Project-2 Report

PATH COMPUTATION & TRANSITIVE CLOSURE MAINTENANCE IN DIRECTED GRAPH

By: Khushboo Agarwal &
Maksim Egorov (Team 1)

Introduction:

In this project, we came up with an idea to compute path by using EQU (final table) with two additional columns i.e. Edge (mid-point between I and j) and LW (last weight value added to this particular vertex pair). For path computation, we calculated transitive closures using partitioning procedure. For accelerating the transitive closure calculation process, we decided not to use intermediate tables and make full computation less depending on programming language. In result, our method computes all transitive closures approximately twice faster than the previous implementation.

Approach Used:

1) Transitive Closure Computation: To compute transitive closure, we designed and implemented partition computation. Firstly, we retrieved unique source vertex from Vertica and then executed recursive query for each vertex. All data computed will be stored in HP Vertica table name final Table= EQU. The code to run computation with partition in file main.java same as the code to run recursive query on full table.

2) <u>Path Computation</u>: For path computation, we added following two extra columns in EQU table.

1. Edge – mid-point between source and destination vertex.

2. LW – last weight value added to this particular vertex pair.

To compute path, for certain pair vertex on certain depth level we use only JOINS. For reference, we can refer PathComputation.java for the code. Finally, PATH will be stored in HP Vertica table PATH.

To compute path for given pair sourceVertex, destinationVertex and depth. We need to use command:

java PathComputation sourceVertex, destinationVertex depth

Below you can see query which computes path between source vertex 'i' = 1, destination vertex 'j' = 5 and depth 'd' = 4:

```
select distinct EQU.d,EQU.i,EQU.j,EQU.p,EQU.v,EQU.Edge,R3.Edge,
R2.Edge,R1.Edge from EQU
left join (select * from EQU where d= 3 and i=1) as R3 on EQU.Edge=R3.j and
R3.v=(EQU.v-EQU.LW)
left join (select * from EQU where d= 2 and i=1) as R2 on R3.Edge=R2.j and
R2.v=(EQU.v-EQU.LW-R3.LW)
left join (select * from EQU where d=1 and i=1) as R1 on R2.Edge=R1.j and
R1.v=(EQU.v-EQU.LW-R3.LW-R2.LW)
where  EQU.i = 1 and EQU.j = 5 and EQU.d=4;
```

3) Accelerating Transitive Closure Computation: For accelerating the transitive closure computation, we decided not to use intermediate tables and make full computation less dependent on Java. For code implementation, refer TranClosSpeedUpAttempt1.java file. As per TranClosSpeedUpAttempt1.java file, it is not creating any intermediate tables, instead it is using only initial table.

OBSERVATIONS:

