PROJECT REPORT

ON

WRITE OPTIMIZE B+ TREE

ON POSTGRES DB

Guided by:
Professor S. Sudarshan

Submitted by:

Pratik Satapathy(153050036)
Sanjay Kumar Dahirey(153050089)
Rahul Sharma(153050019)
Rohit Singh(153050055)

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

MOTIVATION:

Most of the Query optimization are based on data retrieval, very less focus on optimization exists on the data insertion part. Specifically in high update environment e.g. (recording temparatures and pressure at various heights in an weather balloon, GPS co-ordinate tracking of specific object in very small intervals etc.) there arises a need for write optimization.

GOALS:

Write optimization is postgres using Level-1 LSM tree approach where each block of an LSM tree resembles one inherited subtable of postgres Database.

IMPLEMENTAION:

1) Table named X is created. The table has frequent insertions as per high update environment.

2) We have considered a particular count value of rows, as the threshold for table X, so that X stays in-memory. On the condition where the table X achieves this threshold, we create a new subtable of table X named X_a, and copy all data from X to X_a. On the same transaction we also truncate X to make it empty.

3) When again table X reaches threshold value, we copy data from X and X_a and merge it to a new table X_b. We truncate X and drop table X_a. On the subsequent full threshold of table X we perform the above task and toggle using the subtable names X_a and X_b alternatively.

At any instant we have one small table X (assumed to be in-memory) and one larger table X_a or X_b (assumed to be in disk).

Inserting bulk data into subtable takes advantage of faster insertion and index creation using the concept of bottom-up loading of B+ trees.

SOURCE CHANGES: (USING POSTGRES REL_9_4_STABLE 5)(9.4.5)

postgres.h

Declaring a new structure for efficient file write operation on preserving the subtable STATE for a relation.

```
typedef struct
{
      int         relId;                  // rel id of the base relation
      int         cnt;            // rowcount in the base relation
      int         state;                  // state of Writeoptimization
} Table_Details;
```

Declaring two global variables

```
extern int optFlag;
extern int inheritedTableIndex;
```

optFlag is used to prevent control from control going into the writeOptimizer action again in the same run of executor. We set it to 1 as soon as our control goes into the createsubtable() function.

InheritedTableIndex is used to pass the base relation Oid to the execCreateAsStmt.(further details on this are on the Createas.c code change segment)

postgres.c

int optFlag = 0;
int inheritedTableIndex = 0;

First time declaration of the two global variable.

Inside `exec_simple_query(const char *query_string)` function

```
        optFlag = 0;//reset WriteopVal, details on this folowing on another section
        inheritedTableIndex = 0; //reset global value, details on this folowing on another section
```

tablecmds.c

In the `DefineRelation(CreateStmt *stmt, char relkind, Oid ownerId)` function we are storing the intial Write optimization state at a file for the newly created relation.

This state includes <tableOID, numOf Rows, STATE>

STATE can be 0 or 1 or 2

0 = no subtable exists
1 = table_a subtable exists
2 = table_b subtable exists

```
saveStateInfile(relationId,0,0);
```

nodemodifytable.c

This is the primary function which performs write optimize action
`createSubtable(Table_Details t){`

//here we are getting the writeOptimize state from the file for current table

```
        int optimizableTableId = t.relId;
        if (optimizableTableId == 0) return; // file has no data yet
        int rowCount = t.cnt;
        int state = t.state;
        inheritedTableIndex = optimizableTableId;

        if(rowCount < 2000)return;   // checking the table threshold condition

        char *relToOptimize = get_rel_name(optimizableTableId); // getting relation
name from relation Oid
        char *newTbl = get_rel_name(optimizableTableId);
        char *oldTbl = get_rel_name(optimizableTableId);
```

//Checking the Write optimization state and choosing the next subtable name as per t he last state

```
        if(state==0 || state==2)
              strcat(newTbl,"_a");
        if(state==1)
              strcat(newTbl,"_b");
```

