**CIS 611 Enterprise Database Systems and Data Warehouse with OLAP**

**PROJECT REPORT**

**![A close up of a sign

Description automatically generated]()**

**Building a Simple Query Optimizer with Performance Evaluation Experiment on Query Rewrite Optimization**

**Professor**:

Dr. Sunnie Sun Chung

**Group Members:**

Megha Sravani. Lavu (2762646)

Shameena Taj Sultana Begum. Shaik (2774042)

**Aim:** To build a Simple Query Optimizer for Performance Evaluation experiment with Optimized Query and original query by finding the best cost evaluation join algorithm and minimum cost.

**Design of Query Optimizer Implementation:**

We have written a Java program with class name as Query Optimizer to calculate the cost i.e. number of Disk I/O’s required for given Q1 and RQ1 queries using the join algorithms of relational operators such as JOIN, Projection and Group By with Aggregation by iterating all different join orders and join methods for each possible query processing plan. And we have got the optimal plan to process the query at the end.

We have calculated the cost for each join statement and our code gives the best join algorithm below for Q1 and RQ1 in below given order:

**Join Methods:**

1) Tuple Nested Loop Join: TNL

2) Page Nested Loop Join: PNL

3) Block Nested Loop Join: with Buffer memory B =50: BNJM

4) Sort Merge Join with buffer memory B=50: SMJM

5) Hash Join with Buffer memory requirement B=50 pages for hash table: HJM

6) Hash Join with less Buffer memory B= 30 pages: HJL

7) Block Nested Loop Join: with less Buffer memory B =30: BNJL

8) Sort Merge Join with less buffer memory B=30: SMJL

Firstly, we have calculated the Join for given tables t1, t2 and then applied the Predicate % for Temp table calculation which is used as join table in next step of Q1 or RQ1. And the Predicate % was 10% \* 20% for Temp1 calculation of Q1 and TNL Join was calculated. Group by cost is also calculated and added to total cost. Projection cost is neglected as it is same for all queries.

For RQ1 we have calculated the Temp table of t1 and t3 and then used this as the join table at each step of query execution Joins by applying given Predicate percentages Temp1 table Pages calculation. Later applied the 12 possibilities of 3 tables t1,t2 and temp0. Then by comparing each query plan cost, we found the least cost of join and then added the calculated group by cost in total query processing cost. Then displayed the Q1 and RQ1 costs and the optimal plan for query processing.

Query costs are calculated using the below formulas for each Join methods:

M = Number of Pages in R table

N = Number of Pages in S table

pR = Number of tuples per page in R

B = Buffer Size

**1) Tuple Nested Loop Join: TNL**

TNLJoin calculation Cost = M (to scan R) + (pR \* M) \* N (to scan S) where,

**2) Page Nested Loop Join: PNL**

PNLJoin Cost: M (to scan R) + M\*N (to scan S)

3) **Block Nested Loop Join: with Buffer memory B =50: BNJM or B =30: BNJL (Less Buffer memory)**

BNLJoin Cost = M (Scan of outer table Ex: R) + Number of outer blocks \* scan of inner (scan of inner table Ex: S)

Where Number of outer blocks = Number of Pages in outer (i.e. Pages in R) / Buffer Memory

**4) Sort Merge Join with buffer memory B=50: SMJM or B=30: SMJL**

SMJM or SMJL Join cost = 2 (M + N) (1+ Ceiling (logB-1 M/B)) + M + N

If we have Buffer size B2 > Size of bigger Table i.e. B > √max (M, N) then

SMJM or SMJL Join cost = 3 \* (M + N)

**5) Hash Join with Buffer memory requirement B=50 pages for hash table: HJM or B= 30 pages: HJL**

If we have Buffer size B2 > Size of bigger Table i.e. B > √max (M, N) then

In partitioning phase, each table has to be read = 2(M+N)

In matching phase, only one read = M+N I/Os.

HJM or HJL cost: 3(M + N)

Otherwise we have used the below formula for remaining buffer sizes,

HJM or HJL cost = 2 \* (M+N) \* (1+logB-1(M)/B-1) + (M + N)

**Group by with Aggregation Calculation:**

Group by is nothing but Sorting cost.

Both Sorting base algorithms and Hash based algorithms for Group By:

Group by cost = 3(M + N)

If last step is Sort Merge Join then, Group by Cost is 2(M+N) as output will be in sorted manner.

