An Approach to Querying Multiple Object Databases*

JIA-LING KOH AND ARBEE L. P. CHEN*
Department of Computer Education
National Taiwan Normal University
Taipei, 106 Taiwan
E-mail: jlkoh@ice.ntnu.edu.tw

+Department of Computer Science
National Tsing Hua University
Hsinchu, 300 Taiwan
E-mail: alpchen@cs.nthu.edu.tw

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 to help users retrieve data. The mapping between the global and component object schemas is complicated due to schema restructuring conducted to resolve various conflicts among component schemas before conducting schema integration. This mapping information is important for 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 between a virtual class and its constituent classes. In addition, a mapping graph is used to describe the mapping equation. Based on the mapping information, a mechanism for processing global queries in parallel is introduced. One processing unit is responsible for decomposing the global query into subqueries against the component databases. To handle the effects of schema restructuring, preprocessing and postprocessing units are also provided for each local DBMS. The results returned from component databases need to be integrated. The concept of object isomerism, where a real-world entity is represented by more than one object in different component databases, is considered for integrating query results.

Keywords: object database, multidatabase system, schema mapping strategy, global query processing, object isomerism

1. INTRODUCTION

Because of the rapid development of computer networks and the need for data sharing among multiple existing databases, issues related to heterogeneous database systems have become more and more important. In a heterogeneous database system, the autonomy of the component databases should be preserved. That is, the schemas and data in each component database should be designed and manipulated independently. In order to provide a uniform interface and high level location transparency so that users can retrieve data, a global schema is usually created by integrating component schemas in the heterogeneous database system.

Received April 17, 2000; revised October 16, 2000; accepted November 30, 2000.
Communicated by Gen-Huey Chen
*This work was partially supported by the Republic of China National Science Council under Contract No. NSC89-2213-E-003-006 and NSC89-2213-E-007-044.
A variety of approaches to schema integration for multidatabase systems have been proposed [1-7]. Batini et al. discussed twelve methodologies for database or view integration [1]. Czejdo et al. used a language with a graphical user interface to perform schema integration in federated database systems [8]. Schema and domain incompatibilities were considered in [8, 9] and [10]. Issues related to implementing schema integration tools were reported in [11] and [12]. In [13], for automation of much of the integration process, tools for expressing similarities between structures in two schemas were embedded within the view integration process. A formal semantic model for specifying the correspondences between schemas was proposed in [14]. The integration strategy based on the operational mapping was provided in [15] for integrating heterogeneous data management systems. Other approaches defined a set of operators used to build a virtual integration of multiple databases or to customize virtual classes [16, 17].

In our previous work, we proposed a schema integration mechanism for deriving global object schemas from multiple existing object databases [18]. 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 perform the integration. The integration operators are used to virtually restructure or integrate the component schemas, and the rules specify which integration operators should be applied in which order in different situations.

The mapping between the global object schema and the component object schemas is complicated due to schema restructuring for resolving various conflicts among component object schemas before performing schema integration. The mapping information is important for processing of a global query against the global schema. However, most research on schema integration has not provided mapping strategies or query processing mechanisms for global queries, especially for the object data model. Class constructors for deriving view classes from underlying classes were provided in [6]. That work also proposed techniques for decomposing a query into subqueries and materializing the result. However, only query processing for class hierarchies was considered.

In this paper, a strategy for mapping between global and component object schemas is discussed. The mapping equation is used to denote mappings of attributes and object instances between a virtual class and its constituent classes. Then, a mapping graph is employed to represent a corresponding mapping equation. For each class in the global schema, an attribute mapping graph and an object mapping graph are constructed to store the mapping information with component schemas. Via traversing mapping graphs, a parallel global query processing mechanism is introduced. Some processing units and auxiliary units are provided for query processing. One processing unit is responsible for decomposing a global query into the subqueries against the component databases. To handle the effects of schema restructuring, preprocessing and post processing 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
concept of object isomerism is considered; that is, the results for a real-world entity from different component databases are integrated to obtain a more informative query answer.

This paper is organized as follows. The next section presents some basic concepts used throughout the paper and briefly introduces our previous work. The mechanism for mapping between global and component object schemas is provided in section 3. Section 4 presents strategies for processing a query against the global schema. Finally, section 5 concludes this paper and includes a discussion of future work.

2. BACKGROUND

2.1 Basic Concepts

Inheritance model In a class hierarchy, classes are linked according to the IS-A relationships among them. There are two kinds of object inheritance models in a class hierarchy. In one model, the objects in subclasses are also in their super class. Thus, when a class is queried, all the objects in the class hierarchy rooted in the class are accessed. In the other model, the objects in a class are those objects which do not belong to its subclasses. When a class is queried, only the objects in the class are accessed unless some special notation is specified in the query; in this case, all the objects in the class hierarchy rooted in the class are 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 [23] for finding such objects, called isomeric objects. The data for these isomeric objects need to be combined in query processing to provide complete information about the real-world entity. Since the process of identifying the isomeric objects is time-consuming, it is unrealistic to perform identification every time a global query is processed. In this paper, we assume that isomorphic objects have been determined. To each object in the multidatabase system is assigned a global object identifier (GOid), and the GOids for the isomeric objects are the same. It is easier to integrate the data of isomeric objects using GOids. The object identifiers defined in a component database are called Oids. The mappings among Oids and GOids are stored in the GOid mapping table. The process of identification is executed periodically to identify isomeric objects for newly added objects. The GOid mapping tables are also updated for each new identification.

We will use examples to show how the concept of object isomerism is applied to provide complete information about a real-world entity. Fig. 1 shows the schemas for class Person in DB, and class Person in DB. Since these two classes are semantically equivalent, they are integrated to obtain a global class G-Person shown in the global schema. Suppose a person P is represented by object p1 in Person@DB. If there is an object p2 in Person@DB, which also represents P, then p1 and p2 are assigned the same GOid g1. Therefore, the data in p1 and p2 can be combined during query processing to provide users with complete information about P, including its ss#, name, salary, blood_type, and birthday.

Similarly, Fig. 2 shows the schemas for class Person in DB, and class Student in DB. There is an IS-A relationship in the semantics between these two classes. Thus, a
class hierarchy is constructed between them. Moreover, Person@DB_i and Student@DB_k are renamed G-Person and G-Student, respectively, in the global schema. Suppose a student $S$ is represented by object $s_1$ in Student@DB_k, and that there is an object $p_3$ in Person@DB_i, which also represents $S$. The GOid $g_2$ is assigned to both $s_1$ and $p_3$. According to the global schema, an object in G-Student inherits attributes from G-Person. By combining $s_1$ with its isomeric object $p_3$ during query processing, users can get complete information about $S$.

![Fig. 1. Class Person in Db_i, class Person in DB_j, and global class G-Person.](image1)

![Fig. 2. Class Person in Db_i, class Student in DB_k, and the constructed global classes.](image2)

### 2.2 Previous Work

The mapping strategy is based on the integration operators defined in our previous work [18]. The resultant classes of these operations are called virtual classes. That is, no actual data are stored for the resultant classes. The data of the resultant classes are derived from the component databases and the mapping information. The attributes and objects associated with the virtual classes are called virtual attributes and virtual objects, respectively. For a virtual class $C_v$, the classes involved in the operations used to construct $C_v$ are called the constituent classes of $C_v$. Each integration operator is briefly introduced in the following. These integration operators can be categorized as class restructuring or class integration operators.

