Introduction

The number of applications using RDBMS has grown rapidly over the past few decades, since its conception by E.F Codd in 1974, touching all domains like Inventories, Supply chain management, Hospitals, Schools, Libraries, etc where data and historical data are important. Despite this popularity, there is another type of application that has not been able to take advantage of this system. This type of application requires temporal support from the database system.

**PostgreSQL** is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.

A temporal table is a table that records the period of time when a row is valid. This validity is checked by looking at two automatically generated columns "From" and "Upto". "From" gives the time from when a record is valid and "Upto" the time it becomes invalid. There are two types of periods:

1. The application period (also known as valid-time or business-time)
2. The system period (also known as transaction-time).

System-period data versioning allows you to specify that old rows are archived into another table (that is called the history table).
Problem Statement

Providing Temporal Support in postgreSQL

1. We have to extend create table statement as follows:

   CREATE TEMPORAL TABLE <table name> (column list)

   For this we need to extend the processing of "CREATE TABLE" statement by automatically including two new columns called "FROM" and "UPTO", both of timestamp type.

2. We have to extend the update statements as follows:

   - **Insert statements**: It should initialize FROM with current system time and keep UPTO as very large value (something like 'forever' time)
   - **Delete statement**: Create a trigger that does not delete the tuple(s) but initializes the UPTO value to the system time
   - **Update statement**: Create triggers to add UPTO values in the before−tuple and add FROM time into the after−tuple (thus producing two tuples in place of just one for every modified tuple)

Approach

The approach we took to solve this problem is rather straight forward in which we traced the code to understand the flow of program execution by using two software tools:

- **CSCOPE**: A Developer's tool for browsing source code.
- **ECLIPSE**: An Integrated development environment (IDE) used in Computer programming for both program executions and debugging.
Solution Details

To implement the functionality of TEMPORAL TABLES we have changed following files:

- **gram.y**: Converts SQL query to query tree. The SQL is lexically analyzed into keywords, identifiers, and constants, and passed to the parser. The parser creates command-specific structures to hold the elements of the query.

- **postgres.c**: This is defined under Traffic cop (tcop), which dispatches request to proper module. This contains the postgres backend main handler, as well as the code that makes calls to the parser, optimizer, executor, and commands functions.

- **kwlist.h**: It contains list of all reserved and unreserved keywords.

To modify **CREATE** statement:

First we understood the rules of how the "CreateStmt" has been defined, in the gram.y file and based on this we defined the rules for our create statement is used to create TEMPORAL tables.

We introduced the keyword **TEMPORAL** in our new rule and accordingly defined it in %token <keyword > section. This new keyword have also been defined in the kwlist.h file which contains all the reserved and unreserved keywords.

While creating TEMPORAL table, we wrote Triggers which will get append with user input query and creates trigger function for update and delete for that particular table.

To modify **INSERT** statement:

We have implemented INSERT statement in two ways

- We modified the original INSERT statement such that it will insert default values for "_from " and "_upto " columns in a temporal table. But if the table is not of TEMPORAL type then INSERT executes normally.

- The second way is that we allow the user to write query as:

  INSERT INTO TEMPORAL TABLE <tablename> VALUES(.....);
Here our code automatically inserts values for “_from” column with the `current_timestamp` and “_upto” column with a `very large timestamp` in this case, we are assigning `9999-12-31 23.59.59`.

To modify **UPDATE** statement:

For temporal table, update statement should update key row with new data but keep the previous data as it is with setting _upto column as current_timestamp and add new row with _from set as current_timestamp and _upto will get assign maximum value as timestamp.

To modify **DELETE** statement:

At the time of creating TEMPORAL table, we fire trigger which wont deletes specified row physically from table but sets its _upto value as current_timestamp.

**Test Scenarios**

- Validate the **Creation** of a Temporal Table in which two columns “_from” and “_upto” will be automatically created of timestamp type.

- Validate the **Insertion** of records into the Temporal Table where the “_from” and “_upto” attributes will be filled automatically by the system.

- Validate the **Updation** of records into the Temporal Table.

- Validate the **Deletion** of records into the Temporal Table where the records will not be deleted instead they will be made invalid by changing the “_upto” value to the current time.

**Test Data**

**Creation:**

```sql
CREATE TEMPORAL TABLE Student
(Name varchar(50),
Department varchar(50),
Hostel_Number int);
```
Insertion:

    INSERT INTO TEMPORAL TABLE Student
    VALUES ('Ashish','Computer Science',15);

    INSERT INTO TABLE Student
    VALUES ('Rahul','Physics',1);

Updation:

    UPDATE Student SET Hostel_Number = 4
    WHERE Hostel_Number = 15;

Deletion:

    DELETE FROM Student WHERE Hostel_Number = 1 ;

Pseudo Code

Pseudo-code of routines added in Postgres.c are:

Procedure Parse(str* query_string):
1) IF First token == "CREATE"
   1.1) IF Second token == "TEMPORAL"
      1.1.1) Break the input query into tokens and extract the relation name and column name fields.
      1.1.2) After getting relation name and fields from input query, create the Trigger Function for DELETE and UPDATE Query.
      1.1.3) Write Triggers for DELETE and UPDATE.
      1.1.4) Append the Trigger Functions and Triggers to input query.
      1.1.5) Pass this generated query to execution engine for execution.
Changes Made in Postgres - W2H (What, Where and How)

