Project Report

on

Materialized View Definition and Maintenance

Group No : 2

CS631 - Autumn 2007

By

Amita Savagaonkar(07305002)

Girija Limaye(07305905)

Namrata Nikam(07305082)

Sayali Kulkarni(07305401)

Under Prof. N. L. Sarda

Indian Institute of Technology,
Bombay

November 13, 2007

Contents

1 Introduction

At present, creation of normal views is implemented in postgres. We have implemented materialized view creation, i.e. a table will be created to store the tuples resulting from the view definition and while executing the queries on the view, the tuples will be taken from the table instead of generating those on the fly as in case of normal views. We have also added the materialized view maintenance functionality, wherein, as and when updates are made to the base tables, the materialized view will also be updated.

The advantage of using materialized views is that for the retrieval type of queries which may have to be performed quite frequently, the base table need not be queried each time. Instead, a simple query on the view would retrieve all the required data include the aggregate functions like max, min, avg, sum etc. In case of the standard views, which are currently available in PostGres, whenever data is retrieved from the view, the query is actually fired on the base table on which the view is created. Hence, this is a costly operation. This can be avoided using materialized views which are extremely benificial for aggregate type of queries.

2 Description

Following new commands are added:

1. `CREATE MATERIALIZED VIEW viewname AS view definition`

   - This command will create a table named "viewname" and the tuples resulting from the query in view definition will be inserted into the table.
   - This will also create a function and a trigger on each of the base tables which will be used for view maintenance. Function is required to specify the behaviour of the trigger.

2. `DROP MATERIALIZED VIEW viewname`

   - This command will drop the table corresponding to the viewname.
   - The function and the trigger(s) will also be dropped.

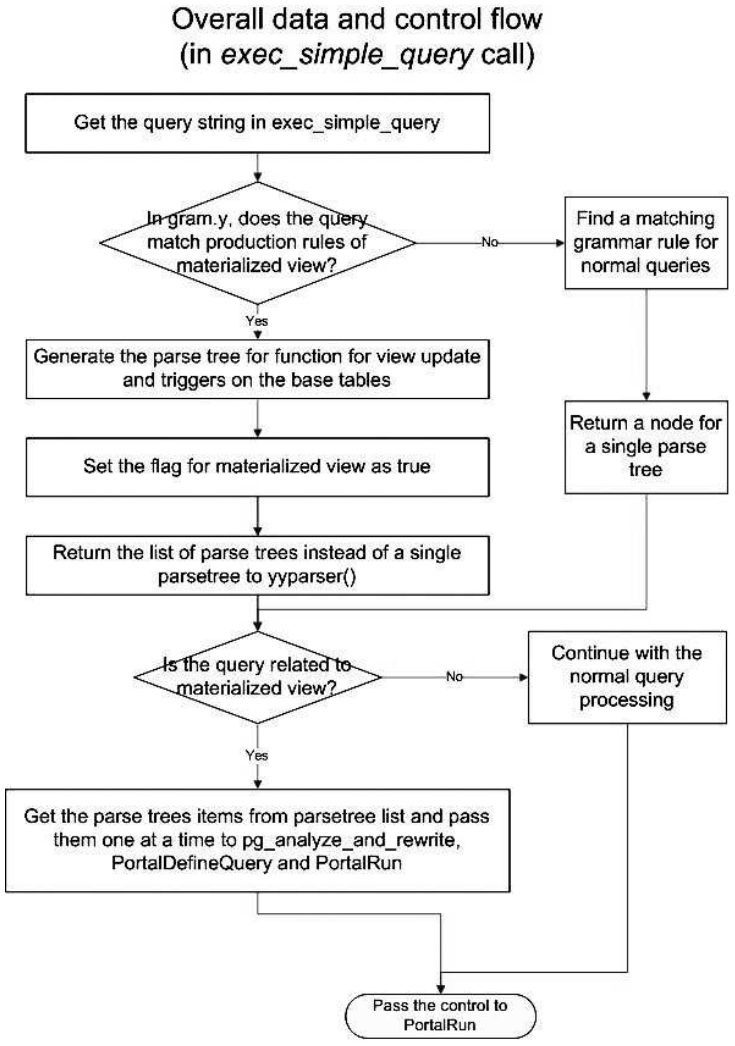
3 Scope of Project

The scope of our project is:

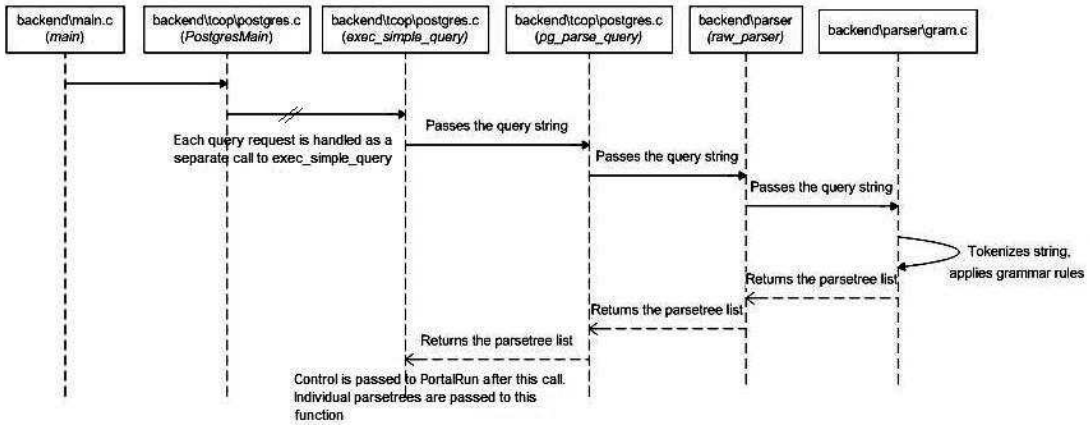
1. Definition of materialized view and creation as actual table.
2. Maintenance of materialized view on insert/delete/update on base table(s).

The materialized view definition supports handling of select, project, join and aggregation queries.

4 Design



Sequence of function calls.



5 Implementation Details

The implementation included the following main steps:

1. Creation of materialized view
2. Addition of triggers (and corresponding functions) on the base tables for the view maintenance
3. Clearing of all the triggers and functions after the view is dropped from the database

5.1 Initial implementation issues

The earlier approach with which we started was to modifiy the current view definition statement in the grammar which is defined in backend/parser/gram.y.
The original ViewStmt production as defined in this file is as follows:

```
ViewStmt: CREATE OptTemp VIEW qualified_name opt_column_list
            AS SelectStmt
```

In this case, we planned to add a new option in the production of *OptTemp* .

Parser stage PostgreSQL parser returns the parse tree for given query string. So in order to do above mentioned operations (function, trigger creation etc.) we need to do one of the following:

1. Write multiple query strings, which parser will translate to corresponding parse trees.
2. Generate multiple parse trees in parser module in order to return a list of parse trees to execute.

PostgreSQL Triggers PostgreSQL supports triggers which execute procedures. So we planned to write a function that will take care of updating the materialized view after updations in table. We then call such a function from the triggers defined on the base tables.

PostgreSQL Functions PostgreSQL supports functions to be written in plpgsql. The functions when written get converted into a parsetree, where the function body remains in the form of a string itself, that corresponds to the plpgsql statements written. So we planned to keep the materialized view definition as the function body itself and generate such function runtime.

5.2 Actual Implementation

The following section includes the code changes done for the implementation of the materialized view creation and updation.

Changes for addition of the keyword MATERIALIZED Added the keyword "materialized" to the file *backend/parser/keywords.c* in the list defined as :

`static const ScanKeyword ScanKeywords[]`

After the code is compiled an entry gets added to the file *backend/parser/parse.h* in the enumeration `enum yytokentype` in the appropriate order as per the alphabetical ordering. Also a definition for enumeration is added in the same file.

