Technical Report Self-Organizing Database

March 10, 2005

Contents

1 License

2 Installation Instructions

[Taken from www.postgresql.org]

2.1 Requirements

1. GNU make is required; other make programs will not work. To test for GNU make enter

   gmake –version

   It is recommended to use version 3.76.1 or later.

2. You need an ISO/ANSI C compiler.

3. gzip is needed to unpack the distribution in the first place.

2.2 Steps

1. Download and unzip the file. Execute the following two commands to unpack the files.

   gunzip postgresql-7.3.4.tar.gz

   tar xf postgresql-7.3.4.tar

   This will create a directory postgresql-7.3.4 under the current directory with the PostgreSQL source code. Change into this directory for the rest of the installation procedure.

2. Run the following sequence of commands to execute the code

   (a) The first step of the installation procedure is to configure the source tree for your system and choose the options you would like. This is done by running the configure script. For a default installation simply enter
   ./configure

   This script will run a number of tests to guess values for various system dependent variables and detect some quirks of your operating system, and finally will create several files in the build tree to record what it found.

   The default configuration will build the server and utilities, as well as all client applications and interfaces that require only a C compiler. All files will be installed under /usr/local/pgsql by default.

(b) Build the code with the following command.

    gmake

    The last line displayed should be

    All of PostgreSQL is successfully made. Ready to install.

(c) Before continuing with the rest of the installation you will have to have the

    su

(d) To install PostgreSQL enter

    gmake install

    This will install files into the directories that were specified in step 1. Make sure that you have appropriate permissions to write into that area.

(e) To add a Unix user account to your system, look for a command useradd or adduser. The user name postgres is often used but is by no means required.

(f) To create a database cluster

    i. mkdir /usr/local/pgsql/data
    ii. chown postgres /usr/local/pgsql/data
    iii. su - postgres
    iv. /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data

(g) Starting the database server: Before anyone can access the database, you must start the database server. The database server is called postmaster. The postmaster must know where to find the data it is supposed to use. This is done with the -D option. Thus, the simplest way to start the server is:

    /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data which will leave the server running in the foreground. To start the postmaster in the background, use the usual shell syntax:

    /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data > logfile 2> &1 &

(h) To start the client give the following command:

    /usr/local/pgsql/bin/psql database-name

2.3 Configuration

There are a number of configuration variables that affect the behavior of the database system. We describe here some of them.You can set them by editing the postgresql.conf file in the /usr/local/pgsql/data directory or wherever your data directory is located.

1. Debug Levels:

   SERVER_MIN_MESSAGES (string)

   This controls how much message detail is written to the server logs. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, and PANIC. Later values send less detail to the logs. The default is NOTICE. We have used DEBUG5 for our experiments.

2. DEBUG_PRINT_PARSE (boolean)

   DEBUG_PRINT_REWRITTEN (boolean)

   DEBUG_PRINT_PLAN (boolean)

   DEBUG_PRETTY_PRINT (boolean)

   These flags enable various debugging output to be sent to the server log. For each executed query, print either the query text, the resulting parse tree, the query rewriter output, or the execution plan. DEBUG_PRETTY_PRINT indents these displays to produce a more readable but much longer output format.

3. ENABLE_HASHJOIN $(boolean)$

   Enables or disables the query planner's use of hash-join plan types. The default is on. This is used for debugging the query planner. Other join types can also be enabled/disabled in a similar way.

   Other Join Types

   - Merge Join
   - Nested-Loop Join

NOTE: There are various other configuration variables but are not relevant for our experiments.

3 Overview

3.1 Introduction

We have mainly implemented three main parts in this project:

1. Proposeview Command

2. Generatelattice Command

3.2 Proposeview Command

1. The Proposeview command takes as input a query, reads the definitions of materialized views from a file and lists down the views that can possibly be used for answering the query efficiently. As a result of this command, we get a new plan tree for the execution of the query. If proposeview command has an ’0’ option, the result shows the plan tree indicating which views are the best for the efficient answering of the query in the log file whether the views are materialized or not. Otherwise, proposeview command has an ’1’ option, the query is executed using the materialized views. If the best views for the query are not materialized, a warning message to alert that these views are not materialized is presented.

2. The Proposeview command is executed when the parser establishes that ’PROPOSEVIEW’ was entered on the command line and the syntax was correct.

3. The function extracts the query from the input statement data structure into a query list. The query structure is sent to convert function (/src/backend/parser/convert.c) to obtain the datalog equivalent for the same.

4. View definitions,required for the execution of the command, are read using the file_parser() function (/src/backend/command/proposeview.c) into a view list.

5. For each query in the query list, and for each view in the view list, unification function is called. As a result of the unification function, we obtain a maptable for each query and store it with that query in the query list. A maptable shows which views can be used for answering the queries.

6. The statistics module traverses the query list analyzing each view definition found in the maptable.

Figure 1: Implementation Framework Block Diagram

3.2.1 Implementation Framework

For the implementation of the PROPOSEVIEW command the following basic steps have been implemented in addition to the existing Postgresql-7.3.4 code.

1. Conversion of SQL to DataLog

   The input query, after it is parsed by the Postgresql parser is converted into a query tree. We have implemented a module to convert the query tree into a DataLog structure. It is to ease the execution of the further modules.

2. Unification Algorithm

   This module takes the view definitions and the query (in DataLog format) as an input and generates a Map Table showing which views can be used for the execution of the input query. The Map Table lists each view along with a list of relations for which the view could be used.

3. Join Enumeration Algorithm

   This module checks the Map Table generated by the above module for each RelOptInfo created.A RelOptInfo is created for each base relation and for each node in the join tree. If a view exits for a RelOptInfo it checks the size of the view (generated by the View Size Estimation module) versus the size of the stored relations. A view is selected if it is cheaper to answer the query (or part it) using the view.

4. View Size Estimation Module

   This module is used for estimating the sizes of the views. It takes a viewname as an input and returns the cost of the view.

5. Execute Version using Meterialized Views Module

   This module is called when all the views suggested by the Proposeview command are already materialized, in which case the query is executed using these views.

Each of the above mentioned modules are discussed in detail in Section 4.

3.3 GenerateLattice Command

1. The GenerateLattice Command is used for queries with aggregation. When a query with aggregation is entered using the GenerateLattice command, the module considers all views that could be used to answer the queries efficiently and outputs the most efficient views which should be materialized for the entire query workload.

2. To retrieve the most efficient views, we can set either the maximum number of views we wish to materialize or the maximum size limit for the views.

3. In the case of GenerateLattice, unlike Proposeview, the view definitions are not read from a file, instead the module considers all views that can be formed using the various combinations of the grouping arguments used in the query.

4. For eg, if the query has grouping arguments A,B and C, then the views that will be considered will have grouping arguments A,B,C,AB,AC,BC and ABC.

3.3.1 Implementation Framework

The following sub-modules have been developed for the GenerateLattice Command.

1. Creation of the Lattice

   Once the query is input using the Generatelattice command, a lattice framework is created. The nodes of the lattice represent the views that will be considered. The Lattice has levels, equal to the number of grouping arguments under consideration and has nodes equal to 2^n-1.

2. Estimation the view sizes

   For each view, that is represented by a node in the lattice, we have to estimate the size of the view.

3. Greedy Algorithm

   The Greedy Algorithm selects the most efficient views that should be materialized for efficient answering of the queries. The views are selected by scanning the lattice structure iteratively till we meet our threshold condition, which could be either the number of views selected or the total size of the selected views. In each scan of the lattice, one view is selected for materialization.

4. Query Rewriting using Views

   Once we have the views we wish to materialize, the queries are re-written in terms of the materialized views.

Each of the above mentioned modules are discussed in detail in Section 6.

4 Proposeview Command

4.1 Introduction

The Proposeview command is used for the execution of queries using views.During the construction of a query plan tree it considers materialized as well as non-materialized views along with the stored relations. A view is chosen in the execution plan if it is cheaper to execute the query using the view rather than the base relation. The proposeview command has two versions, the non-execute version and the execute version. The non-execute version is invoked whether all views that could be used for the efficient execution of the query are materialized or not. It lists the views that may be used for the efficient execution of the query. On the other hand the execution version is invoked when all views needed for the execution of the query are materialized. It actually executes the query using the materialized views. If any views in the view definition are not materialized, it shows a warning message to user. The non execute version can also be explicitly invoked using the 0 option and execute version is invoked using 1 option.

4.2 Syntax

- **Non-execute version**

  PROPOSEVIEW 0 valid-select-statement;

  Proposeview can be used with any valid select statement(without aggregation). The command outputs all views which can be used for the execution of the query along with the sizes of the views as well as the result of the Unification Algorithm discussed later.

- **Execute Version**

  PROPOSEVIEW 1 valid-select-statement;

  Execute version of proposeview shows the query result for valid-select-statement as a output. This case just deals with materialized views.

Example

Consider we have the following schema

P(X,Y), R(I,J,K), S(A,B)

and the following proposeview command,

```
PROPOSEVIEW 0 select * from p,r,s where p.y = r.i and r.j=s.a and s.b=3;
```

The following output will be generated. Please refer to the appendix for the complete output. This is the Plan Tree that will be generated, it indicates that view V4 can be used for the execution of the query in place of the stored relation r.

```
***************************************************
                PLAN   TREE
***************************************************
   MERGEJOIN :c=180.84..368.21 :r=14282 :w=20
      l: SORT :c=141.10..146.15 :r=2020 :w=8
         l: SEQSCAN :c=0.00..30.20 :r=2020 :w=8  ( p )
      r: SORT :c=39.74..40.15 :r=166 :w=12
         l: MERGEJOIN :c=30.46..33.62 :r=166 :w=12
            l: SORT :c=0.01..0.02 :r=0 :w=0
              VIEW : V4  ( r )
            r: SORT :c=18.81..18.93 :r=47 :w=4
               l: SEQSCAN :c=0.00..17.51 :r=47 :w=4  ( s )

***************************************************<>
```

4.3 Conversion of SQL to DataLog

4.3.1 Introduction

In this section we discuss the conversion of an SQL Query to Datalog. We assume only simple queries with no sub-queries.The query can have aggregatation or group by clause. The where clause can have multiple AND clauses.NOT and OR clauses, in the where clause are not considered.

All queries are represented as a parse tree by the parser module in Postgresql.The parse tree is an internal representation of an SQL statement. The query is parsed into the Query data structure. Pleas refer to the appendix for the complete structure. The relevant elements of the Query data structure are as follows:
(defined in /src/include/nodes/parsenodes.h)
{
CmdType commandType;   -indicates whether it is a select/create/delete/update statement
Node *utilityStmt;   -non null if its a non-optimizable statement
RangeVar *into;
List *rtable;   -list of rabge table entries
FromExpr *jointree;   -represents a from where construct
List *targetList;   -a list of projection variables along with their result domains

–internal to planner–
List *base_rel_list;   -list of the base relations used in the query
}

For now, we will only be looking at the following attributes of the structure:

- List *rtable
- List *targetList
- FromExpr *jointree

The lists which are marked as internal to planner are used for the RelOptInfo structure and so we may need to look into it.

List *rtable:
The range table is a list of base relations that are used in the query. Every range table entry identifies a table or a view and tells by which name it is called in other parts of the query.In the parse tree range table entries are referenced by index rather than by name. In a SELECT statement these are the relations given after the FROM keyword. Each base relation is assigned a relid which is also stored in the range table. Each attribute in a relation is assigned an attribute number and a table number.
For eg: If we have a table A with m attributes. Suppose A is the nth entry in the range table. If we want to refer to the kth attribute of the table we can reference it as varattno k of varno n,or the kth attribute of the nth entry in the table.

List *targetList:
The target list is a list of expressions that define the result of the query.In case of a Select statement,they are the expressions that build the final output of the query. All attributes mentioned in the select clause are in this list.
Every entry in the target list contains an expression that can be a constant value,a variable pointing to an attribute of one of the relations in the range table, a parameter,or an expression tree made of functioncalls, constants, variables, operators etc. The varibles are in terms of there corresponding range table entries as explained above.

FromExpr *jointree
FromExpr is another primitive node type defined in /src/include/nodes/primnodes.h
It has a fromList and a node to represent the qualifiers on joins. The query's qualification is an expression which corresponds to the WHERE clause of an SQL statement The qualifier has an operator, and arguments which could be variables or constants. The operator is assigned an operator Id.