Query processing time calculation is done using below formula:

Query Processing Cost = Disk I/O Cost

= Number of Disk I/O \* Disk Access Time

= Number of Disk I/O \* (8 ms + 4 ms)

= Total Number of Disk Block access needed \* 12 ms

The result of above cost will be in milliseconds and we are displaying the result in hours: minutes: seconds format.

**Description:**

A possible Query Execution Steps of Q1:

1. Join t1 t2 : any join methods for (temp1 <= t1 join t2)
2. Join temp1 t3 : For each tuple of temp1: only for TNJ (because of correlation) with two selectivities 10 % \* 20 % for the result

temp2 <= temp1 join t3 on the predicates

T2.x2 = T3.x3 : 10% and T1.x1 = T3.x3 : 20 %

1. Project temp2 : temp3 <= temp2 // neglected this cost as it is same for all queries

Aggregate without Group By

1. GroupBy temp3 with Aggregation

A possible Query Execution Steps of RQ1:

1. Join t1 t3 : any join methods for (temp1 <= t1 join t3)

with selectivity 20 % on T1.x1 = T3.x3 for the result temp0 <= t1 join t3

1. Group By temp0 on T1.Rowid with Aggregate Result table called Temp1
2. Join t1 Temp1 :
3. Join t2 temp2 : The Cost will be calculated for all 12 cases and leastcost out of all is added to total cost.

temp0 join (t1 join t2)

temp0 join (t2 join t1)

(t1 join t2) join temp0

(t2 join t1) join temp0

t1 join (temp0 join t2)

t1 join (t2 join temp0)

(temp0 join t2) join t1

(t2 join temp0) join t1

t2 join (temp0 join t1)

t2 join (t1 join temp0)

(temp0 join t1) join t2

(t1 join temp0) join t2

1. Project temp3: temp3 <= temp2 // neglected this cost as it is same for all queries
2. GROUP BY temp3. X1 with Aggregate

**Brief description of code:**

Methods:

**Tuplesperpage** => will return the number of tuples per page for a table.

**Tuplesperblock** => will return the number of tuples per block for a table.

**Numofblocks** => will return the total number of blocks for a table.

**Totaltuples** => will return total tuples in a relation

**costOfTNLJoin** => will calculate cost of TNL join for Q1.

**costOfJoinOptimization** => will return the cost of the best join and the join method is stored.

/\*\*\*\*\*\*\* Java Code for Simple Query Optimizer to find optimal plan\*\*\*\*\*\*\*\*\*/

import java.io. \*;

public class QueryOptimizer

{

// calculation

long ts\_t1 = 20, ts\_t2 = 40, ts\_t3 = 100;

long pages\_t1 = 1000, pages\_t2 = 500, pages\_t3 = 2000;

long pagesize = 4096; // 4096 bytes

long blocksize = 100; // 100 pages= 409600 bytes

static String joinMethod;

static long tempPages;

static long temptuplesperpage;

static double totalquerycost;

static double totalquerycost\_RQ;

static long tempPages\_RQ;

static long temp1Pages\_RQ;

static long temptuplesperpage\_RQ;

static long temp1tuplesperpage\_RQ;

static long mincost = 0;

static long mincost\_RQ = 0;

static long totalCost = 0;

static long totalCost\_RQ = 0;

static long leastcost\_RQ = 0, leastcost\_RQ2 = 0, leastcost\_RQ3 = 0, leastcostfinal = 0;

static long tuplesperpage (long pagesize, long tuplesize)

{

return pagesize / tuplesize;

}

static long tuplesperblock (long blocksize, long tuplesperpage)

{

return blocksize \* tuplesperpage;

}

static long numofblocks (long pagesintable, long blocksize)

{

return pagesintable / blocksize;

}

static long totaltuples (long pagesintable, long tuplesperpage)

{

return pagesintable \* tuplesperpage;

}

static long costOfTNLJoin (long ltPages, long rtPages, long ltTuplsperPage)

{

long TNLJoinCost = ltPages + ((ltTuplsperPage \* ltPages) \* rtPages);

double leastCostOfJoin = TNLJoinCost;

joinMethod = "TNLJoin";

return TNLJoinCost;

}

static long costOfJoinOptimization (long ltPages, long rtPages,

long ltTuplsperPage)

{

//Tuple Nested Loop Join:TNL: M (to scan R) + (pR \* M) times \* N (to scan S)

double TNLJoinCost = 0;

TNLJoinCost = ltPages + ((ltTuplsperPage \* ltPages) \* rtPages);

double leastCostOfJoin = TNLJoinCost;

joinMethod = "TNLJoin";

