

**Hands-on Labs on
PostgreSQL
internals.**

**Krishnamoorthy
Balaraman**



Hewlett Packard Enterprise

This page is left blank intentionally

PostgreSQL Internals - Hands-on Labs

This hands-on session consists of six exercises with increasing degree of complexity and interestingness. At the end of the six exercises, the student is expected to be comfortable with the following:

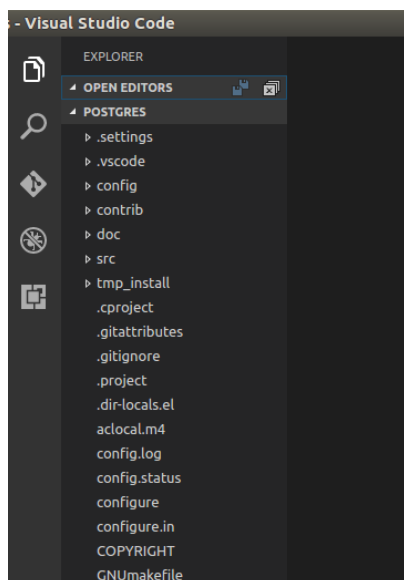
1. Building, installing and starting Postgres database server on a Linux system.
2. Running built-in tests that validate fundamental features of PostgreSQL.
3. Walking through the execution of simple SQL statements via the debugger.
4. Inspecting the output of the stages of the SQL compilation via the debugger.
5. Modifying an existing built in function of Postgres and rebuilding Postgres after the change.
6. Adding your own custom function as a built-in function of Postgres.

The labs are based on an Ubuntu based VM with PostgreSQL source code and related tools pre-installed. In the interest of readability, the remainder of this document will use the term Postgres instead of PostgreSQL. The correct term is PostgreSQL.

Lab 01 – Building, installing and starting PostgreSQL server on a Linux system

Steps:

1. Launch the virtual machine provided for your hands-on session. The default logon name is 'hpenonstop'. The password is hpenonstop.
2. The following folders constitute the content you will be using in this lab:
 - a. `$HOME/dbsrvr/postgres` – Postgres source code and build location.
 - b. `$HOME/dbsrvr` – Postgres install location
 - c. `$HOME/IISC-picasso2.1` – Picasso install location.
3. Launch a terminal. Run the command 'code'. This will launch the Microsoft Visual Studio Code IDE. The IDE should open the Postgres source code folder by default. If it does not, navigate to File -> Open Folder. Select `$HOME/dbsrvr/postgres` as the root folder of your source code. You should see the following screen once opened.



4. In the IDE navigate to `src/backend/tcop/postgres.c` and open the file. This will open the file in the built in editor. We will be revisiting this file in a later lab. As of now, it suffices to know that this file contains a routine named `exec_simple_query()` that orchestrates the execution of a simple SQL statement.
5. Press `shift + cntrl + p` to launch the command palette drop down box. Type "Build". This will bring up the "**Run Build Task**" Option. Select this option to start the build. In the future, you can also use the shortcut `cntrl + shift + B` to start a build. This step is equivalent to executing "make all" from the command line. The last line (as shown below) in the terminal window will tell you if the Postgres build completed successfully.

```
make[1]: Leaving directory '/home/hpenonstop/dbsrvr/postgres/config'
All of PostgreSQL successfully made. Ready to install.
```

If it is done successfully, it would have created a set of binaries, libraries and include files that constitute the Postgres database server product. The next step is to install Postgres using the objects you have just built.

From the terminal, navigate to "\$HOME/dbsrvr/postgres/src/backend" folder. Look for the binary file named "*postgres*". This is the main Postgres server executable. Notice the timestamp of the file. Does this reflect the build you have just completed?

6. Press `shift + cntrl + p` to launch the command palette drop down box. Type "run" and select "Tasks: Run Task" option from the list of options shown. This will bring up another box with three options
 - a. All
 - b. Clean
 - c. Install

Choose **Install**. This will begin "Installing" Postgres. Installation is essentially a copying of the objects built in the previous step to predefined locations under \$HOME/dbsrvr. This step is equivalent to executing "make install" from the command line. Once completed, you should see the following line at the end of the output window:

```
make[1]: Leaving directory '/home/hpenonstop/dbsrvr/postgres/config'
PostgreSQL installation complete.
```

From the terminal navigate to `$HOME/dbsrvr/bin` folder. Look for the *postgres* file in this folder. The timestamp of this file should match the timestamp of the file you checked in the previous step. The install process would have copied this file to the current location from `$HOME/dbsrvr/postgres/src/backend`. Also notice the executable files named *creatdb*, *initdb* and *psql*. You will be using these three in the next step.

7. In this step, you will launch the Postgres server from the command line and verify that the service is running by launching a Postgres command line client that allows users to execute SQL statements against the database. Before starting the Postgres server, you need to initialize the database cluster.

Though the database cluster has been created on the VM already, we will do it once more in order to learn the process of doing it. Before initializing, delete any existing database objects. Run the following commands (in the same order) to delete the existing cluster and initialize the database again.

<code>rm -rf \$PGDATA</code>	Delete all existing database objects. You can check the contents of \$PGDATA location before deleting. In your VM \$PGDATA points to \$HOME/dbsrvr/dbstore.
<code>export PATH=\$HOME/dbsrvr/bin:\$PATH</code> <code>export PGDATA=\$HOME/dbsrvr/dbstore</code>	Setup path and pgdata environment variable again. Though this is set in your profile, initdb seems to require this to be setup after the previous step.
<code>initdb</code>	Initialize the Postgres database cluster. For more information on what initdb does, refer to the Postgres documentation.

You should see the following output in your screen after the database cluster initialization completes successfully.

Success. You can now start the database server using:

```
pg_ctl -D /home/hpenonstop/dbsrvr/dbstore -l logfile start
```

`pg_ctl` is a wrapper script that can be used to start the Postgres server. In the above command, notice the `-l logfile` option. This indicates the name of the file the Postgres server will use for logging. We will be looking at the contents of this log while debugging. Ensure that the file (if it exists) is deleted before executing the above command to start the Postgres server. You can also use the script `start_server.ksh` located at `$HOME/dbsrvr` to start the server. This script uses the same command as above to start the server.