**Class restructuring operators** are used to virtually restructure the classes in a component schema in order to resolve structural conflicts among component schemas. They may virtually change the structure of the 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 in 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, to the value of the new-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-attribute in the source-class as 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) in the source-class into the new-complex-attribute. Moreover, a new virtual class new-domain-class is created to form the domain class of the new-complex-attribute.

Fig. 3 shows the schemas of two databases, DB1 and DB2, in different schools. They are used to store personal information of students. This figure is used for the examples presented in this paper. When operation Aggregate Student@DB2, city, street, no, address, V-Address is performed, the set of primitive attributes city, street, no is restructured to form a complex attribute address. Moreover, a virtual class V-Address is constructed to form the domain class of address, whose values come from the sets of values of city, street, no in Student@DB2. The resultant virtual classes V-Student2 and V-Address in DB2 are shown in Fig. 4(a).

Fig. 3. Component schemas for DB1 and DB2.

Fig. 4. The resultant schemas for the Aggregate and Invert operations.
5. **Invert** (source-class, inverted-class, inverted-attribute, new-complex-attribute)
   
   For the source-class which is 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.

   For example, the operation Invert (Motorcycle@DB2, Student@DB2. vehicle, owner) constructs a virtual attribute owner in Motorcycle@DB2. For each object o1 in Motorcycle@DB2, the value of owner comes from the Oid of object o2 in Student@DB2 if the vehicle of o2 is o1. If no such associated object exists in Student@DB2, then the value of owner of o1 is a null value. The resultant virtual class V_Motorcycle is shown in Fig. 4(b).

6. **Demolish** (source-class)
   
   The division characteristic of a class is defined as the property which distinguishes the subclasses of the class. We assume that 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 to the source-class. Moreover, a new attribute which denotes the division characteristic of the class is also added to the source-class. The objects in the resultant virtual class come from source-class and its subclasses. The added attribute is assigned a constant value for the objects in the same subclass. The values in the new attribute denote which subclass an object comes from.

   After the operation Demolish (Student@DB1) is applied, the resultant virtual class V-Student1 in DB1 is that shown in Fig. 5(a). The attribute degree, which is the division characteristic of the class Student@DB1, is added, and its value is assigned as “graduate” for the objects in Graduate@DB1 and as “undergraduate” for the objects in Undergraduate@DB1.

![Diagram](image)

Fig. 5. The resultant schemas for the Demolish and Build operations.

7. **Build** (source-class, new-class, predicate-clause)
   
   The Build operator creates a subclass (new-class) of the source-class, which contains virtual objects satisfying the predicate-clause in the source-class. The predi-
cate-clause is assumed to be a simple predicate on an attribute in the source-class. For example, the resulting schema for the operation Build (V-Student1, V-CS-Student, department = CS) is that shown in Fig. 5(b).

After the restructuring process is performed, class integration operators are used to virtually 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 new-class will appear in the global schema.

2. **Generalize** (source-class1, source-class2, common-superclass)
   The Generalize operator creates the virtual class common-superclass, which is the common superclass of source-class1 and source-class2. Two more virtual classes corresponding to source-class1 and source-class2 produced as subclasses of the common-superclass in the global schema.

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

4. **Inherit** (source-subclass, source-superclass)
   The Inherit operator builds the IS-A relationship between 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 the purpose of identification if the same class names exist in different component databases.

The integration rules provided in [18] comprise the major part that guides the schema integrator as it performs schema integration. The process guided by the integration rules resolves conflicts among the attributes or class-hierarchies of the component schemas. After that, the semantics-related classes in different component schemas are 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 include the mappings of attributes and object instances between a virtual class and the classes in the component schemas. Attribute $m_{term}$ and object $m_{term}$ represent the attributes and objects in a class, which are denoted by the class name with subscript “a” or “o,” respectively. For example, attribute $m_{term}$ and object $j_{term}$ for class Student@DB1 shown in Fig. 3 are denoted as Student@DB1 and Student@DB1, where Student@DB1$_a = \{s\_no, \text{name}, \text{age}, \ldots \}$ and
Student@DB1, contains all the 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 existing in the component databases are called actual classes, whose m_terms are called actual m_terms.

In the process of schema integration, virtual classes are produced after each integration operation. A mapping equation is used to represent the mapping between a virtual class and its constituent classes. The left-hand side of a mapping equation is an m_term of the virtual class. The right-hand side is a mapping expression, which consists of a sequence of m_terms, for the constituent classes, connected by 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 called a derived attributes may appear. A derived attribute is a virtual attribute which is added to a class after class restructuring operations are performed; examples are new-attribute in the Refine operator, new-name in the Rename operator, and new-complex-attribute in the Aggregate operator. A derived attribute a’ 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.

- a’[c]: c is a constant value. The notation denotes the constant value assigned to the refined attribute a’.
- a’[a]: a is an attribute name which denotes the old name for the renamed attribute a’.
- a’[a1, a2, . . . , an]: A set of primitive attributes are listed, which denote the source attributes for the complex attribute a’ produced by the Aggregate operation.
- a’[(D.a)]: a is an attribute in class D. The notation represents the source of an attribute a’, which is the inverse of a and is produced by the Invert operation.
- a’[C1.a’ = d1, C2.a’ = d2, . . . , Cn.a’ = dn]: C1 to Cn are class names, and d1 to dn are constant values. The notation denotes the values of the division characteristic attribute a’ when it is added due to the Demolish operation.

According to the kinds of operands they have, mapping operators are classified as attribute mapping operators or object mapping operators as shown in Table 1. Four attribute mapping operators can be performed on attribute m_terms. Those attributes having the same semantics in different component schemas [24] are assumed to be the same. Alternately, eight object mapping operators are operated on object m_terms. In particular, π is applied to retrieve the data of the virtual objects in the new-domain-class for the Aggregate operation. Therefore, each of projection result is considered a virtual object and assigned a virtual Oid. For operations on ∪o, ∩o and −o, isomeric objects should be treated as the same object. In addition, X ⊆ Y means that the objects represented by X inherit the information from the objects represented by Y.

Among these mapping operators, ∪o, ∩o, −o, and ⊆o arc for the m_terms whose corresponding classes come from different component schemes. Therefore, they are called external mapping operators. The other mapping operators are called internal mapping operators.
Table 1. The mapping operators.

