Postgres Plus Database Administration

Postgres Plus Advanced Server (PPAS) 9.5

Day 4 - Developer

Postgres Plus Advanced Server (PPAS) 9.5

Day 4-1
Procedural Languages

Objectives

- In this module you will learn:
  - PostgreSQL Procedural Languages
  - Introduction to PL/PGSQL
  - How it works
  - PL/pgSQL Block Structure
  - Declaring Variables
  - Writing Executable Statements
  - Declaring Function Parameters
  - Control Structures
  - Exception Handling
  - PL/pgSQL Cursors
  - Triggers
  - Examples & Lab

4

User-defined Functions

- PostgreSQL provides four kinds of functions:
  - query language functions (functions written in SQL)
  - procedural language functions (functions written in, for example, PL/pgSQL or PL/Tcl)
  - internal functions
  - C-language functions

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Query Language Function

- SQL functions execute an arbitrary list of SQL statements.
- The first row of the last query's result will be returned.

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;

SELECT tf1(17, 100.0);
```

- You can return multiple records using SETOF

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

PostgreSQL Procedural Languages

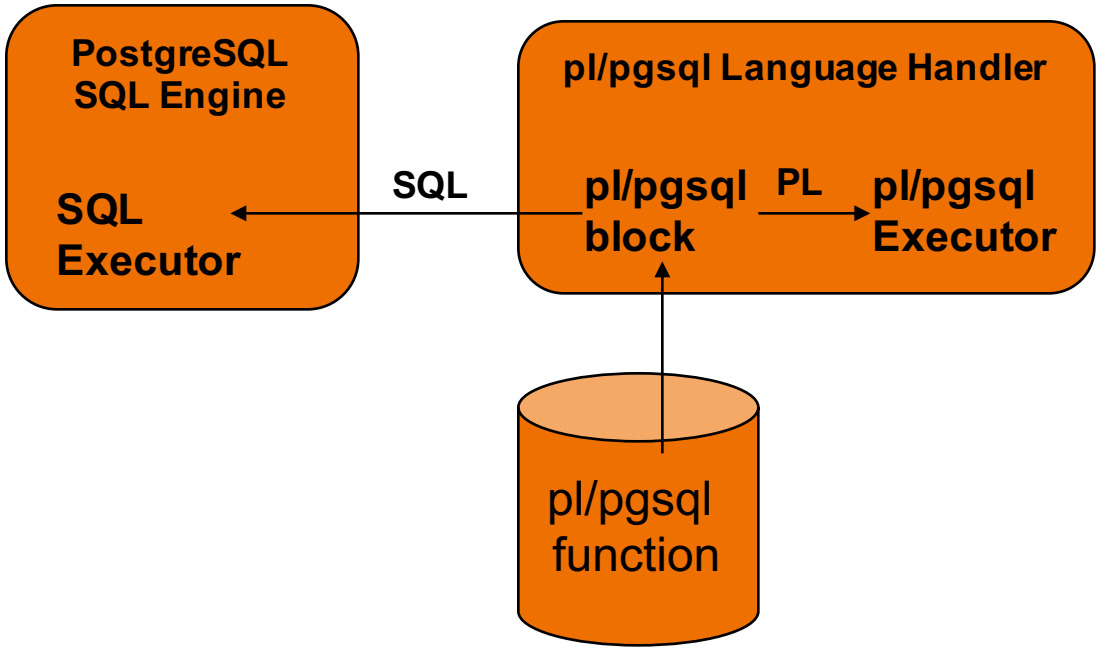
- PostgreSQL allows user-defined functions to be written in a variety of procedural languages.
- PostgreSQL currently supports several standard procedural languages :
  - PL/pgSQL
  - PL/Tcl
  - PL/Perl
  - PL/Python
  - PL/Java
  - PL/Ruby
  - Other languages can be defined by users

Introduction to PL/PGSQL

- PL/pgSQL is a loadable procedural language for the PostgreSQL database system.
- PL/pgSQL stand for Procedural language extension for postgres SQL
- PL/pgSQL has several distinct features:
  - Can be used to create functions and trigger procedures,
  - Adds control structures to the SQL language,
  - Can perform complex computations,
  - Inherits all user-defined types, functions, and operators,
  - Can be defined to be trusted by the server,
  - Is easy to use.

How it works

- PL/pgSQL defines a block structure for writing code
- PL/pgSQL functions can be compiled and stored inside database server offering better performance



PL/pgSQL Block Structure

- PL/pgSQL is a block-structured language.
- The complete text of a function definition must be a block.
  - DECLARE (optional)
    - Variables, cursors, user-defined exceptions
  - BEGIN (mandatory)
    - SQL statements
    - PL/pgSQL statements
  - EXCEPTION (optional)
    - Actions to perform when errors occur
  - END; (mandatory)
- Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.

10

PL/pgSQL Structure

- All key words and identifiers can be written in mixed upper and lower case
- Identifiers are implicitly converted to lowercase unless double-quoted.
- There are two types of comments in PL/pgSQL.
  - -- starts a comment that extends to the end of the line.
  - /* multi-line comments */

Declaring Variables

Variables

- Temporary storage of data
- Can be reused in the plpgsql block
- Declared and initialized in the declarative section
- Used and assigned new values in the executable section
- Passed as parameters to PL/pgSQL subprograms
- Used to hold the output of a PL/pgSQL subprogram
- Can be any valid SQL data type
- Can contain a DEFAULT or CONSTANT clause

12

Declaring Variables

- Examples:
  - `user_id integer;`
  - `quantity numeric(5);`
  - `url varchar;`
  - `myrow tablename%ROWTYPE;`
  - `myfield tablename.columnname%TYPE;`
  - `arow RECORD;`
  - `quantity integer DEFAULT 32;`
- The general syntax of a variable declaration is:
  - `name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];`

Assign value to Variables

- Assignment
  - `identifier := expression;`
  - `user_id := 20;`
  - `tax := subtotal * 0.06;`
- SELECT INTO
  - `SELECT INTO target select_expressions FROM ...;`
  - `SELECT INTO myrec * FROM emp WHERE empname = myname;`

14

Local Variable

FOUND

- Boolean local variable
- Starts out false within each PL/pgSQL function call.
- It is set by each of the following types of statements:
  - A SELECT INTO statement sets FOUND true if it returns a row, false if no row is returned
  - A PERFORM statement sets FOUND true if it produces (and discards) a row, false if no row is produced
  - UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected
  - A FETCH statement sets FOUND true if it returns a row, false if no row is returned
  - A FOR statement sets FOUND true if it iterates one or more times, else false.

15

Writing Executable Statements

- Executable Statements are written inside BEGIN .. END block
- Blocks can be nested
- Statements inside a block can continue over several lines
- All statements must end with ;

Declaring Function Parameters

- Parameters passed to functions are named with the identifiers $1, $2, etc
- Optionally alias names can be used
- Either the alias or the numeric identifier can then be used
- E.g
  - `CREATE FUNCTION sales_tax(subtotal real)`

Example 1

- Following function will sum two numerics which are passed at function call:

```
create or replace  function sum_val(i numeric, z numeric) returns
  numeric as $$
begin
return i+z;
end;
$$ language plpgsql;
```

- Invoke the function

```
Select sum_val(10,20);
```

Example 2

- Following function will show the difference between two passed values:

```
create or replace function differ(x numeric, y numeric) returns
  void as $$
declare
d numeric;
begin
D := x - y;
raise notice 'Difference is: %',d;
return;
end; $$ language plpgsql;
```

- Execute the function

