DEGREE OF MASTER OF SCIENCE

Database Systems Implementation

Hilary Term 2018

Candidate number: 1019470

Overview

This reports consist of three parts. In Part A, detailed analysis on CoverDB without GROUP-BY clause will be given. In Part B, CoverDB for answering queries with GROUP-BY will be explained. For each part, the pseudo code will be given first. Analysis on the complexity and performance of the algorithm will also be given. Then the implementation of CoverDB in C++ will be explained in a top-down approach. Finally in part C testing results will be shown.

Part A The Design of CoverDB without GROUP-BY Clause

1 Introduction

CoverDB is a database that computes aggregates over covers of relations. As described in the exam paper, a cover of a relation satisfies the recoverability property, which means a relation can be recovered by taking the natural join of the projections of a subset of the relation onto the schemas defined by a decomposition[1]. Therefore, the problem of computing aggregates over the subset of the relation can be considered as aggregating over the result of the natural join.

In this project, the idea of factorised database is adopted to represent the join results and aggregates are performed over this representation[2]. In order to achieve this, the algorithm first builds a variable order given the decomposition, which guides how the factorised database is built. Queries with and without group-by clause are both taken into consideration. Different aggregate s can lead to different variable orders which will be explained in detail later.

CoverDB implements the idea in existing research literature including hyper tree decompositions[1], variable orders[3] and factorised database[2]. There are some simplifications in this algorithm. The first three are as stated in the exam paper. First, the tables do not have duplicates. Second, the datatype of all values are integers. Third, the relation can be kept in main memory. Fourth, the variable order is represented by a binary tree and the algorithm produces one possible order for the give query. Fifth, the factorised database is represented according to the variable order therefore each node has at most two types of children. Sixth, the variable order is considered as known at the compile stage for aggregates over group-by clause.

Strategies based on depth-first-search and breadth-first-search are proposed to increase the probability of building a variable order in one pass in different situations. As the algorithm only produces one possible variable order for each query, the strategies proposed here are not based on current research projects. Methods are also proposed to determine the root in a possible variable order. A map container is used to accelerate the sorting process. It is shown in the benchmark that these strategies speed up the building of variable orders.

2 Pseudo Code

Below is the pseudo code of CoverDB. The inputs are an aggregate query Q without any group-by clause and a cover (K, C) of a relation R where the decomposition C is acyclic. The output is the answer of Q over relation R.

(a) CoverDB

CoverDB calls three main sub-functions: BuildVariableOrder, BuildFactorisedDB and Sum. First, BuildVariableOrder takes the query and the cover and build the variable order accordingly. A variable order Δ is returned, which is defined as a tree and will be explained later. Here the simplification is made such that it returns only one possible variable order given the query. Details on strategies will be explained later. Second, BuildFactorisedDB uses the variable order built in the last step and the cover K to build a factorised database. Every tuple in the given cover is inserted into the factorised database according to the variable order. For example, the variable order shows the order of attributes given in K. Then the tuples inK will be inserted to factorised database according to this order. Third, function Sum takes the factorised database and the query, and returns the answers to the query via the database, which uses memory liner to the side of K.

```
  CoverDB (Q,K,C,R)
1 Δ ← BuildVariableOrder(Q,C)
2 FDB ← BuildFactorisedDB(K,Δ)
3 return Sum(Q,FDB)
```

(b) BuildVariableOrder

Below gives the pseudo code for function BuildVariableOrder. It starts by building a relation map which gives is a map from each attribute to the related attributes. For example, with decomposition {A,B,C}, we can build a map from A to {B,C}. Then the variable order would be built based on the map so that the properties of variable order are preserved.

```
  BuildVariableOrder(Q,C)
1 map ← BuildRelationMap(C)
2 Δ ← InsertAttributes(map,C,Δ,encode)
```

Function BuildRelationMap takes the decomposition C and build a mapping for each relation. Every attribute is mapped to a set of attributes that are in the same relation. There are no repeating attributes in the set. The pseudo code is given below.

```
   BuildRelationMap(C)
1  For every relation in the decomposition
2       For every attribute in the relation
3           map the attribute to all the other attributes in this relation
```

The second function used in BuildVariableOrder is InsertAttributes where the map is used to create the variable order. The inputs are the mapping created before, the decomposition, the root of variable order to store information and a vector called encode to store positions of each attribute in the order for easy access. There are two main properties of variable orders [OZ15]. The first one is that attributes of the same relations are in the same path of the tree. For for every child B of a node A, the key of B should be included in A and its keys. Attributes are inserted in this way so that the relations can be represented for further analysis. In CoverDB, creating the variable order can link the cover to the relation by joining the cover based on the variable order. Pseudo code is given below.

```
InsertAttributes(map,C,Δ,encode)
1 For each attribute a
2   If Δ is empty
3       insert attribute
4   ELSE
5       IF the attribute is already in the order
6           continue
7       ELSE
8           For each attribute b in the same relation
9               IF attribue b is in the order
10                  IF b has no left child
11                      Insert attribute a to the left
12                  ELSE IF b has no right child
13                      Insert attribute a to the right
14                  ENDIF
15              ENDIF
16      ENDIF
17   ENDIF
```

(c) BuildFactorisedDB

Below gives the pseudocode for function BuildFactorisedDB. It reads one tuple at a time from the cover and insert it into the factorised database which follows the variable order from the last step. Details of the function InsertTuple will be explained later.

```
  BuildFactorisedDB(K,Δ)
1 FOR tuple in K
2   InsertTuple(tuple,Δ)
```

(d) Sum

Below gives the pseudocode for function Sum, which takes a query without group-by clause and computes the sum according to the factorised database. There are two types of queries this function can take. One is "SUM(1)" which gives the total number of tuples in the cover. The second query is SUM(EXPR) where EXPR is one of the attributes in R. Details of the sum operations for each case will be explained later.

```
  Sum(Q,FDB)
1 If Q contains "SUM(1)"
2   COUNT aggregate over all nodes in FDB
3 ELSE
4   COUNT aggregate over the given attribute
```

3 Analysis

The following gives the comparison of CoverDB and StandardDB as well as the relation between the time and space complexity and the size of the input cover. Further analysis is presented on the benefits of CoverDB.

For CoverDB, it needs to store distinct data in the factorised database. The extra memory used and time used are both linear to the size of the input cover: $f = a * n$ where a is the number of attributes and n is the number of tuples in the cover.

CoverDB is superior to StandardDB in terms of storage, performance, data and instruction locality. Both databases perform one scan over their input to answer non group-by aggregates. However, the size of cover can be much smaller than the relation if there are plenty of duplicates in the data, the time and space complexity are much lower for CoverDB which only needs to store and scan tuples in the cover, and therefore memory and time usage can be much lower when there are enough duplicates in the dataset. Data and instruction locality is better in CoverDB since it does a single scan over the cover relation.

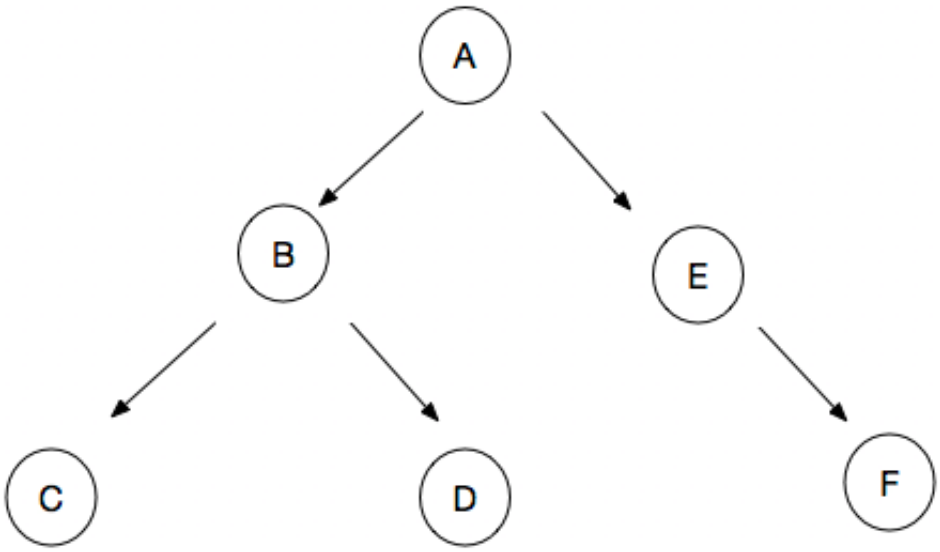
CoverDB is able to reach constant delay in this case by having EXPR attribute above the others. For instance, counting the number of tuples in the relation can be seen as the sum of multiplications over cardinalities in the factorised database. The computation of CoverDB depends on the size of the cover and therefore the gap between standard and cover representations. By Enumerating tuples in the factorised database, we can compute aggregates on the fly. Therefore, the time between listing a tule and listing the next is constant and does not rely on the number of tuples.

4 Example

Here shows the example on a relation R = {A,B,C,D,E,F} with cover (K,C) where C = {{A,B,C},{A,B,D},{A,E},{E,F}}. BuildVariableOrder is called first. A mapping is created as follows.

```
A → {B,C,D,E}
B → {A,C,D}
C → {A,B}
D → {A,B}
E → {A,F}
F → {E}
```

The attributes are inserted in this order: A,B,C,D,E,F using the relations from the mapping. A depth-first-search approach is adopted which will be explained later. The result of the variable order is shown as the below figure.



After creating the variable order, the tuples in the cover will be inserted based on the variable order. The number of tuples in the relation R is computed based on the factorised database where each node has a value 1 and union is considered as summation and cartesian product is considered as products over integers. The calculation is executed in a bottom up approach starting from the leaf node. In this case it would first calculates the number of C and D and computes the products of them which later times with the number of B and assigns to B. Similar calculation applies to E and F. Finally product of the value of B and E is given to the parent A node and summation over all A nodes are performed. For example, if each attribute has two different values, the calculation would be $(2*2*2)*(2*2)+(2*2*2)*(2*2)=2^6$. This result is equal to the number of tuples in R since the permutation of all nodes in the cover.

5 Implementation

The description of CoverDB is given below. As mentioned before, there are mainly three components for CoverDB: BuildVariableOrder, BuildFactorisedDB and Sum. A detailed explanation will be given for each of these functions. Main part of the implementations in C++ will be given for clarification.

(a) BuildRelationMap

The first function BuildVariableOrder consists of two parts. Strategies based on depth-first-search are proposed to increase the probability of building a variable order in one pass. In other words, properties of the variable order are preserved in this implementation. Tests show that the strategies are effective in creating a reasonable variable order in one pass which is convenient to the summation later.

The first part is BuildRelationMap, which builds a mapping from attributes to their related attributes. The input is the decomposition and the output is the map of relations. The decomposition is defined by a 2D matrix, where each row contains one relation. For example, for C = {{A,B,C},{A,B,D},{A,E},{E,F}}, a corresponding matrix would be created which uses standard library vector in C++. Each attribute is represented as a character.

```
 vector<vector<char>> decomposition = {{'A','B','C'},{'A','B','D'},{'A','E'},
{'E','F'}};
```

The algorithm creates a map container that maps an attribute to a set attributes in the relation. The use of set container here is to avoid duplicates since the same pair of attributes may appear in different relations but we only need to record one. For example, 'A' and 'B' appear both in {'A','B','C'} and {'A','B','D'}.

```
 map<char,set<char>> mp;
```

The function iterates over every element in the decomposition and inserts attributes in the same relation into the map. It is noted that within the same relation we need to avoid the attribute mapping to itself. By creating this mapping it would be more convenient for the other parts of the program to check whether two attributes are related.

```
//-------------------------------------------------------------------
// BuildRelationMap
//
// Purpose : Build relation map according to the input decomposition.
// Input   : decomposition - input of CoverDB.
// Output : the map of relations
//-------------------------------------------------------------------
map<char,set<char>> BuildRelationMap(vector<vector<char>> decomposition){
    map<char,set<char>> mp; // a map from one attribute to its relating attributes
    // for every element in the given decomposition
    for (int i = 0; i < decomposition.size(); i++)
        for (int j = 0; j< decomposition[i].size(); j++)
            // find other attributes in the same relation
            for (int k = 0; k < decomposition[i].size(); k++)
                if (j != k ) // not itself
                    mp[decomposition[i][j]].insert(decomposition[i][k]); // insert
into the mapping
    return mp;
}
```

After building the map, variable order is built according to the type of the query. In this implementation, queries are classified into four categories: type 1, type2, type 3 and type 4. Four types of queries are considered here according to the benchmarks given in Question 3. Type 1 considers those with "SUM(1)" that counts the number of tuples in the relation. Type 2 is for those with "SUM(EXPR)" where EXPR is an attribute in the relation. Type 3 considers queries with group-by clause whose attributes are in the same relation. Type 4 queries contain group-by clause but the attributes are not in the same relation. The difference of type 3 and 4 is that, attributes can be connected in the variable order for the former but not the latter. The reason is that, in order to construct a variable order where the group-by variables are above the other variables[BKOZ13], these attributes should be connected. If the query belongs to type 3, the group-by variables can be connected and therefore reaches constant delay. Therefore, we can classify the four queries in Question 3 into four types.

