CS631 - Implementation Techniques for Relational Database Systems
(Project Report)

Stream Processing in PostgreSQL

Nadesh Seen(21Q050003) Pritam Sil (21Q05R001) Saurish Darodkar (213050046)

November 29, 2022

1 Introduction

PostgreSQL or Postgre as it is known is an open source relational database management system(RDBMS). Currently Postgre does not support stream processing queries. Streaming data refers to data that arrives in continuous fashion. There are many applications that has streaming data like stock market, e-commerce site, sensor readings etc. Queries on these streams can be very useful for monitoring, alerts and automated triggering of actions.

This project aims to introduce the stream processing in PostgreSQL. We have used approach similar to windowing, in which streams is broken up into windows and queries run on windows. The system will create an UNLOGGED table named `tmp_table` whenever the server is started. It will serve as a temporary table for storing metadata from insert queries. Whenever the user executes an insert query, the system will automatically push the values in it into the `tmp_table` as per the format `(relation_name,attribute_name, attribute_values)`. Whenever the user executes a stream aggregate query, the system will automatically calculate the aggregate from the `tmp_table` and display the output.

In the earlier system for each insert, aggregate results are calculated as it arrives. Five aggregates are calculated i.e. sum, min, max, avg and count. The results are stored in a temporary relation that resides in-memory. The problem here is that this currently works on only a table named "student" with column "marks" and "id" We propose the following solution to this problem.

Goal

- Implement stream inserts at the execution level and make the current system more generic. For any insert query, postgres will automatically update the temp table and reduce the time taken to calculate aggregation queries.

- Modify existing stream queries at the parsing level to support the above changes.

2 Prior Work

In this section, we describe how prior work was implemented. They have implemented a stream processing system which processes the input stream of insert queries for a given table and calculates the aggregate values of all attributes of the given table for each insert query. The aggregate values are sum, max, min, count and average. All of these values are updated in the in-memory temporary relation.

Firstly, query is checked if it is a stream query based on the "STREAM" keyword present in the beginning of the query. If it is a stream query then, get the values of the attributes for the insert query using the PostgreSQL parser. Raw parse tree is constructed then using this query tree is constructed. Using the query tree's target list and values list, column names and the values of the insert query are retrieved and then these are stored in memory to compute the aggregates for the table in a C structure. Then the current aggregate values for all the attributes are retrieved from an in-memory temporary relation and updated using the values obtained via the parser which is stored in the C structure.

Instead of using a query such as select SUM(marks) from student; The client can simply call `stream_sum` and it will return the SUM of marks. Similarly for `stream_max`, `stream_min`, `stream_avg` and `stream_cnt`, which return the maximum, minimum, average and the count values for the relation.

3 Description

We have implemented a stream processing system in PostgreSQL which is an extension to the prior work. Instead of implementing the entire stream processing system at parsing level we have modified the stream inserts to be done at execution level. We have also made the system such that it can work any relation and attribute name. The earlier system could only work on single relation called 'student' and two attributes called 'id' and 'marks'. Our system stores the integer type attributes with the relation name, attribute name and attribute value in the TEMPORARY/UNLOGGED table. We have tested our system with both TEMPORARY table and UNLOGGED table for caching the rows in-memory.

3.1 Query Paths

In this section, we have noted the path of some queries that we used to understand the execution flow.

- Insert

    1. `exec_simple_query() - postgres.c` : Runs the query which is passed as char*

2. `PortalRun() - pquery.c` : Run a Portal's query
3. `PortalRunMulti() - pquery.c` : Execute a portal's queries which are multi queries or non-SELECT like queries
4. `ProcessQuery() - pquery.c` : Execute a single plannable query
5. `ExecutorRun() - execMain.c` : Executes the query plan
6. `standard_ExecutorRun() - execMain.c` : Executes the query plan
7. `ExecutePlan() - execMain.c` : Processes the query plan until we have retrieved 'numberTuples' tuples, moving in the specified direction.
8. `ExecProcNode() - executor.h` : Execute the given node to return another tuple.
9. `ExecProcNodeFirst() calls ExecModifyTable() - execProcnode.c` : Performs some one-time checks, before calling the relevant node method
10. `ExecModifyTable() - nodeModifyTable.c` : Perform table modifications as required, and return RETURNING results if needed.
11. `ExecModifyTable() : ExecInsert() - nodeModifyTable.c` : For executing an insert query
12. `table_tuple_insert() - tableam.h` : Insert a tuple from a slot into table AM routine.
13. `heapam_tuple_insert() - heapam_handler.c` : Insert a physical tuple into the heap AM
14. `heap_insert() - heapam.c` : Insert tuple into a heap

- Select

1. `exec_simple_query() - postgres.c`
2. `PortalRun() - postgres.c`
3. `PortalRunSelect() - pquery.c`
4. `ExecutorRun() - pquery.c`
5. `standard_ExecutorRun() - execMain.c`
6. `standard_ExecutorRun() : debugStartup() - printtup.c` Prints the attributes name
7. `ExecutePlan() - execMain.c`
8. `ExecProcNode() - execMain.c`
9. `ExecProcNodeFirst() calls ExecScan() - execProcnode.c`
10. `ExecScan() - execScan.c` - Returns the qualifying tuple
11. `ExecutePlan() : dest->receiveSlot() - execMain.c`
12. `dest->receiveSlot() calls debugtup() - printtup.c`

- Update

1. Exec_simple_query() - postgres.c
2. PortalRun() - pquery.c
3. PortalRunMulti() - pquery.c
4. ProcessQuery() - pquery.c
5. ExecutorRun() - pquery.c
6. standard_ExecutorRun() - execMain.c
7. ExecutePlan() - execMain.c
8. ExecProcNode() - executor.h
9. ExecProcNodeInstr() - execProcnode.c
10. ExecModifyTable() - nodeModifyTable.c
11. ExecUpdate() - nodeModifyTable.c
12. table_tuple_update() - tableam.h
13. heapam_tuple_update() - heapam_handler.c
14. heap_update() - heapam.c

