Advanced Algorithm and Programming Method 2 project

The automated data warehouse must allow the user to specify - entities in the database, assuming that said entities will correspond to columns in tables that have the same name as the entity itself - facts, that are numerical entities that will be added in the query - tables connecting such entities, specifying which tables have which entities as their column for example, your data warehouse might have entities.

- shop, city, region, day, month, year, product, category a single fact

- sales

and tables

- Transactions, containing sales, product, shop, day.

- Time, containing day, month, year.

- Product, containing product, category.

- Location, containing shop, city, region.

The data warehouse must get a specification of a report in terms of entities to be displayed and produce the SQL query obtaining the data for the report. Assume all queries are by equality on the same-named entities in different tables.

For example, assuming the specification provided above, assuming you want the total monthly sales split up by region, you would provide - "sales, region, month" as input, and the data warehouse should provide the SQL string "SELECT SUM (Transactions.sales), Time.month, Location.region FROM Transactions, Time, Product WHERE Transactions.day = Time.day, Transactions.shop= Location.shop"

The system should automatically find which tables to join to get the entities specified in the report and how to join them.

Before explaining the project, I should list the classes I used.

- Class Date
- Class Product
- Class Location
- Class Transaction
- Class Input
- Class Query

We create 4 classes as tables (**Date, Product, Location, Transaction**) that all of them have the entities are in the question. Also, two operators istream and ostream for input and output the result.

Class Date containing day, month, year.

```
class Date {
   public:
   int day  ;
   int month;
   int year ;

           Date(){
              time_t now  = time(0);
              tm *ltm     = localtime(&now);
              day   = ltm->tm_mday;
              month = ltm->tm_mon ;
              year  = ltm->tm_year + 1900;}

           Date(int m, int d, int y){
              month = m;
              day   = d;
              year  = y;}

           friend ostream& operator<<(ostream& os, const Date& dt){
              os << dt.year << '/' << dt.month << '/' << dt.day;
              return os;}

   friend istream& operator>> (istream& is, Date& dt){
      is>> dt.year >> dt.month>> dt.day;
      return is;}
};
```

Class Product containing product, category

```
class Product {
   public:
   string product ;
   string category;

   Product() {
      product  = "";
      category = "";}

   Product(string prdct, string ctgry) {
      product  = prdct;
      category = ctgry; }

   friend ostream & operator << (ostream & os, const Product & p) {
      os << p.product << ' ' << p.category << endl;
      return os;}

   friend istream & operator >> (istream & is, Product & p) {
      is >> p.product >> p.category ;
      return is;}
};
```

Class Location containing shop, city, region.

```
class Location {
   public:
   string shop;
   string city ;
   string region;

   Location() {
      shop     = "";
      city     = "Venice";
      region   = "Europe"; }

   Location(string shp, string cty, string rgn) {
      shop     = shp;
      city     = cty;
      region   = rgn;}

   friend ostream & operator << (ostream & os, const Location & l) {
      os << l.shop << ' ' << l.city << ' ' << l.region << endl;
      return os;}

   friend istream & operator >> (istream & is, Location & l) {
      is >> l.shop >> l.city >> l.region ;
      return is;}

};
```

.

Transactions, containing sales, product, shop, day.

```
class Transactions {
   public:
   string   sales;
   int      productId;
   string   shop;
   int      dateId ;

   Transactions() {
      sales       = "";
      productId   = 1;
      shop        = "";
      dateId      = 1; }

   Transactions(string sls, int prdctId, string shp, int dateId) {
      sales       = sls;
      productId   = prdctId;
      shop        = shp;
      dateId      = dateId;}

   friend ostream & operator << (ostream &os, const Transactions &t) {
      os << t.sales << ' ' << t.shop << endl;
      return os;}

   friend istream & operator >> (istream & is, Transactions & t) {
      is >> t.sales >> t.productId >> t.shop >> t.dateId ;
      return is;}
};
```

In class **Input** we first have a function introduce which is just tell us some information about how we should insert our data.

```
void Introduce() {
      cout << "Hello \nThis is a final project for AA2\nPlease follow the instruction:\n";
      cout << "1-Please Enter the number of tables" << endl;
      cout << "2-Please Enter names for each table follow by their columns\n";
      cout << "3-At the End please enter the desired input to get your query!" << endl;
      for (int ind = 0 ; ind < TERMINALMAXCOL ; ++ind) cout << "-";
      cout << endl;
   }
```

