Assignment 1

Adding an Email Data Type to PostgreSQL

Last updated: **Monday 17th June 11:05pm**
Most recent changes are shown in red;
older changes are shown in brown.

Aims

This assignment aims to give you

- an understanding of how data is treated inside a DBMS
- practice in adding a new base type to PostgreSQL

The goal is to implement a new data type for PostgreSQL, complete with input/output functions, comparison operators and the ability to build indexes on values of the type.

Summary

Deadline	Sunday 7 July, 11:59pm
Pre-requisites:	before starting this assignment, it would be useful to complete Prac Work P04
Late Penalty:	0.08 *marks* off the ceiling mark for each *hour* late
Marks:	This assignment contributes **12 marks** toward your total mark for this course.
Submission:	Webcms3 > Assignments > Ass1 Submission > Make Submission or, on CSEmachines, `give cs9315 ass1 email.c email.source`

Make sure that you read this assignment specification *carefully and completely* before starting work on the assignment.
Questions which indicate that you haven't done this will simply get the response "Please read the spec".

We use the following names in the discussion below

- *`PG_CODE`* ... the directory where your PostgreSQL source code is located (typically `/srvr/`*`YOU`*`/postgresql-11.3/`)
- *`PG_HOME`* ... the directory where you have installed the PostgreSQL binaries (typically `/srvr/`*`YOU`*`/pgsql/bin/`)
- *`PG_DATA`* ... the directory where you have placed PostgreSQL's `data` (typically `/srvr/`*`YOU`*`/pgsql/data/`)
- *`PG_LOG`* ... the file where you send PostgreSQL's log output (typically `/srvr/`*`YOU`*`/pgsql/data/log/`)

Introduction

PostgreSQL has an extensibility model which, among other things, provides a well-defined process for adding new data types into a PostgreSQL server. This capability has led to the development by PostgreSQL users of a number of types (such as polygons) which have become part of the standard distribution. It also means that PostgreSQL is the database of choice in research projects which aim to push the boundaries of what kind of data a DBMS can manage.

In this assignment, we will be adding a new data type for dealing with **email addresses**. You may implement the functions for the data type in any way you like *provided that* they satisfy the semantics given below (in the Email Address Data Type section).

The process for adding new base data types in PostgreSQL is described in the following sections of the PostgreSQL documentation:

- 38.12 User-defined Types
- 38.10 C-Language Functions
- 38.13 User-defined Operators
- SQL: CREATE TYPE

- SQL: CREATE OPERATOR
- SQL: CREATE OPERATOR CLASS

Section 38.12 uses an example of a complex number type, which you can use as a starting point for defining your `EmailAddr` data type (see below). There are other examples of new data types under the directories:

- *PG_CODE*`/contrib/chkpass/` ... an auto-encrypted password datatype
- *PG_CODE*`/contrib/citext/` ... a case-insensitive character string datatype
- *PG_CODE*`/contrib/seg/` ... a confidence-interval datatype

These may or may not give you some useful ideas on how to implement the Email address data type.

Setting Up

You ought to start this assignment with a fresh copy of PostgreSQL, without any changes that you might have made for the Prac exercises (unless these changes are trivial). Note that you only need to configure, compile and install your PostgreSQL server once for this assignment. All subsequent compilation takes place in the `src/tutorial` directory, and only requires modification of the files there.

Once you have re-installed your PostgreSQL server, you should run the following commands:

```
$ cd PG_CODE/src/tutorial
$ cp complex.c email.c
$ cp complex.source email.source
```

Once you've made the `email.*` files, you should also edit the `Makefile` in this directory and add the green text to the following lines:

```
MODULES = complex funcs email
DATA_built = advanced.sql basics.sql complex.sql funcs.sql syscat.sql email.sql
```

The rest of the work for this assignment involves editing only the `email.c` and `email.source` files. In order for the `Makefile` to work properly, you must use the identifier `_OBJWD_` in the `email.source` file to refer to the directory holding the compiled library. You should never modify directly the `email.sql` file produced by the `Makefile`. Place *all* of you C code in the `email.c` file; do not create any other `*.c` files.

Note that your submitted versions of `email.c` and `email.source` should not contain any references to the `complex` type. Make sure that the documentation (comments in program) describes the code that *you* wrote.

The Email Address Data Type

We wish to define a new base type `EmailAddr` to represent RFC822-style email addresses (actually a subset of RFC822). We also aim to define a useful set of operations on values of type `EmailAddr` and wish to be able to create indexes on `EmailAddr` attributes. How you represent `EmailAddr` values internally, and how you implement the functions to manipulate them internally, is up to you. However, they must satisfy the requirements below.