```
select differ(10,5);
```

Anonymous code block - DO

DO [LANGUAGE ***lang_name***] ***code***

- `lang_name`: the default is plpgsql.

- execute an anonymous code block

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

Control Structures

- Control Structures change the logical flow of statements with a plpgsql block
- Three main types of control structures are available in plpgsql:
  - IF statement
  - Case expressions
  - Loop control structures

IF statement

- plpgsql IF statement structure is similar to other procedural languages
- It allows plpgsql to perform actions selectively based on conditions
- Syntax:

```
IF boolean-expression THEN
statements
[ ELSIF boolean-expression THEN
statements]
[ ELSE
statements ]
END IF;
```

Example

- Following function will return account details from customer table for a given customerid:

```
create or replace function dis_cus(c_id numeric) returns void as $$
declare
rec RECORD;
begin
select into rec * from customer where custid=c_id;
if found then
raise notice 'Customer Name: %', rec.custname;
raise notice 'Account No:    %', rec.accountid;
else
raise notice 'No Data';
end if;
return;
end;$$ language plpgsql;
```

- Execute the function

```
Select dis_cus(130);
```

Case expression

- A `CASE` expression returns a result based on one or more alternatives
- The value of the selector determines which result is returned
- Syntax:

```
CASE search-expression
WHEN expression [, expression [ ... ]] THEN
statements
[ WHEN expression [, expression [ ... ]] THEN
statements
... ]
[ ELSE
statements ]
END CASE;
```

Example

- Following function will display temparature string based on input temperature reading using searched case:

```
CREATE OR REPLACE FUNCTION dis_temp(tem numeric) RETURNS varchar AS $$ DECLARE
msg varchar;
BEGIN
CASE WHEN tem < 0  THEN
msg := 'ICY';
WHEN tem between 0 and 10  THEN
msg := 'COLD';
WHEN tem > 10  THEN
msg := 'NORMAL';
ELSE
msg := 'Cannot determine';
END CASE;
return msg;
END; $$ language plpgsql;
```

- Execute the function:

```
select dis_temp(-10);
```

LOOP Statements

- Loops are mainly used to execute statements repeatedly until an exit condition is reached
- There are three loop types:
  - Basic loop
  - WHILE loop
  - FOR loop

Simple Loop

- Simple loop that performs repetitive actions without overall conditions
- A Simple loop must have an EXIT
- Syntax:

```
LOOP
statements
END LOOP;
```

- Syntax for EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

Example

- Following function will display a series 0-10 using simple loop:

```
create or replace function dis() returns void as $$
declare
rec numeric;
begin
rec:=0;
loop
raise notice '%',rec;
rec:=rec+1;
EXIT when rec>10;
end loop;
return;
end; $$ language plpgsql;
```

- Execute funtion:

```
Select dis();
```

While Loop

- While loops that perform iterative actions based on a condition
- The condition is evaluated at the start of each iteration
- If the condition yields NULL loop will exit
- Syntax:

```
WHILE boolean-expression LOOP
statements
END LOOP;
```

Example

- Following function will display a series 0-10 using while loop:

```
create or replace function dis() returns void as $$
declare
rec numeric;
begin
rec:=0;
while rec<=10 loop
raise notice '%',rec;
rec:=rec+1;
end loop;
return;
end; $$ language plpgsql;
```

- Execute funtion:

```
Select dis();
```

For Loop

- FOR loops that perform iterative actions based on a count
- FOR loops have the same general structure as the basic loop
- A control statement is used before the LOOP keyword to set the number of iterations
- Syntax:

```
FOR name IN [ REVERSE ] expression .. expression [ BY expression]
  LOOP
statements
END LOOP;
```

Example

- Following function will display sequence of numbers 0-10 with increment of 2:

```
create or replace function dis1() returns void as
$$
declare
rec numeric;
begin
for rec in 0..10 by 2 loop
raise notice '%',rec;
end loop;
return;
end;$$ language plpgsql;
```

- Execute function:

```
Select dis1();
```

Example

- Following function will list name and account number for all the customers in customer table:

```
create or replace function dis_cus() returns void as
$$
declare
rec record;
begin
for rec in select*from customer loop
raise notice 'Customer Name: %', rec.custname;
raise notice 'Account No:    %', rec.accountid;
end loop;
return;
end;$$ language plpgsql;
```

- Execute function:

```
Select dis_cus();
```

Trapping Errors

- Syntax errors are handled at compile time
- plpgsql code may cause some unanticipated errors at run time
- To deal with such errors plpgsql provide EXCEPTION block
- Syntax:

```
[ DECLARE
declarations ]
BEGIN
statements
EXCEPTION
WHEN condition [ OR condition ... ] THEN
handler_statements
[ WHEN condition [ OR condition ... ] THEN
handler_statements
... ]
END;
```


34

Trapping Errors

- The EXCEPTION keyword starts the exception handling section
- Several exception handlers are allowed
- Only one handler is processed before leaving the block
- WHEN OTHERS is the last clause and can handle all types of exceptions
- EXCEPTION block must be at the last in a BEGIN..END block

Obtaining information about an error

- Exception handlers frequently need to identify the specific error that occurred.
- There are two ways to get information about the current exception:
  - Special variables
  - GET STACKED DIAGNOSTICS command.

    Syntax: GET STACKED DIAGNOSTICS variable = item [, ...];
- Each item is a key word: RETURNED_SQLSTATE, MESSAGE_TEXT, PGPG_EXCEPTION_HINT, _EXCEPTION_DETAIL, PG_EXCEPTION_CONTEXT

Example

- Following function will divide first parameter with second and return the result:

```
create or replace function dis(a numeric, b numeric) returns numeric
  as $$
declare
result numeric;
begin
result=a/b;
EXCEPTION
When others then
Raise notice 'Wrong value for second parameter. Must be a non-zero
  value';
return result;
```

  - `end;$$ language plpgsql;`
  - Execute function:

```
 Select dis();
```

PL/pgSQL Cursors

- Declaring Cursors
  - `DECLARE curs1 refcursor;`
  - `DECLARE curs2 CURSOR FOR SELECT * FROM tenk1;`
  - `DECLARE curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;`
- Opening Cursors
  - `OPEN FOR query`
  - `OPEN unbound_cursor FOR query;`
    - `OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;`
  - `OPEN FOR EXECUTE`
  - `OPEN unbound_cursor FOR EXECUTE query_string;`
    - `OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);`

PL/pgSQL Cursors (cont)

- Opening a Bound Cursor
  - OPEN bound_cursor [(argument_values)];
    Opening CursorsOPEN FOR query
  - DECLARE
    - curs2 CURSOR FOR SELECT * FROM tenk1;
    - curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
  - OPEN curs2;
  - OPEN curs3(42);

39

PL/pgSQL Cursors (cont)

- FETCH
  - FETCH cursor INTO target;
  - FETCH retrieves the next row from the cursor into a target, which may be a row variable, a record variable, or a comma-separated list of simple variables, just like SELECT INTO.
  - As with SELECT INTO, the special variable FOUND may be checked to see whether a row was obtained or not.
    - `FETCH curs1 INTO rowvar;`
    - `FETCH curs2 INTO foo, bar, baz;`
- CLOSE
  - CLOSE closes the portal underlying an open cursor.
  - This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.
    - `CLOSE curs1;`

Example

- Following function will list name and account number for all the customers in customer table:

```
create or replace function dis_cus() returns void as $$
declare
cur CURSOR for select*from customer;
rec RECORD;
begin
open cur;
fetch cur into rec;
while found loop
raise notice 'Customer Name: %', rec.custname;
raise notice 'Account No:    %', rec.accountid;
fetch cur into rec;
end loop;
close cur;
return;
end; $$ language plpgsql;
```

- Execute function:

