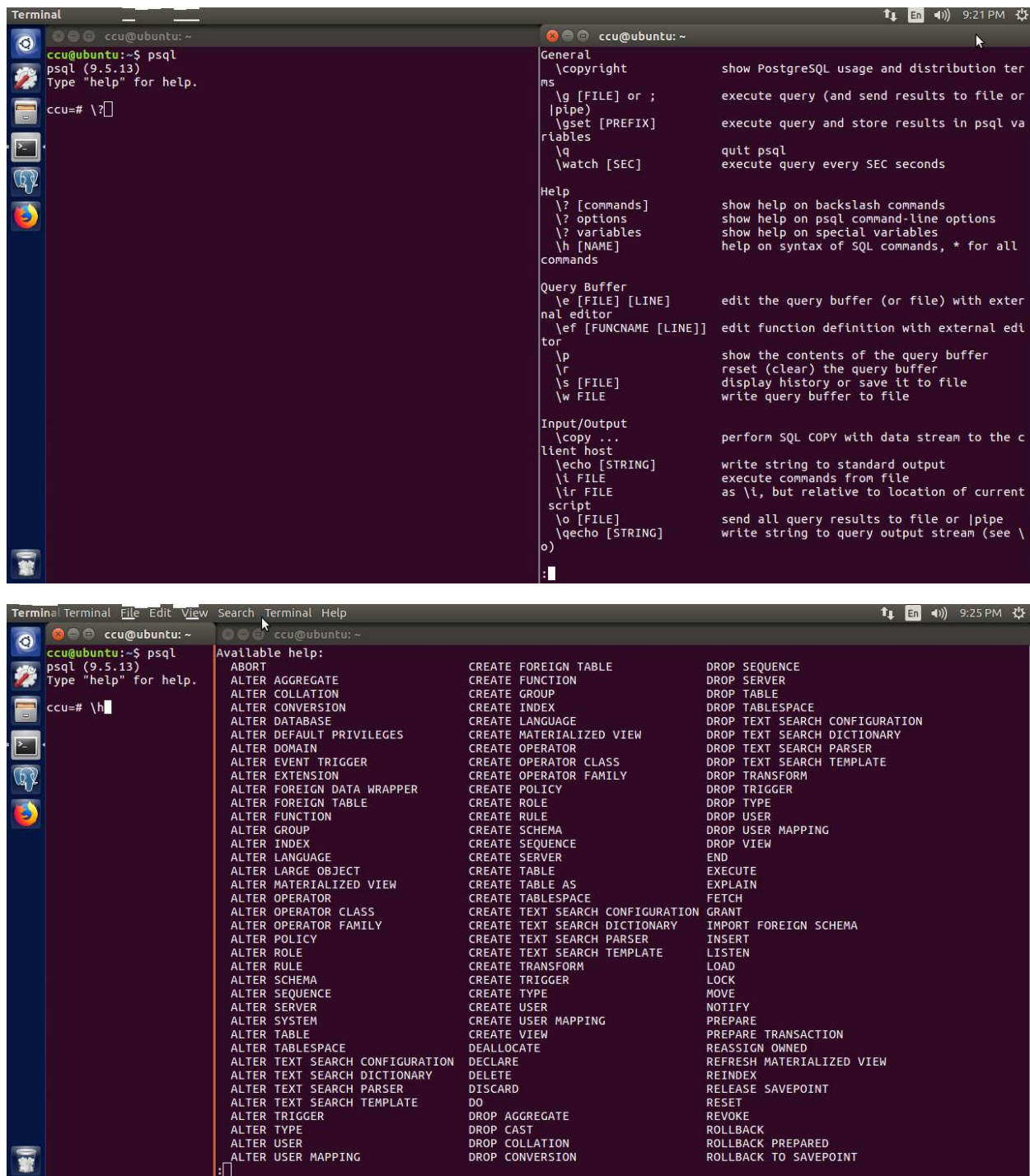
Find:

1. Bookmark the online PostgreSQL FAQ and documents.



For this I simply pressed *Ctrl+D* in Firefox and added the bookmarks to a folder of my choice.

2. Acquaint yourself with the command-line \? and \h output.



```
ccu@ubuntu:~$ psql
psql (9.5.13)
Type "help" for help.

ccu=# \?
```

General

- \copyright show PostgreSQL usage and distribution terms
- \g [FILE] or ; execute query (and send results to file or pipe)
- \gset [PREFIX] execute query and store results in psql variables
- \q quit psql
- \watch [SEC] execute query every SEC seconds

Help

- \? [commands] show help on backslash commands
- \? options show help on psql command-line options
- \? variables show help on special variables
- \h [NAME] help on syntax of SQL commands, * for all commands

Query Buffer

- \e [FILE] [LINE] edit the query buffer (or file) with external editor
- \ef [FUNCNAME [LINE]] edit function definition with external editor
- \p show the contents of the query buffer
- \r reset (clear) the query buffer
- \s [FILE] display history or save it to file
- \w FILE write query buffer to file

