Assignment 1

Adding a PersonName Data Type to PostgreSQL

Last updated: **Tuesday 15th March 9:56pm**
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, formatting functions, and the ability to build indexes on values of the type.

Summary

Deadline ~~Friday 18 March~~ Monday 21 March, 9:00pm

Pre-requisites: before starting this assignment, it would be useful to complete Prac Work P04

Late Penalty: 0.04 *marks* off the ceiling mark for each *hour* late

Marks: This assignment contributes **15 marks** toward your total mark for this course.

Submission: Webcms3 > Assignments > Ass1 Submission > Make Submission
or, on CSEmachines, `give cs9315 ass1 pname.c pname.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 `/localstorage/YOU/postgresql-14.1/`)
- *`PG_HOME`* ... the directory where you have installed the PostgreSQL binaries (typically `/localstorage/YOU/pgsql/bin/`)
- *`PG_DATA`* ... the directory where you have placed PostgreSQL's `data` (typically `/localstorage/YOU/pgsql/data/`)
- *`PG_LOG`* ... the file where you send PostgreSQL's log output (typically `/localstorage/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 people's names. "Hmmm", you say, "but aren't they just text strings, typically implemented as two attributes, one for family name and one for given names?". That may be true, but making names into a separate base data type allows us to explore how we store and manipulate them.

One common way of writing names (e.g. used in UNSW student systems) is

```
Shepherd,John Andrew
Lin,Xuemin
Eilish,Billie
Martin, Eric Andre
Lakshminarasimhan,Venkateswaran Chandrasekara
Marshall-Martin, Sally Angela
Featherstone,Albert Basil Ernest George Harold Randolph William
i.e.
FamilyName,GivenNames
```

We give a more precise description of what text strings are valid `PersonName`s below.

Adding Data Types in PostgreSQL

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

- 37.13 User-defined Types
- 37.10 C-Language Functions
- 37.14 User-defined Operators
- SQL: CREATE TYPE
- SQL: CREATE OPERATOR
- SQL: CREATE OPERATOR CLASS

Section 37.13 uses an example of a complex number type, which you can use as a starting point for defining your `PersonName` 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 PersonName data type. For example, many of these data types are fixed-size, while `PersonName`s are variable-sized. A potentially useful example of implementing variable-sized types can be found in:

- `PG_CODE/src/tutorial/funcs.c` ... implementation of several data types

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 pname.c
$ cp complex.source pname.source
```

Note the `pname.*` files will contain *many* references to `complex`; I do not want to see any remaining occurences of the word `complex` in the files that you eventually submit. These files simply provide a template in which you create *your* `PersonName` type.

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

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

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

Note that your submitted versions of `pname.c` and `pname.source` should not contain any references to the `complex` type. Make sure that the documentation (comments in program) describes the code that *you* wrote. Leaving the word `complex` anywhere in a `pname.*` file will cost 1 mark.

The Person Name Data Type

We wish to define a new base type `PersonName` to represent people's names, in the format *`FamilyName,GivenNames`*. We also aim to define a useful set of operations on values of type `PersonName` and wish to be able to create indexes on `PersonName` attributes. How you represent `PersonName` 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 Students (
   zid       integer primary key,
   name      PersonName not null,
   degree    text,
   -- etc. etc.
);

insert into Students(zid,name,degree) values
(9300035,'Shepherd, John Andrew', 'BSc(Computer Science)'),
(5012345,'Smith, Stephen', 'BE(Hons)(Software Engineering)');

create index on Students using hash (name);

select a.zid, a.name, b.zid
from   Students a join Students b on (a.name = b.name);

select family(name), given(name), show(name)
from   Students;

select name,count(*)
from   Students
group  by name;
```

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 Students where name='Smith,John';
```

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. Note that this will only be evident if you use a large amount of data (e.g. one of the larger test data samples to be provided).

Person Name values

Valid `PersonName`s will have the above format with the following qualifications:

- there may be a single space after the comma
- there will be **no** people with just one name (e.g. *no* Prince, Jesus, Aristotle, etc.)
- there will be **no** numbers (e.g. *no*Gates, William 3rd)
- there will be **no** titles (e.g. *no* Dr, Prof, Mr, Ms)
- there will be **no** initials (e.g. *no* Shepherd,John A)

In other words, you can ignore the possibility of certain types of names while implementing your input and output functions.

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

```
PersonName ::= Family','Given | Family', 'Given

Family     ::= NameList
Given      ::= NameList

NameList   ::= Name | Name' 'NameList

Name       ::= Upper Letters

Letter     ::= Upper | Lower | Punc

Letters    ::= Letter | Letter Letters

Upper      ::= 'A' | 'B' | ... | 'Z'
Lower      ::= 'a' | 'b' | ... | 'z'
Punc       ::= '-' | "'"
```

You should not make any assumptions about the maximum length of a `PersonName`.

Under this syntax, the following are valid names:

```
Smith,John
Smith, John
O'Brien, Patrick Sean
Mahagedara Patabendige,Minosha Mitsuaki Senakasiri
I-Sun, Chen Wang
Clifton-Everest,Charles Edward
```

The following names are *not* valid in our system:

```
Jesus                     # no single-word names
Smith  ,  Harold          # space before the ","
Gates, William H., III    # no initials, too many commas
A,B C                     # names must at least 2 letters
Smith, john               # names begin with an upper-case letter
```

Think about why each of the above is invalid in terms of the syntax definition.

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

- the ordering is determined initially by the ordering on the Family Name
- if the Family Names are equal, then the ordering is determined by the Given Names
- ordering of parts is determined lexically

There are examples of how this works in the section on Operations on `PersonName`s below.

Representing Person Names

The first thing you need to do is to decide on an internal representation for your `PersonName` 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 `PersonName` values.

When you read strings representing `PersonName` values, they are converted into your internal form, stored in the database in this form, and operations on `PersonName` values are carried out using this data structure. It is useful to define a *canonical form* for names, which may be slightly different to the form in which they are read (e.g. "Smith, John" might be rendered as "Smith,John"). When you display `PersonName` 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 `PersonName`. You should write analogues of the functions `complex_in()`, `complex_out` that are defined in the file `complex.c`. Call them, e.g., `pname_in()` and `pname_out()`. 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 `PersonName` value internally in a different form such as splitting it into two strings: one for the family name(s), and one for the given name(s)).

One thing that `pname_in()` must do is determine whether the name has the correct structure (according to the grammar above). Your `pname_out()` should display each name in a format that can be read by `pname_in()`.

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.

As noted above, you cannot assume anything about the maximum length of names. If your solution uses two fixed-size buffers (one for family, one for given) then your mark is limited to 6/10.

Operations on person names

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

- ***$PersonName_1$ = $PersonName_2$*** ... two names are equal

  Two `PersonName`s are equivalent if, they have the same family name(s) and the same given name(s).

  ```
  PersonName1: Smith,John
  PersonName2: Smith, John
  PersonName3: Smith, John David
  PersonName4: Smith, James

  (PersonName1 = PersonName1) is true
  (PersonName1 = PersonName2) is true
  (PersonName2 = PersonName1) is true        (commutative)
  (PersonName2 = PersonName3) is false
  (PersonName2 = PersonName4) is false
  ```

- ***$PersonName_1$ > $PersonName_2$*** ... the first `PersonName` is greater than the second

  $PersonName_1$ is greater than *$PersonName_2$* if the Family part of *$PersonName_1$* is lexically greater than the Family part of *$PersonName_2$*. If the Family parts are equal, then *$PersonName_1$* is greater than *$PersonName_2$* if the Given part of *$PersonName_1$* is lexically greater than the Given part of *$PersonName_2$*.

  ```
  PersonName1: Smith,James
  PersonName2: Smith,John
  PersonName3: Smith,John David
  PersonName4: Zimmerman, Trent

  (PersonName1 > PersonName2) is false
  (PersonName1 > PersonName3) is false
  (PersonName3 > PersonName2) is true
  (PersonName1 > PersonName1) is false
  (PersonName4 > PersonName3) is true
  ```

- Other operations: `<>`, `>=`, `<`, `<=`

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

- **family(*PersonName*)** returns just the Family part of a name

  ```
  PersonName1: Smith,James
  PersonName2: O'Brien,Patrick Sean
  PersonName3: Mahagedara Patabendige,Minosha Mitsuaki Senakasir
  PersonName4: Clifton-Everest,David Ewan

  family(PersonName1) returns "Smith"
  family(PersonName2) returns "O'Brien"
  family(PersonName3) returns "Mahagedara Patabendige"
  family(PersonName4) returns "Clifton-Everest"
  ```

- **given(*PersonName*)** returns just the Given part of a name

  ```
  PersonName1: Smith,James
  PersonName2: O'Brien,Patrick Sean
  PersonName3: Mahagedara Patabendige,Minosha Mitsuaki Senakasir
  PersonName4: Clifton-Everest,David Ewan

  given(PersonName1) returns "James"
  given(PersonName2) returns "Patrick Sean"
  given(PersonName3) returns "Minosha Mitsuaki Senakasir"
  given(PersonName4) returns "David Ewan"
  ```

- **show(*PersonName*)** returns a displayable version of the name

  It appends the entire Family name to the first Given name (everything before the first space, if any), separated by a single space.

  ```
  PersonName1: Smith,James
  PersonName2: O'Brien,Patrick Sean
  PersonName3: Mahagedara Patabendige,Minosha Mitsuaki Senakasir
  PersonName4: Clifton-Everest,David Ewan
  PersonName5: Bronte,Greta-Anna Maryanne

  show(PersonName1) returns "James Smith"
  show(PersonName2) returns "Patrick O'Brien"
  show(PersonName3) returns "Minosha Mahagedara Patabendige"
  show(PersonName4) returns "David Clifton-Everest"
  show(PersonName5) returns "Greta-Anna Bronte"
  ```

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 `PersonName` type can create **hash-based** indexes on an attribute of type `PersonName`.

Submission

You need to submit two files: `pname.c` containing the C functions that implement the internals of the `PersonName` data type, and `pname.source` containing the template SQL commands to install the `PersonName` data type into a PostgreSQL server. Do not submit the `pname.sql` file, since it contains absolute file names which are not helpful in our test environment.

Have fun, *jas*