A Mapping Strategy for Querying Multiple Object Databases with a Global Object Schema*

Jia-Ling Koh and **Arbee L.P. Chen**

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
Email : alpchen@cs.nthu.edu.tw

Abstract

In a multidatabase system which consists of object databases, a global schema created by integrating schemas of the component databases provides a uniform interface and high level location transparency for the users to retrieve data. The mapping information between the global and component schemas is important for the global query processing. In this paper, a mapping strategy is presented. A mapping equation is defined to denote the mappings for attributes and object instances among a virtual class and its constituent classes. In addition, a mapping graph is used to describe the mapping equation. According to the mapping information, the mechanism to process global queries is introduced. One processing unit is responsible to decompose the global query into the subqueries against the component databases. Moreover, to handle the effects of schema restructuring, preprocessing and postprocessing units are provided for each local DBMS. Finally, the results returned from component databases need to be integrated. The certain result and maybe result for a global query are discussed. Also, the situation where a real-world entity is represented in different component databases as different objects is considered for the result integration.

1 Introduction

A variety of approaches to schema integration for a multidatabase system have been proposed [1], [7], [8], [10], [11], [14]. Batini et al. discussed twelve methodologies for database or view integration [1]. Czejdo, et al. used a language with graphical user interface to perform schema integration in federated database systems [5]. Schema and domain incompatibilities were considered in [5], [9] and [15]. The issues on implementing schema integration tools were reported in [12] and [20]. In [11], for automating much of the integration process, the tools to express similarities between structures in two schemas were embedded within the view integration process. The assertion-based approach was used in [21]. Other approaches defined a set of operators to build a virtual integration of multiple databases or to customize virtual classes [18], [19].

In our previous work, we had proposed a schema integration mechanism to achieve a global object schema for multiple existing object databases [16]. We first define *corresponding assertions* for the database administrator (DBA) to specify the semantic correspondences among component schemas. Based on these assertions, *integration rules* are designed, which use a set of primitive integration operators to do the integration. The integration operators are used to restructure or integrate the component schemas, and the rules specify what integration operators should be applied in what order in different situations.

The mapping information is important for the decomposition of a global query against the global schema. The mapping for the global object schema is more complicated due to schema restructuring for resolving the various conflicts among component object schemas before schema integration. However, most research for schema integration did not provide the mapping strategy and the query processing mechanism for global queries, especially for the object data model. The class constructors for deriving view class from the underlying classes were provided in [14]. It also proposed the techniques for decomposing the query into subqueries, and materializing the result. However, only query processing for class hierarchies was considered. Jeng, et al. discussed query processing strategies for homogeneous distributed object database systems, no schema conflicts were considered [13]. A message-based approach used to retrieve data in the class composition hierarchies was proposed in [6], which did not support the integration of partial results from subqueries.

In this paper, the mapping strategy between the global and component object schemas is discussed. A form of equation called *mapping equation* is defined to denote the mappings for attributes and object instances among a virtual class and its constituent classes. In addition, a *mapping graph* is used to describe the mapping equation. According to the mapping information, a mechanism to process a global query against the global schema is introduced. Some processing units and auxiliary units are provided in the query processing system. One processing unit is responsible to decompose the global query into the subqueries against the component databases. To handle the effects of schema restructuring, the preprocessing and postprocessing units are also needed for each local DBMS. Both the class hierarchies and class composition hierarchies are considered in the strategies for query processing. Finally, the results returned from the component databases need to be integrated. In our query model, the information for a real-world entity which is represented in different component databases will be integrated for a more informative query answer.

This paper is organized as follows. The next section presents some basic concepts used throughout this paper, and briefly introduces our previous work. The mapping mechanism between the global and the component ob-

*This work was partially supported by the Republic of China National Science Council under Contract No. NSC 84-2213-E-007-008.

ject schemas is provided in Section 3. Section 4 presents the strategies for processing the query against the global schema. Finally, Section 5 concludes this paper with a discussion of the future work.

2 Background

2.1 Basic Concepts

Inheritance model
In a class hierarchy, classes are linked according to the IS-A relationship among them. There are two kinds of object inheritance models in a class hierarchy. One model describes that the objects in the subclasses are also in their superclass. Thus, when querying a class, all objects in the class hierarchy rooted at the class will be accessed. The other model defines that the objects in a class are those objects which do not belong to its subclasses. When querying a class, only the objects in the class are accessed unless some special notation is specified in the query; in which case, all classes in the class hierarchy rooted at the class will be accessed [22]. In this paper, we adopt the latter inheritance model.

Concept of object isomerism
In a multidatabase system, a real-world entity may exist in different databases as different objects. We have discussed a strategy in [4] to find such objects, called *isomeric objects*. The data for those isomeric objects needs to be combined for providing more integrated information of a real-world entity presented in the multidatabase system. In this paper, we assume that the isomeric objects have been determined. Each object in the multidatabase system is assigned a global object identifier (global Oid), and the global Oids for the isomeric objects are the same. It is easier to integrate the data of the isomeric objects by using the global Oids.

2.2 Previous Work

The mapping strategy is related to the integration operators defined in our previous work [16]. Each integration operator is briefly introduced as follows. The integration operators can be categorized into *class restructuring* and *class integration operators*. All the resultant classes of these operators are virtual classes.

Class restructuring operators are used to restructure the classes in component schemas to resolve their conflicts. They may change the structure of attributes in a class and the structure of a class hierarchy. Note that the first five *class restructuring operators* are used to restructure the attributes of a class. Also, all subclasses rooted at the class inherit the restructured attributes.

1. *Refine*(*source-class.new-attribute*, *constant-value*)
The *Refine* operator adds the *new-attribute* to the *source-class*. In addition, the value of the added attribute is assigned the *constant-value*.

2. *Hide*(*source-class.hidden-attribute*)
The *Hide* operator removes the *hidden-attribute* from the *source-class*.

3. *Rename*(*source-class*(.*source-attribute*), *new-name*)
The *Rename* operator renames the *source-class* or the *source-class.source-attribute* by the *new-name*.