Input/Output

- \copy ... perform SQL COPY with data stream to the client host
- \echo [STRING] write string to standard output
- \i FILE execute commands from file
- \ir FILE as \i, but relative to location of current script
- \o [FILE] send all query results to file or pipe
- \qecho [STRING] write string to query output stream (see \o)

```
ccu@ubuntu:~$ psql
psql (9.5.13)
Type "help" for help.

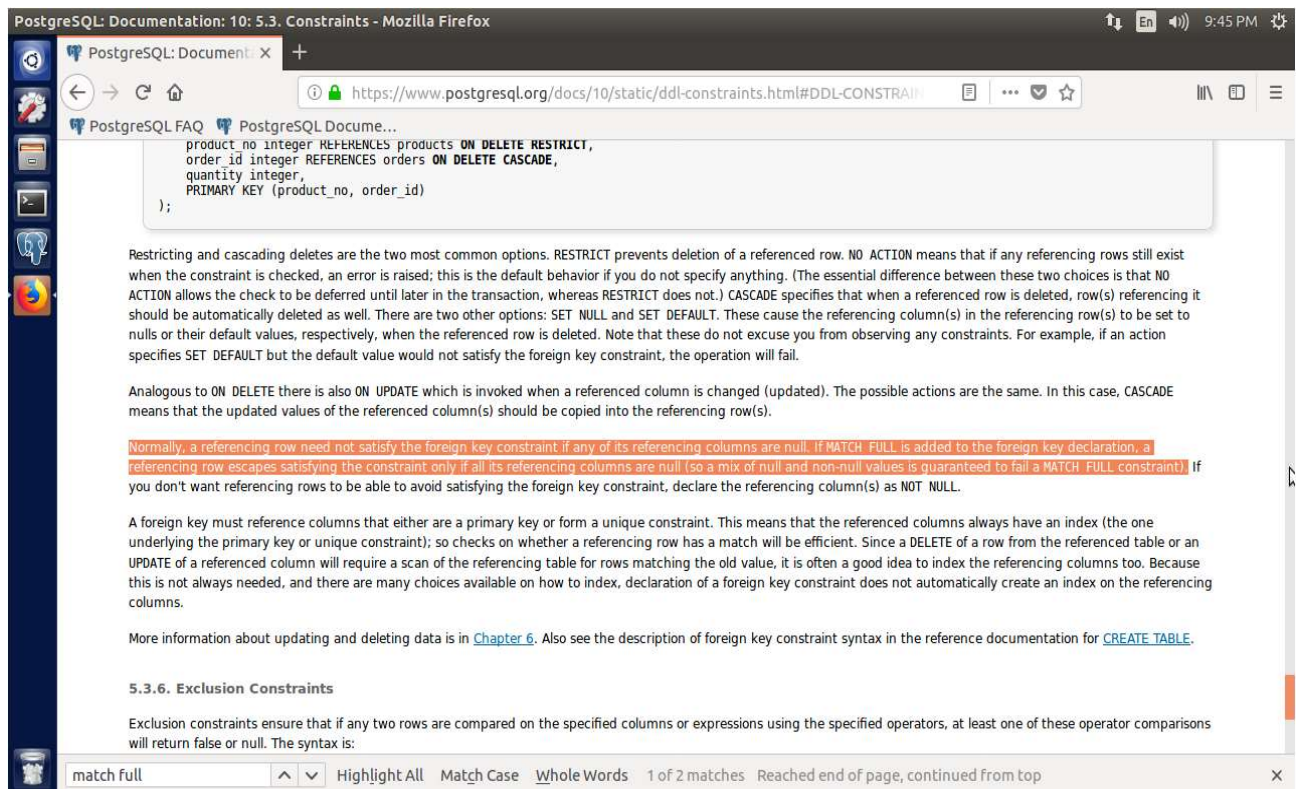
ccu=# \h
```

Available help:

ABORT	CREATE FOREIGN TABLE	DROP SEQUENCE
ALTER AGGREGATE	CREATE FUNCTION	DROP SERVER
ALTER COLLATION	CREATE GROUP	DROP TABLE
ALTER CONVERSION	CREATE INDEX	DROP TABLESPACE
ALTER DATABASE	CREATE LANGUAGE	DROP TEXT SEARCH CONFIGURATION
ALTER DEFAULT PRIVILEGES	CREATE MATERIALIZED VIEW	DROP TEXT SEARCH DICTIONARY
ALTER DOMAIN	CREATE OPERATOR	DROP TEXT SEARCH PARSER
ALTER EVENT TRIGGER	CREATE OPERATOR CLASS	DROP TEXT SEARCH TEMPLATE
ALTER EXTENSION	CREATE OPERATOR FAMILY	DROP TRANSFORM
ALTER FOREIGN DATA WRAPPER	CREATE POLICY	DROP TRIGGER
ALTER FOREIGN TABLE	CREATE ROLE	DROP TYPE
ALTER FUNCTION	CREATE RULE	DROP USER
ALTER GROUP	CREATE SCHEMA	DROP USER MAPPING
ALTER INDEX	CREATE SEQUENCE	DROP VIEW
ALTER LANGUAGE	CREATE SERVER	END
ALTER LARGE OBJECT	CREATE TABLE	EXECUTE
ALTER MATERIALIZED VIEW	CREATE TABLE AS	EXPLAIN
ALTER OPERATOR	CREATE TABLESPACE	FETCH
ALTER OPERATOR CLASS	CREATE TEXT SEARCH CONFIGURATION	GRANT
ALTER OPERATOR FAMILY	CREATE TEXT SEARCH DICTIONARY	IMPORT FOREIGN SCHEMA
ALTER POLICY	CREATE TEXT SEARCH PARSER	INSERT
ALTER ROLE	CREATE TEXT SEARCH TEMPLATE	LISTEN
ALTER RULE	CREATE TRANSFORM	LOAD
ALTER SCHEMA	CREATE TRIGGER	LOCK
ALTER SEQUENCE	CREATE TYPE	MOVE
ALTER SERVER	CREATE USER	NOTIFY
ALTER SYSTEM	CREATE USER MAPPING	PREPARE
ALTER TABLE	CREATE VIEW	PREPARE TRANSACTION
ALTER TABLESPACE	DEALLOCATE	REASSIGN OWNED
ALTER TEXT SEARCH CONFIGURATION	DECLARE	REFRESH MATERIALIZED VIEW
ALTER TEXT SEARCH DICTIONARY	DELETE	REINDEX
ALTER TEXT SEARCH PARSER	DISCARD	RELEASE SAVEPOINT
ALTER TEXT SEARCH TEMPLATE	DO	RESET
ALTER TRIGGER	DROP AGGREGATE	REVOKE
ALTER TYPE	DROP CAST	ROLLBACK
ALTER USER	DROP COLLATION	ROLLBACK PREPARED
ALTER USER MAPPING	DROP CONVERSION	ROLLBACK TO SAVEPOINT