//creating RangeVar variable with the new subtable name to be used in the next structure
```
        RangeVar *rltn1 = makeRangeVar(NULL, NULL, -1);
        rltn1->relname = newTbl;
        rltn1->type=T_RangeVar;
        rltn1->inhOpt = INH_DEFAULT;
// including the previous RangeVar structure here
        IntoClause *intoC;
        intoC = makeNode(IntoClause);
        intoC->rel = rltn1;
        intoC->viewQuery = NULL;
        intoC->skipData = false;
//rangevar structure creation for the base table
        RangeVar *rltn2 = makeRangeVar(NULL, NULL, -1);
        rltn2->relname = relToOptimize;
        rltn2->type=T_RangeVar;
        rltn2->inhOpt = INH_DEFAULT;

//CoumnRef structure creation for inclduing all columns of the base relation in the new subtable
        ColumnRef *cr = makeNode(ColumnRef);
        cr->fields = list_make1(makeNode(A_Star));
        cr->location = -1;

//using the above structure in this structure as s
        ResTarget *rt = makeNode(ResTarget);
        rt->name = NULL;
        rt->indirection = NIL;
        rt->val = (Node *)cr;
        rt->location = -1;

// preparing structure of select operation
        SelectStmt *n = makeNode(SelectStmt);
                                   n->targetList = list_make1(rt);
                                   n->fromClause = list_make1(rltn2);
                                   n->intoClause = NULL;
                                   //n->distinctClause = $2;

//preparing createtable as structure to provide to the ExecCreateTableAs() function
        CreateTableAsStmt *ctas = makeNode(CreateTableAsStmt);
        ctas->query = (Node *)n;
        ctas->into = intoC;
        ctas->relkind = OBJECT_TABLE;
        ctas->is_select_into = false;

        ParamListInfo params = NULL;


        //Parse analyze query to convert into T_Query type node
        Query *q = parse_analyze((Node*)ctas->query,"",NULL,NULL);
        ctas->query = q;

        // Providing the parameters(structures) to the  ExecCreateTableAs() function to create new subtable
and copy all data to the new subtable.
        ExecCreateTableAs((CreateTableAsStmt *) ctas,"", params,NULL);



        // emptying the base table after data copy to the subtable
```

```
      truncateBase(relToOptimize);

      if(state == 0){  //no subtables present till now
            saveStateInfile(optimizableTableId,0,1);
            //do nothing
            printf("Wrt.Opt. Success table_a created\n");
      }
      else if(state == 1){ //tablename_a present already
            saveStateInfile(optimizableTableId,0,2);
            //drop tablename_a;
            printf("Wrt.Opt. Success table_b created table_a dropped\n");
            dropSubtables(strcat(oldTbl,"_a")); //dropping the subtable after data copy

      }else if(state == 2){ //tablename_b present already
            saveStateInfile(optimizableTableId,0,1);
            printf("Wrt.Opt. Success table_a created table_b dropped\n");

            dropSubtables(strcat(oldTbl,"_b")); //dropping the subtable after data copy
      }

}
//method that accepts relation name and truncates it, required when truncating the base table
truncateBase(char *tablename ){
      RangeVar *rltn3 = makeRangeVar(NULL, tablename, -1);
                                      rltn3->type=T_RangeVar;
                                      rltn3->inhOpt = INH_NO;
                                      rltn3->alias = NULL;

                  TruncateStmt *trunc = makeNode(TruncateStmt);
                              trunc->relations = list_make1(rltn3);
                              trunc->restart_seqs = false;
                              trunc->behavior = DROP_RESTRICT;
                              ExecuteTruncate(trunc);


}




//method that accpets relation name and drops the relation, Required when droping obsolete subtables
dropSubtables(char* tbname)
{
      DropStmt *dropObject = makeNode(DropStmt);
      dropObject->removeType = OBJECT_TABLE;
      dropObject->missing_ok = FALSE;
      dropObject->objects =
      list_make1(list_make1(makeString(tbname)));
      dropObject->arguments = NIL;
      dropObject->behavior = DROP_RESTRICT;
      dropObject->concurrent = false;
      RemoveRelations(dropObject);
}
```

changes at
ExecInsert(TupleTableSlot *slot, TupleTableSlot *planSlot,EState *estate, bool canSetTag)
function

//Storing the updated table state after one insertion.
```
incrRowCount(estate->es_result_relation_info->ri_RelationDesc->rd_id);
```

createas.c

In the function
```
intorel_startup(DestReceiver *self, int operation, TupleDesc typeinfo)
```

inheritedTableIndex is set to relation Oid when calling the CreateAsStatement so that we can check and use that value to mark the new relation that will be created as the child of basetable.