4.3.2 Data Structure used for DataLog

(/src/include/proposeview/unify.h Section 1)
Each DataLog Query has the following structure:

1. A list of sub goals that form the RHS of the datalog query

2. A target sub goal which is the LHS of the datalog query

A Sub goal list is a linked lists of all the subexpressions in the query. Each subgaol in turn has the following structure

1. A sub goal head or the name of the sub goal

2. A linked list of attributes

An attribute has the following structure:

1. Attribute value

2. Flag ’type’ which shows whether it is a variable or a constant

3. Flag ’proj’ which shows whether it is a projected attribute or not

4. Value ’original_str_value’ which stores the name of the attribute.

5. Value ’original_type’ shows whether the attribute is a variable or a constant.

6. Flag ’equal_to_constant’ that indicates whether the attribute is in the where clause with an equality condition to a constant value.

7. Flag ’aggregated_argument’ that indicates whether the attribute in the target subgoal is an argument to any aggregate function.

8. Value ’aggregated_function_id which saves the aggregate function.

4.3.3 Algorithm for conversion of a SQL Query to DataLog

(/src/backend/parser/convert.c Section 1)
Input: SQL Query
Output: Datalog Query
Procedure:

1. Check the query whether it is a Simple Select Statement.

   (a) Check the CmdType field. It is set to 1 in case of a select statement
   { command=1 for select statements }

   (b) For now we only consider simple queries with no sub queries or having clause so the following two attributes should be false
   {
   hasSubLinks=false (this is true if there are any subqueries)
   hasAggs=false (this is true if some aggregate function is used)
   }

2. Write sub goals along with the attribute names

   (a) For each entry in the rtable list create a subgoal
   { create subgoal rtable->eref->aliasname }

   (b) Add the subgoal to the subgoal list for the query

   (c) For each subgoal add the attributes from the colnames field in the corresponding rtable entry

   (d) Add the created subgoal to the goal list

3. Check for the where clause in the SQL Query

   (a) Check if a join condition exists
   {
   jointree->quals is not null
   }

      i. If a join exists check for the operator type.

         A. If it is an Operator Expression i.e there is only clause with an equality condition,call the opclause function

         B. If it is an AND Expression call the andclause function

4. Create the left hand side of our datalog view.
   Refer to the TargetList

   (a) If the target list entry refers to a varible, then get the attribute from the goal list

   (b) If the target list entry refers to a constant, then store the value of the constant

4.3.4 Algorithm for translating the Operator Expression

(/src/backend/parser/convert.c Section 2)
Input: Expression Node
Output: Modified Goal List for the query
Procedure:

1. There can be two cases:
   - Equality between two variables
   - Equality between a variable and a constant
2. Case 1: Set variable on the right of the equality equal to the one on the left and set the appropriate flags. In case the 2nd variable has already been set to a constant value as a result of some previous condition, then set variable on the left equal to the one on the right.
3. Case 2: Set the variable equal to the constant and set the appropriate flags. Constants are stored as an attribute binary value called Datum in Postgresql, change the datum to its corresponding value using functions defined in Postgresql such as DatumGetInt32 etc depending on the data type.

4.3.5 Algorithm for translating the And Expression

(/src/backend/parser/convert.c Section 3)
Input: Expression Node
Output: Modified Goal List for the query
Procedure:

1. Use the canonicalize qual function (defined in /optimizer/prep/prequal.c) to convert the tree of AND clauses into a simple list of AND clauses.
   - Canonicalize Qual Function Procedures
     - Flatten AND and OR groups throughout the tree.
     - Push down NOTs.
     - Choose whether to convert to CNF, or DNF.
     - Normalize into conjunctive normal form or disjunctive normal form.
     - Convert to implicit-AND list if requested.
2. Scan the list, to first translate the AND clauses having equality conditions with constant values.

3. Call the opclause function for each such clause
4. Now translate the AND clauses having equality between two variables.
5. Call the opclause function for each such clause

Note: If we first check for equality conditions between two variables, and if there exists more than one conditions on the same variable we might reset the values. For Eg: Consider the query select * from R,S where a=c and c=10; If we first set c equal to a, then we might not be able to translate the clause c=10. On the other hand if we first translate c=10, then when we translate a=c we know that c is already set to a constant and so we will set a =10 giving us the correct query.

4.3.6 Modifications for Queries with Aggregation

For ProposeView to execute queries with aggregation, the following changes were made to this module:

1. New fields added to the Datalog structure:

   (a) original_str_value and original_type: original_str_value stores the name of the attribute. It is a copy of the attributes value field. original_type shows whether the attribute is a variable or a constant. It is a copy of the type field. The Operator Expression translation algorithm does the following:

      i. Set variable on the right of the equality equal to the one on the left

      ii. Set the variable equal to the constant

   As a result attributes original value and type is lost. This original value and type is required to form the final list of grouping arguments. Example:

   ```
   Select a, b, c, x, y from p, q
   where p.a = q.x and q.x =9
   ```

   Operator Exp translation algorithm will set p.as value to 9 and type to constant. Thus original_str_value and original_type is used to retain the name and type of the attribute.

   (b) equal_to_constant: It is a flag that indicates whether the attribute is in the where clause with an equality condition to a constant value. Example:

   ```
   Select a, b, c, x, y from p, q where p.a = q.x and q.x =9
   q.x will have equal\_to\_constant = 1
   ```

(c) aggregated_argument and aggregated_function_id aggregated_argument is a flag that indicates whether the attribute in the target subgoal is an argument to any aggregate function. And if it appears as an argument to any aggregate function then the aggregated_function_id determines the aggregate function. aggregated_argument is obtained from target $->expr->$ aggfnoid node of the Query structure.

```
Aggregate Function aggregated_function_id
AVG       1
SUM    2
MAX      3
MIN       4
COUNT 5
```

Example:

```
Select a, b, SUM(c) from p group by a, b
```

In the target subgoal c will have aggregated_argument = 1 with aggregated_function_id = 2

4.4 Unification Algorithm

4.4.1 Introduction

As discussed earlier, this module is used to basically determine which of the views can be used for the execution of the query. It reads the view definitions and the input query and matches the subgoal heads and the attributes in the query with those in the views. It also considers the join conditions in the query. The output from this module is in the form of a Map Table,which lists the views that may be used for the execution of the query along with the subgoals it can replace. It also gives the substitutions, i.e if a view is used for the execution of the query, then how should the attributes of the view be mappedto the query attributes.

4.4.2 The Algorithm

(/src/backend/command/proposeview.c)

- **Input** — The algorithm gets datalog for a query and view definition.
- **Output** — Creates a maptable for the query and view that incorporates the substitutions and the delete list. The delete list contains relation ids which are in the views.The maptable is stored with the query.

- **Assumptions**
  - — Considers presence of unlimited number of self joins in the view definitions.
  - — Still not considering inequalities.
  - — Still not considering multiplicity of self join in the query definition.
  - — The view variables are renamed by appending '_'. Renaming strategy presumes that query datalog renaming strategy does not do the same.
- **Notes**
  - — Mapping of multiple view subgoals to same query subgoal is valid. Also the order in which the substitutions are computed for view-query subgoals; doesn't matter because there are no self-join allowed in the query and hence there is only one way in which the substitutions can be made.
  - — Algorithm considers mapping of the same constants onto themselves. [Case: View p(x,x) Query p('a','a')]
  - — Algorithm considers the valid substitution criteria given in definition 3.4 in the paper [1]. That is,
    - only projected variables may be mapped to constant
    - only projected view variables can map to same query variable

Algorithm

1. Setup for running the process
   (a) Rename the view variables such that there is no overlap in the variable-names of query and view.
   (b) Number the list of subgoals in query and view RHS from 0 onwards, this is just to ease the bookkeeping for the Map Table. (NOTE: This is where we find it plausible to assign actual relids of the relations the view/query are defined on. And use these relids in the delete list of maptable cells as well.)
   (c) Reflect in the view and query definition, which variables are projected. This helps in checks in the algorithm.
   (d) Sort the list of subgoals for query and view RHS, this is just to ease the traversal of list to find a matching.
2. Traverse the list of view RHS subgoals one by one and do this step for each

(a) Find query RHS subgoal with same predicate name as the selected view subgoal and try to see if we can find a mapping. For now, since we have not considered self joins in query, we dont have to worry about back-tracking and checking ”potential” permutations. So the search for the matching query predicate is limited to traversing the list of subgoals and this is also where sorting the subgoals on predicate name is helpful.

(b) If no matching query subgoal found, then FAIL.

(c) Find substitution for the subgoal, Traverse through the attributes of the two subgoals one by one: say view subgoal attribute is X, and query subgoal attribute is Y

   i. If X is a constant: Check if Y is the same constant,

      A. if it is, then it is a trivial substitution, traverse to next attribute, i.e. go to beginning of STEP 2.1

      B. else FAIL as we cannot substitute a different constant/variable for a constant.

   ii. If X is a variable:

   iii. Check if X is a substituted query argument and if it is then check if Y is the same as X

      A. If it is, then it is a trivial substitution, traverse to next attribute, i.e. go to beginning of STEP 2.1 after incrementing pointer to attribute lists.

      B. Else FAIL, as we attempted to map a query variable

   iv. Check if Y is a constant (Def. 3.4: valid substitution requirement check: only projected variables may be mapped to constant)

      A. If it is then, Check if X is in the projected attributes,

      B. if it is not then FAIL as it is violation of def 3.4

      C. Else add the substitution X $\diamondsuit$ Y, move to STEP 2.1.3

   v. Check if Y is already substituted by another view variable
   (Def. 3.4: valid substitution requirement: only projected view variables can map to same query variable)

      A. If it is, check if X is projected variable and check if in earlier substitution X’ is a projected variable

      B. If they both are, then add the substitution X$\diamondsuit$Y, move to STEP 2.1.3

      C. If they are not, then FAIL due to not meeting the valid substitution requirement.

   vi. Add the substitution, X$\diamondsuit$ Y. move on to the STEP 2.1.3

   vii. Substitute the new found mapping to the rest of the subgoal and the view. Move attribute pointers for view and query subgoal in consideration and loop back to STEP

2.1. Also check if the subgoals are defined with different number of attributes, then FAIL.

(d) (Add the subgoal mapped to by the view to the delete list) We make a dummy delete list for each query/view unification. If the subgoal substitutions have come to this step; then all its attributes were substituted and hence add it to the delete list. If we are able to unify the view definition with the query then we would put this delete list in the maptable cell.

3. Check projection from the view. If there was no FAIL then we can substitute the view. But after making the check that the view is able to give out information that is required elsewhere in the query. Here is my strategy to find out if the view is projecting all it HAS to.

   (a) Make the list of n distinct variables appearing in the query
   (b) Initialize a n X 2 array of integers to keep count of variable appearances in the query with 0.
   (c) Traverse the subgoal list of the query and for each attribute that is a variable, find its entry i in the variable list.
      i. If the subgoal is in the delete list: set count[i][0] = 1
      ii. If the subgoal is not in the delete list: set count[i][1] = 1
      iii. Also check for the case when variable is only mentioned in the delete subgoal and projected. In this case force the entry count[i][1] to 1.
   (d) Traverse the list of variables. The variables that have 1 in both columns in the count array are the ones that are deleted by the view substitution and are required by the query elsewhere. Check if the view projects these variables. If not, then FAIL.

4. Make the map Table Entry. Make an entry in the Map Table recording the query subgoal numbers, and substitutions.

4.4.3 File Parser for reading View Definitions

The file_parser function (/backend/commands/proposeview.c Section 3) is used to read the view definitions from a file. The views read from the file are an input to the above discussed Unification Algorithm. The view definitions are stored in a specific format as a text file which this function reads. It converts these view definitions into the corresponding data structure. See appendix for the file format used.
Algorithm

1. Read the file contents from the file (as of now /usr/local/pgsql/data/file.txt) in the file_parser() function.
2. All view definitions are delimited by the newline character. Therefore,each view definition is recognized by the start and end of a line.
3. For each view definition, find ';' character to parse head and subgoal lists.
4. Attributes in the view head and subgoal lists are delimited by character ','
5. In each step,string constant value and integer value or variable name are checked. If double quotes are in the outside of string,it is treated as a string constant value Otherwise,is is regarded as a integer value or variable name
6. If the value is regarded as a integer value or variable name,first character in the string is checked. If the first character starts from integer value,it is treated as a integer value. Otherwise,it is treated as a variable name

4.4.4 Files created or modified

- backend/commands/proposeview.c— This has the functionality of the command.
- include/proposeview/unify.h— Defines all the data structures used in the command execution.
- backend/nodes/copyfuncs.c— Postgresql advises creating functions to copy any data structure you extend.
- backend/nodes/equalfuncs.c— Postgresql advises creating functions to compare any data structure you extend.
- backend/parser/analyze.c— Postgresql analyzer that maps statements to corresponding data structures.It transforms the parse tree into a query tree.
- backend/parser/gram.y— Postgresql parser; this is where the syntax of the statement is determined. Bison parser makes gram.c from this.
- backend/parser/keywords.c— The parser determines the reserved words from this file.
- backend/parser/parse.h— The parser assigns case numbers to strings representing language tokens here.