In the terminal, I typed **psql** which opens a PostgreSQL shell for a database. I then entered **\?** for psql command help and **\h** for SQL command help.

3. In the addresses **FOREIGN KEY**, find in the docs what **MATCH FULL** means.

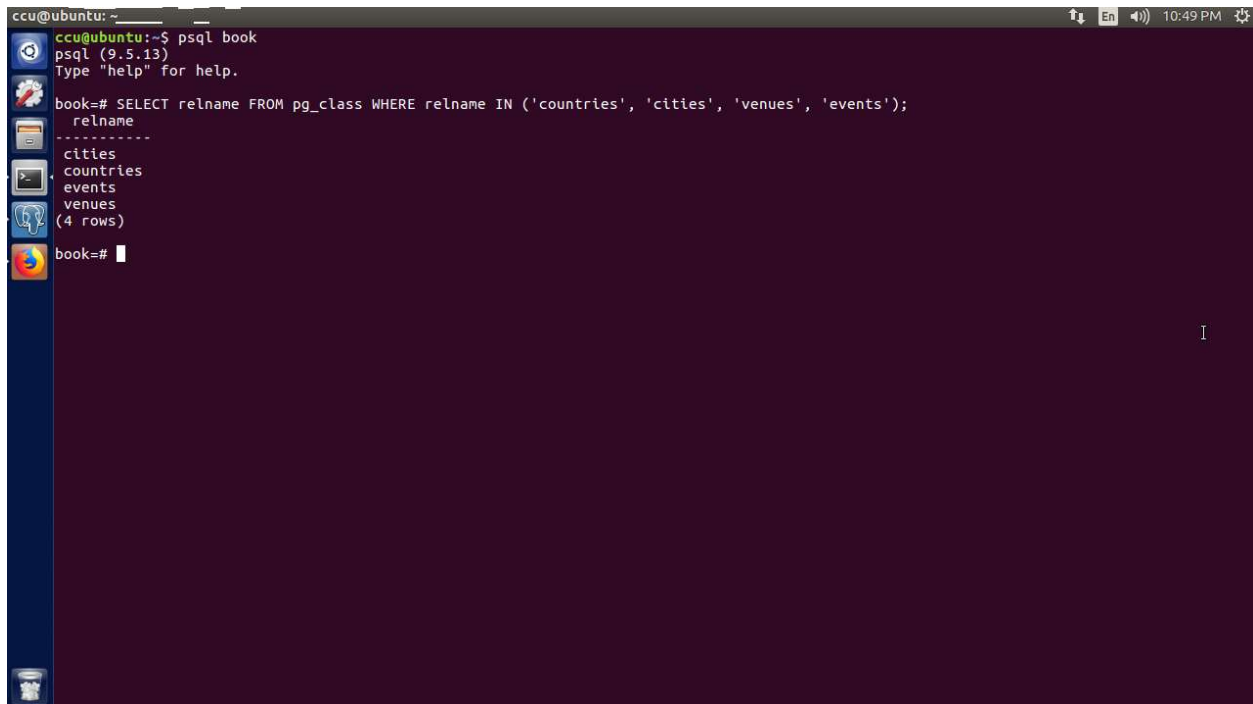


To find the use of **MATCH FULL**, I went to the PostgreSQL [10.4] Documentation bookmark and followed this series of links: II. The SQL Language -> 5.3 Constraints -> 5.3.5 Foreign Keys and then used Ctrl+F to search the page for **MATCH FULL**.

According to the documentation, **MATCH FULL** means that a referencing row can avoid referential integrity if its referencing columns are all **null**.

