COMP9315 19T2 DBMS Implementation

Assignment 1

Adding a Set Data Type to PostgreSQL

Last updated: **Monday 27th August 9:09pm**
Most recent changes are shown in red;
older changes are shown in brown.

jump to ... summary introduction setup intsets (values operations representing) changelog submission

Summary

Deadline	Sunday 2 September, 11:59pm
Late Penalty	0.083 *marks* off the ceiling mark for each hour late
Marks	This assignment contributes **10 marks** toward your total mark for this course.
Submission	COMP9315 website > *Assignments* > *Ass1 Specification*; then, *Make Submission*, upload `intset.c`, `intset.source` or, from CSE, **`give cs9315 ass1 intset.c intset.source ...`**
Pre-requisites	Before starting this assignment, it would be useful to complete Prac Work P04.

This assignment aims to give you

- an understanding of how data is treated inside a DBMS, and
- 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 and a range of operations.

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`)
`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/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 **sets of integers**. You may implement the functions for the data type in any way you like, *provided that* they satisfy the semantics given below (in the intSets section).

Note that arrays in PostgreSQL have some properties and operations that make them look a little bit like sets. However, they are *not* sets and have quite different semantics to the data type that we are asking you to implement.

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

- 37.11 User-defined Types
- 37.9 C-Language Functions
- 37.12 User-defined Operators
- SQL: CREATE TYPE

- SQL: CREATE OPERATOR
- SQL: CREATE OPERATOR CLASS

Section 37.11 uses an example of a complex number type, which you can use as a starting point for defining your `intSet` 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 `intSet` 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 intset.c
$ cp complex.source intset.source
```

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

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

The rest of the work for this assignment involves editing only the `intset.c` and `intset.source` files. In order for the `Makefile` to work properly, you must use the identifier `_OBJWD_` in the `intset.source` file to refer to the directory holding the compiled library. You should never directly modify the `intset.sql` file produced by the `Makefile`. If you want to use other `*.c` files along with `intset.c`, then you can do so, but you will need to make further changes to the `Makefile` to ensure that they are compiled and linked correctly into the library.

Note that your submitted versions of `intset.c` and `intset.source` should not contain any references to the `complex` type (because that's not what you're implementing). Make sure that the comments in the program describes the code that *you* wrote.

The intSet Data Type

We aim to define a new base type `intSet`, to store the notion of sets of integer values. We also aim to define a useful collection of operations on our `intSets`.

How you represent `intSet` values, and implement functions to manipulate them, 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 Features (
    id primary key,
    name text
);

create table DBSystems (
    name text,
    features intSet
);

insert into Features (name) values
```

```
    (1, 'well designed'),
    (2, 'efficient'),
    (3, 'flexible'),
    (4, 'robust');
insert into DBSystems (name, value) values
    ('MySQL', '{}').
    ('MongoDB', '{}'),
    ('Oracle', '{2,4}'),
    ('PostgreSQL', '{1,2,3,4}');
```

intSet values

In mathematics, we represent a set as a curly-bracketed, comma-separated collection of values: $\{1, 2, 3, 4, 5\}$. Such a set contains only distinct values. No particular ordering can be imposed.

Our `intSet` values can be represented similarly: we can have a comma-separated list of integers, surrounded by a set of curly braces, which is presented to and by PostgreSQL as a string: For example, `'{ 1, 2, 3, 4, 5 }'`.

Whitespace should not matter, so `'{1,2,3}'` and `'{ 1, 2, 3 }'` are equivalent. Similarly, a set contains distinct values, so `'{1,1,1,1,1}'` is equivalent to `'{1}'`. And ordering is irrelevant, so `'{1,2,3}'` is equivalent to `'{3,2,1}'`.

You may not assume a fixed limit to the size of the set. It may contain zero or more elements, bounded by the database's capacity to store values.

Valid intSets	**Invalid intSets**
`'{ }'`	`'{a,b,c}'`
`'{2,3,1}'`	`'{ a, b, c }'`
`'{6,6,6,6,6,6}'`	`'{1, 2.0, 3}'`
`'{10, 9, 8, 7, 6,5,4,3,2,1}'`	`'{1, {2,3}, 4}'`
`'{1, 999, 13, 666, -5}'`	`'{1, 2, 3, 4, five}'`
`'{    1  ,  3  ,  5 , 7,9 }'`	`'{ 1 2 3 4 }'`

Operations on intSets

You must implement all of the following operations for the `intSet` type (assuming A, B, and S are `intSets`, and i is an integer):

`i <@ S`	intSet S contains the integer i; that is, $i \in S$.
`@ S`	give the *cardinality*, or number of distinct elements in, intSet S; that is, $\|S\|$.
`A @> B`	does intSet A contain only values in intSet B? that is, for every element of A, is it an element of B? ($A \subset B$)
`A = B`	intSets A and B are equal; that is, intSet A contains all the values of intSet B and intSet B contains all the values of intSet A, or, every element in A can be found in B, and vice versa.
`A && B`	takes the *set intersection*, and produces an intSet containing the elements common to A and B; that is, $A \cap B$.
`A || B`	takes the *set union*, and produces an intSet containing all the elements of A and B; that is, $A \cup B$.
`A !! B`	takes the *set disjunction*, and produces an intSet containing elements that are in A and not in B, or that are in B and not in A.
`A - B`	takes the *set difference*, and produces an intSet containing elements that are in A and not in B. Note that this is *not* the same as `A !! B`.