<table>
<thead>
<tr>
<th>Mapping operators</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute mapping operators</td>
<td></td>
</tr>
<tr>
<td>∪</td>
<td>Perform set-union on two sets of attributes defined in the same component schema.</td>
</tr>
<tr>
<td>−</td>
<td>Perform set-difference on two sets of attributes defined in the same component schema.</td>
</tr>
<tr>
<td>∪_a</td>
<td>Perform set-union on two sets of attributes defined in different component schema.</td>
</tr>
<tr>
<td>∩</td>
<td>Perform set-intersection on two sets of attributes defined in different component schema.</td>
</tr>
<tr>
<td>object mapping operators</td>
<td></td>
</tr>
<tr>
<td>σ</td>
<td>Perform selection on a set of objects represented by an object m-term.</td>
</tr>
<tr>
<td>π</td>
<td>Perform projection on a set of objects represented by an object m-term.</td>
</tr>
<tr>
<td>∪</td>
<td>Perform set-union on two sets of objects in the same component database.</td>
</tr>
<tr>
<td>−</td>
<td>Perform set-difference on two sets of objects in the same component database.</td>
</tr>
<tr>
<td>∪_o</td>
<td>Perform set-union on two sets of objects in different component databases.</td>
</tr>
<tr>
<td>∩_o</td>
<td>Perform set-intersection on two sets of objects in different component databases.</td>
</tr>
<tr>
<td>−_o</td>
<td>Perform set-difference on two sets of objects in different component databases.</td>
</tr>
<tr>
<td>( \cup_o )</td>
<td>Perform on two sets of objects in different component databases.</td>
</tr>
<tr>
<td>( \cap_o )</td>
<td>The result of ( X \cup_o Y ) contains the objects in ( X ). Moreover, if the objects represented by ( X ) have isomeric objects represented by ( Y ), then the objects in ( X ) will contain the additional information in their isomeric objects.</td>
</tr>
</tbody>
</table>

The formal definition of the mapping equation is given as follows:

**Attribute mapping equation:**

\[
<\text{vir-att-m_term}> = <\text{a-m_expression}>,
\]

where the left-hand side is a virtual attribute m-term. In addition, the right-hand side, \(<\text{a-m_expression}>\), is an attribute mapping expression which is a sequence of attribute m_terms connected by attribute mapping operators. All the attribute mapping operators are left-associative and have the same precedence. However, parentheses can be used to force grouping. \(<\text{vir-att-m_term}>\) and \(<\text{a-m_expression}>\) are defined by a BNF grammar as shown below. The notations contained in \(< \ >\) are nonterminal symbols, and the others are terminal symbols:
Object mapping equation:

\[ <\text{vir-obj-m_term}> = <\text{o-m_expression}>; \]

where the left-hand side is a virtual object m_term. The right-hand side, \(<\text{o-m_expression}>;\), is an object mapping expression which is a sequence of object m_terms connected by object mapping operators. The unary object mapping operators, \(\sigma\) and \(\pi\), have higher precedence than the binary object mapping operators. In addition, the unary operators are right-associative and have the same precedence. The binary operators are left-associative and have the same precedence. Furthermore, parentheses can also be used to force grouping. \(<\text{vir-obj-m_term}>\) and \(<\text{o-m_expression}>\) are defined by a BNF as follows:

\[
<\text{o-m_expression}> \rightarrow <\text{o-m_factor}> | <\text{o-m_expression}> <\text{o-m_op2}> <\text{o-m_factor}> \\
<\text{o-m_factor}> \rightarrow <\text{o-m_term}> | <\text{o-m_op1}> <\text{o-m_term}> \\
<\text{o-m_term}> \rightarrow <\text{act-obj-m_term}> | <\text{vir-obj-m_term}> | ( <\text{o-m_expression}> | \phi) \\
<\text{act-obj-m_term}> \rightarrow \text{actual-class-name}_o \\
<\text{vir-obj-m_term}> \rightarrow \text{virtual-class-name}_o \\
<\text{o-m_op1}> \rightarrow \sigma | \pi \\
<\text{o-m_op2}> \rightarrow \cup | \cap | - | \cap_
\]

The situation \(<\text{o-m_term}> \rightarrow \phi\) will be explained in the next subsection.

Consider a mapping equation, in which each virtual m_term on the right-hand side can be replaced by the right-hand side of its corresponding mapping equation enclosed in a pair of parentheses. After a sequence of replacements, we can get a mapping equation with a virtual m_term on the left-hand side and only actual m_terms, derived attributes, or \(\phi\) on the right-hand side. This means that we can get an equation to denote the mapping between a virtual class and the actual classes in a component database.

Some useful properties among the internal object mapping operators \(\sigma\), \(\pi\), \(\cap\), and \(\cup\) are described in the following. These properties can be used for query transformation in query processing.

- The commutativity and associativity properties of \(\cup\)
  
  \[ \text{P1: } m_{exp_1} \cup m_{exp_2} = m_{exp_2} \cup m_{exp_1} \]
  
  \[ \text{P2: } (m_{exp_1} \cup m_{exp_2}) \cup m_{exp_3} = m_{exp_1} \cup (m_{exp_2} \cup m_{exp_3}). \]

- The distributivity property among \(\cup\), \(\sigma\), \(\pi\), and \(\cap\)
\[ P3: \sigma_p(m_{\exp i} \cup m_{\exp j}) = (\sigma_p m_{\exp i}) \cup (\sigma_p m_{\exp j}); \]
\[ P4: \pi_s(m_{\exp i} \cup m_{\exp j}) = (\pi_s m_{\exp i}) \cup (\pi_s m_{\exp j}); \]
\[ P5: (m_{\exp i} \cup m_{\exp j}) - m_{\exp k} = (m_{\exp i} - m_{\exp k}) \cup (m_{\exp j} - m_{\exp k}); \]
\[ P6: \sigma_p(m_{\exp i} - m_{\exp j}) = (\sigma_p m_{\exp i}) - (\sigma_p m_{\exp j}); \]
\[ P7: \pi_s(m_{\exp i} - m_{\exp j}) = (\pi_s m_{\exp i}) - (\pi_s m_{\exp j}); \]

- An equivalence for degrading the \(-\) operation to \(\sigma\) operation
\[ P8: m_{\exp i} - (\sigma_p m_{\exp i}) = \sigma_p m_{\exp i}; \]

- Other equivalences among the internal object mapping operators
\[ P9: m_{\exp i} - (m_{\exp i} \cup m_{\exp j}) = m_{\exp i} - m_{\exp j} - m_{\exp j}; \]
\[ P10: (m_{\exp i} - m_{\exp j}) \cup m_{\exp k} = (m_{\exp i} \cup m_{\exp k}) - m_{\exp j}; \]
\[ P11: \sigma_p(\pi_s m_{\exp i}) = \pi_s(\sigma_p m_{\exp i}); \]

\(m_{\exp i}, m_{\exp j}\) and \(m_{\exp k}\) shown above denote object mapping expressions which only consist of internal object mapping operations.

### 3.2 Mapping Equations for Integration Operators

For an integration operator, the mappings between the produced virtual classes and their constituent classes can be represented by mapping equations. The \(m\)-term of each produced virtual class appears on the left-hand side of a mapping equation, with the right-hand side being a mapping expression which consists of the \(m\)-terms of the operands involved in the operator. In the following, we will introduce mapping equations for the virtual classes produced after each integration operation.