4. *Aggregate*(*source-class.*{*attribute-list*},
new-complex-attribute, *new-domain-class*)
The *Aggregate* operator aggregates a set of primitive attributes ({*attribute-list*}) of the *source-class* into the *new-complex-attribute*. Moreover, a new virtual class *new-domain-class* is created to be the domain class of the *new-complex-attribute*.

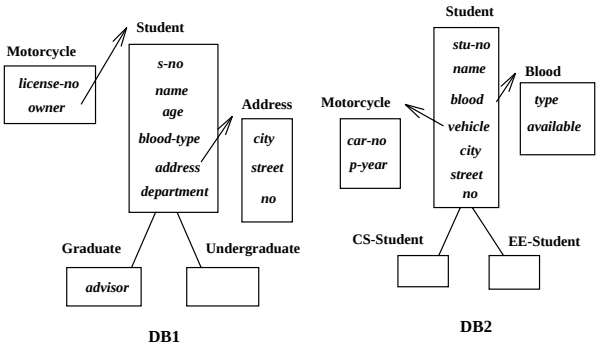

Figure 1: Component schemas for **DB1** and **DB2**

5. *Invert*(*source-class*, *inverted-class.inverted-attribute*,
new-complex-attribute)
For the *source-class* as the domain class of the *inverted-attribute* in class *inverted-class*, the *Invert* operator adds the *new-complex-attribute* to the *source-class*, with *inverted-class* as its domain class. The *new-complex-attribute* is called the *inverse* of the *inverted-attribute*.

6. *Demolish*(*source-class*)
The *division characteristic* of a class is defined as the property which distinguishes the subclasses of the class. We assume there exists a division characteristic for each class. The *Demolish* operator demolishes all the subclasses of the *source-class*, and adds all the attributes in the subclasses into the *source-class*. Moreover, an attribute which denotes the division characteristic of the class is also added to the *source-class*.

Figure 1 shows the schemas of two databases: **DB1** and **DB2**, in different schools. They are used to store the personal information for students. This figure is used for the examples in this paper. When the operation *Demolish*(**Student@DB1**) is applied, the resultant virtual class **V-Student1** in **DB1** is shown in Figure 2(a). The attribute *degree* which is the division characteristic of the class **Student@DB1** is added.

7. *Build*(*source-class*, *new-class*, *predicate-clause*)
The *Build* operator creates a subclass (*new-class*) of the *source-class*, which contains the virtual objects satisfying the *predicate-clause* in the *source-class*. The *predicate-clause* is assumed a simple predicate clause on an attribute in the *source-class*.

For example, the resultant schema for the operation *Build*(**V-Student1**, **V-CS-Student**, department=CS) is shown in Figure 2(b).

After the restructuring process, *class integration operators* are then used to integrate classes from different component databases.

1. *OUnion*(*source-class1*, *source-class2*, *new-class*)
The *OUnion* operator integrates *source-class1* and *source-class2* into a virtual class *new-class*. Only the *new-class* will appear in the global schema.

2. *Generalize*(*source-class1*,*source-class2*,
common-superclass)
The *Generalize* operator creates the *common-superclass* which is the common superclass of *source-class1* and *source-class2*. There will be two more virtual classes, cor-

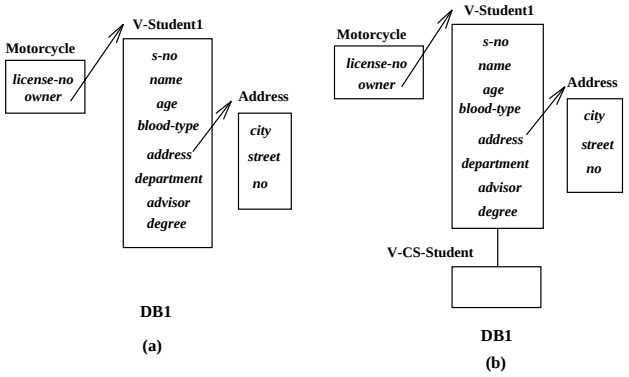

Figure 2: The resultant schemas for restructuring operations

responding to *source-class1* and *source-class2*, produced to be the subclasses of the *common-superclass* in the global schema.

3. *Specialize*(*source-class1*,*source-class2*,*common-subclass*)
The *Specialize* operator creates the *common-subclass* which is the common subclass of *source-class1* and *source-class2*. In addition, there will be two more virtual classes, corresponding to *source-class1* and *source-class2*, produced to be the superclasses of the *common-subclass* in the global schema.

4. *Inherit*(*source-subclass*, *source-superclass*)
The *Inherit* operator builds the IS-A relationship of *source-subclass* and *source-superclass*. The *source-superclass* is built as the superclass of the *source-subclass*. Two corresponding virtual classes are produced in the global schema.

Note that the database name is appended to the class names for identification if the same class names exist in different component databases.

The *integration rule* provided in [16] is the major part that guides the integrator to do the schema integration. The process guided by the integration rules is to resolve the conflicts among attributes or class-hierarchies of the component schemas first. After that, the semantics-related classes in different component schemas will be integrated into virtual classes in the global schema.

3 A Mapping Strategy

3.1 Mapping Equation

Attributes and object instances are the major components of a class. Thus, the mapping information should specify the mapping of attributes and object instances between a virtual class and the classes in the component schemas. The *attribute m_term* and *object m_term* are defined to represent the identifiers of the attributes and objects in a class, denoted as the name of the class with subscript "a" and "o," respectively. For example, the *attribute m_term* and *object m_term* for class **Student@DB1** shown in Figure 1 are denoted as $\mathbf{Student@DB1}_a$ and $\mathbf{Student@DB1}_o$, where $\mathbf{Student@DB1}_a$={*s_no, name, age, ...*} and $\mathbf{Student@DB1}_o$ contains the Oids of all objects belonging to **Student@DB1**. For each virtual class, the *m_terms* denote the virtual attributes and virtual objects in this virtual class. Therefore, the m_terms of virtual classes are called *virtual m_terms*. On the other hand, the classes exist actually in the component schemas are called *actual classes* and whose m_terms are thus called *actual m_terms*.