You now need to create an empty database (to hold your database objects) before launching the `psql` SQL client. Use the `createdb` executable mentioned above to create a database named `sampledb` as follows. Execute the following command from the terminal.

```
createdb sampledb
```

You will not see any output on the screen when the database is created successfully.

Launch the SQL client for Postgres. This is an executable named `psql`. You need to specify the name of the database to connect to while launching the `psql` client.

```
psql -d sampledb
```

You should now be inside the `psql` SQL client for Postgres. You will be able to execute any SQL statement from this client. Create a table named `T1` with one column named `I` of the `INT` data type. Insert some records into this table and select the records inserted.

Basic `psql` commands:

- i. List all databases - `\list` or `\l`
- ii. List all tables in current database - `\dt`
- iii. Quit - `\q`

8. To stop the database server, use the `stop_server.ksh` script located at `$HOME/dbsrvr` or execute the following command:

```
pg_ctl -D /home/hpenonstop/dbsrvr/dbstore -l logfile stop
```

Outcome:

In this lab you should have successfully done the following:

1. Built Postgres from its sources.
2. Installed Postgres to a predefined location on your system.
3. Initialized the database cluster.
4. Created an empty database named `sampledb`.
5. Launched the `psql` client and created a simple table within the `sampledb` database.

Questions:

1. Try and connect to the same database from the `pgadmin3` GUI client.
2. What do think is the purpose of the "clean" option under "Run Tasks". Try if after this session (do not try it now). What happens if you run a build after running "clean"? Do you see any difference? Save the output of the terminal window for the build command and inspect the contents of the file.
3. Try and prepare a complete list of all the objects that are created when you build Postgres (hint: All objects built will be in subfolders within `$HOME/postgres`. A folder diff may yield useful information). Ignore the object files (files with `.o` extension).

Additional Information:

1. What is the meaning of "creating a database cluster"?
<https://www.postgresql.org/docs/9.5/static/creating-cluster.html>
2. Additional information regarding `pg_ctl` script:
<https://www.postgresql.org/docs/9.5/static/app-pg-ctl.html>

Your Notes/Observations:

Lab 02 – Running built-in tests that validate fundamental features of PostgreSQL.

All good open source software come packaged with a set of tests that validate the basic functionality of the product. Postgres comes with a suite of tests too. These tests are invoked using the make infrastructure. Follow the steps below to run the test suite that comes with Postgres:

1. Ensure the database server is stopped. The server will be started automatically when the tests are run. You can use the `stop_server.ksh` script located at `$HOME/dbsrvr` or the following command:

```
pg_ctl -D /home/hpenonstop/dbsrvr/dbstore -l logfile stop
```

2. Open a terminal window and navigate to `$HOME/dbsrvr/postgres` folder. Run "`make check`" from the command line. This will start executing the test suite. This will take a few minutes to complete. You should see the following output in your terminal window once the tests complete:

```
plancache      ... ok
limit          ... ok
plpgsql        ... ok
copy2          ... ok
temp           ... ok
domain         ... ok
rangefuncs     ... ok
prepare        ... ok
without_oid    ... ok
conversion     ... ok
truncate       ... ok
alter_table    ... ok
sequence       ... ok
polymorphism   ... ok
rowtypes       ... ok
returning      ... ok
largeobject    ... ok
with           ... ok
xml            ... ok
test event_trigger ... ok
test stats     ... ok
===== shutting down postmaster =====
===== removing temporary instance =====

All 157 tests passed.
```

3. The purpose of doing this lab is to show you a way to quickly validate any changes you may make to the Postgres source code. It is necessary to run the built-in test suite to validate that the changes you have made do not cause any of the tests in the built in test suite to fail. These built-in tests are referred to as regression tests in the Postgres documentation. We will be revisiting this lab again in lab-05.

Outcome:

In this lab, you have used a single line command to invoke a set of 157 tests that validate the basic functionality of Postgres.

Questions:

1. How do you add the "make check" option to the Visual Studio Code GUI? (Hint: Check *Configure Task Runner* option in the command palette).
2. How do you add a new test to the existing test suite? (Refer to Postgres documentation)
3. Did the "make check" command run all of the regression tests that come packaged with Postgres? Read the following page from the documentation:

<https://www.postgresql.org/docs/9.5/static/regress-run.html>

How do you run a test with many concurrent connections? What is the purpose of 'make check-world'?

Your Notes/Observations:

Lab 03 – Walking through the execution of simple SQL statement via the debugger.

In this lab, we will get a peek into the internals of Postgres. The lab assumes knowledge of the basics of using the GNU debugger – gdb. We will debugging using the Visual Studio Code IDE. The IDE will use gdb in the background.

1. Check if Postgres is currently running.

```
ps | grep "postgres"
```

If the command does not return anything, start the Postgres server.

2. The above command will list one process id for the Postgres server. There, will, however be multiple Postgres processes running in the background. Run the following command to see all Postgres process instances running on the system. Note down the process ids listed by the ps command above.

```
pstree -p <pid returned by ps command above>
```

The output of the command above should be something similar to the following (the process ids will be different):

```
hpenonstop@ubuntu:~/dbsrvr$ ps | grep "postgres"
6992 pts/4    00:00:00 postgres
hpenonstop@ubuntu:~/dbsrvr$ pstree -p 6992
postgres(6992)
├── postgres(6994)
├── postgres(6995)
├── postgres(6996)
├── postgres(6997)
└── postgres(6998)
hpenonstop@ubuntu:~/dbsrvr$
```

