Final Project Report

Topic: Temporal Data Support for PostgreSQL

Team: 1. Sriram Yenamandra 16D070017
2. Deep Karkhanis 160100024
3. Pranav Rao 160100021
4. Gaurav Didwania 160050020

Test Runs: Please find the file 'sample-output.txt' in the submission

Source Code: https://github.com/pranavrao870/database_postgres

Functional specifications:

1. We have added a new column in the pg_attribute which denotes whether a column defined as tsrange in the relation is a **valid_time** column or not.(Checked by a variable **is_temporal**)
2. Only one attribute of type valid_time is allowed in a relation. This attribute in a relation is used to indicate the 'valid_time' of each tuple in the relation.
3. A relation is considered temporal if it has one attribute of type valid_time.
4. For creating a new temporal table, the user has to include a field of type valid_time in the 'create table' clause.
5. The user is allowed to drop a column of type valid_time. Then the relation will no longer be temporal.
6. The valid_time attribute is not allowed to be null. In case the user does not specify any defaults, the default of (,)::tsrange is used, denoting logical -inf to inf. This logical mapping of (,)::tsrange to (-inf,inf) was already implemented, we added the default value.
7. One can use the predicates **precedes, succeeds, intersects, contains and overlaps**. These are already implemented for tsrange in postgreSQL. The syntax for these follow from there.
8. Support for joins (natural and theta join) between relations of any type (temporal/non-temporal).

Implementation Semantics
Join semantics

Temporal relation with non-temporal relation

The natural join would be similar to the existing join. The outer joins will result in temporal relations but if the valid-time is required to be NULL, it will instead be negative to positive infinity.

Both relations are temporal

Here, the result can be thought of as the result obtained by (although note that our implementation may be different):

1. Performing a join by considering (or, projecting on) all the attributes except valid-time.
2. Filtering the tuples of join which have a non-empty intersection of the 'valid-time' attributes.
3. The result of join will consist of all the attributes in both the relations and finally a new column containing the intersection of all the temporal columns of the result.

Changes to existing codebase and implementation

DDL commands

Changes in gram.y

```
            |   ColId VALIDTIME TimeDefault
                {
                    ColumnDef *n = makeNode(ColumnDef);
                    char* type = "tsrange";
                    n->colname = $1;
                    n->typeName = makeTypeName(type);
                    ...
                    n->is_not_null = true;
                    n->is_valid_time = true;
                    SplitColQualList($3, &n->constraints, &n->collClause,
                                     yyscanner);
                    n->location = @1;
                    $$ = (Node *)n;
                }
        ;

TimeDefault:    DEFAULT b_expr
                {
                    Constraint *n = makeNode(Constraint);
                    n->contype = CONSTR_DEFAULT;
                    ...
                    $$ = lappend(NIL, (Node*)n);
                }
                | /*empty*/
                {
                    char* d = "(,)";
                    Constraint *n = makeNode(Constraint);
                    ...
                    n->cooked_expr = NULL;
```

```
                    $$ = lappend(NIL, (Node*)n);
                }
```

Changes in pg_attribute.h

```
    /* Checks if attribute is temporal or not */
    bool        attistemporal BKI_DEFAULT(f);
```

(corresponding changes to makefuncs.c, outfuncs.c, copyfuncs.c and equalfuncs.c were made)

Changes in parsenode.h

```
typedef struct ColumnDef
{
    ...
    bool        is_valid_time;  /* is the ts range given to be valid time */
    ...
} ColumnDef;
```

Changes in tupdesc.c

```
        att->attistemporal = false;

        if(entry->is_valid_time){
            if(istemporal){
                // Error: A validtime attribute already exists
                elog(ERROR,
                    "CREATE TABLE: Only one validtime attribute allowed per
relation");
            }
            else{
                att->attistemporal = true;
                istemporal = true;
            }
        }
```