Once implemented correctly, you should be able to use your PostgreSQL server to build the following kind of SQL applications:

```
create table UserSessions (
   username  EmailAddr,
   loggedIn  timestamp,
   loggedOut timestamp
   -- etc. etc.
);

insert into UserSessions(username,loggedIn,loggedOut) values
('jas@cse.unsw.edu.au','2012-07-01 15:45:55','2012-07-01 15:51:20'),
('jas@cse.unsw.EDU.AU','2012-07-01 15:50:30','2012-07-01 15:53:15'),
('z9987654@unsw.edu.au','2012-07-01 15:51:10','2012-07-01 16:01:05'),
('m.mouse@disney.com','2012-07-01 15:51:11','2012-07-01 16:01:06'),
('a-user@fast-money.com','2012-07-01 15:52:25','2012-07-01 16:10:15');

create index on UserSessions using hash (username);

select a.username, a.loggedIn, b.loggedIn
from   UserSessions a, UserSessions b
where  a.uname = b.uname and a.loggedIn <> b.loggedIn;

select path,count(*)
from   Users
group  by path;
```

Having defined a hash-based file structure, we would expect that the queries would make use of it. You can check this by adding the keyword `EXPLAIN` before the query, e.g.

```
db=# explain analyze select * from UserSessions where username='a@b.com';
```

which should, once you have correctly implemented the data type and loaded sufficient data, show that an index-based scan of the data is being used.

Email address values

The precise format of email addresses is defined in the RFC822 standard, which describes the overall structure of emails (headers, content format, etc.), including addresses. An alternative, and perhaps more user-friendly, description of email addresses can be found on Wikipedia. We will implement a subset of the RFC822 address standard, which can described as follows:

- an email address has two parts, Local and Domain, separated by an '@' char
- both the Local part and the Domain part consist of a sequence of Words, separated by '.' characters
- the Local part has one or more Words; the Domain part has two or more Words (i.e. at least one '.')
- each Word is a sequence of one or more characters, starting with a letter
- each Word ends with a letter or a digit
- between the starting and ending chars, there may be any number of letters, digits or hyphens ('-')

A more precise definition can be given using a BNF grammar:

```
EmailAddr ::= Local '@' Domain

Local         ::= NamePart NameParts

Domain        ::= NamePart '.' NamePart NameParts

NamePart      ::= Letter | Letter NameChars (Letter|Digit)

NameParts     ::= Empty | '.' NamePart NameParts

NameChars     ::= Empty | (Letter|Digit|'-') NameChars

Letter        ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... 'Z'
Digit         ::= '0' | '1' | '2' | ... | '8' | '9'
```

You may assume that the maximum length of the Local part is 128 chars, and that the maximum length of the Domain part is also 128 chars.

Under this syntax, the following are valid email addresses:

```
jas@cse.unsw.edu.au
john-shepherd@hotmail.com
john.a.shepherd@gmail.com
J.Shepherd@unsw.edu.au
j.a.shepherd@acm.org
j-a-shepherd@bargain-hunter.com
jas@a-very-silly-domain.org
```

```
john1988@my-favourite.org
x-1@gmail.com
a@b.c.com
```

The following addresses are not valid in our system, despite some of them being allowed under RFC822:

```
jas@cse
"jas"@cse.unsw.edu.au
j..shepherd@funny.email.org
123jas@marketing.abc.com
john@123buynow.com.au
john@cse.unsw@edu.au
x--@gmail.com
```

Think about why each of the above is invalid in terms of the syntax definition For example, `jas@cse` is invalid because the domain part is required to have at least two dot-separated components (in our system, we are dealing only with full domain names). Another example: `123jas` is invalid because each part of a name must start with a letter.

Important: for this assignment, we define an ordering on email addresses as follows:

- the ordering is determined initially by the ordering on the Domain parts
- if the Domains are equal, then the ordering is determined by the Local parts
- ordering of parts is determined lexically and is case-insensitive

There are examples of how this works in the section on Operations on email addresses below.

Representing email addresses

The first thing you need to do is to decide on an internal representation for your `EmailAddr` data type. You should do this, however, after you have looked at the description of the operators below, since what they require may affect how you decide to structure your internal `EmailAddr` values. You should also take into account that email addresses are to be treated as case-insensitive. E.g. all of the following addresses are equivalent:

```
jas@cse.unsw.edu.au   jas@cse.unsw.EDU.AU   JAS@cse.unsw.EDU.AU
JaS@CsE.UnSw.eDu.aU   jAs@CSE.UNSW.edu.au   jas@CSE.unsw.EDU.au
```

The *canonical form* of an email address is one in which all letters are lower-case, e.g. the first address above. At the SQL level, email addresses are entered and displayed as SQL strings. These strings will not necessarily be in canonical form when they are entered via a database client.

