Journal of Computer Science and Cybernetics, V.32, N.1 (2016), 45–58
DOI: 10.15625/1813-9663/32/1/5772

HEURISTIC ALGORITHM FOR FRAGMENTATION AND ALLOCATION IN DISTRIBUTED OBJECT ORIENTED DATABASES

MAI THUY NGA[1], DOAN VAN BAN[2]

[1]*Information Technology Department, Thang Long University, mai_nga@yahoo.com,*
[2]*Institute of Information Technology, Vietnam Academy of Science and Technology*
[*]*Corresponding author: MAI THUY NGA; mai_nga@yahoo.com*

Abstract. Class fragmentation and allocation are important techniques to improve the performance of distributed object oriented database systems. A class fragmentation schema, which is to split a class into smaller pieces in distributed databases, intends to reduce the unnecessary data access, while an allocation algorithm is to locate fragmented classes into sites in a connected network in such the way that minimizes the cost of data transmission. A class in object databases consists of attributes describing the characteristics of the object, methods describing the behavior and relationships with other classes. With such characteristics, class fragmentation and allocation in distributed object oriented database systems are more complex than those in relational databases. Fragmentation techniques applied in distributed object-oriented databases often do not take into account communication costs between sites; fragments are allocated to the site after completing a fragmentation option of data objects. This paper proposes an algorithm of simultaneous class fragmentation and allocation in which the costs of data communication between the sites are used for fragmentation to reduce communication costs when processing and querying distributed data.

Keywords. Object oriented database, distributed database, distributed object-oriented database, fragmentation, allocation, class fragmentation, class allocation.

1. INTRODUCTION

Object-oriented databases (OODBs) have been widely used in practice and have overcome the limitations of relational databases. However, with the basic characteristics of object-oriented technologies such as encapsulation, inheritance, class hierarchies, OODBs require new techniques and effective data management [1]. OODBs were developed in network environments then made up of distributed object-oriented databases (DOODBs). In DOODBs, data is distributed over sites in a computer network, while applications have to access and process data at different sites. In fact, how to design DOODBs effectively in order to improve their overall system performances has recently received great interest from numerous researchers.

The problem of DOODB design is divided into two phases: (1) data fragmentation to split the data into smaller pieces, and (2) data allocation to allocate such data pieces into the respective sites. Fragmentation in DOODB is performed on object classes by using two techniques: vertical and horizontal fragmentation. Vertical fragmentation aims at dividing a class into smaller fragments, each of which includes a number of attributes and methods. Horizontal fragmentation is to divide objects in the same class into different fragments; each fragment is composed of some objects. Our work focuses on vertical fragmentation algorithm.

Many algorithms for vertical fragmentation have been proposed for relational model, such as the algorithms presented in [1–5]. In object models, fragmentation raises few other issues due to the complex characteristics of object-oriented methods; Karlapalem and Li [6] introduced a partition scheme for OODBs, Ezeife and Barker [7] presented a vertical fragmentation algorithm for each case of attributes and methods, Lee and Lim [8] presented a fragmentation algorithm based on attributes and Saravanan and Rajan [9] proposed a vertical fragmentation algorithm using intelligent agents. The allocation problem in the DOODBs has been studied in [10–12] as well.

In the aforementioned approaches, the fragmentation and allocation are conducted in two consequence phases, fragmentation is executed first, then locating fragments into corresponding sites. The fragmentation phase does not use data transimission cost information between sites. This information can be exploited for a more effectively fragmenttaion-solution for DOODBs.

Inspired by the approach from Hui Ma and Markus [13] applied to relational database, a new heuristic algorithm to fragment and allocate classes simultaneously in DOODBs is proposed. The method considers the costs of information transmittion between sites during querying information in order to generate a plan for vertical fragmentation and allocation. This heuristic approach can increase the effectiveness of DOODBs.

This paper consists of six sections. Section 2 introduces the vertical fragmentation and allocation in DOODBs. Section 3 presents the required information which will be used for fragmentation and allocation. Section 4 introduces the cost model, referring to the cost calculation formula as a basis for decision heuristic option. Section 5 proposes the algorithm with an illustration. The last section concludes the paper and discusses future developments.

2. CLASS VERTICAL FRAGMENTATION AND ALLOCATION

2.1. OODB Model

The data in the OODBs consists of a set of objects to be encapsulated, each of which includes attributes and methods. The object is instanced from the class, inherited in the hierarchy level. A class in a hierarchical relationships is represented by $C = (K,\ A,\ M,\ I)$, where K is the set of identifiers, A is the set of attributes, M is a set of methods and I the set of objects as defined by A and M [7].

Attributes of a class are divided into two categories: simple and complex. Simple attribute is the attribute that has a value domain of primitive types such as int, long, float, double, boolean, char, string, ... Complex attribute has a value domain that is not primitive but is a reference to other objects through their identity.