Changes in the grammar Definition of the production for the creation of materialized view

```
CreateMatView: CREATE MATERIALIZED VIEW qualified_name OptCreateAs WithOidsAs Sel
{
char *grp2_fun_name;
char *grp2_trg_name;

// create the materialized view
SelectStmt *n = findLeftmostSelect((SelectStmt *) $7);
if (n->into != NULL)
ereport(ERROR,
(errcode(ERRCODE_SYNTAX_ERROR),
 errmsg("My error message <group2>\n ")));
$4->istemp = 0;
n->into = $4;
n->intoColNames = $5;
n->intoHasOids = $6;

grp2_fun_name = (char *) malloc (strlen(n->into->relname) + 4);
sprintf (grp2_fun_name, "%s_fun", n->into->relname);


// create function required to be called from the triggers on base tables
CreateFunctionStmt *f = makeNode (CreateFunctionStmt);
f->replace = FALSE;
f->funcname = list_make1(makeString((char *)grp2_fun_name));
f->parameters = NIL;
f->returnType = makeTypeName(pstrdup("trigger"));

List *l = (List *) malloc (sizeof(List));
char *funbody = (char *) malloc (strlen(selectSub) + 1);

strcpy(funbody, selectSub);
l = list_make1(makeDefElem("as", (Node *)(list_make1(makeString(funbody)))));

l = lappend(l, makeDefElem("language", (Node *)(makeString("plpgsql"))));
f->options = l;
f->withClause = NIL;
```

```
List *lst = NULL;
List *lst_fun = NULL;
lst_fun = lappend(lst_fun, (Node *)f);
lst = lappend(lst, $7);
lst = list_concat(lst, lst_fun);

// iterate on each base table so that we can create a trigger on each
// base table
ListCell *basetable_item;
foreach(basetable_item, n->fromClause)
{
RangeVar *basetable_info = (RangeVar *)lfirst(basetable_item);
/* Create Trigger*/
CreateTrigStmt *t = makeNode(CreateTrigStmt);
List *lst_trg = NULL;
RangeVar *r = makeNode(RangeVar);

r->catalogname = NULL;
r->schemaname = NULL;

r->relname = basetable_info->relname;
grp2_trg_name =
(char *) malloc (strlen(n->into->relname) + strlen(r->relname) + 5);
sprintf (grp2_trg_name, "%s_%s_trg", n->into->relname, r->relname);

t->trigname = grp2_trg_name;
t->relation = r;
t->funcname = list_make1(makeString(grp2_fun_name));
t->args = NIL;
t->before = FALSE;
t->row = TRUE;

char *e = palloc(4);
e[0] = 'i'; e[1] = 'd'; e[2] = 'u'; e[3] = '\0';

memcpy(t->actions, e, 4);
t->isconstraint  = FALSE;
t->deferrable  = FALSE;
t->initdeferred  = FALSE;
t->constrrel = NULL;
lst_trg = lappend(lst_trg, (Node *)t);
lst = list_concat(lst, lst_trg);
}

// this flag is set to true which is a global flag
```

```
// this will be accessed in the file backend/tcop/postgres.c for
// handling certain operations which are specific to materialized views
ismaterialized = 1;

// here instead of returning a single node, we are returning a list
// which will contain all the nodes - one for each of the parse trees
$$ = lst;
}
;
```