The *mapping equation* is used to represent the mappings between one virtual class and its constituent classes. The left-hand side of a *mapping equation* is an *m_term* of a virtual class. The right-hand side is a *mapping expression* which is a sequence of *m_terms* connected by the *mapping operators*. The *mapping operators* are used to perform different kinds of combining operations among m_terms.

In a mapping expression, in addition to the actual m_terms and virtual m_terms, another kind of m_term which is called *derived attribute* may appear. A derived attribute is the attribute which is added to a class after the class restructuring operations, such as the *new-attribute* in *Refine* operator, *new-name* in *Rename* operator, or *new-complex-attribute* in *Aggregate* operator. A derived attribute associates with a *deriving function* (enclosed in a pair of square brackets) which is used to represent the value or the source of the derived attribute. There are five deriving functions as discussed in the following.

- [$=c$]: c is a constant value. The notation denotes the constant value assigned to a *refined* attribute.
- [a]: a is an attribute name which denotes the old name for a *renamed* attribute.
- [{ a_1, a_2, ..., a_n }] : A set of attributes are listed, which denote the source attributes for a complex attribute produced from the *Aggregate* operation.
- [$\mathbf{C}.a^{-1}$]: a is an attribute in class **C**. The notation represents the inverse of a to denote the source for an attribute produced from the *Invert* operation.
- [$\mathbf{C1}.d{=}d1,\mathbf{C2}.d{=}d2,...,\mathbf{Cn}.d{=}dn$]: **C1** to **Cn** are class names and *d1* to *dn* are constant values. The notation denotes the values for the division characteristic attribute d when it is added after the *Demolish* operation.

According to the kind of the operands, the mapping operators are divided to *attribute mapping operators* and *object mapping operators*. For attribute m_term, four attribute mapping operators can be performed. $\cup$ and $-$ are used to perform the *set-union* and *set-difference* on two sets of attributes defined in a single component schema, where the attributes inherited from the same class are considered the same attributes. However, $\cup_a$ and $\cap_a$ are used to perform the *set-union* and *set-intersection* on two sets of attributes defined in different component schemas. We assume that the degree of similarity between attributes defined in different component schemas has been determined [17]. Thus, for those attributes with the same semantics, they will be considered as the same attribute. On the other hand, there are eight object mapping operators used to operate on object m_terms. σ and Π are used to perform the selection and projection on an object m_term, respectively. Each result of the projection is considered an object and assigned a virtual local Oid. $\cup$ and $-$ are the *set-union* and *set-difference* operators for two sets of objects in a single component database. Nevertheless, $\cup_o$, $\cap_o$, and $-_o$ are used to perform *set-union*, *set-intersection*, and *set-difference* on two sets of objects in different component databases. Those objects with the same global Oid are considered as the same object. Another object mapping operator, $\uplus_o$, is also performed on two sets of objects in different component component databases. The result of the $\uplus_o$ operation contains the objects in the left operand and their isomeric objects in the right operand. Among these mapping operators, $\cup_a$, $\cap_a$, $\cup_o$, $\cap_o$, $-_o$, and $\uplus_o$ concern the operations for the m-terms whose corresponding classes come from different component schemas. Therefore, they are called *external mapping operators*. The other mapping operators are named *internal mapping operators*.

Now the formal definition of the mapping equation is given:

• **attribute mapping equation**

$<v\text{-}a\text{-}m\_term> = <a\text{-}m\_exp>$

where the left-hand side is a single virtual attribute m_term. In addition, the right-hand side denoted by $<a\text{-}m\_exp>$ is an *attribute mapping expression* which is a sequence of attribute m_terms connected by attribute mapping operators. The $<v\text{-}a\text{-}m\_term>$ and $<a\text{-}m\_exp>$ can be defined by a BNF grammar as follows. The notations contained in < > are nonterminal symbols and the others are terminal symbols.

$<a\text{-}m\_exp> \rightarrow <a\text{-}m\_term> | <a\text{-}m\_exp><a\text{-}m\_op><a\text{-}m\_term>$
$<a\text{-}m\_term> \rightarrow <d\text{-}a\text{-}m\_term> | <a\text{-}a\text{-}m\_term> | <v\text{-}a\text{-}m\_term> | (<a\text{-}m\_exp>)$
$<d\text{-}a\text{-}m\_term> \rightarrow derived\text{-}attribute\ [deriving\text{-}function]$
$<a\text{-}a\text{-}m\_term> \rightarrow actual\text{-}class\text{-}name_a$
$<v\text{-}a\text{-}m\_term> \rightarrow virtual\text{-}class\text{-}name_a$
$<a\text{-}m\_op> \rightarrow \cup \mid - \mid \cup_a \mid \cap_a$

• **object mapping equation**

$<v\text{-}o\text{-}m\_term> = <o\text{-}m\_exp>$

where the left-hand side is a single virtual object m_term. The right-hand side denoted by $<o\text{-}m\_exp>$ is an *object mapping expression* which is a sequence of object m_terms connected by object mapping operators. $<v\text{-}o\text{-}m\_term>$ and $<o\text{-}m\_exp>$ are defined by a BNF as follows.

$<o\text{-}m\_exp> \rightarrow <o\text{-}m\_factor> | <o\text{-}m\_exp><o\text{-}m\_op2><o\text{-}m\_factor>$
$<o\text{-}m\_factor> \rightarrow <o\text{-}m\_term> \mid <o\text{-}m\_op1> <o\text{-}m\_term>$
$<o\text{-}m\_term> \rightarrow <a\text{-}o\text{-}m\_term> | <v\text{-}o\text{-}m\_term> | (<o\text{-}m\_exp>) | \phi$
$<a\text{-}o\text{-}m\_term> \rightarrow actual\text{-}class\text{-}name_o$
$<v\text{-}o\text{-}m\_term> \rightarrow virtual\text{-}class\text{-}name_o$
$<o\text{-}m\_op1> \rightarrow \sigma \mid \Pi$
$<o\text{-}m\_op2> \rightarrow - \mid \cup \mid \cup_o \mid \cap_o \mid -_o \mid \uplus_o$

The situation of $<o\text{-}m\_term> \rightarrow \phi$ will be explained in the next subsection.

