Assignment 1

Adding an GeoCoord Data Type to PostgreSQL

Last updated: **Friday 17th March 10:37pm**

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 Friday 17 March, 9pm

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

Late Penalty: 5% of the max assessment mark per-day reduction, for up to 5 days

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

Submission: Moodle > Assignment > Assignment 1

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-15.1/`)
- *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 **geographical coordinates**. You may implement the functions for the data type in any way you like *provided that* they satisfy the semantics given below (in the The Geographical Coordinate Data Type section).

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

- 38.13 User-defined Types
- 38.10 C-Language Functions
- 38.14 User-defined Operators
- SQL: CREATE TYPE
- SQL: CREATE OPERATOR
- SQL: CREATE OPERATOR CLASS

Section 38.13 uses an example of a complex *number* type, which you can use as a starting point for defining your GeoCoord data type (see below). There are other examples of new data types under the directories:

- *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 GeoCoord 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 gcoord.c
$ cp complex.source gcoord.source
```

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

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

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

Note that your submitted versions of `gcoord.c` and `gcoord.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 Geographical Coordinate Data Type

We wish to define a new base type `GeoCoord` to represent location based on geographical coordinate system. We also aim to define a useful set of operations on values of type `GeoCoord` and wish to be able to create indexes on `GeoCoord` attributes. How you represent `GeoCoord` 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 StoreInfo (
   id  integer primary key,
   location GeoCoord,
   -- etc. etc.
);

insert into StoreInfo(id, location) values
(1,'Sydney,33.86°S,151.21°E'),
(2,'Melbourne,37.84°S,144.95°E');
```

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=# create index on StoreInfo using hash (location);
db=# explain analyze select * from StoreInfo where location='Melbourne,37.84°S,144.95°E';
```

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.

Geographical Coordinate values

The precise format of geographical coordinates is defined as following:

- a geographical coordinate has 3 parts: LocationName, Latitude and Longitude;
- the LocationName has one or more words, the words are seperated by space;
- each Word is a sequence of one or more letters;
- the Latitude or the Longitude has two parts: a coordinate value and a direction;
- the coordinate value is a real number that is greater than or equal to *0 (for simplicity,* you can assume no coordinate value using more than 4 decimal places. In other words, you can ignore the possibility of processing coordinate value with long decimal places);
- the coordinate value of the Latitude should be less than or equal to 90;
- the coordinate value of the Longitude should be less than or equal to 180;
- the direction of latitude is either 'N' or 'S'; the direction of longitude is either 'W' or 'E';

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

```
GeoCoord ::=  LocationName ',' Latitude ',' Longitude | LocationName ',' Latitude ' ' Longitude | LocationName ',' Longitude ',' Latitude

LocationName ::= WordList

Latitude ::= CoordValue ''' LatDir

LatDir ::= 'N' | 'S'

Longitude  ::=  CoordValue ''' LongDir

LongDir ::= 'W' | 'E'

WordList ::= Word | Word' 'WordList

Word ::= Letter | Letter Word

Letter ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
```


You may assume that the maximum length of the LocationName part is 256 chars.

Under this syntax, the following are valid geographical coordinates:

```
Melbourne,37.84°S,144.95°E
Melbourne,37.84°S 144.95°E
Melbourne,144.95°E,37.84°S
San Francisco,37.77°N,122.42°W
San Francisco,122.42°W,37.77°N
san francisco,122.42°W 37.77°N
```

The following geographical coordinate are not valid in our system.

```
Melbourne,37.845,144.95E
Melbourne,37.84,144.95
Melbourne,37.84
Melbourne:37.84°S,144.95°E
Melbourne ,37.84°S,144.95°E
12Melbourne,37.84°S,144.95°E
Melbourne,-37.84°S,144.95°E
San francisco,122.42°W,37.77
 San Francisco\n,37.77°N,122.4194°W
San francisco,165°W,22°W
37.84°S,144.95°E
```

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

- the ordering is determined initially by the ordering on the latitude and longitude
- if the both are equal, then the ordering is determined by LocationName.
- ordering of LocationName is determined lexically and is case-insensitive.

There are examples of how this works in the section on Operations on Geographical Coordinates below.

Representing Geographical Coordinates

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

When you read strings representing `GeoCoord` values, they are converted into your internal form, stored in the database in this form, and operations on `GeoCoord` values are carried out using this data structure. It is useful to define a canonical form for geographical coordinates, which may be slightly different to the form in which they are read (e.g. "Melbourne,144.95°E 37.84°S" should be rendered as "melbourne,37.84°S,144.95°E"). When you display `GeoCoord` values, you should show them in canonical form, regardless of how they were entered or how they are stored.

The initial functions you need to write are ones to read and display values of type `GeoCoord`. 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 `GeoCoord` value internally in a different form such as splitting it into several parts).

One thing that `gcoord_in()` must do is determine whether the input string has the correct structure (according to the gammmer above). Your `gcoord_out()` should display each geographical coordinate in a format that can be read by `gcoord_in()`.

You are not required (but you can) 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 input length of the geographical coordinates. Using a fixed-size representation for `GeoCoord` limits your maximum possible mark to 10/15.

Operations on Geographical Coordinates

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

- ***GeoCoord*$_1$ = *GeoCoord*$_2$** ... two geographical coordinates are equivalent

  Two geographical coordinates are equivalent if, in their canonical forms, they have the same LocationName, Longitude and Latitude.