Do:

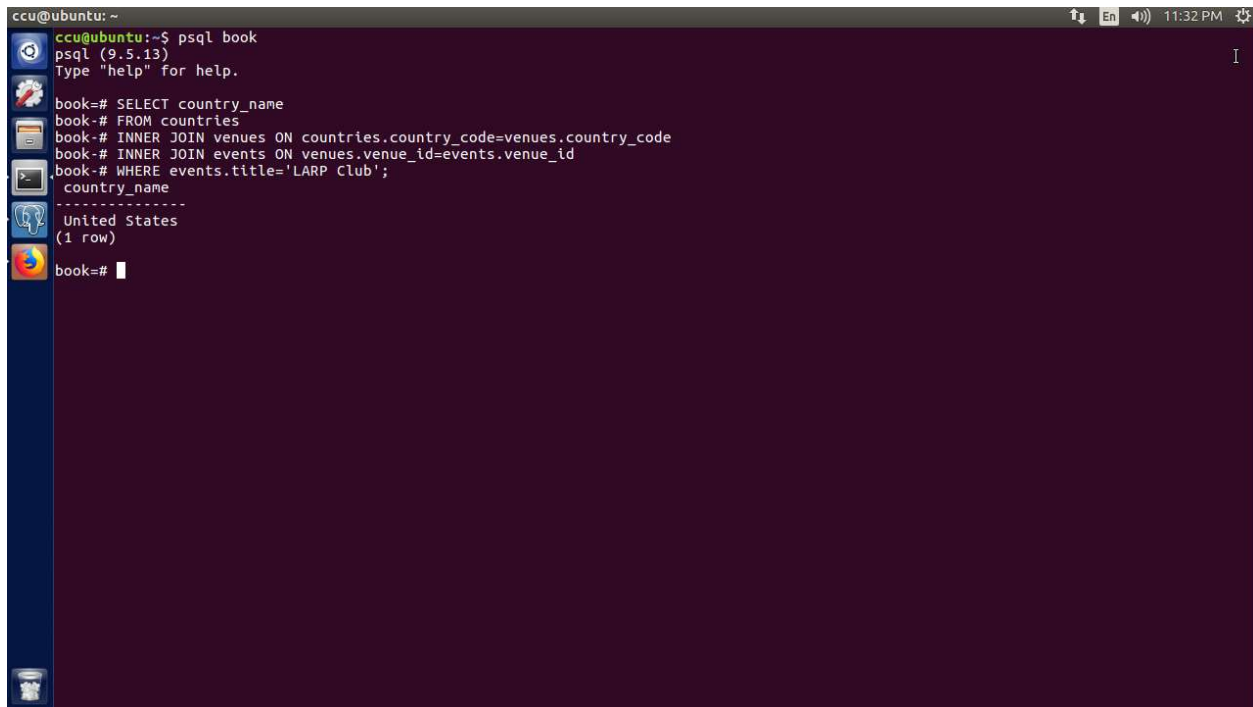
1. Select all the tables we created (and only those) from **pg_class**.

A terminal window on an Ubuntu system. The prompt is 'ccu@ubuntu: ~'. The user has entered 'psql book', which has opened the PostgreSQL interactive shell. The prompt is now 'book=#'. The user has entered the SQL query 'SELECT relname FROM pg_class WHERE relname IN ('countries', 'cities', 'venues', 'events');'. The output shows a single column 'relname' with four rows: 'cities', 'countries', 'events', and 'venues'. The prompt is now 'book=#' again.

```
ccu@ubuntu: ~  
ccu@ubuntu:~$ psql book  
psql (9.5.13)  
Type "help" for help.  
  
book=# SELECT relname FROM pg_class WHERE relname IN ('countries', 'cities', 'venues', 'events');  
 relname  
-----  
cities  
countries  
events  
venues  
(4 rows)  
  
book=#
```

First, I had to read the documentation to find out what **pg_class** is. Apparently, **pg_class** is a catalog that catalogs tables and anything that resembles a table. One of the columns in the catalog is **relname**, which contains the names of all the tables within **pg_class**. So I took this variable and used it in a basic query to list all of the tables in the **book** database.