You can also run the following command to view all running Postgres instances with human readable details about each process.

```
ps aux | grep "postgres"
```

```
hpenonstop@ubuntu:~/dbsrvr$ ps aux | grep "postgres"
hpenons+ 6639  0.0  2.3 931220 48544 ?        SL   07:56   0:00 /usr/share/code/code /usr/share/code/resource
hpenonstop/dbsrvr/postgres utf8 /usr/share/code/code
hpenons+ 6992  0.0  0.8 179676 17168 pts/4    S    08:26   0:00 /home/hpenonstop/dbsrvr/bin/postgres -D /home
hpenons+ 6994  0.0  0.1 179676 2752 ?        Ss   08:26   0:00 postgres: checkpoint process
hpenons+ 6995  0.0  0.2 179676 4256 ?        Ss   08:26   0:00 postgres: writer process
hpenons+ 6996  0.0  0.1 179676 2748 ?        Ss   08:26   0:00 postgres: wal writer process
hpenons+ 6997  0.0  0.2 179984 5936 ?        Ss   08:26   0:00 postgres: autovacuum launcher process
hpenons+ 6998  0.0  0.1 34684 2152 ?        Ss   08:26   0:00 postgres: stats collector process
hpenons+ 7054  0.0  0.0 21292 980 pts/4    S+   08:27   0:00 grep --color=auto postgres
hpenonstop@ubuntu:~/dbsrvr$
```

As you can see above, there are a number of postgres process instances running on the system. The names of the processes (checkpoint processes, writer process etc) give an indication as to main functionality of the process. For more details on the process architecture of Postgres, refer to the following page in the documentation:

<https://www.postgresql.org/docs/9.5/static/tutorial-arch.html>

3. Launch the `psql` client with a connection to the `sampledb` database created in lab-01. We now need to find out the process id of the postgres process that has been 'forked' or created for the psql client instance. Run the `pstree` command again. You should now see an additional process id. This should correspond to the postgres process id servicing the psql client. An easier way to find this out is to execute the `select pg_backend_pid();` command from the psql prompt as shown below:

```

hpenonstop@ubuntu:~/dbsrvr$ psql -d sampledb
psql (9.5.3)
Type "help" for help.

sampledb=# select pg_backend_pid();
 pg_backend_pid
-----
          7178
(1 row)

sampledb=#

```

Note down this process id. You will need it in the next step.

4. Switch to the Visual Studio Code IDE. You will now "attach" to the above process and start debugging. Before starting the debugging session, set the breakpoints at the following lines in the method `exec_simple_query()` located in the file `src/backend/tcop/postgres.c`

```

CommandDest dest = whereToSendOutput;
...
parsetree_list = pg_parse_query(query_string);
...
querytree_list= pg_analyze_and_rewrite(parsetree, query_string,NULL,
0);

plantree_list = pg_plan_queries(querytree_list, 0, NULL);

```

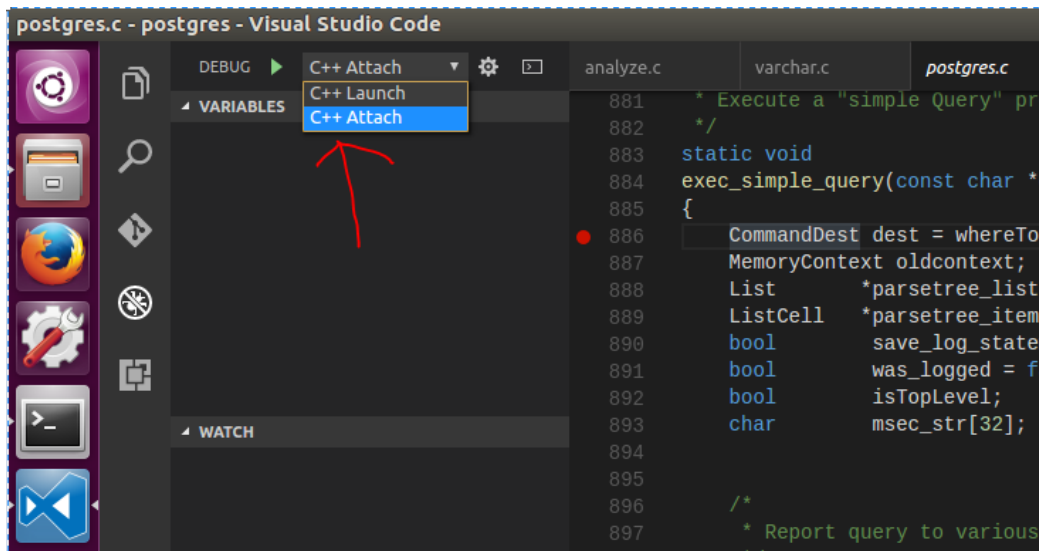
To set a break point click on the area of the editor window to the right left of the line number. A breakpoint is set by the click of a button and is indicated by the red circle as shown below. Clicking on the circle will remove the breakpoint.

```

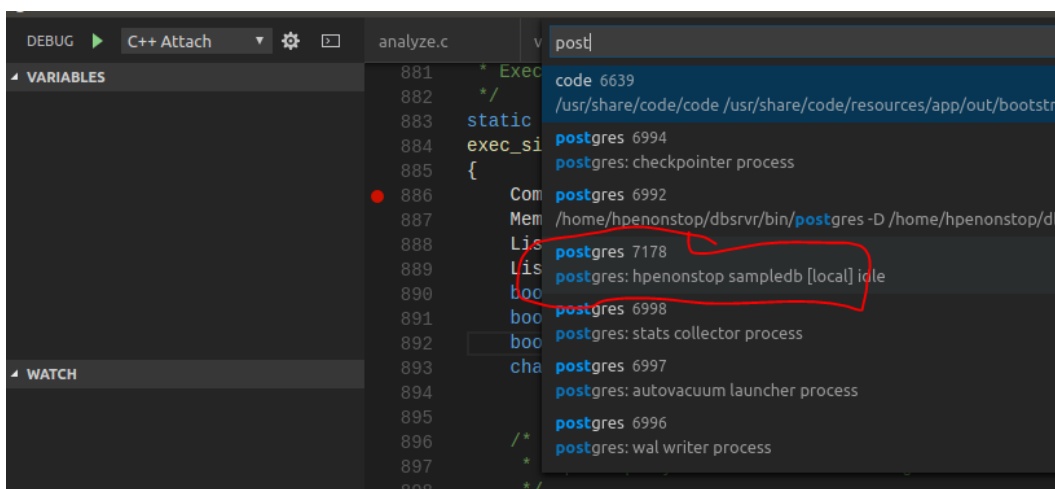
883 static void
884 exec_simple_query(const char *query_string)
885 {
886     CommandDest dest = whereToSendOutput;
887     MemoryContext oldcontext;
888     List *parsetree_list;
889     ListCell *parsetree_item;
890     bool save_log_statement_stats = log_statement_stats;
891     bool was_logged = false;
892     bool isTopLevel;
893     char msec_str[32];
894
895