//Page Nested Loop Join:PNL: M (to scan R) + M times \*N

double PNLJoinCost = 0;

PNLJoinCost = ltPages + (ltPages \* rtPages);

if (leastCostOfJoin > PNLJoinCost)

{

leastCostOfJoin = PNLJoinCost;

joinMethod = "PNLJoin";

}

//Block Nested Loop Join: with Buffer memory B =50: BNJM

//BNJM cost: M (to scan R) + (M/B) times \*N

double BNJMJoinCost = 0;

BNJMJoinCost = ltPages + ((ltPages / 50) \* rtPages);

if (leastCostOfJoin > BNJMJoinCost)

{

leastCostOfJoin = BNJMJoinCost;

joinMethod = "BNJMJoin";

}

//Sort Merge Join with buffer memory B=50: SMJM

double SMJMJoinCost = 0;

if (50 > Math.sqrt (ltPages))

{

SMJMJoinCost = 3 \* (ltPages + rtPages);

}

else

{

SMJMJoinCost = (2 \* (ltPages + rtPages) \*(1 + Math.ceil ((Math.log10 (ltPages / 50)) / (Math.log10 (50 - 1))))) + ltPages + rtPages;

}

if (leastCostOfJoin > SMJMJoinCost)

{

leastCostOfJoin = SMJMJoinCost;

joinMethod = "SMJMJoin";

}

//Hash Join: HJM(Buffer = 50)

double HJMJoinCost = 0;

if (50 > Math.sqrt (ltPages))

{

HJMJoinCost = 3 \* (ltPages + rtPages);

}

else

{

HJMJoinCost = (2 \* (ltPages + rtPages) \*(1 +Math.ceil ((Math.log10 (ltPages / 50 - 1)) / (Math.log10 (50 - 1))))) + ltPages + rtPages;

}

if (leastCostOfJoin > HJMJoinCost)

{

leastCostOfJoin = HJMJoinCost;

joinMethod = "HJMJoin";

}

//Hash Join: HJL(Buffer = 30)

double HJLJoinCost = 0;

if (30 > Math.sqrt (ltPages))

{

HJLJoinCost = 3 \* (ltPages + rtPages);

}

else

{

HJLJoinCost = (2 \* (ltPages + rtPages) \*(1 +Math.ceil ((Math.log10 (ltPages / 30 - 1)) / (Math.log10 (30 - 1))))) + ltPages + rtPages;

}

if (leastCostOfJoin > HJLJoinCost)

{

leastCostOfJoin = HJLJoinCost;

joinMethod = "HJLJoin";

}

//Block Nested Loop Join: with Buffer memory B =30: BNJL

//BNJM cost: M (to scan R) + (M/B) times \*N

double BNJLJoinCost = 0;

BNJLJoinCost = ltPages + ((ltPages / 30) \* rtPages);

if (leastCostOfJoin > BNJLJoinCost)

{

leastCostOfJoin = BNJLJoinCost;

joinMethod = "BNJLJoin";

}

//Sort Merge Join with buffer memory B=30: SMJL

double SMJLJoinCost = 0;

if (30 > Math.sqrt (ltPages))

{

SMJLJoinCost = 3 \* (ltPages + rtPages);

}

else

{

SMJLJoinCost = (2 \* (ltPages + rtPages) \* (1 +Math.ceil ((Math.log10 (ltPages / 30)) / (Math.log10 (30 - 1))))) + ltPages + rtPages;

}

if (leastCostOfJoin > SMJLJoinCost)

{

leastCostOfJoin = SMJLJoinCost;

joinMethod = "SMJLJoin";

}

return (long) leastCostOfJoin;

}

public static void main (String[]args) throws Exception

{

long ltPages, rtPages, ltTuplsperPage;

QueryOptimizer q = new QueryOptimizer ();

long tuplesperpage\_t1 = tuplesperpage (q.pagesize, q.ts\_t1);

long tuplesperblock\_t1 = tuplesperblock (q.blocksize, tuplesperpage\_t1);

long totaltuples\_t1 = totaltuples (q.pages\_t1, tuplesperpage\_t1);

long numofblocks\_t1 = numofblocks (q.pages\_t1, q.blocksize);

long tuplesperpage\_t2 = tuplesperpage (q.pagesize, q.ts\_t2);

long tuplesperblock\_t2 = tuplesperblock (q.blocksize, tuplesperpage\_t2);

long totaltuples\_t2 = totaltuples (q.pages\_t2, tuplesperpage\_t2);

long numofblocks\_t2 = numofblocks (q.pages\_t2, q.blocksize);

long tuplesperpage\_t3 = tuplesperpage (q.pagesize, q.ts\_t3);

long tuplesperblock\_t3 = tuplesperblock (q.blocksize, tuplesperpage\_t3);

long totaltuples\_t3 = totaltuples (q.pages\_t3, tuplesperpage\_t3);

long numofblocks\_t3 = numofblocks (q.pages\_t3, q.blocksize);