And after that **GetInput function** get the number of tables, name of tables, and the number of columns for tables from input and pushback it to **table vector.** Also, for saving column we use by defining adjacency vector in vector like a graph. Each table has an index and by using index and with **Red Black Theory** which is usable in binary search, we can access to correct table that we want.

```
void GetInput(){
      cout << "Please Enter number of tables : \n";
      cin >> tableCount ;

      for(int ind = 0 ; ind < tableCount ; ++ind) {
         cout << "Well Please Enter Table Name : \n" ;
         string table ; cin >> table;
         tables.push_back(table);

         indexes[table] = ind+1;

         cout << "Please Enter count of column's " + table << endl;
         int columnsCount ; cin >> columnsCount;
         for(int columndId = 0 ; columndId < columnsCount ; columndId++) {
            cout << "Enter column " << columndId+1 << " : " ;
            string column ; cin >> column ;
            adj[ind].push_back(column);
         }
         for(int i = 0 ; i < TERMINALMAXCOL ; ++i) cout << "-" ;
         cout << endl;
      }
   }

   friend ostream & operator << (ostream &os, Input &i) {
      return os;
   }
};
```

For class **query** which is the main part of project and has a lot of function inside, we have bool visited variable which move on adjacency matrix for survey and match the equal columns.

toLowerString function convert all character to lowercase because sensitively and after that put it in our data structure.

```
string toLowerString(string q) {
      string query = q;
      for (int ind = 0 ; ind < query.size() ; ind ++) {
            query[ind] = tolower(query[ind]);
      }
      return query;
   }
```

DFS function is for finding leaves of graphs which it means columns of each table.

```
void DFS(int u) {
      visited[u] = true;
      for(int ind = 0 ; ind < in.adj[u].size() ; ind ++) {
         int v = in.indexes[in.tables[ind]];
         if (!visited[v]) {
            DFS(v);
         }
      }
   }
```

isSpecialKeyword check the keyword used for query is inside query keywords or not.

```
bool isSpecialKeyword(string query) {
      for (int ind = 0 ; ind < SQLKEYWORDCOUNT ; ind++) {
         string key = toLowerString(keywords[ind]);
         query = toLowerString(query);
         if(key == query) {
            return true;
         }
      }
      return false;
   }
```

List of the all keywords we use as a list of SQL define in the project as **Query::keywords**.

```
const string Query::keywords [] = {
   "CREATE",   "PRIMARY KEY",   "INSERT",   "INTO",   "SELECT",   "FROM",   "ALTER",   "DROP",
   "ADD",   "DISTINCT",   "UPDATE",   "SET",   "DELETE",   "TRUNCATE",   "AS",   "ORDER BY",
   "ASC",   "DESC",   "BETWEEN",   "WHERE",   "AND",   "OR",   "NOT",   "LIMIT",   "IS NULL",
   "DROP COLUMN",   "DROP DATABASE",   "DROP TABLE",   "GROUP BY",   "HAVING",   "IN",   "JOIN
",   "UNION",   "UNION ALL",   "EXISTS",   "LIKE",   "CASE"
};
```

findTableIndex which is one of sub-methods we use in the project is duty to find table indexes and access to adjacency list.

```
int findTableIndex(string q) {
      string query = toLowerString(q);
      for (int ind = 0 ; ind < in.tables.size() ; ind++) {
         if (query == in.tables[ind]) {
            // index of table
            return ind;
         }
      }
      // in case if query does not in tables !
      return -1;
   }
```

findTableColumn find column of table when table was found by findTableIndex.

```
pair<string, string> findTableColumn(string q) {
      string query = toLowerString(q);

      for (int ind = 0 ; ind < in.tables.size() ; ind++) {

         for(int columnId = 0 ; columnId < in.adj[ind].size() ; columnId++) {
            if (query == in.adj[ind][columnId]) {
               return make_pair(in.tables[ind],in.tables[ind] + "." + in.adj[ind][columnId]);
            }
         }
      }
      return make_pair("", "");
   }
```

splitQuery split each keyword of input into a single string. we need to split each key into a vector. If size of splits vector is less than 1 then input is invalid.

```
vector <string> splitQuery(string input) {
      istringstream iss(input);
      vector<string> results((istream_iterator<string>(iss)), istream_iterator<std::string>();
      return results;
   }
```

replaceToSpace replace all comma in input to a space. First thing we need to parse input into list of string. For example, if someone insert total, monthly, sales as a input of query. This function replace all comma to space because comma is not a SQL keyword.

```
string replaceToSpace(string input) {
      string in = input;
      for(int ind = 0 ; ind < in.size() ; ind ++) {
         if(in[ind] == ','){
            in[ind] = ' ';
         }
      }
      return in;
   }
```