```

5. Ensure that the "C++ Attach" configuration is chosen as the default for the debugging. The C++ attach configuration is used when you want to debug a running process. In our case, our aim is to debug the postgres process already launched to service the psql client. All the work in steps 1 -3 was to understand the process architecture and to identify the pid of the process we are going to attach the debugger to. Navigate to menu *view -> Debug* and select "**C++ Attach**" from the drop down box as shown below:



6. You are now all set to start the debugging process. Press **F5** key or open the command palette and select "Start Debugging" to start the debug process. You will now be asked to select the process you want to debug. The dialog is not intelligent enough to list only the Postgres processes. It will list all running processes on the system. Type postgres in the text box. This will list all the postgres processes running on the system. From this box, select the postgres process whose pid matches the pid returned by the `select pg_backend_pid();` command that you executed in the psql client. In the screen grab below, I am selecting 7178 since that is the pid of the postgres process servicing my psql client instance.



You will now see a terminal window pop up asking whether you are ok with executing something with sudo privileges. Say yes. You will then be prompted for the password for the hpenonstop user. Enter it and continue. Your postgres process should now be in debug mode.

7. Go back to the psql prompt and execute the following SQL statement (assuming table t1 exists since it was created in lab-01).

```
SELECT * FROM T1;
```

The query will not complete immediately since the Postgres process is now in debug mode. Switch back to the IDE and you will see the execution paused at your first breakpoint in `exec_simple_query` method as shown below:

```

882
883 static void
884 exec_simple_query(const char *query_string)
885 {
886     CommandDest dest = whereToSendOutput;
887     MemoryContext oldcontext;
888     List      *parsetree_list;
889     ListCell   *parsetree_item;
890     bool       save_log_statement_stats = log_statement_stats;
891     bool       was_logged = false;
892     bool       isTopLevel;
893     char       msec_str[32];
894

```

From this point onward, you can use the debugger commands to step through the execution one line at a time.

Step Over one Line – **F10**

Step into a function/method – **F11**

Continue until next breakpoint – **F5**

Ensure that the four breakpoints you set are hit. Press F5 when you reach the last breakpoint. This will complete the execution of the query. You can also execute any gdb command in the debug console shown below the editor window:

```

894
895
DEBUG CONSOLE

Breakpoint 2, exec_simple_query (query_string=0x2ec1608 "insert into t1 values(1);") at postgres.c:938
    parsetree_list = pg_parse_query(query_string);

Breakpoint 1, pg_analyze_and_rewrite (parsetree=0x2ec22c8, query_string=0x2ec1608 "insert into t1 values(1);") at postgres.c:666
    if (log_parser_stats)

Program received signal SIGINT, Interrupt.
0x00007f86ddc64e70 in __poll_nocancel () at ../sysdeps/unix/syscall-template.S:84

Breakpoint 3, exec_simple_query (query_string=0x2ec1608 "select * from t1;") at postgres.c:886
    CommandDest dest = whereToSendOutput;

```

Prefix any gdb command with `-exec`. This prefix is needed only by the Visual Studio Code IDE.

Common gdb commands:

`bt` – backtrace showing method invocations that precedes the current frame.

`p <variable name>` - print the value of a variable.

`p *<variable name>` - print the contents of a pointer variable.

Execute the `bt` command to see the backtrace

`-exec bt`

Also notice the useful debug related information shown to the left of the editor window. Hovering over a variable in the editor window will show the value of the variable.

Once you are comfortable with the debug process, detach the postgres process from the debugger by pressing the red square shaped icon shown on top of the editor window when in debug mode. Alternatively press **Shift + F5**.

Exit from the psql client (**\q**) and stop the Postgres server.

Outcome:

In this lab, you have placed the Postgres server process into debug and stepped through the main steps in the execution of a SQL statement. It is okay if you do not understand what each of these steps do at this point. We will build that knowledge gradually.

Questions:

1. Change the select statement to **SELECT I FROM T1** (replace * with column name) and run it through the debugger. What is the difference you see in the value of the **parsetree_list** variable (pprint output).
2. Build a flowchart of the action happening inside the method **exec_simple_query**. You can ignore the internal working of the transaction related method invocations such as **start_xact_command()** etc. You can generalize them to 'transaction related commands'.
3. What happens inside the methods **pg_parse_query**, **pg_analyze_and_rewrite** and **pg_plan_queries**? Step into these functions (F11) via the debugger and walk through their execution.
4. What do you think "portal run" means towards the end of the method?
5. What is the gdb command to print an arbitrary number of bytes starting at a given pointer location? (Refer to the gdb documentation online). Use the command to print the first four bytes of the **parsetree_list** variable.

Your Notes/Observations:

This page is left blank intentionally

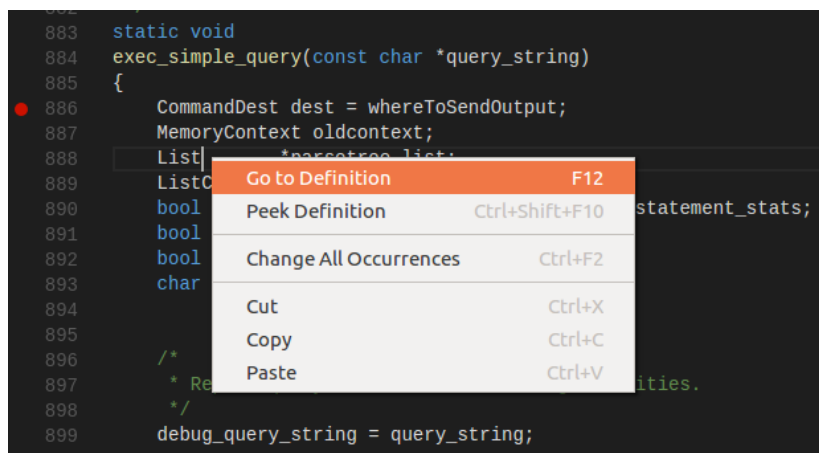
Lab 04 – Inspecting the output of the stages of the SQL compilation via the debugger.