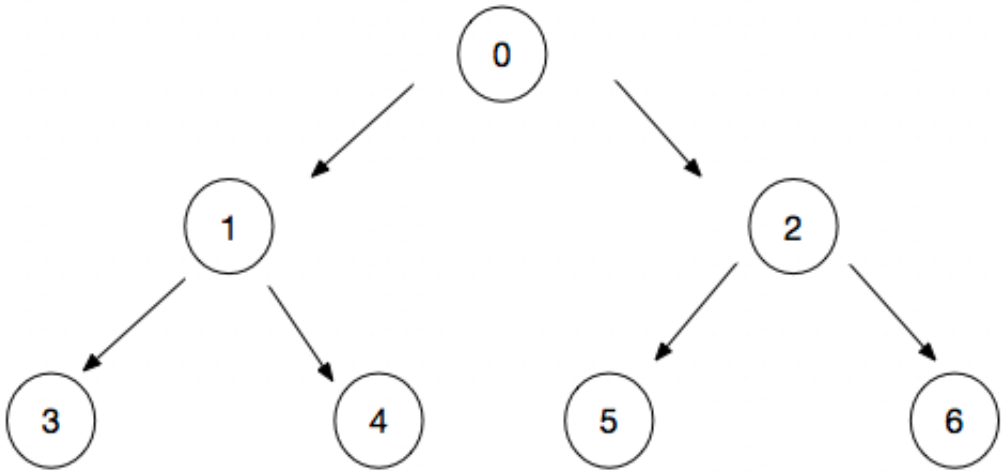
```
 SELECT SUM(1) FROM R; // type 1
 SELECT SUM(A) FROM R; // type 2
 SELECT A,B, SUM(F) FROM R GROUP BY A,B; // type 3
 SELECT C,D, SUM(F) FROM R GROUP BY C,D; // type 4
```

Next the function for determining the type of query QueryType is explained. This function takes in the query and finds the keywords in the query that can be used to determine the type of the query. Here type 3 and 4 cannot be differentiate at this stage because the relations of the attributes cannot be found here. When building the variable order, the relation will be explored and the type of the query can be further analysed. In this function, we only use the keywords. Therefore, with "SUM(1)" we can determine that this query belongs to type 1, whereas with "GROUP BY" they would be type 3 or type 4 (type 3 is used here first for both type 3 and type 4, which will be determined later). The remaining queries where neither of these two key strings are found would be categorised as type 2. C++ implementation is shown below.

```
//------------------------------------------------------------------
// QueryType
//
// Purpose : Determine the type of the query
// Input   : query - string for the input query
// Output :  the type of the query
//------------------------------------------------------------------

int QueryType(string query){
    int type = 0;
    if (query.find("SUM(1)") !=std::string::npos) // SUM(1)
        type = 1;
    else if  (query.find("GROUP BY") !=std::string::npos){ //GROUP BY CLAUSE
        type = 3;
    }
    else{
        type = 2; // SUM(EXPR)
    }
    return type;
}
```

After knowing the type of the query, the building of the variable order is implemented based on it. A structure is deigned to store the information of the variable order. It is noted that simplification is made such that the order is stored in a binary tree instead of a forest as defined. It is sufficient for the test cases given in Question 3. The implementation is similar to the definition of a node in the binary tree. Each node has a value which is the name of the attribute. An integer is defined to record the position of the node in the tree. It is noted that the position of the attribute in the order is recorded using a method similar to building a maximum or minimum heap. A graph of how to record the position of each node is given below. Given the position of the parent node, the position of the children can be computed. For example, for the root whose position is 0, its left child has the position $0 * 2 + 1 = 1$, and the right child is $0 * 2 + 2 = 2$. The same rule applies to all other nodes. In this way each attribute inserted in the variable order will be mapped to an integer which indicates its position in the tree. A vector called encode is used to record these integers. The information is stored in order so that the encode[1] has the position for A and encode[2] has the position for B and so on.




```
//-------------------------------------------------------------------
// VarOrder
//
// Purpose: The unit for variable order
//
// Simplification: the varible order is represented via a binary tree
// instead of a forest
//
// Variables: val - name of the attribute
//            num - position of the attribute in the tree
//            left - a pointer to the left child
//            right - a pointer to the right child
//-------------------------------------------------------------------
struct VarOrder{
    char val;
    int num;
    VarOrder* left;
    VarOrder* right;
    VarOrder(char x): val(x),left(NULL),right(NULL){}
};
```

It is noted that, simplification is made here such that one possible variable order will be built. Strategies are proposed to increase the probability of building a current variable order. In other words, properties of the variable order are preserved in this implementation. Tests show that the strategies are effective in creating a reasonable variable order in one pass which is convenient to the summation later. Current research work on the strategies of building variable order is not referred here. While the implementations here are efficient for test cases in Question 3, they are not designed in a general approach to satisfy other cases. However, they are sufficient to show the idea of creating a variable order given the decomposition, and can be extended to a general version that is able to create all possible variable orders.

For type 1, a function called InsertAttributes is implemented to insert all attributes in R into the variable order. There are four inputs: the map of relations, the decomposition, the root of the variable order and the positions of each attribute in the order. The root of the variable order and the positions are called by reference in order to record the results of building the variable order. The main idea is to insert each attribute based on the mapping created before so that the properties of variable orders are preserved[OZ15]. There are two properties considered here. The first one is that attributes of the same relations are in the same root-to-

leaf path in the order. The second one is that for every child of a node , the key of the child belongs to the set that consists of the node and its keys.

As we can see from the pseudo code shown before, a special case is considered first where there are no attributes in the variable order. In this case the first attribute will be inserted directly. The order is considered such that attributes are inserted to the left of the parent node first. When the left child is occupied, the position of right child would be considered. If not the special case, the function determines whether the attribute is already in the order. No insertion is made in this case. Otherwise, the attribute should be inserted as a child of one of the attributes in the same relation. The parent attribute is locate in the order and the left and right child positions are considered in order. It is noted that the search for parent node is designed in a depth-first-search manner so that the leaf nodes in the order have priority to be chosen. The reason for this strategy is to reduce the case where the left and right child positions of a parent node are both occupied and a position cannot be found in the order. For example, in the given decomposition, E needs be connected to A since F is the only other node that connects to E but it does not connect to other attributes. A position of the child of A should be left for E and it is convenient if not both of the children are set for A before the insertion of E. To be concrete, after the first node A being inserted, B is inserted as a left child of A. Then D can be either the child of A or B, a depth-first-search approach would result in D being the child of B, and therefore a position of the child of A is preserved. The same rule applies for other attributes. Therefore, the depth-first-search strategy is chosen here in order to construct a legal variable order in one pass. The implementation of this strategy is given below in the function FindVarOrder_DFS. It is implemented recursively using an in-order traverse. A vector called res is used to store all the matches.

```
//--------------------------------------------------------------------
// FindVarOrder_DFS
//
// Purpose : Find a node in the variable order in a Depth-First-Search way
// Input   : root - the root of the variable order.
//           relation - the set of relation whose positions are to be found
//           res - the result of finding
// Output :  none
//--------------------------------------------------------------------


void FindVarOrder_DFS(VarOrder*& root,set<char> relation,vector<VarOrder*>& res){
    if (root == NULL)
        return ;
    FindVarOrder_DFS(root->left,relation,res); // left sub-tree

    if (relation.find(root->val) != relation.end()){ // if the root is the target
node
        res.push_back(root);
    }
    FindVarOrder_DFS(root->right,relation,res); // right sub-tree
}
```

After calling FindVarOrder_DFS, a list of the positions for all relational attributes are found. An iteration is run to find a relational attribute that has already been inserted.

The implementation of an example is given below where a new attribute is inserted in the variable order. First a new node is constructed and the position is given. Second the new node is inserted into the variable order and the position is recorded in the vector encode for easy access. The left child position is considered first, and then the right child is considered later.

```
//--------------------------------------------------------------------
// InsertAttributes
```

```
//
// Purpose : Build variable order by inserting all attributes in R by finding
relational attributes in a depth-search-first manner
// Input   : mp - the map of relations.
//           decomposition - input of CoverDB.
//           root - the root of the variable order
//           encode - the vector that contains the positions of all attributes in
the variable order
// Output : none
//-----------------------------------------------------------------------

void InsertAttributes(map<char,set<char>> mp, vector<vector<char>> decomposition,
VarOrder* root,vector<int>& encode){

    map<char,set<char>> :: iterator iter;

    for (iter = mp.begin(); iter != mp.end(); iter ++){
        // first attribute to be inserted into the variable order
        // default positiobn for the first attribute is the left child of the root
        if (root->left == NULL){
            root->left = new VarOrder(iter->first);
            root->left->num =root->num * 2 + 1; // the number for the new node
            encode[iter->first - 'A' + 1] = root->left->num; // put the number of
the node into a vector
            continue; // next attribute

        }

        VarOrder* position = NULL; // the position of the current attribute in the
variable order
        FindVarNode(root->left,iter->first,position); // find attribute in the
order
        if (position == NULL){ // attribute not found: find the attributes it
relates to in the order
            set<char> relation = iter->second;
            set<char> :: iterator siter;
            vector<VarOrder*> res;
            FindVarOrder_DFS(root,relation,res); // find the position of other
attributes in the same relation in a depth-search-first manner
            for (auto pos:res){ // for every attribute in the relation

                if (pos != NULL){ // found
                    if (pos->left == NULL){ // try to insert to the left child
first
                        pos->left = new VarOrder(iter->first); // create a new
node for the current attribute
                        pos->left->num = pos->num * 2 + 1; // number for the new
node
                        encode[iter->first - 'A' + 1] = pos->left->num; // record
the number in the vector
                        break;
                    }
                    else if (pos-> right == NULL){ // try the right child if the
left child is occupied
                        pos->right = new VarOrder(iter->first); // create a new
node for the current attribute
                        pos->right->num = pos->num * 2 + 2;// number for the new
node
```

```
                        encode[iter->first - 'A' + 1] = pos->right->num; // record
the number in the vector
                        break;

                    }
                }
            }

        }

    }

}
```

For type 2 query which contains a parameter EXPR that indicates the name of a column, the attribute for this column is inserted first. In this way we can enumerating the tuples in the factorised database with constant delay by executing partial aggregates on the other attributes on the fly. The implementation is similar to type 1. The only difference is that the first node is defined as the attribute EXPR. A depth-first-search approach is also used to find the parent node for each new attribute to be inserted.

(b) BuildFatorisedDB

After building the variable order, function BuildFatorisedDB is called to insert tuples from the cover to the database. There are three inputs. The first one is the tuple. The function reads in one tuple at a time therefore the space for extra memory is low. The variable order is represented by a pointer to the root. Lastly the positions of each attribute in R are passed by a vector of integers.

The structure for the database is designed as a tree. Each node has the value that stores the data which in this case are integers. Each node also has a left and a right pointer to a vector of nodes. The implementation is shown below.

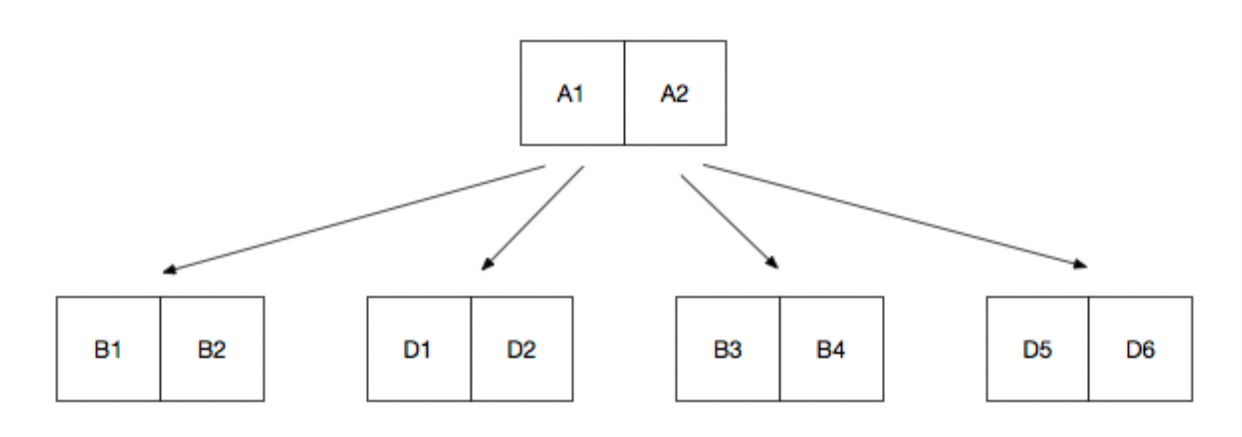
```
//-----------------------------------------------------------------
// Node
//
// Purpose: The unit for storing information in fatorised database
//
// Simplification: the fatorised database has a binary structure and
// each node has two children of vector of Nodes.
//
// Variables: val - value stored in the database
//          left - pointer to the left child which is a vector of Nodes
//          right - pointer to the right child which is a vector of Nodes
//-----------------------------------------------------------------


struct Node{
    int val;
    vector<Node*> left;
    vector<Node*> right;
    Node(int x): val(x),left(NULL),right(NULL){}
};
```

The strategy proposed here makes use of a vector which stores the matches for each attribute in the variable order. For each input tuple, the algorithm iterates over each position in the variable order. A function FindIndex is implemented to find which attribute is located in the current position. For example, if attribute A lies in position 1, FindIndex(1) would return the number indicating A. -1 is returned if the position is empty. C++ implementation is shown below.

```
//-------------------------------------------------------------------
// FindIndex
//
// Purpose : Return the attribute in the given position in the
          variable order
// Input   : encode - vetcor  which contains the position of each attribute in the
                              variable order.
//           x - position of the given attribute in the variable order
//
// Output : the assigned number for the attribute. -1 if not found.
//-------------------------------------------------------------------
int FindIndex(vector<int> encode,int x){
    for (int i = 1; i < encode.size() ; i++){
        if (x == encode[i]) // the attribute has the position x
            return i; // return the assigned number of the attribute
    }
    return -1; // not found
}
```

The positions in the database correspond to the positions in the variable order. If the position is not empty, we can try to insert the corresponding entry in the tuple to the database. For example, if FindIndex(1) returns A, the first entry of the tuple would be inserted into the position of 1 in the database. The structure of the database is demonstrated by the figure below.