makeQuery is the main logic of the program. So first we insert a select by addSelect() to query because all queries start by select. After select we need an operator for query. So, by **mathKeywords** which we choose optionally 4 operators we can choose the operator and also as default it choses Sum and go to next word.

```
string makeQuery(string input) {
      string ret = addSelect();

      input = replaceToSpace(input);
      vector <string> splits = splitQuery(input) ;
      if (splits.size() < 1) {
         return "input is invalid, please enter a valid input!";
      }

      int mathKeywordInd = -1;
      string mathKeywords[] = {"AVG", "MAX", "MIN", "SUM"};
      for (int ind = 0 ; ind < 4 ; ++ind) {
         if(mathKeywords[ind] == splits[0]) {
            mathKeywordInd = ind ;
            break;
         }
      }
```

Next make from as second part of query and go to tables and find the column needs.

```
    bool isFirstMath = false;
    int wordInd = 0;
    vector <string> tableToAdd ;
    if (mathKeywordInd > -1) {
       ret         += mathKeywords[mathKeywordInd] ;
       isFirstMath = true;
    }
    else {
       ret                        += "SUM" ;
       ret                        += "(";
       pair<string, string> temp  = findTableColumn(splits[0]);
       tableToAdd.push_back(temp.first);
       ret += temp.second;
       ret += "),";
       wordInd ++ ;
    }

    if (isFirstMath) {
       wordInd = 2 ;
       ret += "(";
       pair<string, string> temp = findTableColumn(splits[1]);
       tableToAdd.push_back(temp.first);
       ret += "),";
    }

    for (; wordInd < splits.size() ; wordInd++) {
       ret += "(";
       pair<string, string> temp = findTableColumn(splits[wordInd]);
       ret += temp.second;
       tableToAdd.push_back(temp.first);
       ret += ")";
       if (wordInd < splits.size()-1 ) {
          ret += ",";
       }
    }

    ret += addFrom();

    for(int tableInd = 0 ; tableInd < tableToAdd.size() ; tableInd ++) {
       ret += tableToAdd[tableInd];

       if (tableInd < tableToAdd.size()-1) {
          ret += ", ";
       }
    }
```

After that use **addWhere** to insert this keyword to query.

```
ret += addWhere();
ret += findSubQuery(tableToAdd) ;
ret += addSemicolon() ;
```

findSubQuery function is for finding the condition we need for statement. It finds the equal columns from different table and matched them as a condition to show us result.

```
string findSubQuery(vector <string> tableToAdd) {
      int numberOfEquality = tableToAdd.size()-1;
      string ret = "";
  for(int tableInd = 1 ; tableInd < tableToAdd.size() ; tableInd++) {
   for(int columnId = 0 ; columnId < in.adj[in.indexes[tableToAdd[tableInd]]-1].size() ; columnId ++ ) {
     for(int firstTableInd = 0 ; firstTableInd < in.adj[in.indexes[tableToAdd[0]]-1].size() ; firstTableInd++) {

if(in.adj[in.indexes[tableToAdd[0]]1][firstTableInd] == in.adj[in.indexes[tableToAdd[tableInd]]1][columnId]) {
                  ret += tableToAdd[0] + "." + in.adj[in.indexes[tableToAdd[0]]-1][firstTableInd];
                  ret += "=";
                  ret += tableToAdd[tableInd] + "." + in.adj[in.indexes[tableToAdd[tableInd]]-1][columnId];
                  if (numberOfEquality > 1) {
                     ret += ",";
                  }
                  numberOfEquality -- ;
                  if (numberOfEquality == 0) {
                     return ret ;
                  }
                  break;
               }
            }
         }
      }
      return ret;
   }
```

and at last set a semicolon to query.

```
ret += addSemicolon() ; // add semicolon to query
     // and the answer is here !
     return ret;
```

This is main() and how we can import information from file.

```
int main() {
   /* you can also import input file
      please make sure put your input file in same directory !
   */
   freopen("input.txt", "r", stdin);
   Input in ; // get data from user and insert into input class

   in.Introduce(); // some introduction messages
   in.GetInput();

   cout << "Please Enter input to generate query : " << endl;
   string input ;
   getline(cin, input);
   getline(cin, input);

   cout << input << endl;

   Query q (in);
   string ans = q.makeQuery(input);
   cout << ans << endl;

   return 0;
}
```