- backend/tcop/postgres.c— The tcop(traffic cop) for the backend determines what tag to assign to the query structure root node.
- backend/tcop/utility.c— This is the file which invokes the corresponding command handler function.
- bin/psql/sql_help.h— Help on the syntax and purpose of the command.
- bin/psql/tab-complete.c— This is called to complete options from the configuration for the command. Since proposeview is emulating explain; parser still takes in the options, we could use later for anything we desire.
- include/commands/proposeview.h— This defines the command handler prototype for rest of postgresql.
- include/nodes/nodes.h— Defines node type postgresql uses internally to mark this commands structure.
- include/nodes/parsenodes.h— Defines the query structure as entered with proposeview command.

4.5 Join Enumeration Algorithm

4.5.1 Introduction

This section discusses the implementation of the join enumeration algorithm.As discussed before this module is used for selecting the views which lead to efficient query execution. It only considers the views that have been selected by the Unification Algorithm. The main goal of this module is to determine which of the base relations or join relations should be replaced by which view. There may be cases when the same base relation/join relations may be replaces by more than one views, in which case we determine the best plan. A base relation or join relations are replaced by a view only if it is more efficient to answer the query using views than it is using the base relation or join relations. For the implementation the following modifications have been done to the existing code and data structures.

1. Added a flag ”proposeview” to the Query data structure.(/include/nodes/parsenodes.h)
2. Added a ”viewname” field to RelOptInfo data structure.(/include/nodes/relation.h). The optimizer builds a RelOptInfo structure for each base relation used in the query. A RelOptInfo is also built for each join relation that is considered during planning. A join rel is simply a combination of base rels. There is only one join RelOptInfo for any given set of baserels — for example, the join A B C is represented by the same RelOptInfo no matter whether we build it by joining A and B first and then adding C, or joining B and C first and then adding A, etc.
3. Made a copy of the from clause (/backend/parser/sortrelid.c)
   - For this a data structure ”srel” has been created and defined in /include/proposeview/unify.h.
   - All entries of the from clause are appended to a linked list. This linked list is created in /backend/optimizer/util/relnode.c.
   - The linked list is then sorted. The sorting function is called from map_relid() [in proposeview.c Section 2] and defined in /backend/parser/sortrelid.c
4. The data structure defined stores the original relid and the mapped relid. It also stores the relation names. After the sorting is completed, mapped relids are assigned.

4.5.2 Algorithm

1. For each RelOptInfo structure created (in /backend/optimizer/path/allpaths.c) map_relid() (defined in /backend/command/proposeview.c Section 2) is called. The function accepts a relid list as one of its arguments. The relid list is the one for which the RelOptInfo data

structure is being created. It is mapped according to the relids in the srel data structure (defined in /include/proposeview/unify.h).

2. map_relid() function scans the delete list of the map table created by the unification algorithm to check for corresponding views which have the same relid list as the RelOptInfo structure.
3. If any such view exists, it checks for the views with the cheapest cost (in case more than one view exists for the same relid list)
4. The function returns the view with the cheapest cost.
5. After the function has returned the view, the cost of the view is checked against the cost of the base relation (that is the cost stored in the RelOptInfo). If the cost of the view is cheaper the viewname field in the RelOptInfo structure is set to the viewname.

4.5.3 Files modified or created

1. /include/proposeview/unify.h :To add the data structure and function definitions
2. /backend/optimizer/allpaths.c :To call the map_relid function as and when the RelOptInfos are created
3. /backend/optimizer/util/relnode.c :To make a copy of the from clause
4. /backend/parser/sortrelid.c :This is a new file that has been included. It sorts the copied from clause and assigns the new relids.
5. /include/nodes/parsenodes.h :To add a flag proposeview to the query data structure
6. /include/nodes/relation.h :To add a viewname field to the RelOptInfo structure

4.6 View Size Estimation

4.6.1 Introduction

View size estimation module gives the sizes for both, materialized as well as non-materialized views. It is an integral part of the Join Enumeration Algorithm. As discussed earlier, in the join enumeration algorithm, the size of the view is compared with the size of the base tables or even with other views in case there are more than one views which can be used.If the cost of the view is cheaper than the base relations,then the view is used for the construction of plan tree instead of the base relations. The optimizer generates a plan tree for each query which the executor uses for the execution of the query. The view size depends on the number of distinct values and the number of total values which are stored in the system catalog (i.e.,pg_class and pg_statistic). We have assumed that only selection,join and projection operations are used in the query and only equality predicate is treated as all conditions.

4.6.2 Properties

In order to estimate view size,some system properties are used. They are as follows:

- System Catalog: Postgresql stores the number of distinct values and the number of tuples for each relation in a system catalog The number of distinct values (denoted by V(R,a), i.e the the number of distinct values for attribute a in relation R) are stored in the stadistinct attribute in the system catalog which is named pg_statistic. In the system catalog,there are three kinds of values. The pg_statistic catalog is updated by analyze command.

  - A value greater than zero is the actual number of distinct values.
  - A value less than zero is the negative of a fraction of the number of rows in the table
  - A zero value means the number of distinct values is unknown.

  The number of tuples for the relation (denoted by T(R), i.e the number of tuples in relation R) The number of tuples are stored in the reltuples attribute in the pg_class system catalog. The pg_class catalog is updated by vacuum or analyze command.

- Block Size: How many bytes can be contained in one block. It is needed how many block we have to use to carry result tuples. This size is defined during configuration of Postgresql. The default value is 9182.

- CPU Cost per tuple: Sets the query planner's estimate of the cost of processing each row during a query. This is measured as a fraction of the cost of a sequential page fetch. The default is 0.01.

4.6.3 Procedure

1. Estimation of the number of tuples when selection conditions are applied
2. Estimation of the number of tuples when join conditions are applied
3. Estimation of the number of blocks required to retrieve all projected values in the view
4. Estimation of the cpu cost to retrieve blocks that contain result values for the view

4.6.4 View Size Estimation for Selection Condition

The view size in this case depends on the number of tuples resulting from the selection conditions. Any constant values in the datalog definition of the query is regarded as a selection condition. The number of tuples in a relation divided by the number of distinct values gives the estimated view size.

```
V10(l, n) has p(l, 13) t(13, n)
Let the number of tuples in P : 800, denoted by T(P)
Let the number of tuples in T : 1200, denoted by T(T)
Let table P attribute has a,b
Let table T attribute has g,h
Denote the number of distinct value V(a),V(b)..
V(b) = 10 , V(g) = 20

T(P) = T(P) / V(b) = 80
T(T) = T(T) / V(g) = 60
V(b) = 1
V(g) = 1
```

Example for View Size Estimation for Selection Condition

4.6.5 View Size Estimation for Join Operation

Assumptions

1. Containment of value sets
   If relation R and S are two relations with an attribute Y and $V(S,Y) >= V(R,Y)$ then every Y-value of R will be a Y-value of S.

2. Preservation of value sets
   If A is an attribute of R but not of S,then V(R theta join S,A) = V(S,A).

Join size is based on two assumptions which are containment of value sets and preservation of value sets. Therefore,estimated join size is the product of the number of tuples / the multiplication of the distinct number of tuples except minimum one.

```
Let V10(l, n) has p(l, 13) t(13, n)
Let the number of tuples in P : 800,Denote T(P)
Let the number of tuples in T : 1200,Denote T(T)
Let table P attribute has a,b
Let table T attribute has g,h
Denote the number of distinct value V(a),V(b)..
V(b) = 10 , V(g) = 20

T(P) is changed to 80
T(T) is changed to 60
V(b) and V(g) is changed to 1 by selection.

Size = T(P) * T(T) = 80 * 60 = 4800

Let V15(l, n) has p(a, 13) t(13, a)
V(a) = 20, V(h) = 30
Size = T(P) * T(T) / Max(V(a),V(h)) = 4800 / 30 = 160
```

Example for Join Condition without Selection

View Size Estimation with Projection Condition

The projection condition is different from other conditions discussed earlier. In this case we have to consider the number of blocks that will be retrieved as a result of the projection. In the query execution plan, projection condition is considered after all other conditions and operations have been processed. To calculate the size in this case,we have to know the system block size and the size of each attribute. The system block size can be retrieved from the pg_config file. And stawidth value in pg_statistic is the average stored width, in bytes. If the attribute is an integer type, the size of each attribute to be projected is set to 4 because all integer value are 4 bytes. For our implementation we first check for all values in subgoal list. Then we calculate the total bytes for all attributes that are retrieved as a result of the projection condition. Finally, we estimate the total number of blocks.

To calculate this projection size,I have to know how many tuples are in one block.So I used floor(System Block Size/ bytes for projection attribute).And finally to get the number of block I used ceil(result size of all operation / the number of tuples in one block). The final view size means this number of blocks to be retrieved.

```
Let the result size is 160 for selection and join condition.
Let V10(l, n) has p(l, 13) t(13, n) .
Table P and T have all integer type.
Bytes for projection are 4+4 = 8.
Let result size for selection is 4800.
Let system block size is 8192(this is current size in postgres)

the number of tuples in one block = floor(8192 / 8) = 1024
the number of block for this view = ceil(4800 / 1024) = 5.

Final View Size is 5.
```

The projection size example

The final view size

This is the cost that applies all available query condition (selection, join and projection). The final view size is the number of blocks that has to be retrieved plus cpu time to process all result tuples.

4.6.6 Algorithm for View Size Estimation

Input : View name, Map Table **Output :** View cost

1. Find a view definition in datalog for input view name in the maptable using the getDefinition() function.
2. For the view definition retrieved from the above step, check for the existence of where clause by finding same attribute names among subgoal lists - If where clause doesn't exist, the total number of tuples for this query result is regarded as a cartesian product of all relations in the query. - If where clause exists,selection and join size estimation methods are called
3. find the number of tuples and the number of distinct values in the system catalogs by using system cache - Whenever finding the number of tuples in a system catalog for relations in a datalog,these values are multiplied to use later

4. Calculate selection size
5. Find available number of distinct values to prune minimum number of distinct values for join condition
6. join size is getting by multiplied tuple numbers(step 3) and the number of distinct values from step 5
7. gets the total number of bytes for projection condition. - gets width for each attribute that has to be projected from pg_statistic catalog
8. Calculates how many tuples are contained in one block.
9. Calculates how many blocks are retrieved for the query
10. Calculates run cost

4.7 Execute Version of Proposeview

4.7.1 Introduction

The purpose of the execute version of proposeview is making a plan tree and execution of the query with materialized views. The plan tree which comprises the cheapest path information will be input to the executor. The results of proposeview and original SELECT statements should be same. In brief,if a user uses same select statement as an input of proposeview command and the original SELECT command,user has to get same result. The main part of the execute version is how to make a correct plan tree with materialized views because execution of all queries is totally derived from plan tree. When the POSTGRESQL makes a plan tree for original SELECT statement,it recursively makes a plan tree from the bottom and contains all related arguments such as join condition and selection condition which is copied from RelOptInfo to a query. The procedures for a proposeview command are very similar. The proposeview command needs to do more actions to make correct plan tree however,since RelOptInfo for the materialized view just have one more argument to keep view name in order to make a plan tree later and other fields have values for original base relations. It's the role of planner to apply materialized view into the plan tree. Planner has to consider the order of FROM clause because all arguments in the RelOptInfo have table number (the order of the table in the from clause) and attribute number (the order of the attribute in the table) as a pair and values in the RelOptInfo are copied as a argument in the plan tree without any change. There are several required things to apply materialized views to the plan tree. First,new selection, join and projection conditions for them are needed. For example,if we have materialized view **v1** which is the substitution for base relations **p** and **r** for query **Q** ,all conditions for **p** and **r** have to be changed for use of **v1**. Second,new FROM clause is needed because all arguments in the plan tree are derived from the order of tables in the from clause. Third, arguments in the plan tree have to be created or modified with above two requirements. Further details would be explained in other sections.