As shown before, the positions in the variable order is encoded with numbers. It can be observed that the left children always have an even number and the right children have odd numbers. Therefore we can know which branch to go to when given a position. For example, position 3 must be the left child and position 4 must be the right child of position 1. Then the vector of nodes that the parent node points to will be searched for a match for the current attribute. A function called Find is implemented to search for a node in the vector.

```
//-------------------------------------------------------------------
// Find
//
// Purpose : find a Node in the given vector
// Input   : left - input tuple in the type of string.
//           delim - separator of the tuple
```

```
//
// Output : tuple in a vector form
//-----------------------------------------------------------------------
Node* Find(vector<Node*> left,Node* x){
    if (left.size() == 0)
        return NULL;
    for (auto l:left){
        if (l->val == x->val)
            return l;
    }
    return NULL;
}
```

If there is a match, it means a duplicate entry has been found. In this case there is no need to insert another node. Otherwise, the new node will be added to the vector. The position of the new node would be recorded so that the following entries would follow the same path in the database. The implementation of the function BuildFatorisedDB is shown below.

```
//-----------------------------------------------------------------------
// ;
//
// Purpose : Build a factorised database
// Input   : root - the root of the database
//           tuple - the input tuple from the cover
//           encode - the position of each attribute in the variable order
//
// Output : none
//-----------------------------------------------------------------------
void BuildFatorisedDB(Node* root,vector<int> tuple,vector<int> encode){

    // build a list of nodes for the input tuples
    vector<Node*> t(numAttributes);
    for (int i = 0; i < tuple.size(); i++){
        t[i+1] = new Node(tuple[i]);
    }

    vector<Node*> start(numNode); // record the match of each level
    start[0] = root;


    for (int i = 1; i <= numNode - 1; i++){
        int index = FindIndex(encode, i); // encode[] == i


        if (index != -1 ){
            if (i % 2 != 0){ //the current node should be on the left
                Node* match = NULL;
                match = Find(start[(i - 1)/2]->left,t[index]); // find the match
of the last level
                if (match == NULL){ // not found
                    Node* n = t[index]; // create a new node
                    start[(i - 1)/2]->left.push_back(n); // insert to the left
                    start[i] = n; // new match
                }
                else{
                    start[i] = match; // found
                }
```

```
            }
            else if (i % 2 == 0){//the current node should be on the right
                Node* match = NULL;
                match = Find(start[(i - 2)/2]->right,t[index]);// find the match
of the last level

                if (match == NULL){// not found
                    Node* n = t[index];
                    start[(i - 2)/2]->right.push_back(n); // insert to the right
                    start[i] = n; // new match

                }
                else{
                    start[i] = match; // found
                }
            }

        }


    }

    root = start[0];

}
```

(c) Sum

For type-1 and type-2 queries, functions for summation are implemented respectively.

For type-1 queries, they require the number of all the tuples in the relation R. By building a factorised database on the cover, one can easily compute the number of all tuples in R. Since the relation R can be recovered from the cover by joining relational columns defined in the decomposition, the count aggregate over factorised join can be used where each node in the database is considered as 1 and union operators are + and product operators are ∗[BKOZ13] .

This can be implemented using a recursive approach using a slightly different definition for the value of each node. Here the base case is defined such that an empty node has value 1. The value of each node is equal to the product of the value of its two children. The implementation is shown below.

```
//-------------------------------------------------------------------
// Count
//
// Purpose : count the number of tuples in the relation using a factorised
representation of the cover
// Input   : root - the root of the factorised database
//
// Output : the number of tuples in the relation
//-------------------------------------------------------------------
int Count(vector<Node*> root){
    if (root.size() == 0) // base case
        return 1;
    else{
        int sum = 0;
```

```
        // the value of the node = the value of the left child * the value of the
right child
        for (auto i:root){
            sum +=  Count(i->left) * Count(i->right); // recursive method
        }
        return sum;
    }
}
```

For type-1 query, one can simply call the following to calculate the number of all tuples.

```
 Count(root->left) * Count(root->right)
```

Similarly, type-2 queries can be answered by summing the product of the number of attributes and their value. A function SumColumn is defined for this purpose.

```
//-----------------------------------------------------------------
// SumColumn
//
// Purpose : Give the sum of a column
// Input   : root - the root of the factorised database
//
// Output : the number of tuples in the relation
//-----------------------------------------------------------------
int SumColumn(Node* root,vector<int> encode){
    int sum = 0;
    for (auto a:root->left){
        sum += a->val * Count(a->left) * Count(a->right);
    }
    return sum;
}
```

Part B The Design of CoverDB with GROUP-BY Clause

1 Introduction

Part B introduces the design of CoverDB with group-by clause. Specifically, this part looks at how to answer queries 3 and 4 in the benchmark.

The difference between queries with and without group-by clause is that, when building the variable order, the algorithm tries to insert group-by variables above others when answering group-by queries. However, this is not always possible. Hence there are the type 3 and 4 of queries.

2 Pseudo Code

(a) BuildVariableOrder

In this part, the function BuildVariableOrder is modified to process type-3 and type-4 queries. For each type of the query, it leads to different methods of inserting attributes. Below gives the extended version of BuildVariableOrder which includes the type 2 query that has been discussed and type 3 and 4 query to be explained.

```
  BuildVariableOrder(Q,C)
1 map ← BuildRelationMap(C)
2 If it is type 1 query
3   Δ ← InsertAttributes(map,C,Δ,encode)
4 If it is type 2 query
5   Δ ← InsertEXPR(map,C,Δ,encode)
6   Δ ← InsertAttributes(map,C,Δ,encode)
7 If it is type 3 query
8   (Δ,type) ← InsertATTR(map,C,Δ,encode)
9   If it is type 3 query
10      Δ ← InsertOtherAttributes(map,C,Δ,encode)
11  Else
12      Δ ← InsertAttributes(map,C,Δ,encode)
```

3 Analysis

For StandardDB, the input needs to be sorted, and the time complexity for this step can be considered as $O(mlgm)$ where m is the size of the relation. Then a full scan is performed on the sorted result. Therefore the time complexity is $O(mlgm)$ and the space complexity is $O(m)$. After sorting the relation, the output can reach a constant delay. The time complexity is linear to the size of the relation.

The performance of the extended CoverDB depends on the group-by attributes. If all the group-by attributes are either the root or the child of another attribute, the output of CoverDB can reach constant delay. Otherwise it cannot. In the given test cases, "group by A,B" can reach constant delay while "group by C,D" cannot.

For type-3 query, the group-by attributes can be inserted above other attributes. Therefore, if sorting is performed on the cover, it can result in the output of the algorithm being sorted as well. The time complexity is $O(nlgn)$ and the space complexity is $O(n)$ where n is the size of the cover. It is also possible that a map container is used which is able to sort the elements automatically. This is the same as sorting the output. The size of the map equals to the number of permutations over the group-by attributes which is at most n^a where a is the number of attributes. It is also possible that the size of the map is much smaller than n, resulting the sorting to be much faster. Apart from the sorting, the time complexity for aggregates is linear to the size of the over.

For type-4 query, the complexity will be much lower if a map is used to store the query results. For example, by iterating over the root nodes, a map can be created to map the group-by attributes to the root node. The root node is also associated with the aggregates which in this case is the sum of F. By adding all the values of the sum from the root nodes that each permutation of group-by attributes has, the results can be computed. Because the map can sort the results automatically, the output will be in order. Except the sorting, the time complexity is linear to the size of the cover.

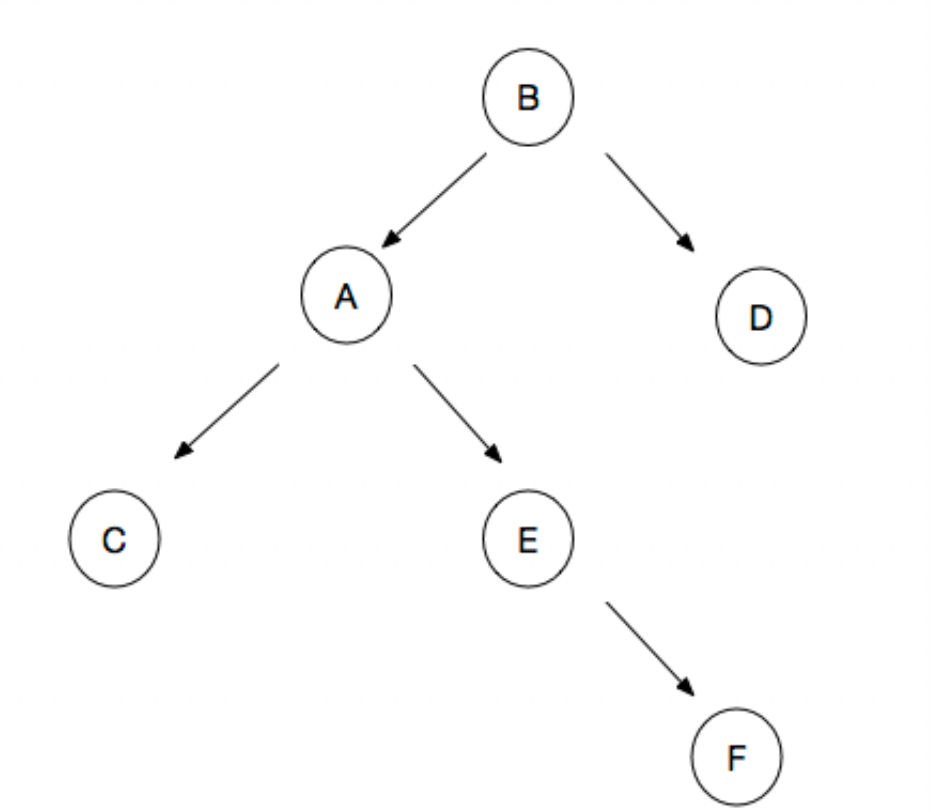
To reach fair comparison, one can also sort the input of CoverDB. For type-3 queries, sorting would be based on the EXPR attribute. For type-4 queries, this would depends on the root attribute.

4 Example

For Count query with group-by attribute A and B, the algorithm places A and B on the above as shown below.

For counting the number of tuples in the relation, the same method applies to this structure.

For summing the value of F given A and B, for each node of B, the algorithm first counts the number of D. Then for each node of A below B, the algorithm sums the value of F. The result is multiplied by the number of D under this node of B which indicates the number of duplicates in the relation.