**Class restructuring operators:**

- \( C' = \text{Refine}(C, a', c) \)
  
  Class \( C \) is refined by an attribute \( a \) with a constant value \( c \) in order to get a virtual class \( C' \). The \( m\)-terms of \( C' \) are
  \[ C' = C_a \cup \{ a'[= c] \}, \quad C'_o = C_o. \]

- \( C' = \text{Hide}(C, a) \)
  
  The attribute \( a \) in class \( C \) is hidden in order to get a virtual class \( C' \). The \( m\)-terms of \( C' \) are
  \[ C' = C_a - \{ a \}, \quad C'_o = C_o. \]

- \( C' = \text{Rename}(C, C') \)
  
  The Class \( C \) is renamed in order to get a virtual class \( C' \). The \( m\)-terms of \( C' \) are
  \[ C'_a = C_a, \quad C'_o = C_o. \]

- \( C' = \text{Rename}(C, a, a') \)
The attribute $a$ in class $C$ is renamed to $a'$ in order to get a virtual class $C'$. The m_terms of $C'$ are

$$C'_a = C_o \cup \{a'[a]\} - \{a\}, \quad C'_o = C_o.$$

- **$C' = \text{Aggregate}(C, \{a_1, a_2, ..., a_n\}, a', A)$**
  The set of primitive attributes $\{a_1, a_2, ..., a_n\}$ in class $C$ are aggregated to obtain a complex attribute $a'$, which results in a virtual class $C'$. A new virtual class $A$ is created in order to get the domain class of $a'$. Since the added attribute $a'$ should be inherited by the subclasses in the class hierarchy $H$ rooted in $C$, the virtual class $A$ should contain the virtual objects form the projection on $\{a_1, a_2, ..., a_n\}$ on the objects in $H$. To each object in the result of the project on $\{a_1, a_2, ..., a_n\}$ is assigned a virtual $Oid$. Such a virtual $Oid$ is useful when processing queries against the virtual class $A$. Assuming that classes $C_1$ to $C_n$ are the subclasses in the class hierarchy rooted in class $C$, the m_terms of $C'$ and $A$ are

$$C'_a = C_o \cup \{a'[a_1, a_2, ..., a_n]\} - \{a_1, a_2, ..., a_n\}, \quad C'_o = C_o,$$

$$A_o = \{a_1, a_2, ..., a_n\}, \quad A_o = \pi_{a_1, a_2, ..., a_n}(C_o \cup C_1 \cup ... \cup C_n).$$

- **$C' = \text{Invert}(C, D.a, a')$**
  The attribute $a'$ is added to class $C$, where $a'$ is the inverse of attribute $a$, in order to get a virtual class $C'$. The m_terms of $C'$ are

$$C'_a = C_o \cup \{a'[D.a]\}, \quad C'_o = C_o.$$

- **$C' = \text{Demolish}(C)$**
  The subclasses of class $C$ are demolished in order to get a virtual class $C'$. Assume that classes $C_1$ to $C_n$ are the subclasses of $C$, that $d'$ is the division characteristic of $C$, and that $d1$ to $dn$ are the values assigned to $d'$ for the objects in $C_1$ to $C_n$, respectively. The m_terms of $C'$ are

$$C'_a = C_o \cup C_1 \cup C_2 \cup ... \cup C_n \cup \{a'[C_1.a' = d1, C_2.a' = d2, ..., C_n.a' = dn]\},$$

$$C'_o = C_o \cup C_1 \cup C_2 \cup ... \cup C_n.$$

- **$C' = \text{Build}(C, \text{Sub}_C, p)$**
  A new virtual class $\text{Sub}_C$ is created from $C$, in which all the virtual objects satisfy the predicate $p$. Furthermore, another virtual class $C'$ corresponding to $C$ is created. The m_terms of $\text{Sub}_C$ and $C'$ are

$$C'_a = C_o, \quad C'_o = C_o - (\sigma_p C_o),$$

$$\text{Sub}_C = C_o, \quad \text{Sub}_C = \sigma_p C_o.$$

By observing the object mapping equations for class restructuring operators, the following properties can be found.
P12: $\cup$ is the only object mapping operator used to combine different object m_terms as shown in the object mapping equations for Aggregate and Demolish operators.

P13: The object mapping operator $-$ is only used in the mapping equation for class $C'$ produced in the Build operator.

These properties will be used in the next section in strategies for global query processing.

Class integration operators:

- **OUnion**($C_1$, $C_2$, GC)
  
  Class $C_1$ and $C_2$ are ounioned. Only a virtual class GC is created. The m_terms of GC are
  
  $$GC_a = C_1_a \cup_a C_2_a \quad \text{and} \quad GC_o = C_1_o \cup_o C_2_o.$$  

- **Generalize**($C_1$, $C_2$, Super_C)
  
  Class $C_1$ and $C_2$ are generalized in order to produce a virtual class Super_C as their common subclass. In addition, two virtual classes $C_1'$ and $C_2'$ are created, corresponding to classes $C_1$ and $C_2$, respectively. The m_terms of $C_1'$, $C_2'$ and Super_C are
  
  $$Super_C_a = C_1_a \cap_a C_2_a, \quad Super_C_o = \phi,$$
  
  $$C_1' a = C_1 a, \quad C_1' o = C_1 o,$$
  
  $$C_2' a = C_2 a, \quad C_2' o = C_2 o.$$  

  $\phi$ denotes an empty object set. According to the inheritance model adopted in this paper, class Super_C contains objects which do not belong to class $C_1'$ or $C_2'$. Thus, to the object m_term of Super_C is assigned $\phi$.

- **Specialize**($C_1$, $C_2$, Super_C)
  
  Class $C_1$ and $C_2$ are specialized in order to produce a virtual class Sub_C as their common subclass. In addition, two virtual classes $C_1'$ and $C_2'$ are created, corresponding to classes $C_1$ and $C_2$, respectively. The m_terms of $C_1'$, $C_2'$ and Sub_C are
  
  $$Sub_C_a = C_1_a \cap_a C_2_a, \quad Sub_C_o = C_1_o \cap_o C_2_o,$$
  
  $$C_1' a = C_1 a, \quad C_1' o = C_1 o \setminus Sub_C o,$$
  
  $$C_2' a = C_2 a, \quad C_2' o = C_2 o \setminus Sub_C o.$$  

- **Inherit**($C_1$, $C_2$)
  
  Class $C_1$ is specified as a subclass of class $C_2$, and the attributes of $C_2$ are inherited by $C_1$. Two $C_1'$ and $C_2'$ are created, corresponding to classes $C_1$ and $C_2$, respectively, where $C_2'$ is the superclass of $C_1'$. The m_terms of $C_1'$, $C_2'$ are
If the objects in $C_1$ have isomeric objects in $C_2$, then the values of the newly inherited attributes from $C_2$ are taken from $C_2$ and integrated in $C_1'$. The inherited attributes are filled with null values for the other objects in $C_1'$.