3.2 Modifications

We have modified `postgres.c`, `heapam_handler.c` files and have used functions from `heaptuple.c` and `printtup.c`. The parsing level code is in `postgres.c` and the execution level changes are done in `heapam_tuple_insert()` function in `heapam_handler.c`. We have studied the following data structures `TupleTableSlot` and `HeapTupleHeader`. These are some of the important data structures that we needed to study to in order to implement stream inserts at execution level. `debugtup()`, `RelationIdGetRelation()`, `CreateTupleDesc()` and `heap_form_tuple()` are some of the important functions that are used in our system.

After creating `tmp_table` we find the id of this relation using a dummy insert and store the id a `tmp_table_id` variable. The `RelationIdGetRelation()` function helps to find a specific relation using the id of that relation. We use this function to find the `tmp_table` relation inside the `heapam_tuple_insert()` function in `heapam_handler.c`. Now, whenever there is an insert on any relation other than the `tmp_table` relation we find which of the attributes are of type int. All the attributes which are of type int are stored in the `tmp_table`.

In `heapam_insert_table()` we have a TupleTableSlot data structures which we use to find the values of the tuple. After we stored the values of the attribute we use `heap_form_tuple()` to form a tuple of `tmp_table`. It takes input as TupleDesc and values we want to store. We get TupleDesc information from the Relation data structure. When we get the tuple in the required format we use the `heap_insert` function to insert the new tuple in the `tmp_table` relation.

3.3 Demo

Initially, when the system starts TEMPORARY/UNLOGGED table is created named `tmp_table`. Whenever a insert query is run it stores the integer attributes with their value, attribute name and relation name. To find sum from the `tmp_table` we use `stream` query. The `stream` query has the following format `stream aggregation_op relation_name attribute_name`.
Example -

1. Instead of using `stream insert into student(1,20);` we can use normal insert and it will store values in the `tmp_table`.
2. We use `insert into student(1,20);` where attribute name are `id` and `marks`.
3. The above query will store the values in TEMPORARY relation as student id 1 and student marks 20
4. To calculate sum, we use another query instead of using `select sum(marks) from student;`
5. `stream sum student marks;` query is used to fetch the values of marks attribute from student relation and then sum is calculated.

3.4 Experiments

The first set of experiments serves as Proof of Work. In it two queries have been executed and their results have been recorded. From Figure1 it can be seen that, the first query which is a simple SELECT query with COUNT aggregation produces the same results as a `stream cnt` query.

```
backend> select count(*) from testing;
         1: count       (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
         1: count = "1000"      (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
```

(a) Result from Normal SELECT query with COUNT

```
backend> stream cnt testing marks;
         1: count       (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
         1: count = "1000"      (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
```

(b) Result from stream count query

Figure 1: Proof of Work

The second set of experiments shows the time taken between two `stream sum` queries. The testing table just has one attribute named marks. Here the number of tuples in the relation are 100000 and value of the attribute marks is 1. One makes use of a TEMPORARY `tmp_table` while another makes

use an UNLOGGED `tmp_table` and it can be seen from Figure 2 that the one with a TEMPORARY table shows the better performance in the first run. After the first run both perform the same taking almost 30 millisecond for execution. In some cases, the normal sum query takes the least time which could be due to the fact we don't use any conditional clause in the query. This could lead to optimization in the normal sum query ultimately giving better performance. Other sum queries have to conditions that they need to find in the `tmp_table` that is the relation name and attribute name.

Note that all these queries are executed on different run and each query is the first query to be executed for a particular table. This ensures that caching does not interfere with the execution time.

```
backend> stream sum testing marks;
         1: sum (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
         1: sum = "100000"      (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
Time taken to execute query: 0.030999
```

(a) Time taken to execute sum on TEMPORARY tmp_table

```
backend> stream sum testing marks;
         1: sum (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
         1: sum = "100000"      (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
Time taken to execute query: 0.064689
```

(b) Time taken to execute sum on UNLOGGED tmp_table

```
backend> select SUM(marks) from testing;
         1: sum (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
Value: 100062    1: sum = "100062"      (typeid = 20, len = 8, typmod = -1, byval = t)
        ----
Time taken to execute query: 0.078482
```

(c) Time taken to execute SELECT with SUM on testing table

Figure 2: Comparisons of sum queries

3.5 Challenges & Learnings

Postgres has a huge codebase to navigate through. Understanding the flow of execution itself took a lot of time. We learned to use the debugger to navigate through this large codebase. Understood how the queries can be modified at execution level. Found out various functions which were used in our implementation to form tuples and print attributes of those tuples.

4 Conclusion & Future Work

The main goal of this project was to introduce stream query processing at execution level. Whenever an insert query is being executed, the data will be stored in a `temp_table` and data from the same will be used for stream queries. The main advantage is that data will be fetched from the main memory instead of any persistent storage device. This will in turn reduce the time taken to execute the whole query.

Future work will include implementing the concept of windows. We can improve the performance by calculating the aggregates and store them in another in-memory table. But for this we need to study update query in more depth so that we can update the in-memory table after the insert command. Here, we need to get the values from the `tmp_table` and then calculate aggregates and update the `tmp_table`.

5 Acknowledgement

A big thanks to Pramod S Rao, Pasi Piyush Singh Umesh Chandra and Manjusree M P for taking initiative and implementing stream query processing at the parser level. Without their work, our project would have taken a lot of time. Their implementation gave us the necessary motivation to execute it at execution level and hence our work.