JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA 62

INTRODUCTION TO LARGE SCALE DATABASE SYSTEMS

PROJECT REPORT

Improvement In Bond Energy Algorithm For Vertical Fragmentation Of a Relational Database

SEMESTER 6

SUBMITTED BY:

MANEKA SINGH - B3 20103074

SURYA PRATAP SINGH - 20104049

SUBMITTED TO:

DR. PARMEET KAUR SODHI

JIIT-Sector 62, Noida

Table of Content

INTRODUCTION

- **Bond Energy Algorithm (BEA)**

The bond energy algorithm (BEA) was developed and it is used in the database design area to determine how to group data and how to physically place data on a disk. It can be used to cluster attributes based on usage and then perform logical or physical design accordingly. With BEA, the affinity (bond) between database attributes is based on common usage. This bond is used by the clustering algorithm as a similarity measure. The actual measure counts the number of times the two attributes are used together in a given time. To find this, all common queries must be identified. The idea is that attributes that are used together form a cluster and should be stored together. In a distributed database, each resulting cluster is called a vertical fragment and may be stored at different sites from other fragments.

- **The Algorithm of Bond Energy states that :**

Let R($A_1, A_{2...} A_n$) be the relation into consideration where
A_1, A_2 are the attributes of the relation R. Q be be the set of user queries defined as $\{q_1, q_{2..}q_m\}$.
S be the total number of sites.

The Attribute Usage Value, a$uv(q_i, A_j)$ is defined as follows:

$$\boldsymbol{auv}(\boldsymbol{q_i A_j}) = \begin{cases} 1, \textit{if } \boldsymbol{A_j} \textit{ is used by } \boldsymbol{q_i} \\ 0, \textit{ otherwise} \end{cases}$$

The Attribute Affinity Measure between any two attributes, aamðAi ; Aj Þ that determines the closeness between the two attri- butes on basis of the queries accessed by the sites. It can be defined as:

$$aam(A_i, A_j) = \sum_{k|auv(q_k, A_i)=1 \wedge auv(q_k, A_j)=1} \sum_{\forall S_l} naa_l(q_k) aafm_l(q_k)$$

where naa*i* (q_k) = number of accesses to attributes (A_i , A_j) for each execution of query q_k at site S_l and

aaf $m_{l(q_k)}$ = number of times the query q_k is accessed at siteS_l.

The BEA takes the Attribute Affinity Matrix as input and then outputs the clustered Affinity Matrix. The attributes are permu- tated with the objective to maximize the value of Global Affinity Measure (GAM) . The GAM value indicates that how close the attri- butes with greater affinities with each other are grouped :

$$GAM = \sum_{i}^{n} \sum_{j=1}^{n} aam(A_i, A_j) * (aam(A_i, A_{j-1}) + aam(A_i, A_{j+1}))$$

Notations	Meanings
R	Relation into consideration
Ai	Attributes of Relation R
Q	Set of Queries
S	Total number of sites
auv	Attribute usage value
aam	Attribute affinity measure
naa_1 (q_k)	Number of access to attributes for each execution ofquery q_k at site s_i
$aafm_i$ (q_k)	Number of times the query (q_k) is accessed at site s_iGAM Global affinity measure
AA	Attribute affinity matrix
CA	Clustered affinity matrix
cntr (A_i, A_k, A_j)	Net contribution made by placing A_k between A_i and A_jb (A_x, A_y) Bond between two attribute A_x and A_y
TL	Attribute after partition point
BR	Attribute before partition point
Cr	Crossover rate
F	Scale factor
P	Population Size
g	Number of generations
n	Number of attributes
TQ	group of queries that access only TL
BQ	group of queries that access only BR
OQ	group of queries that access both TL and BR
CTQ	total number of accesses to attributes by queries thataccess only TL
CBQ	total number of accesses to attributes by queries thataccess only BR
COQ	total number of accesses to attributes by queries thataccess both TL and BR

- **Improvement in Bond Energy Algorithm for optimal vertical fragmentation of distributed database - Differential Bond Energy Algorithm**

The Differential Bond Energy (DBE) algorithm is a molecular modeling technique that can also be applied to relational database design, specifically for vertical fragmentation. Vertical fragmentation involves dividing a table into two or more smaller tables, based on the attributes that are frequently accessed together.

To implement the DBE algorithm for vertical fragmentation of a relational database, you would need to calculate the energy differences between different attribute sets, and use this information to determine the optimal fragmentation scheme.

Differential bond energy algorithm (DBE) is proposed by incorporating evolutionary mechanisms of differential evolution algo- rithm (DE) in classical bond energy algorithm (BEA). BEA gets more combinations of attributes to find optimal ordering in clustering affinity matrix. Mutation and crossover strategies of DE could improve the clustering efficacy. DE is applied to explore different permutations of the attributes in AA so as to give maximum value of GAM. The permutation of the best solution gives the optimal Clustered Affinity Matrix (CA). Details of proposed algorithm are given in Algorithm 2.

- **The Algorithm of Differential Bond Energy states that :**

Input: The Attribute Affinity matrix (AA)

Output: The clustered affinity matrix CA which is aperturbation of AA

STEP A. Initialize the Control parameters:

Crossover rate, CR = 0.5 (0 < CR < 1)

Scale factor, F = 1.8 (0 < F < 2)

(The values of F and CR have been selected as 1.8 and 0.5 respectively because the experimental studies have provedthat using these values provide the best result.)

Initialize the population size ‘p’ and number of generations (g) and limit where limit is the stopping parameter for the algorithm. The algorithm stops if the solution does not improve continuously for ‘limit’ number of times.

STEP B. Initialize the population.

(Each vector is a randomly generated possible permutationof the given input matrix.)

Example: If number of attributes (n) = 5, then Pl = [2,4,0,3,1],

P2 = [1,3,0,4,2], and so on.

STEP C. Calculate fitness value for each vector.

Fitness (Pi) = Global Affinity Measure (GAM)

STEP D. Memorize the vector with the best fitness value.STEP E. gen = 0, count = 0

STEP F. while gen < generations and count < limit:for each Population vector i:

//Mutation (rand/1/bin)

Sort V and map the indices to the sorted array.This is our mutated vector.