```
Select dis_cus();
```

Triggers

- Created with the CREATE FUNCTION command
- Several special variables are created automatically in the top-level block.
  - NEW
    - Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is NULL in statement-level triggers.
  - OLD
    - Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers.
  - TG_NAME
    - Data type name; variable that contains the name of the trigger actually fired.

42

Triggers

- TG_WHEN
  - Data type text; a string of either BEFORE or AFTER depending on the trigger's definition.
- TG_LEVEL
  - Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.
- TG_OP
  - Data type text; a string of INSERT, UPDATE, or DELETE telling for which operation the trigger was fired.
- TG_RELNAME
  - Data type name; the name of the table that caused the trigger invocation.
- TG_NARGS
  - Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.

Triggers

- Note: A trigger function must return either NULL or a record/row value having exactly the structure of the table the trigger was fired for.
- The return value of a BEFORE or AFTER statement-level trigger or an AFTER row-level trigger is always ignored; it may as well be null. However, any of these types of triggers can still abort the entire operation by raising an error.

44

Example

- In this example city table store total population of a city and population will increase by 1 when a birth registration is done for that city:
- Create base tables:
  - `create table city(cityid numeric, population numeric);`
  - `create table birth(regid numeric, name varchar, cityid numeric);`
  - `insert into city values(1,0),(2,0);`
  - `select*from city;`
- Create Trigger function that will be called by trigger:

```
create or replace function trg_ins() returns trigger as $$
begin
update city set population=population+1 where cityid=NEW.cityid;
return null;
end; $$ language plpgsql;
```

- Create Trigger on birth table:

```
create trigger trg_ins_brth after insert on birth
for each row execute procedure trg_ins();
```

- Test whether trigger is working:

```
insert into birth values(101,'Raj',2);
select*from birth;
select*from city;
```

Lab Exercise - 1

- Create a plpgsql function
  - Declare two variables in declare section
  - Assign today's date to the first variable
  - Assign tomorrow's date to the second variable
  - Return first variable to the calling context
  - Display tomorrow's date on screen using notice messages

Lab Exercise - 2

- ```
  DECLARE
     citizen      VARCHAR(50) := 'SENIOR';
     credit_rating   VARCHAR(50) := 'EXCELLENT';
  BEGIN
      DECLARE
         citizen    NUMBER(7) := 201;
         nameVARCHAR(25) := 'JUNIOR';
      BEGIN
         credit_rating :='GOOD';
         …
      END;
    …
  END;
  ```
- In the plpgsql function above determine the value of following variable:
  - citizen in nested block:
  - credit_rating in main block:
  - citizen in main block:
  - name in main block:
  ```

Lab Exercise - 3

- Create a plpgsql function which can be called with a customerid and display hello and firstname of that customer
- Connect with edbstore database and create a table using following statement:
- create table saleproducts (like products);
- Create a plpgsql function to copy all products priced 9.99,10.99,11.99 from products table to saleproducts table

Lab Exercise - 4

- Create a plpgsql function which can be called with price as input and delete all the rows from saleproducts table for given price tag
- Create a plpgsql function using cursors which display category and highest priced prod_id in that category

PL/Python

- Add environment variables

```
Export PYTHONHOME=/opt/EnterpriseDB/LanguagePack/9.5/Python-3.3

export PATH=$PYTHONHOME/bin:$PATH
export LD_LIBRARY_PATH=$PYTHONHOME/lib:$LD_LIBRARY_PATH
```

- How to install

```
postgres@pglab1:~$ psql -c 'CREATE LANGUAGE plpython3u;'
CREATE LANGUAGE
```

- Example

```
CREATE FUNCTION gethostbyname(hostname text)
  RETURNS inet
AS $$
  import socket
  return socket.gethostbyname(hostname)
$$ LANGUAGE plpython3u SECURITY DEFINER;
```

- Result

```
postgres=# SELECT gethostbyname('www.postgresql.org');
 gethostbyname
---------------
 87.238.57.232
(1 row)
```

PL/Python Table functions

```
CREATE FUNCTION even_numbers_from_list(up_to int)
  RETURNS SETOF int
AS $$
    return range(0,up_to,2)
$$ LANGUAGE plpython3u;


CREATE OR REPLACE FUNCTION even_numbers_from_generator(up_to int)
  RETURNS TABLE (even int, odd int)
AS $$
    return ((i,i+1) for i in range(0,up_to,2))
$$ LANGUAGE plpython3u;


CREATE OR REPLACE FUNCTION even_numbers_with_yield(up_to int,
                                   OUT even int, OUT odd int)
  RETURNS SETOF RECORD
AS $$
    for i in range(0,up_to,2):
        yield i, i+1
$$ LANGUAGE plpython3u;
```

PL/Python Query

- plpy.execute, plpy.prepare

```
CREATE OR REPLACE FUNCTION get_table_oid(table_name text)
  RETURNS OID
AS $$
        stmt = plpy.prepare("select oid from pg_class where relname = $1", ["text"])
        res = plpy.execute(stmt, [table_name])
        return res[0]["oid"]
$$ LANGUAGE plpython3u;
```

```
postgres=# select get_table_oid('test');
 get_table_oid
---------------
         16408
(1 row)
```

PL/Python Logging

- 화면 출력을 위해서는 plpy.notice() 함수를 사용

```
CREATE OR REPLACE FUNCTION fact(x int) RETURNS int
AS $$
    global x
    f = 1
    while (x > 0):
        f = f * x
        x = x - 1
        plpy.notice('f:%d,  x:%d' % (f, x))
    return f
$$ LANGUAGE plpython3u;
```

```
postgres=# select * from fact(3);
NOTICE:  f:3, x:2
CONTEXT:  PL/Python function "fact"
NOTICE:  f:6, x:1
CONTEXT:  PL/Python function "fact"
NOTICE:  f:6, x:0
CONTEXT:  PL/Python function "fact"
 fact
------
    6
(1 row)
```

Summary

- In this module you learned:
  - PostgreSQL Procedural Languages
  - Introduction to PL/PGSQL
  - How it works
  - PL/pgSQL Block Structure
  - Declaring Variables
  - Writing Executable Statements
  - Declaring Function Parameters
  - Control Structures
  - Exception Handling
  - PL/pgSQL Cursors
  - Triggers
  - Examples & Lab

54

Day 4-2
Extension Developement

Writing Functions in C - Example

- add_func.c

```
#include "postgres.h"
#include "fmgr.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(add_ab);

Datum
add_ab(PG_FUNCTION_ARGS)
{
    int32   arg_a = PG_GETARG_INT32(0);
    int32   arg_b = PG_GETARG_INT32(1);

    PG_RETURN_INT32(arg_a + arg_b);
}
```

- Makefile

```
MODULES = add_func

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

- Compile

```
pglab@pglab1:~$ sudo su -
root@pglab1:~# cd ~postgres/temp
root@pglab1:/opt/PostgreSQL/9.4/temp# ls
add_func.c  Makefile
root@pglab1:/opt/PostgreSQL/9.4/temp# make
gcc -Wall ... -o add_func.o add_func.c
gcc -Wall ... -o add_func.so add_func.o
root@pglab1:/opt/PostgreSQL/9.4/temp# ls
add_func.c  add_func.o  add_func.so  Makefile
root@pglab1:/opt/PostgreSQL/9.4/temp# make install
/bin/mkdir -p '/opt/PostgreSQL/9.4/lib/postgresql'
/usr/bin/install -c -m 755  add_func.so
'/opt/PostgreSQL/9.4/lib/postgresql/'
```

Create function using C library

```
CREATE OR REPLACE FUNCTION add(int, int) RETURNS INT
 AS '/opt/PostgreSQL/9.4/lib/postgresql/add_func', 'add_ab'
LANGUAGE C STRICT;
```