When you read strings representing `EmailAddr` values, they are converted into your internal form, stored in the database in this form, and operations on `EmailAddr` values are carried out using this data structure. When you display `EmailAddr` values, you should show them in canonical form, regardless of how they were entered or how they are stored.

The first functions you need to write are ones to read and display values of type `EmailAddr`. You should write analogues of the functions `complex_in()`, `complex_out` that are defined in the file `complex.c`. Make sure that you use the `V1` style function interface (as is done in `complex.c`).

Note that the two input/output functions should be complementary, meaning that any string displayed by the output function must be able to be read using the input function. There is no requirement for you to retain the precise string that was used for input (e.g. you could store the `EmailAddr` value internally in canonical form).

If you want to use an open-source email address parser from the Net, that's fine, but you will most likely need to modify it so that it only accepts valid addresses as defined by the grammar above. If you include in your submission an email parser and if it accepts email addresses that are not valid according to our grammar, you will lose marks. You must insert all of the parser code in the `email.c` file.

Note that you are *not* required to define binary input/output functions, called `receive_function` and `send_function` in the PostgreSQL documentation, and called `complex_send` and `complex_recv` in the `complex.c`file.

A possible representation for `EmailAddr` values would involve fixed-size buffer(s), large enough to hold the longest possible email address: 256 + 256 + '@' + ... However, this is a poor representation because most email addresses are much shorter than 512+ characters. Using a fixed-size representation for `EmailAddr` limits your maximum possible mark to 7/12.

Operations on email addresses

You must implement all of the following operations for the `EmailAddr` type:

- ***$Email_1$ = $Email_2$*** ... two email addresses are equivalent

  Two email addresses are equivalent if, in their canonical forms, they have the same Local part and the same Domain part.

  ```
  Email_1: jas@cse.unsw.edu.au
  Email_2: jas@cse.unsw.EDU.AU
  Email_3: jas@abc.mail.com
  Email_4: richard@cse.unsw.EDU.AU

  (Email_1 = Email_1) is true
  (Email_1 = Email_2) is true
  (Email_2 = Email_1) is true        (commutative)
  (Email_2 = Email_3) is false
  (Email_2 = Email_4) is false
  ```

- ***$Email_1$ > $Email_2$*** ... the first email address is greater than the second

  $Email_1$ is greater than *$Email_2$* if, in their canonical forms, the Domain part of *$Email_1$* lexically greater than the Domain part of *$Email_2$*. If the Domain parts are equal, then *$Email_1$* is greater than *$Email_2$* if the Local part of *$Email_1$* is lexically greater than the Local part of *$Email_2$*.

  ```
  Email_1: jas@cse.unsw.edu.au
  Email_2: jas@cse.unsw.EDU.AU
  Email_3: jas@abc.mail.com
  Email_4: richard@cse.unsw.EDU.AU

  (Email_1 > Email_2) is false
  (Email_1 > Email_3) is true
  (Email_2 > Email_3) is true
  (Email_1 > Email_1) is false
  (Email_4 > Email_3) is true
  ```

- ***$Email_1$ ~ $Email_2$*** ... Email addresses have the same Domain (note: the operator is a tilde, *not* a minus sign)

  Two email addresses come from the same Domain if their Doman parts are equal.

  ```
  Email_1: jas@cse.unsw.edu.au
  Email_2: jas@cse.unsw.EDU.AU
  Email_3: jas@abc.mail.com
  Email_4: richard@cse.unsw.EDU.AU

  (Email_1 ~ Email_1) is true
  (Email_1 ~ Email_2) is true
  (Email_2 ~ Email_3) is false
  (Email_3 ~ Email_2) is false       (commutative)
  (Email_2 ~ Email_4) is true
  ```

- Other operations: **`<>`**, **`>=`**, **`<`**, **`<=`**, **`!~`**

  You should also implement the above operations, whose semantics is hopefully obvious from the three descriptions above. The operators can typically be implemented quite simply in terms of the first three operators.

Hint: test out as many of your C functions as you can *outside* PostgreSQL (e.g. write a simple test driver) before you try to install them in PostgreSQL. This will make debugging much easier.

You should ensure that your definitions *capture the full semantics of the operators* (e.g. specify commutativity if the operator is commutative). You should also ensure that you provide sufficient definitions so that users of the `EmailAddr` type can create hash-based indexes on an attribute of type `EmailAddr`.

Submission

You need to submit two files: `email.c` containing the C functions that implement the internals of the `EmailAddr` data type, and `email.source` containing the template SQL commands to install the `EmailAddr` data type into a PostgreSQL server. Do not submit the `email.sql` file, since it contains absolute file names which are not helpful in our test environment. If your system requires other `*.c` files, you should submit them along with the modified `Makefile` from the `src/tutorial` directory.

Have fun, *jas*