Methods of a class are also divided into two categories: simple and complex. Simple method does not invoke or refer to other methods while complex method invokes other methods in the same class or methods of other classes.

2.2. Vertical fragmentation

The goal of vertical fragmentation of classes is to break them up into smaller pieces (called class fragments), each containing only some of the attributes and methods. Vertical fragmentation of class C =(K, A, M, I) is the set of fragments $\{F_1, F_2, ..., F_n\}$, each fragment $F_v = \{K, A_v, M_v, I\}$,

$v = 1, ..., n$, where A_v is a subset of A, M_v is a subset of M. In vertical fragmentation, it is to concentrate only attributes and methods of classes, the remaining factors including identifiers and objects are considered in horizontal fragmentation.

Vertical fragmentation has to ensure the following three rules.

- Completeness: Each data item (attribute or method) of class C is found in one or more fragment F_v.
- Reconstruction: Class C can be reconstructed from its fragments
- Disjointedness: Only identifiers and methods accessing identifiers are typically repeated in all its fragments, each other data item of class C is only in one fragment F_v.

The class fragmentation enables the application to execute on a certain number of fragments instead of accessing to the whole class as the other fragments are not necessary for these application's queries. Vertical fragmentation splits attributes and methods into groups that has the highest possible access frequency and thus optimizes the application's query execution time.

The class is encapsulated the access is only executed on methods without manipulating directly to attributes. To find a suitable fragmentation, first the authors group methods based on query information and once the fragmentation is complete, the authors add to each method group all attributes accessed by methods of group...

2.3. Allocation

Allocation in DOODBs is to locate and distribute class fragments into respective sites in a connected network. Assuming the database is divided into fragments $F = \{F_1, ..., F_v\}$ and a distributed system consists sites $S = \{s_1, ..., s_k\}$ on which a set of application $Q = \{q_1, \dots q_h\}$ is running. The allocation problem is to find the "optimal" distribution of F to S.

The allocation problem in DOODBs involves the allocation for both methods and classes. The problem of method allocation is almost the same as the class allocation problem due to the encapsulation characteristic of object-oriented methods. The allocation for methods that accesses multiple classes at different sites is a difficult problem; this problem has been proven to be NP-complete [6].

The application accessing to the attributes and methods of a class C is divided into three categories:

- Applications execute directly on the class C.
- Applications execute on subclasses of the class C.
- Applications execute on methods of other classes in the database and use methods of the class C.

The approach to the allocation problem is to reduce the cost of data allocation in distributed systems. Cost allocation is the sum of all cost components: the cost of data storage, the cost of query processing and the cost of data transmission between the sites

3. INFORMATION REQUIREMENTS FOR VERTICAL FRAGMENTATION AND ALLOCATION

3.1. Database Information

Database information is a collection of classes, their structures and relationships. Relationships in OODBs are inherited and composite. Composite relationships present complex attributes. Each class in the OODBs encapsulates attributes and methods; a method is one interface of an object to interact with other objects outside the class. A method that uses only attributes of the class called the simple one whereas a method that calls methods in the same class or another class considered complex

Example: Assuming object database has following classes: *Person*, *Professor*, *Staff, Visiting-Prof*, *Faculty* and *Course* with the following characteristics:

Class *Professor* and class *Staff* are inherited from class *Person*, class *VisitingProf* is inherited from class Professor.

Class *Faculty* contains an attribute which is a collection of class *Professor* and class *Professor* contains an attribute which is a collection of class *Course*.

All classes in this example are structured with methods and attributes as below

- Person={{ssno, name, birth, addr}, {getSsno(), getName(), getBirth(), getAddr()}}
- Professor = {{profID, profType, facultyOf, courses}, {displayID_TypeOfProf(), getFaculty(), displayCourses(), calculateTeachingHours()}}
- VisitingProf = {{organisation}, {getOrg()}}
- Staff = {{staffID, unit}, {getStaffID(), getUnit()}}
- Faculty = {{name, addr, head}, {getName(), getAddr(), getHead()}}
- Course = {{ID, hours}, {getID(), getHours()}}

Class *Faculty* is a class that contains class *Professor* because the attribute *head* is one object of class *Professor*; similarly class *Professor* contains class *Course*. Class *Professor* contains method *calculateTeachingHours()* which is referenced in method *getHours()* of class *Course*.

Methods use the class's attributes to execute necessary calculations, and such attribute usage by different methods in one class can be formed up a matrix called MAU (Method Attribute Usage) [8]. In matrix MAU, the title of rows and columns as the methods and attributes, a cell value of 1 indicates the method (in a row) accessing to that attribute (in the respective column of the cell), the opposite is 0. Table 1 is an example of a matrix MAU of the class *Professor*.

