COMP9315 25T1: Assignment 1

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 20:59:59 21st March (Sydney Time).**

Pre-requisites: Before starting this assignment, it would be useful to complete Prac 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.

Note: 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-15.11/)
- 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 **PostAddress.** You may implement the functions for the data type in any way you like provided that they satisfy the semantics given below (in the Post Address Data Type section).

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:

- 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 PostAddress 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 PostAddress data type. For example, many of these data types are fixed-size, while PostAddress 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 postadd.c
$ cp complex.source postadd.source
```

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

```
MODULES = complex funcs postadd

DATA_built = advanced.sql basics.sql complex.sql funcs.sql syscat.sql postadd.sql
```

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

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

Postal Address Data Type

We wish to define a new base type PostAddress to represent postal system. We also aim to define a useful set of operations on values of type PostAddress and wish to be able to create indexes on PostAddress attributes. How you represent PostAddress 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 Addresses (
        id      integer primary key,
        address  PostAddress not null,
        -- etc. etc.
);

insert into Addresses(id, address) values
(1,'U19/36 Queen Ave, Southgate, AR 7279'),
(2,'U3/100 Victoria Ave, Lakeside, AU 5135');
```

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 Addresses using hash (address);
db=# explain analyze select * from Addresses where address='U19/36 Queen Ave, Southgate, AR 7279';
```

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.

Postal Address Values

Valid PostAddress values is defined as following:

- A PostAddress has 4 parts: DetailedUnitRoad, Suburb, State and Postcode
- The DetailedUnitRoad must include a street name and may include a unit number.
- The unit number includes one letter and digits.
- The street name includes digits and word list.
- The unit number and street name in DetailedUnitRoad are separated by '/'.
- The suburb must consist of a word list.
- The state must be a two-letter uppercase abbreviation.
- The postcode must be a four-digit number.
- There will be no non-numeric postcodes.

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

PostAddress	::= DetailedUnitRoad ', ' Suburb ', ' State ' ' Postcode
DetailedUnitRoad	::= Street | Unit '/' Street
Unit	::= Letter Digits
Street	::= Digits ' ' WordList
Suburb	::= WordList
State	::= Upper Upper
Postcode	::= Digit Digit Digit Digit
WordList	::= Word | Word ' ' WordList
Word	::= Letter | Letter Word
Letter	::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... 'Z'
Upper	::= 'A' | 'B' | ... | 'Z'
Digit	::= '0' | '1' | ... | '9'

Under this syntax, the following are valid postal addresses:

```
U19/36 Queen Ave, Southgate, AR 7279
U3/100 Victoria Ave, Lakeside, AU 5135
B33/240 Emerald Forest HWY, Hillsborough, NO 5865
240 Emerald Forest HWY, Hillsborough, NO 5865
```

The following addresses are not valid in our system:

```
U19/   36 Queen Ave, Southgate, AR 7279      # Having space before or after '/'
U19/Queen Ave, Southgate, AR 7279            # Missing street number
U19/36 Queen Ave, Southgate, A 7279          # Incorrect state abbreviation
240 Emerald Forest HWY, Hillsborough, N3 5865
U19/36 Queen Ave, Southgate, AR7279
                                  # Missing space between state abbreviation and postcode
U19/36 Queen Ave, Southgate, AR 72A9         # Non-numeric postcode
19/36 Queen Ave, Southgate, AR 7279          # Unit format error
240 Emerald Forest HWY Hillsborough NO 5865
240 Emerald Forest HWY, Hillsborough, no 5865
```

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

- The ordering is determined initially by the state name.
- If the state names are equal, then the ordering is determined by the suburb name.
- If the suburb names are equal, then the ordering is determined by the street.
- If street names are equal, the ordering is determined by the unit number.
- Ordering of parts is determined lexically and case-insensitive.

Representing Postal Addresses

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

When you read strings representing **PostAddress** values, they are converted into your internal form, stored in the database in this form, and operations on **PostAddress** values are carried out using this data structure. It is useful to define a canonical form for addresses, which may be slightly different from the form in which they are read (e.g. "U19/36 Queen Ave, Southgate, AR 7279" might be rendered as "U19/36 Queen Ave,Southgate,ar 7279"). When you display **PostAddress** values, you should display them exactly in the same way they are input, regardless of how they are stored.

The first functions you need to write are ones to read and display values of type **PostAddress**. 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 **PostAddress** value internally in a different form such as splitting it into several parts). One thing that **postadd_in()** must do is determine whether the input string has the correct structure (according to the grammar above). Your **postadd_out()** should display each postal address in a format that can be read by **postadd_in()**.

You are not required (but you can) 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 postal addresses. **Using a fixed-size representation for PostAddress limits your maximum possible mark to 10/15.**

Operations on Postal Addresses

You must implement all of the following operations for the PostAddress type:

❖ **PostAddress1 = PostAddress2** ... two postal addresses are equivalent

Two postal addresses are equivalent if, in their canonical forms, they have the same DetailedUnitRoad, Suburb, and State, considering case insensitivity.

PostAddress1: U19/36 Queen Ave, Southgate, AR 7279

PostAddress2: u19/36 queen ave, southgate, AR 7279

PostAddress3: U33/240 Emerald Forest HWY, Hillsborough, NO 5865

PostAddress4: U19/36 Queen Ave, Lakeside, AR 7279

(PostAddress1 = PostAddress1) is true

(PostAddress1 = PostAddress2) is true

(PostAddress2 = PostAddress1) is true (Commutative)

(PostAddress2 = PostAddress3) is false

(PostAddress2 = PostAddress4) is false

❖ **PostAddress1 > PostAddress2** ... the postal address is greater than the second

PostAddress1 is greater than PostAddress2 if:

- The State of PostAddress1 is lexically greater than the State of PostAddress2.
- If the State names are equal, then the Suburb is compared lexically.
- If the Suburb names are equal, then the Street name is compared lexically.
- If the Street names are equal, then the Unit number is compared lexically.

All comparisons are case insensitive.

PostAddress1: U19/36 Queen Ave, Southgate, AR 7279

PostAddress2: U3/100 Victoria Ave, Lakeside, AU 5135

PostAddress3: U33/240 Emerald Forest HWY, Hillsborough, NO 5865

PostAddress4: 240 Emerald Forest HWY, Hillsborough, NO 5865