This lab will be a continuation of lab-03. Follow the same steps as lab-03 up to and including step 7. Before starting the Postgres server, ensure that the contents of the server log file are deleted. The log file is named 'logfile' and will be present in the directory from which you start the Postgres server. Bring the Postgres server into debug mode. Now that you are comfortable with the debugging process, we will pause at each breakpoint and inspect a few variables to study the compilation process.

1. Run the debugger until you hit the breakpoint at the following line in the method `exec_simple_query`:

```
parsetree_list = pg_parse_query(query_string);
```

Let the line complete execution (press F10). The execution should now be paused at the next line. At this point, Postgres has completed basic (syntax) parsing of your SQL statement. This is the purpose of the `pg_parse_query` method. The output of the parsing stage is a parse tree. This is stored in the variable `parsetree_list`. Almost all tree representations in Postgres are of the List data type. Look at the definition of the `parsetree_list` variable at the beginning of the method. You will notice that the type of the variable is a pointer to a list. Right click on "list" to see the definition of the `list` data structure. Spend some time studying the definition. Once done, come back to the postgres.c file location you were at before (menu *Go -> Back*)



2. Print the contents of `parsetree_list` using the print command in the gdb window. You will see an output similar to the following:

```
-exec p parsetree_list
$1 = (List *) 0x600000011
```

```
-exec p *parsetree_list
$2 = {type = T_List, length = 1, head = 0x2ec23a0, tail = 0x2ec23a0}
```

3. As you can see above, there is not enough information that will help us understand the 'parse tree' that is being represented by the variable `parsetree_list`. To get that information, you need to manually walk through the List data structure in the debugger. For more information on the List data structure, refer to the documentation at:

https://wiki.postgresql.org/wiki/Developer_FAQ#Why_do_we_use_Node_and_List_to_make_data_structures.3F

Use the `pprint` function in the gdb command window to get a formatted listing of the full parse tree:

```
-exec call pprint(parsetree_list)
```

When you execute the above command, you will not see any output in the gdb window. This is because, the output of the pprint command has been redirected to the server log file (the `-l` option when you start the server). Open the log file and look at the end of the file. You should see something similar to the following:

```
(
  {SELECT
    :distinctClause <>
    :intoClause <>
    :targetList (
      {RESTARTGET
        :name <>
        :indirection <>
        :val
          {COLUMNREF
            :fields (
              {A_STAR
            }
          )
        :location 7
      }
      :location 7
    )
  }
  :fromClause (
    {RANGEVAR
      :schemaname <>
      :relname t1
      :inhOpt 2
      :relpersistence p
      :alias <>
      :location 14
    }
  )
  :whereClause <>
  :groupClause <>
  :havingClause <>
  :windowClause <>
  :valuesLists <>
  :sortClause <>
  :limitOffset <>
  :limitCount <>
  :lockingClause <>
  :withClause <>
  :op 0
  :all false
  :larg <>
  :rang <>
}
```


)

Try and understand this output before proceeding further. What do think the terms `RESTARTGET`, `COLUMNREF` and `RANGEVAR` in the above output mean?

4. Use `pprint` function to dump the output of the following variables as well (after executing them of course!)

```
querytree_list=pg_analyze_and_rewrite(parsetree,query_string,NULL, 0);  
plantree_list = pg_plan_queries(querytree_list, 0, NULL);
```

5. Compare the contents of all three variables (as dumped in the server log file) – `parsetree_list`, `querytree_list` and `plantree_list`. What are the similarities and what are the differences? Can you map the contents of these variable with the stages of SQL compilation?
6. Repeat the execution of another SQL statement via the debugger. This time, pause and inspect the values of variables at the end of each line in the method `exec_simple_query()`. Once you are comfortable with this, you have learnt the basic steps of how Postgres executes a simple select statement. You can use this knowledge to go however deep you want into Postgres internals!

Outcome:

In this lab you have finally delved into the internals of the compilation and execution of a simple `SELECT` statement in Postgres. You are now equipped with the basic skills needed to debug and understand how an RDBMS works from the inside!

Questions:

1. What is the value of the `query_string` variable inside the `exec_simple_query` method?
2. What do the calls to the method `MemoryContextSwitchTo()` mean? (refer to <https://blog.pgaddict.com/posts/introduction-to-memory-contexts>)
3. How do you map the breakpoints you created to the stages of the SQL compilation process explained in the theory section on SQL compilation?

Your Notes/Observations:

This page is left blank intentionally

Lab 05 – Modifying an existing built in function of Postgres and rebuilding Postgres after the change.

In this lab you will build on the confidence and knowledge you have gained in debugging Postgres code in labs-03 and 04. Our overall goal is to be able to add a new built-in function to Postgres. This is what you will be doing in lab-06. This lab is get you familiar with the idea of modifying some code in the Postgres code base, building Postgres again and verifying that your changes work. Postgres has a built in function named `char_length` that returns the length of any character based column (string length function). We will be modifying the implementation of this function to get an understanding of how this function works.

1. Start the Postgres server and the psql client with a connection to the sampledb database.
2. Test the `char_length` function :

```
hpenonstop@ubuntu:~/dbsrvr$ psql -d sampledb
psql (9.5.3)
Type "help" for help.
```

```
sampledb=# select char_length('abc');
 char_length
-----
          3
(1 row)
```

3. We will now modify the function to return twice the length of any character string. For example, the function should return 6 when the input is the string 'abc'. Though this is not the correct value to be returned, this is just a hypothetical exercise to understand how functions work. We will be implementing a useful function in the next lab.
4. Navigate to the file `src/backend/utils/adt/varlena.c`. Open this file in the IDE. Search for implementation of the method `textlen` within the file. You should see the following:

```
/*
 * textlen -
 *   returns the logical length of a text*
 *   (which is less than the VARSIZE of the text*)
 */
Datum
textlen(PG_FUNCTION_ARGS)
{
    Datum        str = PG_GETARG_DATUM(0);

    /* try to avoid decompressing argument */
    PG_RETURN_INT32(text_length(str));
}
```

5. Notice the way arguments are passed in (PG_FUNCTION_ARGS) and the way the function returns a value (PG_RETURN_INT32). Also notice the way arguments are copied to local variable within the method. These are macros and we will be revisiting them in lab-06. The `textlen` method is gets invoked when you use the `char_length` function in any SQL statement. Go ahead and test this. Place a breakpoint within the function and bring the postgres server process into debug. Execute the same statement as before that uses the `char_length` function. Your breakpoint within this function should be hit. Pause here and look at the back trace. Find out which line in the `exec_simple_query()` method is causing this method to be invoked eventually.
6. Detach the debugger from the process. Exit from the psql client and stop the postgres server. We will now make our first (albeit trivial) change to Postgres code!