File file = new File ("Q1.txt");

BufferedReader br = new BufferedReader (new FileReader (file));

String st;

System.out.

println ("----------Q1 Query-------------------------------\n");

while ((st = br.readLine ()) != null)

{

if (st.contains ("Join"))

{

String[]jstr = st.split (" ");

if ((jstr[1].equals ("t1") && jstr[2].equals ("t2"))

|| (jstr[1] == "t2" && jstr[2] == "t1"))

{

ltPages = q.pages\_t1;

rtPages = q.pages\_t2;

ltTuplsperPage = tuplesperpage\_t1;

mincost =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

ltPages = q.pages\_t2;

rtPages = q.pages\_t1;

ltTuplsperPage = tuplesperpage\_t2;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost)

{

mincost =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

}

totalCost = totalCost + mincost;

// new

tempPages = (long) (0.1 \* 0.2 \* ltPages \* rtPages);

long temptuplesize = 60;

temptuplesperpage = tuplesperpage (q.pagesize, temptuplesize);

// display

System.out.println (st + " " + joinMethod + " cost=" +

mincost);

}

else if ((jstr[1].equals ("t1") && jstr[2].equals ("t3"))

|| (jstr[1] == "t3" && jstr[2] == "t1"))

{

ltPages = q.pages\_t1;

rtPages = q.pages\_t3;

ltTuplsperPage = tuplesperpage\_t1;

mincost =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

ltPages = q.pages\_t3;

rtPages = q.pages\_t1;

ltTuplsperPage = tuplesperpage\_t3;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost)

{

mincost =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

}

totalCost = totalCost + mincost;

tempPages = (long) (ltPages \* rtPages);

long temptuplesize = 120;

temptuplesperpage = tuplesperpage (q.pagesize, temptuplesize);

}

else if ((jstr[1] == "t2" && jstr[2] == "t3")

|| (jstr[1] == "t3" && jstr[2] == "t2"))

{

ltPages = q.pages\_t2;

rtPages = q.pages\_t3;

ltTuplsperPage = tuplesperpage\_t2;

mincost =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

ltPages = q.pages\_t3;

rtPages = q.pages\_t2;

ltTuplsperPage = tuplesperpage\_t3;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost)

{

mincost =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

}

totalCost = totalCost + mincost;

tempPages = (long) (ltPages \* rtPages);

long temptuplesize = 140;

temptuplesperpage = tuplesperpage (q.pagesize, temptuplesize);

}

else if ((jstr[1].equals ("temp1") && jstr[2].equals ("t3"))

|| (jstr[1].equals ("t3") && jstr[2].equals ("temp1")))

{

ltPages = tempPages;

rtPages = q.pages\_t3;

ltTuplsperPage = temptuplesperpage;

mincost = costOfTNLJoin (ltPages, rtPages, ltTuplsperPage);

ltPages = q.pages\_t3;

rtPages = tempPages;

ltTuplsperPage = tuplesperpage\_t3;

if (costOfTNLJoin (ltPages, rtPages, ltTuplsperPage) <

mincost)

{

mincost =

costOfTNLJoin (ltPages, rtPages, ltTuplsperPage);

}

totalCost = totalCost + mincost;

System.out.println (st + " " + joinMethod + " cost=" +

mincost);

}

}

else if (st.contains ("Project"))

{

// neglecting the cost by projection as it is same for all

System.out.println (st + " " +

"Neglecting this cost as it is same for all queries");

}

else if (st.contains ("GroupBy"))

{

long groupbycost = 0;

if (joinMethod.equals ("SMJMJoin")

|| joinMethod.equals ("SMJLJoin"))

{

groupbycost = (tempPages + q.pages\_t3);

totalCost += 2 \* (tempPages + q.pages\_t3);

}

else

{

groupbycost = (tempPages + q.pages\_t3);

totalCost += 3 \* (tempPages + q.pages\_t3);

}

System.out.println (st + " " + "cost = " + groupbycost);