Table 1. Matrix MAU of Class Professor

	a_1 **(ProfID)**	a_2 **(ProfType)**	a_3 **(facultyOf)**	a_4 **(Courses)**
m_1 **(displayID_TypeOfProf())**	1	1	0	0
m_2 **(m.getDept())**	0	0	1	0
m_3 **(displayCourses())**	0	0	0	1
m_4 **(calculateTeachingHours())**	0	0	0	1

3.2. Network Information

As said in above section, the different aspect of this paper is to take the cost of communication into account during the fragmentation and allocation. The additional following matrix represents the cost of communication between sites called SSC (Site Site Cost). An example of a cost matrix between sites is in Table 2.

Table 2. Matrix SSC

	s_1	s_2	s_3
s_1	0	50	70
s_2	50	0	30
s_3	70	30	0

In the previous approaches, site information is only used in the allocation phase, in this research the site information will be also used in fragmentation phase.

3.3. Application information

As the object encapsulation characteristic the application is only able to query objects via methods. The method used by the query is represented by the value of QMU (Query Method Usage).

QMU (q_i, m_j) with value 1 if the query q_i uses the method m_j, otherwise QMU (q_i, m_j) has a value of 0. Suppose there are 3 queries q_1, q_2, q_3 accessing to class Professor.

- q_1: give the social security number and courses list of one professor with the specific ID
- q_2: give ID and type of all professors who work for a specific faculty.
- q_3: find all professors who have less than 50 teaching hours.

Matrix QMU of class Professor is in Table 3.

Table 3. Matrix QMU of class professor

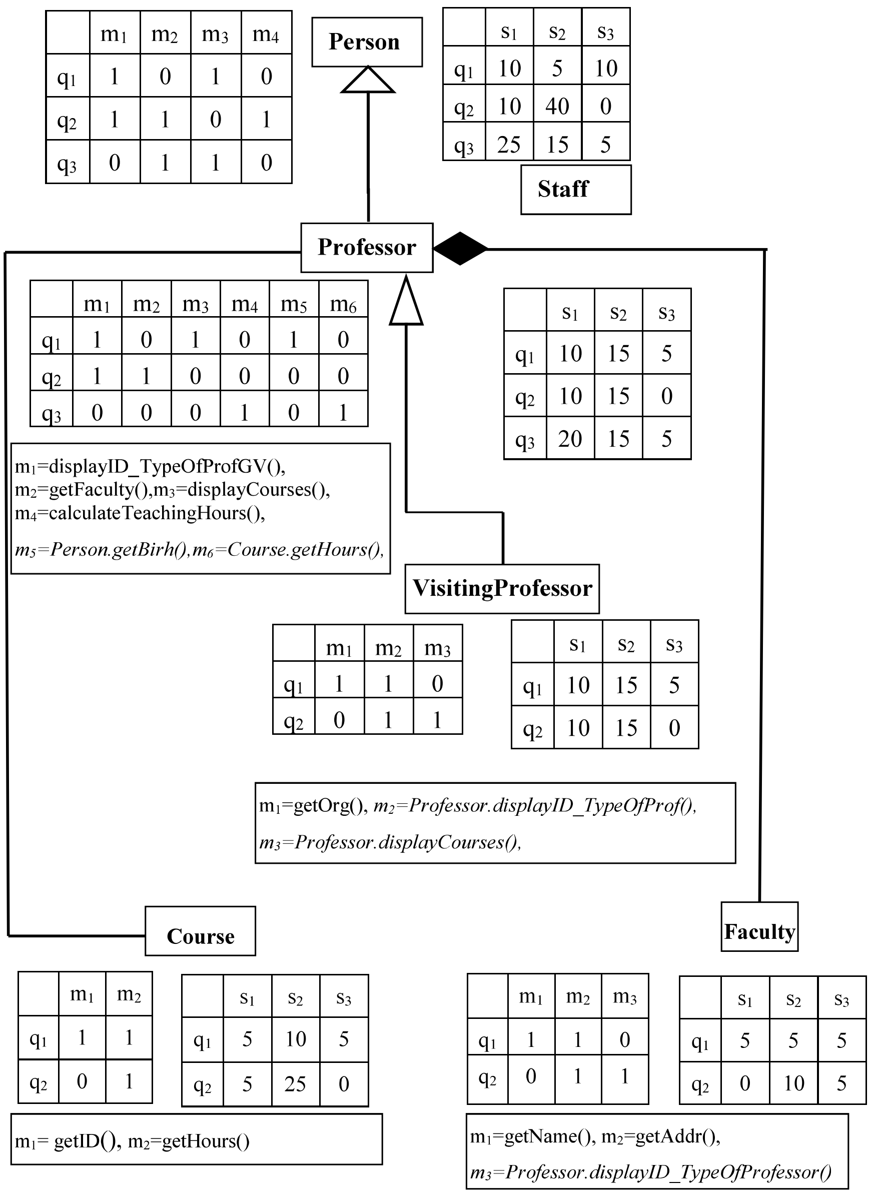
	m_1	m_2	m_3	m_4	m_5 (**Person.getSsno()**)	m_6 (**Course.getHours()**)
q_1	1	0	1	0	1	0
q_2	1	1	0	0	0	0
q_3	0	0	0	1	0	1