//Crossover

Apply crossover operator between P[i] and V and storethe offspring in U.

The crossover operator is influenced by the Liuhai Crossover (Mei et al., 2010) focusing on the distance between the positions of indices of maximum attributeaffinity value in AA.

//Selection

Now, we select the vector among P[i] and U based on themaximum fitness value.

Update the best fitness vector.

If vector is updated, count = 0 else count+=1If (count == limit) break out of the loop.

STEP G. Print the matrix and the GAM value corresponding tothe best order at the end.

SUMMARY OF RESEARCH PAPER

According to the paper studied, " *the differential bond energy (DBE) algorithm is proposed with objective to determine optimal partition point. The performance of proposed algorithm is compared with classical bond energy algorithm (BEA) on basis of global affinity measure (GAM) value. Results are depicted in form of line graphs. The mean difference in GAM values for both algorithms are also illustrated.*" [1]

The objective of the algorithm is to find an optimal vertical fragmentation of a distributed database, which involves dividing the attributes of a relation into different fragments such that the fragmentation minimizes the communication cost between the fragments while satisfying certain constraints.

The algorithm presented in the paper is based on the concept of bond energy, which measures the degree of relatedness between attributes. The higher the bond energy between two attributes, the stronger their relationship, and they are more likely to be placed together in the same fragment.

Here is a summary of the steps involved in the algorithm:

1. **Initialization:** The control parameters, such as crossover rate (CR) and scale factor (F), are set. The population size, number of generations, and a limit parameter are also initialized.
2. **Population Initialization:** A population of possible attribute permutations is randomly generated. Each permutation represents a possible fragmentation of the attributes.
3. **Fitness Calculation:** The fitness value of each permutation is calculated using a fitness function based on the bond energy measure. The fitness function evaluates the degree of relatedness between attributes in a permutation.
4. **Best Vector Memorization:** The permutation with the best fitness value is stored as the best vector.
5. **Iteration and Selection:** The algorithm iterates for a specified number of generations. For each generation, the following steps are performed:
6. **a. Mutation:** A mutation operator is applied to generate a mutated vector by perturbing the current population vector.
7. **b. Crossover:** The crossover operator is applied between the current population vector and the mutated vector to produce an offspring vector.

8. **c. Selection:** The offspring vector is selected based on its fitness value, and it replaces the parent vector if it has a higher fitness value.
9. **Stopping Condition:** The algorithm stops if the solution does not improve continuously for a specified number of generations (limit parameter).
10. **Output:** The resulting best vector, representing the optimal vertical fragmentation of the distributed database, is printed along with its corresponding fitness value.

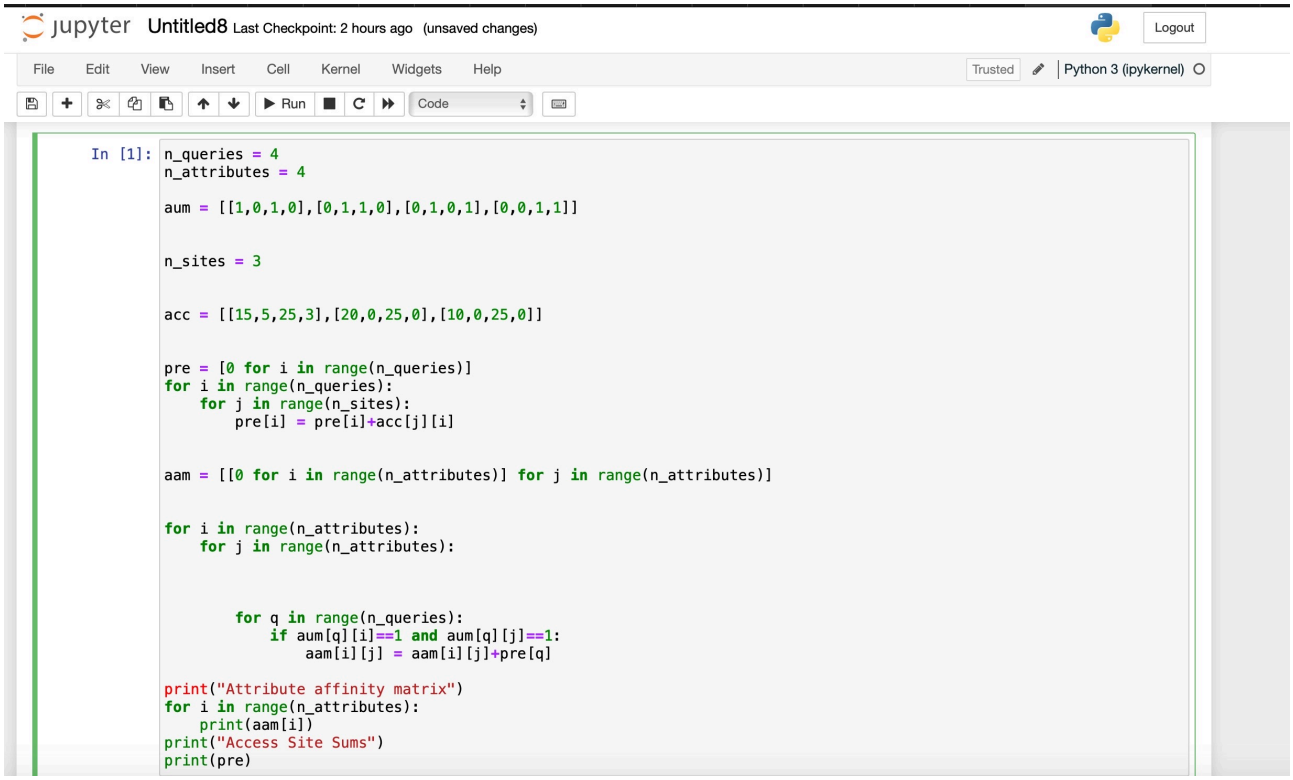
It's important to note that the exact details of the fitness function, mutation operator, and crossover operator may vary based on the specific implementation described in the research paper.

