Prac Exercise 04

Adding New Data Types to PostgreSQL

Aims

This exercise aims to get you to:

- explore the mechanisms provided by PostgreSQL for adding user-defined types
- add a domain, an enumerated type, and a new base type

You ought to get it done before the end of week 3.

Background

One thing that PostgreSQL does better than many other DBMSs, is to provide well-defined and relatively easy-to-use mechanisms for adding new data types. PostgreSQL's view of data types is the standard abstract data type view; a type is a domain of *values* and a collection of *operators* on those values. In addition, the existence of an ordering on the values of a data type and operations that use the ordering allow indexes to be built on attributes of that type. PostgreSQL has several distinct kinds of types:

base types	defined via C functions, and providing genuine new data types; built-in types such as `integer`, `date` and `varchar(n)` are base types; users can also define new base types;
domains	data types based on a constrained version of an existing data type;
enumerated ty…	defined by enumerating the values of the type; values are specified as a list of strings, and an ordering is defined on the values based on the order they appear in the list;
composite types	these are essentially *tuple types*; a composite type is composed of a collection of named fields, where the fields can have different types; a composite type is created implicitly whenever a table is defined, but composite types can also be defined without the need for a table;
polymorphic ty…	define *classes* of types (e.g. `anyarray`), and are used primarily in the definition of polymorphic functions;
pseudo-types	special types (such as `trigger`) used internally by the system; polymorphic types are also considered to be pseudo-types.

In this exercise, we'll look at domains, enumerated types and base types. Assignment 1, which this exercise leads into, is concerned only with base types.

Setup

Re-start your PostgreSQL server on Grieg (reminder: Prac Exercise 01 and don't forget to `source` the `env` file). Create an empty database called `p04`.

Exercises

In the first exercise, we will create a *domain* and an *enumerated type* for a similar purpose, and examine the differences. In the second exercise we will look at the process of creating a new *base type*.

Exercise #1

Consider the problem of defining a data type for the days of the week. We will generally want to represent the days by their names, e.g.

```
Monday  Tuesday  Wednesday  Thursday  Friday  Saturday  Sunday
```

We also normally want some kind of ordering to indicate the order in which days occur, although it is an open question (application specific) which day starts the week. Let's assume that, as above, we start with `Monday` and we will use the above ordering of day names.

Create the following domains and tables in your `p04` database.

The day names are best represented in SQL as strings, so we need a new type that can be represented by a set of strings. There are two ways to produce a type like this in PostgreSQL:

```
create domain Days1 as varchar(9)
       check (value in ('Monday','Tuesday','Wednesday',
                        'Thursday','Friday','Saturday','Sunday'));

create type Days2 as enum
       ('Monday','Tuesday','Wednesday',
        'Thursday','Friday','Saturday','Sunday');
```

Now define a pair of tables that are identical, except that one uses the domain and the other uses the enumerated type:

```
create table Log1 ( name text, day Days1, starting time, ending time );
create table Log2 ( name text, day Days2, starting time, ending time );
```

Populate the tables via the following two commands:

```
copy Log1 (name, day, starting, ending) from '/web/cs9315/20T1/pracs/p04/LogData';
copy Log2 (name, day, starting, ending) from '/web/cs9315/20T1/pracs/p04/LogData';
```

Examine the contents of the tables via `select` statements and then run the following two commands:

```
select * from Log1 where name='John' order by day;
select * from Log2 where name='John' order by day;
```

结果是按照字母表的顺序来的

结果是按照星期1234567的顺序来的

Explain why they are different. Comment on which kind of data type definition is more appropriate in this context.

Exercise #2

In order to define a new base data type, a user needs to provide:

- input and output functions (in C) for values of the type
- C data structure definitions to represent type values internally
- an SQL definition for the type, giving its length, alignment and i/o functions
- SQL definitions for operators on the type
- C functions to implement the operators

The methods for defining the various aspects of a new base type are given in the following sections of the PostgreSQL manual:

- 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, and you would be well advised to at least take a quick look at it before proceeding. This example is available in the directory `/srvr/`*`YOU`*`/postgresql-12.1/src/tutorial`. You should change into that directory now. You will find two files relevant to the definition of the complex number type: `complex.c` and `complex.source`. The `complex.source` file is actually a template that will be converted to an SQL file when you run `make` in the `tutorial` directory. Run the `make` command now. The output should look something like ...

```
$ cd /srvr/YOU/postgresql-12.1/src/tutorial
$ make
rm -f advanced.sql; \
        C=`pwd`; \
        sed -e "s:_OBJWD_:$C:g" < advanced.source > advanced.sql
rm -f basics.sql; \
        C=`pwd`; \
        sed -e "s:_OBJWD_:$C:g" < basics.source > basics.sql
rm -f complex.sql; \
        C=`pwd`; \
```

```
        sed -e "s:_OBJWD_:$C:g" < complex.source > complex.sql
rm -f funcs.sql; \
        C=`pwd`; \
        sed -e "s:_OBJWD_:$C:g" < funcs.source > funcs.sql
rm -f syscat.sql; \
        C=`pwd`; \
        sed -e "s:_OBJWD_:$C:g" < syscat.source > syscat.sql
gcc -O2 -Wall ...lots of compiler options... -c -o complex.o complex.c
gcc -O2 -Wall ...lots of compiler options... -o complex.so complex.o
gcc -O2 -Wall ...lots of compiler options... -c -o funcs.o funcs.c
gcc -O2 -Wall ...lots of compiler options... -o funcs.so funcs.o
```

If `make` produces errors ... are you logged in to `grieg`? ... have you set your environment (`env`)?

The relevant lines above are the ones that mention `complex` (in red). Make sure that you read and understand exactly what is being done here. The first red command creates the `complex.sql` file from the `complex.source` file by filling in the appropriate directory name so that PostgreSQL knows where to find the libraries. The second and third red commands create a library file called `complex.so` containing all the C functions which implement the low-level operations on the `Complex` data type.

Once you have made the complex number library, and while still in the `src/tutorial` directory, start a `psql` session on a `test` database and run the `complex.sql` file as follows:

```
$ createdb test
$ psql test
psql (12.1)
Type "help" for help.

test=# \i complex.sql
psql:complex.sql:39: NOTICE:  type "complex" is not yet defined
DETAIL:  Creating a shell type definition.
CREATE FUNCTION
psql:complex.sql:47: NOTICE:  argument type complex is only a shell
CREATE FUNCTION
psql:complex.sql:55: NOTICE:  return type complex is only a shell
CREATE FUNCTION
psql:complex.sql:63: NOTICE:  argument type complex is only a shell
CREATE FUNCTION
CREATE TYPE
CREATE TABLE
INSERT 0 1
INSERT 0 1
     a     |       b
-----------+---------------
 (1,2.5)   | (4.2,3.55)
 (33,51.4) | (100.42,93.55)
(2 rows)

CREATE FUNCTION
CREATE OPERATOR
        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
(2 rows)

    aa     |       bb
-----------+---------------
 (2,3.5)   | (5.2,4.55)
 (34,52.4) | (101.42,94.55)
(2 rows)

DROP TABLE
psql:complex.sql:228: NOTICE:  drop cascades to 19 other objects
DETAIL:  drop cascades to function complex_in(cstring)
```

```
drop cascades to function complex_out(complex)
... etc etc etc ...
... etc etc etc ...
... etc etc etc ...
drop cascades to operator >(complex,complex)
drop cascades to function complex_abs_cmp(complex,complex)
drop cascades to operator class complex_abs_ops for access method btree
DROP TYPE
test=#
```

The `complex.sql` file sets up the `Complex` type, creates a table that uses the type and then runs some operations to check that it's working. After the testing, it removes the `Complex` type. If you want this type to remain in the `test` database, then you should edit `complex.sql` file and remove the following lines at the end of the file (or simply comment them out).

```
DROP TABLE test_complex;
DROP TYPE complex CASCADE;
```

and then re-enter the `test` database and re-run the `complex.sql` script. This will leave you with a database containing a `Complex` number type and table containing values of that type. You can explore the various operations on the type. Note that you can also create other databases and use the new `Complex` number type in them. The `Complex` type is now included in your PostgreSQL server in much the same way as built-in types like `date`, `integer` and `text`.

Once you have a feel for how the `Complex` type behaves from the SQL level, it's time to take a look at the code that implements it. Read the files `complex.sql` and `complex.c` in conjunction with the PostgreSQL manual sections mentioned above. Once you feel confident that you understand how it all fits together, perhaps you could try making some changes to the `Complex` type (e.g. use `[...]` rather than `(...)` to enclose values of type `complex`) and installing them.

If you do plan to change the type (or implement a new type), I would suggest making copies of the original `complex.c` and `complex.source` (e.g. to `mytype.c` and `mytype.source`), and then editing `mytype.c` and `mytype.source`. You will also need to add lines to the `Makefile` to create `mytype.sql` and `mytype.so`. Once you've modified the code, do the following:

- re-run the `make` command to create `mytype.sql` and `mytype.so`
- create a new database (or simply use the `test` database)
- if the new type is called `Complex`, you'll need to drop the old `Complex` type first
- load up the new data type via `\i mytype.sql`
- experiment with values of the new type

End of Prac

Let me know via the forums, or come to a consultation if you have any problems with this exercise ... *jas*