Database and Information Systems Lab

Team :

200050025 - Bukke Likith Rajesh
200050125 - Sainath Vavilapalli
200050138 - Sri Harsha Thota
200050140 - Suvvari Jaswanth

Detailed Project Report

Title *: Functional Dependency Support in PostgreSQL*

Problem Specification : Currently, we don't have support for functional dependencies for any database today. We would like to add functionality to support functional dependencies like A -> B in PostgreSQL. To be more precise, we would like to add functionality to add functional dependencies and perform functional dependency checks on simple insert statements

Introduction :

Steps involved in the execution of an insert query in PostgreSQL . A few important and relevant functions have been highlighted :

Function call stack :

- ➔ main
- ➔ PostgresMain
- ➔ **exec_simple_query** : This is where query processing starts
- ➔ pg_parse_query : Basic parsing of the query or queries
- ➔ pg_analyze_and_rewrite : Performs parse analysis and rule rewriting
- ➔ PortalStart : Prepares a portal for execution
- ➔ PortalRun : Run a portal's query or queries
- ➔ PortalRunMulti
- ➔ ProcessQuery : Executes a single plannable query within a portal
- ➔ ExecutorRun : This is the main routine of the executor module
- ➔ standard_ExecutorRun

- ➔ **ExecutePlan** : Processes the plan until we have retrieved the required no. of tuples
- ➔ **ExecProcNode** : This executes a node and returns a tuple. Stops when a Null tuple is returned
- ➔ ExecProcNodeFirst
- ➔ ExecModifyTable : Function to perform table modifications
- ➔ **ExecInsert** : If there are any indices associated with the table, this performs ExecInsertIndexTuples before the below
- ➔ table_tuple_insert
- ➔ heapam_tuple_insert
- ➔ heap_insert
- ➔ RelationPutHeapTuple : This actually places the tuple at the specified page

.

What we have implemented :

1) We have added functionality to support queries of the form :

ADD FUNCTIONAL DEPENDENCY IN <table_name> (base_attribute_list) -> (derived_attribute_list)

For example, we can now deal with queries like :
ADD FUNCTIONAL DEPENDENCY IN t (a,b) -> (c,d)
Where t is the name of a relation in the database and a,b,c,d are some attributes in t

2) For every insert statement of the form **INSERT INTO <table_name> VALUES (attribute_list)**, we perform prior functional dependency checks before actually implementing the tuples into the relation. Each relation supports any number of functional dependency constraints

NOTE : Our current implementation doesn't support functional dependency checks for queries like INSERT INTO <table_name> SELECT * FROM …, i.e, when the tuples being inserted into the relation are the result of another query.

Solution Approach :

1) Adding Functional Dependencies :

To handle queries of the form “ADD FUNCTIONAL DEPENDENCY IN …”, we first compare the prefix of the query string with the above string. If there is a match, then we start the actual insertion into a special table called FD_TABLE which we have created for storing all the functional dependencies in the database.

First, we retrieve all the relevant information of the relation on which we are about to create the functional dependency. We have implemented a function called **getRelationMetaData** which fetches the following information about a relation from system information tables :

- Relation ID
- Number of attributes in the relation
- attname, attlen, attnum, attndims, attbyval for each attribute in the relation

Then, we insert a tuple containing the following information into FD_TABLE :

- Relation Name
- Relation ID
- A string containing the attribute numbers of all the base attributes in the FD along with a terminating character
- Number of base attributes in the FD
- A string containing the attribute numbers of all the derived attributes in the FD along with a terminating character
- Number of derived attributes in the FD

2) Performing functional dependency checks before a simple insert statement :

Even here, we first fetch metadata about the relation using the relation name in the insert statement.

We then fetch information about all functional dependencies on the table from the relation FD_TABLE and store it in a special data structure called **fdInfoData**

We store all the attribute values from the query in a special data structure called insert_values. This will be used later for constructing some required queries.

Then, for each functional dependency associated with the table, we do the following :
We construct a query of the form SELECT * FROM <table_name> WHERE b_1 = ".." AND b_2 = ".." AND …. AND b_m = ".." AND (d_1 = ".." OR d_2 = ".." …. OR d_n = "..") where b_1, b_2,..., b_m are base attributes and d_1, d_2,... d_n are derived attributes and the values within ".." are the values of the respective attributes in the query
We must receive a NULL slot for each of the above queries. Otherwise, the functional dependency for which we receive a non null result has been violated.
It is important to note that multiple instances of the same values of the attributes involved in a functional dependency are allowed.