Temporal_Join
In tcop/postgres.c

```
void
getTemporalAttrHelper(Node *rte, Query *query, List **var_list)
{
    int natts, varattno;
    Var * var;
    TupleDesc tupledesc;
    Relation    rel;
    RangeTblEntry * rtentry;

    if(IsA(rte, RangeTblRef)){
        RangeTblRef * rteref = castNode(RangeTblRef, rte);
        rtentry = castNode(RangeTblEntry, list_nth(query->rtable,
rteref->rtindex - 1));
        if(rtentry->rtekind == RTE_RELATION){
            rel = relation_open(rtentry->relid, AccessShareLock);
            tupledesc = rel->rd_att;
            natts = rel->rd_att->natts;
            for (varattno = 0; varattno < natts; varattno++)
            {
                Form_pg_attribute attr = TupleDescAttr(tupledesc, varattno);
                if(attr->attistemporal){
                    var = makeVar(rteref->rtindex, attr->attnum,
attr->atttypid, attr->atttypmod,
                            attr->attcollation, 0);
                    *var_list = lappend(*var_list, var);
                }
            }
            relation_close(rel, AccessShareLock);
            return;
        }
        else if (rtentry->rtekind == RTE_SUBQUERY){

            List * ret_var_list = handle_temporal_helper(rtentry->subquery,
rteref->rtindex);
            ListCell * ret_var_cell;
            foreach(ret_var_cell, ret_var_list){
                *var_list = lappend(*var_list, lfirst(ret_var_cell));
```

```
            }

            return;
        }
        else if (rtentry->rtekind == RTE_CTE){
            int position = find_cte(query->cteList, rtentry->ctename);
            CommonTableExpr * ctentry = castNode(CommonTableExpr,
list_nth(query->cteList, position));
            List * ret_var_list = handle_temporal_helper((Query*)
ctentry->ctequery, rteref->rtindex);
            ListCell * ret_var_cell;
            foreach(ret_var_cell, ret_var_list){
                *var_list = lappend(*var_list, lfirst(ret_var_cell));
            }
            return;
        }
    }
    else if(IsA(rte, JoinExpr)){
        JoinExpr * rteref = (JoinExpr *) rte;
        getTemporalAttrHelper(rteref->larg, query, var_list);
        getTemporalAttrHelper(rteref->rarg, query, var_list);

        if(rteref->isNatural){
            removeTempCols(rteref->quals, *var_list, query->rtable);
        }

        return;
    }
}

List *
handle_temporal_helper(Query * query, int index)
{
    FromExpr * jointree;
    List * fromlist;
    List * var_list;
    List * ret_var_list;
    List * target_list;
    ListCell * rte;
    ListCell * target_list_cell;
    ListCell * var_cell;
    ListCell * var_item;
    List * args_list;
   List * args;
```

```
FuncExpr* isempty;
BoolExpr* isnotempty;
 int target_num;

Node * final_opexpr = (Node*) makeNode(OpExpr);

 int var_i = 0;
 var_list = NIL;
 ret_var_list = NIL;
 if(query == NULL)
     return NULL;
 jointree = query->jointree;
 if(jointree == NULL){
     return NULL;
 }
 fromlist = jointree->fromlist;

 foreach(rte, fromlist){
     getTemporalAttrHelper(lfirst(rte), query, &var_list);
 }

 /* Add to the quals in the fromlist here */
 foreach(var_item, var_list){
     var_i++;
     if(var_i == 1){
         final_opexpr = (Node *) lfirst_node(Var, var_item);
     }
     else{
         OpExpr  *op = makeNode(OpExpr);
         op->opno = 3900;
         op->opfuncid = 3868;
         op->opresulttype = 3908;
         op->opretset = false;
         op->opcollid = 0;
         op->inputcollid = 0;
         op->location = -1;
         args_list = NIL;
         if(var_i == 2)
             args_list = lappend(args_list, castNode(Var, final_opexpr));
         else
             args_list = lappend(args_list, castNode(OpExpr, final_opexpr));
         args_list = lappend(args_list, lfirst_node(Var, var_item));
         op->args= args_list;
         final_opexpr = (Node *) op;
```

```
        }
    }
    if(var_i > 1) {
        args = NIL;
        args = lappend(args, final_opexpr);
        isempty = makeFuncExpr(3850, 16, args,
                   0, 0, COERCE_EXPLICIT_CALL);
        args = NIL;
        args = lappend(args, isempty);
        isnotempty = (BoolExpr *) makeBoolExpr(NOT_EXPR, args, -1);