```
postgres=# select add(1, 20);
 add
-----
  21
(1 row)

postgres=# \df+ add
                                                    List of functions
 Schema | Name | Result data type | Argument data types |  Type  | Security | Volatility |  Owner   | Language | Source code | Description
--------+------+------------------+---------------------+--------+----------+------------+----------+----------+-------------+-------------
 public | add  | integer          | integer, integer    | normal | invoker  | volatile   | postgres | c        | add_ab      |
(1 row)
```

add_funcs.sql.in

- CREATE FUNCTION 구문 자동 생성

```
CREATE OR REPLACE FUNCTION add(int, int) RETURNS INT
 AS 'MODULE_PATHNAME', 'add_ab'
LANGUAGE C STRICT;
```

- Makefile 에 아래와 같이 추가

```
MODULES = add_func
DATA_built = add_funcs.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

- Build

```
root@pglab1:/opt/PostgreSQL/9.4/temp# ls
add_func.c  add_funcs.sql.in  Makefile
root@pglab1:/opt/PostgreSQL/9.4/temp# make
sed 's,MODULE_PATHNAME,$libdir/add_funcs,g' add_funcs.sql.in >add_funcs.sql
...
root@pglab1:/opt/PostgreSQL/9.4/temp# cat add_funcs.sql
CREATE OR REPLACE FUNCTION add(int, int) RETURNS INT
 AS '$libdir/add_funcs', 'add_ab'
LANGUAGE C STRICT;
```

Custom Type

- PostgreSQL의 Type system은 쉽게 확장이 가능
- 정해진 몇개의 C function을 구현하면 누구나 구현 가능
- 타 DB와 차별화 되는 PostgreSQL 만의 특 장점
- Complex number 예제

```
postgres=#  \d test_complex
 Table "public.test_complex"
 Column  |  Type   | Modifiers
--------+---------+-----------
 a       | complex |
 b       | complex |
Indexes:
    "test_cplx_ind"  btree (a)

postgres=#  select  * from test_complex;
      a       |       b
--------------+---------------
 (1,2.5)      | (4.2,3.55)
 (33,51.4)    | (100.42,93.55)
 (56,-22.5)   | (-43.2,-0.07)
 (-91.9,33.6) | (8.6,3)
(4 rows)
```

```
postgres=#  select  a+b abs_sum  from  test_complex;
     abs_sum
----------------
 (5.2,6.05)
 (133.42,144.95)
 (12.8,-22.57)
 (-83.3,36.6)
(4 rows)
```

Sample – 주민번호 Type

- 주민번호를 저장하기 위한 jumin 타입
- 기능
  - 주민번호 뒷자리 암호화 및 마스킹
  - Equality 비교
  - 생년월일 순 정렬 및 비교
  - 주민번호 뒷자리 중 첫 글자를 기준으로 1800, 1900, 2000년대 자동 처리 기능
  - 생년월일 추출
  - 성별 추출

주민번호 Type

```
CREATE TABLE test_jumin (
    a    jumin,
    b    char(14)
);

INSERT INTO test_jumin VALUES ('951010-9723134','951010-9723134');
INSERT INTO test_jumin VALUES ('750107-1253452','750107-1253452');
INSERT INTO test_jumin VALUES ('750107-1253452','750107-1253452');
INSERT INTO test_jumin VALUES ('900208-2575294','900208-2575294');
```

```
postgres=# select * from test_jumin order by a;
       a        |       b
----------------+----------------
 951010-9****** | 951010-9723134
 951010-9****** | 951010-9723134
 951010-9****** | 951010-9723134
 971101-0****** | 971101-0723134
 731218-1****** | 731218-1151958
 750107-1****** | 750107-1253452
 750107-1****** | 750107-1253452
 760907-1****** | 760907-1756006
 761209-2****** | 761209-2436370
 830207-2****** | 830207-2187272
 870520-2****** | 870520-2508631
 880205-1****** | 880205-1163291
 900208-2****** | 900208-2575294
 951227-2****** | 951227-2438973
 960806-2****** | 960806-2405962
 960806-2****** | 960806-2205963
(16 rows)
```

```
postgres=# select * from test_jumin order by b;
       a        |       b
----------------+----------------
 731218-1****** | 731218-1151958
 750107-1****** | 750107-1253452
 750107-1****** | 750107-1253452
 760907-1****** | 760907-1756006
 761209-2****** | 761209-2436370
 830207-2****** | 830207-2187272
 870520-2****** | 870520-2508631
 880205-1****** | 880205-1163291
 900208-2****** | 900208-2575294
 951010-9****** | 951010-9723134
 951010-9****** | 951010-9723134
 951010-9****** | 951010-9723134
 951227-2****** | 951227-2438973
 960806-2****** | 960806-2205963
 960806-2****** | 960806-2405962
 971101-0****** | 971101-0723134
(16 rows)
```

주민번호 Type - Cont

```
postgres=# select * from test_jumin where b like '960806-2%';
       a        |       b
----------------+---------------
 960806-2****** | 960806-2405962
 960806-2****** | 960806-2205963
(2 rows)

postgres=# select * from test_jumin where a = '960806-2405962';
       a        |       b
----------------+---------------
 960806-2****** | 960806-2405962
(1 row)
```

```
postgres=# select a, jumin_birthday(a), jumin_gender(a)
             from test_jumin order by a;
       a        | jumin_birthday | jumin_gender
----------------+----------------+--------------
 951010-9****** | 18951010       | M
 951010-9****** | 18951010       | M
 951010-9****** | 18951010       | M
 971101-0****** | 18971101       | F
 731218-1****** | 19731218       | M
 750107-1****** | 19750107       | M
 750107-1****** | 19750107       | M
 760907-1****** | 19760907       | M
 761209-2****** | 19761209       | F
 830207-2****** | 19830207       | F
 870520-2****** | 19870520       | F
 880205-1****** | 19880205       | M
 900208-2****** | 19900208       | F
 951227-2****** | 19951227       | F
 960806-2****** | 19960806       | F
 960806-2****** | 19960806       | F
(16 rows)
```

What are Extension modules?

- Modules in the “Extension” directory are additional features to PostgreSQL.
- Generally, Extension modules aren’t included in the core database because they don’t appeal to a wide-enough audience, or because they are still under development.
- Extension modules are sometimes pulled into the core distribution.
- The modules in Extension are tied to that particular version of PostgreSQL, and the modules that are available change over time.

Installing Extension Modules

- In order to use a Extension module in a database you need to run CREATE EXTENSION command to install the module's features into that database.
- By default, Extension files are installed at PREFIX/share/extension, but most package management systems change this to share/PostgreSQL/extension or some variant.
- Documentation for Extension modules is installed at share/Extension or equivalent.