5 Implementation

For type-3 and type-4 queries, a function InsertATTR is used to define the type of the query. As mentioned earlier, the differences between type 3 and type 4 is whether the attributes can be inserted above other variables. For example, in the test cases, A and B belongs to the same relation and can be put above others by linking them. However, C and D does not belong to the same relation and therefore they cannot be linked together. As a result, C and D cannot be above all other attributes. Function SameRelation is used to determine the type of query. C++ implementation is shown below where for each attribute

```
//-------------------------------------------------------------------
// SameRelation
//
// Purpose : Determine whether all the attributes in the query belong to the same
relation
// Input   : mp - mapping of relations
//           attributes - a vector that contains all the attributes in the query
//           decomposition - input of CoverDB.
// Output : a boolean value
```

```
//-------------------------------------------------------------------

bool SameRelation(map<char,set<char>> mp,vector<VarOrder*>
attributes,vector<vector<char>> decomposition){
    // for each relation, build a map that maps from the first attribute to the
rest of attributes
    for (auto a:attributes){
        set<char> relation = mp[a->val];
        for (auto r:relation){
            for (auto b:attributes){
                if ( a != b && b->val == r) // found
                    return true;
            }
        }
    }
       return false;
}
```

Next it is necessary to define which attribute can be the root in the variable order. Strategy is proposed here and implemented in function NotRoot. For each attribute, the algorithm looks at whether it is in a relation of size two. If this is true, this attribute would be less likely to be the root of the order. This is because there is another attribute which depends on it, and a position of its child are more likely to be needed. Being the root means that one of its child position is needed, thus the probability of free space is lower. For example, in this case B is selected as the root and A is considered not qualified as a root since E has to be connected to A. The implementation of NotRoot is shown as follows.

```
//-------------------------------------------------------------------
// NotRoot
//
// Purpose : Determine whether an attribute can be the root of the variable order
// Input   : attr - the value of the input attribute
//           decomposition - input for CoverDB .
// Output : a boolean value
//-------------------------------------------------------------------
bool NotRoot(char attr,vector<vector<char>> decomposition){
    for (int i = 0; i < decomposition.size();i++){
        // if the attribute is in the relation of size 2
        if ( decomposition[i].size() == 2)
            for (int j = 0; j < decomposition[i].size(); j++){
                if (decomposition[i][j] == attr)
                    return true;
            }
        }
    return false;
}
```

After deciding the root, other attributes will be inserted to the order if they belong to the same relation (type-3). In InsertATTR this is the final step. As we can see in the following implementation, only type-3 queries require the insertion of other attributes here.

```
//-------------------------------------------------------------------
// InsertATTR
//
// Purpose : Insert attributes in the variable order
// Input   : root - the root of the variable order.
//           encode - the position for each attribute in the variable order
```

```
//           query - the string for the input query
//           decomposition - the input for CoverDB
//           type - the type of the query
// Output :  None
//--------------------------------------------------------------------

void InsertATTR(VarOrder* root,vector<int>& encode,string
query,vector<vector<char>> decomposition,int & type){
    // build a list of attributes
    vector<VarOrder*> attributes = BuildAttributeList(query);
    map<char,set<char>> mp = BuildRelationMap(decomposition);
     // if there are more than one attribute
    if (attributes.size() > 1){
        // if the attribues does not belong to the same relation: type 4
        if (!SameRelation(mp,attributes, decomposition))
            type = 4;
        else
            type = 3;
    }
    // determine the root attribute in the variable order
    VarOrder* rootAttr = NULL;
    for (auto a:attributes)
        if (!NotRoot(a->val,decomposition)){ // if the attribute can be a root
            rootAttr = a;
            break;
        }
    // if the attributes belong to the same relation
    if (type == 3){
        VarOrder* matchR;
        // insert first node
        root->left = rootAttr;
        root->left->num = 2 * root->num + 1;
        encode[rootAttr->val - 'A' + 1] = root->left->num;
        matchR = root->left; // the last attribute inserted

        // change the order in the decompositions so that they start with the root
attribute in the variable order
        for (int i = 0; i < decomposition.size();i++)
            for (int j = 0; j < decomposition[i].size(); j++)
                if (decomposition[i][j] == rootAttr->val && j != 0){
                    int temp =decomposition[i][0];
                    decomposition[i][0] = rootAttr->val;
                    decomposition[i][j] = temp;
                }


        // insert all the other attributes
        for (auto a:attributes)
            if (a != rootAttr){
                VarOrder* match = FindVarOrder(root,a->val); // find this
attribute
                if (match == NULL){ // not found
                    if (matchR->left == NULL){ // insert on the left of the last
attribute inserted
                        matchR->left = a;
                        matchR->left->num = 2 * matchR->num + 1; // record the
position
                        encode[a->val - 'A' + 1] =   matchR->left->num;
                        matchR = matchR->left;
```

```
                    }
                    else{// insert on the right of the last attribute inserted
                        matchR->right = a;
                        matchR->right->num = 2 * matchR->num + 2; // record the
position
                        encode[a->val - 'A' + 1] = matchR->right->num;
                        matchR = matchR->right;
                    }
                }
            }
    }
}
```

For type-4 query, since it is not possible to reach constant delay, we reuse the order from type-1. Therefore InsertAttribute is called in this case.

(B) Sum

It is noted that the summation here is not a general approach and only limited to the variable order given by the function above. However, it can be easily extended to a general form.

For type-3 query, according to the design of BuildVariableOrder, EXPR is the placed at the top of the order. Therefore, the sum of EXPR equals to the value of each entry times the number of each entry which can be easily calculated through aggregates in the factorised join. For the structure where A and B are above F, sum of F for each possible permutation of A and B can be easily calculated by multiplying the number each node of F represents and the F value. Since factorised database removes duplicates in each subtree, all we need to know is that how many tuples each node represents. This can be achieved by counting the number of tuples each subtree represents. In other words, this is to count the number of tuples that satisfy the subtree constraint in the relation R. In GroupByType3, the above method is implemented and the result is stored using map container, which maps a string indicating the attribute A,B and the sum of the F values. Map container makes sure that its elements are ordered thus the output of the answers to the group-by queries are ordered. Details can be found in the following implementation.

```
//-----------------------------------------------------------------
// GroupByType3
//
// Purpose : Compute the result for type 3 group-by clause query
// Simplification : the number of attributes are two
//                  the structure of the variable order is known
// Input   : root - the root of the factorised database
//           x - specification of the subtree
//
// Output : a map between attributes and sum
//-----------------------------------------------------------------


map<string,int> GroupByType3(Node* root){
    map<string,int> mp;

    vector<int> nums;

    // Benchmark group by AB
    for (auto b: root->left){
```

```
        nums.push_back(Sum(b,'R')); // sum the right subtree of the first
attribute

        for (auto a: b->left){
            nums.push_back(Sum(a,'L')); // Get the number of the second attribute
            int sum = 0;
            for (auto e: a->right){
                for (auto f:e->left){ // Sum(F)
                    sum += nums[0] * nums[1] * f->val; // value * number
                }
            }
            string ba = to_string(b->val) + " " + to_string(a->val); // a string
represents the selection of attributes

            mp[ba] = sum; // mapping between attributes and sum

        }
    }
    return mp;


}
```

For type-4 query, a similar approach is used. The idea is to link between the F value we want to sum up and the C,D attributes of them, which can be linked through the root of the variable order, in this case is A. It is noted that in type-4, the variable order is the same as type-1 since the group-by variable cannot be placed above other variables. In function GroupByType4, for each value of A, the sum of F is first calculated. Then for each value of A, the corresponding C and D value are found. In this way the value of C,D and F are linked through A. A map is also used to store the result. C++ implementation can be found below.

```
//-----------------------------------------------------------------------
// GroupByType4
//
// Purpose : Compute the result for type 4 group-by clause query
// Simplification : the number of attributes are two
//                  the structure of the variable order is known
// Input   : root - the root of the factorised database
//
// Output : a map between attributes and sum
//-----------------------------------------------------------------------
map<string,int> GroupByType4(Node* root){
    map<string,int> mp;

    for (auto a:root->left){
        //get value of F
        int fv = 0;
        for (auto e:a->right)
            for (auto f:e->left)
                fv += f->val;

        //get value of C & D
        for (auto b:a->left){
            for (auto c:b->left){
                for (auto d:b->right){
                    string cd = to_string(c->val) + " " + to_string(d->val);
                    mp[cd] += fv;
                }
```

```
            }
        }
    }

    return mp;
}
```

Part C Experimental Result

(a) Hardware specification

The experiment is run on MacOS, Intel(R) Core(TM) i7-7660U CPU @ 2.50GHz. The computer has 1 processor and 2 cores. L2 Cache per core is 256kB and L3 cache is 4MB. It has the total memory of 8GB.

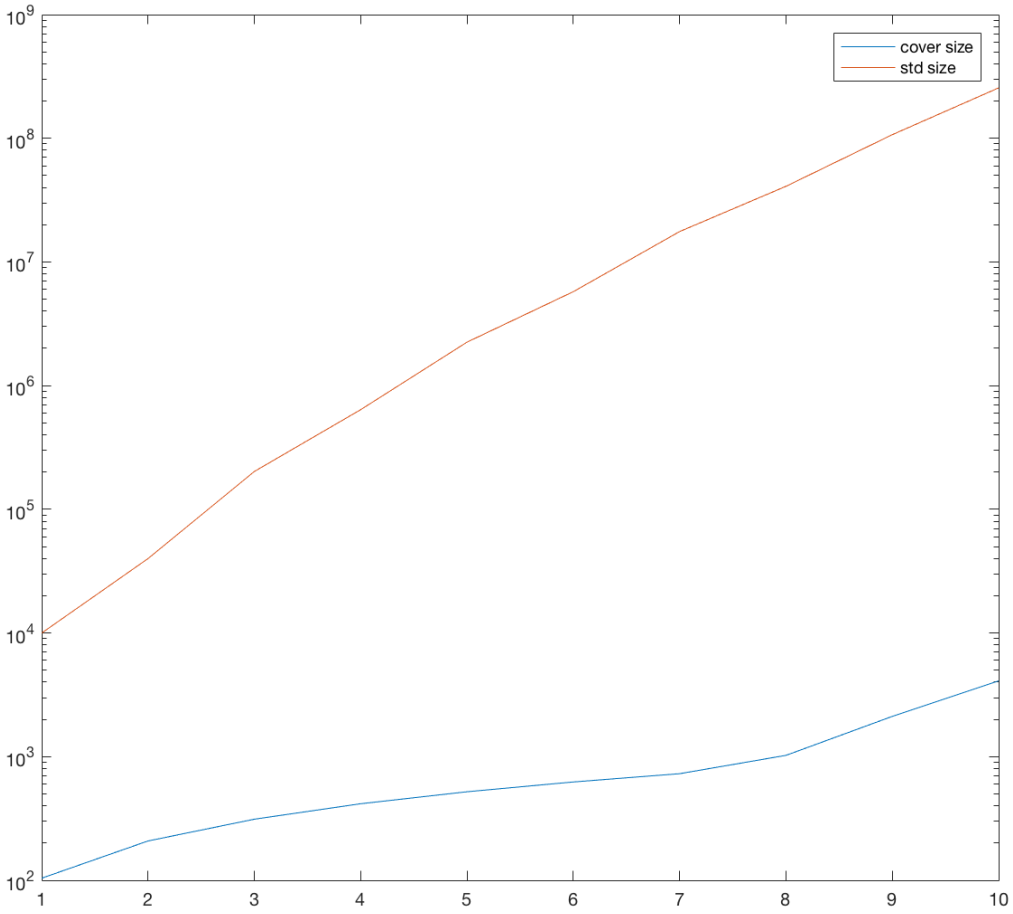
(b) Script

The experiments are run through terminal and a script can be found in the folder accompanying the C++ code. The implementation is integrated in main.cpp. Valgrind can be used to analyse the cache miss. The script is shown below.

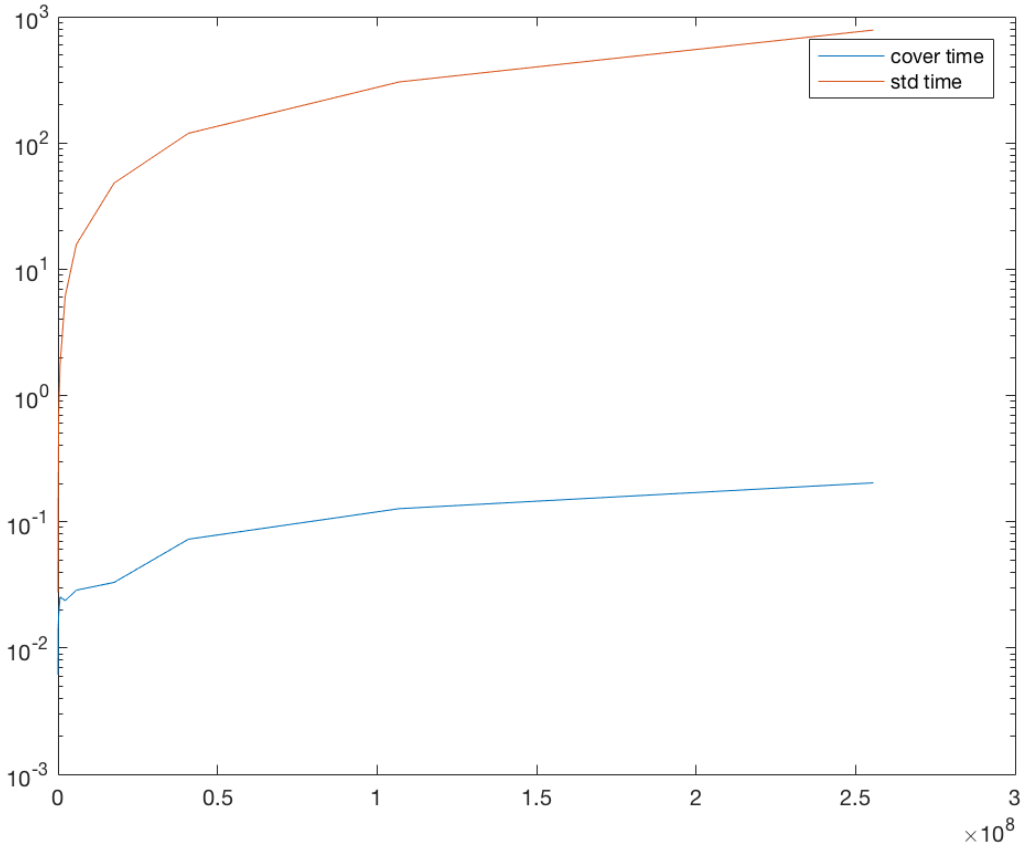
```
g++ -std=c++11 -g main.cpp
valgrind --tool=cachegrind ./a.out
```