Note that the mapping equations for the virtual classes created by the class restructuring operators only contain the internal mapping operators. Similarly, for the class integration operators, the mapping equations for the created virtual classes only 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 into a mapping expression $exp_m$ which is a sequence of actual m_terms or derived attributes connected by mapping operators. The $exp_m$ can be represented by a mapping graph. The mapping graph is similar to the expression DAG (directed acyclic graph) used in a compiler [25]. It is a generalized binary tree structure in which a node may have more than one parent. Any node that has no child nodes is a terminal node; the terminal nodes in a mapping graph are actual m_terms or derived attributes. The nodes which are not terminal nodes are called non-terminal node $N$ can be labeled with a virtual m_term $m$; this indicates that the expression represented by $N$ is the mapping expression for $m$. Among the nonterminal nodes, any node which has no parent node is a root node.

For each global class, an object mapping graph and an attribute mapping graph are constructed. The root nodes in the mapping graphs are labeled using the m_terms of the global class. The mapping graphs of a virtual class $C_v$ which is produced in the integration process are subgraphs rooted in the nonterminal nodes labeled using the m_terms of $C_v$.

According to the integration rules proposed in [18] and the properties stated in section 3.2, the internal mapping operators are performed before the external mapping operators in a mapping expression. Thus, nodes representing internal mapping operators will be located below the nodes representing external mapping operators in a mapping graph.

3.4 Example

We will consider here the previous example shown in Fig. 3 to explain the mapping strategy. The two component schemas are integrated using the class restructuring and class integration operators as follows. Those classes whose names begin with $V$- are the virtual classes produced in the integration process, and those whose names begin with $G$- are the final global virtual classes.

1. Integrate $\text{Student}@DB1$, $\text{Student}@DB2$ and their subclasses:
   - $V$-S1 = Demolish ($\text{Student}@DB1$);
   - $V$-S2 = Invert ($V$-S1, $\text{Motorcycle}@DB1.\text{owner}$, vehicle);
   - $V$-S3 = Aggregate ($V$-S2.\{b-type\}, blood-type, $V$-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-A1’);
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-A1’.G-Address);
V-B1’ = Rename (Blood@DB2.type, b-type);
OUnion (V-B1, V-B1’, G-Blood-type).

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 Fig. 6. Furthermore, the m_terms of V-S6 and V-S3’ are those listed below:

Fig. 6. The constructed global schema.

V-S6 = (Student@DB1, Graduate@DB1, Undergraduate@DB1)
− σdepartment=CS (Student@DB1, Graduate@DB1, Undergraduate@DB1)
− σdepartment=EE (Student@DB1, Graduate@DB1, Undergraduate@DB1)
− σdepartment=CS (Student@DB1, Graduate@DB1, Undergraduate@DB1).
V-S6 = (Student@DB1, Graduate@DB1, Undergraduate@DB1)
∪ {degree [Graduate.degree = graduate, Undergraduate.degree = undergraduate]}
∪ {vehicle [(Motorcycle@DB1.owner)]} ∪ {blood-type [{b-type}]} − {b-type}
\[ \cup \{school [=NTHU]\} \].

\[ V-S3' = \text{Student@DB}_2 \cup \{\text{address }\{\text{city, street, no}\}\} - \{\text{city, street, no}\} - \{\text{stu-no}\} \cup \{\text{school }[= \text{NCTU}]\}. \]

The mapping graphs for the virtual class \textbf{G-Student} constructed in the global schema are shown in Fig. 7. Note that the parentheses around the mapping expressions are not shown in the mapping graphs because the precedence of each operator is represented by the tree structure.

Fig. 7. The mapping graphs for class \textbf{G-Student}.

4. GLOBAL QUERY PROCESSING

In this section, we will discuss strategies for processing global queries against the global schema, which include decomposing a global query into executable subqueries for local DBMSs and integrating the results of these subqueries. The format of global queries is similar to that used in XSQL [22]. A query consists of three parts: Select, From and Where clauses as shown below:

- Select  \langle target attributes \rangle,
- From  \langle range classes \rangle,
- Where  \langle predicate clause \rangle,

where the predicate clause is assumed to be in conjunctive form.

We will discuss global query processing based on the assumption that the result of a query is represented as a table with attribute \textit{Oid} and the complex attribute storing the \textit{Oid} values of the associated objects. Some query processing may need to join the complex attribute with the \textit{Oid} attribute of another class, and some may need to join the \textit{Oid} attributes of two classes. Data inconsistencies among component databases will not be discussed in this paper.
4.1 The Flow of Global Query Processing

The flow of global query processing is shown in Fig. 8. There are four main processing units, which are depicted by ellipses. There is one query decomposer, one result integrator, and a pair of consisting of a local preprocessor and local postprocessor for each local DBMS. The bold lines with arrows show the flow of the query and the result.

First, the global query against the global object schema is sent to the query decomposer. The query decomposer is responsible for checking if the global query needs to be decomposed into several subqueries, each against a single component database. Then, the subqueries are sent to the local preprocessors of the corresponding local DBMSs. The range class in the query sent to the local preprocessor may be a virtual class constructed from more than one actual class. In this situation, the local preprocessor needs to further decompose the query into subqueries whose From clauses contain a single actual class. Moreover, it has to process the derived attributes in the Select and Where clauses in order to produce executable queries for the local DBMS. The local postprocessor has to integrate the results of the subqueries which are produced by the local preprocessor. Before the result is returned to the result integrator, the derived attributes appearing in the Select clause whose values cannot be obtained from the local DBMS should be processed. Finally, the result integrator integrates the results from the local postprocessors to get the final result.

In addition to the processing units, two auxiliary units are provided, which are depicted by rectangles. The mapping information server stores the mapping graph for each virtual class in the global schema. The mapping and the type or scale conversion functions for attributes having the same semantics in different component schemas are also provided. Moreover, function \texttt{V_to_O()} is used to assign an associated virtual Oid to the result of a projection. The Global-Oid mapping server manages the GOid mapping table. Function \texttt{L_to_G()} is used to find the corresponding GOid of an object when an Oid is specified. The dashed lines with arrows show the auxiliary units used by the processing units.
4.2 Strategies for Global Query Processing

The details of the four main processing units will be introduced in this subsection. However, due to page limitations, for detailed algorithms, please refer to [26].

4.2.1 Query decomposer

The tasks involved in query decomposition are divided into two phases: $QD_1$ and $QD_2$. In phase $QD_1$, if more than one range class is specified in the $From$ clause for explicit joins [27], the global query should be decomposed into subqueries, each containing a single range class. Moreover, a modified global query for processing explicit joins on the results of the subqueries needs to be constructed. After phase $QD_1$ is completed, although only one range class is contained in each subquery, the range class may be a virtual class consisting of actual or virtual classes in different component schemas. Therefore, the task of phase $QD_2$ is to check the object mapping graph of the range class for each subquery produced during phase $QD_1$ and to perform further decomposition if possible. After the query decomposer finishes its work, the range class of each resultant subquery should be an actual or virtual class located in a single component database. Then, the subqueries are sent to the local preprocessors of the corresponding local DBMSs.