Consider a mapping equation, each virtual m_term on the right-hand side can be replaced by the right-hand side in its corresponding mapping equation. After a sequence of replacements, we can get a mapping equation with one virtual m_term on the left-hand side and only actual m_terms, derived attributes, or ϕ on its right-hand side. This means that we can get the equation to denote the mappings between a virtual class and the actual classes in component databases.

3.2 Mapping Equations for Integration Operators

In the process of schema integration, the virtual classes will be produced after each integration operation. For an integration operator, the m_term of each produced virtual class can be described by a mapping equation. In the following, we introduce the mapping equations for the virtual classes produced after each integration operation.

⊙ **Class restructuring operators:**

• $\mathbf{C}' = Refine(\mathbf{C}.a, value)$

$$C'_a = C_a \cup a[= value] \qquad C'_o = C_o$$

• $\mathbf{C}' = Hide(\mathbf{C}.a)$

$$C'_a = C_a - a \qquad C'_o = C_o$$

• $\mathbf{C}' = Rename(\mathbf{C}, \mathbf{C}')$

$$C'_a = C_a \qquad C'_o = C_o$$

$\mathbf{C}' = Rename(\mathbf{C}.a, a')$

$$C'_a = C_a \cup a'[a] - a \qquad C'_o = C_o$$

• $\mathbf{C}' = Aggregate(\mathbf{C}.\{a_1, a_2, ..., a_n\}, a', \mathbf{A})$

Since the added attribute a' should be inherited to the subclasses of **C**, the virtual class **A** should contain the virtual objects from the projection on $\{a_1, a_2, ..., a_n\}$ for the objects in class **C** and its subclasses. Each object in the result of the projection on $\{a_1, a_2, ..., a_n\}$ is assigned a virtual local Oid. Such a virtual local Oid is useful when processing the queries against the virtual class **A**. Assuming classes **C1** to **Cn** are the subclasses of class **C**, the m_terms of $\mathbf{C}'$ and **A** are as follows.

$$C'_a = C_a \cup a'[\{a_1, a_2, ..., a_n\}] - \{a_1, a_2, ..., a_n\}$$
$$C'_o = C_o$$
$$A_a = \{a_1, a_2, ..., a_n\}$$
$$A_o = \Pi_{\{a_1, a_2, ..., a_n\}}(C_o \cup C1_o \cup ... \cup Cn_o)$$

• $\mathbf{C}' = Invert(\mathbf{C}, \mathbf{C1}.a, a')$

$$C'_a = C_a \cup a'[\mathbf{C1}.a^{-1}] \qquad C'_o = C_o$$

• $\mathbf{C}' = Demolish(\mathbf{C})$

Assuming classes **C1** to **Cn** are the subclasses of **C**, d is the division characteristic of **C**, and $d1$ to dn are the values assigned to d for the objects in **C1** to **Cn**, respectively. The m_terms of $\mathbf{C}'$ are shown below.

$$C'_a = C_a \cup C1_a \cup C2_a \cup ... \cup Cn_a \cup d[C1.d = d1, C2.d = d2, ..., Cn.d = dn]$$
$$C'_o = C_o \cup C1_o \cup C2_o \cup ... \cup Cn_o$$

• $\mathbf{C}' = Build(\mathbf{C}, \mathbf{Sub\_C}, a = value)$

A new virtual class **Sub_C** is created from **C** according to the predicate "$a = value$." Besides, another virtual class $\mathbf{C}'$ corresponding to **C** is created. The m_terms of **Sub_C** and $\mathbf{C}'$ are shown below.

$$C'_a = C_a \qquad C'_o = C_o - (\sigma_{a=value} C_o)$$
$$Sub\_C_a = C_a \qquad Sub\_C_o = \sigma_{a=value} C_o$$

⊙ **Class integration operators:**

• $OUnion(\mathbf{C1}, \mathbf{C2}, \mathbf{GC})$

Only a virtual class **GC** is created.

$$GC_a = C1_a \cup_a C2_a \qquad GC_o = C1_o \cup_o C2_o$$

• $Generalize(\mathbf{C1}, \mathbf{C2}, \mathbf{Super\_C})$

Super_C is produced to be the common superclass of classes **C1** and **C2**. Moreover, two virtual classes $\mathbf{C1}'$ and $\mathbf{C2}'$ are created corresponding to classes **C1** and **C2**, respectively. The m_terms of $\mathbf{C1}'$, $\mathbf{C2}'$ and **Super_C** are shown below.

$$Super\_C_a = C1_a \cap_a C2_a \qquad Super\_C_o = \phi$$
$$C1'_a = C1_a \qquad C1'_o = C1_o$$
$$C2'_a = C2_a \qquad C2'_o = C2_o$$

ϕ denotes an empty object set. According to the inheritance model adopted in this paper, class **Super_C** contains the objects which do not belong to class $\mathbf{C1}'$ or $\mathbf{C2}'$. Thus, the object m_term of **Super_C** is assigned ϕ.

• $Specialize(\mathbf{C1}, \mathbf{C2}, \mathbf{Sub\_C})$

Sub_C is produced to be the common subclass of classes **C1** and **C2** In addition, two virtual classes $\mathbf{C1}'$ and $\mathbf{C2}'$ are created corresponding to classes **C1** and **C2**, respectively. The m_terms of $\mathbf{C1}'$, $\mathbf{C2}'$ and **Sub_C** are shown below.

$$Sub\_C_a = C1_a \cup_a C2_a \qquad Sub\_C_o = C1_o \cap_o C2_o$$
$$C1'_a = C1_a \qquad C1'_o = C1_o -_o Sub\_C_o$$
$$C2'_a = C2_a \qquad C2'_o = C2_o -_o Sub\_C_o$$

• $Inherit(\mathbf{C1}, \mathbf{C2})$

Class **C1** is specified to be a subclass of class **C2** and the attributes of **C2** are *inherited* to **C1**. Two

virtual classes $\mathbf{C1}'$ and $\mathbf{C2}'$ are created corresponding to classes **C1** and **C2**, respectively, where $\mathbf{C2}'$ is the superclass of $\mathbf{C1}'$. The m_terms of $\mathbf{C1}'$ and $\mathbf{C2}'$ are as follows.

$$C1'_a = C1_a \cup_a C2_a \qquad C1'_o = C1_o \uplus_o C2_o$$
$$C2'_a = C2_a \qquad C2'_o = C2_o -_o C1_o$$

Note that the mapping equations for the virtual classes created from the class restructuring operators only contain the internal mapping operators. However, for the class integration operators, the mapping equations for the created virtual classes mainly contain the external mapping operators. These properties will be used in the next subsection.