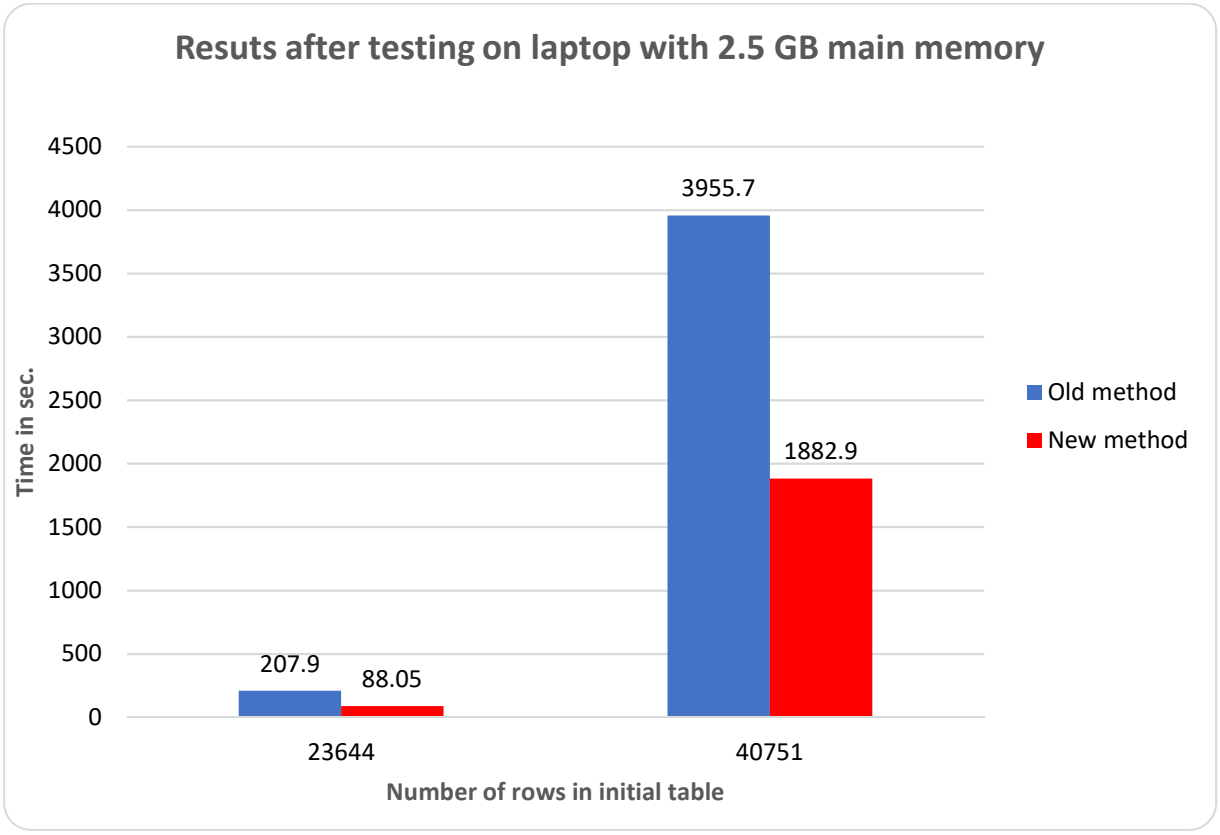
1) Transitive Closure Computation: We came to know later that our assumption that it will reduce execution time proved wrong. However, when we ran our code on server with 8 nodes and the result was negative (full run takes 23 min to compute table EQU, partition run 2 hours). Partition computation process was not speeding up the process to compute transitive closures. Although, if resources limited, for example our personal laptop has only 2.5 GB memory compute recursive query on full facebook table (88, 234 rows), is impossible. In this case, partition run took 8 hours to compute all transitive closures and in output EQU table contains around 2 billion rows.