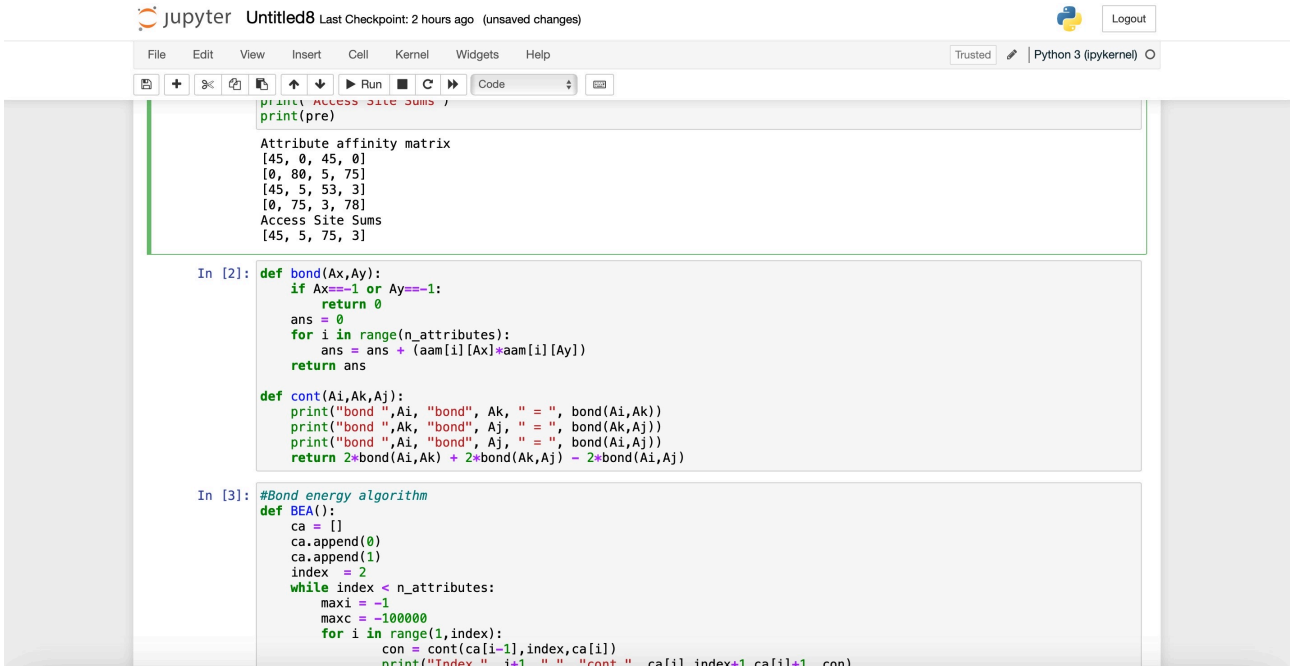
PROPOSED METHODOLOGY

1. **Bond Energy Algorithm:**
   - Attribute Affinity Matrix: The attribute affinity matrix (AA) is constructed to capture the relationships between attributes in the database.
   - Fitness Function: A fitness function is defined based on the attribute affinity matrix. The fitness function evaluates the quality of a fragmentation or permutation of the attributes using the bond energy measure.
   - Population Initialization: An initial population of attribute permutations is generated to represent different fragmentations of the database.
   - Fitness Evaluation: The fitness function is applied to each permutation in the population to calculate their fitness values.
   - Best Vector Memorization: The permutation with the best fitness value, representing the optimal fragmentation, is stored as the best vector.
   - Iterative Optimization: The algorithm iterates through multiple generations, applying mutation and crossover operators to explore and exploit the search space. Selection is performed based on the fitness values to determine the parent and offspring vectors.
   - Stopping Criteria: The algorithm terminates based on a predefined stopping criterion, such as reaching a maximum number of generations or a lack of improvement in fitness values.
   - Output: The best vector obtained after the algorithm terminates represents the optimal vertical fragmentation of the distributed database.

2. **Differential Bond Energy Algorithm:**
   - The differential bond energy algorithm is an extension of the bond energy algorithm that incorporates the concept of differential evolution for optimization.
   - Mutation: In addition to the mutation step in the bond energy algorithm, the differential bond energy algorithm employs a differential mutation operator, such as the "rand/1/bin" strategy.
   - Crossover: The crossover operator is applied between the parent and mutated vectors to generate an offspring vector.
   - Selection: The offspring vector is compared to the parent vector, and the one with the higher fitness value is selected to replace the parent in the population.
   - Stopping Criteria and Output: The stopping criteria and output remain the same as in the bond energy algorithm.

- **Implementation of BEA using Jupyter**





jupyter Untitled8 Last Checkpoint: 2 hours ago (unsaved changes)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3 (ipykernel)

Run Code

In [3]:
```
#Bond energy algorithm
def BEA():
    ca = []
    ca.append(0)
    ca.append(1)
    index  = 2
    while index < n_attributes:
        maxi = -1
        maxc = -100000
        for i in range(1,index):
                con = cont(ca[i-1],index,ca[i])
                print("Index ", i+1, " ", "cont ", ca[i],index+1,ca[i]+1, con)
                if con > maxc:
                    maxi = i
                    maxc = con
        #boundary left
        con = cont(-1,index,ca[0])
        print("Index ", i+1, " ", "cont ", 1,index+1,ca[0]+1, con)
        if con > maxc:
            maxi = 0
            maxc = con
        #boundary right
        con = cont(ca[index-1],index,-1)
        print("Index ", i+1, " ", "cont ", ca[index-1]+1,index+1,index+2, con)
        if con > maxc:
            maxi = index
        if maxi==index:
            ca.append(index)
        else:
            ca.append(0)
            for j in range(index,maxi,-1):
                ca[j]=ca[j-1]
            ca[maxi] = index
        print(ca)
        index = index + 1
    print("FINAL Clustered Affinity Matrix")
```

jupyter Untitled8 Last Checkpoint: 2 hours ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