4.7.2 Algorithm

Example :

1. View Definitions in the file
    - v1(x, z, w) : p(x,y) r(x,z,w)
    - v2(x, y, z, w) : p(x, y) r(y, z, w)
    - v3(x, z) : r(x, y, z)
    - v4(x, y) : r(x, y, z)
    - v5(a, b) : t(a, b, c)
2. Query
    - proposeview 1 select i,k from p,r,t where r.i = p.x and i=2 and m=n; (execute version)
    - proposeview 0 select i,k from p,r,t where r.i = p.x and i=2 and m=n; (Non-execute version)
3. Materialized Views : v1,v2,v3,v5
4. Cheapest Views (Definition in the file) : v3(x, z) : r(x,y, z) for relation r
   v5(a, b) : t(a, b, c) for relation t
5. View Names in the system catalog : v3(x,z), v5(a,b)
6. Attribute Names in the relation : r(i,j,k) t(m,n,l) p(x,y)

r is substituted to v3 and t is substituted to v5.

Procedures for Execute Version

1. When file parser is called,
    - Check whether views in the file are materialized or not. In the example above,v1,v2,v3 and v5 are marked as a materialized flag.

- Form an equivalence class C1(v.attribute-name) for each head argument v.attribute-name of each view v. The goal of class C1 is to rewrite the query. We have to find corresponding arguments for selection,projection and join conditions in the plan tree because scan or join nodes will be replaced with the materialized views. Therefore,we have to know which attribute in the relation is changed to which attribute in the materialized view. For the above example,we have a scan node for materialized view v3 and it has a selection condition;r.i = 2. Because r.i has to be replaced with v3.x,we have to keep this data structure to replace a plan tree. Each class has names of attributes of (a) relations in the deletelist for a view,and of (b) relation for view itself. Recast variable names in terms of attribute positions in the corresponding relations. When file is parsed, the attribute names in the deletelist and in the view head are compared. If the attribute name is same,it is included in the class C1 with the attribute positions in the corresponding relations.

  In the above example,all C1 classes are built (i.e.,C1 from v1 to v5).

  Example for view v3 and v5 :
  C1(v3.x) = {v3.$1, r.$1}
  C1(v3.z) = {v3.$2, r.$3}
  C1(v5.a) = {v5.$1, t.$1}
  C1(v5.b) = {v5.$2, t.$2}

2. Make a warning message for views which are not materialized to show this message to user.

3. If each node in the plan tree will be able to be rewritten using the materialized views, this node type is changed by SCAN node. Because if the materialized view is used, this node is considered as a single relation like SCAN node. So,right and left nodes (below the node for the view) don't need to be maintained any more.

   In the example above, plan tree is built with v3,p and v5.

4. Make a new FROM clause that includes the materialized views. All arguments in the plan tree are based on the position within a FROM clause.

   - Procedures
     (a) Recursively find the SCAN node from top of the plan tree to bottom of the plan tree.
     (b) When SCAN node is found, the name of relation in this node is included in the list.

(c) After finding all SCAN nodes, new FROM clause is made from the final list.

In the above example,
Original FROM clause : p,r,t
New FROM clause : v3,p,v5

5. Make a data structure that contains original and matching new FROM clauses with a corresponding position.

   In the above example,
   {(p : 1,r : 2,t : 3),(v3 : 1, p : 2, v5 : 3)}

6. In order to make a argument for selection condition for the view, we have to find mapping constant values in the output of the unification algorithm. It will be the class C2 that has (1) the constant, (2) all the arguments, from class C1 for the view, that correspond to the argument in the head of the view equated to this particular constant in the rewriting.

   - Procedures
     (a) Find the constant values in the output of the unification algorithm.
     (b) If the names of attributes in the maptable and class C1 for (a) are same, class C2 is added by mergine the contents of class corresponding class C1.
     (c) If other same constant values with collected class C2 are available, these values are added in the class C2 continuously.

   In the above example,
   C2(2) = {2,r.$1,v3.$1}

7. This step is to make arguments for join condition within each view. Any mapping of two or more different variables (that are in the head of the view) into one variable, in the output of the unification algorithm for a view, gives rise to an equivalence class C3 that gives selection conditions on the head of the view, in the node for the view in the query plan.

   - Procedures
     (a) Find any mapping of two or more different variables that are in the head of the view into one variable in the output of the unification algorithm for a view.
     (b) Find a class C1 that contains the same variable name with (a).
     (c) Add into the class C3 by merging class C1 for (b).

   In the above example,
   C3(v5.$1) = {v5.$1,t.$1,v5.$2,t.$2}

8. Update arguments in the plan tree. Plan tree has several arguments that contain projection, selection and join conditions. These conditions are revised by class C1,C2 and C3. Although selection and join conditions for the views are not available, the arguments in the plan tree has to be updated because the contents and the order in the FROM clause are changed (i.e., ’from p, r, t’ ⇒ ’from v3, p, v5’). Plan trees mainly can be separated by scan and join node. Common arguments are ’targetlist’ and ’qual’ and join nodes have additional join clause such as mergeclauses (for merge join) and hashclauses (hash join).

   - Arguments that have to be updated

   - Scan Node
     ’targetlist’ : The argument that has attributes that have to be projected. That means these arguments contain attributes that are needed for join clause later as well as actual projection condition.
     ’qual’ : The argument contains selection and join condition in its own relation. (i.e., i=2 and m=n)
   - Join Node
     ’targetlist’ : The argument that has attributes that have to be projected. That means these arguments contain attributes that are needed for join clause later as well as actual projection condition.
     ’qual’ : There is no attribute
     ’hashclause’ (if join method is hash join) : This clause of the HashJoin node contains information about the join attributes. The values in this clause are 65000 (inner) or 65001 (outer). Because values in the join clause are outer or inner,it doesn’t need to change anything.

   - Procedures to update plan tree

   - Update ’targetlist’ in the scan node.
     - Find the data structure that contains matching information for original and new FROM clause.
     - Change table no. to new table no.
     - If this node has a view name,find class C1 to find correct attribute no.

     In the above example,
     Original ’targetlist’ :
     (table no. : 2, attribute no. : 1) for r.i.
     (table no. : 2, attribute no. : 3) for r.k.

(table no. : 1, attribute no. : 1) for p.x.

Updated ’targetlist’ :
(table no. : 1, attribute no. : 1) for v3.$1.
(table no. : 1, attribute no. : 2) for v3.$2.
(table no. : 2, attribute no. : 1) for p.x.

- Update ’qual’ in the scan node.
  - If the plan tree has view name,make new ’qual’ from class C2 and C3. Also we have to find new table no. from new FROM clause data structure.

  In the above example,
  (table no. : 1, attribute no. : 1) from C2(2).
  (table no. : 3, attribute no. : 1) from C3(v5.$1).
  (table no. : 3, attribute no. : 2) from C3(v5.$2).

  - If the plan tree doesn’t have a view name,replace table number from the data structure for new ’From’ clause.
- Update ’targetlist’ in the join node
  Replace table number by the matching data structure from new FROM clause.

4.7.3 Files created or modified

- commands/proposeview.c
- include/proposeview/unify.h
- parser/gram.y
- parser/analyze.c
- optimizer/plan/createplan.c

4.8 Extension for queries with aggregation

The following changes have been made for executing queries with aggregation using ProposeView.

1. Update View Definitions:

```
Example :
Query : proposeview 1 select x,sum(a) from p,s,r where x = i and y=a gr
Query datalog : V(x, y) <{ p(x, y, z) s(y, b) r(x, j, k)
View Denitions :
v2 ,x,y,w,max(z);p,x,y,z;r,y,z,w
v13,x,y,w;p,x,y,z;r,z,y,w;s,z,u
v14 ,x,sum(y);p,x,y,z;s,y,t
v15,x,y;p,x,y,z
v16 ,x,sum(y);p,x,y,z
```

(a) The character $'\_'$ is added for the view definition that has a aggregation function in the view head.In the above example, V14 and V16 have been appended with $'\_'$

(b) Add aggregation functions with $()$

```
count(variable) : Count function.
max(variable) : Max function.
min(variable) : Min function.
sum(variable) : Sum function.
avg(variable) : Avg function.
```

2. Update File Parser module

(a) If a view definition has a $'\_'$ in the view head, add a flag that indicates that this view has a aggregation function. View name is stored without $'\_'$. In the above example, v2, v14 and v16 are stored to the view head and add a flag aggregation flag to 1.

(b) If a view denition has a '$($' and '$)$' in the view head, finds an aggregation function name between '$($' and '$)$'. Add an aggregation function flag to indicate function name.

```
count : set a flag to 5.
max : set a flag to 3.
min : set a flag to 4.
sum : set a flag to 2.
avg : set a flag to 1.
```

In the above example, we found sum(y) for the aggregation in both v14 and v16. For the sum function, aggregation function name flag set to 2. For max(z) in the view v2, we store aggregation function name flag set to 3.

(c) Store attribute names without '(' and '$)'$. In the above example, we store attribute name for the sum(y), we just store y for v14 and v16. For the v2, we store z.

(d) If aggregated argument in query datalog means that it doesn't contain aggregation function and the view definition contains aggregation function, we set available flag for this view to 0.

3. If aggregated argument in query appears that this query contains aggregation function, update query definition and view definition values. This step is needed to confirm that the number of attributes that has an aggregated function in the query subgoal and view subgoal has to be same.With the above example, the number of attribute ‘y’ is 2 for the query structure

    i. If a query head has an attribute that is aggregated, look through subgoals to find same attribute names.
    ii. If same attribute names are found, count these values and save in the attribute field in the query head.
    iii. If the view and query have an aggregation field and aggregation function numbers are matched with the below Table, we set available flag to 1. Otherwise, we set available flag to 0.
    iv. If view head indicates that the view has an aggregation function, we set a case number to 0. Otherwise, we set to 1. In the above example, the number of attribute ‘y’ in the v14 is 2. the number of attribute ‘y’ in the v16 is 1. For the v2, available flag set to 0.

(a) If available flag in the view definition is 1, only these views are applied to unify module. In the above example, v2 is ignored because available flag is 0.

(b) If heads of view definitions have an aggregation flag(‘1’), compares the number of same attributes that have an aggregation between query and view. The comparison is only applied after confirmation of comparison of subgoal names. If the number of these values are same between views and query, these views are considered. In the above example, v14,v15 and v16 are applied in the unify module. Although v14 and v16 are applied, v16 doesn't have same number of attribute ‘y’ between query and view. Therefore, v16 is ignored.

(c) If aggregation functions in both of view and query are COUNT, we update agg node for the query to SUM.

5 GenerateLattice

5.1 Introduction

The GenerateLattice command is used for queries with aggregation, to get a list of views that should be materialized to answer the queries efficiently. The command takes as input the grouping arguments from the query workload.It generates a lattice structure of all the views that might help in answering the queries. Based on the Greedy Algorithm discussed in the coming section, it picks a set of views which will be optimum for answering the queries.

5.2 Syntax

Generatelattice table-name(arg 1 type-of-arg-1,.........arg n type-of-arg-n);

5.3 Creation of Lattice

5.3.1 Introduction

This section describes the creation of a lattice which is the first step for the execution of the GenerateLattice Command. A lattice structure is used because it helps in maintaining the subset relation for the grouping arguments. Each node in the lattice represents a view which stores information about the view like its grouping attributes, size etc. Also each node in the lattice has pointers to the nodes in the next level of the lattice which can also be referred to as its child nodes. They are those nodes whose grouping arguments are a subset of the grouping arguments of the node under consideration.

5.3.2 Algorithm

1. Input: Given a set of grouping attributes n
2. Output: A lattice of views (Set S of Views)
3. Procedure:
4. Create vertical nodes for the lattice. This will be equal to the number of grouping attributes. This will help us in adding horizontal nodes in the next step.
5. For i=2^ndown to 0
    (a) Create a node, malloc space for a new view

(b) View $\rightarrow$ viewname = string concatenation of V and i

(c) View $\rightarrow$ size = get_view_size()

(d) View $\rightarrow$ grouping_attributes = Binary representation of i

(e) View $\rightarrow$ level = Count_1(binary representation of i)

(f) View $\rightarrow$ flag = false

(g) View $\rightarrow$ next_level = NULL

(h) View $\rightarrow$ query = true/false depends on whether the node represents a query or not.

(i) Check the vertical level at which to add the node. Add the node at the appropriate level.