In the file `src/backend/utils/adt/varlena.c`, change the following line

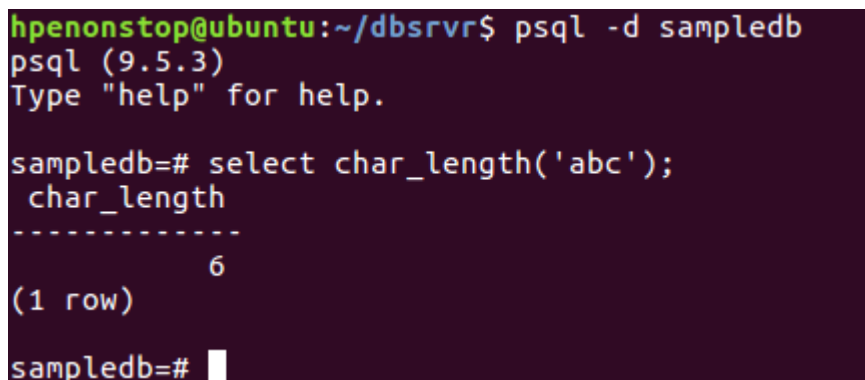
```
PG_RETURN_INT32(text_length(str));
```

to

```
PG_RETURN_INT32(text_length(str) * 2);
```

Notice that `text_length` is the method that computes the actual length of the string. We are multiplying that value by 2 just for fun!

7. Rebuild Postgres. Refer to lab-01 – Step 5. Ensure that there are no errors in compilation.
8. Install Postgres. Refer to lab-01 – Step 6.
9. Run basic sanity tests to ensure you have not broken anything - lab 02. You will see that multiple tests fail. This is because we have changed the semantic behavior of the `char_length` function. Do not worry about the failures for now. We will revert back to the original code at the end of the exercise. These failures demonstrate the benefit of having regression tests packaged with your code!
10. Start the Postgres server and launch the psql client by connecting to the sampledadb database. Test the `char_length` function. You should see the following output:



```
hpenonstop@ubuntu:~/dbsrvr$ psql -d sampledadb
psql (9.5.3)
Type "help" for help.

sampleddb=# select char_length('abc');
 char_length 
-----
           6
(1 row)

sampleddb=#
```

11. If you see the above output, congratulations! You have made your first modification to Postgres code and have made the changes work!
12. Exit the psql client and stop the Postgres server. Roll back the change made in the `textlen` method to ensure the `char_length` function works as expected. Reverse the changes you made in step 6 and repeat steps 7 to 10. Ensure "`make check`" does not report any failures. You should now see the correct value of 3 being returned for the string 'abc'.

Outcome:

In this lab you have made a change to Postgres code and have successfully built the source, installed and tested it.

Questions:

1. Can you get the same effect as returning twice the length of a string without making any changes to the source code? (hint: Think of using the debugger)
2. What happens if you use a non-existent function (say `dummy_function()`) in a select statement as shown below:

```
SELECT DUMMY_FUNCTION();
```

At what point do you think an error should be returned? At what point does Postgres actually detect that such a function does not exist? Walk through the debugger and find out.

3. Why does Postgres use macros for processing arguments to built-in functions? Why is the return type declared as 'Datum'?
4. Look around the `varlena.c` file. What other functions do you see being implemented in this file?
5. How do you figure out which tests in the regression test suite failed because of your changes?

Your notes/observations:

This page is left blank intentionally

Lab 06 – Adding your own custom function as a built-in function of Postgres.

In this lab you will be implementing your own built-in function to Postgres. You will be truly enhancing Postgres! Our goal is to add an encryption function. We will be implementing the popular Jenkins Hash function as a built-in function. For more information on the Jenkins Hash function, read https://en.wikipedia.org/wiki/Jenkins_hash_function. The lab will proceed in three stages:

1. Build the skeleton for the function.
2. Enhance the skeleton to process input arguments using predefined macros.
3. Add the encryption algorithm to the function skeleton.

Stage 1 – Build the skeleton:

In this step we will focus on the steps needed to add a built-in function to Postgres. In the initial version, the function, by itself will not do much. It will return the value 9 for any string input. We will fill in the logic for the function in the next step. Add the following code at the end of the file `src/backend/utils/adt/varchar.c`. Notice that we use a macro to return the value back from the function. This is the recommended way to return values from functions in Postgres. For more information on what the macros mean, refer to section 35.9.4 at:

<https://www.postgresql.org/docs/9.5/static/xfunc-c.html>

```
Datum
jenkins_hash(PG_FUNCTION_ARGS)
{
    PG_RETURN_UINT32(9);
}
```

You now need to assign an object id for your function. The object id is a number that uniquely identifies your function in the database catalog (all RDMS maintain metadata in what is called as the system catalog). Postgres includes a script to find the list of unassigned object ids that are available for use. Navigate to `src/include/catalog` and run the script `unused_oids` from the command prompt. You will get an output similar to the following:



```
hpenonstop@ubuntu:~/dbsrvr/postgres/src/include/catalog$ ./unused_oids
2 - 9
3294
3296
3300
3308 - 3309
3315 - 3328
3330 - 3381
3394 - 3453
3577 - 3579
3997 - 3999
4066
4083
4099 - 4101
4109 - 4565
4569 - 5999
6015 - 8999
9001 - 9999
hpenonstop@ubuntu:~/dbsrvr/postgres/src/include/catalog$
```