Information about Access frequency of queries to sites is represented by the value QSF (Query Site Frequency). QSF(q_i, s_l) is the access frequency of query q_i to site s_l, an example about access frequency matrix sites are shown in Table 4.

Matrix QMU and matrix QSF of that application at sites in the database are shown in Figure 1. Note that queries into 2 classes *Professor* and *Satff* will inherit some methods from the super-class *Person*, so the matrix QMU of class *Professor* and *Satff* will have to add those methods.

Table 4. Matrix QSF of class Professor

	s_1	s_2	s_3
q_1	10	5	10
q_2	5	40	0
q_3	25	15	5



	m_1	m_2	m_3	m_4
q_1	1	0	1	0
q_2	1	1	0	1
q_3	0	1	1	0

	s_1	s_2	s_3
q_1	10	5	10
q_2	10	40	0
q_3	25	15	5

	m_1	m_2	m_3	m_4	m_5	m_6
q_1	1	0	1	0	1	0
q_2	1	1	0	0	0	0
q_3	0	0	0	1	0	1

m_1=displayID_TypeOfProfGV(),
m_2=getFaculty(),m_3=displayCourses(),
m_4=calculateTeachingHours(),
m_5=Person.getBirh(),m_6=Course.getHours(),

	s_1	s_2	s_3
q_1	10	15	5
q_2	10	15	0
q_3	20	15	5

	m_1	m_2	m_3
q_1	1	1	0
q_2	0	1	1

	s_1	s_2	s_3
q_1	10	15	5
q_2	10	15	0

m_1=getOrg(), *m_2=Professor.displayID_TypeOfProf(),*
m_3=Professor.displayCourses(),

	m_1	m_2
q_1	1	1
q_2	0	1

	s_1	s_2	s_3
q_1	5	10	5
q_2	5	25	0

m_1= getID(), m_2=getHours()

	m_1	m_2	m_3
q_1	1	1	0
q_2	0	1	1

	s_1	s_2	s_3
q_1	5	5	5
q_2	0	10	5

m_1=getName(), m_2=getAddr(),
m_3=Professor.displayID_TypeOfProfessor()

Figure 1. Matrice QMU and QSF of classes.

Matrix QMU of class *Professor* has 6 columns. The first 4 columns show the four access methods m_1, m_2, m_3, m_4 of that class *Professor*, the fifth column shows the access to method m_5 of the class

Person, the last column shows the access method m_6 of the class *Course.* Suppose that there is no query in class *Satff.* In Figure 1, methods are not primitive methods of that class are italicized. Also note that indexes of methods and queries belongs to each separated class (m_i of class *Person* is different from m_i of class *Professor*, q_i of class *Person* is different from q_i class *Professor*).

4. COST MODEL

4.1. Cost Model

The most significant cost in DOODBs is the cost of data transmission between sites. The cost function needs identifying to calculate the total cost of method invocation on the remote site. Cost of calling a method is represented by the total cost of the communication between the sites that contains the method being called (used) and the site that makes a call to the other site. Also, on calculating the time taken by any method, its type will need to be determined first. If it is a simple method, the cost only consists of transmission cost of method resulting from the method to the calling site. If it is a complex method, it is required add the cost of the method that is invoked. It must be noted further about the communication required by a remote object, this happens when a method requires the object on a remote site to perform so the cost of communication objects should be added. In summary, the cost of any method also includes:

- Cost of data result returned from calling site.
- Cost of the method that is invoked.
- Cost of remote communication objects.

In the database with many classes, to distinguish the parameters of each class we need to add index of the class, for example m^i_j to indicate the method m_j in class C^i. The symbols used are described in Table 5.

Table 5. Usage symbols

Symbol	*Description*
C	Set of classes (database)
C^i	i^{th}class of data base.
M^i	Set of methods of Class C_i.
m^i_j	j^{th}method of class C_i.
A^i	Set of attributes of Class C_i.
a^i_j	j^{th}attribute of class C_i.
Q^i	Set of queries of class C_i.
q^i_j	j^{th} query of class C_i.
S	Set of sites.
s_k	k^{th} site.
MAU^i	Method Attribute Usage matrix of class C_i.
QMU^i	Query Method Usage matrix of class C_i.
QSF^i	Query Site Frequency matrix of class C_i.
SSC	Site Site Cost Matrix.
F^i	Fragments Set of class C_i
F^i_j	j^{th}fragment of class C_i.