3.3 Mapping Graph

Each virtual m_term can be transformed to a sequence of actual m_terms or derived attributes connected by mapping operators, which can be represented by a *mapping graph*. The mapping graph is similar to the expression *DAG* (directed acyclic graph) used in a compiler, in which a node may have more than one parent. The leave nodes are actual m_terms or derived attributes, and the internal nodes are mapping operators. The root is the result of a sequence of mapping operations.

According to the principle of the integration rules in [16] and the properties observed in Sec 3.2, the internal mapping operators will be performed before the external mapping operators in a mapping expression. Thus, the nodes representing internal mapping operators will be located below the nodes representing external mapping operators in a mapping graph.

3.4 Example

Now, we consider the previous example shown in Figure 1 to explain the mapping strategy. The two component schemas are integrated by using the *class restructuring* and *class integration* operators as follows. Those classes whose names begin with **V** are the produced virtual classes.

1. Integrate **Student@DB1**, **Student@DB2** and their subclasses:

   V-S1 = *Demolish*(**Student@DB1**)
   V-S2 = *Invert*(**V-S1**,**Motorcycle@DB1**.owner,vehicle)
   V-S3 = *Aggregate*(**V-S2**, {blood-type}, blood, **V-B1**)
   V-S4 = *Refine*(**V-S3**.school, "NTHU")
   V-S5 = *Build*(**V-S4**, **V-CS-S1**, "department=CS")
   V-S6 = *Build*(**V-S5**, **V-EE-S1**, "department=EE")
   V-S1$'$ = *Aggregate*(**Student@DB2**.{city, street, no}, address, **V-A**$'$)
   V-S2$'$ = *Rename*(**V-S1**$'$.stu-no, s-no)
   V-S3$'$ = *Refine*(**V-S2**$'$.school, "NCTU")
   OUnion(**V-S6**, **V-S3**$'$, **G-Student**)
   OUnion(**V-CS-S1**,**CS-Student@DB2**,**G-CS-Student**)
   OUnion(**V-EE-S1**,**EE-Student@DB2**,**G-EE-Student**)
   OUnion(**Address@DB1**, **V-A**$'$, **G-Address**)
   V-B1$'$ = *Rename*(**Blood@DB2**.type, blood-type)
   OUnion(**V-B1**, **V-B1**$'$, **G-Blood**)

2. Integrate **Motorcycle@DB1** and **Motorcycle @DB2**:

   V-M1$'$ = *Invert*(**Motorcycle@DB2**, **Student@DB2**.vehicle, owner)
   V-M2$'$ = *Rename*(**V-M1**$'$, car-no, license-no)
   OUnion(**Motorcycle@DB1**,**V-M2**$'$,**G-Motorcycle**)

The constructed global schema is shown in Figure 3. Besides, the m_terms of **V-S6** and **V-S3**$'$ are listed below.

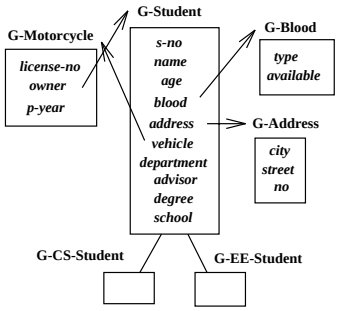

Figure 3: The constructed global schema

V-S6$_o$ = (**Student@DB1**$_o$ ∪ **Graduate@DB1**$_o$ ∪ **Undergraduate@DB1**$_o$)
− $\sigma_{department=CS}$(**Student@DB1**$_o$ ∪**Graduate@DB1** ∪ **Undergraduate@DB1**$_o$)
− $\sigma_{department=EE}$((**Student@DB1**$_o$ ∪**Graduate@DB1**$_o$ ∪ **Undergraduate@DB1**$_o$)
− $\sigma_{department=CS}$(**Student@DB1**$_o$ ∪**Graduate@DB1**$_o$ ∪ **Undergraduate@DB1**$_o$))
V-S6$_a$ = (**Student@DB1**$_a$ ∪ **Graduate@DB1**$_a$ ∪**Undergraduate@DB1**$_a$)
∪ degree [**Graduate**.degree = graduate, **Undergraduate**.degree = undergraduate]
∪ vehicle [**Motorcycle@DB1**.$owner^{-1}$]
∪ blood[{blood-type}] − blood-type ∪ school[=NTHU]

V-S3$'_o$ = **Student@DB2**$_o$
V-S3$'_a$ = **Student@DB2**$_a$ ∪ address [{city, street,no}]
− {city,street,no} ∪ s-no [stu-no] − stu-no ∪ school [=NCTU]

Furthermore, the mapping graphs for the virtual class **G-Student** constructed in the global schema are shown in Figure 4.

4 Global Query Processing

In this section, we discuss the strategies for processing queries whose format is similar to that used in XSQL [22] against the global schema. A query consists of three parts: *Select*, *From* and *Where* clauses. The format of a query is shown below:

Select <*target attributes*>
From <*range classes*>
Where <*predicate clause*>

where the *predicate clause* is assumed as in conjunctive form. Due to page limit, a complete description of the processing strategies is not included in this paper. Instead, we present the flow of query processing for the global query, introduce the processing units and auxiliary units in this system, and give an example to show the main ideas used for processing global queries.

4.1 The Flow of Global Query Processing

The flow of global query processing is shown in Fig 5. There are four main processing units in the system, which are depicted by ellipses. In the system, there is one global decomposer and one global result integrator. However, it is necessary to have a pair of local preprocessor and local postprocessor for every local DBMS. The bold lines with arrow show the flow of the query and the result.