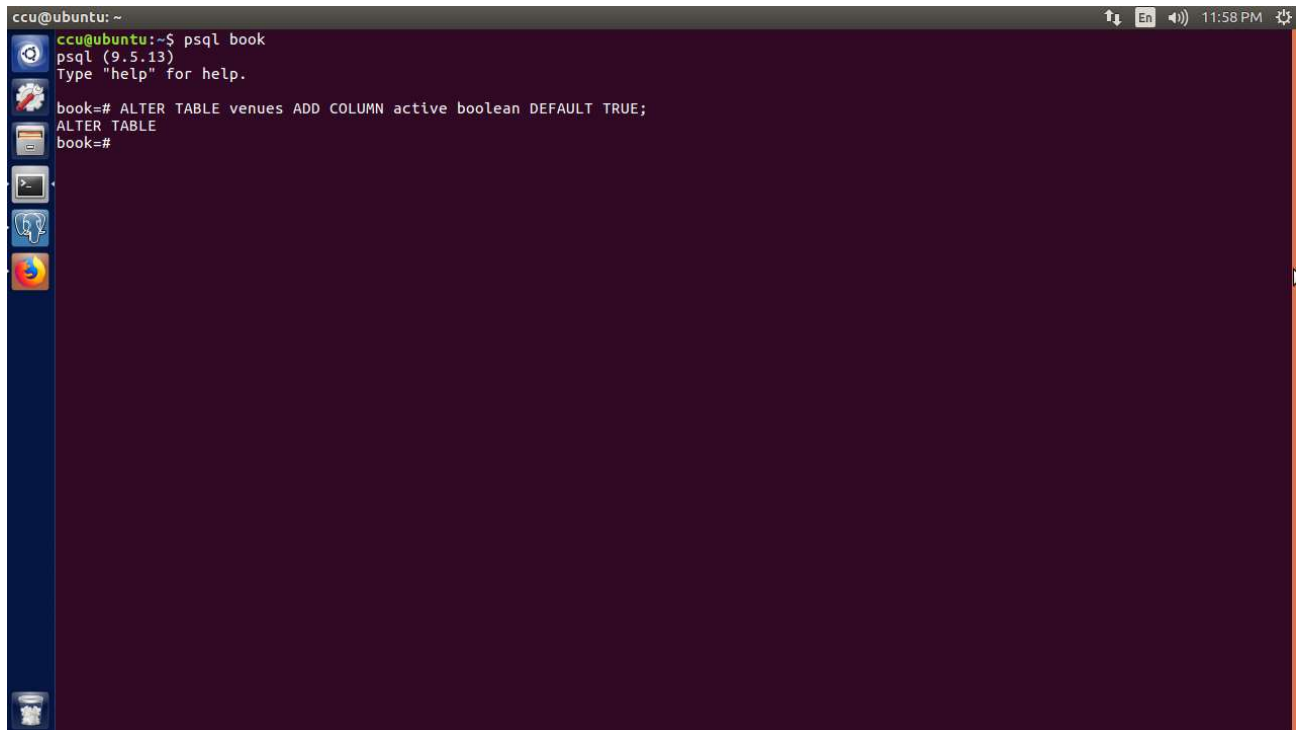
2. Write a query that finds the country name of the LARP Club event.

A terminal window on an Ubuntu system showing a PostgreSQL session. The user has entered a query with two inner joins to find the country name for the 'LARP Club' event. The output shows 'United States' as the result.

```
ccu@ubuntu: ~  
ccu@ubuntu:~$ psql book  
psql (9.5.13)  
Type "help" for help.  
  
book=# SELECT country_name  
book=# FROM countries  
book=# INNER JOIN venues ON countries.country_code=venues.country_code  
book=# INNER JOIN events ON venues.venue_id=events.venue_id  
book=# WHERE events.title='LARP Club';  
country_name  
-----  
United States  
(1 row)  
  
book=#
```

This involved a query with two simple inner joins. Column **country_code** connects tables **countries** and **venues**, and column **venue_id** connects tables **venues** and **events**. All that was needed were two inner joins to connect the three tables.

3. Alter the **venues** table to contain a Boolean column called **active**, with the default value of **TRUE**.

A terminal window on an Ubuntu system. The prompt is 'ccu@ubuntu: ~'. The user has entered 'psql book', which has opened a psql session. The prompt is now 'psql (9.5.13)' and it says 'Type "help" for help.' The user has entered 'book=# ALTER TABLE venues ADD COLUMN active boolean DEFAULT TRUE;', followed by 'ALTER TABLE' and then 'book=#'. The terminal has a dark purple background and a blue sidebar on the left with various application icons.

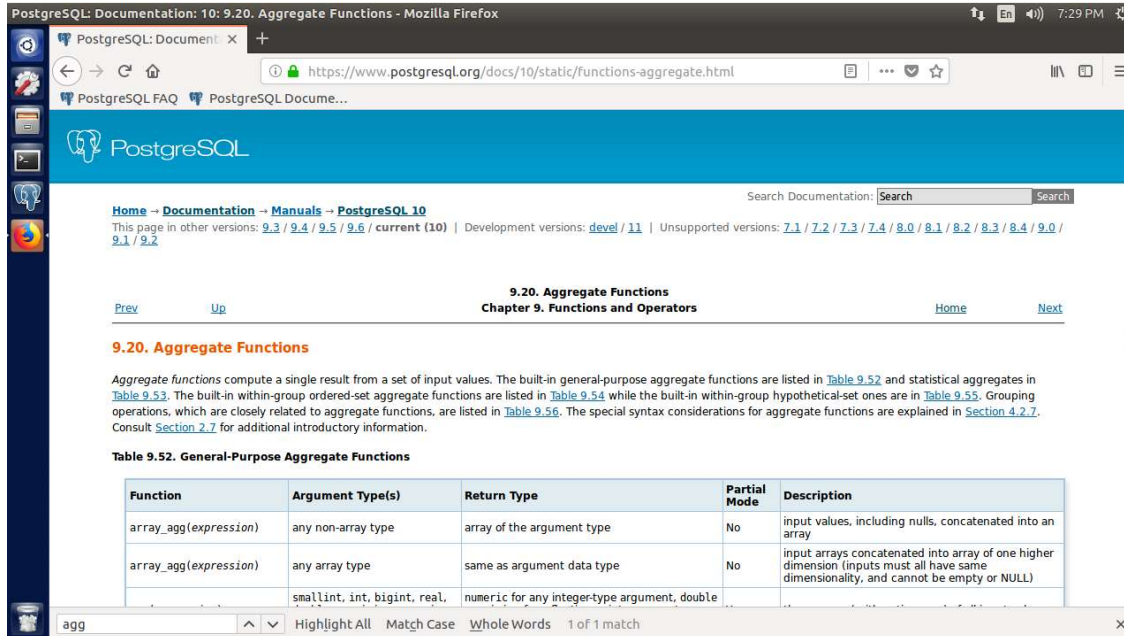
```
ccu@ubuntu: ~  
ccu@ubuntu:~$ psql book  
psql (9.5.13)  
Type "help" for help.  
book=# ALTER TABLE venues ADD COLUMN active boolean DEFAULT TRUE;  
ALTER TABLE  
book=#
```