  ```
  GeoCoord1: Sydney,33.86°S,151.21°E
  GeoCoord2: sydney,151.2100°E 33.8600°S
  GeoCoord3: Sydney,35.96°S,121.78°E
  GeoCoord4: Melbourne,33.86°S,151.21°E

  (GeoCoord1 = GeoCoord1) is true
  (GeoCoord1 = GeoCoord2) is true
  (GeoCoord1 = GeoCoord3) is true       (commutative)
  (GeoCoord3 = GeoCoord1) is false
  (GeoCoord1 = GeoCoord4) is false
  ```

- ***GeoCoord*$_1$ > *GeoCoord*$_2$** ... the geographical coordinates is greater than the second

  GeoCoord$_1$ is greater than *GeoCoord*$_2$ if, the latitude of *GeoCoord*$_1$ is closer to equator than the latitude of *GeoCoord*$_2$ (If the value of latitude is the same, then we deduce by the direction, i.e., the one in North is considered to be greater than the one in South).

  If the latitude are equal, then *GeoCoord*$_1$ is greater than *GeoCoord*$_2$ if the longitude of *GeoCoord*$_1$ is closer to prime meridian than the longitude of *GeoCoord*$_2$ (If the value of longitude is the same, then we deduce by the direction, i.e., the one in East is considered to be greater than the one in West).

  If both of the latitude and longitude are equal, then *GeoCoord*$_1$ is greater than *GeoCoord*$_2$ if the LocationName of *GeoCoord*$_1$ is lexically greater than the LocationName of *GeoCoord*$_2$.

  ```
  GeoCoord1: Sydney,33.86°S,151.21°E
  GeoCoord2: Sydney,35.86°S 150.21°E
  GeoCoord3: sydney,35.86°S,162.78°E
  GeoCoord4: melbourne,33.86°S 151.21°E

  (GeoCoord1 > GeoCoord2) is true
  (GeoCoord2 > GeoCoord3) is true
  (GeoCoord3 > GeoCoord1) is true
  (GeoCoord1 > GeoCoord3) is false
  (GeoCoord4 > GeoCoord1) is true
  (GeoCoord1 > GeoCoord4) is true
  ```

- ***GeoCoord*$_1$ ~ *GeoCoord*$_2$** ... Geographical Coordinates are in the same time zone. (*note:* the operator is a tilde, *not* a minus sign)

  The time zone of a geographical coordinate is determined by the longitude. For simplicity, we define the time zone as the floor value of longitude divided by 15. (**Note you should also consider the direction of longitude.**)

  ```
  GeoCoord1: Sydney,33.86°S,151.21°E
  GeoCoord2: Chuuk Islands,5.4579,153.51°E
  GeoCoord3: Melbourne,37.84°S,144.95°E
  GeoCoord4: Alaska,70.2°S,144.0°W

  (GeoCoord1 ~ GeoCoord1) is true
  (GeoCoord1 ~ GeoCoord3) is true
  (GeoCoord1 ~ GeoCoord2) is true  (commutative)
  (GeoCoord1 ~ GeoCoord4) is false
  (GeoCoord2 ~ GeoCoord3) is false
  (GeoCoord2 ~ GeoCoord4) is true
  ```

- ***Convert2DMS(GeoCoord*$_1$*)*** ... Convert decimal Geographical Coordinates into DMS.

  It converts the geographical coordinates from decimal to DMS(degree, minute, second). The calculation can be described as:
  D = floor($|D_{dec}|$),
  M = floor(60 × $|D_{dec}$-D|),
  S = floor(3600 × $|D_{dec}$-D| - 60 × M),
  where D_{dec} represents the CoordValue in Latitude and Longitude of the *GeoCoord* and the definition of the floor function can be found here floor. If the value of M or S is 0, then it will not be displayed. The output should be shown in the canonical form you defined, where only the CoordValue is converted into DMS.

  ```
  GeoCoord1: sydney,33.86°S,151.21°E
  GeoCoord2: sydney,151°E 33°S
  GeoCoord3: melbourne,37.8°S,144.2°E

  Convert2DMS(GeoCoord1) returns "sydney,33°51'36"S,151°12'36"E"
  Convert2DMS(GeoCoord2) returns "sydney,151°E 33°S"
  Convert2DMS(GeoCoord3) returns "melbourne,37°48'S,144°12'E"
  ```

- 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.

Testing

You can testing your solution by writing some simple SQL codes. As a reference, we also have written some scripts for testing. The tutorial to use that can be found in the testing page. Note that more test cases will be used in marking.

Submission

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

Have Fun~