(c) Results

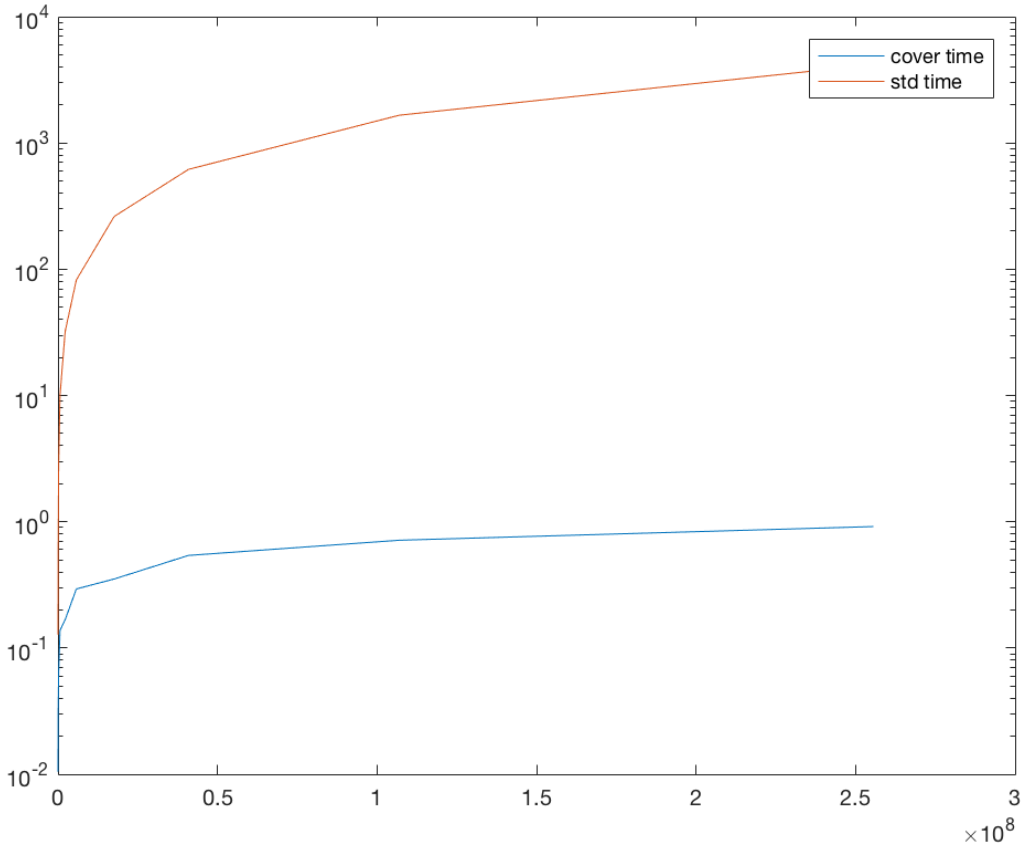
CoverDB requires a search in the factorised database which is built according to the variable order. The time complexity is linear to the size of the cover thus both the time and space complexity are $O(n)$ where n is the size of the cover. As for StandardDB, aggregates require scanning all tuples in the relation. The time complexity is linear to the size of the relation thus both the time and space complexity are $O(m)$ where m is the size of the relation. The relation between m and n depends on the number of duplicates in the relation. As shown in the below graph, when the size of the relation increases, the number of tuples in the cover also increases. It is noted that the increase of size for StandardDB is much faster compared to CoverDB. In other words, CoverDB requires much fewer tuples to represent the relation compared to the actual size of the relation when the relation is large. For example, when m is about 10^4, n is about 10^2 and therefore m is about 100 * n. When m is about 10^8, n is about 10^4 and therefore m is about 10^4 * n. In this case, CoverDB requires much less memory.



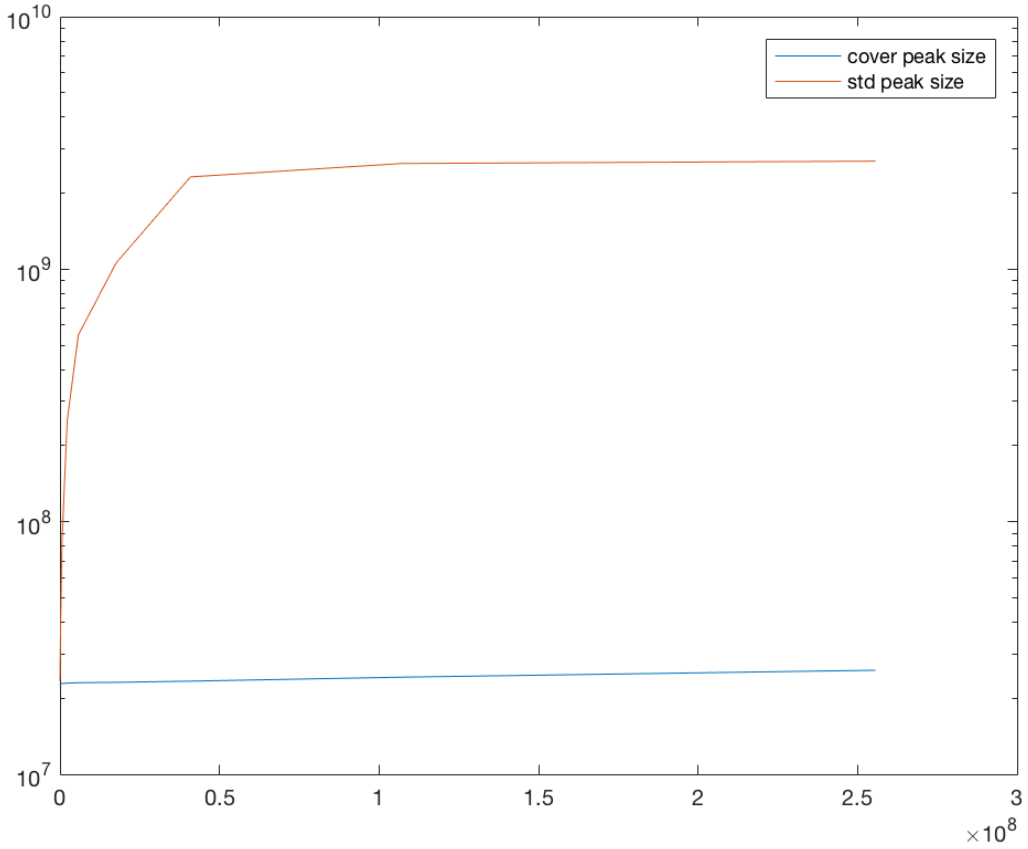
The actual time used in both programs are query 1 and query 2 are shown in the below graph. The x-axis denotes the size of the relation and the y-axis shows the time each program uses in seconds. The running time for StandardDB is much more than CoverDB. When the size of the relation is about 10^8, CoverDB is about 100 times faster. When the size of the relation is about $2 * 10^8$, CoverDB is approximately 10000 times faster.



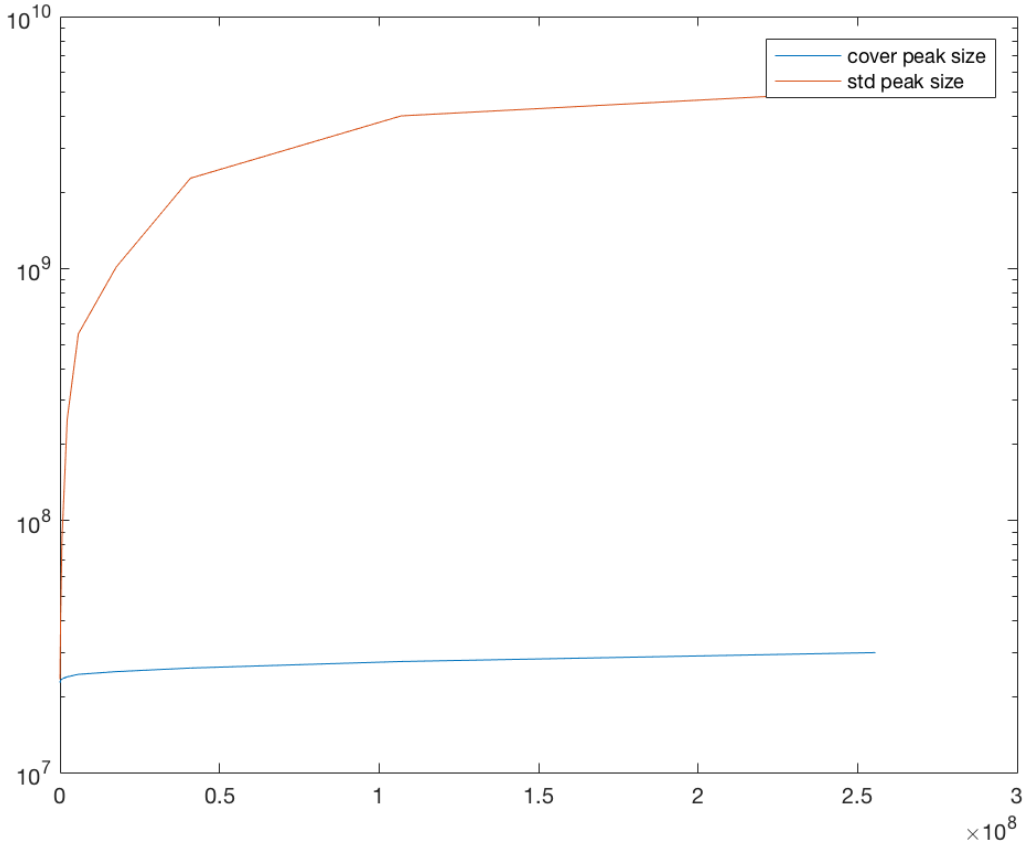
The actual time used in both programs are query 3 and query 4 are shown in the below graph. The x-axis denotes the size of the relation and the y-axis shows the time each program uses in seconds. The running time for StandardDB is much more than CoverDB. When the size of the relation is about 10^8, CoverDB is about 1000 times faster. When the size of the relation is about $2 * 10^8$, CoverDB is approximately 10000 times faster. For fair comparison, sorting on the input is also performed in CoverDB.



The actual memory usages for query 1 and query 2 are shown in the below graph.The x-axis denotes the size of the relation and the y-axis shows the memory each program uses. It can be clearly seen that StandardDB has much higher memory usage compared to CoverDB. It is also observed that with the increase of number of tuples in the relation, the memory usage of CoverDB does not increase significantly. It can be concluded that CoverDB is especially memory efficient when processing large datasets. The reason why the usage of memory does not change much after the size of the relation reaching $0.5 * 10^8$ is that the system reaches the limit of memory size.

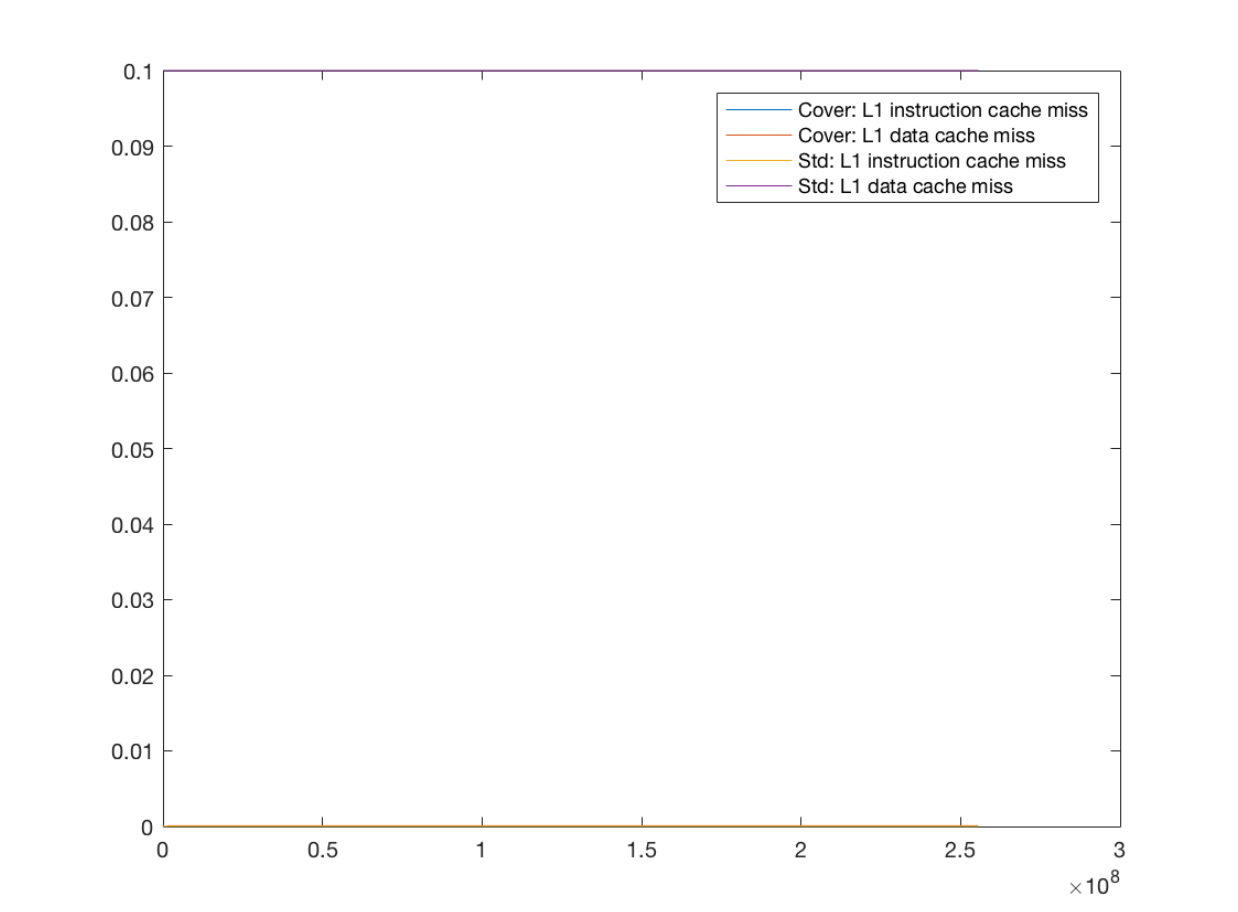


Memory usages for query 3 and query 4 are shown below.