(j) Each time a new view is added to the lattice, we will check all existing views for the presence of any parent node at the next higher level only. If a parent node exists then initialize the next_level array of pointers to point to this new view added.

5.3.3 Lattice Structure

We can store the lattice as a linked list or array of structures. We can have the following attributes for a view:

1. Viewname: character type
2. Size: float type
3. List of grouping attributes: list of integers (0's and 1's)
4. Level: integer type, to indicate at which level is the view in the lattice
5. Flag : boolean type to show whether the view is already in the set of selected views or not
6. Next_Level: An array of pointers to point to the child nodes or dependent views in our case.

Note:

1. The above algorithm as been implemented as generatelattice() function in backend/commands/generatelattic
2. Count_1 is a function which counts the number of 1's in the binary number.
3. get_view_size() returns the sizes of the views.

5.4 Estimation of View Sizes

As there are a large number of nodes in the lattice, we used the following algorithm to reduce the number of views for which we needed view size estimates.

```
for each view in the lattice

  if any or of the attributes in this view could be used
  to answer any of the (modified) TPC-H queries used in our testing
  then
          mark this view a needing a size estimate
  else
          mark this view as NOT needing a size estimate
```

For each view marked as needing a size estimate, we generated SELECT count(*) SQL statements that were subsequently executed to obtain the view size estimates.

5.5 Greedy Algorithm

5.5.1 Introduction

The Greedy Algorithm is used to pick the most optimal views from the lattice which may be used for answering the queries.It takes the lattice created above as an input. Also the nodes which correspond to the queries are marked explicitly as query nodes.The execution continues till a user defined limit to the number or size of views is met.

5.5.2 Algorithm

- Input: Given a lattice of views
- Output: A selected set S of views to materialize
- Procedure:
- Select the top view, i.e the view with all 1's in its binary representation.
    1. Selecting a view would mean setting the flag to true
- For i=1 to k,where k is the number of views to choose repeat the following steps:
    1. For each node in the lattice:

2. Check if flag = false
3. Clear the mark in the lattice. Mark is a boolean variable used by the depth first search function. For each node that is considered, the dfs() function is called to get a list of all dependent nodes by tracing the $next_level$ pointer recursively. As and when a descendent node is found, MARK is set to true.
4. Find the set of all views which are dependent on V. Call dfs() for View V.
5. For all marked nodes (which means for all dependent nodes of the node under consideration)
6. Look for the cheapest view in the set of already selected views. Let this ne C(u)
7. If C(v)< C(u), then B_w= C(u)- C(v), where B_w is the benefit for w
8. Sum the values of B_w to get B(V,S)

- Choose the view with the maximum value of B(V,S)
- Add V to set S
    1. Set flag to true

5.6 Query Rewriting

The following two algorithms are used to select the best view, based on the output from the generatelattice command, to substitute into each of the query:

1. POD algorithm:

   INPUTS

   - (modified) TPC-H query
   - list of views output from generatelattice
   - view sizes for all views (from I. above)

   Sort views output by generatelattice command by size, ascending

   - In the case of ties in view size, minimal-size
   - view(s) should be the next sort criteria

```
for each view v in the sorted list
    if canBeSubstitutedForPOD(v, q) then
        **return** view name
        exit for
    end if
loop
```

canBeSubstitutedForPOD(view v, query q) returns a boolean

```
    : Assumes that one of the tables in query q is the POD table

    For each attribute in query q, make a list, l, of all attributes that:
        appear in the head of the query (aggregated or otherwise) or
        are used in comparison with a constant value
    loop


    If all the attributes in list l appear in view v then
            : v can be substituted for the fact table in query q
            return true
    else
            : v cannot be substituted for the fact table in query q
            return false
    end if
```

2. Fact algorithm:
   INPUTS

   - (modified) TPC-H query
   - list of views output from generatelattice
   - view sizes for all views (from I. above)

   Sort views output by generatelattice command by size, ascending

   - In the case of ties in view size, minimal-size
   - view(s) should be the next sort criteria

```
for each view v in the sorted list
    if canBeSubstitutedForFact(v, q) then
            return view name
            exit for
```

```
        end if
    loop


canBeSubstitutedForFact(view v, query q) returns a Boolean
        : Assumes that one of the tables in query q is the fact table
        : How to determine if the view, v, can be substituted for the
        fact table the query, q.
    For each attribute from the fact table in query q, make a
    list, l, of all the attributes that are:
        used in a join to another table in the query or
        appear in the head of the query (aggregated or otherwise) or
        are used in comparison with some other attribute or a constant value
    loop
    If all the attributes in list l appear in view v then
        : v can be substituted for the fact table in query q
        return true
    else
        : v cannot be substituted for the fact table in query q
        return false
    end if
```

Finally, the views returned from III. above were substituted into the modified TPC-H queries and executed.

5.7 Implementation Details:

To implement the above the following files have been created/modified:

1. backend/commands/generatelattice.c— This has the functionality for the command.
2. include/generatelattice/lattice.h— Defines all the data structures used in the command execution.
3. backend/nodes/copyfuncs.c— Postgresql advises creating functions to copy any data structure you extend.

4. backend/nodes/equalfuncs.c— Postgresql advises creating functions to compare any data structure you extend.

5. backend/parser/analyze.c— Postgresql analyzer that maps statements to corresponding data structures.It transforms the parse tree into a query tree.

6. backend/parser/gram.y— Postgresql parser; this is where the syntax of the statement is determined. Bison parser makes gram.c from this.

7. backend/parser/keywords.c— The parser determines the reserved words from this file.

8. backend/parser/parse.h— The parser assigns case numbers to strings representing language tokens here.

9. backend/tcop/postgres.c— The tcop(traffic cop) for the backend determines what tag to assign to the query structure root node.

In backend/commands/generatelattice.c, three main functions have been defined, GenerateLattice(), greedy() and dfs().

GenerateLattice () This function creates the lattice as discussed in Section 1. The only input it takes is the number of grouping attributes.

Greedy() This function implements the greedy algorithm as discussed in Section 2.

DFS() This function does a search similar to the depth first search. It is called on a specific node. So the function will trace the $next_level$ pointer recursively down the lattice and mark all such nodes.

6 TPC-H Benchmarking Data

The TPC-H Data has been used for benchmarking. It can be downloaded from [**?**] (We would like to thank Jia Li from University of California, Irvine for her help in setting up the TPC-H data)

1. Download and unzip the file.
2. This will create a directory Appendix in your home directory.
3. The QGEN and DBGEN programs created are used to generate the executable query text and the data that populate the TPC-H Database.
4. To complie the program you have to create a Makefile depending on the database and complier you wish to use. A Makefile-suite is provided.
5. The following changes need to be made

```
CC      = gcc
DATABASE= INFORMIX
MACHINE = SUN
WORKLOAD = TPCH
```

6. Type make to make both qgen and dbgen
7. To generate the database type ./dbgen -s 1. -s 1 represents the scale factor we wish to use to generate the database.
8. As a result of the above command *.tbl files are created in the dbgen directory containing the database.
9. Create tables in Postgresql to store the TPC-H data (See Appendix for SQL Statements)
10. To copy the data to Postgresql the COPY command has been used.

    Example: To copy the orders table, we use the following statement

    COPY orders from ’/appendix/dbgen/orders.tbl’ with delimiter $'|'$;

A Appendix

A.1 Output of PROPOSEVIEW Command

```
PROPOSEVIEW: The datalog format of the query is...
PROPOSEVIEW:            V(x, j) <--  p(x, y) r(y, j, k) s(j, 3)
PROPOSEVIEW: View Defintions...
PROPOSEVIEW:            v13(x, y, w) <--  p(x, y) r(z, y, w) s(z, u)
PROPOSEVIEW:            v12(x, y, w) <--  p(x, y) r(y, z, w) s(z, u)
PROPOSEVIEW:            v11(x, z, u, w) <--  p(x, y) r(y, z, w) s(z, u)
PROPOSEVIEW:            v10(x, z, u) <--  p(x, y) r(y, z, w) s(z, u)
PROPOSEVIEW:            v9(x, z) <--  r(x, y, z) s(y, w)
PROPOSEVIEW:            v8(x, y, w) <--  r(x, y, z) s(y, w)
PROPOSEVIEW:            v7(x, z, w) <--  r(x, y, z) s(y, w)
PROPOSEVIEW:            v6(y, w) <--  r(x, y, z) s(y, w)
PROPOSEVIEW:            V5(a, b) <--  t(a, b, c)
PROPOSEVIEW:            V4(x, y) <--  r(x, y, z)
PROPOSEVIEW:            V3(x, z) <--  r(x, y, z)
PROPOSEVIEW:            V2(x, y, z, w) <--  p(x, y) r(y, z, w)
PROPOSEVIEW:            V1(x, z, w) <--  p(x, y) r(x, z, w)
PROPOSEVIEW: Query with sorted subgoals is ...
PROPOSEVIEW:            V(x, j) <--  p(x, y) r(y, j, k) s(j, 3)

PROPOSEVIEW: The mapTable entries for the workload...
PROPOSEVIEW: For query V :
PROPOSEVIEW:    Mapping with View : V2
PROPOSEVIEW:            Delete List: 0 1
PROPOSEVIEW:            Substitutions:
PROPOSEVIEW:                     x --> x
PROPOSEVIEW:                     y --> y
PROPOSEVIEW:                     z --> j
PROPOSEVIEW:                     w --> k
PROPOSEVIEW:    Mapping with View : V4
PROPOSEVIEW:            Delete List: 1
PROPOSEVIEW:            Substitutions:
PROPOSEVIEW:                     x --> y
PROPOSEVIEW:                     y --> j
PROPOSEVIEW:                     z --> k
```

```
PROPOSEVIEW:    Mapping with View : v8
PROPOSEVIEW:            Delete List: 1 2
PROPOSEVIEW:            Substitutions:
PROPOSEVIEW:                     x --> y
PROPOSEVIEW:                     y --> j
PROPOSEVIEW:                     z --> k
PROPOSEVIEW:                     w --> 3
PROPOSEVIEW:    Mapping with View : v10
PROPOSEVIEW:            Delete List: 0 1 2
PROPOSEVIEW:            Substitutions:
PROPOSEVIEW:                     x --> x
PROPOSEVIEW:                     y --> y
PROPOSEVIEW:                     z --> j
PROPOSEVIEW:                     w --> k
PROPOSEVIEW:                     u --> 3
PROPOSEVIEW:    Mapping with View : v11
PROPOSEVIEW:            Delete List: 0 1 2
PROPOSEVIEW:            Substitutions:
PROPOSEVIEW:                     x --> x
PROPOSEVIEW:                     y --> y
PROPOSEVIEW:                     z --> j
PROPOSEVIEW:                     w --> k
PROPOSEVIEW:                     u --> 3
No matching view found
RELOPTINFO (3): rows=47 width=4

Cost from view 3.000000
Cost from base tables 4.000000
Viewname set to V4
RELOPTINFO (2): rows=200 width=8

No matching view found
RELOPTINFO (1): rows=2020 width=8

Cost from view 2884.000000
Cost from base tables 33.624158
View cost is greater than the cost from base relations
RELOPTINFO (2 3): rows=166 width=12
```

```
Cost from view 6021.000000
Cost from base tables 377.027380
View cost is greater than the cost from base relations
RELOPTINFO (1 2): rows=17222 width=16

Cheaper view already found
Cost of the other view is 5825556.000000
Cost from view 5825556.000000
Cost from base tables 368.211040
View cost is greater than the cost from base relations
RELOPTINFO (2 3 1): rows=14282 width=20
```

A.1.1 Output of Non-Execute Version

```
 ***************************************************
                PLAN   TREE
 ***************************************************
    MERGEJOIN :c=180.84..368.21 :r=14282 :w=20
       l: SORT :c=141.10..146.15 :r=2020 :w=8
          l: SEQSCAN :c=0.00..30.20 :r=2020 :w=8  ( p )
       r: SORT :c=39.74..40.15 :r=166 :w=12
          l: MERGEJOIN :c=30.46..33.62 :r=166 :w=12
             l: SORT :c=0.01..0.02 :r=0 :w=0
               VIEW : V4  ( r )
             r: SORT :c=18.81..18.93 :r=47 :w=4
                l: SEQSCAN :c=0.00..17.51 :r=47 :w=4  ( s )

 ***************************************************<>
```