Installing Extension Modules

- You need to register the new objects in the database system by running create extension command
- Alternatively, run it in database template1 so that the module will be copied into subsequently-created databases by default.
- Syntax:

```
CREATE EXTENSION module_name;
```

-This command must be run by a database superuser

Day 4-3
Connectors

Objectives

- This module will cover:
  - Installing JDBC Connectors
  - Installing .NET Connectors
  - Extensions
  - Other resources

PostgreSQL JDBC driver

- JDBC API defines how clients may access the database
- Postgres JDBC Drivers provide connection to Postgres for Java applications
- Drivers available in for of Jar files
- Download from http://jdbc.postgresql.org/download.html
- Use JDBC4 for JDK1.6 or higher
- Other versions of JDBC drivers also available
- Source code also available

Connecting Java to Postgres

- Step 1: Import java.sql
- Step 2: Load the drivers Class.forName("org.postgresql.Driver");
- Step 3: Prepare Connection URL
  - jdbc:postgresql:database
  - jdbc:postgresql://host/database
  - jdbc:postgresql://host:port/database
- Step 4: get a Connection instance from JDBC
  - Connection con = DriverManager.getConnection(url, username, password);

EDB Postgres JDBC driver

- The EDB JDBC driver is a super set of community JDBC driver.
- Provides additional features for Oracle compatibilities.
  - It's highly recommended to use EDB driver with Oracle compatible mode.
- http://www.enterprisedb.com/docs/en/9.5/jdbc/toc.html
- Drivers available in for of Jar files
  - edb-jdbc15.jar supports JDBC version 3
  - edb-jdbc16.jar supports JDBC version 4
  - edb-jdbc17.jar supports JDBC version 4.1
- Driver class name is different
  - com.edb.Driver (not com.postgresql.Driver)
- Conn string scheme is different
  - jdbc:edb:database
  - jdbc:edb://host/database
  - jdbc:edb://host:port/database

Connection Parameters

- Example of Various Connection Properties:
  - String url = "jdbc:postgresql://localhost/edbstore";
  - Properties prt = new Properties();
  - prt.setProperty("user",“postgres");
  - prt.setProperty("password",“postgres1");
  - prt.setProperty("ssl","true");
  - Connection conn = DriverManager.getConnection(url, prt);

  - String url = "jdbc:postgresql://localhost/edbstore?user=postgres&password=postgres1&ssl=true";
  - Connection conn = DriverManager.getConnection(url);

JDBC CopyManager

- Normal insert
  - 800 seconds
- JDBC Batch update
  - 25 seconds
- Copy Manager
  - 7 seconds

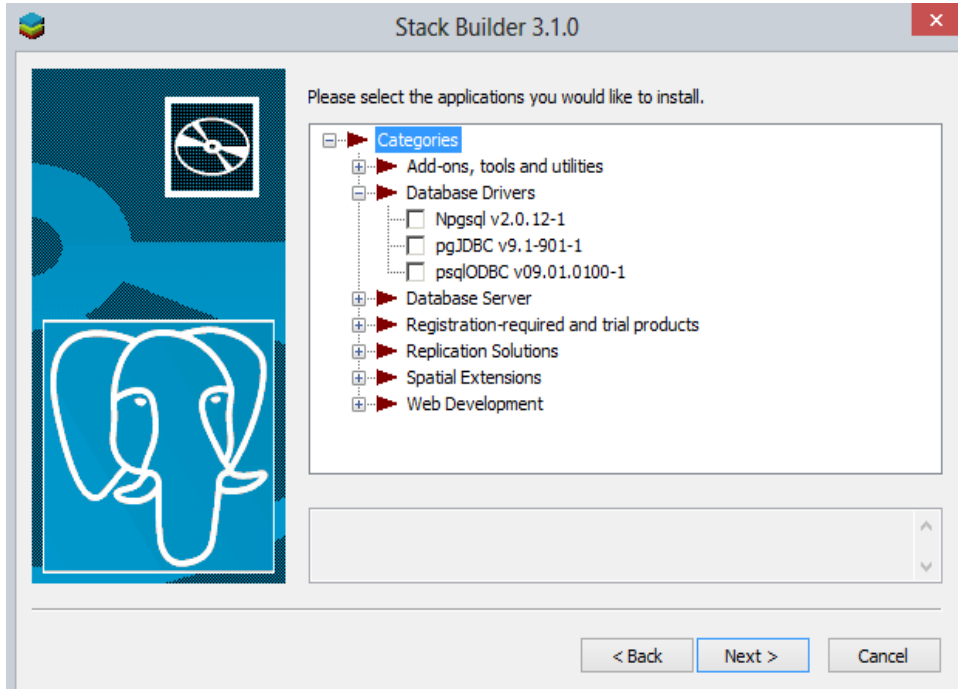
```
CopyManager cm = new CopyManager((BaseConnection) conn);
cpIN= cm.copyIn("COPY test(id, dat1, dat2) FROM STDIN WITH DELIMITER '|'");
StringBuffer buf = new StringBuffer();
byte[] data = buf.append(i).append("|ABCDE|ABCDEFGHIJKLMN\n").toString().getBytes();
cpIN.writeToCopy(data, 0, data.length);
cpIN.endCopy();
pyManager cm = new CopyManager((BaseConnection) conn);
```

.NET Connectors

- Microsoft .NET CLR can connect to Postgres using:
  - Npgsql .NET Data Provider
  - Via ODBC Data Source
- Allows .NET application to connect with Postgres database
- Easy installation using Stack Builder from EnterpriseDB
- Connector Sources are available
- http://www.enterprisedb.com/docs/en/9.5/dotnet/toc.html

.NET Connectors Installation

- Open Stack Builder from start menu and click next after select your Postgres installation
- Expand database drivers node and select the npgsql or psqlODBC



.NET Npgsql Driver Features

- You can send select, insert, delete queries
- You can call functions
- You can get resultset from functions
- You can use parameters in your queries. Input, Output and InputOuput parameters are supported
- Support for transactions

Connecting .NET to Postgres

- Npgsql data access is very similar to .NET connecting to SQL Server or OleDB
  - Add required namespace: Using Npgsql
  - Gather Connection String parameters information
  - Establish Connection:
    - NpgsqlConnection conn = new NpgsqlConnection("Server=127.0.0.1;Port=5432;User Id=postgres;Password=postgres1;Database=edbstore;");
  - Ope connect using open()
  - Use NpgsqlCommand class to create a query:
    - NpgsqlCommand command = new NpgsqlCommand("insert into sample values(1, 1)", conn)
  - Execute Query using ExecuteNonQuery()
  - Close connection using Close()

Day 4-4
Useful Data Types

bytea

- The bytea data type allows storage of binary strings. BLOB

```
edb=# select 'abcde', 'abcde'::bytea;
 ?column? |    bytea
----------+--------------
 abcde    | \x6162636465
```

- Easily convert between types.

```
edb=# select '안녕하세요'::bytea;
                bytea
------------------------------------
 \xec9588eb8595ed9598ec84b8ec9a94

edb=# select convert('안녕하세요', 'utf8', 'euc-kr');
        convert
------------------------
 \xbec8b3e7c7cfbcbcbfe4

edb=# select convert_from(E'\\xbec8b3e7c7cfbcbcbfe4'::bytea, 'euc-kr');
 convert_from
--------------
 안녕하세요
```

Lab

Network Address Types

Name	Storage Size	Description
cidr	7 or 19 bytes	IPv4 and IPv6 networks
inet	7 or 19 bytes	IPv4 and IPv6 hosts and networks
macaddr	6 bytes	MAC addresses

