Lab 5 : Text Operations, Indexes, and Triggers
CMSC 362
Spring 2014

Due Fri. Apr 4, 2014 by 5pm

Some of the most common queries involve searching for a keyword in a block of text or finding some pattern in a data set. PostgreSQL provides some limited regular expression capabilities for performing these kinds of searches.

Step 1. Setting Up

Create a new folder named Lab 5. In that folder, create a file named lab5.sql.

As usual, you will be able to use

psql -h torvalds -U lastName -d lastName -f lab5.sql

to connect to the database and execute the script.

Edit lab5.sql and insert a valid SQL comment that says "Lab 5 : " and your full name.

Step 2. A simple schema

In your lab5.sql file, add SQL code to create a table named "profiles" which has three fields: a field named “username” which should hold up to 40 characters of data, a field named “profile” which has can hold arbitrarily long text, and a num_logins field which is an integer. Make username a primary key and give num_logins a default of 1.

Step 3. Inserting Data

Add "Insert Into" lines to the file which populate your schema with at least three entries. Each of these entries should have a profile that contains at least 300 words. I recommend copying and pasting text from a website such as "Project Gutenburg", "Blueletter Bible", or "Slashdot" to save yourself some time. (You will need to be VERY careful about escaping quote marks and such, however). Or, if you're feeling very bored you could type something out yourself. To simplify the remainder of the lab (and my grading), give the entries the user names **Susie, Stephen, and Kevin**.

Step 4. Keyword Searches

Postgres provides three ways to perform pattern matching:

LIKE and ILIKE
SIMILAR TO
Regular Expressions (POSIX flavor)

The LIKE and SIMILAR TO matches must match the entire field. Regular expressions are allowed to match any substring of a field. Here is an example of using "LIKE" to search for usernames that start with "S":

```
SELECT username FROM Profiles WHERE username ILIKE 'S%';
```

The % character is a wildcard -- it is like the "*" character we use for filenames in Unix (don't ask me why SQL uses a different character – I blame Oracle). So 'S%' means "any string that starts with S".

You can also use the underscore character, "_", as a single character wildcard. So "Jo_" would match both "Joe" and "Joy" (or even "Job"), but not "Joey".

You can use a backslash to escape the % and _ characters if you need to match them.

The difference between using "LIKE" and "ILIKE" is that "ILIKE" ignores case. That is, it will match both upper and lowercase strings that match.

Write (at the bottom of your lab5.sql file) a query for: all usernames that end with "n".

Step 5. Regular Expressions

You can (but don't do this now) replace "LIKE" with "SIMILAR TO" to perform slightly more sophisticated queries. With SIMILAR TO, you can use the kleene star (*) and kleene plus (+) to get "zero/one or more repetitions of a string". You can use the alternatives symbol (|) to match one of a set of strings. You can also use character classes (such as [abc] or [^xyz]) and parentheses to disambiguate the pattern.

Postgres also supports matching with regular expressions. Instead of the "LIKE" or "ILIKE" keyword, we simply use the ~ character. This will find all records in the database for which any part of a field matches the regular expressions.

Regular expressions are built using the same tools as the "SIMILAR TO" operator with these differences:

1. The % and _ wildcards are replaced with the "." single character wild card. You can use this together with the Kleene star to get a "match any string" wildcard.

2. Any part of the string is allowed to match the regular expression, unless you specifically anchor the expression to the beginning of the string using the "^" character or to the end of the string using the "$" character.

Create a regular expression which searches for any word that starts with "st" in each profile and lists the username and profile of every row that matches.

Step 6. Triggers and Indexing

In class we have talked about ways to speed up queries using indexing. We have also talked about using triggers to enforce more sophisticated conditions than a simple constraint can handle.

Before continuing, log into the database in interactive mode:
psql -h torvalds -U lastName -d lastName

Then type "CREATE LANGUAGE plpgsql;"

This ensures that postgres has loaded the PG-SQL language interpreter into your database.

Hit enter and then quit (perhaps by typing CTRL-D).

Step 7. Speeding up queries with an index

Create an index for the profile attribute by adding the following text to your lab7.sql file:

```
CREATE INDEX ON profiles(profile);
```

Then create an index for the num_logins attribute.

Step 8. Creating a Function

Suppose that we wanted to increment the number of logins every time someone modifies their profile. One way to do that would be to create a trigger. A trigger is an SQL feature that allows us to automatically execute a function whenever a table is changed. In order to execute a trigger, we

must first create a function to associate with that trigger.

Let's create a function named "update_logins".
Type the following code at the bottom of your sql script file:

```
CREATE OR REPLACE FUNCTION update_logins()
RETURNS trigger
AS $$
        BEGIN
        UPDATE profiles SET num_logins = num_logins + 1 WHERE username = OLD.username;
        RETURN NULL;
        END;
$$ LANGUAGE 'plpgsql';
```

You'll notice that the syntax for this function is a little different from what we've done in class. That's because this is a special kind of function called a “trigger procedure” (Notice that it returns “trigger”.) This special return type says that the function will only be used from a trigger and takes no parameters. Instead, we can get information about the query/insert/update that triggered the function using the OLD and NEW keyword. The keyword NEW can be used to access the values of attributes in the triggering row. The keyword OLD can be used to access the original values of the triggering row's attributes BEFORE the modification that set off the trigger.

A function with return type “trigger” must either return NULL or a row which has the same attributes as the row that was modified to set off the trigger. If you return a row, that row will replace the triggering row.

Step 9. Creating a Trigger

To cause our function to be called when someone changes a profile, we need to add an UPDATE trigger. **Add these lines of code to your file.** They must be *below* the definition of the update_logins function.

```
CREATE TRIGGER update_login_trigger
AFTER UPDATE ON profiles
FOR EACH ROW
WHEN (OLD.profile IS DISTINCT FROM NEW.profile)
EXECUTE PROCEDURE update_logins();
```

The first line of this statement creates a trigger named “update_login_trigger”.

If you goof things up and need to delete it, you can use:

```
DROP TRIGGER update_login_trigger ON profiles;
```

The second line specifies that the trigger is to be called AFTER any row is updated in the profiles table. It is also possible to use the “BEFORE” keyword to trigger a function after the update has been called, but before it has done anything. Additionally, you can use the keyword “INSERT” instead of UPDATE or use the keywords “INSERT OR UPDATE” to also trigger the function when a new row is created.

The last lines specify that this is a “per-row” trigger that calls the function update_logins if the profile column has been changed (we don't want an update of the num_logins field to trigger the function, because update_logins changes the num_logins field itself).

> **Create a trigger function and trigger that changes the profile to “You win” if the number of logins reaches 10.**

Submitting

As usual, save your work and upload your lab5.sql file to
http://narnia.homeunix.com/~robert/submit by 5:00pm today.