Report on

Vertical Fragmentation Lab Assignment

under the guidance of

J R Shruti

Submitted by

Nilita Anil Kumar 16IT122

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

Department of Information Technology
National Institute of Technology Karnataka, Surathkal.

June 2020

Vertical Fragmentation

1. Obtain **attribute affinity matrix**; this matrix tells how closely related attributes are
2. Use a **clustering algorithm** to group some attributes together based on the attribute affinity matrix. This algorithm produces a clustered affinity matrix.
3. Use a **partitioning algorithm** to partition attributes such that sets of attributes are accessed solely or for the most part by distinct sets of applications.

The following are the attributes corresponding to the relation:

Q1: Select BUDGET from PROJ where PNO = VALUE;

Q2: Select PNAME, BUDGET from PROJ;

Q3: Select PNAME from PROJ where LOC = VALUE

Q4: Select SUM(BUDGET) from PROJ where LOC = VALUE

A1 = PNO, A2 = PNAME. A3 = BUDGET, A4 = LOC

INPUTS:

Query access matrix

Query access matrix - use of attributes in application queries

```
                         A1  A2  A3  A4
Query access matrix =  [[1,   0,   1,   0], Q1
                         [0,   1,   1,   0], Q2
                         [0,   1,   0,   1], Q3
                         [0,   0,   1,   1]] Q4
```

Frequency Access Matrix

Frequency Access Matrix - no of times the queries accesses the sites in a day

```
Frequency_access_matrix =
   S1     S2       S3
 [[15,     20,     10],   Q1
  [5,       0,       0],    Q2
  [25,     25,      25],  Q3
  [3,       0,       0]]    Q4
```

OUTPUT : Clustered Affinity Matrix

ALGORITHM

Step 1: Taking the sum of attribute access by each query from the frequency access matrix

Q1 Q2 Q3 Q4

[45 5   75 3]

Step 2 : Finding out the attribute usage with frequency called the Attribute Affinity matrix (AA Matrix)

The attribute affinity represents the strength of bond between the two attributes. The attribute affinity for two attributes Ai and Aj defined as

aff(Ai, Aj)=k | use(qk, Ai) =1^ use(qk,Aj) =1 acc (qk)

where aff(Ai, Aj) is the affinity value between Ai and Aj, acc(qk) is the total number of access of query qk generated in multiple sites.

```
def affinity_calc():
    global attr_affinity_matrix
    for col_attr in range(no_of_attr):
        for row_attr in range(1, no_of_attr + 1):
            affinity_value = 0
            for q in range(no_of_queries):
                if query_access_matrix[q][col_attr] == 1 & query_access_matrix[q][row_attr - 1] == 1:
                    affinity_value += sum_attr_access[q]
            attr_affinity_matrix[row_attr][col_attr] = affinity_value
    return attr_affinity_matrix
```

```
attribute affinity matrix =
[[ 1.  2.  3.  4.]
 [45.  0. 45.  0.]
 [ 0. 80.  5. 75.]
 [45.  5. 53.  3.]
 [ 0. 75.  3. 78.]]
```

Step 3 : Finding out the clustered affinity matrix using Bond Energy Algorithm

The purpose of clustering is to combine large affinity values of AA matrix with large affinity values, a
the small one with small ones.

Function to calculate the bond between two columns

```
def bond(left, right):
    bond_value = 0
    # Boundary conditions
    if left == -1 or left == no_of_attr or right == -1 or right == no_of_attr:
        return bond_value
    else:
        bond_value = np.sum(np.multiply(attr_affinity_matrix[1:, left], attr_affinity_matrix[1:, right]))
        return bond_value
```

Function to calculate the contribution of a certain configuration of columns.
Example: to calculate contribution of placement of (A2, A4, A3), this function is called with left = middle = 4, right = 3, and a reference to the Affinity Matrix object.

```
def contribution(left, middle, right):
    a = bond(left, middle)
    b = bond(middle, right)
    c = bond(left, right)
    if right == middle + 1:
        cont = 2 * a
    else:
        cont = 2 * (a + b - c)
    return cont
```