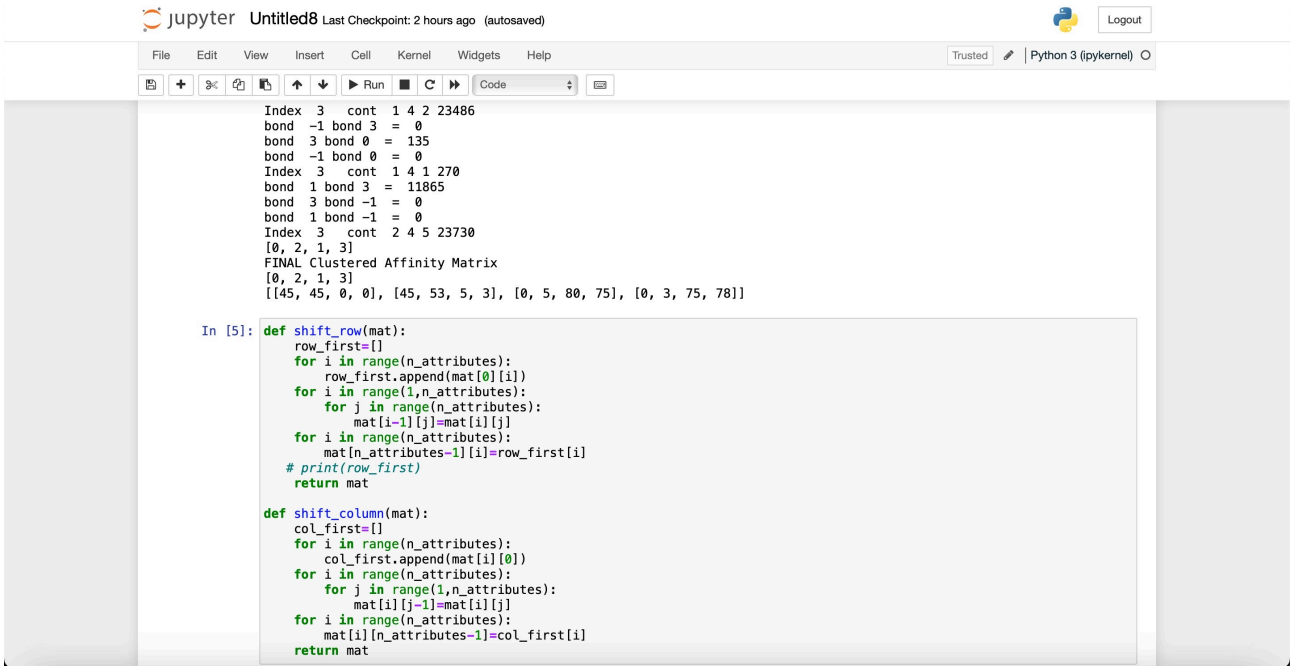
Trusted | Python 3 (ipykernel)

Run Code

```
    print("FINAL Clustered Affinity Matrix")
    print(ca)
    return ca
```

In [4]:
```
CA = BEA()
ca = [[0 for i in range(n_attributes)] for j in range(n_attributes)]
for i in range(n_attributes):
    for j in range(n_attributes):
        ca[i][j] = aam[CA[i]][CA[j]]

print(ca)
```

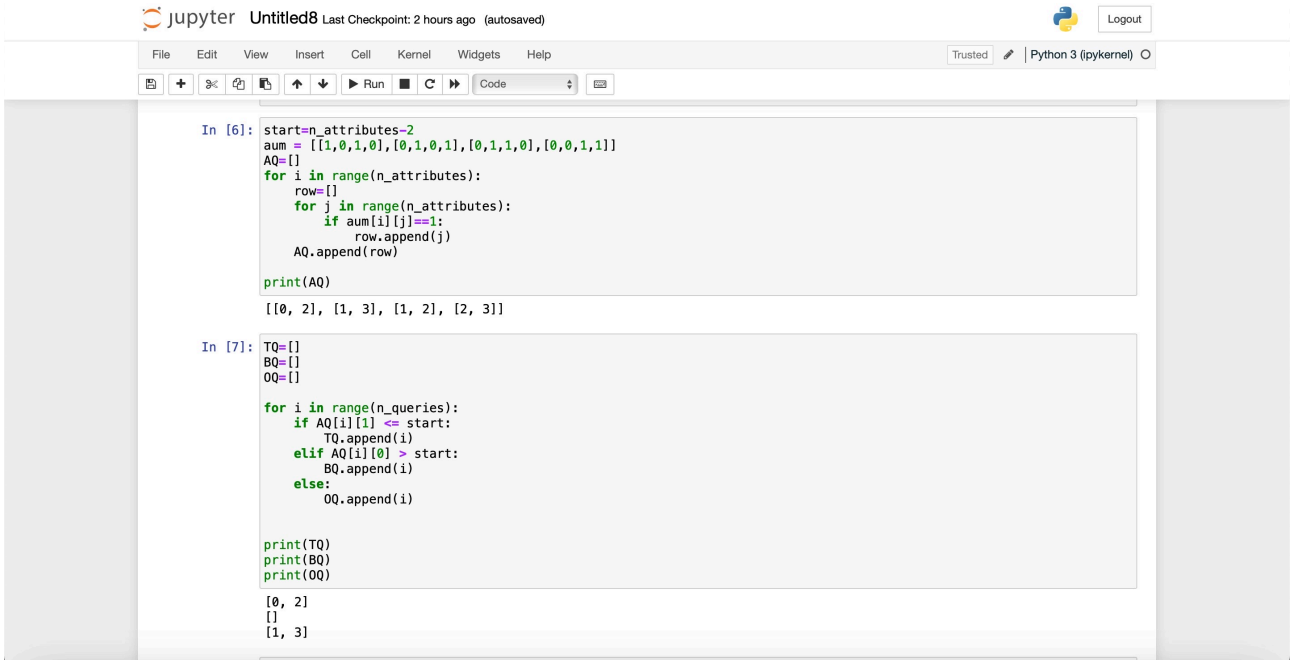
```
bond  0 bond 2  =  4410
bond  2 bond 1  =  890
bond  0 bond 1  =  225
Index  2   cont  1 3 2 10150
bond  -1 bond 2  =  0
bond  2 bond 0  =  4410
bond  -1 bond 0  =  0
Index  2   cont  1 3 1 8820
bond  1 bond 2  =  890
bond  2 bond -1  =  0
bond  1 bond -1  =  0
Index  2   cont  2 3 4 1780
[0, 2, 1]
bond  0 bond 3  =  135
bond  3 bond 2  =  768
bond  0 bond 2  =  4410
Index  2   cont  2 4 3 -7014
bond  2 bond 3  =  768
bond  3 bond 1  =  11865
bond  2 bond 1  =  890
Index  3   cont  1 4 2 23486
bond  -1 bond 3  =  0
bond  3 bond 0  =  135
bond  -1 bond 0  =  0
```




```
Index  3   cont  1 4 2 23486
bond  -1 bond 3  =  0
bond  3 bond 0  =  135
bond  -1 bond 0  =  0
Index  3   cont  1 4 1 270
bond  1 bond 3  =  11865
bond  3 bond -1  =  0
bond  1 bond -1  =  0
Index  3   cont  2 4 5 23730
[0, 2, 1, 3]
FINAL Clustered Affinity Matrix
[0, 2, 1, 3]
[[45, 45, 0, 0], [45, 53, 5, 3], [0, 5, 80, 75], [0, 3, 75, 78]]
```