In this paper, it is to extend the formula of Hui Ma and Markus [13] applied to relational model to calculate the cost. The *request* of a method m_j^i at a site s_k is the total access frequency of all queries at the site s_k accessing the method m_j^i, this value can be calculated as follows

$$request^i(s_k, m_j^i) = \sum_{q_l^i \in Q^i} QSF^i(q_l^i, s_k) * QMU^i(q_l^i, m_j^i)$$

Request value is calculated based on the following parameters: queries frequency at sites and methods usage of queries. These two parameters are modified belong to the relationships in OODBs (inheritance, complex attribute, complex method), algorithms 1, 2, 3 in subsection 4.2 indicate this modification. To set up accessing cost for each method of class C_i from the site the matrix $request^i$ will be built, this matrix is the product of two matrices QSF^i và QMU^i

Cost for locating method m_j^i into site s_k is the cost of accessing this method m_j^i from all other sites $s_l \neq s_k$, that is defined as follows:

$$pay^i(s_k, m_j^i) = \sum_{s_l \in S} request^i(s_k, m_j^i) * SSC(s_k, s_l)$$

Pay value is calculated based on *Request* value and network information, so *Pay* value depends on queries frequency, method usage of queries, and communication cost between sites. To set up cost for locating each method of class C^i into sites the matrix pay^i will be built, this matrix is the product of two matrices $request^i$ and SSC.

Based on the matrix pay^i to define a plan to locate methods of class C^i, the authors propose a heuristic algorithm that aims to locate method m_j^i into site s_k with value of $pay^i(s_k, m_j^i)$ smallest.

4.2. Modification of method usage and site access frequency

Before calculating the value of a *request* by the formula (1) it is required to transform matrices QMU and QSF according to relationships in DOODBs because these relationships affect the fragmentation and allocation. The following relationships are considered for the modification process: inheritance and include relationships and complex method.

Firstly, consider about the inheritance relationship. Queries on inherited classes can fully use methods from superclasses, so it is needed to provide this additional information about the queries into QSF and QMU matrices of the superclass. Algorithm 1 describes this modification.

```
**Algorithm 1:** *Modify QMU$^i$ and QSF$^i$ according to the inheritance relationship*
**Algorithm** *Modify_1($C^i$)*
*Input:* Set of class $C$, Class $C^i$ need fragmented in database of classes, and their matrices QMU and QSF.
*Output: QMU$^i$ and QSF$^i$.*
*Algorithm Steps:*
for each $C^h \in C$ do
    if $C^h$ inherited from class $C^i$do
        for each $q_k^h \in$Q$^h$ do //queries on class $C^h$
            if $q_k^h$ uses methods $m_j^i$ of class $C^i$
                begin
                    //Add a row corresponding $q_k^h$ to QMU$^i$;
                    //Add a row corresponding $q_k^h$ to QSF$^i$;
                AddRow($q_k^h$, QMU$^i$, QSF$^i$);
            end {if $q_k^h$}
        end {for $q_k^h$}
end {for $C^h$}
end {Algorithm 1}
```

Next is to consider include relationship, also called "container" (in classes with complex attributes). Queries on the container class possibly use the methods of the class are contained within, so we need provide these additional information about these queries into QSF and QMU matrices of contained class. Algorithm 2 describes this modification.

Algorithm 2: *Modification QMU^i and QSF^i according to the include relationship*
Algorithm *Modify_2(C^i)*
Input: Set of class C, Class C^i need fragmented in database of classes, and their matrices QMU and QSF.
Output: QMU^i and QSF^i.
Algorithm Steps:

```
for each $C^h \in C$ do
    if $C^h$ is a container of class $C^i$
    // $C^h$ has one attribute which is an object of class $C^i$
        for each $q_k^h$ ∈Q$^h$ do //queries on class C$^h$
            if $q_k^h$ uses methods $m_j^i$ of class C$^i$
                begin
                    //Add a row corresponding $q_k^h$ to QMU$^i$;
                    //Add a row corresponding $q_k^h$ to QSF$^i$;
                    AddRow($q_k^h$, QMU$^i$, QSF$^i$);
                end {if $q_k^h$}
        end {for $q_k^h$}
end {for $C^h$}
end { Algorithm 2}
```

Finally consider the complex methods, this is a case that a method in one class can invoke methods of this class or another class. Queries on the class containing complex method can fully use methods from other classes, so it is needed to provide this additional information about these queries into QMU and QSF matrices of the class containing methods which is called. Algorithm 3 describes this modification.