        args = NIL;
        if(query->jointree->quals == NULL){
            query->jointree->quals = (Node *) isnotempty;
        }
        else{
            args = lappend(args, isnotempty);
            args = lappend(args, query->jointree->quals);
            query->jointree->quals = (Node *) makeBoolExpr(AND_EXPR, args, -1);
        }
    }

    /* Now seeing the target list and returning the appropriate temporal cols
man */
    if(index == -1){
        return var_list;
    }


    target_list = query->targetList;
    foreach(var_cell, var_list){
        target_num = 1;
        Var * ret_var = (Var *) copyObjectImpl((void *) lfirst_node(Var
,var_cell));
        foreach(target_list_cell, target_list){
            TargetEntry * target_var = lfirst(target_list_cell);
            Var * expr = (Var *) target_var->expr;
            if((expr->varno == ret_var->varno) && (expr->varattno ==
ret_var->varattno)){
                ret_var->varno = index;
                ret_var->varattno = target_num;
                ret_var_list = lappend(ret_var_list, ret_var);
            }
            else if(expr->varno <= (query->rtable)->length){
                RangeTblEntry * rtentry = castNode(RangeTblEntry,
```

```
list_nth(query->rtable, expr->varno - 1));
                List * joinaliasvars = rtentry->joinaliasvars;
                if(rtentry->rtekind != 2 ){
                    target_num ++;
                    continue;
                }
                else if(expr->varattno <= joinaliasvars->length){
                    Var * varit = (Var *)
copyObjectImpl((void*)list_nth(joinaliasvars, expr->varattno - 1));
                    if((varit->varno == ret_var->varno) && (varit->varattno ==
ret_var->varattno)){
                        ret_var->varno = index;
                        ret_var->varattno = target_num;
                        ret_var_list = lappend(ret_var_list, ret_var);
                    }
                }
            }
            target_num ++ ;
        }
    }
    return ret_var_list;
}


// Entry point for handling temporal joins
List *
handle_temporal_joins(List *querytrees)
{
    ListCell   *var_item;
    List       *var_list;
    List       *args_list;
    List       *modified_query_list = NIL;
    ListCell   *query_list;

    foreach(query_list, querytrees)
    {
        Node       *final_opexpr = (Node *) makeNode(OpExpr);
        TargetEntry * tentry;
        int var_i = 0;
        Query   *query = lfirst_node(Query, query_list);
        var_list = handle_temporal_helper(query, -1);
        if(query->commandType == CMD_SELECT && var_list->length != 0){
            foreach(var_item, var_list){
                var_i++;
```

```
                if(var_i == 1){
                    final_opexpr = (Node *) lfirst_node(Var, var_item);
                }
                else{
                    OpExpr  *op = makeNode(OpExpr);
                    op->opno = 3900;
                    op->opfuncid = 3868;
                    op->opresulttype = 3908;
                    op->opretset = false;
                    op->opcollid = 0;
                    op->inputcollid = 0;
                    op->location = -1;
                    args_list = NIL;
                    if(var_i == 2)
                        args_list = lappend(args_list, castNode(Var,
final_opexpr));
                    else
                        args_list = lappend(args_list, castNode(OpExpr,
final_opexpr));
                    args_list = lappend(args_list, lfirst_node(Var, var_item));
                    op->args= args_list;
                    final_opexpr = (Node *) op;
                }
            }
            tentry = (TargetEntry*)makeNode(TargetEntry);
            tentry->expr = (Expr*)final_opexpr;
            tentry->resname = "Intersection";
            tentry->ressortgroupref = 0;
            tentry->resorigtbl = 0;
            tentry->resorigcol = 0;
            tentry->resjunk = false;
            tentry->resno = query->targetList->length + 1;
            lappend(query->targetList, tentry);
        }
        modified_query_list = lappend(modified_query_list, query);
    }
    return modified_query_list;
}
```