Another simple query, this one using **ALTER TABLE** to alter the table **venues**, **ADD COLUMN** to add the column **active** as a boolean data type to **venues**, and **DEFAULT TRUE** to set the default value of **active** to **TRUE**.

Day 2

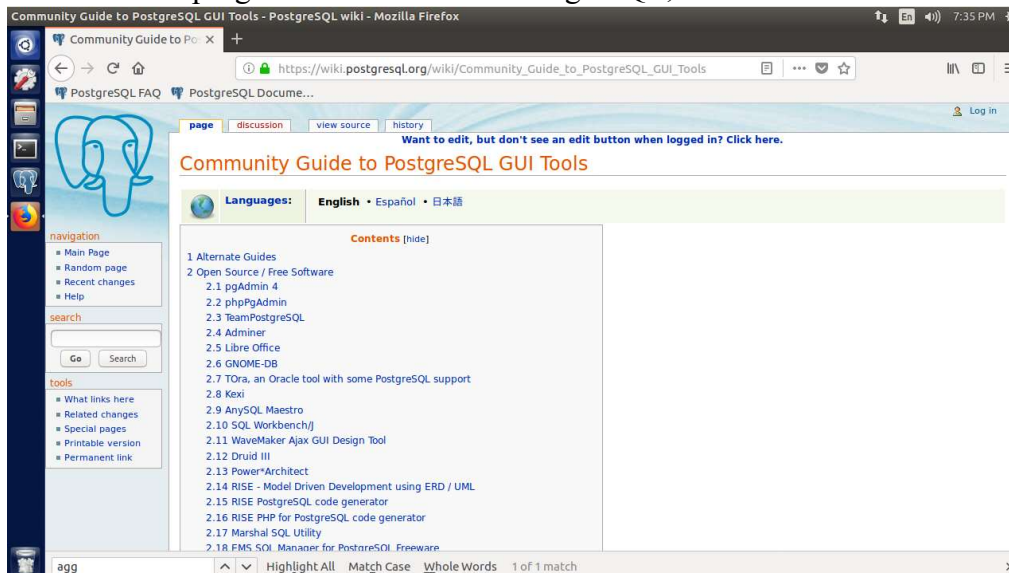
Find:

1. Find the list of aggregate functions in the PostgreSQL docs.



I simply went to the PostgreSQL [10.4] Documentation page and followed these links: II. The SQL Language -> 9.20 Aggregate Functions.

2. Find a GUI program to interact with PostgreSQL, such as Navicat.



After a quick Google search for PostgreSQL GUI tools, it turns out that the PostgreSQL wiki has an extensive listing of both open source and proprietary GUI programs for PostgreSQL.

Do:

1. Create a rule that captures **DELETEs** on venues and instead sets the active flag (created in the Day 1 homework) to **FALSE**.

```
ccu@ubuntu: ~  
ccu@ubuntu:~$ psql book  
psql (9.5.13)  
Type "help" for help.  
  
book=# CREATE RULE delete_venues AS ON DELETE TO venues DO INSTEAD  
book=# UPDATE venues SET active=FALSE WHERE venue_id=OLD.venue_id;  
CREATE RULE  
book=# SELECT * FROM venues;  
venue_id | name | street_address | type | postal_code | country_code | active  
-----  
3 | House | 666 Elm Street | private | M4C 5K7 | ca | f  
4 | Run's House | | public | 97205 | us | t  
1 | Crystal Ballroom | | public | 97205 | us | t  
2 | Voodoo Donuts | | public | 97205 | us | t  
(4 rows)  
  
book=# DELETE FROM venues WHERE name='Crystal Ballroom';  
DELETE 0  
book=# SELECT * FROM venues;  
venue_id | name | street_address | type | postal_code | country_code | active  
-----  
3 | House | 666 Elm Street | private | M4C 5K7 | ca | f  
4 | Run's House | | public | 97205 | us | t  
2 | Voodoo Donuts | | public | 97205 | us | t  
1 | Crystal Ballroom | | public | 97205 | us | f  
(4 rows)  
  
book=# DELETE FROM venues WHERE name='Voodoo Donuts';  
DELETE 0  
book=# SELECT * FROM venues;  
venue_id | name | street_address | type | postal_code | country_code | active  
-----  
3 | House | 666 Elm Street | private | M4C 5K7 | ca | f  
4 | Run's House | | public | 97205 | us | t  
1 | Crystal Ballroom | | public | 97205 | us | f  
2 | Voodoo Donuts | | public | 97205 | us | f  
(4 rows)  
  
book=#
```

For this rule, I first tried to update the **active** column to **FALSE** with no other parameters, but this ended up converting the entire column to **FALSE** (which I suspected). Then, I set **active** to **FALSE** where **venue_id=venue_id**, but this resulted in the same behavior. Finally, I used **venue_id=OLD.venue_id**, which selects the **venue_id** of the row you are referring to in the **DELETE** statement. This caused the rule to work perfectly.