For Create Temporal Table statement

At Line No: 2720 in src/backend/parser/gram.y

CreateStmt:

CREATE TEMPORAL OptTemp TABLE qualified_name ' ( ' OptTableElementList ' ) ' OptInherit OptWith OnCommitOption OptTableSpace
{
  CreateStmt *n = makeNode(CreateStmt);
  TypeName *y_tmp=SystemTypeName("timestamp");
  Node *n_tmp; List * tmp;
  $5->relpersistence = $3;
  n->relation = $5;

  n_tmp=(Node *) makeFuncCall(SystemFuncName("now"), NIL, @6);
  tmp=lappend($7, myColumnDef("_from",@7,n_tmp));

  n_tmp=makeStringConstCast("9999-12-31 23:59:59", @6, y_tmp);
  tmp=lappend(tmp, myColumnDef("_upto",@7,n_tmp));

  n->tableElts = $7;
  n->inhRelations = $9;
  n->constraints = NIL;
  n->options = $10;
  n->oncommit = $11;
  n->tablespacename = $12;
  n->if_not_exists = false;
  $$ = (Node *)n;
}

At Line No: 13233 in src/backend/parser/gram.y

reserved_keyword:

    .. | TEMPORAL | ..
Node *myColumnDef(char *x, int loc, Node * cons) {
    ColumnDef *n = makeNode(ColumnDef);
    char *tmpStr = (char*)malloc(sizeof(char)*100);
    Node *tmp;
    Constraint *n_c = makeNode(Constraint);

    n->colname = strcpy(tmpStr, x);
    n->typeName = SystemTypeName("timestamp");
    n->inhcount = 0;
    n->is_local = true;
    n->is_not_null = false;
    n->is_from_type = false;
    n->storage = 0;
    n->raw_default = NULL;
    n->cooked_default = NULL;
    n->collOid = InvalidOid;
    n->fdwoptions = NIL;

    n_c->contype = CONSTR_DEFAULT;
    n_c->location = loc;
    n_c->raw_expr = cons;
    n_c->cooked_expr = NULL;
    tmp = (Node *)n_c;

    n->constraints = lappend(NIL, tmp);

    n->location = loc;

    return (Node *)n;
}
At Line No: **595** in src/backend/parser/gram.y

Keyword: TEMPORAL

At Line No: **369** in src/backend/include/kwlist.h

```
PG_KEYWORD("temporal", TEMPORAL, RESERVED_KEYWORD)
```

**For Insert Statement**

At Line No: **9086** in src/backend/parser/gram.y

```
InsertStmt:
  opt_with_clause INSERT INTO TEMPORAL qualified_name insert_rest returning_clause
  {
    List *tmp = ((SelectStmt*)$6->selectStmt)->valuesLists;
    Node *x = (Node *) makeFuncCall(SystemFuncName("now"), NIL, @6);
    TypeName *y_tmp=SystemTypeName("timestamp");
    Node *y = makeStringConstCast("9999-12-31 23:59:59", @6, y_tmp);
    y_tmp->location = @6;
    tmp=(List*)tmp->head->data.ptr_value;
    tmp=lappend(tmp, x);
    tmp=lappend(tmp, y);
    $6->relation = $5;
    $6->returningList = $7;
    $6->withClause = $1;
    $$ = (Node *) $6;
  }
```

For Update Statement

Created Trigger as follows:

```c
void parse(char *orig_query){

    char * my_query = (char*)malloc(sizeof(char)*1000);
    char * delete_trigger = (char*)malloc(sizeof(char)*1000);
    char * delete_trigger_table = (char*)malloc(sizeof(char)*1000);
    char * update_trigger = (char*)malloc(sizeof(char)*1000);
    char * update_trigger_table = (char*)malloc(sizeof(char)*1000);
    char * col_tmp = (char*)malloc(sizeof(char)*100);
    char * qtmp = (char*)malloc(sizeof(char)*1000);
    char * query,* search, *token;

    strcpy(my_query,orig_query);
    query=my_query;

    search=" \n\t();";
    token = getToken(query,search);

    if(strcmp(token,"create")==0){
        token = getToken(NULL,search);
        if(strcmp(token,"temporal")==0){
            do{
                token = getToken(NULL,search);
            }while(strcmp(token,"table"));

            token = getToken(NULL,search);
            sprintf(delete_trigger,"CREATE OR REPLACE FUNCTION %s_delete_fn()",token);
            sprintf(delete_trigger,"%s RETURNS trigger AS $$ BEGIN IF OLD._upto < now() THEN
                RETURN NULL;
            ELSE INSERT INTO %s VALUES(",delete_trigger,token);

            sprintf(delete_trigger_table,"CREATE TRIGGER %s_delte_ BEFORE DELETE
            ON %s FOR EACH ROW EXECUTE PROCEDURE %s_delete_fn()");",token,token,token);
```