In [5]:

```
def shift_row(mat):
    row_first=[]
    for i in range(n_attributes):
        row_first.append(mat[0][i])
    for i in range(1,n_attributes):
        for j in range(n_attributes):
            mat[i-1][j]=mat[i][j]
    for i in range(n_attributes):
        mat[n_attributes-1][i]=row_first[i]
   # print(row_first)
    return mat

def shift_column(mat):
    col_first=[]
    for i in range(n_attributes):
        col_first.append(mat[i][0])
    for i in range(n_attributes):
        for j in range(1,n_attributes):
            mat[i][j-1]=mat[i][j]
    for i in range(n_attributes):
        mat[i][n_attributes-1]=col_first[i]
    return mat
```



In [6]:

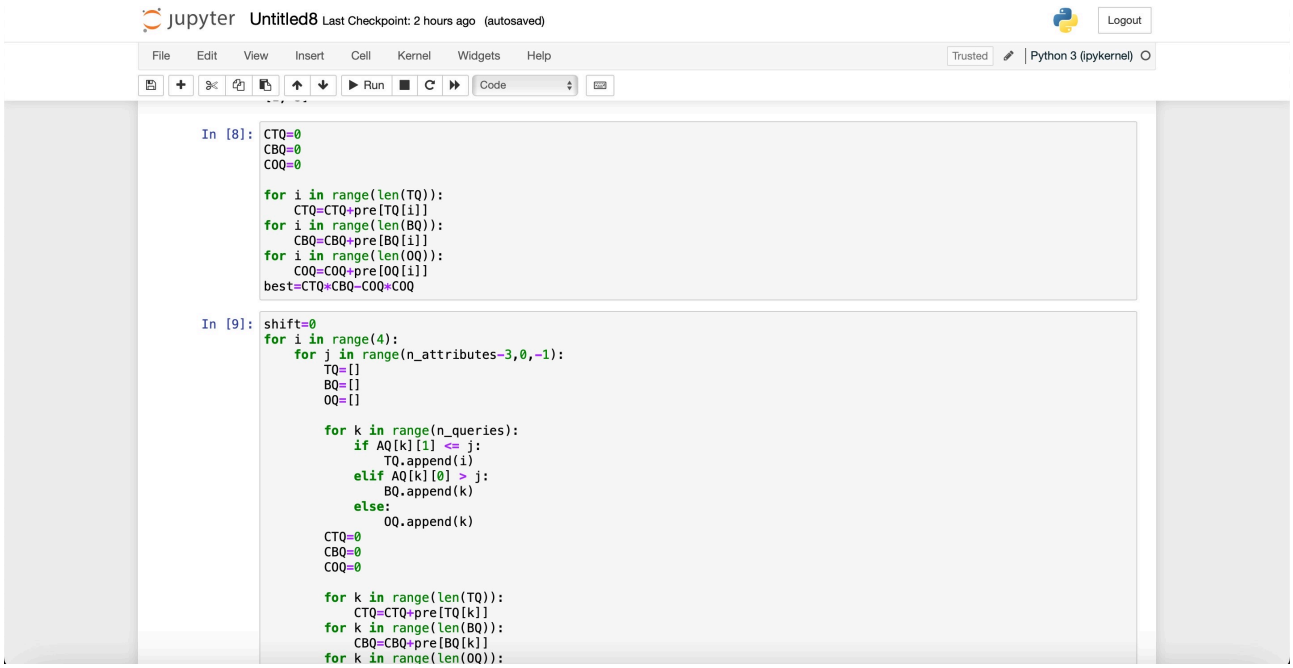
```
start=n_attributes-2
aum = [[1,0,1,0],[0,1,0,1],[0,1,1,0],[0,0,1,1]]
AQ=[]
for i in range(n_attributes):
    row=[]
    for j in range(n_attributes):
        if aum[i][j]==1:
            row.append(j)
    AQ.append(row)

print(AQ)
```