A.1.2 Output of Execute-Version

```
Query : proposeview 1  select i,k from p,r,t where r.i = p.x and
i=2 and m=n ;

Result:
```

```
WARNING:  v4  is not in the system catalog
 i | k
---+---
 2 | 2
 2 | 2
 2 | 2
(3 rows)
```

A.2 Query Data Structure

```
typedef struct Query
{
NodeTag type;
CmdType commandType;     -indicates whether it is a select/create/delete/update statement
QuerySource querySource;
Node *utilityStmt;     -non null if its a non-optimizable statement
int resultRelation;
RangeVar *into;
bool isPortal;
bool isBinary;
bool hasAggs;     -has aggregate functions
bool hasSubLinks;     -has a sub-query
List *rtable;     -list of rabge table entries
FromExpr *jointree;     -represents a from where construct
List *rowMarks;
List *targetList;     -a list of projection variables along with their result domains
List *groupClause;     -a list of group clauses
Node *havingQual;
List *distinctClause;
List *sortClause;
Node *limitOffset;
Node *limitCount;
Node *setOperations;
List *resultRelations;     -this is used in case of a select into.
–internal to planner–
List *base_rel_list;
List *other_rel_list;
List *join_rel_list;
List *equi_key_list;
– PathKeyItems –
List *query_pathkeys;
} Query;
```

A.3 The Query Tree

The parse tree is generated as an output from the parser module. Consider the following query:

Select a,c from r,s where r.a=s.c;

The following parse tree will be generated:

```
LOG: parse tree:
{ QUERY
:command 1 /*value 1 indicates that it is a select statement*/
:source 0
:utility <>
:resultRelation 0
:into ¡¿
:isPortal false
:isBinary false
:hasAggs false
:hasSubLinks false
:rtable ( /*these are the range table entries for the two tables referenced in the from clause*/
{ RTE
:alias <>
:eref
{ ALIAS
:aliasname r
:colnames ( ”a” ”b” )
}
:rtekind 0
:relid 16977
:inh true
:inFromCl true
:checkForRead true
:checkForWrite false
:checkAsUser 0
}
{ RTE
:alias <>
:eref
{ ALIAS
:aliasname s
:colnames ( ”c” ”d” ”e” )
```

```
}
:rtekind 0
:relid 33370
:inh true 5
:inFromCl true
:checkForRead true
:checkForWrite false
:checkAsUser 0
}
)
:jointree
{ FROMEXPR
:fromlist (
{ RANGETBLREF 1
}
{ RANGETBLREF 2
}
)
:quals /*This is the where clause */
{ EXPR
:typeOid 16
:opType op
:oper /* The operator in the where clause has opno 96*/
{ OPER
:opno 96
:opid 0
:opresulttype 16
:opretset false
}
:args ( /* The arguments in the where clause are variables referenced by there range table entries*/
{ VAR
:varno 1
:varattno 1
:vartype 23
:vartypmod -1
:varlevelsup 0
:varnoold 1
:varoattno 1
}
{ VAR
:varno 2
:varattno 1
:vartype 23
:vartypmod -1 6
:varlevelsup 0
:varnoold 2
:varoattno 1
}
)
}
}
:rowMarks ()
:targetList ( /*This represents the entries in the select clause or the projection variables*/
{ TARGETENTRY
:resdom
{ RESDOM
:resno 1
:restype 23
:restypmod -1
:resname a
:reskey 0
:reskeyop 0
:ressortgroupref 0
:resjunk false
}
:expr
{ VAR
:varno 1
:varattno 1
:vartype 23
:vartypmod -1
:varlevelsup 0
:varnoold 1
:varoattno 1
}
}
```

```
{ TARGETENTRY
:resdom
{ RESDOM
:resno 2
:restype 23
:restypmod -1
:resname c
:reskey 0
:reskeyop 0 7
:ressortgroupref 0
:resjunk false
}
:expr
{ VAR
:varno 2
:varattno 1
:vartype 23
:vartypmod -1
:varlevelsup 0
:varnoold 2
:varoattno 1
}
}
)
:groupClause <>
:havingQual <>
:distinctClause <>
:sortClause <>
:limitOffset <>
:limitCount <>
:setOperations <>
:resultRelations ()
}
```

A.4 File Parser for the Unification Algorithm

The view definitions are stored as follows in a text file.

```
V1,x,z,w;p,x,y;r,x,z,w
V2,x,y,z,w;p,x,y;r,y,z,w
V3,x,z;r,x,y,z
V4,x,y;r,x,y,z
V5,a,b;t,a,b,c
v6,y,w;r,x,y,z;s,y,w
v7,x,z,w;r,x,y,z;s,y,w
v8,x,y,w;r,x,y,z;s,y,w
v9,x,z;r,x,y,z;s,y,w
v10,x,z,u;p,x,y;r,y,z,w;s,z,u
v11,x,z,u,w;p,x,y;r,y,z,w;s,z,u
v12,x,y,w;p,x,y;r,y,z,w;s,z,u
v13,x,y,w;p,x,y;r,z,y,w;s,z,u
```

The file_parser function converts them to the following (Data structure defined in /include/proposeview/unify.h)

```
V1(x, z, w) <--  p(x, y) r(x, z, w)
V2(x, y, z, w) <--  p(x, y) r(y, z, w)
V3(x, z) <--  r(x, y, z)
V4(x, y) <--  r(x, y, z)
V5(a, b) <--  t(a, b, c)
v6(y, w) <--  r(x, y, z) s(y, w)
v7(x, z, w) <--  r(x, y, z) s(y, w)
v8(x, y, w) <--  r(x, y, z) s(y, w)
v9(x, z) <--  r(x, y, z) s(y, w)
v10(x, z, u) <--  p(x, y) r(y, z, w) s(z, u)
v11(x, z, u, w) <--  p(x, y) r(y, z, w) s(z, u)
v12(x, y, w) <--  p(x, y) r(y, z, w) s(z, u)
v13(x, y, w) <--  p(x, y) r(z, y, w) s(z, u)
```

A.5 Code Excerpts

A.6 Code from Unification Algorithm

(/src/backend/commands/proposeview.c Section 1)

```
/*————————————————————————————-
* Algorithm STEP 2: Traverse through subgoal list of view one by one
*————————————————————————————
*/
for (sgls=view.sg_list;sgls!=NULL;sgls=sgls→next) {
        flag=0;
/*——————————————————————————
* Algorithm STEP 2: Find query subgoal with same head.
*——————————————————————————
*/
        for (sgl2=query→sg_list; sgl2!=NULL; sgl2=sgl2→next) {
            i = strcmp(sgl2→sg→head, sgls→sg.head);
            if (i==0)         // Matching head sg found
                break;
            if (i > 0) {
                flag=1;
                return (flag);
            }
        }
       if (i!=0) {
            flag=1;
            return (flag);
        }
/*————————————————————————————
* Algorithm STEP 2.1: Find substitution for the subgoal.
* Traverse through the attribute lists and see if we can find a substitution.
*————————————————————————————
*/
        for (temps=sgls→sg.attlist, temp2=sgl2→sg→attlist;
            temps!=NULL && temp2!=NULL; temps=temps→next, temp2=temp2→next)
        {
            flag = 0;
```

```
/*————————————————————————————————————
* Algorithm STEP 2.1.1: If X is a constant.
*————————————————————————————————————
*/
        if (temps→type != 0) {
        // Check if Y is the same constant
           if(temp2→type != 0) {
              if( compare_att_d (temps→value, temp2→value, temps→type, temp2→type)) {
                 continue;
              }
              else {
                 flag=1;
                 return(flag);
              }
           }
        // In case Y is a variable
           else {
              flag=1;
              return(flag);
           }
        }
/*————————————————————————————————————
* Algorithm STEP 2.1.2: If X is a variable.
*————————————————————————————————————
*/
        else {
/*————————————————————————————————————
* Algorithm STEP 2.1.2.1: Check if X is a substituted query argument.
*————————————————————————————————————
*/
           if (sub!=NULL) {
           // Only if some substitution present
              i=0;
              for(tsub=sub;tsub!=NULL;tsub=tsub→next)
              {
                 if(compare_atts(tsub→q_att.value,temps→value,tsub→q_att.type,temps→type))
                 {
                    if(compare_att_d(temps→value,temp2→value,temps→type,temp2→type))
                    {
                        i=1;
                        break;
                    }
                    else {
                        flag=1;
                        return(flag);
                    }
                }
            }
            if(i!=0)
            {
                // Trivial sub found, move to next attribute
                continue;
            }
        }
/*——————————————————————————————————————————
* Algorithm STEP 2.1.2.2: Check if Y is a constant.
*——————————————————————————————————————————
*/
        if (temp2→type!=0) {
        // Check if X is projected, as only projected are mapped to constants
            if (temps→proj == 1) {
                tsub2 = (subs) malloc (sizeof(struct substitute));
                if (tsub2==NULL) {
                    printf(”PROPOSEVIEW FATAL: Memory Unavailable”);
                    flag = 1;
                    return (flag);
                }
                tsub2→next=NULL;
                if (temps→type==1)
                tsub2→v_att.value.int_val = temps→value.int_val;
                else
                    strcpy(tsub2→v_att.value.str_val, temps→value.str_val);
                    tsub2→v_att.type = temps→type;
                tsub2→v_att.proj = temps→proj;
                if(temp2→type==1)
                    tsub2→q_att.value.int_val = temp2→value.int_val;
                else
                    strcpy(tsub2→q_att.value.str_val, temp2→value.str_val);
                tsub2→q_att.type = temp2→type;
                tsub2→q_att.proj = temps→proj;
                if (sub==NULL)
                    sub = tsub2;
                else {
                    for(tsub=sub;tsub→next!=NULL;tsub=tsub→next)
                        tsub→next=tsub2;
                }
                flag=2;
            }
            else {
                flag = 1;
                return(flag);
            }
        }
/*——————————————————————————————————————
* Algorithm STEP 2.1.2.3: Check if Y is already substituted by another view variable
*——————————————————————————————————————
*/
        if (flag!=2) {
            for (tsub=sub;tsub!=NULL;tsub=tsub→next)
            {
                if(compare_att_d(tsub→q_att.value,temp2→value,tsub→q_att.type,temp2→type))
                    {
                    // Y is already substituted
                        if( temps→proj==1 && tsub→v_att.proj == 1)
                        {
                        // both view var are projected
                            tsub2 = (subs) malloc (sizeof(struct substitute));
                            if (tsub2==NULL) {
                                printf("PROPOSEVIEW FATAL: Memory Unavailable");
                                flag = 1;
                                return (flag);
                            }
                            tsub2→next=NULL;
                            if (temps→type==1)
                                tsub2→v_att.value.int_val = temps→value.int_val;
                            else
                                strcpy(tsub2→v_att.value.str_val, temps→value.str_val);
                            tsub2→v_att.type = temps→type;
                            tsub2→v_att.proj = temps→proj;
                            if (temp2→type==1)
                                tsub2→q_att.value.int_val = temp2→value.int_val;
                            else
                                strcpy(tsub2→q_att.value.str_val, temp2→value.str_val);
                            tsub2→q_att.type = temp2→type;
                            tsub2→q_att.proj = temps→proj;
                            if (sub==NULL)
                                sub = tsub2;
                            else {
                                for (tsub=sub;tsub→next!=NULL;tsub=tsub→next);
                                    tsub→next=tsub2;
                                }
                                flag=2;
                                break;
                            }
                            else {
                                flag=1;
                                return (flag);
                            }
                        }
                    }
            }
/*————————————————————————————————————————————————————————
* Algorithm STEP 2.1.2.4: Add the substitution in faith that if it has reached
* this far, it must be OK
*————————————————————————————————————————————————————————
*/
            if (flag!=2) {
                tsub2 = (subs) malloc (sizeof(struct substitute));
                if (tsub2==NULL) {
                    printf(”PROPOSEVIEW FATAL: Memory Unavailable”);
                    flag = 1;
                    return (flag);
```

```
                    }
                    tsub2→next=NULL;
                    if (temps→type==1)
                        tsub2→v_att.value.int_val = temps→value.int_val;
                    else
                        strcpy(tsub2→v_att.value.str_val, temps→value.str_val);
                    tsub2→v_att.type = temps→type;
                    tsub2→v_att.proj = temps→proj;
                    if (temp2→type==1)
                        tsub2→q_att.value.int_val = temp2→value.int_val;
                    else
                        strcpy(tsub2→q_att.value.str_val, temp2→value.str_val);
                        tsub2→q_att.type = temp2→type;
                    tsub2→q_att.proj = temps→proj;
                    if (sub==NULL)
                        sub = tsub2;
                    else {
                        for (tsub=sub;tsub→next!=NULL;tsub=tsub→next);
                            tsub→next=tsub2;
                    }
                }
/*——————————————————————————————————————
* Algorithm STEP 2.1.3: Propogate that change to rest of the view
*——————————————————————————————————————
*/
                // First rest of the subgoal
                for(temp3s=temps→next;temp3s!=NULL;temp3s=temp3s→next) {
                    if ( compare_atts(temp3s→value,tsub2→v_att.value,temp3s→type,tsub2→v_att.type))
                        {
                            if (temp3s→type ==1)
                                temp3s→value.int_val = tsub2→v_att.value.int_val;
                            else
                                strcpy(temp3s→value.str_val,tsub2→q_att.value.str_val);
                            temp3s→type = tsub2→q_att.type;
                        }
                }
                //And rest of the subgoals
                for(dummys=sgls→next;dummys!=NULL;dummys=dummys→next)
```

```
                    {
                        for(temp3s=dummys→sg.attlist;temp3s!=NULL;temp3s=temp3s→next)
                        {
                            if ( compare_atts(temp3s→value,tsub2→v_att.value,temp3s→type,tsub2→v_att.type))
                            {
                                if (temp3s→type ==1)
                                    temp3s→value.int_val = tsub2→v_att.value.int_val;
                                else
                                    strcpy(temp3s→value.str_val,tsub2→q_att.value.str_val); temp3s→type
= tsub2→q_att.type;
                            }
                        }
                    }
                }
            }
            if ((temps==NULL && temp2!=NULL) || (temps!=NULL && temp2==NULL))
            {
                flag=1;
                return (flag);
            }
   /*————————————————————————————
   * Algorithm STEP 2.2: Add the subgoal id to delete list in faith that if it has
   * reached this far, it must be OK :-)
   *————————————————————————————
   */
            del_lst[lptr]= sgl2→id;
            lptr += 1;
        }
```