sprintf(update_trigger,"CREATE OR REPLACE FUNCTION %s_before_update_fn()",token);
sprintf(update_trigger,"%s RETURNS TRIGGER AS $$ BEGIN IF OLD._upto > now() THEN NEW._from = now(); INSERT INTO %s VALUES("$$语言plpgsql; %s",delete_trigger,delete_trigger_table);

query=getToken(NULL,"");

if(*query=='(')query++;
search="",";

token = getToken(query,search);
int end=1;
do{
    strcpy(col_tmp,token);
    token=getToken(NULL,"");
}

if(token)
    strcpy(qtmp,token);
else {free(qtmp),qtmp=NULL;end=0;}

if(token)
    strcpy(qtmp,token);
else {free(qtmp),qtmp=NULL;end=0;}
token=getToken(col_tmp," ");

sprintf(delete_trigger,"%s OLD.%s,"delete_trigger,token);

}while(end && token);

sprintf(delete_trigger,"%s OLD._from, now()); RETURN OLD;
END IF;
END; $$ language plpgsql; %s",delete_trigger,delete_trigger_table);

sprintf(update_trigger,"%s OLD._from, now()); RETURN NEW;
ELSE RETURN NULL;
END IF;
END; $$ language 'plpgsql'; %s",update_trigger,update_trigger_table);

exec_simple_query(delete_trigger);
exec_simple_query(update_trigger);
}
}

free(my_query);
free(delete_trigger);
free(delete_trigger_table);
free(update_trigger);
free(update_trigger_table);
free(col_tmp);
free(qtmp);
}
Implementation Details

1. Create Statement

For implementing "CREATE TEMPORAL TABLE" statement in postgresQL system, we updated the "src/backend/parser/gram.y" file as mentioned above. After tracking down how the normal "CREATE TABLE" statement was working and figuring out how the columns were being added to the table. We added in our own code to add two new columns, as column structure "ColumnDef".

This code was added to gram.y as we wanted to add columns in the parse-tree formed, so that the statement could execute, creating a temporal table.

2. Insert Statement

For implementing "INSERT", we came across two strategies, as described below:

(A) Strategy I: In this, we created a new "INSERT" statement, named "INSERT INTO TEMPORAL", where the table name specified have to be a temporal table for this to work properly.

For this we had to add an extra rule in the gram.y file so that postgresQL can parse our new statement.

(B) Strategy II: While implementing the different functionalities to support temporal support in postgresQL, we thought that it was much user-friendly if we can support the original "INSERT" statement with our added temporal table facility.

So we came to the conclusion that we should add some "DEFAULT" values while creating the table. For this, while creating the table with "CREATE TEMPORAL TABLE" statement, we again changed the parse tree slightly so as to add "DEFAULT" constraints to "FROM" and "UPTO" fields. So that, while inserting, user don't have to give values for these fields. And this strategy was intuitive as these two columns initially were taking the same value while creating a new row every time.

3. Delete Statement

For implementing "DELETE", we dynamically created a trigger while the table was being created using the "CREATE TEMPORAL TABLE" query.

This code, as shown above, was added to postgres.c file. Whenever a "CREATE TEMPORAL TABLE" query was generated by user, we checked whether it is such a query or not and if
it is we defined a trigger and a function for that trigger for deleting rows using the query given by the user. This is done by our "parse()" function.

4. Update Statement

For implementing "UPDATE" statement, we used the same implementation technique as followed in "DELETE" statement.
Result - Screen Shots

On small data set:

```
psql (9.4.4)
Type "help" for help.

test=# CREATE TEMPORAL TABLE Student
     ( Name varchar(50),
     Department varchar(50),
     Hostel_Number int);
CREATE TRIGGER
```

**Figure 1:** Create Query

```
test=# INSERT INTO TEMPORAL Student
     (test=# VALUES ('Ashish','Computer Science',15);
 INSERT 0 1

 test=# INSERT INTO Student
     (test=# VALUES ('Bahal','Physics',1);
 INSERT 0 1

 test=# SELECT * FROM Student;

<table>
<thead>
<tr>
<th>name</th>
<th>department</th>
<th>hostel_number</th>
<th>from</th>
<th>upto</th>
</tr>
</thead>
</table>
(2 rows)
```

**Figure 2:** Insert Query
On Large Data Set:

**Figure 3:** Delete Query

```sql
DELETE FROM Student WHERE Hostel_Number = 1;
```

**Figure 4:** Update Query

```sql
UPDATE Student SET Hostel_Number = 4
WHERE Hostel_Number = 15;
```

**Figure 5:** Create Query

```sql
CREATE TEMPORAL TABLE Students

CREATE TRIGGER
```

(0 rows)
Conclusions

Our project provided temporal table support in PostgreSQL. In this, we managed to change the INSERT, UPDATE and DELETE commands to work on temporal tables. And these temporal tables can be created, by using a new command called CREATE TEMPORAL TABLE.

In our project, we made changes to the source code of PostgreSQL version 9.4.4. And added about few 100s of LOC to the source. Basically, the main changes were made to files "gram.y" and "postgres.c".

Figure 6: Insert Query