Algorithm 3: *Modification QMU^i and QSF^i according to the complex method*
Algorithm *Modify_3(C^i)*
Input: Set of class C, Class C^i need fragmented in database of classes, and their matrices QMU and QSF.
Output: QMU^i and QSF^i.
Algorithm Steps:

```
for each $C^h \in C$ do
    for each $m_j^h$ ∈ M$^h$ do
        if $m_j^h$ invokes $m_l^i$ //method of C$^h$ invoke method of C$^i$
            for each $q_k^h$ ∈ Q$^h$ do //queries on class C$^h$
```

```
                    if $q_k^h$ uses methods $m_j^h$ of class C$^h$
                        begin
                            //Add a row corresponding $q_k^h$ to QMU$^i$;
                            //Add a row corresponding $q_k^h$ to QSF$^i$;
                            AddRow($q_k^h$, QMU$^i$, QSF$^i$);
                        end {if $q_k^h$}
                end {for $q_k^h$}
        end {for $m_j^h$}
    end {for C$^h$ }
    end {Algorithm 3}
```

Algorithm 0 describes the addition a row corresponding q_k^h to QMU^i and QSF^{i}which is used in three algorithms above.

```
Algorithm 0: Add a row corresponding $q_k^h$ to QMU$^i$ and QSF$^i$
Algorithm AddRow($q_k^h$,QMU$^i$, QSF$^i$)
Input: $q_k^h$, QMU$^i$ and QSF$^i$
Output: QMU$^i$ and QSF$^i$.
Algorithm Steps:
$Q_i = Q_i \cup \{q_k^h\}$
for each $m_j^i \in$ M$^i$ do
    if $q_k^h$ use $m_j^i$
        QMU$^i$($q_k^h$,$m_j^i$) = 1;
    else
        QMU$^i$($q_k^h$,$m_j^i$) = 0;
    end {if $q_k^h$}
end {for $m_j^i$}
for each s$_l \in$ S
    QSF$^i$($q_k^h$,s$_l$) = QSF$^h$($q_k^h$,s$_l$)
end {for s$_l$}
end {Algorithm 0}
```

5. ALGORITHM OF FRAGMENTATION AND ALLOCATION SIMULTANEOUSLY

5.1. Algorithm development

Fragmentation and allocation algorithm proposes the heuristic approach as follows: Based on the matrix pay^i, allocating methods m_j^i to site s$_k$ where the communication cost is the smallest. Propose a plan for allocation, then clustering methods in the same site in one fragment. In each fragment, with each method the authors will determine attributes that these methods used then put them into this fragment. Algorithm of fragmentation and allocation is simultaneously constructed as follows.

Algorithm 4: Fragmentation and Allocation

Algorithm Frgamentation_Allocation(C^i)

Input: Class Ci need fragmented in database of classes, matrices MAU, QMU, QSF of classes, matrix SSC.

Output: Vertical fragmentation and allocation for class C^i.

```
Algorithm Steps:
//Step 1: Modify matrices QMU and QSF of class $C^i$ according to //relationships
Modify_1($C^i$);
Modify_2($C^i$);
Modify_3($C^i$);
//Step 2: Build matrix request$^i$ of class $C^i$ by multiplying 2
//matrices QMU$^i$ and SQF$^i$(SQF$^i$ is transposed matrix of QSF$^i$).
request$^i$ = Multiple2Matrix (QMU$^i$, SQF$^i$);
//Step 3: Build matrix pay$^i$ of class C$^i$ by multiplying 2 matrices
//request$^i$ and SSC.
pay$^i$ = Multiple2Matrix (request$^i$, SSC);
//Step 4: Determine the allocation plan based on matrix pay$^i$
//Find the smallest value in each column of matrix pay$^i$
for each $m_j^i \in$ M$^i$ do
    begin
        Find s$_k$ which $pay^i(s_k, m_j^i)$ is the smallest;
        Allocate $m_j^i$ into site $s_k$;
        Add $m_j^i$ to F$_k$;
        //Based on matrix MAU, add attributes to fragments
        for each $a_l^i \in A^i$ do
            if (MAU($m_j^i$, $a_l^i$) = 1 ) and ($a_l^i$ is not in a fragment)
                begin
                    Add $a_l^i$ to fragment that has $m_j^i$
                    Allocate $a_l^i$ into the site that $m_j^i$ is allocated
                end {if}
        end {for $a_l^i$}
    end {for $m_j^i$}
Add identifier to each fragment.
Add method accessing identifier to each fragment.
end {Algorithm 4}
```

5.2. Algorithm illustration

This example just creates fragmentation for class Professor so the upper index of parameters is temporarily removed.