Grammar for dropping the materialized view:

```
DropMatView: DROP MATERIALIZED VIEW any_name_list
{
DropStmt *n = makeNode(DropStmt);
n->removeType = OBJECT_TABLE;
n->objects = $4;
n->behavior = DROP_CASCADE;

ismaterialized = 2;

List *lst = NULL;
lst = lappend(lst, (Node *)n);
ListCell *outerlistcell = NULL;
foreach(outerlistcell, $4)
{
// drop function corresponding which is created for view maintenance
List *lst_fun = NULL;
RemoveFuncStmt *f = makeNode(RemoveFuncStmt);
ListCell *innerlistcell = NULL;
List *outerlist = NULL;
char *viewname;

outerlist = (List *)lfirst(outerlistcell);

foreach(innerlistcell, outerlist)
{
Value *vl_viewname = NULL;
vl_viewname = (Value *)lfirst(innerlistcell);
viewname = (char *)vl_viewname->val.str;

char *function_name = (char *) malloc (strlen(viewname) + 4);
sprintf(function_name, "%s_fun", viewname);
f->funcname = list_make1(makeString(function_name));
f->args = NIL;
f->behavior = DROP_CASCADE;
```

```
lst_fun = lappend(lst_fun, (Node *)f);
lst = list_concat(lst, lst_fun);
break;
}
}

$$ = (Node *)lst;
}
;
```

Changes for creation of the view This will actually create a table in the underlying database schema which includes all the data from the base tables as defined in the view. This will also create the function which is required to be invoked from the triggers and the actual triggers on the base table(s). In case there are more than one base tables defined in the view, multiple triggers will be created, one per table.

Since the materialized view creation command will now match with this LHS, the production corresponding to this will be used for the execution of the query. This implementation is exactly similar to the one which was already existing in the postgres code earlier for the creation of a new table from an existing table. This statement is termed as "*CreateAsStmt*" in the grammar file. The queries which are executed using this command are as follows:

`create table <table name> as <select query>`

Such queries get transformed into a select statement of the form

`select <columns> into <table name> from <table list> where <conditions>`

Note that this select statement could also include the group by, having clauses etc.

We are implementing the creation of materialized views in exactly the same fashion as is done in this case. So the parse tree which will be generated for the create materialized view statement will be similar to the one which is generated for one of the above forms.

After the views are created, we are also creating the appropriate functions which will update the contents of the view (required for refreshing the view after the base tables are updated) and the creation of the triggers on the base tables.

The normal implementation of the creation of the table with the above method returns a single node. We have modified this code to return a list, instead of a single node, which includes the following:

1. parse tree for the creation of the table (for the materialized view)
2. parse tree for the creation of the function which will be called from the trigger(s)
3. parse tree(s) for the creation of the trigger(s) for each of the base tables which are referenced in the creation of the materialized view.

The function which is called from the triggers needs to specify the language in which it is written. These functions are written in the language "plpgsql" which needs to be defined before hand. If this language does not exist earlier, the materialized view creation will fail.

The set of above operations is treated to be atomic. Hence if any of the operation fails, the view will not be created. This part of treating the operation as atomic is handled in function *exec_simple_query* in *backend/tcop/postgres.c*. In this function, the for loop which iterates over all the parsetree items in the list will ensure the atomicity of the operation as a whole.

5.3 Files modified

backend/parser/keywords.c Added materialized view keyword

backend/parser/gram.y Added 2 rules - one for creation of the materialized view and one for dropping it.

backend/tcop/postgres.c Handling lists when the materialized view is created/dropped. In all other cases, a single node is returned from *pg_parse_query*. This file also contains the changes for getting the query string which is to be filled in the function body. This is retreived from the original view definition query.

5.4 Sample code output

DB Schema is as follows:

```
create table employee(empid int primary key, ename varchar(100),
 address varchar(100), dateofjoin date);
create table financial_info(empid int references employee(empid),
salary float);
create table reportsto(subordinate int references employee(empid),
 manager int references employee(empid));
```

Example 1

Materialized view query

```
create materialized view empinfo as
select empid, ename, dateofjoin from employee;
```