Pseudo Code for Bond Energy Algorithm

```
Algorithm: BEA
Input: AA attribute affinity matrix
Output: CA clustered affinity matrix
Begin
{Initialize; remember that AA is an n x n Matrix}
CA(R· ,1) ←AA(R,1)
CA(R ,2) ←AA(R,2)
index←3
while index≤n do {choose the "best" location for attribute AAindex}
begin
for I from 1 to index-1 by 1 do
calculate cont(Aindex-1, Aindex, Aindex+1)
end-for
calculate cont(Aindex-1, Aindex, Aindex+1) {boundary condition}
location←placement given by maximum cont value
for j from index to location by -1 do {shuffle the two matrices}
CA(R ,j) ←AA(R,j-1)
end-for
CA(R ,location) ←AA(R,index)
index←index+1
```

```
end-while
order the rows according to the relative ordering of columns
end.{BEA}
```

```
# Bond Energy Algorithm
def bea_algo():
    clustered_affinity_matrix = np.zeros((no_of_attr + 1, no_of_attr))
    # Copy the first and second columns from the Attribute Affinity Matrix
    clustered_affinity_matrix[:, 0] = attr_affinity_matrix[:, 0]
    clustered_affinity_matrix[:, 1] = attr_affinity_matrix[:, 1]
    print("After shifting rows 0 and 1=")
    print(clustered_affinity_matrix)

    # Best Placement of attributes starting from index 2 that is the third attribute
    for index in range(2, no_of_attr):
        contribution_array = []
        print("Best location for attribute = ", attr_affinity_matrix[:, index])
        # Calculating the contribution value
        for i in range(index):
            contribution_value = contribution(i - 1, index, i)
            contribution_array.append(contribution_value)

        contribution_array.append(contribution(index - 1, index, index + 1))
        # loc <- placement given by max contribution value
        loc = contribution_array.index(max(contribution_array))
        print("Location of max cont = ", loc + 1)

        # Shifting attribute to the location of max contribution in CA
        for k in range(index, loc, -1):
            clustered_affinity_matrix[:, k] = clustered_affinity_matrix[:, k - 1]
        clustered_affinity_matrix[:, loc] = attr_affinity_matrix[:, index]
        print("CA after swapping attribute", index + 1)
        print(clustered_affinity_matrix)

        # Shifting attribute to the location of max contribution in AA
        temp = attr_affinity_matrix[:, index].copy()
        for m in range(index, loc, -1):
            attr_affinity_matrix[:, m] = attr_affinity_matrix[:, m - 1]
        attr_affinity_matrix[:, loc] = temp
        print("AA after swapping attribute", index + 1)
        print(attr_affinity_matrix)

    # Interchanging of rows in CA after the BEA algorithm
    CA_ordered_row = np.zeros((no_of_attr, no_of_attr))
    n = 0
    for m in range(no_of_attr):
        order = clustered_affinity_matrix[0, :]
        CA_ordered_row[n, :] = clustered_affinity_matrix[int(order[m]), :]
        n += 1
    clustered_affinity_matrix[1:][:] = CA_ordered_row
    print("CA after interchanging rows = ")
    print(clustered_affinity_matrix)

    return clustered_affinity_matrix
```

CONSOLE OUTPUT :

```
After shifting rows 0 and 1=
[[ 1.  2.  0.  0.]
 [45.  0.  0.  0.]
 [ 0. 80.  0.  0.]
 [45.  5.  0.  0.]
 [ 0. 75.  0.  0.]]
Best location for attribute =  [ 3. 45.  5. 53.  3.]
cont( A 0 , A 3 , A 1 ) = 2 * ( 0  +  4410.0  -  0 ) =  8820.0
cont( A 1 , A 3 , A 2 ) = 2 * ( 4410.0  +  890.0  -  225.0 ) =  10150.0
cont( A 2 , A 3 , A 4 ) = 2 * ( 890.0  +  768.0  -  11865.0 ) =  1780.0
Location of max cont =  2
CA after swapping attribute 3
[[ 1.  3.  2.  0.]
 [45. 45.  0.  0.]
 [ 0.  5. 80.  0.]
 [45. 53.  5.  0.]
 [ 0.  3. 75.  0.]]
AA after swapping attribute 3
[[ 1.  3.  2.  4.]
 [45. 45.  0.  0.]
 [ 0.  5. 80. 75.]
 [45. 53.  5.  3.]
 [ 0.  3. 75. 78.]]
```

```
Best location for attribute =  [ 4.  0. 75.  3. 78.]
cont( A 0 , A 4 , A 1 ) = 2 * ( 0  +  135.0  -  0 ) =  270.0
cont( A 1 , A 4 , A 2 ) = 2 * ( 135.0  +  768.0  -  4410.0 ) =  -7014.0
cont( A 2 , A 4 , A 3 ) = 2 * ( 768.0  +  11865.0  -  890.0 ) =  23486.0
cont( A 3 , A 4 , A 5 ) = 2 * ( 11865.0  +  0  -  0 ) =  23730.0
Location of max cont =  4
CA after swapping attribute 4
[[ 1.  3.  2.  4.]
 [45. 45.  0.  0.]
 [ 0.  5. 80. 75.]
 [45. 53.  5.  3.]
 [ 0.  3. 75. 78.]]
AA after swapping attribute 4
[[ 1.  3.  2.  4.]
 [45. 45.  0.  0.]
 [ 0.  5. 80. 75.]
 [45. 53.  5.  3.]
 [ 0.  3. 75. 78.]]
CA after interchanging rows =
[[ 1.  3.  2.  4.]
 [45. 45.  0.  0.]
 [45. 53.  5.  3.]
 [ 0.  5. 80. 75.]
 [ 0.  3. 75. 78.]]
CA =
[[ 1.  3.  2.  4.]
 [45. 45.  0.  0.]
 [45. 53.  5.  3.]
 [ 0.  5. 80. 75.]
 [ 0.  3. 75. 78.]]
```