- Data types to store IPv4, IPv6, and MAC addresses
- cidr is more strict. Validate subnet.

```
edb=# select cidr '192.168.56';
      cidr
-----------------
 192.168.56.0/24

edb=# select inet '192.168.56.200/24';   # invalid subnet
       inet
-------------------
 192.168.56.200/24

edb=# select cidr '192.168.56.200/24';
ERROR:  invalid cidr value: "192.168.56.200/24"
LINE 1: select cidr '192.168.56.200/24';
                    ^
DETAIL:  Value has bits set to right of mask.
```

cidr and inet Containment test

- Containment test

Operator	Description	Example
<<	is contained by	`inet '192.168.1.5' << inet '192.168.1/24'`
<<=	is contained by or equals	`inet '192.168.1/24' <<= inet '192.168.1/24'`
>>	contains	`inet '192.168.1/24' >> inet '192.168.1.5'`
>>=	contains or equals	`inet '192.168.1/24' >>= inet '192.168.1/24'`
&&	contains or is contained by	`inet '192.168.1/24' && inet '192.168.1.80/28`

```
edb=# select cidr '192.168.56.192/26' >> inet '192.168.56.200';
 ?column?
----------
 t
(1 row)

edb=# select cidr '192.168.56.192/26' >> inet '192.168.56.100';
 ?column?
----------
 f
(1 row)
```

cidr and inet functions

Function	Return Type	Description	Example	Result
`abbrev(inet)`	`text`	abbreviated display format as text	`abbrev(inet '10.1.0.0/16')`	`10.1.0.0/16`
`abbrev(cidr)`	`text`	abbreviated display format as text	`abbrev(cidr '10.1.0.0/16')`	`10.1/16`
`broadcast(inet)`	`inet`	broadcast address for network	`broadcast('192.168.1.5/24')`	`192.168.1.255/24`
`family(inet)`	`int`	extract family of address; 4 for IPv4, 6 for IPv6	`family('::1')`	`6`
`host(inet)`	`text`	extract IP address as text	`host('192.168.1.5/24')`	`192.168.1.5`
`hostmask(inet)`	`inet`	construct hostmask for network	`hostmask('192.168.23.20/30')`	`0.0.0.3`
`masklen(inet)`	`int`	extract netmask length	`masklen('192.168.1.5/24')`	`24`
`netmask(inet)`	`inet`	construct netmask for network	`netmask('192.168.1.5/24')`	`255.255.255.0`
`network(inet)`	`cidr`	extract network part of address	`network('192.168.1.5/24')`	`192.168.1.0/24`
`set_masklen(inet, int)`	`inet`	set netmask length for inet value	`set_masklen('192.168.1.5/24', 16)`	`192.168.1.5/16`
`set_masklen(cidr, int)`	`cidr`	set netmask length for cidr value	`set_masklen('192.168.1.0/24'::cidr, 16)`	`192.168.0.0/16`
`text(inet)`	`text`	extract IP address and netmask length as text	`text(inet '192.168.1.5')`	`192.168.1.5/32`
`inet_same_family(inet, inet)`	`boolean`	are the addresses from the same family?	`inet_same_family('192.168.1.5/24', '::1')`	`FALSE`
`inet_merge(inet, inet)`	`cidr`	the smallest network which includes both of the given networks	`inet_merge('192.168.1.5/24', '192.168.2.5/24')`	`192.168.0.0/22`

macaddr

- The macaddr type stores MAC addresses. 6bytes
- Supported input formats

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

```
'08:00:2b:01:02:03'
```

```
edb=# select macaddr '08002b:010203', macaddr '08-00-2b-01-02-03';
      macaddr       |      macaddr
--------------------+-------------------
 08:00:2b:01:02:03  | 08:00:2b:01:02:03
(1 row)
edb=# select trunc(macaddr '08002b:010203');
       trunc
-------------------
 08:00:2b:00:00:00
```

Lab

UUID

- Stores Universally Unique Identifiers (UUID). 16bytes.
- Supported input formats

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

```
edb=# select uuid 'A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11';
                uuid
--------------------------------------
 a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11

edb=# create extension pgcrypto;
CREATE EXTENSION

edb=# select gen_random_uuid();
           gen_random_uuid
--------------------------------------
 58deed14-e3b8-4799-b394-6e182eb66626
```

Lab

Day 4-5
JSON & JSONB

Why NoSQL?

- Where did NoSQL come from?
  - Where all cool tech stuff comes from – Internet companies
- Why did they make NoSQL?
  - To support huge data volumes and evolving demands for ways to work with new data types
- What does NoSQL accomplish?
  - Enables you to work with new data types: email, mobile interactions, machine data, social connections
  - Enables you to work in new ways: incremental development and continuous release
- Why did they have to build something new?
  - There were limitations to most relational databases

Postgres' Response

- HSTORE
  - Key-value pair
  - Simple, fast and easy
  - Postgres v 8.2 – pre-dates many NoSQL-only solutions
  - Ideal for flat data structures that are sparsely populated
- JSON
  - Hierarchical document model
  - Introduced in Postgres 9.2, perfected in 9.3
- JSONB
  - Binary version of JSON
  - Faster, more operators and even more robust
  - Postgres 9.4

HStore: Key-value Store

- Supported since 2006, the HStore contrib module enables storing key/value pairs within a single column
- Allows you to create a schema-less, ACID compliant data store within Postgres
- Create single HStore column and include, for each row, only those keys which pertain to the record
- Add attributes to a table and query without advance planning
- Combines flexibility with ACID compliance

HSTORE Examples

- Create a table with HSTORE field

```
CREATE TABLE hstore_data (data HSTORE);
```

- Insert a record into hstore_data

```
INSERT INTO hstore_data (data) VALUES ('
            "cost"=>"500",
            "product"=>"iphone",
            "provider"=>"apple"');
```

- Select data from hstore_data

```
SELECT data FROM hstore_data ;
----------------------------------------
"cost"=>"500”,"product"=>"iphone”,"provider"=>"Apple"
(1 row)
SELECT data->'product' FROM hstore_data;
----------
 iphone
(1 row)
```

Postgres: Document Store

- JSON is the most popular data-interchange format on the web
- Derived from the ECMAScript Programming Language Standard.
- Supported by virtually every programming language
- New supporting technologies continue to expand JSON's utility
  - PL/V8 JavaScript extension
  - Node.js
- Postgres has a native JSON data type (v9.2) and a JSON parser and a variety of JSON functions (v9.3)
- Postgres will have a JSONB data type with binary storage and indexing (coming – v9.4)

JSON Examples

- Creating a table with a JSONB field

```
CREATE TABLE json_data (data JSONB);
```

- Simple JSON data element:

```
{"name": "Apple Phone", "type": "phone", "brand": "ACME", "price": 200,
"available": true, "warranty_years": 1}
```

- Inserting this data element into the table json_data

```
INSERT INTO json_data (data)  VALUES
    (' {   "name": "Apple Phone",
           "type": "phone",
           "brand": "ACME",
           "price": 200,
           "available": true,
           "warranty_years": 1
    } ')
```

JSON Examples

- JSON data element with nesting:

```
{“full name”: “John Joseph Carl Salinger”,
“names”:
    [
    {"type": "firstname", “value”: ”John”},
    {“type”: “middlename”, “value”: “Joseph”},
    {“type”: “middlename”, “value”: “Carl”},
    {“type”: “lastname”, “value”: “Salinger”}
    ]
}
```

A simple query for JSON data

```
SELECT DISTINCT
    data->>'name' as products
FROM json_data;

            products
------------------------------
 Cable TV Basic Service Package
 AC3 Case Black
 Phone Service Basic Plan
 AC3 Phone
 AC3 Case Green
 Phone Service Family Plan
 AC3 Case Red
 AC7 Phone
```

This query does not return JSON data – it returns text values associated with the key ‘name’

A query that returns JSON data

```
SELECT data FROM json_data;

data

--------------------------------
{"name": "Apple Phone", "type":
"phone", "brand": "ACME", "price":
200, "available": true,
"warranty_years": 1}
```

This query returns the JSON data in its original format

JSON and ANSI SQL - PB&J for the DBA

- JSON is naturally integrated with ANSI SQL in Postgres
- JSON and SQL queries use the same language, the same planner, and the same ACID compliant transaction framework
- JSON and HSTORE are elegant and easy to use extensions of the underlying object-relational model

JSON and ANSI SQL Example

```
SELECT DISTINCT
    product_type,
    data->>'brand' as Brand,
        data->>'available' as Availability
FROM json_data
JOIN products
ON (products.product_type=json_data.data->>'name')
WHERE json_data.data->>'available'=true;

 product_type   | brand     | availability
----------------------------+-----------+--------------
 AC3 Phone                  | ACME   | true
```