2. A temporary table was not the best way to implement our event calendar pivot table. The **generate_series(a, b)** function returns a set of records, from a to b. Replace the **month_count** table **SELECT** with this.

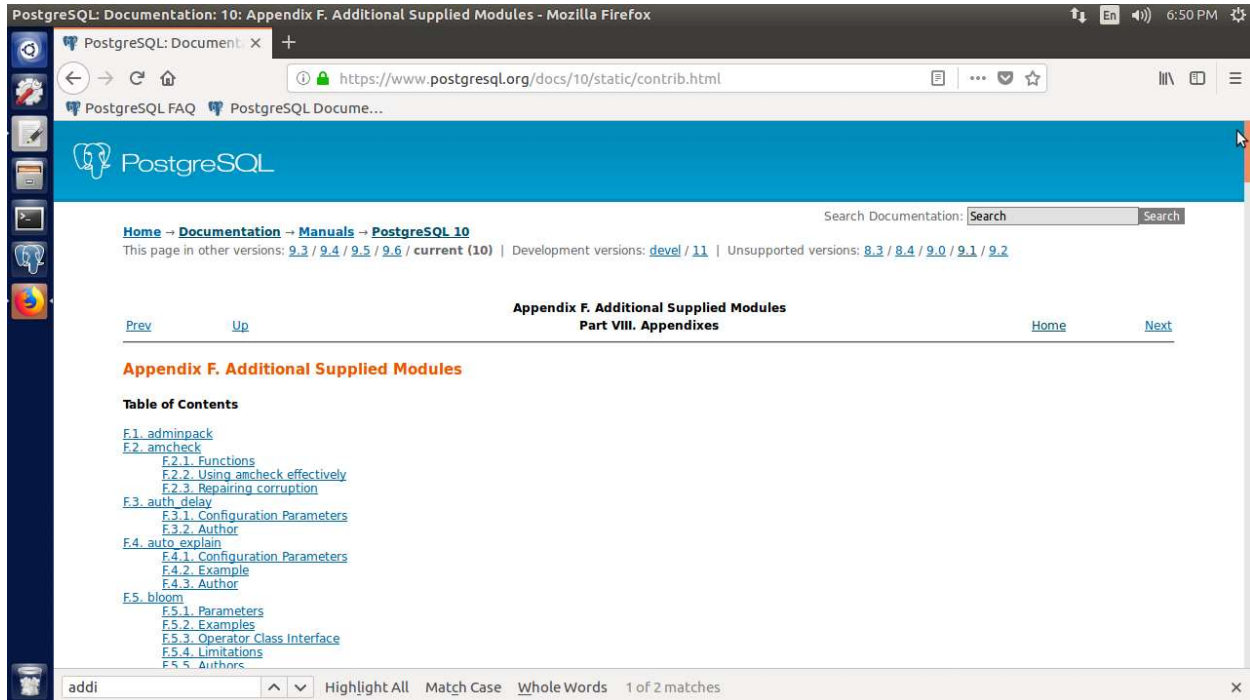
```
ccu@ubuntu: ~  
ccu@ubuntu:~$ psql book  
psql (9.5.13)  
Type "help" for help.  
  
book=# SELECT * FROM crosstab(  
book(# 'SELECT extract(year from starts) as year,  
book'# extract(month from starts) as month, count(*)  
book'# FROM events  
book'# GROUP BY year, month  
book'# ORDER BY year, month',  
book(# 'SELECT * FROM generate_series(1, 12)'  
book(# ) AS (  
book(# year int,  
book(# jan int, feb int, mar int, apr int, may int, jun int,  
book(# jul int, aug int, sep int, oct int, nov int, dec int  
book(# ) ORDER BY YEAR;  
year | jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec  
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----  
2012 |    | 4 |    | 1 | 1 | 1 | 1 | 1 |    |    | 1 | 1 | 1  
2013 | 1 |    |    |    |    |    |    |    |    |    |    |    |  
(2 rows)  
  
book=#
```

This required going to the documentation and reading about the **generate_series** function. The function **generate_series(a, b)** returns a series of integers or big integers where **a** is the start of the series, and **b** is the end. Then, I just had to replace the '**SELECT * FROM month_count**' statement in the book's example query with '**SELECT * FROM generate_series(1, 12)**', as highlighted above.

Day 3

Find:

1. Find online documentation of all contributed packages bundled into Postgres.



For this, I went to the PostgreSQL documentation and followed these links: VIII. Appendixes -> F. Additional Supplied Modules. “Additional Supplied Modules” is just another name for contributed packages.

2. Find online POSIX regex documentation (it will also be handy for future chapters).



Through a Google search, I found a website called regular-expressions.info. It contains everything from quick references to in-depth guides, and also includes specific documentation for many different languages and databases.

