Schema Integration and Query Processing for Multiple Object Databases *

Arbee L.P. Chen, **Jia-Ling Koh**, **Tony C.T. Kuo**, and **Chih-Chin Liu**

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

Abstract

In a multiple database system, 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 main problem for constructing a global schema is to resolve conflicts among component schemas. In this paper, we define *corresponding assertions* for the database administrators to specify the semantic correspondences among component object schemas. Based on these assertions, *integration rules* are designed, which use a set of primitive integration operators to restructure the component schemas for resolving the conflicts and do the integration. The principle of our integration strategy is to keep the data of component databases retrievable from the global schema without losing information. Moreover, more informative query answers may be derived from the multiple databases due to schema integration. The strategies for processing the global queries are proposed, which use the provided mapping information between global schema and component schemas to decompose the global queries into a set of subqueries. A Flow Control Language is then defined to specify the execution flow of the subqueries as well as the integration of the partial results. Some query optimization techniques are considered in the specification of the execution flow.

Keyword: Multiple Object Databases, Schema Integration, Global Query Processing.

*This work was partially supported by the Republic of China National Science Council under Contract No. NSC 83-0408-E-007-030.

1 Introduction

In recent years, computer networks and database systems develop rapidly. In this environment, it is more and more important to share data in distributed autonomous databases. Since the databases were designed independently, one of the difficult problems is the incompatibility among databases. In order to provide a uniform interface and high level location transparency for the users to retrieve data, a global schema is needed which can be created by integrating schemas of the component databases. In this paper, we study the strategy for integrating multiple object schemas, and the mechanism to process global queries against the integrated global schema.

A variety of approaches to schema integration have been proposed [1], [2], [8], [6], [9], [10], [14], [17]. 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 [4]. Schema and domain incompatibilities were considered in [4], [7], [15]. The issues on implementing schema integration tools were reported in [10], [11], [23]. In [11], a knowledge based system was developed to support the view integration. On the other hand, an interactive tool to get the information required for the integration from a database administrator (DBA) and to integrate schemas according to the provided semantics was presented in [23]. In [10], for automating much of the integration process, the idea is to embed corresponding tools within the view integration process. As a similar approach to [23], the assertion-based approach was used in [18], [24]. In [24], this approach developed integration rules for a variety of data models. A generic description of schema correspondences among different data models was provided. Other approaches defined a set of operators to build a virtual integration of multiple databases or to customize virtual classes [19], [20], [22]. Yet another approach asserted that the different constructs of component schemas be standardized before the integration. Several transformation rules were then proposed for the view integration process [12].

In this paper, we present a schema integration mechanism to achieve a global object schema for existing object databases. The main problem is to resolve different conflicts

among component object schemas, such as attribute conflicts, class hierarchy conflicts, etc. We first define *corresponding assertions* for the 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 principle of our integration strategy is to keep the data of component databases retrievable from the global schema without losing information. Moreover, more informative query answers may be derived from the multiple databases due to schema integration. Besides, the mapping information between the global schema and component schemas will be generated in the process of schema integration.

Our approach is similar to the assertion-based approach in [24]. However, we consider only object schemas; there is no need to transform the component schemas to ones in certain generic data model. Besides, we consider the construction of class hierarchies in the integrated schema, which was not discussed in [24]. In the process of schema integration, we use integration operators to restructure and integrate the component schemas. Different from the operators provided in [19] and [22], our *class restructuring operators* can be used to restructure the attributes and class hierarchies of a class. Thus, the conflicts in component schemas can be resolved before the integration. The *class integration operators* can be used to integrate the information in two classes or a set of attributes and a class. In addition, inheritance models of objects in the class hierarchy are considered in our discussion. Furthermore, our integration operators contrast with the integration and transformation primitives in [10], where only the conditions and the result when they are applied are specified. No explicit primitive operators are provided for the process of schema integration.

Since a global schema is actually a virtual schema, queries against the global schema should be decomposed and dispatched to corresponding local databases for execution. Issues about transforming a query into a set of subqueries were discussed in [14], where only query processing for class hierarchies was considered. In [21], a Distributed Operation Language

was proposed for the specification of query execution plans in multidatabase systems. This language lacks the capability to specify the execution coordination among local databases. [5] proposed a message-based approach to retrieve data in class composition hierarchies. However, the integration of partial results from subqueries was not supported. [13] proposed query processing strategies in the distributed object database system. However, the proposed strategies only dealed with queries on complex objects without a discussion on class hierarchies. Moreover, semantic discrepencies which exist in multiple object database systems were not considered. In this paper, the strategies for processing global queries against the integrated global schema are proposed. The existence of objects which represent the same entity in the real world is considered, and if they qualify the query predicates, these objects are integrated as an object in the query result. Also, a Flow Control Language for handling the execution flow of the subqueries as well as the integration of the partial results is presented.