Code for Join Enumeration Algorithm

(/src/backend/commands/proposeview.c Section 2)

```
   /*—————————————————————————————-
   * Traverse the maptable to check is there exists a view that matches with the relids
   *—————————————————————————————
   */
       for (query_lst_ptr=query_list;query_lst_ptr!=NULL;query_lst_ptr=query_lst_ptr→next)
       {
           counter=0;
           if (query_lst_ptr→mapped==NULL)
```

```
        {
            v_cost→viewname=NULL;
            v_cost→viewcost=0;
        }
        /* If a maptable exists then check for matching views*/
        else
        {
            mtemp2 = (mtable) malloc (sizeof(struct map));
            for (mtemp2=query_lst_ptr→mapped;mtemp2!=NULL;mtemp2=mtemp2→next)
            {
/*
*---------------------------------------------------------------------------
* First check if the lengths of the 2 lists are the same.The two lists are the relid list obtained
* and passed from the RelOptInfo data structure and the other list is the delete list in the
maptable
*---------------------------------------------------------------------------
*/
                if (lev == *(mtemp2→del_list))
                {
                    flag=false;
                    for (j=0;j<lev;j++)
                    {
                        for(i=0;i<*(mtemp2→del_list);i++)
                        {
                            if (mrel[j]== mtemp2→del_list[i+1])
                            {
                                flag = true;
                                break;
                            }
                            else
                                flag = false;
                        }
                        if (flag ==false)
                            break;
                    }
                    if (flag == true)
                    {
/* get_stats is the function used to get the statistics of views */
viewct = get_stats(mtemp2→view_id,query_lst_ptr→mapped);
fflush(stdout);
/* counter =0 implies that it is the first view found in which
case the minimum cost is initialized to vcost which is the cost of
the first view found */
    if (counter ==0)
    {
        mincost=viewct;
        v_cost→viewname = mtemp2→view_id;
        v_cost→viewcost = viewct;
    }
    /* If it is not the first view found then compare the cost of the
    view with the previously stored minimum cost */
    if (counter != 0) {
        if (viewct < mincost)

            v_cost→viewname = mtemp2→view_id;
            v_cost→viewcost = viewct;
        }
        else
        {
            printf(”Cheaper view already found ”);
            printf(”Cost of the other view is %f”,viewct);
        }
    }
    fflush(stdout);
    counter++;
}
```

;

Code for Obtain view size estimates

```
import java.text.DecimalFormat;
/**
* Author: Charles
* Version: 1.0
*
* This program is used to generate a SQL script of SELECT count(*) statements.
*
**/
public class fact_7 {
      public static boolean isInView(int viewId, int attrId) {
         if ((viewId & attrId) == attrId)
            return true;
         else
            return false;
       //end isInView
         public static void main(String args[]) {
            String LINEITEM = ”lineitem”;
            String VIEW_SIZES = ”vw_sizes_fact”;
            String sql = ””;
            boolean isProd = true;
            String USER = ””;
            String ENTITY = ”TABLE”;
            final int ORDERKEY = 1;
            final int PARTKEY = 2;
            final int SUPPKEY = 4;
            final int DISCOUNT = 8;
            final int RETURNFLAG = 16;
            final int LINESTATUS = 32;
            final int SHIPDATE = 64;
            String[] attr={”orderkey”, //1
               ”partkey”, //2
               ”suppkey”, //4
               ”discount”, //8
               ”returnflag”, //16
               ”linestatus”, //32
               ”shipdate”}; //64
```

```
DecimalFormat fmtName = new DecimalFormat(”000”);
if (args.length > 0 && args[0].equals(”TEST”)) {
    isProd = false;
} //if else
if (isProd) {
    USER = ”postgres”;
} else {
    USER = ”hzgt9b”;
} //end if else

sql = ”” +
    ”\\!echo DELETE FROM ” + VIEW_SIZES + ”\n” +
    ”\\!date’+%D %r’\n” +
    ”DELETE FROM ” + VIEW_SIZES + ”;\n” +
    ”\\!date’+%D %r’\n” +
    ”\\!echoDELETE FROM ” + VIEW_SIZES + ” COMPLETE!\n” +
    ”\\!echo\n” +
    ”\\!echo\n” +
    ”\\!echo \n\n\n”;

System.out.println(sql);

int vwCnt = 0;

for(int i=0; (isProd && (i < Math.pow(2, attr.length))) ||
    (!isProd && (i < 16)); ++i)
{
    String attrs = ””;

    for(int j=0; j < attr.length; ++j)
    {
        if ((i & (int) Math.pow(2, j)) == (int) Math.pow(2, j)) {
            attrs += attr[j] + ”, ”;
        } //end if
    } //end for

    if (
        /* Q1 */ (isInView(i, RETURNFLAG) &&
 isInView(i,LINESTATUS) &&
 isInView(i, SHIPDATE)) ||
 /* Q3 */ (isInView(i,ORDERKEY) &&
 isInView(i, SHIPDATE) &&
 isInView(i, RETURNFLAG)) ||
 /* Q5 */ (isInView(i, ORDERKEY) &&
 isInView(i, SUPPKEY)) ||
 /* Q6 */ (isInView(i, DISCOUNT) &&
 isInView(i, SHIPDATE) &&
isInView(i, RETURNFLAG)) ||
 /* Q7 */ (isInView(i, ORDERKEY) &&
 isInView(i, SUPPKEY) &&
 isInView(i, SHIPDATE)) ||
 /* Q8 */ (isInView(i, PARTKEY) &&
 isInView(i, SUPPKEY) &&
 isInView(i,ORDERKEY)) ||
 /* Q9 */ (isInView(i, ORDERKEY) &&
 isInView(i, PARTKEY) &&
 isInView(i, SUPPKEY)) ||
 /* Q10*/(isInView(i, RETURNFLAG) &&
 isInView(i, ORDERKEY)))
 {
     ++vwCnt;
     attrs = attrs.substring(0, attrs.length()-2);
     sql = ”” +
         ”INSERT INTO ” + VIEW_SIZES + ” (size, name)\n” +
         ”SELECT count(*), ’” + fmtName.format(i) + ”’\n” +
         ”FROM (\n” +
         ” SELECT ” + attrs + ”\n” +
         ” FROM ” + LINEITEM + ”\n” +
         ” GROUP BY ” + attrs + ”\n” +
         ”) As vw;\n”;
     } else {
         sql = ”” +
             ”INSERT INTO ” + VIEW_SIZES + ” (size, name)\n” +
             ”VALUES (0, ’” + fmtName.format(i) + ”’);\n”;
```

```
        } //end if else

    sql = ”” +
    ”\\!echo Begin processing for VIEW ” + fmtName.format(i) + ”\n” +
    ”\\!date ’+%D %r’\n” +

    sql +
    ”\\!date ’+%D %r’\n” +
    ”\\!echo Processing for VIEW ” + fmtName.format(i) + ” COMPLETE!\n” +
    ”\\!echo \n” +
    ”\\!echo \n” +
    ”\\!echo \n\n\n”;

    System.out.println(sql);

} //end for

sql = ”” +
”\\! echo BEGIN COPY into /home/” + USER + ”/” + VIEW_SIZES + ”.txt\n” +
”\\!date ’+%D %r’\n” +
” COPY ” + VIEW_SIZES + ” TO ’/home/” + USER
+ ”/” + VIEW_SIZES + ”.txt’ WITH DELIMITER ’\\t’;\n" +
”\\!date ’+ %D %r’ \n” +
”\\!echo COPY COMPLETE!\n” +
”\\!echo \n” +
”\\!echo \n” +
”\\! echo \n\n\n”;

System.out.println(sql);

sql = ”” +
”\\!date ’+%D %r’ \n” +
”\\!echo Script complete...\n” +
”\\!echo \n” +
”\\!echo \n” +
”\\! echo \n\n\n”;
”\\!echo Number of views: ” + vwCnt + ” \n”;
```

```
        System.out.println(sql);
    } //end main
} //end class fact_7
```

```
import java.text.DecimalFormat;
/**
* Author: Charles
* Version: 1.0
*
* This program is used to generate a SQL script of SELECT count(*) statements.
*
**/
    public class pod_13 {
      public static boolean isInView(int viewId, int attrId)
      {
        if ((viewId & attrId) == attrId)
          return true;
        else
          return false;
      } //end isInView
        public static void main(String args[]) {
          String PROD_OF_DIMS = ”prodOfDims”;
          String VIEW_SIZES = ”vw_sizes”;
          String sql = ””;
          boolean isProd = true;
          String USER = ””;
          String ENTITY = ”TABLE”;
          final int ORDERKEY = 1;
          final int PARTKEY = 2;
          final int SUPPKEY = 4;
          final int DISCOUNT = 8;
          final int RETURNFLAG = 16;
          final int LINESTATUS = 32;
          final int SHIPDATE = 64;
          final int CUSTKEY = 128;
          final int NATIONKEY2 = 256;
          final int ORDERDATE = 512;
          final int TYPE = 1024;
          final int NATIONKEY1 = 2048;
          final int REGIONKEY = 4096;
          String[] attr={”orderkey”,
                   ”partkey”,
              ”suppkey”,
              ”discount”,
              ”returnflag”,
              ”linestatus”,
              ”shipdate”,
              ”custkey”,
              ”nationkey2”,
              ”orderdate”,
              ”type”,
              ”nationkey1”,
              ”regionkey”};

DecimalFormat fmtName = new DecimalFormat(”00000”);

if (args.length > 0 && args[0].equals(”TEST”)) {
    isProd = false;
} //if else

if (isProd) {
    USER = ”postgres”;
} else {
    USER = ”hzgt9b”;
} //end if else

sql = ”” +
”\\!echo CREATE ” + ENTITY + ” ” + PROD_OF_DIMS + ”;\n” +
”\\!date ’+%D %r’\n”;

if (ENTITY == ”VIEW”) {
    sql += ”” +
        ”CREATE ” + ENTITY + ” ” +
        PROD_OF_DIMS + ” AS\n”;
} //end if

sql += ”” +
    ”SELECT –bitvalue\n” +
    ” l.orderkey, –1\n” +
    ” l.partkey, –2\n” +
    " l.suppkey, –4\n" +
    " l.discount, –8\n" +
    " l.returnflag, –16\n" +
    " l.linestatus, –32\n" +
    " l.shipdate, –64\n" +
    " c.custkey, –128\n" +
    " c.nationkey as nationkey2, –256\n" +
    " o.orderdate, –512\n" +
    " p.type, –1024\n" +
    " s.nationkey as nationkey1, –2048\n" +
    " n.regionkey –4096\n";

if (ENTITY == "TABLE") {
    sql += "" +
    "INTO\n" +
    " " + PROD_OF_DIMS + "\n";
} //end if

sql += "" +
    "FROM\n" +
    " lineitem2 l,\n" +
    " customer2 c,\n" +
    " orders2 o,\n" +
    " part2 p,\n" +
    " supplier2 s,\n" +
    " nation2 n\n" +
    "WHERE\n" +
    " l.orderkey = o.orderkey\n" +
    "AND l.partkey = p.partkey\n" +
    "AND l.suppkey = s.suppkey\n" +
    "AND o.custkey = c.custkey\n" +
    "AND c.nationkey = n.nationkey;\n" +
    "\\!date '+%D %r'\n"+
    "\\!echo CREATE " + ENTITY + " " +
    PROD_OF_DIMS + " COMPLETE!\n" +
    "\\!echo \n" +
    "\\!echo \n" +
    "\\!echo \n\n\n";
```

```
System.out.println(sql);

sql = ”” +
”\\!echo DELETE FROM ” + VIEW_SIZES + ”\n” +
”\\!date ’+%D %r’\n” +
”DELETE FROM ” + VIEW_SIZES + ”;\n” +
”\\!date ’+%D %r’\n” +
”\\!echo DELETE FROM ” + VIEW_SIZES + ” COMPLETE!\n” +
”\\!echo\n” +
”\\!echo\n” +
”\\! echo \n\n\n”;

System.out.println(sql);

int vwCnt = 0;
for(int i=0; (isProd && (i < Math.pow(2, attr.length))) ||
(!isProd && (i < 16)); ++i) {
    String attrs = ””;
    for(int j=0; j < attr.length; ++j) {
        if ((i & (int) Math.pow(2, j)) == (int) Math.pow(2, j)) {
            attrs += attr[j] + ”, ”;
        } //end if
    } //end for

    if ( /* Q1 */ (isInView(i, RETURNFLAG) &&
        isInView(i, LINESTATUS) && isInView(i, SHIPDATE)) ||
        /* Q3 */ (isInView(i, ORDERKEY) &&
        isInView(i, ORDERDATE) &&
        isInView(i, SHIPDATE) && isInView(i, RETURNFLAG)) ||
        /* Q5 */ (isInView(i, NATIONKEY1) && isInView(i, ORDERDATE)
        && isInView(i, REGIONKEY)) ||
        /* Q6 */ (isInView(i, DISCOUNT)
        && isInView(i, SHIPDATE) && isInView(i, RETURNFLAG)) ||
        /* Q7 */ (isInView(i, NATIONKEY1) && isInView(i, NATIONKEY2)
        && isInView(i, SHIPDATE)) ||
        /* Q8 */ (isInView(i, NATIONKEY1) && isInView(i, NATIONKEY2)
        && isInView(i, TYPE) && isInView(i, ORDERDATE)
        && isInView(i, REGIONKEY)) ||
        /* Q9 */ (isInView(i, NATIONKEY1) && isInView(i, PARTKEY)) ||
        /* Q10*/ (isInView(i, CUSTKEY) && isInView(i, NATIONKEY2)
        && isInView(i, ORDERDATE) && isInView(i, RETURNFLAG)) )
        {
            ++vwCnt;
            attrs = attrs.substring(0, attrs.length()-2);
            sql = ”” +
            ”INSERT INTO ” + VIEW_SIZES + ” (size, name)\n” +
            ”SELECT count(*), ’” + fmtName.format(i) + ”’\n” +
            ”FROM (\n” +
            ” SELECT ” + attrs + ”\n” +
            ” FROM ” + PROD_OF_DIMS + ”\n” +
            ” GROUP BY ” + attrs + ”\n” +
        ”) As vw;\n”;

    } else {
        sql = ”” +
        ”INSERT INTO ” + VIEW_SIZES + ” (size, name)\n” +
        ”VALUES (0, ’” + fmtName.format(i) + ”’);\n”;
    } //end if else

    sql = ”” +
    ”\\!echo Begin processing for VIEW ” + fmtName.format(i) + ”\n” +
    ”\\!date ’+%D %r’\n” +

    sql +
    ”\\!date ’+%D %r’\n” +
    ”\\! echo Processing for VIEW ” + fmtName.format(i) +
    ”COMPLETE!\n” +
    ”\\!echo \n” +
    ”\\!echo \n” +
    ”\\!echo \n\n\n”;

    System.out.println(sql);
} //end for

sql = ”” +
”\\!echo BEGIN COPY into /home/” + USER + ”/vw_sizes.txt\n” +
        ”\\!date ’+%D %r’\n” +
        ”COPY vw_sizes TO ’/home/” + USER + ”/vw_sizes.txt’
        WITH DELIMITER '\\t';\n” +
        ”\\!date ’+ %D %r’ \n” +
        ”\\!echo COPY COMPLETE!\n” +
        ”\\!echo\n” +
        ”\\!echo\n” +
        ”\\!echo \n\n\n”;

        System.out.println(sql);

        sql = ”” +
        ”\\!date ’+%D %r’\n” +
        ”\\!echo Script complete...\n” +
        ”\\!echo \n” +
        ”\\!echo \n” +
        ”\\! echo \n\n\n”; +
        ”\\!echo Number of views: ” + vwCnt + ”\n”;

        System.out.println(sql);
  } //end main
} //end class pod_13
```