Parts of existing codebase that we have changed :

Many global variables are shared between the files postgres.c and execMain.c

1) **exec_simple_query** - postgres.c

Here, we have written code to handle new syntax for adding functional dependencies.
We first extract the functional dependencies on a relation before actually performing an insert and execute various select queries as mentioned above. If any of the functional dependencies is violated, we'll have a flag called check_violated set to 1. In that case, we flag an error and exit the function.

2) **standard_ExecutorRun** - execMain.c

Whenever we execute a query for fetching the metadata or for checking a functional dependency, we set the sendTuples variable to 0, so that the results of the queries are not printed on the screen and the fact that we are executing more queries in the background is hidden from the user.

3) **ExecutePlan** - execMain.c

Whenever we execute a query for obtaining the metadata, we extract the information from the tuples returned and populate the global data structures with this information.
Also, when performing functional dependency checks, we check only the first tuple returned by the select query. If the tuple is null, we move on to

the next functional dependency, otherwise, we set a flag and break from the loop.
4) New data structures in postgres.h
We have created the following data structures for storing the results of various queries in memory :

- relationAttributeInfo
- relationAttributeInfoData
- fdInfo
- fdInfoData

Note : A few useful utility functions
1) We used the function **slot_getattr(slot, i, &is_bool)** to get the ith attribute from a given slot. This function returns a Datum which is basically a pointer to data
2) To get the actual values of particular types from the slot, the following functions have been used :

- DatumCString - For getting relation and attribute names
- DatumGetInt16 - For attribute metadata
- DatumGetInt32 - For attribute metadata and attribute numbers in an FD
- DatumGetBool - For attribute metadata

3) We have used palloc and pfree instead of malloc to allocate and deallocate memory

Demo :

```
backend> INSERT INTO test_fd VALUES ('a','b','c','d','e','f');
backend> SELECT * FROM test_fd;
        1: a   (typeid = 1043, len = -1, typmod = 14, byval = f)
        2: b   (typeid = 1043, len = -1, typmod = 14, byval = f)
        3: c   (typeid = 1043, len = -1, typmod = 14, byval = f)
        4: d   (typeid = 1043, len = -1, typmod = 14, byval = f)
        5: e   (typeid = 1043, len = -1, typmod = 14, byval = f)
        6: f   (typeid = 1043, len = -1, typmod = 14, byval = f)
       ----
        1: a = "a"     (typeid = 1043, len = -1, typmod = 14, byval = f)
        2: b = "b"     (typeid = 1043, len = -1, typmod = 14, byval = f)
        3: c = "c"     (typeid = 1043, len = -1, typmod = 14, byval = f)
        4: d = "d"     (typeid = 1043, len = -1, typmod = 14, byval = f)
        5: e = "e"     (typeid = 1043, len = -1, typmod = 14, byval = f)
        6: f = "f"     (typeid = 1043, len = -1, typmod = 14, byval = f)
       ----
backend> ADD FUNCTIONAL DEPENDENCY IN test_fd (b,c) -> (d,e);
backend> INSERT INTO test_fd VALUES ('a','b','c','d1','e','f');
2023-05-01 14:03:56.886 IST [12438] ERROR:  Functional dependency check violated
```

Possible optimizations and future work :

1) Parser support :

Instead of identifying the add functional dependency query by comparing strings, we can add a new rule in the parser for identifying this. Then, we can create an InsertStmt* node as we basically just insert into the FD_TABLE for storing the specified functional dependency.

2) Checking functional dependencies at the tuple level :

In the current approach, we can only handle simple insert statements as we retrieve the tuple values to be inserted in exec_simple_query. Instead of this, we can do this in the function ExecInsert, where we can retrieve the data from the slot parameter (which is of the type TableTupleSlot*) and then perform the functional dependency checks using select queries.
Doing it this way would enable us to handle queries like INSERT INTO .. SELECT * FROM … as we check the constraint for every tuple we are about to insert.

3) Using indices :

We can create an index on all the attributes involved in a functional dependency. Then, we can try to closely imitate the function **check_exclusion_or_unique_constraint**. Basically, the function uses a scan key to check for duplicate values. We can initialise the scan key with only the attributes on the left hand side of the functional dependency. Then, while performing the index scan, we can see if there are any tuples with the same values as that of the tuple we are about to insert for the attributes on the LHS of the FD but different values for atleast one of the attributes on the RHS of the FD. The actual comparison between tuples is done in the function **_bt_compare** (BT index is the default in Postgres) in the file nbtinsert.c. We can modify the function or write a new function to perform the check.

4) Handling update statements :

We can also handle update statements in a similar way by capturing the exact location at which the actual update occurs and where the uniqueness checks are performed.

Our Learnings :

1) Dealing with a large codebase and lot of interdependent files
2) Efficiently using GDB for debugging and learning about the function call stack
3) There is a lot to learn from the design structure of the overall postgresql code