Below are examples of how you might use `intSets`, to illustrate the semantics. You can use these as an initial test set; we will supply a more comprehensive test suite later.

```
db=# create table mySets (id integer primary key, iset intSet);
NOTICE:  ...
CREATE TABLE
db=# insert into mySets values (1, '{1,2,3}');
```

```
INSERT 0 1
db=# insert into mySets values (2, '{1,3,1,3,1}');
INSERT 0 1
db=# insert into mySets values (3, '{3,4,5}');
INSERT 0 1
db=# insert into mySets values (4, '{4,5}');
INSERT 0 1
db=# select * from mySets;
 id |  iset
----+---------
  1 | {1,2,3}
  2 | {1,3}
  3 | {3,4,5}
  4 | {4,5}
(4 rows)
-- get all pairs of tuples where th iset in one is a subset of the iset in the other
db=# select a.*, b.* from mySets a, mySets b
db-# where (b.iset @> a.iset) and a.id != b.id;
 id |  iset   | id | iset
----+---------+----+-------
  1 | {1,2,3} |  2 | {1,3}
  3 | {3,4,5} |  4 | {4,5}
(2 rows)
-- insert extra values into the iset in tuple #4 via union
db=# update mySets set iset = iset || '{5,6,7,8}' where id = 4;
UPDATE 1
db=# select * from mySets where id=4;
 id |    iset
----+-------------
  4 | {4,5,6,7,8}
(1 row)
-- tuple #4 is no longer a subset of tuple #3
db=# select a.*, b.* from mySets a, mySets b
db-# where (b.iset @> a.iset) and a.id != b.id;
 id |  iset   | id | iset
----+---------+----+-------
  1 | {1,2,3} |  2 | {1,3}
(1 row)
-- get the cardinality (size) of each intSet
db=# select id, iset, (@iset) as card from mySets order by id;
 id |    iset     | card
----+-------------+------
  1 | {1,2,3}     |    3
  2 | {1,3}       |    2
  3 | {3,4,5}     |    3
  4 | {4,5,6,7,8} |    5
(4 rows)
-- form the intersection of each pair of sets
db=# select a.iset, b.iset, a.iset && b.iset
db-# from mySets a, mySets b where a.id < b.id;
  iset   |    iset     | ?column?
---------+-------------+----------
 {1,2,3} | {1,3}       | {1,3}
 {1,2,3} | {3,4,5}     | {3}
 {1,2,3} | {4,5,6,7,8} | {}
 {1,3}   | {3,4,5}     | {3}
 {1,3}   | {4,5,6,7,8} | {}
 {3,4,5} | {4,5,6,7,8} | {4,5}
(6 rows)
db=# delete from mySets where iset @> '{1,2,3,4,5,6}';
DELETE 3
db=# select * from mySets;
 id |    iset
----+-------------
```

```
  4 | {4,5,6,7,8}
(1 row)
-- etc. etc. etc.
```

You should think of some more tests of your own. In particular, make sure that you check that your code works with large `intSets` (e.g. cardinality ≥ 1000). If you come up with any tests that you think are particularly clever, feel free to post them in th Comments section below.

Representing intSets

The first thing you need to do is to decide on an internal representation for your `intSet` data type. You should do this *after* you have understood the description of the operators above, since what they require may affect how you decide to structure your internal `intSet` values.

Note that because of the requirement that `intSets` can be arbitrarily large (see above), you cannot have a representation that uses a fixed-size object to hold values of type `intSet`.

When you read strings representing `intSet` values, they are converted into your internal form, stored in the database in this form, and operations on `intSet` values are carried out using this data structure. When you display `intSet` values, you should show them in a canonical form, regardless of how they were entered or how they are stored. The canonical form for output (at least) should include no spaces, and should have elements in ascending order.

The first functions you need to write are ones to read and display values of type `intSet`. You should write analogues of the functions `complex_in()` and `complex_out()` that are defined in the file `complex.c`. Suitable names for these functions would be e.g. `intset_in()` and `intset_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 `intSet` value internally in canonical form).

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.

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).

ChangeLog

- **v1.0** (2018-08-13 21:30:47 +1000)
  - released onto unsuspecting students
- **v1.1** (2018-08-14 17:38:21 +1000)
  - fix a typo in the example ('<@' instead of '@>')
  - fix another typo in the example (subset operator is backwards)
  - clarify @> (⊂) operator
  - clarify the set difference and set disjunction operators, '!!' and '-'
  - add a changelog
- **v1.2** (2018-08-27 21:00:15 +1000)
  - fix a lingering typo in example (order of relations)

Submission

You need to submit two files: `intset.c` containing the C functions that implement the internals of the `intSet` data type, and `intset.source` containing the template SQL commands to install the `intSet` data type into a PostgreSQL server. Do *not* submit the `intset.sql` file, since it contains absolute file names which are not useful 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* and *jashank*