A.7 Output of View Size Estimation

```
*******************   VIEW NAME   ********************
              V1(l m n o )
-------------------------------------------------------------
                   PROJECT SIZE ESTIMATION
-------------------------------------------------------------
 ATTRIBUTE NAME | BYTE SIZE
-------------------------------------------------------------
         l      |         4
         m      |         4
         n      |         4
         o      |         4
    Total byte =========>  16
-----------------------------------------------------
   SYSTEM BLOCK SIZE : 8192
-------------------------------------------------------
 TOTAL SIZE OF QUERY BEFORE PROJECT : 215.000000
 RESULT VIEW SIZE (THE NUMBER OF BLOCKS)
 ceil(SIZE OF QUERY BEFORE PROJECT
      / floor(SYSTEM BLOCK SIZE / TUPLE NUMBER PER BLOCK)
                   =======> 1
*******************   VIEW NAME   ********************
              V3(l m n )
-------------------------------------------------------------
                   PROJECT SIZE ESTIMATION
-------------------------------------------------------------
 ATTRIBUTE NAME | BYTE SIZE
-------------------------------------------------------------
         l      |         4
         m      |         4
         n      |         4
    Total byte =========>  12
-----------------------------------------------------
   SYSTEM BLOCK SIZE : 8192
-------------------------------------------------------
 TOTAL SIZE OF QUERY BEFORE PROJECT : 24.000000
 RESULT VIEW SIZE (THE NUMBER OF BLOCKS)
 ceil(SIZE OF QUERY BEFORE PROJECT
      / floor(SYSTEM BLOCK SIZE / TUPLE NUMBER PER BLOCK)
                   =======> 1
*******************   VIEW NAME   *******************
              V10(l n )
-----------------------------------------------------------
               SELECTION ESTIMATION
------------------------------------------------------------
SUBGOALS : p(l 13 )   t(13 n )
-------------------------------------------------------------
TABLE NAME | TABLE SIZE | NUM. OF DISTINCT VAL. | SELECTION SIZE
------------------------------------------------------------
     p     | 5.000000    | 3.000000 (p.13)       |   1.666667
     p     | 1.666667    | 1.000000 (p.l)        |
     t     | 43.000000    | 8.000000 (t.13)      |   5.375000
     t     | 5.375000    | 0.875000 (t.n)        |
-----------------------------------------------------------------
ALL TABLE SIZE   |   THE MAX. NUM. OF DISTINCT VALUES | SIZE AFTER JOIN
-------------------------------------------------------------
     8.958333   |                1.000000           |    8.958333
------------------------------------------------------------------
                   PROJECT SIZE ESTIMATION
----------------------------------------------------------------------
 ATTRIBUTE NAME | BYTE SIZE
----------------------------------------------------------------------
         l      |          4
         n      |          4
    Total byte =========>  8
----------------------------------------------------------------------
   SYSTEM BLOCK SIZE : 8192
----------------------------------------------------------------------
 TOTAL SIZE OF QUERY BEFORE PROJECT : 8.958333
 RESULT VIEW SIZE (THE NUMBER OF BLOCKS)
 ceil(SIZE OF QUERY BEFORE PROJECT
      / floor(SYSTEM BLOCK SIZE / TUPLE NUMBER PER BLOCK)
                   =======> 1
*******************   VIEW NAME   *******************
              V11(m l k )
```

```
-------------------------------------------------------------------------
               SELECTION ESTIMATION
-------------------------------------------------------------------------
SUBGOALS : p(m 13 )   t(13 l )   s(l 13 k )
-------------------------------------------------------------------------
TABLE NAME | TABLE SIZE | NUM. OF DISTINCT VAL. | SELECTION SIZE
-------------------------------------------------------------------------
     p     | 5.000000    | 3.000000 (p.13)       |   1.666667
     p     | 1.666667    | 1.000000 (p.m)        |
     t     | 43.000000    | 8.000000 (t.13)      |   5.375000
     t     | 5.375000    | 0.875000 (t.l)        |
     s     | 24.000000    | 6.000000 (s.13)      |   4.000000
     s     | 4.000000    | 0.833333 (s.l)        |
     s     | 4.000000    | 1.166667 (s.k)        |
-------------------------------------------------------------------------
          JOIN CONDITION ESTIMATION
-------------------------------------------------------------------------
ALL TABLE SIZE   |   THE MAX. NUM. OF DISTINCT VALUES | SIZE AFTER JOIN
-------------------------------------------------------------------------
     35.833332   |               0.875000            |    40.952379
-------------------------------------------------------------------------
                   PROJECT SIZE ESTIMATION
-------------------------------------------------------------------------
 ATTRIBUTE NAME | BYTE SIZE
-------------------------------------------------------------------------
         m      |          4
         l      |          4
         k      |          4
    Total byte =========>  12
-----------------------------------------------------
   SYSTEM BLOCK SIZE : 8192
-------------------------------------------------------
 TOTAL SIZE OF QUERY BEFORE PROJECT : 40.952379
 RESULT VIEW SIZE (THE NUMBER OF BLOCKS)
 ceil(SIZE OF QUERY BEFORE PROJECT
      / floor(SYSTEM BLOCK SIZE / TUPLE NUMBER PER BLOCK)
                  =======> 1
```

A.8 Data Structure for View Size

```
    typedef struct datalog_size {
        definition_s defi;
        int viewsize;
        struct datalog_size *next;
} *logsize;

logsize lgsize;
```

The structure definition_s defi is the structure to save all view names and subgoals.viewsize that is integer type is the final number of blocks to carry out all operations.

A.9 SQL Statements

The following SQL Statement were used to create tables in Postgresql to load the TPC-H data.

1. create table region (regionkey text, name char(25), comment varchar(152), primary key (regionkey));

2. create table nation (nationkey text, name char(25), regionkey text references region(regionkey), comment varchar(152),primary key (nationkey));

3. create table part (partkey text, name varchar(55), mfgr char(25),brand char(10), type varchar(25),size int, container char(10), retailprice decimal, comment varchar(23),primary key (partkey));

4. create table supplier (suppkey text, name char(25), address varchar(40), nationkey text references nation(nationkey), phone char(15),acctbal decimal, comment varchar(101), primary key (suppkey));

5. create table partsupp (partkey text references part(partkey), suppkey text references supplier(suppkey), availqty int, supplycost decimal, comment varchar(199), primary key (partkey, suppkey));

6. create table customer (custkey text, name varchar(25), address varchar(40), nationkey text references nation(nationkey), phone char(15), acctbal decimal, mktsegment char(10), comment varchar(117),primary key (custkey));

7. create table orders (orderkey text, custkey text references customer(custkey), orderstatus char(1), totalprice decimal, orderdate date, orderpriority char(15), clerk char(15), shippriority int,comment varchar(79),primary key (orderkey));

8. create table lineitem (orderkey text references orders(orderkey), partkey text references part(partkey), suppkey text references supplier(suppkey), linenumber integer, quantity decimal, extendedprice decimal, discount decimal, tax decimal, returnflag char(1), linestatus char(1), shipdate date, commitdate date, receiptdate date, shipinstruct char(25), shipmode char(10), comment varchar(44),primary key (orderkey, linenumber));