}

}

//Query Processing Cost = Disk I/O Cost = # of Disk I/O \* Disk Access Time(i.e 8 ms + 4 ms)

totalquerycost = totalCost \* 0.012; // In milliseconds

long hh = (long) totalquerycost / 3600; // In hours

long mm = (long) (totalquerycost - (hh \* 3600)) / 60;

long ss = (long) (totalquerycost - (hh \* 3600) - (mm \* 60));

System.out.println ("Total Disk I/O cost of Q1: " + totalCost);

System.out.println ("Query Processing Time: HH:MM:SS = " + hh + " hours" +

":" + mm + " mins" + ":" + ss + " sec");

System.out.

println ("----------RQ1 Query-------------------------------\n");

File file1 = new File ("RQ1.txt");

BufferedReader b = new BufferedReader (new FileReader (file1));

String st1;

int flag = 0;

while ((st1 = b.readLine ()) != null)

{

if (st1.contains ("Join"))

{

String[]jstr1 = st1.split (" ");

if ((jstr1[1].equals ("t1") && jstr1[2].equals ("t3"))

|| (jstr1[1].equals ("t3") && jstr1[2].equals ("t1")))

{

ltPages = q.pages\_t1;

rtPages = q.pages\_t3;

ltTuplsperPage = tuplesperpage\_t1;

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

ltPages = q.pages\_t3;

rtPages = q.pages\_t1;

ltTuplsperPage = tuplesperpage\_t3;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost\_RQ)

{

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

}

totalCost\_RQ = totalCost\_RQ + mincost\_RQ;

tempPages\_RQ = (long) (0.20 \* ltPages \* rtPages);

long temptuplesize = 120;

temptuplesperpage\_RQ =

tuplesperpage (q.pagesize, temptuplesize);

System.out.println (st1 + " " + joinMethod + " cost=" +

mincost\_RQ);

}

else if ((jstr1[1].equals ("t2") && jstr1[2].equals ("temp2"))

|| (jstr1[1].equals ("temp2") && jstr1[2].equals ("t2")))

{

ltPages = q.pages\_t1;

rtPages = q.pages\_t2;

ltTuplsperPage = tuplesperpage\_t1;

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

leastcost\_RQ = mincost\_RQ;

ltPages = q.pages\_t2;

rtPages = q.pages\_t1;

ltTuplsperPage = tuplesperpage\_t2;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost\_RQ)

{

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

leastcost\_RQ = mincost\_RQ;

}

temp1Pages\_RQ = (long) (0.15 \* ltPages \* rtPages);

long temptuplesize = 60;

temp1tuplesperpage\_RQ =

tuplesperpage (q.pagesize, temptuplesize);

// temp1 temp to be joined

ltPages = temp1Pages\_RQ;

rtPages = tempPages\_RQ;

ltTuplsperPage = temp1tuplesperpage\_RQ;

long mincost\_RQ1 =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

ltPages = tempPages\_RQ;

rtPages = temp1Pages\_RQ;

ltTuplsperPage = temptuplesperpage\_RQ;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost\_RQ1)

{

mincost\_RQ1 =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

}

leastcost\_RQ = leastcost\_RQ + mincost\_RQ1;

leastcostfinal = leastcost\_RQ;

// 2. tmp and t2 join with t1

ltPages = tempPages\_RQ;

rtPages = q.pages\_t2;

ltTuplsperPage = temptuplesperpage\_RQ;

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

leastcost\_RQ2 = mincost\_RQ;

ltPages = q.pages\_t2;

rtPages = tempPages\_RQ;

ltTuplsperPage = tuplesperpage\_t2;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost\_RQ)

{

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

leastcost\_RQ2 = mincost\_RQ;

}

temp1Pages\_RQ = (long) (0.10 \* ltPages \* rtPages);

long temp1tuplesize = 180;

temp1tuplesperpage\_RQ =

tuplesperpage (q.pagesize, temp1tuplesize);

ltPages = temp1Pages\_RQ;

rtPages = q.pages\_t1;

ltTuplsperPage = temp1tuplesperpage\_RQ;

long mincost\_RQ2 =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

ltPages = q.pages\_t1;

rtPages = temp1Pages\_RQ;

ltTuplsperPage = tuplesperpage\_t1;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost\_RQ1)

{

mincost\_RQ2 =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

}

leastcost\_RQ2 += mincost\_RQ2;

if (leastcost\_RQ2 < leastcostfinal)

{

leastcostfinal = leastcost\_RQ2;

}

// perm 3

ltPages = tempPages\_RQ;

rtPages = q.pages\_t1;

ltTuplsperPage = temptuplesperpage\_RQ;