Phase $QD_1$:
If only one range class is specified in the $From$ clause of the global query, then phase $QD_1$ is skipped, and the global query is passed on to be processed in phase $QD_2$. Otherwise, suppose there are $n$ range classes. Let each range class be denoted by $R_i$, $1 \leq i \leq n$. For each $R_i$, $P_i$ denotes its associated predicates in the $Where$ clause, which connected by $\text{and}$, and $S_i$ denotes its associated target attributes. In addition, the predicates for processing explicit joins are called $explicit$ $join$ $predicates$. The attributes belonging to $R_i$ and appearing in explicit join predicates are denoted by $J_i$. Then, the global query is decomposed into $n$ subqueries. The $i$th subquery is shown as follows, where $1 \leq i \leq n$:

$$
Select \quad S_i, \quad J_i, \\
From \quad R_i, \\
Where \quad P_i.
$$

Moreover, the global query is modified to obtain the following, where $t_i$ denotes the table for storing the result of the $i$th subquery:

$$
Select \quad S_1, S_2, \ldots, S_n, \\
From \quad t_1, t_2, \ldots, t_n, \\
Where \quad explicit \ join \ predicates.
$$

The modified global query is sent to the result integrator and will be used to integrate the results of the subqueries.

Phase $QD_2$: 
Suppose the subquery produced during phase **QD_1** is denoted as:

```
Select S,  
From R,  
Where P.
```

Then the object mapping graph of \( R \) is traversed from its root. When an external object mapping operator is visited, we know that \( R \) is created by integrating two actual or virtual classes from different component databases. Let \( LC_o \) and \( RC_o \) denote the left and right operands of the external mapping operator, respectively. Two subqueries are then constructed as follows:

```
Select S,  
From LC,  
Where P,
```

```
Select S,  
From RC,  
Where P.
```

The results of these two subqueries are then integrated in the result integrator to obtain the result of the original query. The subqueries are further decomposed by calling algorithm **QD_2** recursively until no external mapping operator can be visited. That is, each From clause in the newly constructed subqueries contains an actual class or a virtual class defined in a single component database. These subqueries are sent to the local preprocessor of the associated component database.

### 4.2.2 Local preprocessor

The range class of the subquery sent to a local preprocessor is an actual class or a virtual class produced by the class restructuring operations. If the range class is a virtual class constructed from more than one actual class, the local preprocessor has to further perform a subquery on each actual class. Moreover, it has to process the derived attributes in the Select and Where clauses in order to produce executable queries for the local DBMS. These queries are then sent to the underlying local DBMS for execution. The tasks of the local preprocessor are performed in the three phases described below.

**Phase LPR_1:**

Let the query sent to phase **LPR_1** be denoted as:

```
Select S,  
From C,  
Where P.
```

The object mapping subgraph rooted in the m_tern if \( C \) is checked. The object mapping expression represented by the mapping subgraph is a sequence of internal object mapping operations composed of \( \cup, \sigma, \pi \), and/or \( - \) operations.

According to property **P13**, the only situation in which the internal object mapping operator \( - \) appears in our mapping model is that shown on the left hand side of property **P8** (produced after a **Build** operation). Therefore, all the \( - \) operations in the mapping expression can be degraded to \( \sigma \) operations according to property **P8**.
From property P12, if there is a sequence of \( \cup \) operations in the resultant expression, then the range class \( C \) must have been constructed from more than one actual class. According to P3 and P4, the mapping expression is transformed into a sequence of subexpressions linked by the \( \cup \) operator. Each subexpression contains a single actual m_term corresponding to a terminal node. The resultant subexpression is, thus, only composed of \( \sigma \) and/or \( \pi \) operations. A subquery is then constructed for each subexpression.

Suppose the \( i \)th subexpression is denoted as \( \pi_{s_1}, \pi_{s_2}, \ldots, \pi_{s_m} \sigma_{p_1}, \sigma_{p_2}, \ldots, \sigma_{p_k} \) \( C_i \). \( C_i \) is specified as the range class, which is the associated actual class of \( C_o \). \( p_1, p_2, \ldots, p_k \) are added to the Where clause as predicates and are combined with boolean and operations. As mentioned previously, \( \pi \) is used to retrieve the virtual objects produced after the Aggregate operation. Therefore, \( s_1, s_2, \ldots, s_m \) are added to the target attributes for retrieval of values from the underlying DBMS. The constructed subquery for the \( i \)th subexpression is

\[
\text{Select} \quad S, s_1, s_2, \ldots, s_m, \\
\text{From} \quad C_i, \\
\text{Where} \quad P \text{ and } p_1 \text{ and } p_2 \text{ and } \ldots \text{ and } p_k.
\]

Phase LPR_2:
For each subquery produced in phase LPR_1, the Select and Where clauses need to be modified if different derived attributes are specified in the subquery. The derived attributes in the query are processed in phase LPR_2. Let the query submitted to phase LPR_2 be called \( q \). The derived attributes can be recognized by referring to the attributed mapping graph.

1. attribute \( a' \) is a Refined or division characteristic attribute
   The value of attribute \( a' \) is the constant value in its deriving function. If \( a' \) appears in the Where clause of \( q \), then let \( p' \) denote an associated predicate. Each \( p' \) is evaluated by checking its constant value:
   (1) If \( p' \) is true, then \( p' \) is removed from the predicate clause.
   (2) If \( p' \) is false, then the whole predicate clause is false no matter what the other predicates are. \( q \) is eliminated without the need for it to the result integrator. If \( a' \) is specified in the Select clause of \( q \), then it is removed because its value is not retrievable from the underlying DBMS. Such value will be appended to the result in the postprocessor.

2. complex attribute \( a' \) produced by an Invert operation
   For query \( q \), let \( P \) denote the predicates, and let \( T \) denote the target attributes, which contain a nested attribute rooted at \( a' \). Since \( a' \) is a virtual attribute, \( P \) and \( T \) cannot be processed directly by the local DBMS, so they are removed from \( q \). Let attribute \( D.a \) be the inverse of \( a' \) (that is, the domain class of \( a' \) is \( D \)). This implies that the nested attributes rooted at \( a' \) come from the attributes of \( D \). Therefore, an additional new subquery should be constructed on \( D \) for retrieving \( T \) and evaluation \( P \). Let \( P' \) and \( T' \) denote \( P \) and \( T \), respectively, after the path expression "\( a'.X \)" is replaced with \( X \). The new subquery is
Select $T^\prime, a$
From $D$
Where $P^\prime$.

Not that $D.a$ is retrieved such that the results of the new subquery and $q$ can be integrated later. (In this case, the $Oid$ of the range class of $q$ must also be retrieved.)

3. attribute $a^\prime$ produced by an **Rename** operation

Let the deriving function of $a^\prime$ be $[a]$. The attribute $a^\prime$ should be replaced with the old name $a$ when $a^\prime$ exists in the **Select** and/or the **Where** clause of $q$.

4. complex attribute $a^\prime$ produced by an **Aggregate** operation

The path expression $a^\prime.a_i$ is replaced with the primitive attribute $a_i$, where $a_i$ is one of primitive attributes specified in the deriving function of $a^\prime$. The replacement action is performed when $a^\prime$ exists in the **Select** and/or the **Where** clause of $q$.