When considering queries to the class *VisitingProf*, q_1 and q_2 queries have access to methods m_1 and m_3 of class *Professor*, therefore it will add two rows to QMU matrix corresponding to q_4 and q_5 (which are q_1 and q_2 of class *VisitingProf*). When considering queries to class *Faculty*, query q_2 has access to the method m_1 of class *Professor* so it will add one row to QMU matrix corresponding to q_6 (which is q_2 of class *Course*).

Matrix QSF is also added 3 rows correspondingly for the queries considered above. Matrices QMU and SQF (SQF is transposed matrix of QSF) of class Professor in Table 3 and Table 4 will be changed as Tables 6 and 7. Multiple matrix SQF with matrix QMU to build matrix *request* as Table 8. Multiple matrix SSC in Table 2 with matrix *request* to have matrix *pay* as Table 9.

Table 6. Matrix QMU after modification

	m_1	m_2	m_3	m_4	m_5	m_6
q_1	1	0	1	0	1	0
q_2	1	1	0	0	0	0
q_3	0	0	0	1	0	1
q_4	1	0	0	0	0	0
q_5	1	0	1	0	0	0
q_6	1	0	0	0	0	0

Table 7. Matrix QSF after modification (order by column vs Table 4)

	q_1	q_2	q_3	q_4	q_5	q_6
s_1	10	10	20	10	10	0
s_2	15	15	15	15	15	15
s_3	5	0	5	5	5	5

Table 8. Matrix request

	m_1	m_2	m_3	m_4	m_5	m_6
s_1	40	10	20	20	10	20
s_2	75	15	30	15	15	15
s_3	20	0	10	5	5	5

Table 9. Matrix pay

	m_1	m_2	m_3	m_4	m_5	m_6
s_1	5150	750	2200	1100	1100	1100
s_2	2600	500	1300	1150	650	1150
s_3	5050	1150	2300	1850	1150	1850

The allocation plan is as follows: m_1 is located in s_2, m_2 located in s_2, m_3 located in s_2, m_4 located in s_1, m_5 located in s_2, and m_6 located s_1. Thus m_4 and m_6 are grouped to the same fragment, $m_1, m_{2,} m_3$, and m_5 are grouped to the 2^{nd} fragment. Because m_1 is the method accessing identifier, m_1 is added to the fragment containing m_4 and m_6.

So these methods are fragmented vertically into two fragments:

$$F_1 = \{m_1, m_4, m_6\},\ F_2 = \{m_1, m_2, m_3, m_5\}.$$

Based on matrix MAU in table 1, the algorithm continues allocating a_1, a_2, a_4 to F_1, and $a_1, a_2, a_{3,} a_4$ to F_2. The final result is $F_1 = \{a_1, a_2, a_4, m_1, m_4, m_6\}$, $F_2 = \{a_{1,} a_2, a_3, a_{4,} m_1, m_2, m_3, m_5\}$.

5.3. Algorithm Evaluation

Fragmentation algorithm is correct because it satisfies the three basic rules of fragmentation: Each attribute or method of a class is in one fragment; the class can be re-structured from its fragments because all of its fragments have the same identifier and identifier-accessing method; except identifier and identifier-accessing method, the remaining attributes and methods belong to just one fragment. The maximum number of fragments according to this algorithm is only as the number of sites.

This algorithm complexity is defined as follows. If a class that need fragmented has set of attributes A, set of methods M, set of queries Q and set of sites S then the complexity of algorithm 1 and algorithm 2 is $|C| * |Q|$, the complexity of algorithm 3 is $|C| * |M| * |Q|$. The complexity of multiplying two matrices (step 3 of algorithm 4) is $|M| * |Q| * |S|$. The complexity of step 4 of algorithm 4 is $|M| * |S| + |M| * |A|$. The complexity of algorithm 4 is $(|C| * |Q| + |C| * |Q| + |C| * |M| * |Q| + |M| * |Q| * |S| + |M| * |S| + |M| * |A|)$ which can simplify to $(|C| * |M| * |Q| + |M| * |Q| * |S|)$. Moreover, the algorithm of multiplication of two $N * N$ matrices is currently enhanced [14] to achieve the complexity at approximately $N^{2.38}$, so the complexity of algorithm 3 is only $N^{2.38}$, of which N is the max value of the following 3 values: $|M|, |Q|, |S|$.

The key note in this algorithm is that it is able to complete both phases of fragmentation and allocation with this complexity meanwhile other algorithms are able to complete only one phase, e.g. the complexity of Ezeife [7] algorithm for vertical fragmentation is $|C| * |Q| * |M| + |M| * |M| + |F| * |M| * |A|$ and Barker & Bhar [12] algorithm for allocation is $|S|^3|Q| + |F|^3/|S|^3$whereas $|F|$ is the number of fragments (so it is certain that $|F| > |C|$).