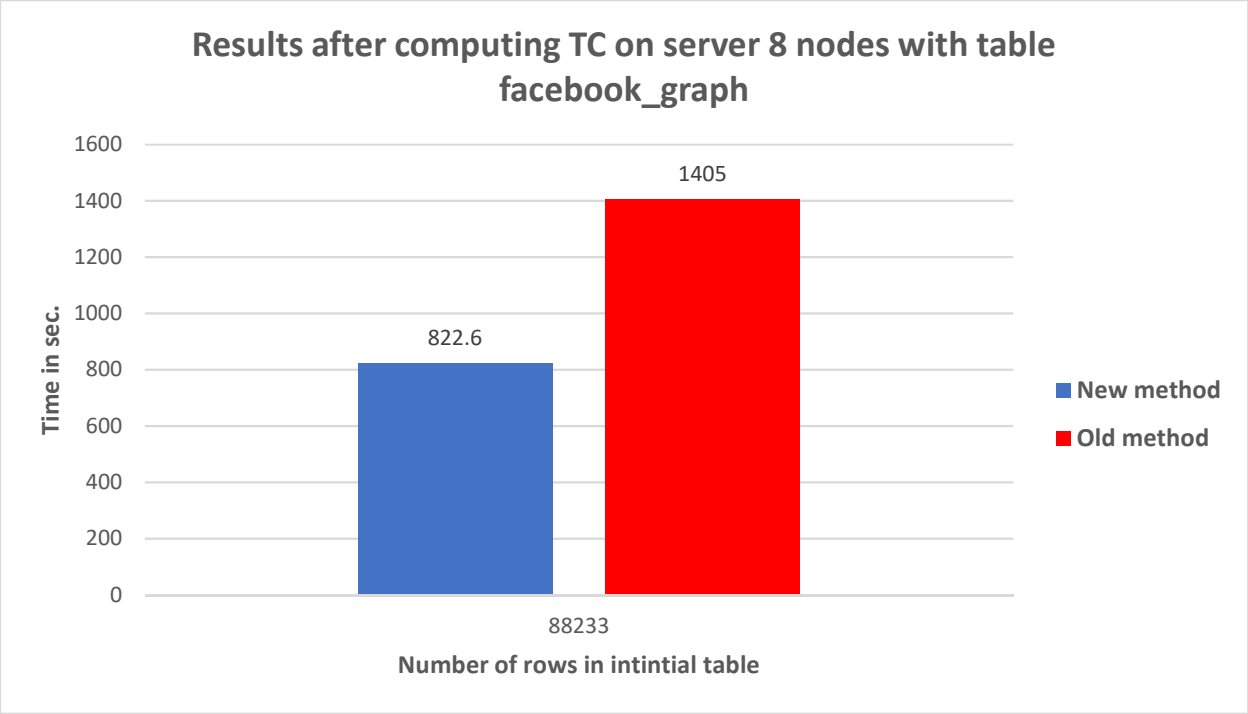
Our method is not reducing execution time but provides possibility to compute EQU table with limited hardware resources.

2) Path Computation: We tested this method on EQU table with 2 billion rows and it takes only 1-2 second to execute and provide path. It is a very efficient way to compute path for certain depth level. It may be modified and pivot results of the query to store paths in vertical form.

3) Accelerating Transitive Closure Computation: Our method computes transitive closures approximately twice faster than previous implementation.

PERFORMANCE ANALYSIS:





FINAL COMMENTS:

1) We are computing path using transitive closures which are stored in EQU table. We tried on small dataset on our laptop and it is calculating all paths between 'sourceVertex' and 'destinationVertex' at any depth level using the following command:

java PathComputation sourceVertex, destinationVertex depth

For example: taking sourceVertex 'i' = 2, destinationVertex 'j' = 6, depth 'd' = 3

Command: java PathComputation 2 6 3 will execute query:

```
select distinct EQU.d,EQU.i,EQU.j,EQU.p,EQU.v,EQU.Edge,R3.Edge,
R2.Edge,R1.Edge from EQU
left join (select * from EQU where d= 2 and i=2) as R2 on R3.Edge=R2.j and
R2.v=(EQU.v-EQU.LW-R3.LW)
left join (select * from EQU where d=1 and i=2) as R1 on R2.Edge=R1.j and
```

R1.v=(EQU.v-EQU.LW-R3.LW-R2.LW)
where EQU.i = 2 and EQU.j = 6 and EQU.d=3;

Above query is calculating all the possible paths between ‘i’ & ‘j’.

2) After applying our approach for speeding up the transitive closure (i.e. adding two additional columns in EQU table) we tested on small graph that our new transitive closure is calculating each and every path between the two vertices.

3) We made sure while applying our approach for speeding up transitive closure computation that the path from older version of transitive closure and from newer version of transitive closure should stay same.

REFERENCES:

[1] C. Ordonez. Optimization of linear recursive queries in SQL. IEEE Transactions on Knowledge and Data Engineering (TKDE),22(2):264–277, 2010.

[2] C. Ordonez, W. Cabrera, and A. Gurram. Comparing columnar, row and array dbmss to process recursive queries on graphs. Information Systems, 2016.