Code quality

In this program Cachegrind is used to measure the number of cache misses. It simulates a machine with two levels of independent instruction and data caches. In this case it will report the usage of L1 caches. As shown below, the L1 data miss rate of CoverDB is lower than StandardDB. Instruction cache miss rates are similar.



References

[1] G.Gottlob, N.Leone and F. Scarcello. Hypertree Decomposition and Tractable Queries. Journal of Computer and System Sciences, 64(3):579-627, 2002.

[2] N. Bakibayev, T. Kocisky, D. Olteanu and J. Zavodny. Aggregation and Ordering in Factorised Databases. Proceedings of the VLDB Endowmen. Volume 6 Issue 14, September 2013 , Pages 1990-2001.

[3] D. Olteanu and J. Zavodny. Size Bounds for Factorised Representations of Query Results. ACM Transactions on Database Systems (TODS). Volume 40 Issue 1, March 2015, Article No. 2.

Appendix

```
//
//  main.cpp
//  CoverDB
//
//  Created by Charlotte Zhao on 06/04/2018.
//  Copyright © 2018 Zheyi Zhao. All rights reserved.
//

#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <map>
#include <unordered_map>
#include <set>
#include <algorithm>
#include <time.h>

#include<mach/mach.h>
struct task_basic_info t_info;
mach_msg_type_number_t t_info_count = TASK_BASIC_INFO_COUNT;

using namespace std;

const int numNode = 40; // define the size of the variable order
const int numAttributes = 7; // number of attributes in R
#include <stdio.h>
#include <sys/types.h>
#include <sys/sysctl.h>
#define BUFFERLEN 128
#include <unistd.h>
#include <ios>
#include <iostream>
#include <fstream>
#include <string>

////////////////////////////////////////////////////////////////////////////
//http://nadeausoftware.com/articles/2012/07/c_c_tip_how_get_process_resident_set_
size_physical_memory_use

// process_mem_usage(double &, double &) - takes two doubles by reference,
// attempts to read the system-dependent data for a process' virtual memory
// size and resident set size, and return the results in KB.
//
// On failure, returns 0.0, 0.0
//
void process_mem_usage(double& vm_usage, double& resident_set)
{
    using std::ios_base;
    using std::ifstream;
    using std::string;

    vm_usage     = 0.0;
    resident_set = 0.0;

    // 'file' stat seems to give the most reliable results
    //
    ifstream stat_stream("/proc/self/stat",ios_base::in);
```

```
    // dummy vars for leading entries in stat that we don't care about
    //
    string pid, comm, state, ppid, pgrp, session, tty_nr;
    string tpgid, flags, minflt, cminflt, majflt, cmajflt;
    string utime, stime, cutime, cstime, priority, nice;
    string O, itrealvalue, starttime;

    // the two fields we want
    //
    unsigned long vsize;
    long rss;

    stat_stream >> pid >> comm >> state >> ppid >> pgrp >> session >> tty_nr
    >> tpgid >> flags >> minflt >> cminflt >> majflt >> cmajflt
    >> utime >> stime >> cutime >> cstime >> priority >> nice
    >> O >> itrealvalue >> starttime >> vsize >> rss; // don't care about the rest

    stat_stream.close();

    long page_size_kb = sysconf(_SC_PAGE_SIZE) / 1024; // in case x86-64 is
configured to use 2MB pages
    vm_usage     = vsize / 1024.0;
    resident_set = rss * page_size_kb;
}
/*
 * Author:  David Robert Nadeau
 * Site:    http://NadeauSoftware.com/
 * License: Creative Commons Attribution 3.0 Unported License
 *          http://creativecommons.org/licenses/by/3.0/deed.en_US
 */

#if defined(_WIN32)
#include <windows.h>
#include <psapi.h>

#elif defined(__unix__) || defined(__unix) || defined(unix) || (defined(__APPLE__)
&& defined(__MACH__))
#include <unistd.h>
#include <sys/resource.h>

#if defined(__APPLE__) && defined(__MACH__)
#include <mach/mach.h>

#elif (defined(_AIX) || defined(__TOS__AIX__)) || (defined(__sun__) ||
defined(__sun) || defined(sun) && (defined(__SVR4) || defined(__svr4__)))
#include <fcntl.h>
#include <procfs.h>

#elif defined(__linux__) || defined(__linux) || defined(linux) ||
defined(__gnu_linux__)
#include <stdio.h>

#endif

#else
#error "Cannot define getPeakRSS( ) or getCurrentRSS( ) for an unknown OS."
#endif
```

```
/**
 * Returns the peak (maximum so far) resident set size (physical
 * memory use) measured in bytes, or zero if the value cannot be
 * determined on this OS.
 */
size_t getPeakRSS( )
{
#if defined(_WIN32)
    /* Windows -------------------------------------------------- */
    PROCESS_MEMORY_COUNTERS info;
    GetProcessMemoryInfo( GetCurrentProcess( ), &info, sizeof(info) );
    return (size_t)info.PeakWorkingSetSize;

#elif (defined(_AIX) || defined(__TOS__AIX__)) || (defined(__sun__) ||
defined(__sun) || defined(sun) && (defined(__SVR4) || defined(__svr4__)))
    /* AIX and Solaris ------------------------------------------ */
    struct psinfo psinfo;
    int fd = -1;
    if ( (fd = open( "/proc/self/psinfo", O_RDONLY )) == -1 )
        return (size_t)0L;      /* Can't open? */
    if ( read( fd, &psinfo, sizeof(psinfo) ) != sizeof(psinfo) )
    {
        close( fd );
        return (size_t)0L;      /* Can't read? */
    }
    close( fd );
    return (size_t)(psinfo.pr_rssize * 1024L);

#elif defined(__unix__) || defined(__unix) || defined(unix) || (defined(__APPLE__)
&& defined(__MACH__))
    /* BSD, Linux, and OSX -------------------------------------- */
    struct rusage rusage;
    getrusage( RUSAGE_SELF, &rusage );
#if defined(__APPLE__) && defined(__MACH__)
    return (size_t)rusage.ru_maxrss;
#else
    return (size_t)(rusage.ru_maxrss * 1024L);
#endif

#else
    /* Unknown OS ----------------------------------------------- */
    return (size_t)0L;          /* Unsupported. */
#endif
}




/**
 * Returns the current resident set size (physical memory use) measured
 * in bytes, or zero if the value cannot be determined on this OS.
 */
size_t getCurrentRSS( )
{
```

```
#if defined(_WIN32)
    /* Windows -------------------------------------------------- */
    PROCESS_MEMORY_COUNTERS info;
    GetProcessMemoryInfo( GetCurrentProcess( ), &info, sizeof(info) );
    return (size_t)info.WorkingSetSize;

#elif defined(__APPLE__) && defined(__MACH__)
    /* OSX ------------------------------------------------------ */
    struct mach_task_basic_info info;
    mach_msg_type_number_t infoCount = MACH_TASK_BASIC_INFO_COUNT;
    if ( task_info( mach_task_self( ), MACH_TASK_BASIC_INFO,
                    (task_info_t)&info, &infoCount ) != KERN_SUCCESS )
        return (size_t)0L;      /* Can't access? */
    return (size_t)info.resident_size;

#elif defined(__linux__) || defined(__linux) || defined(linux) ||
defined(__gnu_linux__)
    /* Linux ---------------------------------------------------- */
    long rss = 0L;
    FILE* fp = NULL;
    if ( (fp = fopen( "/proc/self/statm", "r" )) == NULL )
        return (size_t)0L;      /* Can't open? */
    if ( fscanf( fp, "%*s%ld", &rss ) != 1 )
    {
        fclose( fp );
        return (size_t)0L;      /* Can't read? */
    }
    fclose( fp );
    return (size_t)rss * (size_t)sysconf( _SC_PAGESIZE);

#else
    /* AIX, BSD, Solaris, and Unknown OS ------------------------ */
    return (size_t)0L;          /* Unsupported. */
#endif
}

void SystemInfo(){
    char buffer[BUFFERLEN];
    size_t bufferlen = BUFFERLEN;
    sysctlbyname("machdep.cpu.brand_string",&buffer,&bufferlen,NULL,0);
    printf("%s\n", buffer);


    size_t currentSize = getCurrentRSS( );
    size_t peakSize    = getPeakRSS( );
    cout <<currentSize <<" " << peakSize << endl;
}









//------------------------------------------------------------------
// Node
//
```

```
// Purpose: The unit for storing information in fatorised database
//
// Simplification: the fatorised database has a binary structure and
// each node has two children of vector of Nodes.
//
// Variables: val - value stored in the database
//          left - pointer to the left child which is a vector of Nodes
//          right - pointer to the right child which is a vector of Nodes
//-----------------------------------------------------------------


struct Node{
    int val;
    vector<Node*> left;
    vector<Node*> right;
    Node(int x): val(x),left(NULL),right(NULL){}
};

//-----------------------------------------------------------------
// VarOrder
//
// Purpose: The unit for variable order
//
// Simplification: the varible order is represented via a binary tree
// instead of a forest
//
// Variables: val - name of the attribute
//            num - position of the attribute in the tree
//            left - a pointer to the left child
//            right - a pointer to the right child
//-----------------------------------------------------------------

struct VarOrder{
    char val;
    int num;
    VarOrder* left;
    VarOrder* right;
    VarOrder(char x): val(x),left(NULL),right(NULL){}
};

//-----------------------------------------------------------------
// FindVarOrder
//
// Purpose : Find the pointer to the node of the given attribute
//           in the variable order .
// Input   : root - the root of the variable order tree
//           x - the given attribute
// Output : the pointer to the node of the given attribute
//-----------------------------------------------------------------

VarOrder* FindVarOrder(VarOrder* root,char x){
    VarOrder* res = NULL;

    if (root == NULL)
        return root;
    if (root->val == x){ // if the root contains the given attribute, return the
root
        return root;
    }
```

```
    res = FindVarOrder(root->left,x); // recursively searching in the left sub-
tree
    if (res != NULL) // found
        return res;
    else {
        res = FindVarOrder(root->right,x); // recursively searching in the right
sub-tree
        return res;

    }
}
//---------------------------------------------------------------------
// BuildRelationMap
//
// Purpose : Build relation map according to the input decomposition.
// Input   : decomposition - input of CoverDB.
// Output : the map of relations
//---------------------------------------------------------------------
map<char,set<char>> BuildRelationMap(vector<vector<char>> decomposition){
    map<char,set<char>> mp; // a map from one attribute to its relating attributes
    // for every element in the given decomposition
    for (int i = 0; i < decomposition.size(); i++)
        for (int j = 0; j< decomposition[i].size(); j++)
            // find other attributes in the same relation
            for (int k = 0; k < decomposition[i].size(); k++)
                if (j != k ) // not itself
                    mp[decomposition[i][j]].insert(decomposition[i][k]); // insert
into the mapping
    return mp;
}


//---------------------------------------------------------------------
// BuildAttributeList
//
// Purpose : Extract the ATTRs from the query
// Simplification: All the attributes are represented by alphabets
// Input   : query - the given query that contains attributes
// Output : a vector of pointers to the attributes in the variable order
//---------------------------------------------------------------------

vector<VarOrder*> BuildAttributeList(string query){
    vector<VarOrder*> res;
    int begin = (int)query.find(" ") + 1; // position after "SELECT "
    int end = (int)query.find("SUM") - 1; // position before "SUM"

    string attr = query.substr(begin,end-begin); // sub string between SELECT and
SUM

    for (int i = 0; i < attr.length(); i++)
        if (attr[i] >= 'A' && attr[i] <= 'Z') // alphabets
        {
            VarOrder* v = new VarOrder (attr[i]); // create a new node in the
variable order
            res.push_back(v); // store the node in the vector
        }
    return res;
}
//---------------------------------------------------------------------
```

```
// FindEXPR
//
// Purpose : Extract the EXPR from the query
// Simplification: An EXPR is represented by one alphabet
// Input   : query - the given query that contains attributes
// Output : a pointer to EXPR in the variable order
//-----------------------------------------------------------------
VarOrder* FindEXPR(string query){
    int begin = (int)query.find("SUM(") + 1; // position after "SUM("
    string EXPR = query.substr(begin,1); // extract the sub string after "SUM("
    VarOrder* v = new VarOrder (EXPR[0]); // Get the first character of the sub
string and create a node in the variable order
    return v; // return this node
}

//-----------------------------------------------------------------
// SameRelation
//
// Purpose : Determine whether all the attributes in the query belong to the same
relation
// Input   : mp - mapping of relations
//           attributes - a vector that contains all the attributes in the query
//            decomposition - input of CoverDB.
// Output : a boolean value
//-----------------------------------------------------------------

bool SameRelation(map<char,set<char>> mp,vector<VarOrder*>
attributes,vector<vector<char>> decomposition){
    // for each relation, build a map that maps from the first attribute to the
rest of attributes
    for (auto a:attributes){
        set<char> relation = mp[a->val];
        for (auto r:relation){
            for (auto b:attributes){
                if ( a != b && b->val == r) // found
                    return true;
            }
        }

    }

       return false;
}

//-----------------------------------------------------------------
// InAttr
//
// Purpose : Determine whether an attribute is in the attribute list
// Input   : attributes - a vector that contains all the attributes in the query
//            x - an input attribute .
// Output : a boolean value
//-----------------------------------------------------------------

bool InAttr(vector<VarOrder*> attributes,char x){
    for (auto a:attributes)
        if (x == a->val) // in the list
            return true;
    return false; // not in the list
}
```

```
//-------------------------------------------------------------------
// NotRoot
//
// Purpose : Determine whether an attribute can be the root of the variable order
// Input   : attr - the value of the input attribute
//           decomposition - input for CoverDB .
// Output : a boolean value
//-------------------------------------------------------------------
bool NotRoot(char attr,vector<vector<char>> decomposition){
    for (int i = 0; i < decomposition.size();i++){
        // if the attribute is in the relation of size 2
        if ( decomposition[i].size() == 2)
            for (int j = 0; j < decomposition[i].size(); j++){
                if (decomposition[i][j] == attr)
                    return true;
            }
        }
    return false;
}