The authors have conducted the experiment on some test data and get the same fragmentation result as the algorithm [7] without spending any cost for allocation algorithm. The authors have also implemented the testing with The OO7 benchmark [15] and get the same fragmentation result as the algorithm [16].

In conclusion, there are some advantages of this heuristic approach:

- Only the method accessing the identifier is repeated in all fragmentations
- Information of queries and network is used in both fragmentation and allocation. In [13], Hui Ma et al presented examples about queries information (including the site that the query is executed) affecting the optimized plan for the fragmentation and allocation.
- The algorithm's complexity is low and depends on the number of classes, methods, queries, and sites.
- The maximum number of fragments according to this algorithm is only as the number of sites.

6. CONCLUSIONS AND FUTURE DEVELOPMENT

Most algorithms ever divide fragmentation and allocation to two separated phases, fragmentation completed before coming to allocation. The fragmentation phase does not consider cost for communication between sites, this cost is determined only when performing allocation. However, the allocation is only able to be optimized when the fragmentation takes into account the cost of communication between sites to achieve the least cost. The study proposes a heuristic algorithm that has both fragmentation and allocation performed simultaneously hence makes object fragmentation more efficiently. The algorithm has considered all of the class structure and relationships between the classes in the OODBs.

In the future, the authors will focus on the queries processing in distributed object oriented databases.

REFERENCES

[1] M. T. Özsu and P. Valduriez, *Principles of distributed database systems.* Springer Science & Business Media, 2011.

[2] M. R. Shamkant B. Navathe, “Vertical partitioning for database design: a graphical algorithm,” in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data.* ACM, 1989, pp. 440–450.

[3] J. A. Hoffer and D. G. Severance, “The use of cluster analysis in physical data base design,” in *Proceedings of the 1st International Conference on Very Large Data Bases.* ACM, 1975, pp. 69–86.

[4] S. M. T. R. R. Ladan Golshanara and H. Shah-Hosseini, “A new vertical fragmentation algorithm based on ant collective behavior in distributed database systems,” *Knowledge and Information Systems*, vol. 30, no. 2, pp. 435–455, 2012.

[5] R. M. Al-Sayyed, F. A. Al Zaghoul, D. Suleiman, M. Itriq, and I. Hababeh, “A new approach for database fragmentation and allocation to improve the distributed database management system performance,” *Journal of Software Engineering and Applications*, vol. 7, no. 11, p. 891, 2014.

[6] K. Karlaplem and Q. Li, “Partitioning schemes for object oriented databases,” in *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM’95. Fifth International Workshop on.* IEEE, 1995, pp. 42–49.

[7] C. Ezeife and K. Barker, “Distributed object based design: Vertical fragmentation of classes,” *Distributed and Parallel Databases*, vol. 6, no. 4, pp. 317–350, 1998.

[8] L. Soonmi and L. Haechull, “Attribute partitioning algorithm in doodb,” in *Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on.* IEEE, 1997, pp. 702–707.

[9] R. John and V. Saravanan, “Vertical partitioning in object oriented databases using intelligent agents,” *IJCSNS*, vol. 8, no. 10, p. 205, 2008.

[10] S.-M. Lee, Y. Ha, and H.-S. Park, “Allocation of classes in distributed object-oriented databases,” in *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD’09. 10th ACIS International Conference on.* IEEE, 2009, pp. 237–242.

[11] A. Sarhan, “A new allocation technique for methods and attributes in distributed object-oriented databases using genetic algorithms.” *Int. Arab J. Inf. Technol.*, vol. 6, no. 1, pp. 17–26, 2009.

[12] S. B. Ken Barker, “Agraphical approach to allocation class fragments in distributed object-oriented base systems,” *Distributed and Parallel Databases*, vol. 10, no. 3, pp. 207–239, 2001.

[13] H. Ma, K.-D. Schewe, and M. Kirchberg, “A heuristic approach to fragmentation incorporating query information,” in *Proceedings of the 2007 conference on Databases and Information Systems IV: Selected Papers from the Seventh International Baltic Conference DB&IS’2006.* IOS Press, 2007, pp. 103–116.

[14] F. Le Gall, “Powers of tensors and fast matrix multiplication,” in *Proceedings of the 39th international symposium on symbolic and algebraic computation.* ACM, 2014, pp. 296–303.

[15] M. J. Carey, D. J. DeWitt, and J. F. Naughton, *The 007 benchmark.* ACM, 1993, vol. 22, no. 2.

[16] F. Baião, M. Mattoso, J. Shavlik, and G. Zaverucha, “Applying theory revision to the design of distributed databases,” in *Inductive Logic Programming.* Springer, 2003, pp. 57–74.

Received on October 04 - 2014
Revised on August 31 - 2015