First, the global query against the global object schema is submitted to the *global decomposer*. The global decomposer is responsible to check the range classes in the *From*

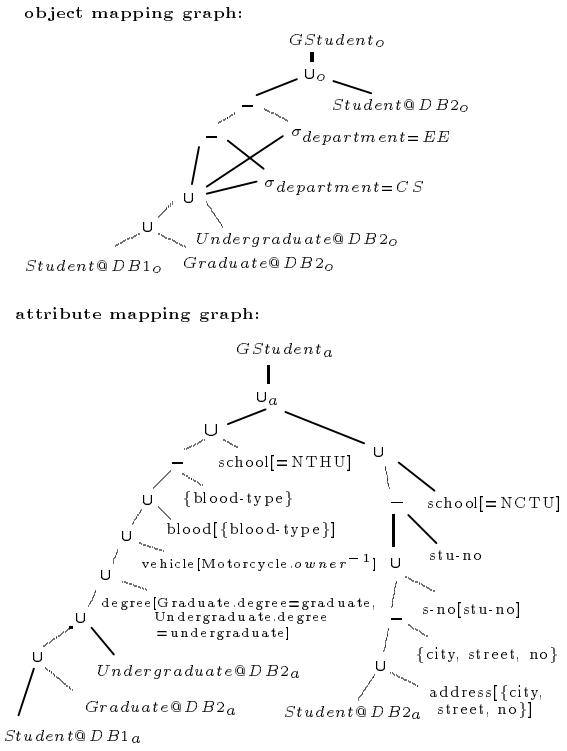

Figure 4: The mapping graph for class **G-Student**

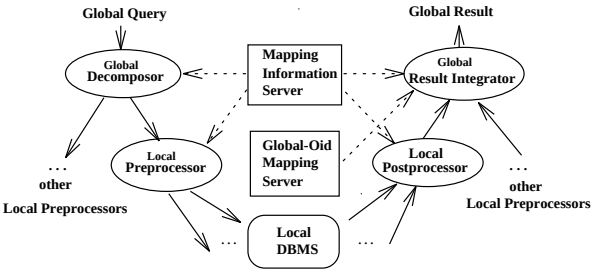

Figure 5: The flow of global query processing

clause for decomposing the global query to subqueries. The tasks for global query decomposition in the global decomposer are divided into two phases. In the first phase, if more than one range class are specified in the *From* clause for *explicit joins* [2], the global query should be decomposed into the subqueries each containing only one range class. Moreover, a modified global query for processing the explicit joins needs to be constructed to integrate the results of the subqueries. After the first phase, although only one range class is contained in each subquery, the range class may be a virtual class consisting of several classes in different component schemas. In this situation, the range class is the result of a class integration operation and the external mapping operators can be found in its mapping graph. Therefore, the task of the second phase is to check the mapping graph of the range class for the subqueries produced from the first phase, and to do possible further decomposition. After the global decomposer completes, the range class of each resultant subquery should be located in a single component database, which may or may not be an actual class. Then the subqueries are sent to the *local preprocessors* of the corresponding local DBMSs.

The virtual class contained in the subquery sent to a local preprocessor must be produced from the class restructuring operations. If it is constructed from more than one actual class, the internal object mapping operator $\cup$ must appear in the corresponding object mapping graph, and the local preprocessor has to further produce a subquery on each actual class. Moreover, it has to process the derived attributes in th *Select* and *Where* clauses in order to produce the executable queries for the local DBMS. Among the derived attributes, the *refined* attributes or *division characteristic* attributes are assigned constant values in their deriving functions. If they appear in the *Where* clause, the associated predicates can be processed according to the constant values. For processing the attributes produced by *Invert*, the classes of the inverted attributes in the same DBMS also need to to be processed. Regarding the attributes produced from *Rename* or *Aggregate*, the preprocessor replaces them with the original attributes which are specified in the deriving functions. Furthermore, the preprocessor examines whether there exist attributes in the *Select* or *Where* clause, which are not in the attribute mapping graph of the range class. If there is one (named *missing attribute*) in the *Where* clause, the result of this query will be marked with *maybe result*. Otherwise, the result will be *certain result*.

By contrast with the local preprocessor, the *local postprocessor* has to integrate the results of the subqueries which are decomposed from the preprocessor. Before returning the result to the *global result integrator*, the derived attributes appearing in the *Select* clause, whose values can not be obtained from the local DBMSs should be processed. Such derived attributes are produced due to the *Refine* or *Demolish* operations, and their values can be found in their deriving functions. In addition, null values should be appended to the results for the missing attributes which appear in the *Select* clause.

At last, the global result integrator integrates the results returned from the local postprocessors to get the final result. When integrating the local results, the local maybe results may be confirmed to be certain results if there exist isomeric objects in other local DBMSs.

The Flow Control Language defined in [5] can be used to specify the flow of global query processing.

In addition to the processing units, two auxiliary units are provided in this system and depicted by rectangles. The *mapping information server* stores the mapping graph for each virtual class in the global schema. Moreover, the mapping and the type or scale conversion functions for those attributes with the same semantics in different component schemas are provided. The *global Oid mapping server* manages the mappings between local Oids and global Oids. Function **L_to_G()** is provided to find the corresponding global Oid of the object by checking the *global Oid - local Oid mapping table* when a local Oid is specified. The dash lines with arrow show the auxiliary units used by the processing units.

4.2 Example

In the following, according to the global schema constructed in the previous example, a query is given to illustrate the procedure for global query processing. Consider query **Q**, "Retrieve the name, school, and available situation of the blood type of the graduate students living in Taipei, who are younger than 30 years old and whose motorcycles are produced after 1990." **Q** is shown in Figure 6(a).

When **Q** is processed by the global decomposer, the first

phase is skipped because no explicit join exists in **Q**. The second phase examines the mapping graph of the range class **G-student**. The external mapping operators $\cup_o$ and $\cup_a$ appearing in the mapping graph show that **G-student** is constructed from **V-S6** and **V-S3′** which come from different component databases. Thus, **Q** is decomposed into **Q1** and **Q2** as in Figure 6(b) and Figure 6(c), and sent to the preprocessors of **DB1** and **DB2**, respectively.