**Phase LPR_3:**
The attributes of a global class are the union of the attributes belonging to its constituent classes. An attribute appearing in the global class may not be defined in a constituent class, in which case it is called a **missing attribute** of the constituent class [28]. Let the query passed to phase **LPR_3** be denoted by $q$. Predicates involving missing attributes of the range class are called **unsolved predicates**. The missing attributes and the unsolved predicates should be removed from the **Select** clause and the **Where** clause, respectively, since they cannot be processed by the underlying DBMS. If the missing attributes appear in the **Where** clause, the result of $q$ will be marked with **maybe result** [29] because the unsolved predicates are not evaluated. Otherwise, the result will be **certain result**. The data for the isomeric objects need to be combined in the result integrator to provide complete information about the isomeric objects. Therefore, the $Oid$ is also specified in the **Select** clause for integration of isomeric objects later. The maybe result of $q$ may be changed into a **certain result** due to the integration of isomeric objects.

In [30], we studied global query processing that involves missing attributes.

**4.2.3 Local postprocessor**

Suppose the results of queries sent from a local preprocessor to the local DBMS have been returned to the local postprocessor. There are two phases of tasks to be performed in the local postprocessor, and they are called **LPO_1** and **LPO_2**. **Phase LPO_1** is responsible to processing the results if the derived attributes appear in the associated queries. The results of the subqueries, which are constructed in phase **LPR_1**, are integrated in phase **LPO_2**.

**Phase LPO_1:**
Given a query result $r$, suppose its associated query is $q$. Further processing on $r$ is needed in the following situations.

1. $q$ has a subquery $q^\prime$ for processing the derived attribute $a^\prime$ generated by an **Invert** operation. Let the result of $q^\prime$ be denoted by $r^\prime$. If $a^\prime$ appears in the **Where** clause of $q$, then $r$ and $r^\prime$ will be joined over the join attributes $Oid$ and $a$, respectively. If $a^\prime$ is
only specified in the Select clause of \( q \), then a left outer join from \( r \) to \( r' \) will be performed.

2. \( q \) has target attributes which are \textit{Refined} or \textit{division characteristic} attributes. The values of these attributes cannot be retrieved from the local DBMSs. Therefore, the values obtained from the associated deriving function are appended to the result. It is necessary to convert the attribute values so that they are of the data type shown in the global schema when necessary.

3. \( q \) has target attributes \( s_1, s_2, \ldots, s_m \) as a result of the Aggregate operator that appears in phase LPR_1. For each result tuple which represents the value of a virtual object, the attribute values of \( s_1, s_2, \ldots, s_m \) will be used by function \( \text{V}_{\text{to}}\text{O}() \) to assign a virtual \( \text{Oid} \). These \( \text{Oids} \) may be used to obtain further information from their isomeric objects.

4. If \( q \) has unsolved predicates, then \( r \) is marked with a maybe result. Otherwise, \( r \) is a certain result.

**Phase LPO_2:**

The union operator is used to integrate the results of subqueries constructed in phase LPR_1. Then, the results are returned to the result integrator.

**4.2.4 Result integrator**

For integration of the results of subqueries produced by the query decomposer, there are two phases of processing in the result integrator. The first phase, phase RI_1, is responsible for integrating the results of subqueries constructed in phase QD_2. In addition, the second phase, called RI_2, executes the modified global query constructed in phase QD_1.

**Phase RI_1:**

Each object \( o \) in the maybe result can be turned into a certain result if its isomeric objects satisfy the associated unsolved predicates. This situation can be stated by the following two conditions.

1. The missing attributes \( A_m \) involved in the query used to obtain \( o \) exist in another component schema \( \xi \).
2. The isomeric objects of \( o \) are also in the result returned from the component database with schema \( \xi \).

The maybe result is eliminated when the first condition is satisfied and when its isomeric objects exist but are not in the result of the subqueries. For more details about global query processing concerning missing attributes, please refer to [30].

The order for integrating the results of subqueries is the opposite of the order for constructing subqueries in phase QD_2. Let a query \( q_{\text{deq}} \) have two subqueries \( q_1 \) and \( q_2 \) constructed in phase QD_2. This implies that the range class of \( q_1 \) and \( q_2 \). \( O_m \) is used to decide \( o \) the best way to integrate the results of these two subqueries in order to obtain the result of \( q_{\text{deq}} \). Let \( cr_1 \) and \( cr_2 \) denote the certain results of \( q_1 \) and \( q_2 \), respectively. In addition, \( mr_1 \) and \( mr_2 \) are used to denote the maybe results. In the following, different ways of integrating the results are presented. Note that the join attribute is the associated \( \text{GOid} \) of the \( \text{Oid} \) for the result.
• case1: \( O_m = \bigcup_o \)
  <certain result> = \( cr_1 \bowtie cr_2 \),
  <maybe result> = \( mr_1 \bowtie mr_2 \),
  where \( \bowtie \) is an outer join.

• case2: \( O_m = \bigcap_o \)
  <certain result> = \( cr_1 \bowtie cr_2 \),
  <maybe result> = \( mr_1 \bowtie mr_2 \),
  where \( \bowtie \) is a join.

• case3: \( O_m = \overline{\bigcup_o} \)
  <certain result> = \( cr_1 - (cr_2 \bowtie cr_2) \),
  <maybe result> = \( mr_1 - (mr_2 \bowtie mr_2) \),
  where \( \bowtie \) is an semijoin.

• case4: \( O_m = \overline{\bigcup_o} \)
  <certain result> = \( cr_1 \bowtie cr_2 \),
  <maybe result> = \( cr_1 \bowtie cr_2 \),
  where \( \bowtie \) is a left outer join.

Phase RI_2:
If a modified global query was sent from phase QD_1, this modified global query is executed to get the final result.

4.3 Example
In the following, based on 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 the availability of the blood type of the graduate students living in Taipei, who are younger than 30 years old and whose motorcycles were produced after 1990.” \( Q \) is shown in Fig. 9(a).

When \( Q \) is processed by the global decompose, phase QD_1 is skipped because no explicit join exists in \( Q \). Phase QD_2 examines the mapping graph of the range class G-student. The external mapping operator \( \bigcup_o \) appearing in the mapping graph shows that G-student is constructed from V-S6 and V-S3’, which come from different component databases. Thus, \( Q \) is decomposed into \( Q_1 \) and \( Q_2 \) as shown in Fig. 9(b) and Fig. 9(c), and is sent to the preprocessors of DB1 and DB2, respectively.

\[
Q: \begin{align*}
&\text{Select} & \text{name, school, blood-type, availability} \\
&\text{From} & \text{G-Student} \\
&\text{Where} & \text{degree = graduate} \quad \text{and} \quad \text{age < 30} \\
& & \text{and} \quad \text{address.city = Taipei} \\
& & \text{and} \quad \text{vehicle.p-year > 1990}
\end{align*}
\] (a)
Q1: Select name, school, blood-type.availability
    From V-S6
    Where degree = graduate and age < 30
    and address.city = Taipei
    and vehicle.p-year > 1990

(b)

Q2: Select name, school, blood-type.availability
    From V-S3′
    Where degree = graduate and age < 30
    and address.city = Taipei
    and vehicle.p-year > 1990

(c)

Q1-1: Select name, school, blood-type.availability
      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)

Q1-2: Select name, school, blood-type.availability
      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)

Q1-3: Select name, school, blood-type.availability
      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)

Fig. 9. The global query and the subqueries produced during processing.
Q1-2-1:  
Select Oid, name
From Graduate@DB1
Where age < 30
and address.city = Taipei
and vehicle.p-year > 1990
and department ≠ CS
and department ≠ EE

(g)

Q1-2-2:  
Select owner
From Motorcycle@DB1
Where p-year > 1990

(h)

Q2-1:  
Select Oid, name, blood-type.availability
From Student@DB2
Where vehicle.p-year > 1990
and city = Taipei

(i)

Fig. 9. (Cont’d) The global query and the subqueries produced during processing.