We will use the oid 9001 for our function. The oid chosen for the function should be configured in the file `src/include/catalog/pg_proc.h`. Add the following code at the end of the file before the last set of `#defines`

```
DATA(insert OID = 9001 ( jenkins_hash          PGNSP PGUID 12 1 0 0 0 f f f
f t f i 1 0 26 "25" _null_ _null_ _null_ _null_ _null_ jenkins_hash _null_
_null_ _null_ ));
DESCR("jenkins_hash");
```

The above code snippet is essentially filling up a structure. Look at the beginning of `pg_proc.h` file for the following structure definition to understand the above content.

```
CATALOG(pg_proc,1255) BKI_BOOTSTRAP BKI_ROWTYPE_OID(81) BKI_SCHEMA_MACRO
```

The number 26 in our oid definition for the `Jenkins_hash` function above represents the data type of the return value of the function. 26 is the oid of a 4 byte unsigned integer. For a complete listing of the oids of all built in data types, refer to `src/include/catalog/pg_type.h`. The numbers 1 and 0 before 26 represent the number of parameters to the function and the number of parameters with default values respectively.

Build Postgres after making the above changes. Once the build is complete run the "`make install`" task. Once the installation is complete, run the `initdb` script from the command prompt. Delete any existing database objects at `$PGDATA` location using the `rm` command:

```
rm -rf $PGDATA
```

```
initdb
```

Start the postgres server and then create/recreate the `sampledb` database (`createdb sampledb`). Launch the `psql` client (`psql -d sampledb`) and then test the function. You should see the following output.

```
hpenonstop@ubuntu:~/dbsrvr$ ./start_server.ksh
server starting
hpenonstop@ubuntu:~/dbsrvr$ createdb sampledb
hpenonstop@ubuntu:~/dbsrvr$ psql -d sampledb
psql (9.5.3)
Type "help" for help.

sampledb=# select jenkins_hash('abc');
 jenkins_hash
-----
          9
(1 row)

sampledb=#
```

Stage 2 – Enhancing the Function:

In this step we will understand a little more about the manner in which arguments are processed within functions. Postgres documentation recommends that macros are used for processing arguments. For examples of version 1 functions, read the documentation page at:

<https://www.postgresql.org/docs/9.5/static/xfunc-c.html>.

We will use the `VARSIZE` macro to make the `Jenkins_hash` function return the size in bytes of the input argument. This value will be needed anyway for the full implementation of the Jenkins hash function. Modify the function written in the previous step to the following:

```
Datum
jenkins_hash(PG_FUNCTION_ARGS)
{
    // pointer to argument 0.
```



```

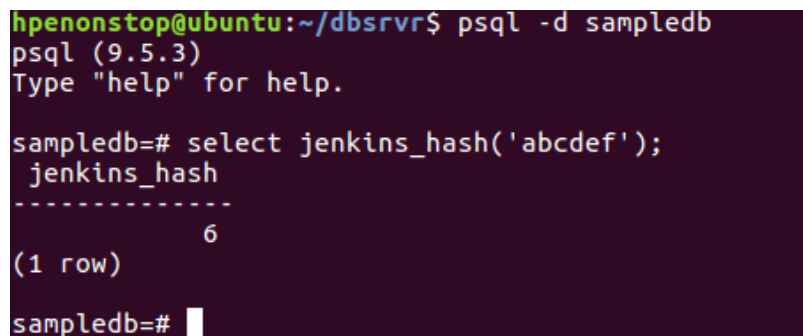
text *txtInput = PG_GETARG_TEXT_P(0);

// Size of argument 0. Refer to the documentation for
// an explanation of VARHDRSZ
uint32 argSize = VARSIZE(txtInput) - VARHDRSZ;

PG_RETURN_UINT32(argSize);
}

```

Compile Postgres again and install. Delete contents of \$PGDATA, run initdb, start the server and create a database named sampledb. You should now be able to invoke your function from the psql prompt as shown below:



```

hpenonstop@ubuntu:~/dbsrvr$ psql -d sampledb
psql (9.5.3)
Type "help" for help.

sampledb=# select jenkins_hash('abcdef');
 jenkins_hash 
-----
              6
(1 row)

sampledb=#

```

Stage 3 – Completing the Jenkins_hash function:

In this step, we will complete the implementation of the Jenkins_hash function. Replace the code written in the previous step with the following:

Datum

```

jenkins_hash(PG_FUNCTION_ARGS)
{
    /* pointer to argument 0. */
    text *txtInput = PG_GETARG_TEXT_P(0);

    /* Size of argument 0. Refer to the documentation for
    an explanation of VARHDRSZ */

    uint32 argSize = VARSIZE(txtInput) - VARHDRSZ;
    char *val = (char*)palloc(argSize);
    memcpy((void*)val, (void*)VARDATA(txtInput), argSize);

    uint32 hashVal=0;
    for(uint32 i = 0; i < argSize; i++)
    {

```

```

        hashVal += val[i];
        hashVal += (hashVal<<10);
        hashVal += (hashVal>>6);
    }
    hashVal += (hashVal<<3);
    hashVal ^= (hashVal>>11);
    hashVal += (hashVal<<15);

    PG_RETURN_UINT32(hashVal);

}

```

Follow the same steps as before to build and install Postgres. Initialize the database, start the database server, create a sample database and launch the psql client. This is what you should see when you use the function:

```

hpenonstop@ubuntu:~/dbsrvr$ createdb sampledb
hpenonstop@ubuntu:~/dbsrvr$ psql -d sampledb
psql (9.5.3)
Type "help" for help.

sampledb=# select jenkins_hash('a');
 jenkins_hash
-----
    4025114553
(1 row)

sampledb=# select jenkins_hash('b');
 jenkins_hash
-----
    29066075
(1 row)

```

Outcome:

You have successfully added your own built-in function to Postgres. You have understood the control and process flow for the compilation and execution of a simple function in Postgres.