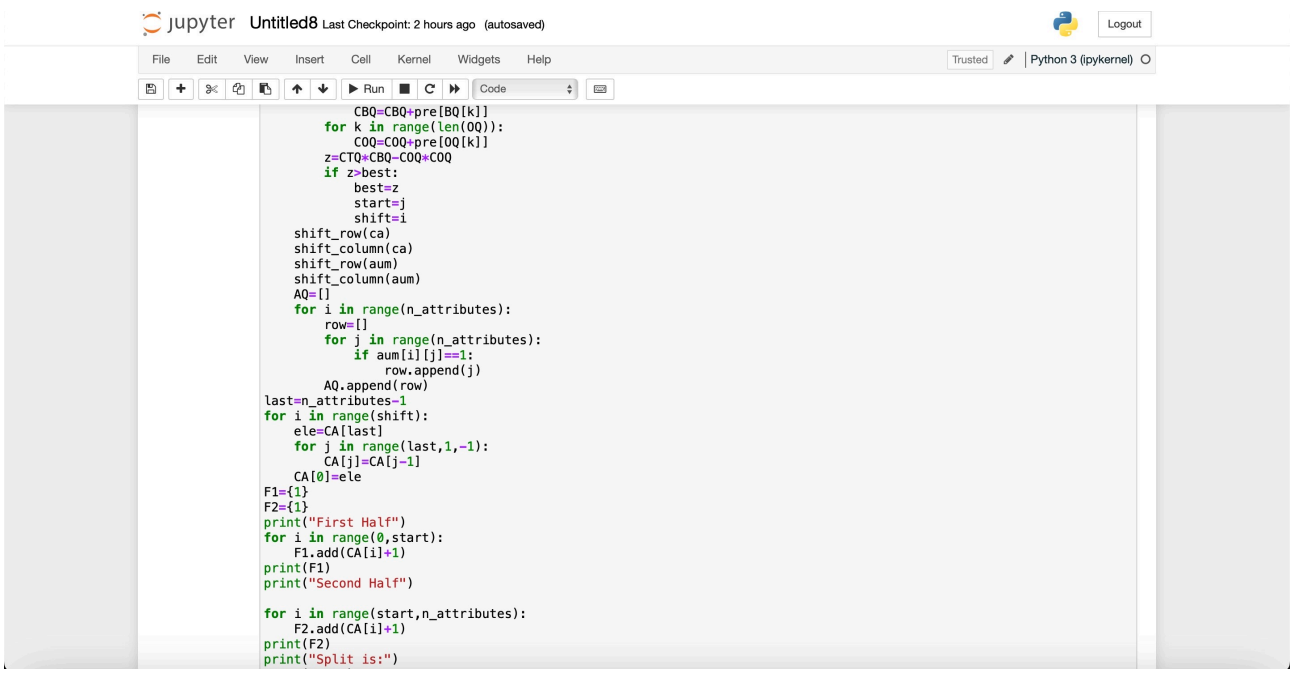
```
[[0, 2], [1, 3], [1, 2], [2, 3]]
```

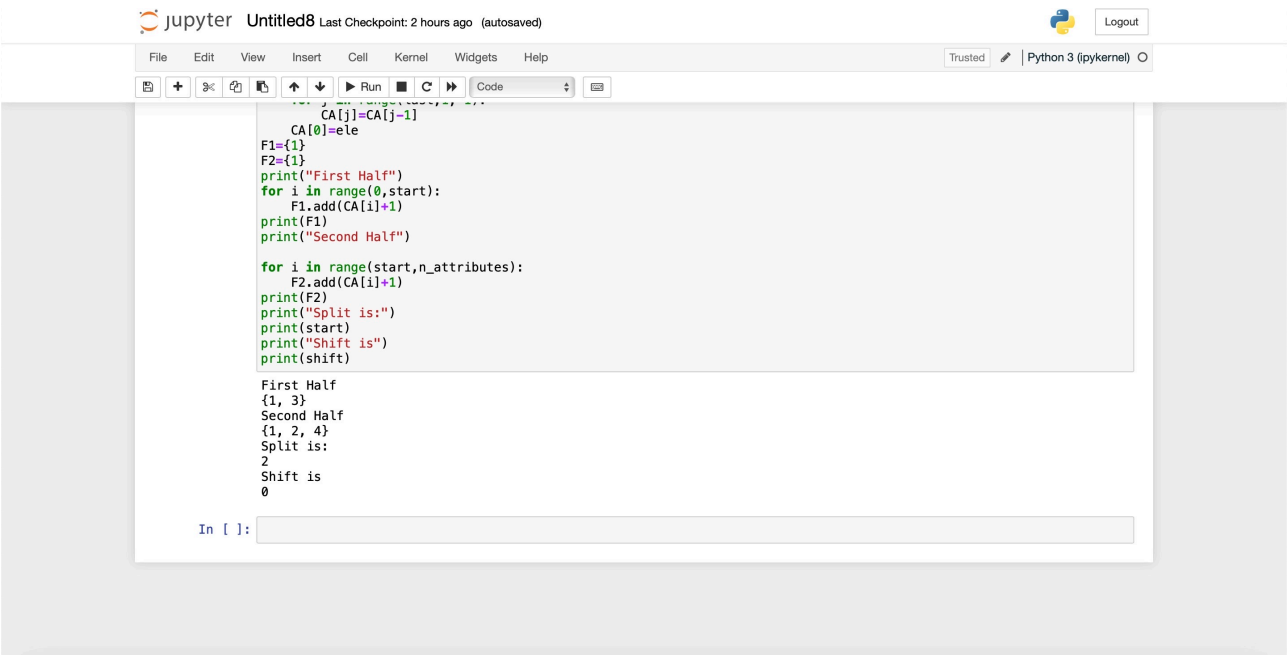
In [7]:

```
TQ=[]
BQ=[]
OQ=[]

for i in range(n_queries):
    if AQ[i][1] <= start:
        TQ.append(i)
    elif AQ[i][0] > start:
        BQ.append(i)
    else:
        OQ.append(i)


print(TQ)
print(BQ)
print(OQ)
```

