I Description

The genetic algorithm is used to optimize the order of table joins. Such as query statement:

select attr from t1 natural join t2 natural join t3.

Different connection sequences among t1, t2, and t3 will get different costs, and the connection method with the smallest cost should be selected as the connection sequence.

II Algorithm

1. Create the ConnectorOptimizer class as the connection optimizer, create the ConnectorNode class as the connection node (gene), and create the Individual class as the individual structure. The structures of the two are as follows:

```
class ConnectorNode {
    //用于进行优化的连接结点
public:
    map<string, int> Vs;//记录需要使用到的属性V值
    int T;//表的T值，即元组数
    string tableName;//表名
    void showInfo();
};
```

```
class Individual{
    //个体结构体
public:
    vector<ConnectorNode*> cnodeNames;
    long long int cost;//当前排序代价
    vector<int> order;
    Individual(vector<ConnectorNode*>names,vector<int>order);
};
```

2. Create the calConnect(vector<ConnectorNode*>& connect) function in the singleton tool class SqlGlobal to calculate the estimated cost required by the connect sequence.

3. First call the findConnect() function to get the first connect node, create a vector container storing ConnectorNode* and call the findConnectTable() function to encapsulate the connection information as ConnectorNode* and put it into the container, then create a ConnectorOptimizer optimization with the container as the construction parameter optimizer object and call its optimizer() function to perform optimization.

4. Obtain the optimal connection scheme calculated by the optimizer through the genetic algorithm and call the updateConnectOrder() function to update the connection order in the order of root traversal.

III Genetic Algorithm

1. Call the getPoolSize() function to calculate the population size, that is, the number of connection sequences to be initialized, and call getRandomIndividual() to obtain a random array of the population size, each random array is a table connection sequence.

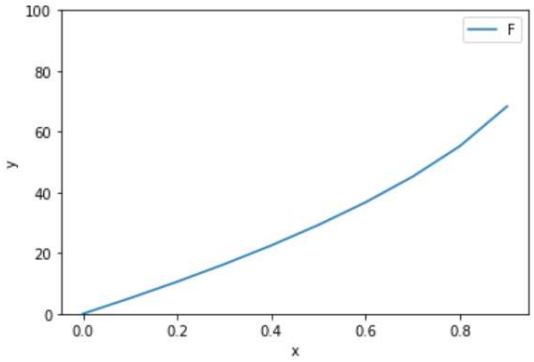
2. Use random arrays to create Individual objects, and call calConnect() method of SqlGlobal in its constructor to calculate the cost of the connection.

3. When all individuals are generated, the sort function is called on the individual container, and the attribute cost is used as the comparison value to sort in ascending order. So far, the initialization of the genetic algorithm is completed.

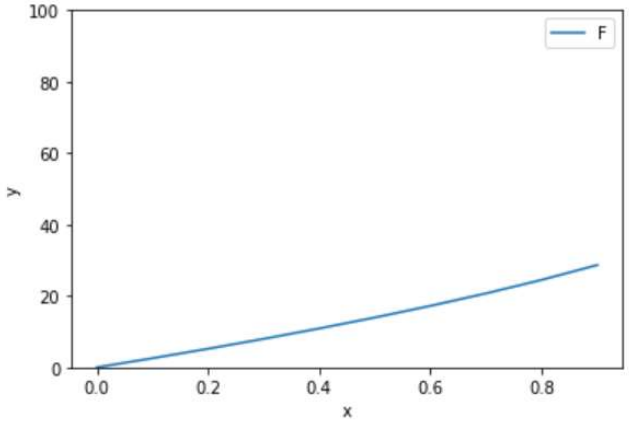
4. Set the evolutionary generation equal to the population size. Call getChoosenIndex() twice to get the chosen parent, and the selection formula used is:

$$f(geo\_rand) = \begin{cases} \max * \dfrac{bias - A}{2 * (bias - 1)} (A < 0) \\ \max * \dfrac{bias - \sqrt{A}}{2 * (bias - 1)} (A \geqslant 0) \\ 其中, A = bias * bias - 4 * (bias - 1) * geo\_rand \end{cases}$$

Set max=100, bias=2, and use geo_rand as an independent variable to draw the following image:



Set max=100, bias=4, and use geo_rand as an independent variable to draw the following image:



We can see that as the bias increases, the function has a greater possibility to take a smaller value, so the top-ranked individuals are more likely to be selected.

5. Call the getChild() function to cross the two parents to get offspring, and the hybridization method adopts position-based hybridization.

6. Call the variation() function to mutate the generated offspring.

7. Add the final offspring to the individual container, call the sort function to sort in ascending order, and discard the last element. Repeat the above steps, and obtain the first individual of the container after the evolution, then it is the optimal solution.