//-------------------------------------------------------------------
// FindVarNode
//
// Purpose : Find a node in the variable order.
// Input   : root - the root of the variable order.
//           x - the value of the node to be found.
//           res - the result of finding
// Output :  none
//-------------------------------------------------------------------
void FindVarNode(VarOrder*& root, char x,VarOrder*& res){
    if (root == NULL)
        return ;
    if (root->val == x) // if the root is the target node
        res = root;
    FindVarNode(root->left,x,res); // recursively search the left sub tree
    FindVarNode(root->right,x,res);// recursively search the right sub tree
}
//-------------------------------------------------------------------
// FindVarOrder_BFS
//
// Purpose : Find a node in the variable order in a Breadth-First-Search way
// Input   : root - the root of the variable order.
//           relation - the set of relation whose positions are to be found
//           res - the result of finding
// Output :  none
//-------------------------------------------------------------------
void FindVarOrder_BFS(VarOrder*& root,set<char> relation,vector<VarOrder*>& res){
    if (root == NULL)
        return ;
    if (relation.find(root->val) != relation.end()){ // found the attribute in the
variable order
        res.push_back(root);
    }
    FindVarOrder_BFS(root->left,relation,res); // left sub-tree
    FindVarOrder_BFS(root->right,relation,res); // right sub-tree
}
//-------------------------------------------------------------------
// FindVarOrder_DFS
//
// Purpose : Find a node in the variable order in a Depth-First-Search way
```

```
// Input   : root - the root of the variable order.
//           relation - the set of relation whose positions are to be found
//           res - the result of finding
// Output :  none
//-----------------------------------------------------------------

void FindVarOrder_DFS(VarOrder*& root,set<char> relation,vector<VarOrder*>& res){
    if (root == NULL)
        return ;
    FindVarOrder_DFS(root->left,relation,res); // left sub-tree

    if (relation.find(root->val) != relation.end()){ // if the root is the target
node
        res.push_back(root);
    }
    FindVarOrder_DFS(root->right,relation,res); // right sub-tree
}
//-----------------------------------------------------------------
// QueryType
//
// Purpose : Determine the type of the query
// Input   : query - string for the input query
// Output :  the type of the query
//-----------------------------------------------------------------
int QueryType(string query){
    int type = 0;
    if (query.find("SUM(1)") !=std::string::npos) // SUM(1)
        type = 1;
    else if  (query.find("GROUP BY") !=std::string::npos){ //GROUP BY CLAUSE
        type = 3;
    }
    else{
        type = 2; // SUM(EXPR)
    }
    return type;
}

//-----------------------------------------------------------------
// InsertEXPR
//
// Purpose : Insert attribute EXPR in the variable order
// Input   : root - the root of the variable order.
//           encode - the position for each attribute in the variable order
//           query - the string for the input query
//           decomposition - the input for CoverDB
// Output :  None
//-----------------------------------------------------------------
void InsertEXPR(VarOrder* &root,vector<int>& encode,string
query,vector<vector<char>> decomposition){
    int pos = (int)query.find("SUM(") + 1; // the position in the query string
after "SUM("
    char attr = query[pos]; // get EXPR

    VarOrder* rootAttr = new VarOrder(attr);
    // insert first node
    root->left = rootAttr;
    root->left->num = 2 * root->num + 1; // position in the variable order
    encode[rootAttr->val - 'A' + 1] = root->left->num;
}
```

```
//-------------------------------------------------------------------
// InsertATTR
//
// Purpose : Insert attributes in the variable order
// Input   : root - the root of the variable order.
//           encode - the position for each attribute in the variable order
//           query - the string for the input query
//           decomposition - the input for CoverDB
//           type - the type of the query
// Output :  None
//-------------------------------------------------------------------

void InsertATTR(VarOrder* root,vector<int>& encode,string
query,vector<vector<char>> decomposition,int & type){
    // build a list of attributes
    vector<VarOrder*> attributes = BuildAttributeList(query);
    map<char,set<char>> mp = BuildRelationMap(decomposition);
     // if there are more than one attribute
    if (attributes.size() > 1){
        // if the attribues does not belong to the same relation: type 4
        if (!SameRelation(mp,attributes, decomposition))
            type = 4;
        else
            type = 3;
    }
    // determine the root attribute in the variable order
    VarOrder* rootAttr = NULL;
    for (auto a:attributes)
        if (!NotRoot(a->val,decomposition)){ // if the attribute can be a root
            rootAttr = a;
            break;
        }
    // if the attributes belong to the same relation
    if (type == 3){
        VarOrder* matchR;
        // insert first node
        root->left = rootAttr;
        root->left->num = 2 * root->num + 1;
        encode[rootAttr->val - 'A' + 1] = root->left->num;
        matchR = root->left; // the last attribute inserted

        // change the order in the decompositions so that they start with the root
attribute in the variable order
        for (int i = 0; i < decomposition.size();i++)
            for (int j = 0; j < decomposition[i].size(); j++)
                if (decomposition[i][j] == rootAttr->val && j != 0){
                    int temp =decomposition[i][0];
                    decomposition[i][0] = rootAttr->val;
                    decomposition[i][j] = temp;
                }


        // insert all the other attributes
        for (auto a:attributes)
            if (a != rootAttr){
                VarOrder* match = FindVarOrder(root,a->val); // find this
attribute
                if (match == NULL){ // not found
```

```
                    if (matchR->left == NULL){ // insert on the left of the last
attribute inserted
                        matchR->left = a;
                        matchR->left->num = 2 * matchR->num + 1; // record the
position
                        encode[a->val - 'A' + 1] =   matchR->left->num;
                        matchR = matchR->left;

                    }
                    else{// insert on the right of the last attribute inserted
                        matchR->right = a;
                        matchR->right->num = 2 * matchR->num + 2; // record the
position
                        encode[a->val - 'A' + 1] = matchR->right->num;
                        matchR = matchR->right;
                    }
                }
            }
    }
}


//-----------------------------------------------------------------------
// InsertAttributes
//
// Purpose : Build variable order by inserting all attributes in R by finding
relational attributes in a depth-search-first manner
// Input   : mp - the map of relations.
//           decomposition - input of CoverDB.
//           root - the root of the variable order
//           encode - the vector that contains the positions of all attributes in
the variable order
// Output : none
//-----------------------------------------------------------------------
void InsertAttributes(map<char,set<char>> mp, vector<vector<char>> decomposition,
VarOrder* root,vector<int>& encode){

    map<char,set<char>> :: iterator iter;

    for (iter = mp.begin(); iter != mp.end(); iter ++){
        // first attribute to be inserted into the variable order
        // default positiobn for the first attribute is the left child of the root
        if (root->left == NULL){
            root->left = new VarOrder(iter->first);
            root->left->num =root->num * 2 + 1; // the number for the new node
            encode[iter->first - 'A' + 1] = root->left->num; // put the number of
the node into a vector
            continue; // next attribute

        }

        VarOrder* position = NULL; // the position of the current attribute in the
variable order
        FindVarNode(root->left,iter->first,position); // find attribute in the
order
        if (position == NULL){ // attribute not found: find the attributes it
relates to in the order
            set<char> relation = iter->second;
            set<char> :: iterator siter;
```

```
            vector<VarOrder*> res;
            FindVarOrder_DFS(root,relation,res); // find the position of other
attributes in the same relation in a depth-search-first manner
            for (auto pos:res){ // for every attribute in the relation

                if (pos != NULL){ // found
                    if (pos->left == NULL){ // try to insert to the left child
first
                        pos->left = new VarOrder(iter->first); // create a new
node for the current attribute
                        pos->left->num = pos->num * 2 + 1; // number for the new
node
                        encode[iter->first - 'A' + 1] = pos->left->num; // record
the number in the vector
                        break;
                    }
                    else if (pos-> right == NULL){ // try the right child if the
left child is occupied
                        pos->right = new VarOrder(iter->first); // create a new
node for the current attribute
                        pos->right->num = pos->num * 2 + 2;// number for the new
node

                        encode[iter->first - 'A' + 1] = pos->right->num; // record
the number in the vector
                        break;

                    }
                }
            }

        }


    }

}
//-----------------------------------------------------------------
// InsertOtherAttributes
//
// Purpose : Build variable order by inserting all attributes in R except the EXPR
by finding relational attributes in a breadth-search-first manner
// Input   : mp - the map of relations.
//           decomposition - input of CoverDB.
//           root - the root of the variable order
//           encode - the vector that contains the positions of all attributes in
the variable order
//           type - the type of the query
// Output : none
//-----------------------------------------------------------------
void InsertOtherAttributes(map<char,set<char>> mp, vector<vector<char>>
decomposition, VarOrder* root,vector<int>& encode,int type){

    map<char,set<char>> :: iterator iter;

    for (iter = mp.begin(); iter != mp.end(); iter ++){

            VarOrder* position = NULL;
            // if the order is empty
```

```
            if (root->left == NULL){
                root->left = new VarOrder(iter->first);
                root->left->num = root->num * 2 + 1;
                encode[iter->first - 'A' + 1] = root->left->num;
                continue;

            }
            // find the current attribute
            FindVarNode(root->left,iter->first,position);
            if (position == NULL){
                set<char> relation = iter->second;
                set<char> :: iterator siter;
                vector<VarOrder*> res;
                //  finding relational attributes in a depth-search-first manner
                FindVarOrder_BFS(root,relation,res);

                for (auto pos:res){

                    if (pos != NULL){
                        if (pos->left == NULL){
                            pos->left = new VarOrder(iter->first);
                            pos->left->num = pos->num * 2 + 1;
                            encode[iter->first - 'A' + 1] = pos->left->num;
                            break;
                        }
                        else if (pos-> right == NULL){
                            pos->right = new VarOrder(iter->first);
                            pos->right->num = pos->num * 2 + 2;
                            encode[iter->first - 'A' + 1] = pos->right->num;
                            break;

                        }
                    }
                }

            }


    }


}
//-------------------------------------------------------------------
// BuildVariableOrder
//
// Purpose : Build Variable Order Delta according to the given query and
decomposition.
// Input   :  encode - vetcor  which contains the position of each attribute in
the                              variable order;
//           query - the query sentence;
//           decomposition - input of CoverDB;
//
// Output : the root of the variable order tree
//-------------------------------------------------------------------

VarOrder* BuildVariableOrder(vector<int>& encode, string
query,vector<vector<char>> decomposition,int& type){
    // create a root for variable order tree
    // use '?' symbol to distinguish it between other nodes
```

```
    VarOrder* root = new VarOrder('?');
    root->num = 0; // the number of the root is zero
    //Build a relation map according to the decomposition
    map<char,set<char>> mp = BuildRelationMap(decomposition);

    // type 1: Sum(1)
    if (type == 1){
        // insert all the attribues in R
        InsertAttributes(mp,decomposition,root,encode);
    }
    // type 2: Sum of EXPR in R
    else if (type == 2){
        // insert EXPR first
        InsertEXPR(root,encode, query,decomposition);
        // insert other attribues except EXPR
        InsertAttributes(mp,decomposition,root,encode);

    }
    // type 3: Sum of two attributes with group-by clause where the group-by
variables can be inserted on the above of other variables
    else if (type == 3)
    {
        // insert ATTR first
        InsertATTR(root,encode,query,decomposition,type);
        if (type == 3)
            // insert other attribues except ATTR
            InsertOtherAttributes(mp,decomposition,root,encode,type);
        // type 4: Sum of two attributes with group-by clause where the group-by
variables cannot be inserted on the above of other variables
        else if (type == 4){
            // insert all attributes
            InsertAttributes(mp,decomposition,root,encode);

        }
    }

    return root; // the root of the variable order tree

}
//-----------------------------------------------------------------
// Split
//
// Purpose : separate the the input tuple string by the delim into vector of
integer
// Input   : s - input tuple in the type of string.
//           delim - separator of the tuple
//
// Output : tuple in a vector form
//-----------------------------------------------------------------