The preprocessor in **DB1** is responsible to process **Q1**. From the $\cup$ operations on **Student@DB1**$_o$, **Graduate@DB1**$_o$, and **Undergraduate@DB1**$_o$ as shown in the object mapping graph, we know that **V-S6** is a virtual class consisting of three actual classes. Therefore, additional decomposition for **Q1** is needed. The object mapping expression which is used to represent the object m-term of **V-S6** needs some transformations to get a sequence of subexpressions connected by $\cup$ operations as follows.

$$\begin{aligned}
\textbf{V-S6}_o &= (\textbf{Student@DB1}_o \cup \textbf{Graduate@DB1}_o \\
&\quad \cup \textbf{Undergraduate@DB1}_o) \\
&\quad - \sigma_{department=CS}(\textbf{Student@DB1}_o \\
&\quad \cup \textbf{Graduate@DB1} \cup \textbf{Undergraduate@DB1}_o) \\
&\quad - \sigma_{department=EE}((\textbf{Student@DB1}_o \\
&\quad \cup \textbf{Graduate@DB1}_o \cup \textbf{Undergraduate@DB1}_o) \\
&\quad - \sigma_{department=CS}(\textbf{Student@DB1}_o \\
&\quad \cup \textbf{Graduate@DB1}_o \cup \textbf{Undergraduate@DB1}_o)) \\
&= \sigma_{department\neq EE}((\textbf{Student@DB1}_o \\
&\quad \cup \textbf{Graduate@DB1}_o \cup \textbf{Undergraduate@DB1}_o) \\
&\quad - \sigma_{department=CS}(\textbf{Student@DB1}_o \\
&\quad \cup \textbf{Graduate@DB1}_o \cup \textbf{Undergraduate@DB1}_o)) \\
&= \sigma_{department\neq CS \text{ and } department\neq EE}(\textbf{Student@DB1}_o \\
&\quad \cup \textbf{Graduate@DB1}_o \cup \textbf{Undergraduate@DB1}_o) \\
&= (\sigma_{department\neq CS \text{ and } department\neq EE}\textbf{Student@DB1}_o) \\
&\cup (\sigma_{department\neq CS \text{ and } department\neq EE}\textbf{Graduate@DB1}_o) \\
&\cup (\sigma_{department\neq CS \text{ and } department\neq EE}\textbf{Undergraduate@DB1}_o)
\end{aligned}$$

Then the three subqueries **Q3**, **Q4**, and **Q5** against classes **Student@DB1**, **Graduate@DB1** and **Undergraduate@DB1** can be constructed as shown in Figure 6(d), 6(e), and 6(f), respectively. The derived attributes in the *Select* and *Where* clauses are then considered. For processing the predicate involving the *division characteristic* attribute *degree*, the value in the deriving function is checked. **Q3** and **Q5** can thus be eliminated and the predicate "*degree* = graduate" in **Q4** is true, which can be removed. A new subquery **Q7** is constructed against **Motorcycle@DB1** as in Figure 6(h) for processing the predictate of *vehicle.p-year*. Then the predicate for *vehicle.p-year* and the target attribute *blood.available*, which is a missing attribute, are removed from **Q4**. The attribute *school* in the *Select* clause is also removed because its value will be appended to the result in the postprocessor. However, *Oid* should be appended to the *Select* clause in **Q4** for further result integration. **Q4** is modified to **Q6** shown in Figure 6(g). Both **Q6** and **Q7** are submitted to **DB1** for execution. Now, we consider the local preprocessing for **Q2**. The **V-S3′** is constructed from only one actual calss **Student@DB2**. Thus, no further decomposition is needed. Only the range class is replaced by **Student@DB2**. The predicates for the missing attributes *degree* and *age* in **Q2** are removed because they are not defined in **Student@DB2**. Moreover, the result of **Q2** will be marked with maybe result. The *refined* attribute *school* is also removed. Then the path expression *address.city* which is produced from the *Aggregate* operator is replaced by *city*. Finally, *Oid* also needs to be added to the *Select* clause. **Q2** is modified to **Q8** shown in Figure

Q:
Select *name, school, blood.available*
From **GStudent**
Where *degree = graduate* **and** *age < 30*
and *address.city = Taipei*
and *vehicle.p-year > 1990*
(a)

Q1:
Select *name, school, blood.available*
From **V-S6**
Where *degree = graduate* **and** *age < 30*
and *address.city = Taipei*
and *vehicle.p-year > 1990*
(b)

Q2:
Select *name, school, blood.available*
From **VS3′**
Where *degree = graduate* **and** *age < 30*
and *address.city = Taipei*
and *vehicle.p-year > 1990*
(c)

Q3:
Select *name, school, blood.availabel*
From **Student@DB1**
Where *degree = graduate* **and** *age < 30*
and *address.city = Taipei*
and *vehicle.p-year > 1990*
and *department ≠ CS*
and *department ≠ EE*
(d)

Q4:
Select *name, school, blood.available*
From **Graduate@DB1**
Where *degree = graduate* **and** *age < 30*
and *address.city = Taipei*
and *vehicle.p-year > 1990*
and *department ≠ CS*
and *department ≠ EE*
(e)

Q5:
Select *name, school, blood.available*
From **Undergraduate@DB1**
Where *degree = graduate* **and** *age < 30*
and *address.city = Taipei*
and *vehicle.p-year > 1990*
and *department ≠ CS*
and *department ≠ EE*
(f)

Q6:
Select *Oid, name*
From **Graduate@DB1**
Where *age < 30*
and *address.city = Taipei*
and *department ≠ CS*
and *department ≠ EE*
(g)

Q7:
Select *owner*
From **Motorcycle@DB1**
Where *p-year > 1990*
(h)

Q8:
Select *Oid, name, blood.available*
From **Student@DB2**
Where *vehicle.p-year > 1990*
and *city = Taipei*
(i)

Figure 6: The global query and the produced subqueries in the processing

6(i) and submitted to **DB2** for execution.