Actual table creation query

```
{SELECT
   :distinctClause <>
   :into
      {RANGEVAR
      :schemaname <>
      :relname empinfo
      :inhOpt 0
      :istemp false
      :alias <>
```

```
   }
:intoColNames <>
:intoHasOids 2
:targetList (
   {RESTARGET
   :name <>
   :indirection <>
   :val
      {COLUMNREF
      :fields ("empid")
      }
   }
   {RESTARGET
   :name <>
   :indirection <>
   :val
      {COLUMNREF
      :fields ("ename")
      }
   }
   {RESTARGET
   :name <>
   :indirection <>
   :val
      {COLUMNREF
      :fields ("dateofjoin")
      }
   }
)
:fromClause (
   {RANGEVAR
   :schemaname <>
   :relname employee
   :inhOpt 2
   :istemp false
   :alias <>
   }
)
:whereClause <>
:groupClause <>
:havingClause <>
:sortClause <>
:limitOffset <>
:limitCount <>
:lockingClause <>
```

```
   :op 0
   :all false
   :larg <>
   :rarg <>
   }
```

Function creation query

```
(
   {DEFELEM
   :defname as
   :arg ("begin\ delete\ from\ empinfo;\ insert\ into\ empinfo\ select\ empid,
   \ ename,\ dateofjoin\ from\ employee;\ return\ new;\ end;")
   }
   {DEFELEM
   :defname language
   :arg "plpgsql"
   }
)
```

Trigger creation query(s)

```
Creating trigger [empinfo_employee_trg]
```

Example 2

Materialized view query

```
create materialized view latestemployees as
select empid, ename from employee where dateofjoin > '1-1-2007';
```

Actual table creation query

Similar to the one generated in the previous query

Function creation query

```
(
   {DEFELEM
   :defname as
   :arg ("begin\ delete\ from\ latestemployees;\ insert\ into\ latestemployees
   \ select\ empid,\ ename\ from\ employee\ where\ dateofjoin\ >\ '1-1-2007';\
    return\ new;\ end;")
   }
   {DEFELEM
   :defname language
   :arg "plpgsql"
```

```
   }
)
```

Trigger creation query(s)

Contents from log file: `Creating trigger [latestemployees_employee_trg]`

Example 3

Materialized view query

```
create materialized view joiningcount as
select count(*) as empcount, dateofjoin from employee group by dateofjoin;
```

Actual table creation query

Similar to the one generated in the previous query

Function creation query

```
(
   {DEFELEM
   :defname as
   :arg ("begin\ delete\ from\ joiningcount;\ insert\ into\ joiningcount\ sele
   ct\ count\(*\)
   \ as\ empcount,\ dateofjoin\ from\ employee\ group\ by\ dateofjoin;\ return
   \ new;\ end;")
   }
   {DEFELEM
   :defname language
   :arg "plpgsql"
   }
)
```

Trigger creation query(s)

Contents from log file: `Creating trigger [joiningcount_employee_trg]`

Example 4

Materialized view query

```
create materialized view all_salaries as
select employee.ename, financial_info.salary
from employee, financial_info
where employee.empid = financial_info.empid order by salary desc;
```

Actual table creation query

Similar to the one generated in the previous query

Function creation query

```
(
   {DEFELEM
   :defname as
   :arg ("begin\ delete\ from\ all_salaries;\ insert\ into\ all_salaries\ sele
   ct\ employee.ename,\ financial_info.salary\ from\ employee,\ financial_info
   \ where\ employee.empid\ =\ financial_info.empid\ order\ by\ salary\ desc;\
    return\ new;\ end;")
   }
   {DEFELEM
   :defname language
   :arg "plpgsql"
   }
)
```

Trigger creation query(s)

Contents from log file:

```
Creating trigger [all_salaries_employee_trg]
Creating trigger [all_salaries_financial_info_trg]
```

6 Future Work

Following are some of the proposed future developments:

1. **Incremental updations of materialized views** This will be efficient as it reduces the overhead of deleting entire data and reinserting it again.

2. **Updations through Materialized Views** This refers to updating base tables accordingly when materialized views are updated.