In case of natural joins, handle same validtime attribute names (remove equality condition)

```
void
```

```
removeTempCols(Node * quals, List * var_list, List * rtable)
{
    if(quals == NULL){
        return;
    }
    else if(IsA(quals, Var)){
        return;
    }
    else if(IsA(quals, OpExpr)){
        OpExpr * op = (OpExpr *)quals;
        if(op->opno == 3882 && op->opfuncid == 3855){
            List * args = op->args;
            if(args->length == 2){
                if(IsA(linitial(args), Var) && IsA(lsecond(args), Var)){
                    Var * left = (Var *) linitial(args);
                    Var * right = (Var *) lsecond(args);
                    int lfound = 0;
                    int rfound = 0;
                    ListCell * var_item;
                    foreach(var_item, var_list){
                        Var * var = lfirst_node(Var, var_item);
                        if(var->varno == left->varno && var->varattno &&
left->varattno){
                            lfound = 1;
                        }
                        if(var->varno == right->varno && var->varattno &&
right->varattno){
                            rfound = 1;
                        }
                        if(lfound == 1 && rfound == 1){
                            *right = *left;
                            return;
                        }
                    }
                    if(lfound == 0){
                        if(left->varno <= rtable->length){
                            RangeTblEntry * rtentry = castNode(RangeTblEntry,
list_nth(rtable, left->varno - 1));
                            List * joinaliasvars = rtentry->joinaliasvars;
                            if(left->varattno <= joinaliasvars->length &&
rtentry->rtekind == 2){
                                Var * varit = castNode(Var,
list_nth(joinaliasvars, left->varattno - 1));
                                foreach(var_item, joinaliasvars){
```

```
                                    Var * var = lfirst_node(Var, var_item);
                                    if(var->varno == varit->varno &&
var->varattno && varit->varattno){
                                        lfound = 1;
                                    }
                                }
                            }
                        }

                        if(rfound == 0){
                            if(right->varno <= rtable->length){
                                RangeTblEntry * rtentry = castNode(RangeTblEntry,
list_nth(rtable, right->varno - 1));
                                List * joinaliasvars = rtentry->joinaliasvars;
                                if(right->varattno <= joinaliasvars->length &&
rtentry->rtekind == 2){
                                    Var * varit = castNode(Var,
list_nth(joinaliasvars, right->varattno - 1));
                                    foreach(var_item, joinaliasvars){
                                        Var * var = lfirst_node(Var, var_item);
                                        if(var->varno == varit->varno &&
var->varattno && varit->varattno){
                                            rfound = 1;
                                        }
                                    }
                                }

                            }
                        }
                        if(lfound == 1 && rfound == 1){
                            *right = *left;
                            return;
                        }
                    }
                }
            }

    }
    else if(IsA(quals, BoolExpr)){
        BoolExpr * boolExpr = (BoolExpr *) quals;
        ListCell * item;
        foreach(item, boolExpr->args){
            Node * q = lfirst(item);
```

```
            removeTempCols(q, var_list, rtable);
        }
    }
    return;
}
```

The way the join is implemented is that we have added the condition of non-null intersection of the temporal attributes of the columns. This predicate is added after rewriting the query.

Points

One way to do the temporal join is to remove the original valid time attributes and keep only the intersection. But this will create problems, as, if the user sends the query like “select time1, time2 from (r1 join r2)” ,where time1, time2 are valid time attributes of the r1 and r2, then the analyzer will proceed with the query, giving no error, but if we remove the attribute time1 and time2 after analyze and rewrite, then we will be in a fix. So we have kept all the valid time columns and then shown the intersection at the end.

While performing natural join of temporal relations, if the validtime attribute names are same, then by default, these attributes would be compared for equality. We have removed these predicates so that it works correctly.This has been done in removeTempCols()

We have handled temporal joins in all cases and combination involving subqueries and ‘with’ clause.
However, for outer joins, since we have added the not(isempty(....intersection of temporal attributes....)) condition in the quals list of FromExpr (query->jointree), this condition is evaluated after the outer join is executed. Hence our outer join does not always produce the desired result.