No need for programmatic logic to combine SQL and NoSQL in the application

Bridging between SQL and JSON

Simple ANSI SQL Table Definition

```
CREATE TABLE products (id integer, product_name text );
```

Select query returning standard data set

```
SELECT * FROM products;

 id | product_name
----+--------------
  1 | iPhone
  2 | Samsung
  3 | Nokia
```

Select query returning the same result as a JSON data set

```
SELECT ROW_TO_JSON(products)  FROM products;

 {"id":1,"product_name":"iPhone"}
 {"id":2,"product_name":"Samsung"}
 {"id":3,"product_name":"Nokia”}
```

JSON Data Type Example

```
{
  "firstName": "John",           -- String Type
  "lastName": "Smith",           -- String Type
  "isAlive": true,               -- Boolean Type
  "age": 25,                     -- Number Type
  "height_cm": 167.6,            -- Number Type
  "address": {                   -- Object Type
    "streetAddress": "21 2nd Street”,
    "city": "New York”,
    "state": "NY”,
    "postalCode": "10021-3100”
  },
  "phoneNumbers": [         // Object Array
    {                     // Object
      "type": "home”,
      "number": "212 555-1234”
    },
    {
      "type": "office”,
      "number": "646 555-4567”
    }
  ],
  "children": [],
  "spouse": null          // Null
}
```

JSONB

- Canonical representation
  - Whitespace and punctuation dissolved away
  - Only one value per object key is kept
  - Last insert wins
  - Key order determined by length, then bytewise comparison
- Equality, containment and key/element presence tests
- New JSONB creation functions
- Smaller, faster GIN indexes
- jsonb subdocument indexes
  - Use “get” operators to construct expression indexes on subdocument:
  - `CREATE INDEX author_index ON books USING GIN ((jsondata -> 'authors'));`
  - `SELECT * FROM books WHERE jsondata -> 'authors' ? 'Carl Bernstein'`

JSON, JSONB or HSTORE?

- JSON/JSONB is more versatile than HSTORE
- HSTORE provides more structure
- JSON or JSONB?
  - if you need any of the following, use JSON
    - Storage of validated json, without processing or indexing it
    - Preservation of white space in json text
    - Preservation of object key order Preservation of duplicate object keys
    - Maximum input/output speed
- For any other case, use JSONB

Lab

Day 4-6
Foreign Data Wrapper

Foreign Data Wrapper

- Foreign data wrappers (SQL MED) allow queries to read and write data to foreign data sources.
- Foreign data wrappers allow SQL access to data in:
  - Postgres clusters on the same server, perhaps in different databases or clusters
  - Remote Postgres servers
  - Stored in non-Postgres data repositories
  - Stored in data repositories with different performance and storage characteristics

Why use Foreign Data Wrapper?

- Using PostgreSQL as a central interface to connect to other systems / databases to gather data and issue queries or joins.
- Push-down for WHERE and Column to improve performance
- SELECT syntax; including useful clauses like DISTINCT, ORDER BY, GROUP BY and more.
- JOIN external data with internal tables
- FUNCTIONS for comparison, math, string, pattern matching, date/time, etc
- DML Operations INSERT / UPDATE / DELETE
- Make external data sources look like local tables

The detail of all the available foreign data wrapper available
https://wiki.postgresql.org/wiki/Foreign_data_wrappers

How to configure FDW?

- Creating an extension.

```
CREATE EXTENSION dummy_fdw;
```

- Create a foreign server.

```
CREATE SERVER dummy_server
FOREIGN DATA WRAPPER dummy_fdw
OPTIONS (host '127.0.0.1', port '3306');
```

- Creating a User-Mappings

```
CREATE USER MAPPING FOR postgres
SERVER dummy_server
OPTIONS (username 'foo', password 'bar');
```

- Creating a foreign table

```
CREATE FOREIGN TABLE f_table(id int, name text) SERVER dummy_server
OPTIONS (dbname 'db', table_name 'r_table');
```

How to Use FDW?

- The usage of Foreign Table is almost the same as the PostgreSQL table.

- Selecting Data From foreign table

```
SELECT id, name FROM f_table WHERE id = 1;
```

- Inserting data into foreign table.

```
INSERT INTO f_table values (1, 'foor',);
INSERT INTO f_table values (2, 'bar',);
```

- Deleting data from Foreign Table

```
DELETE FROM f_table where id= 3;
```

- Update Foreign Table

```
UPDATE f_table set name= 'bar' WHERE id = 1;
```

- Explain Table

```
EXPLAIN SELECT id, name FROM f_table WHERE name LIKE 'foo' limit 1;
```

FDW internal

```
extern Datum ifx_fdw_handler(PG_FUNCTION_ARGS);
extern Datum ifx_fdw_validator(PG_FUNCTION_ARGS);

CREATE FUNCTION ifx_fdw_handler() RETURNS fdw_handler
  AS 'MODULE_PATHNAME'
  LANGUAGE C STRICT;

CREATE FUNCTION ifx_fdw_validator(text[], oid) RETURNS void
  AS 'MODULE_PATHNAME'
  LANGUAGE C STRICT;

CREATE FOREIGN DATA WRAPPER informix_fdw
  HANDLER ifx_fdw_handler
  VALIDATOR ifx_fdw_validator;
```

FDW internal

```
Datum
ifx_fdw_handler(PG_FUNCTION_ARGS)
{
  FdwRoutine  *fdwRoutine                = makeNode(FdwRoutine);
/* Required  */
  fdwroutine->GetForeignRelSize          = blackholeGetForeignRelSize;        /* S   U D */
  fdwroutine->GetForeignPaths            = blackholeGetForeignPaths;          /* S   U D */
  fdwroutine->GetForeignPlan             = blackholeGetForeignPlan;           /* S   U D */
  fdwroutine->BeginForeignScan           = blackholeBeginForeignScan;         /* S   U D */
  fdwroutine->IterateForeignScan         = blackholeIterateForeignScan;       /* S       */
  fdwroutine->ReScanForeignScan          = blackholeReScanForeignScan;        /* S       */
  fdwroutine->EndForeignScan             = blackholeEndForeignScan;           /* S   U D */

/* Optional  - use NULL if not required  */
  ...
  ...

  PG_RETURN_POINTER(fdwRoutine);
}
```