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

leastcost\_RQ3 = mincost\_RQ;

ltPages = q.pages\_t1;

rtPages = tempPages\_RQ;

ltTuplsperPage = tuplesperpage\_t1;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost\_RQ)

{

mincost\_RQ =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

leastcost\_RQ3 = mincost\_RQ;

}

temp1Pages\_RQ = (long) (0.15 \* ltPages \* rtPages);

long temp2tuplesize = 160;

temp1tuplesperpage\_RQ =

tuplesperpage (q.pagesize, temp2tuplesize);

// temp1 temp to be joined

ltPages = temp1Pages\_RQ;

rtPages = q.pages\_t2;

ltTuplsperPage = temp1tuplesperpage\_RQ;

long mincost\_RQ3 =

costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage);

ltPages = q.pages\_t2;

rtPages = temp1Pages\_RQ;

ltTuplsperPage = tuplesperpage\_t2;

if (costOfJoinOptimization (ltPages, rtPages, ltTuplsperPage)

< mincost\_RQ1)

{

mincost\_RQ3 =

costOfJoinOptimization (ltPages, rtPages,

ltTuplsperPage);

}

leastcost\_RQ3 += mincost\_RQ3;

if (leastcost\_RQ3 < leastcostfinal)

{

leastcostfinal = leastcost\_RQ3;

}

System.out.println ("\n" + st1 + " " + joinMethod +

" Cost= " + leastcostfinal);

}

else if ((jstr1[1].equals ("t1") && jstr1[2].equals ("temp1"))

|| (jstr1[1].equals ("temp1") && jstr1[2].equals ("t1")))

{

System.out.print (st1 + " " + "SMJMJoin");

}

totalCost\_RQ += leastcostfinal;

}

else if (st1.contains ("Project"))

{

System.out.println (st1 + " " +

"Neglecting this cost as it is same for all queries");

}

else if (st1.contains ("GroupBy"))

{

long groupbycost = 0;

if (joinMethod.equals ("SMJMJoin")

|| joinMethod.equals ("SMJLJoin"))

{

if (flag == 0)

{

groupbycost = 2 \* (q.pages\_t1 + q.pages\_t3);

totalCost\_RQ += 2 \* (q.pages\_t1 + q.pages\_t3);

flag++;

}

else

{

groupbycost = 2 \* (temp1Pages\_RQ + q.pages\_t3);

totalCost\_RQ += 2 \* (temp1Pages\_RQ + q.pages\_t3);

}

}

else

{

if (flag == 0)

{

totalCost\_RQ += 3 \* (q.pages\_t1 + q.pages\_t3);

flag++;

}

else

{

totalCost\_RQ += 3 \* (temp1Pages\_RQ + q.pages\_t3);

}

}

System.out.println (st1 + " " + "cost = " + groupbycost);

}

}

// RQ1 total query processing time

totalquerycost\_RQ = totalCost\_RQ \* 0.012; // In milliseconds

long hh\_RQ = (long) totalquerycost\_RQ / 3600; // In hours

long mm\_RQ = (long) (totalquerycost\_RQ - (hh\_RQ \* 3600)) / 60;

long ss\_RQ = (long) (totalquerycost\_RQ - (hh\_RQ \* 3600) - (mm\_RQ \* 60));

System.out.println ("Total Disk I/O cost of RQ1: " + totalCost\_RQ);

System.out.println ("Query Processing Time: HH:MM:SS = " + hh\_RQ +

" hours" + ":" + mm\_RQ + " mins" + ":" + ss\_RQ +

" sec");

System.out.println ("\n-----------Best Query------------------");

if (totalCost > totalCost\_RQ)

{

System.out.

println

("The Best Query Plan is RQ1 as Cost of RQ1 is less than Q1");

}

else

{

System.out.

println

("The Best Query Plan is Q1 as Cost of Q1 is less than RQ1");

}

}

}

**Input for Query Optimizer:**

|  |  |
| --- | --- |
| **Q1.txt** | **RQ1.txt** |
| Join t1 t2  Join temp1 t3  Project temp2  GroupBy temp3 | Join t1 t3  GroupBy temp0  Join t1 temp1  Join t2 temp2  Project temp3  GroupBy temp3 |

**Output of code:**

A screenshot of a newspaper

Description automatically generated

**Conclusion:**

Two queries Q1 and RQ1 are taken from txt file and analyzed the cost of each statement in those queries to determine the best plan.