The preprocessor in DB1 is responsible for processing Q1. From the ∪ operations performed on Student@DB1, Graduate@DB1, and Undergraduate@DB1, as shown in the object mapping graph, we know that V-S6 is a virtual class consisting of three actual classes. Therefore, additional decomposition of Q1 is needed in phase LPR_1. The object mapping expression which is used to represent the object m-term of V-S6 requires some transformation to get a sequence of subexpressions connected by ∪ operations as follows.

V-S6_o = (Student@DB1_o ∪ Graduate@DB1_o ∪ Undergraduate@DB1_o)
− σ_{department=CS} (Student@DB1_o ∪ Graduate@DB1_o ∪ Undergraduate@DB1_o);
− σ_{department=EE} ((Student@DB1_o ∪ Graduate@DB1_o ∪ Undergraduate@DB1_o);
− σ_{department=CS} (Student@DB1_o ∪ Graduate@DB1_o ∪ Undergraduate@DB1_o));
= σ_{department ≠ CS and department ≠ EE} (Student@DB1_o ∪ Graduate@DB1_o ∪ Undergraduate@DB1_o);
∪ σ_{department=CS} (Student@DB1_o ∪ Graduate@DB1_o ∪ Undergraduate@DB1_o);
∪ σ_{department=CS and department=EE} Graduate@DB1_o
∪ σ_{department=CS and department=EE} Undergraduate@DB1_o).

Then, the three subqueries Q1-1, Q1-2, and Q1-3 against classes Student@DB1, Graduate@DB1 and Undergraduate@DB1 can be constructed as shown in Figs. 9(d), 9(e), and 9(f), respectively. In phase LPR_2, the derived attributes in the Select and Where clauses are then considered. In order to process a predicate involving the division characteristic attribute degree, the value in the deriving function is checked. Q1-1
and Q1-3 can, thus, be eliminated, and the predicate “degree = graduate” in Q1-2 is true and can be removed. The refined attribute school in the Select clause is also removed because its value will be appended to the result in the postprocessor. A new subquery Q1-2-2 is constructed against Motorcycle@DB1 as shown in Fig. 9(h) so that the predicate on the derived attribute vehicle.p-year can be processed. Then, the predicate for vehicle.p-year is removed from Q1-2. Finally, the target attribute blood-type.availability, which is a missing attribute, is removed from Q1-2 in phase LPR.3. In addition, Oid should be appended to the Select clause in Q1-2 for further result integration. Q1-2 is modified to obtain Q1-2-1 as shown in Fig. 9(g). Both Q1-2-1 and Q1-2-2 are submitted to DB1 for execution.

Now, we will consider local preprocessing for Q2. V-S3’ is constructed from only one actual class Student@DB2. Thus, no further decomposition is needed in phase LPR_. Only the range class is replaced with Student@DB2. In phase LPR_2, the refined attribute school is removed from Q2. Then, the path expression address.city which is produced from the Aggregate operation is replaced with city. The predicates for the missing attributes degree and age in Q2 are removed in phase LPR_3 because they are not defined in Student@DB2. Moreover, the result of Q2 is marked with a maybe result. Finally, Oid also needs to be added to the Select clause. Q2 is modified to obtain Q2-1 as shown in Fig. 9(i) and is submitted to DB2 for execution.

After 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 Q1-2-1 and Q1-2-2 over the joining attributes Oid and owner in phase LPO_. Then, the constant value “NTHU” for the attribute school and null values for the missing attribute blood-type.availability are appended to the results of Q1-2. Since the results of Q1-1 and Q1-3 are empty sets, the results of Q1 are the results of Q1-2 after the processing of phase LPO_. Similarly, the postprocessor of DB2 appends the constant value “NCTU” for the attribute school to the results of Q2-1 in phase LPO_. Then, the results of Q2 are obtained. Finally, the results of Q1 and Q2, called r1 and r2, are returned to the 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 isomeric objects which have the same GOid. For each objec in r2, if its isomeric object 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, then this maybe result is eliminated from the results. Otherwise, it remains as a maybe result. Therefore, the certain result cr2 and maybe result mr2 of r2 are obtained after processing in phase RI_. Finally, the certain result of the global query is r1 ⊕ cr2. Since there is no maybe result in r1, the global maybe result is mr2. Phase RI_2 is omitted because no explicit join is specified in query Q.

5. CONCLUSIONS

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

There are many ways to decompose a global query; thus, many different query execution plans can be produced. In this paper, we have only provided one way to decompose a global query. For a defined cost model, query transformation based on the estimated cost can provide a method to find an efficient query execution plan. Alternatively, increased intersite parallelism can reduce the response time of query processing. The query optimization strategies will be subjects of future research.

REFERENCES


**Jia-Ling Koh** (柯佳伶) received the B.S. degree in computer science from National Chiao-Tung University, Taiwan, R.O.C. in 1991, the M.S. and the Ph.D. degrees in computer science form the National Tsing Hua University, Taiwan in 1993 and 1997, respectively. She is currently an associate professor in the Department of Computer Education, National Taiwan Normal University. Her currently research interests include multimedia information retrieval and data mining.

**Arbee L. P. Chen** (陳良弼) received the B.S. degree in computer science from National Chiao-Tung University, Taiwan in 1977, and the Ph.D. degree in computer engineering form the University of Southern California in 1984.

He joined National Tsing Hua University (NTHU), Taiwan, as a National Science Council (NSC) sponsored Visiting Specialist in August 1990, and became a Professor of the Department of Computer Science, NTHU, in 1991. In August 2001 he took a leave from NTHU and assumed the position of the Chairman of the Department of Computer Science and Information Engineering at National Dong Hwa University, Taiwan. He was a Member of Technical Staff at Bell Communications Research, New Jersey, from 1987 to 1990, an Adjunct Associate Professor in the Department of Electrical Engineering and Computer Science, Polytechnic University, New York, and a Research Scientist at Unisys, California, from 1985 to 1986. His current research interests include multimedia databases, data mining and mobile computing.

Dr. Chen has organized (and served as a Program Co-Chair) 1995 IEEE Data Engineering Conference and 1999 International Conference on Database Systems for Advanced Applications (DASFAA) and served as a Program Co-Chair for 2000 Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD) and 1997 International Conference on Cooperate Information Systems (CooPIS). He is an editor