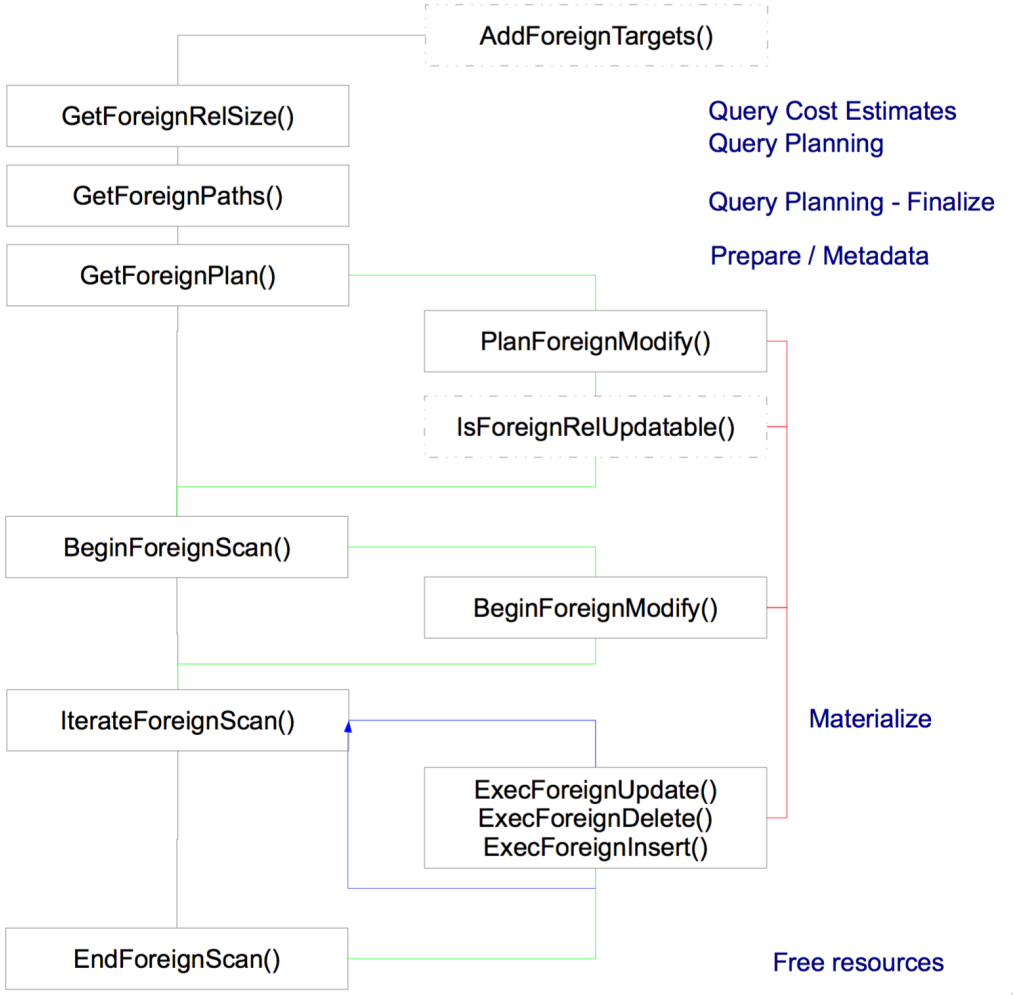
FDW internal

```
/* Optional - use NULL if not required */
#if (PG_VERSION_NUM >= 90300)
  fdwroutine->IsForeignRelUpdatable     = blackholeIsForeignRelUpdatable;
  fdwroutine->AddForeignUpdateTargets   = blackholeAddForeignUpdateTargets;  /*     U D */
  fdwroutine->PlanForeignModify         = blackholePlanForeignModify;        /*   I U D */
  fdwroutine->BeginForeignModify        = blackholeBeginForeignModify;       /*   I U D */
  fdwroutine->ExecForeignInsert         = blackholeExecForeignInsert;        /*   I     */
  fdwroutine->ExecForeignUpdate         = blackholeExecForeignUpdate;        /*     U   */
  fdwroutine->ExecForeignDelete         = blackholeExecForeignDelete;        /*       D */
  fdwroutine->EndForeignModify          = blackholeEndForeignModify;         /*   I U D */
#endif

  /* support for EXPLAIN */
  fdwroutine->ExplainForeignScan        = blackholeExplainForeignScan;       /* S   U D  EXPLAIN */
  fdwroutine->ExplainForeignModify      = blackholeExplainForeignModify;     /*   I U D  EXPLAIN */
  fdwroutine->AnalyzeForeignTable       = blackholeAnalyzeForeignTable;      /* ANALYZE */


#if (PG_VERSION_NUM >= 90500)
  fdwroutine->ImportForeignSchema       = blackholeImportForeignSchema;      /* IMPORT FOREIGN SCHEMA */
  fdwroutine->GetForeignJoinPaths       = blackholeGetForeignJoinPaths;      /* Scanning foreign joins */
  fdwroutine->GetForeignRowMarkType     = blackholeGetForeignRowMarkType;    /* Locking foreign rows */
  fdwroutine->RefetchForeignRow         = blackholeRefetchForeignRow;        /* Locking foreign rows */
#endif
```

FDW Flow



FDW Query Planing

- Setup and Planning a scan or modify action on a foreign datasource
- E.g. establish and cache remote connection
- Initialize required supporting structures for remote access
- Planner info and cost estimates via baserel and root parameters.

```
GetForeignRelSize()
GetForeignPaths()
GetForeignPlan()
```

FDW Query Scanning - `BeginForeignScan()`

```
void
BeginForeignScan  (ForeignScanState  *node,
                   int eflags);
```

- Execute startup callback for the FDW.
- Basically prepares the FDW for executing a scan.
- **`ForeignScanState`** saves function state values.
- Use **`node->fdw_state`** to assign your own FDW state structure.
- Must handle EXPLAIN and EXPLAIN ANALYZE by checking eflags & EXEC_FLAG_EXPLAIN_ONLY

FDW Query Scanning - `IterateForeignScan()`

```
TupleTableSlot *
IterateForeignScan (ForeignScanState *node);
```

- Fetches data from the remote source.
- Data conversion
- Materializes a physical or virtual tuple to be returned.
- Needs to return an empty tuple when done.
- Private FDW data located in `node->fdw` state

FDW Query Scanning - `ReScanForeignScan()`

```
void ReScanForeignScan (ForeignScanState *node);
```

- Prepares the FDW to handle a rescan
- Begins the scan from the beginning, e.g when used in scrollable cursors
- Must take care for changed query parameters!

End FDW Query Scanning - `EndForeignScan()`

```
void EndForeignScan (ForeignScanState *node);
```

- **`EndForeignScan()`** run when **`IterateForeignScan`** returns no more rows
- Finalizes the remote scan
- Close result sets, handles, connection, free memory, etc...

Memory Management

- PostgreSQL uses **`palloc()`**
- Memory is allocated in **`CurrentMemoryContext`**
- Use your own MemoryContext where necessary
  - e.g. **`IterateForeignScan()`**)
- Memory allocated in external libraries need special care

More Information

- Writing A Foreign Data Wrapper
  - http://www.slideshare.net/psoo1978/pg-fdw
- Official Documentation
  - http://www.postgresql.org/docs/current/static/index.html
  - Chapter 35. Extending SQL
  - Chapter 54. Writing A Foreign Data Wrapper
- Wiki page
  - https://wiki.postgresql.org/wiki/Foreign_data_wrappers

file_fdw

- http://www.postgresql.org/docs/9.5/static/file-fdw.html
- Can be used to access data files in the server's file system.
- Internally, uses COPY.

```
CREATE EXTENSION file_fdw;
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  ...
  application_name text
) SERVER pglog
OPTIONS ( filename '/home/josh/9.1/data/pg_log/pglog.csv', format 'csv' );
```

postgres_fdw

- http://www.postgresql.org/docs/current/static/postgres-fdw.html
- Similar with dblink module but can give better performance in many cases.

```
CREATE EXTENSION postgres_fdw;

CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');

CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');

CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

oracle_fdw

- https://github.com/laurenz/oracle_fdw

```
pgdb=# CREATE EXTENSION oracle_fdw;

pgdb=# CREATE SERVER oradb FOREIGN DATA WRAPPER oracle_fdw
          OPTIONS (dbserver '//dbserver.mydomain.com/ORADB');

pgdb=# GRANT USAGE ON FOREIGN SERVER oradb TO pguser;

pgdb=> CREATE USER MAPPING FOR pguser SERVER oradb
          OPTIONS (user 'orauser', password 'orapwd');

pgdb=> CREATE FOREIGN TABLE oratab (
          id        integer            OPTIONS (key 'true')  NOT NULL,
          text      character varying(30),
          floating  double precision   NOT NULL
       ) SERVER oradb OPTIONS (schema 'ORAUSER', table 'ORATAB');
```

Lab