Questions:

1. Why are we using `malloc` instead of the C runtime function `malloc` to allocate memory in the `jenkins_hash` function?
2. How would you change the implementation of the `Jenkins_hash` function to return signed integers (negative valued included). Will making this change the hash value for strings? Test and validate.
3. What happens if we pass a null value as input to the Jenkins Hash function? How do you think this is handled? Modify the implementation to return a value of -1 for any null inputs. Is this a safe approach?
4. Analyze the build log and look for any C compiler warnings generated for the segment of code you added. If there are any, resolve them.

5. The function you implemented takes only character input. How do I make it accept an integer input?
6. Why do think we need to call initdb when we add a new function? Change the definition of the function in pg_type.h to accept two arguments. Recompile Postgres, install and launch the client without running initdb. What happens?
7. Look at other popular hash functions. Enhance the Jenkins hash function to be a more generic hash function that takes a second parameter that allows the user to specify the hash algorithm to be used.
8. Study the existing encryption capabilities of PostgreSQL. Provide a way to mark a column as needing encryption while creating the table. Once a column is marked as needing encryption, ensure that any IUD statement on the column will encrypt the data by default. This may require changes in the compiler to automatically insert a call to the encryption function while generating the plan. Select statements with predicates on the column should also work only on encrypted form of the data.

Your notes/observations:

The road ahead

The aim of the hand-on sessions were to get you familiar with a popular, complex relational database management system. PostgreSQL has a large number of users worldwide. These customers have their businesses dependent on the database features provided by PostgreSQL.

The first two labs were designed to make you comfortable with the process of building open source software from its sources and testing your build. Remember the following four steps that are common for all software that depend on the make infrastructure for building:

1. Configure the build scripts. You did not do this in the lab. It was done for you in the VM. The following is the configure command that was used.

```
./configure --prefix=$HOME/dbsrvr --enable-depend --enable-cassert -  
--enable-debug CFLAGS="-ggdb -O0 -g3 -fno-omit-frame-pointer"
```
2. Make – This step compiles the source code.
3. Make Install – This step installs the software.
4. Test – This is optional.

You can try all of the above from the terminal though the Visual Studio Code IDE was used in the labs. The IDE calls these same commands internally.

The third and fourth labs were designed to make you familiar with the basic steps of debugging software. Only the fifth and sixth labs were designed to work on Postgres internals. The first four labs are supposed to provide you with foundational skills to work with open source software.

The exercises have hopefully given you the confidence to work with Postgres internals. Implementing a built-in function was chosen since it provides a relatively easy way to get your hands dirty with Postgres. Now that you know the basics, you should not have trouble reading any portion of the Postgres code. When in doubt, put it through the debugger! We have not touched upon the core database features and algorithms (for example the optimizer or the executor) since it is beyond the scope of a one credit course. You would need the foundations laid by this course anyway. If database systems excite you, you can use the knowledge and skills you have gained in this course to delve deeper into the subject. The following references should help you deepen your knowledge about certain aspects of database systems:

1. A good starting point is the book "Database Systems Implementation" by Hector Garcia et al. Refer specifically to Chapters 6 and 7 on Query Execution and the Query Compiler.
2. Michael Stonebraker et al have written a paper that provides an architectural overview of a database system. You can access it at cs.brown.edu/courses/cs295-11/2006/anatomyofadatabase.pdf. There is a longer version of the paper at db.cs.berkeley.edu/papers/fntdb07-architecture.pdf
3. If you are looking for some core algorithms that form the foundations of most database systems, you should get familiar with at least one or two of the following:
 - a. The Cascades framework for Query Optimization:
15721.courses.cs.cmu.edu/spring2016/papers/graeffe-ieee1995.pdf
 - b. The Aries Recovery Algorithm.
 - c. The Paxos distributed consensus protocol:
research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf
 - d. The B Tree and B+ Tree data structures – These are used in almost all database systems.

- e. If you are interested in optimizing the B-Tree algorithms, look for the publications of Goetz Graefe: <http://dblp.uni-trier.de/pers/hd/g/Graefe:Goetz>
 - f. If you are interested in transaction processing concepts:
A Critique of ANSI SQL isolation levels - <https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/>
You will need access to the ACM digital library to get this paper.
 - g. Database systems usually manage their own memory. The runtime component called the executor allocates a large chunk of memory from the operating system and manages this memory internally. This portion of the executor is called the buffer manager. If you are interested in memory management concepts, lookup implementations of popular buffer managers. Try writing a simple one using the placement new operator of C++.
 - h. Another important algorithm for database systems is sorting. Think of the order by clause you can use in a SQL statement to order the result on a column or set of columns. Database systems typically use an external merge sort algorithm for this. You can check out the sort algorithm used in Postgres by running a select statement with an order by clause through the debugger. Try writing your own merge sort algorithm that can work on data sets larger than what the main memory on your system can hold. The algorithm should also work on all data types (both built in and compound data types).
4. If you are interested in building fault tolerant systems in general (and by induction fault tolerant databases), read this technical report titled *Why do computers stop and what can be done about it?*
www.hpl.hp.com/techreports/tandem/TR-85.7.pdf
 5. To see how the concepts of a database system can be extended to solve a real problem, read the paper titled "*Here are my data files. Here are my queries. Where are my results*" by Ailamaki et al. In this work, they have provided a database layer with SQL querying capabilities over arbitrary log files. This is of immense use when you want to run arbitrary queries over data generated by sensors etc. Many people do not want (or do not have the skills needed) to load such log data into a database before it can be queried using SQL. This was before the era of Hadoop and Hive!
 6. To learn more about Postgres internals:
 - a. The source code comes with some good documentation. Some folders contain a file name README. This file has a good high level overview of the logic/functionality being implemented by the code within the folder. To start with, look at the readme file within src/backend/parser.
 - b. The starting point in the Postgres documentation should be the developer FAQ at https://wiki.postgresql.org/wiki/Developer_FAQ
 - c. Look at the articles written by Pat Shaughnessy. One example is the following:
<http://patshaughnessy.net/2015/11/24/a-look-at-how-postgres-executes-a-tiny-join>
 - d. Tomas Vondra's blog - <https://blog.pgaddict.com>. He is a self-professed Postgres addict.