```
(PostAddress1 > PostAddress2) is false        #(AR < AU)
(PostAddress3 > PostAddress1) is true         #(NO > AR)
(PostAddress3 > PostAddress4) is true
                                  #(Same address, but unit exists in PostAddress3)
(PostAddress1 > PostAddress1) is false
```

❖ **PostAddress1 ~ PostAddress2** ... Postal Addresses are in the same Suburb. (note: the operator is a tilde, not a minus sign)

Two postal addresses are considered in the same suburb if they share the same Suburb (case insensitive).

```
PostAddress1: U19/36 Queen Ave, Southgate, AR 7279
PostAddress2: U3/100 Victoria Ave, Southgate, AU 5135
PostAddress3: U33/240 Emerald Forest HWY, Hillsborough, NO 5865
PostAddress4: 240 Emerald Forest HWY, Hillsborough, NO 5865

(PostAddress1 ~ PostAddress1) is true
(PostAddress1 ~ PostAddress2) is false
(PostAddress2 ~ PostAddress1) is false        #(Commutative)
(PostAddress2 ~ PostAddress3) is false
(PostAddress3 ~ PostAddress4) 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.

❖ **show_postcode(PostAddress)** returns the postcode of the address

PostAddress1: U19/36 Queen Ave, Southgate, AR 7279

PostAddress2: U3/100 Victoria Ave, Lakeside, AU 5135

PostAddress3: U33/240 Emerald Forest HWY, Hillsborough, NO 5865

PostAddress4: 240 Emerald Forest HWY, Hillsborough, NO 5865

show_postcode(PostAddress1) returns "7279"

show_postcode(PostAddress2) returns "5135"

show_postcode(PostAddress3) returns "5865"

show_postcode(PostAddress4) returns "5865"

❖ **show_unit(PostAddress)** returns the unit number of the address

PostAddress1: U19/36 Queen Ave, Southgate, AR 7279

PostAddress2: U3/100 Victoria Ave, Lakeside, AU 5135

PostAddress3: R33/240 Emerald Forest HWY, Hillsborough, NO 5865

PostAddress4: 240 Emerald Forest HWY, Hillsborough, NO 5865

show_unit(PostAddress1) returns "U19"

show_unit(PostAddress2) returns "U3"

show_unit(PostAddress3) returns "R33"

show_unit(PostAddress4) returns "NULL"

❖ **show(PostAddress)** returns a displayable version of the address

For privacy and security reasons, the displayable version of a PostAddress only includes the short street name and the abbreviation name of the state.

PostAddress1: U19/36 Queen Ave, Southgate, AR 7279
PostAddress2: U3/100 Victoria Ave, Lakeside, AU 5135
PostAddress3: U33/240 Emerald Forest HWY, Hillsborough, NO 5865
PostAddress4: 240 Emerald Forest HWY, Hillsborough, NO 5865
show(PostAddress1) returns "Queen Ave, AR"
show(PostAddress2) returns "Victoria Ave, AU"
show(PostAddress3) returns "Emerald Forest HWY, NO"
show(PostAddress4) returns "Emerald Forest HWY, NO"

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 test 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.pdf**. Note that more test cases will be used in marking.

Assignment Submission

Submission

- You are required to submit two files via **Moodle**.
  - **postadd.c** containing the C functions that implement the internals of the PostAddress data type.
  - And **postadd.source** containing the template SQL commands to install the PostAddress data type into a PostgreSQL server.
- Do not submit the postadd.sql file, since it contains absolute file names which do not work in our test environment.

Note:

1. If you have problems relating to your submission, please email to xingyu.tan@unsw.edu.au.
2. If there are issues with Moodle, send your assignment to the above email with the subject title "<zid> COMP9315 Ass1 Submission".

Late Submission Penalty

- 5% of the max mark (15 marks) will be deducted for each additional day.
- Submissions that are more than five days late will not be marked.

Plagiarism

The work you submit must be your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline.

All submissions will be checked for plagiarism. The university regards plagiarism as a form of academic misconduct and has very strict rules. Not knowing the rules will not be considered *a valid* excuse when you are caught.

- For UNSW policies, penalties, and information to help avoid plagiarism, please see: https://student.unsw.edu.au/plagiarism.
- For guidelines in the online ELISE tutorials for all new UNSW students: https://subjectguides.library.unsw.edu.au/elise/plagiarism.