Partitioning

Finding the sets of attributed that are accessed for the most part, by distinct sets of applicatio
(Queries)
We look for dividing points along the diagonal such that
1. Total accesses to only one fragment is maximized, while
2. Total accesses to more than one fragments are minimized

```
# Find the partitioning point such that cost Z is maximized
def partition():
    z = []
    fragments = []
    for split_point in range(1, len(order_CA)):
        frag1 = order_CA[0:split_point]
        frag2 = order_CA[split_point:len(order_CA)]
        fragments.append([frag1, frag2])
        print("Fragments =", frag1, frag2)

        TA = []  # Cluster for attributes in frag 1
        TB = []  # Cluster for attributes in frag 2
        for i in range(no_of_queries):
            use_frag1 = 0
            for items in frag1:
                # if queries accesses any of the attributes in fragment 1
                # then TA of that query will be 1
                if QA_matrix[i, items - 1] == 1:
                    use_frag1 = 1
                    break
            TA.append(use_frag1)
            use_frag2 = 0
            for items in frag2:
                # if queries accesses any of the attributes in fragment 2
                # then TB of that query will be 1
                if QA_matrix[i, items - 1] == 1:
                    use_frag2 = 1
                    break
            TB.append(use_frag2)
        print("TA =", TA)
        print("TB =", TB)

        # Queries are classified only into three sets
        TQ = []  # Applications that only use attributes in TA
        BQ = []  # Applications that only use attributes in BA
        OQ = []  # Applications that only use attributes in both TA and BA
        for i in range(no_of_queries):
            if TA[i] == 0 and TB[i] == 1:
                BQ.append(i + 1)
            elif TA[i] == 1 and TB[i] == 0:
                TQ.append(i + 1)
            else:
                OQ.append(i + 1)
        print("TQ =", TQ)
        print("BQ =", BQ)
        print("OQ =", OQ)

        CTQ = sum_access(TQ # Total number of accesses to attributes by TQ
        CBQ = sum_access(BQ)# Total number of accesses to attributes by BQ
        COQ = sum_access(OQ)# Total number of accesses to attributes by OQ

        # Find the partitioning point such that cost Z is maximized
```

```
        z.append(CTQ * CBQ - math.pow(COQ, 2))
        print("z =", z)

    if max(z) < 0:
        print("Vertical Fragmentation not possible.")
    else:
       print("Best partitioning point = ", fragments[z.index(max(z))][0], fragments[z.index(max(z))][
```

OUTPUT CONSOLE

```
Fragments = [1] [3 2 4]
TA = [1, 0, 0, 0]
TB = [1, 1, 1, 1]
TQ = []
BQ = [2, 3, 4]
OQ = [1]
z = [-2025.0]
Fragments = [1 3] [2 4]
TA = [1, 1, 0, 1]
TB = [0, 1, 1, 1]
TQ = [1]
BQ = [3]
OQ = [2, 4]
z = [-2025.0, 3311.0]
Fragments = [1 3 2] [4]
TA = [1, 1, 1, 1]
TB = [0, 0, 1, 1]
TQ = [1, 2]
BQ = []
OQ = [3, 4]
z = [-2025.0, 3311.0, -6084.0]
Best partitioning point =  [1 3] [2 4]
```