vector<int> split(const string &s, char delim) {
    stringstream ss(s);
    string item;
    vector<int> tokens;
    while (getline(ss, item, delim)) { // for each line in the csv file
        tokens.push_back(stoi(item)); // convert string to integer
    }
    return tokens;
}
//-----------------------------------------------------------------
```

```
// Find
//
// Purpose : find a Node in the given vector
// Input   : left - input tuple in the type of string.
//           delim - separator of the tuple
//
// Output : tuple in a vector form
//-----------------------------------------------------------------
Node* Find(vector<Node*> left,Node* x){
    if (left.size() == 0)
        return NULL;
    for (auto l:left){
        if (l->val == x->val)
            return l;
    }
    return NULL;
}
//-----------------------------------------------------------------
// FindIndex
//
// Purpose : Return the attribute in the given position in the
          variable order
// Input   : encode - vetcor  which contains the position of each attribute in the
                              variable order.
//           x - position of the given attribute in the variable order
//
// Output : the assigned number for the attribute. -1 if not found.
//-----------------------------------------------------------------
int FindIndex(vector<int> encode,int x){
    for (int i = 1; i < encode.size() ; i++){
        if (x == encode[i]) // the attribute has the position x
            return i; // return the assigned number of the attribute
    }
    return -1; // not found
}

//-----------------------------------------------------------------
// BuildFatorisedDB
//
// Purpose : Build a factorised database
// Input   : root - the root of the database
//           tuple - the input tuple from the cover
//           encode - the position of each attribute in the variable order
//
// Output : none
//-----------------------------------------------------------------
void BuildFatorisedDB(Node* root,vector<int> tuple,vector<int> encode){

    // build a list of nodes for the input tuples
    vector<Node*> t(numAttributes);
    for (int i = 0; i < tuple.size(); i++){
        t[i+1] =new Node(tuple[i]);
    }

    vector<Node*> start(numNode); // record the match of each level
    start[0] = root;


    for (int i = 1; i <= numNode - 1; i++){
```

```
        int index = FindIndex(encode, i); // encode[] == i


        if (index != -1 ){
            if (i % 2 != 0){ //the current node should be on the left
                Node* match = NULL;
                match = Find(start[(i - 1)/2]->left,t[index]); // find the match
of the last level
                if (match == NULL){ // not found
                    Node* n = t[index]; // create a new node
                    start[(i - 1)/2]->left.push_back(n); // insert to the left
                    start[i] = n; // new match
                }
                else{
                    start[i] = match; // found
                }
            }
            else if (i % 2 == 0){//the current node should be on the right
                Node* match = NULL;
                match = Find(start[(i - 2)/2]->right,t[index]);// find the match
of the last level

                if (match == NULL){// not found
                    Node* n = t[index];
                    start[(i - 2)/2]->right.push_back(n); // insert to the right
                    start[i] = n; // new match

                }
                else{
                    start[i] = match; // found
                }
            }

        }


    }

    root = start[0];

}
//-----------------------------------------------------------------------
// SumVector
//
// Purpose : Give the sum of a vector of integers
// Input   : x - input vector
//
// Output : the sum of the vector
//-----------------------------------------------------------------------
int SumVector(vector<int> x){
    int sum = 0;
    for (auto i:x)
        sum += i;
    return sum;
}

//-----------------------------------------------------------------------
// Count
//
```

```
// Purpose : count the number of tuples in the relation using a factorised
representation of the cover
// Input   : root - the root of the factorised database
//
// Output : the number of tuples in the relation
//-----------------------------------------------------------------
int Count(vector<Node*> root){
    if (root.size() == 0) // base case
        return 1;
    else{
        int sum = 0;
        // the value of the node = the value of the left child * the value of the
right child
        for (auto i:root){
            sum +=  Count(i->left) * Count(i->right); // recursive method
        }
        return sum;
    }
}
//-----------------------------------------------------------------
// SumColumn
//
// Purpose : Give the sum of a column
// Input   : root - the root of the factorised database
//
// Output : the number of tuples in the relation
//-----------------------------------------------------------------
int SumColumn(Node* root,vector<int> encode){
    int sum = 0;
    for (auto a:root->left){
        sum += a->val * Count(a->left) * Count(a->right);
    }
    return sum;
}
//-----------------------------------------------------------------
// Sum
//
// Purpose : Give the sum of the number of tuples represeneted in a subtree
// Input   : root - the root of the factorised database
//           x - specification of the subtree
//
// Output : sum
//-----------------------------------------------------------------

int Sum(Node* root,char x){
    if (x == '1')// sum(1)
        return  Count(root->left) * Count(root->right);

    else if (x == 'L'){ // count left sub-tree
        int sum = 0;
        for (auto a:root->left){
            sum +=  Count(a->left);
        }
        return sum;
    }
    else if (x == 'R'){ // count right sub-tree
        int sum = 0;
        for (auto a:root->right){
            sum +=  Count(a->right);
```

```
        }
        return sum;
    }
    else return 0;

}

//-----------------------------------------------------------------
// GroupByType3
//
// Purpose : Compute the result for type 3 group-by clause query
// Simplification : the number of attributes are two
//                  the structure of the variable order is known
// Input   : root - the root of the factorised database
//           x - specification of the subtree
//
// Output : a map between attributes and sum
//-----------------------------------------------------------------


map<string,int> GroupByType3(Node* root){
    map<string,int> mp;

    vector<int> nums;

    // Benchmark group by AB
    for (auto b: root->left){
        nums.push_back(Sum(b,'R')); // sum the right subtree of the first
attribute

        for (auto a: b->left){
            nums.push_back(Sum(a,'L')); // Get the number of the second attribute
            int sum = 0;
            for (auto e: a->right){
                for (auto f:e->left){ // Sum(F)
                    sum += nums[0] * nums[1] * f->val; // value * number
                }
            }
            string ba = to_string(b->val) + " " + to_string(a->val); // a string
represents the selection of attributes

            mp[ba] = sum; // mapping between attributes and sum

        }
    }
    return mp;



}
//-----------------------------------------------------------------
// GroupByType4
//
// Purpose : Compute the result for type 4 group-by clause query
// Simplification : the number of attributes are two
//                  the structure of the variable order is known
// Input   : root - the root of the factorised database
//
// Output : a map between attributes and sum
```

```
//-------------------------------------------------------------------
map<string,int> GroupByType4(Node* root){
    map<string,int> mp;

    for (auto a:root->left){
        //get value of F
        int fv = 0;
        for (auto e:a->right)
            for (auto f:e->left)
                fv += f->val;

        //get value of C & D
        for (auto b:a->left){
            for (auto c:b->left){
                for (auto d:b->right){
                    string cd = to_string(c->val) + " " + to_string(d->val);
                    mp[cd] += fv;
                }
            }
        }
    }

    return mp;
}
//-------------------------------------------------------------------
// Benchmark
//
// Purpose : List the four test cases
// Input   : none
// Output : a vector of queries
//-------------------------------------------------------------------
vector<string> Benchmark(){
    vector<string> queries;
  //  queries.push_back("SELECT SUM(1) FROM R;");
    //queries.push_back("SELECT SUM(A) FROM R;");
    queries.push_back("SELECT A,B, SUM(F) FROM R GROUP BY A,B;");
    queries.push_back("SELECT C,D, SUM(F) FROM R GROUP BY C,D;");
    return queries;
}
int pos1 = -1;
int pos2 = -1;
int pos3 = -1;
struct
{
    bool operator()(const vector< int >& lhs, vector< int >& rhs)
    {
        return ((to_string(lhs[pos1]) < to_string(rhs[pos1])) ||
((to_string(lhs[pos1]) == to_string( rhs[pos1])) && (to_string(lhs[pos2]) <
 to_string(rhs[pos2])) ));
    }
} ColumnLess;
void StandardDB(){

    vector<string> queries = Benchmark() ;

    /* StandardDB*/
    cout << "StandardDB" << endl;
    /* read Cover */
    ifstream R("dsi-exam18/5/R.csv");
```

```
    vector<vector<int>> SDB;
    string relation;
    while (getline(R, relation, '\n')){
        vector<int> tuple = split(relation,'|');
        SDB.push_back(tuple);
    }

    for (auto q:queries){

        int type = QueryType(q);
        if (type == 1){
            int cnt = (int) SDB.size();
            cout << "Answers to query 1" << endl;
            cout << cnt << endl;
        }
        else if (type == 2){
            int begin = (int)q.find("SUM(") + 4; // position before "SUM"
            string attr = q.substr(begin,1); // sub string between SELECT and SUM

            int pos = attr[0] - 'A';
            int sum = 0;
            for (int i = 0; i < SDB.size(); i++){
                sum += SDB[i][pos];
            }
            cout << "Answes to query 2" << endl;
            cout << sum << endl;
        }
        else{
            // type 3 or 4
            pos1 = -1;
            pos2 = -1;
            int begin = (int)q.find(" ") + 1; // position after "SELECT "
            int end = (int)q.find("SUM") - 1; // position before "SUM"

            string attr = q.substr(begin,end-begin); // sub string between SELECT
and SUM

            for (int i = 0; i < attr.length(); i++)
                if (attr[i] >= 'A' && attr[i] <= 'Z') // alphabets
                {

                    if (pos1 == -1 )
                    {pos1 = attr[i] - 'A';}
                    else
                        pos2 = attr[i] - 'A';
                }


            sort(SDB.begin(), SDB.end(),ColumnLess );

            int a1 = SDB[0][pos1];
            int b1 = SDB[1][pos2];
            map<string,int> mpstd;
            int sum = 0;

            for (int i = 0; i < SDB.size(); i++){
                if (SDB[i][pos1] == a1 && SDB[i][pos2] == b1){
                    sum += SDB[i][5];
```

```
                }
                else{
                    string ab = to_string(a1) + " " + to_string(b1);
                    mpstd[ab] = sum;
                    sum = SDB[i][5];
                    if (SDB[i][pos1] != a1)
                        a1 =SDB[i][pos1];
                    if (SDB[i][pos2] != b1)
                        b1 =SDB[i][pos2];
                }
            }
            string ab = to_string(a1) + " " + to_string(b1);
            mpstd[ab] = sum;
            sum = 0;

            map<string,int> :: iterator iter2;
            for (iter2 = mpstd.begin(); iter2 != mpstd.end(); iter2 ++){
                cout <<iter2->first << " " << iter2->second << endl;
            }

            cout << "end for query" << endl;

        }
    }
    SystemInfo();

}
void CoverDB(){

    /* parameters */
    string cover; // read in the cover: one tuple at a time
    string query;
    vector<vector<char>> decomposition = {{'A','B','C'},{'A','B','D'},{'A','E'},
{'E','F'}};
    vector<int> encode(numNode);
    vector<string> queries = Benchmark() ;
    cout << "CoverDB" << endl;
    vector<vector<int>> CDB;
    for (auto q:queries){
        int type = QueryType(q);

        /* Build variable order */
        VarOrder* VORoot = BuildVariableOrder(encode,q,decomposition,type);


        /* read Cover */
        ifstream C("dsi-exam18/7/C.csv");
        if (!C)
            cout << "Cannot open files" << endl;
        else{

            Node* root = new Node(-1);
            while (getline(C, cover, '\n')){
                vector<int> tuple = split(cover,'|');
                CDB.push_back(tuple);
                /* Build factorised database */
            }
            pos1 = q[7];
            pos2 = q[9];
```

```
            if (q[7] == 'A' || q[7] == 'C')
             sort(CDB.begin(),CDB.end(),ColumnLess);


             for (auto tuple:CDB)
                 BuildFatorisedDB(root,tuple,encode);

             /* answers to the query */
             if (type == 1){
                 cout << "Answers to query 1" << endl;
                 cout << Sum(root,'1') << endl;

             }
             else if (type == 2){
                 cout << "Answes to query 2" << endl;
                 cout << SumColumn(root,encode) << endl;
             }
             else if (type == 3){
                 cout << "Answers to query 3" << endl;
                 map<string, int> mp = GroupByType3(root);
                 map<string,int> :: iterator iter;
                 for (iter = mp.begin(); iter != mp.end(); iter ++){
                     cout <<iter->first << " " << iter->second << endl;
                 }
             }
             else if (type == 4){
                 cout << "Answer for query 4" << endl;
                 map<string, int> mp = GroupByType4(root);
                 map<string,int> :: iterator iter;
                 for (iter = mp.begin(); iter != mp.end(); iter ++){
                     cout <<iter->first << " " << iter->second << endl;
                 }

             }
             delete root;

         }
         delete VORoot;


     }
     SystemInfo();

}
int main(int argc, const char * argv[]) {
   clock_t t1,t2;
    float diff;

    t1=clock();
    CoverDB();
    t2=clock();
     diff = ((float)t2-(float)t1)/CLOCKS_PER_SEC;
    cout<< diff <<endl;

    t1=clock();
    StandardDB();
    t2=clock();
    diff = ((float)t2-(float)t1)/CLOCKS_PER_SEC;
    cout<< diff <<endl;
```

```
    return 0;
}
```