When the subqueries sent to the local DBMSs are executed, the results are sent back to the corresponding postprocessors. The postprocessor of **DB1** joins the results of **Q6** and **Q7** over the joining attributes **Graduate@DB1**.*Oid* and *owner*. Then both the postprocessors of **DB1** and **DB2** append the constant values "NTHU" and "NTCU" of attribute *school* to the results returned from **DB1** and **DB2**, respectively. Moreover, null values are also appended to the results of **DB1** for the missing attribute *blood.available*. At last, these results called **R1** and **R2** are returned to the global result integrator.

Since the results in **R2** are maybe results, their isomeric objects in **R1**, which are certain results, should be checked. The function **L-to-G()** is used to identify the isomeric objects which have the same global Oid. For each object in **R2**, if its isomeric objcet is in **R1**, it can be turned into a certain result. If its isomeric object can be found in **DB1** but is not in **R1**, this maybe result is eliminated from the results. Otherwise, the maybe result remains.

5 Conclusion

The mapping information between a global schema and its associated component schemas is important for processing a global query against the global schema. In this paper, we continue our previous research on integrating multiple object schemas to consider the mapping information and query processing strategies. The *mapping equation* is defined to denote the mappings of attributes and object instances among a virtual class and its constituent classes. In addition, the mapping equation is described by the *mapping graph*. These mechanisms provide the mapping information between the global and component object schemas. The query processing flow for the global query is presented. According to the mapping information, the strategies for query decomposition and result integration are discussed. Moreover, the preprocessing and postprocessing units are provided for each local DBMS to handle the virtual classes and virtual attributes produced from schema restructuring. Finally, the concept of object isomerism is applied to derive more informative query answers.

There are many ways to decompose a global query and many different query execution plans can be produced. In this paper, we only provide one way to decompose the global query. The query optimization strategies considering various cost models will be studied in the future.

References

[1] C. Batini, M. Lenzerini, and S.B. Navathe, A comparative analysis of methodologies for database schema integration, *ACM Computing Surveys*, 18 (4) (1986) pp.323-364.

[2] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella, Object-oriented query languages: the notation and the issues, *IEEE Trans. on Knowledge and Data Engineering*, 4(3) (1992) pp. 223-237.

[3] A.L.P. Chen, J.L. Koh, T.C.T. Kuo, and C.C. Liu, Schema Integration and query processing for multiple object databases, *Journal of Integrated Computer-Aided Engineering: Special Issue on Multidatabase and Interoperable Systems, Wiley Interscience* (to appear).

[4] A.L.P. Chen, P.S.M. Tsai, and J.L. Koh, Identifying object isomerism in multiple databases (submitted for publication), 1993.

[5] B. Czejdo, M. Rusinkiewicz and D.W. Embley, An approach to schema integration and query formulation in federated database systems, *IEEE Third International Conference on Data Engineering*, (1987) pp.477-484.

[6] B. Czejdo, M. Tarylar, Integration of database systems using an object-oriented approach, *Proccedings of IEEE Interoperability in Multidatabase Systems*, (1991) pp.30-37.

[7] U. Dayal and H.Y. Hwang, View definition and generalization for database integration in a multidatabase system, *IEEE Transactions on Software Engineering*, 10 (6) (1984) pp.628-644.

[8] S.M. Deen, R.R. Amin, and M.C. Taylor, Data integration in distributed databases, *IEEE Transactions on Software Engineering*, SE-13 (7) (1987) pp.860-864.

[9] L.G. DeMichiel, Resolving database incompatibility: an approach to performing relational operations over mismatched domains, *IEEE Transactions on Knowledge and Data Engineering*, 1 (4) (1989) pp.485-493.

[10] R. Elmasri and S. Navathe, Object integration in logical database design, *IEEE First International Conference on Dta Engineering*, (1984) pp.426-433.

[11] W. Gotthard, P.C. Lockemann, and A. Neufeld, System-guilded view integration for object-oriented databases, *IEEE Transactions on knowledge and Data Engineering*, 4 (1) (1992) pp.1-22.

[12] S. Hayne and S. Ram, Multi-User view integration system (MUVIS) : An expert system for view integration, *IEEE Sixth International Conference on Data Engineering*, (1990) pp.402-409.

[13] B.P. Jeng, D. Woelk, W. Kim, and W.L. Lee, Query processing in distributed ORION, *MCC Technique Report Number: ACA-ST-035-89*, (1989) pp. 1-26.

[14] M. Kaul, K. Drosten, and E.J. Neuhold, View System : integrating heterogeneous information bases by object-oriented views, *IEEE Sixth International Conference on Data Engineering*, (1990) pp.2-10.

[15] W. Kent, Solving domain mismatch and schema mismatch problems with an object-oriented database programming language, *Seventeenth International Conference on Very Large Data Bases*, (1991) pp.147-160.

[16] J.L. Koh and A.L.P. Chen, Integration of Heterogeneous Object Schemas, *Proceedings of the 12th International Conference on Entity-Relationship Approach*, Dec, (1993) pp.289-300.

[17] J.A. Larson, S.B. Navathe, and R. Elmasri, A theory of attribute equivalence in database with application to schema integration, *IEEE Transactions on Software Engineering*, 15 (4) (1989) pp.449-463.

[18] A. Motro, Superviews : Virtual integration of multiple databases, *IEEE Transactions on Software Engineering*, 13 (7) (1987) pp.785-798.

[19] E.A. Rundensteiner, Multiview : A methodology for supporting multiple views in object-oriented databas es, *Eighteenth International Conference on Very Large Data Bases*, (1992) pp.187- 198.

[20] A. Sheth, J. Larson, A. Cornelio, and S. Navathe, A tool for integrating conceptual schemas and user views, *IEEE Fourth International Conference on Data Engineering*, (1988) pp.176-183.

[21] S. Spaccapietra, C. Parent, and Y. Dupont, Model independent assertions for integration of heterogeneous schemas, *VLDB Journal*, (1) (1992) pp.81-126.

[22] UniSQL, Inc., UniSQL/X Database System User's Manual, Release 22.0, Austin, Texas, 1993.