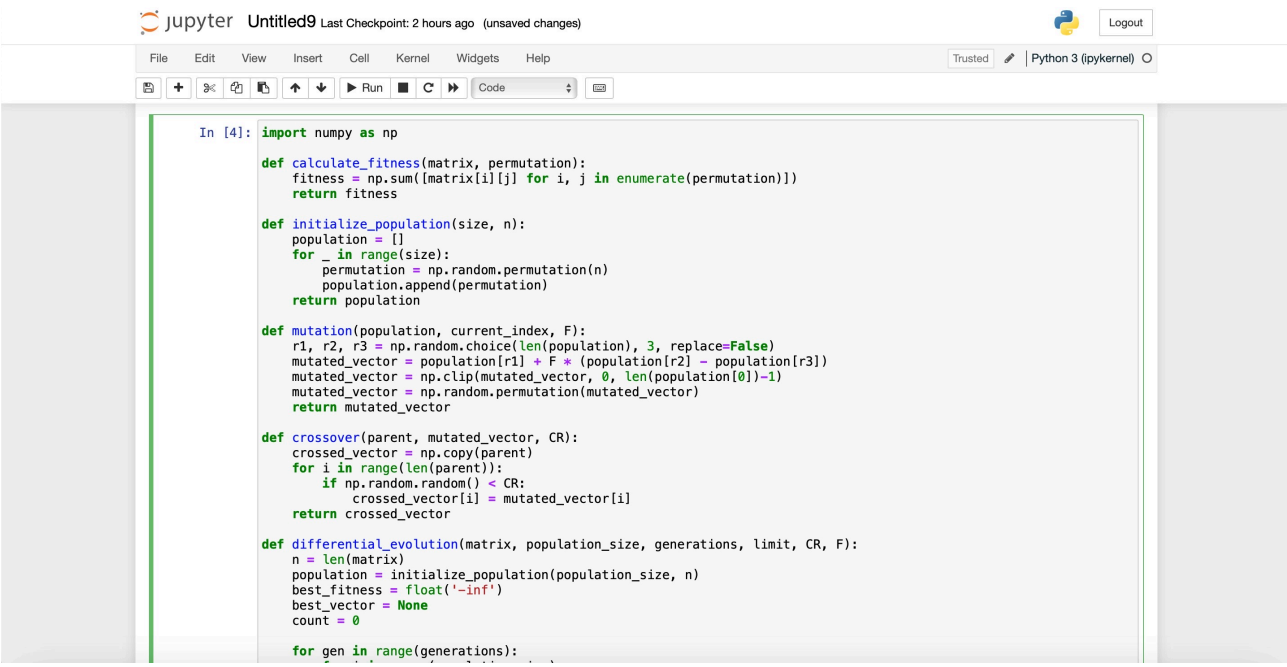
```
[0, 2]
[]
[1, 3]
```

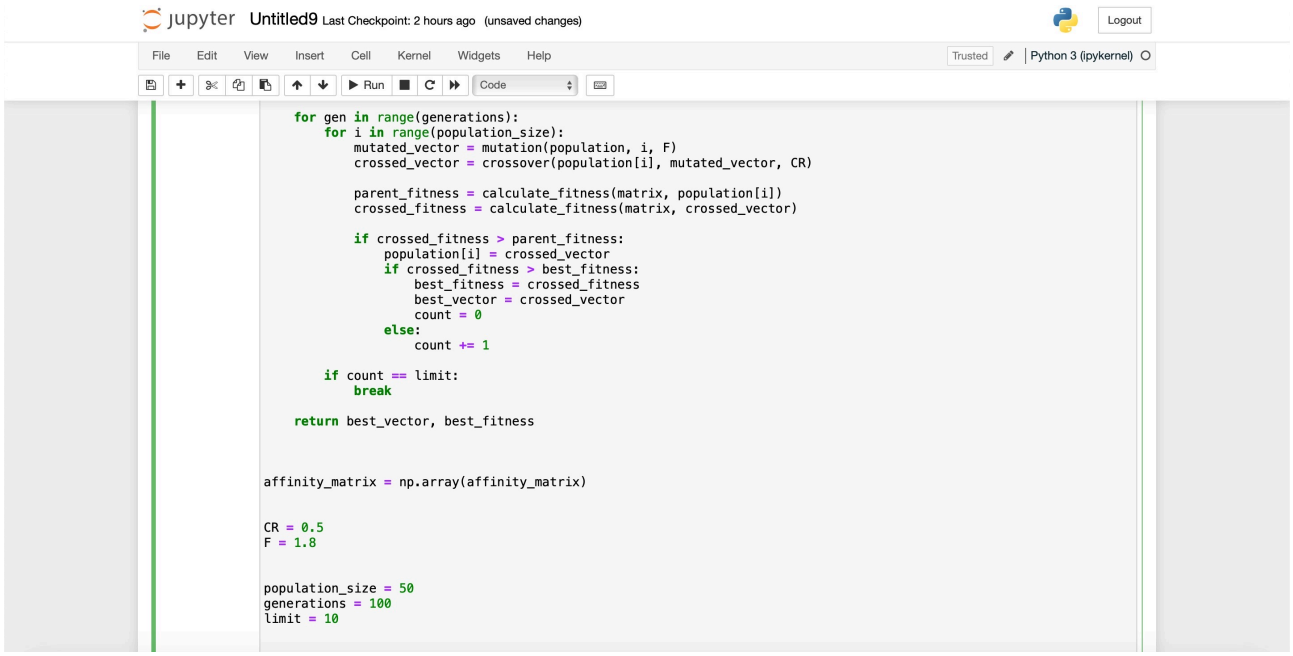

jupyter Untitled8 Last Checkpoint: 2 hours ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
In [8]:
CTQ=0
CBQ=0
COQ=0
for i in range(len(TQ)):
CTQ=CTQ+pre[TQ[i]]
for i in range(len(BQ)):
CBQ=CBQ+pre[BQ[i]]
for i in range(len(OQ)):
COQ=COQ+pre[OQ[i]]
best=CTQ*CBQ-COQ*COQ
In [9]:
shift=0
for i in range(4):
for j in range(n_attributes-3,0,-1):
TQ=[]
BQ=[]
OQ=[]
for k in range(n_queries):
if AQ[k][1] <= j:
TQ.append(i)
elif AQ[k][0] > j:
BQ.append(k)
else:
OQ.append(k)
CTQ=0
CBQ=0
COQ=0
for k in range(len(TQ)):
CTQ=CTQ+pre[TQ[k]]
for k in range(len(BQ)):
CBQ=CBQ+pre[BQ[k]]
for k in range(len(OQ)):


jupyter Untitled8 Last Checkpoint: 2 hours ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
CBQ=CBQ+pre[BQ[k]]
for k in range(len(OQ)):
COQ=COQ+pre[OQ[k]]
z=CTQ*CBQ-COQ*COQ
if z>best:
best=z
start=j
shift=i
shift_row(ca)
shift_column(ca)
shift_row(aum)
shift_column(aum)
AQ=[]
for i in range(n_attributes):
row=[]
for j in range(n_attributes):
if aum[i][j]==1:
row.append(j)
AQ.append(row)
last=n_attributes-1
for i in range(shift):
ele=CA[last]
for j in range(last,1,-1):
CA[j]=CA[j-1]
CA[0]=ele
F1={1}
F2={1}
print("First Half")
for i in range(0,start):
F1.add(CA[i]+1)
print(F1)
print("Second Half")
for i in range(start,n_attributes):
F2.add(CA[i]+1)
print(F2)
print("Split is:")