Do:

1. Create a stored procedure where you can input a movie title or actor's name you like, and it will return the top five suggestions based on either movies the actor has starred in or films with similar genres.

```
psql_day3.sql (~/Desktop/book_code/postgres/Homework) - gedit
Open Save
CREATE OR REPLACE FUNCTION top_five (mov_act text, mov_act_name text)
RETURNS TABLE(list text) AS $$
BEGIN
    IF mov_act='actor' THEN
        RETURN QUERY
        SELECT title
        FROM movies NATURAL JOIN movies_actors NATURAL JOIN actors
        WHERE metaphone(name, 7) = metaphone(mov_act_name, 7)
        ORDER BY movie_id LIMIT 5;
    ELSE IF mov_act='movie' THEN
        RETURN QUERY
        SELECT m.title
        FROM movies AS m, (SELECT genre, title FROM movies WHERE title=mov_act_name) AS s
        WHERE cube_enlarge(s.genre, 5, 18) @> m.genre AND s.title <> m.title
        ORDER BY cube_distance(m.genre, s.genre) LIMIT 5;
    END IF;
END IF;
END;
$$ LANGUAGE plpgsql;
```

```
ccu@ubuntu: ~
ccu@ubuntu:~$ psql book
psql (9.5.13)
Type "help" for help.

book=# \i /home/ccu/Desktop/book_code/postgres/Homework/psql_day3.sql
CREATE FUNCTION
book=# SELECT top_five('actor', 'Christian Bale');
         top_five
-----
Velvet Goldmine
3:10 to Yuma
Empire of the Sun
A Midsummer Night's Dream
Swing Kids
(5 rows)

book=# SELECT top_five('movie', 'Scarface');
         top_five
-----
Lonely Hearts
Heat
The French Connection
Kiss of Death
The Real McCoy
(5 rows)

book=#
```

For this stored procedure, I wrote the function in a **.sql** text file and imported it into the **book** database, as this was easier than typing it directly into the command line. The main body of the function was easy, as the queries to select movies were virtually the same as the examples in the book except for one variable that needed to match the second parameter in the function. I had to research the function return types in the documentation and found out that I needed to use the **RETURN TABLE** type along with the **RETURN QUERY** statement. The **RETURN QUERY** statement returns the results of the query it refers to and **RETURN TABLE** houses the results of the query return. Thus, **RETURN TABLE** displays the list of movies that the user is looking for.

2. Expand the movies database to track user comments and extract keywords (minus English stopwords). Cross-reference these keywords with actors' last names, and try to find the most talked about actors.

```
ccu@ubuntu: ~
ccu@ubuntu:~$ psql book
psql (9.5.13)
Type "help" for help.

book=# ALTER TABLE movies ADD COLUMN comments text;
ALTER TABLE
book=# UPDATE movies
SET comments='Amazingly cool!'
WHERE title='Star Wars';
UPDATE 1
book=# UPDATE movies
SET comments='Extremely funny and cool.'
WHERE title='Forrest Gump';
UPDATE 1
book=# UPDATE movies
SET comments='Simply beautiful and amazing.'
WHERE title='American Beauty';
UPDATE 1
book=# UPDATE movies
SET comments='Amazing and beautiful movie!'
WHERE title='Citizen Kane';
UPDATE 1
book=# UPDATE movies
SET comments='The Dark be spooky and cool.'
WHERE title='The Dark';
UPDATE 1
book=# SELECT * FROM movies ORDER BY movie_id LIMIT 5;
 movie_id | title          | genre | comments
-----+-----+-----+-----
1 | Star Wars     | (0, 7, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 10, 0, 0, 0) | Amazingly cool!
2 | Forrest Gump  | (0, 0, 0, 5, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0) | Extremely funny and cool.
3 | American Beauty | (0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0) | Simply beautiful and amazing.
4 | Citizen Kane  | (0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0) | Amazing and beautiful movie!
5 | The Dark      | (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5, 0) | The Dark be spooky and cool.
(5 rows)

book=#
```

```
ccu@ubuntu: ~  
ccu@ubuntu:~$ psql book  
psql (9.5.13)  
Type "help" for help.  
  
book=# CREATE INDEX keyword ON movies  
USING gin(to_tsvector('english', comments));  
CREATE INDEX  
book=# SELECT * FROM movies WHERE to_tsvector('english', comments) @@ 'cool';  
movie_id | title | genre | comments  
-----  
5 | The Dark | (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5, 0) | The Dark be spooky and cool.  
2 | Forrest Gump | (0, 0, 0, 5, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0) | Extremely funny and cool.  
1 | Star Wars | (0, 7, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 10, 0, 0) | Amazingly cool!  
(3 rows)  
  
book=#
```

```
ccu@ubuntu: ~  
ccu@ubuntu:~$ psql book  
psql (9.5.13)  
Type "help" for help.  
  
book=# SELECT comments FROM movies NATURAL JOIN movies_actors NATURAL JOIN actors  
WHERE metaphone(name, 7)=metaphone('Mark Hamil', 7)  
ORDER BY movie_id;  
comments  
-----  
Amazingly cool!  
(4 rows)  
  
book=#
```

To keep track of comments, I just used the **ALTER TABLE** and **ADD COLUMN** commands to add the column **comments** to the table. To extract keywords, an index was needed on the **comments** column in order to search by keyword with **to_tsvector**. Finally, cross-referencing actors with movie comments was the same as with movie titles where the tables were joined and metaphones compared.