//prepare an inheritance list here if we are doing writeoptimize
```
            if(inheritedTableIndex){/

            //Rangevar structure for the base relation
            RangeVar *rel = makeRangeVar(NULL, NULL, -1);
            rel->relname = get_rel_name(inheritedTableIndex);
            rel->type=T_RangeVar;

            //making a list structure of the basetable RangeVar node
            List *new_list;
            ListCell    *new_head;
            new_head = (ListCell *) palloc(sizeof(*new_head));
            new_head->next = NULL;
            lfirst(new_head) = rel;
            new_list = (List *) palloc(sizeof(*new_list));
            new_list->type = T_List;
            new_list->length = 1;
            new_list->head = new_head;
            new_list->tail = new_head;
            create->inhRelations = new_list;

      }
```

execMain.c

changes at `standard_ExecutorStart(QueryDesc *queryDesc, int eflags)`

```
if(optFlag == 0)
      {
            optFlag = 1;    // flag this to stop re-execution of this function recursively when
ExecCreateAsStmt restarts the executor
            createSubtable(writeOptimize()); // this function starts the write optimize
action
      }
```

// filehandling functions for keeping the states of subtable

```
void saveStateInfile(int relid,int noOfTup,int state){

      FILE *myFile;
         myFile = fopen("tablestate.txt", "r");

    //read file into array
         int numberArray[3];
         int i;

         numberArray[0] = relid;
         if (noOfTup == 0)numberArray[1] = noOfTup;
         else { numberArray[1] += noOfTup;}
         numberArray[2] = state;


         myFile = fopen("tablestate.txt", "w");
              for (i = 0; i < 3; i++)
              {fprintf(myFile,"%d ",numberArray[i]);}
              fclose(myFile);
}
```

// function that increases the rowcount in the state information of the table

```
void incrRowCount(int relid){

      FILE *myFile;
         myFile = fopen("tablestate.txt", "r");

         //read file into array
         int numberArray[3];
         int i;

         if (myFile == NULL)
         {
             return;
         }
         for (i = 0; i < 3; i++)
         {
             fscanf(myFile, "%d,", &numberArray[i] );

         } fclose(myFile);


        numberArray[1] += 1;


              myFile = fopen("tablestate.txt", "w");
              for (i = 0; i < 3; i++)
              {fprintf(myFile,"%d ",numberArray[i]);}
              fclose(myFile);
}
```

// function that returns the state information as a structure after reading it from the file

```
Table_Details writeOptimize(){
        FILE *myFile;
                myFile = fopen("tablestate.txt", "r");

                //read file into array
                int numberArray[3];
                int i;

                if (myFile == NULL)
                {

                Table_Details t1;
                           t1.cnt = 0;
                           t1.relId = 0;
                           t1.state = 0;

                          return t1;

                    //printf("Error Reading File\n");
                    //exit (0);
                }
                for (i = 0; i < 3; i++)
                {
                    fscanf(myFile, "%d,", &numberArray[i] );

                } fclose(myFile);
                Table_Details t2;
                t2.cnt = numberArray[1];
                t2.relId = numberArray[0];
                t2.state = numberArray[2];

               return t2;
}
```

<u>FUTURE WORK:</u>

1 – Currently the writeoptimize table state is kept for only one relation at a time and is kept on a file, we would like to implement it at information_schema

2 – We are not decrementing the row count if any delete happens in between inserts. This results in an early “write optimize operation”.

3 – We are assuming that there are no intermediate reads between “copy basetable” and “truncate basetable” . We need to do these two operation atomically.