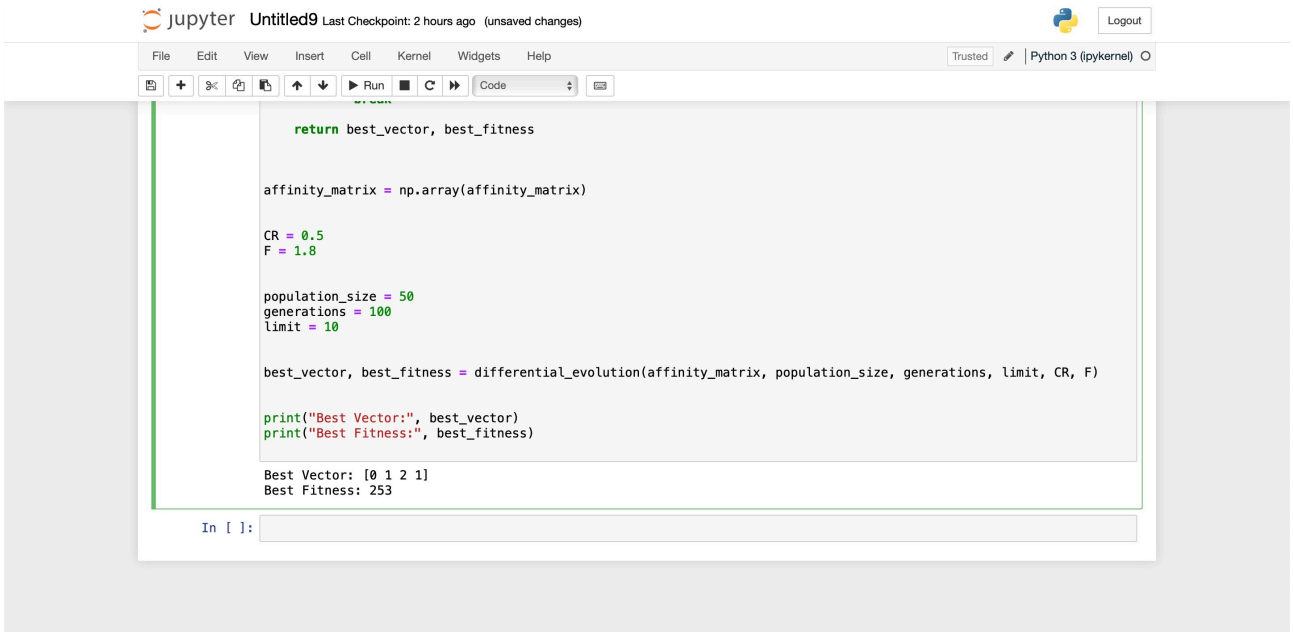


- **Implementation of DBEA using Jupyter**






```
    for gen in range(generations):
        for i in range(population_size):
            mutated_vector = mutation(population, i, F)
            crossed_vector = crossover(population[i], mutated_vector, CR)

            parent_fitness = calculate_fitness(matrix, population[i])
            crossed_fitness = calculate_fitness(matrix, crossed_vector)

            if crossed_fitness > parent_fitness:
                population[i] = crossed_vector
                if crossed_fitness > best_fitness:
                    best_fitness = crossed_fitness
                    best_vector = crossed_vector
                    count = 0
                else:
                    count += 1

        if count == limit:
            break

    return best_vector, best_fitness


affinity_matrix = np.array(affinity_matrix)


CR = 0.5
F = 1.8


population_size = 50
generations = 100
limit = 10
```




```
    return best_vector, best_fitness


affinity_matrix = np.array(affinity_matrix)


CR = 0.5
F = 1.8


population_size = 50
generations = 100
limit = 10


best_vector, best_fitness = differential_evolution(affinity_matrix, population_size, generations, limit, CR, F)


print("Best Vector:", best_vector)
print("Best Fitness:", best_fitness)
```

```
Best Vector: [0 1 2 1]
Best Fitness: 253
```

In [ ]:

RESULTS

The performance of both the algorithms has been measured on the basis of Global Affinity Measure (GAM). The algorithm that gives Maximum Global Affinity Measure is termed as the better algorithm. It has been observed that as population size increases, DBE tends to give better GAM. This is because increasing population size adds more exploration capability in the algorithm. Exploiting the good explored solutions may increase the possibility of getting optimal solution. Increasing number of generations may also lead towards better solutions but may stagnate at a point of time. It can be concluded that for the larger values of ‘n’, DBEA algorithm gives better results than the traditional BEA. We can also find out the mean difference of both the algorithms to find out the efficiency of both of them . The maximum mean means maximum efficiency.

CONCLUSION

In conclusion, the differential bond energy algorithm provides a powerful tool for optimizing the vertical fragmentation of distributed databases. By using this algorithm, it is possible to achieve significant improvements in query performance and data management. However, the fragmentation scheme should be carefully selected and optimized to balance the benefits of fragmentation with the overhead of managing the fragments. By incorporating mutation and crossover operators, it introduces diversity and exploration in the search space, leading to potentially improved results compared to the traditional bond energy algorithm. The efficiency of the differential bond energy algorithm depends on several factors, including the quality of the fitness function, the parameter settings (such as crossover rate, scale factor, population size, and number of generations), and the characteristics of the database being fragmented. In summary, the differential bond energy algorithm offers a promising approach for optimal vertical fragmentation in distributed databases. By leveraging the benefits of both bond energy and differential evolution, it provides a more robust and effective solution for achieving efficient fragmentation.

REFERENCES

[1] Mehta, S., Agarwal, P., Shrivastava, P., & Barlawala, J. (2022). Differential bond energy algorithm for optimal vertical fragmentation of distributed databases. *Journal of King Saud University-Computer and Information Sciences*, *34*(1), 1466-1471.

[2] https://www.ques10.com/p/17611/bond-energy-algorithm-1/

[3] Pérez, J., Pazos, R., Frausto, J., Romero, D., & Cruz, L. (2000). Vertical fragmentation and allocation in distributed databases with site capacity restrictions using the threshold accepting algorithm. In *MICAI 2000: Advances in Artificial Intelligence: Mexican International Conference on Artificial Intelligence, Acapulco, Mexico, April 11-14, 2000. Proceedings 1* (pp. 75-81). Springer Berlin Heidelberg.