This paper is organized as follows. A mechanism for object schema integration is provided in Section 2. Section 3 presents the processing of global queries. Finally, Section 4 concludes this paper with a discussion on the future work.

2 Integration of Multiple Object Schemas

2.1 Corresponding Assertions

In order to provide the information for schema integration, the DBAs have to specify the corresponding assertions between schemas in different component databases. In addition to specifying the corresponding assertions, the DBAs have to specify the *division characteristics* for the classes which have subclasses. The *division characteristics* of a class are properties which can denote the differences among the subclasses of the class. For example, class **Person** has subclasses **Man** and **Woman**. Then, *sex* is the *division characteristic* of **Person**.

There are four kinds of corresponding assertions according to different correspondences among classes, attributes or composition hierarchies.

1. Corresponding assertions among classes

   The correspondences among classes are specified based on the relationships among *semantic domains* of the classes. The *semantic domain* of a class is the set of real world entities that the class can represent.

   - *Class-Equivalent*

     Classes **X1** and **X2** are *class-equivalent* means that the semantic domains of **X1** and **X2** are the same. There are two kinds of *class-equivalent*: *implicit class-equivalent* and *explicit class-equivalent*. If the identities of the databases where **X1** and **X2** come from need to be kept for the virtual objects in the integrated virtual class, the two classes are *explicit class-equivalent*. Otherwise, they are *implicit class-equivalent*.

   - *Class-Correspondent*

     Classes **X1** and **X2** are *class-correspondent* if classes **X1** and **X2** are semantically related but not equivalent. Based on the relationship between *semantic domains* of two classes, three kinds of class correspondences are identified: *class_containment*, *class_overlap* and *class_disjointness*.

2. Corresponding assertions among attributes

   Only when classes are specified as *class-equivalent* or *class-correspondent* can the DBA specify the corresponding assertions among attributes in these classes.

   - *Attribute-Equivalent*

     Two primitive attributes **A1** and **A2** are *attribute-equivalent* if they are semantically equivalent. For example, attribute **Student**.*name* in database1 and attribute **Student**.*s-name* in database2 are *attribute-equivalent*.

   - *Attribute_Set-Equivalent*

     Attribute sets **S1** = { a_1, a_2, ..., a_n } and **S2** = { a'_1, a'_2, ..., a'_m } are *attribute_set-equivalent* if they are semantically equivalent. For example, attribute set { *city,*

street, no } of class **Person** in database1 and attribute set { *address* } of **Person** in database2 are *attribute_set-equivalent.*

3. Corresponding assertions among classes and attributes

   - *Attribute_Set - Class-Equivalent*

     Attribute set $\mathbf{S} = \{ a_1, a_2, ..., a_n \}$ and class **X** are *attribute_set - class-equivalent* if attribute set **S** and class **X** are semantically equivalent. For example, attribute set { *blood-type* } of class **Person** in database1 and class **Blood** in database2 are *attribute_set - class-equivalent.*

   - *Attribute - Class_Set-Equivalent*

     If attribute **A** is semantically equivalent to a *division characteristic* of class **C**, then **A** and the associated subclasses of **C** are *attribute - class_set equivalent.* By this information, **A** can be used to derive a set of subclasses for its associated class. Each subclass will be *class-equivalent* to a subclass of **C**. For example, attribute **Person**.*sex* in database1 and the subclasses { **Man**, **Woman** } of class **Person** in database2 are *attribute - class_set-equivalent.*

4. Corresponding assertions among composition hierarchies

   - *Composition_Hierarchy-Equivalent*

     Let **P1** be a path from class **C1** to class $\vec{C1}$ in the composition hierarchy of database1, and **P2** be a path from class **C2** to class $\vec{C2}$ in the composition hierarchy of database2. Paths **P1** and **P2** are *composition_hierarchy-equivalent* if both **C1** and $\vec{C2}$, and **C2** and $\vec{C1}$ are *class-equivalent.* Besides, paths **P1** and **P2** are semantically equivalent. For example, both **Person**.*car* in database1 and **Car**.*owner* in database2 denote the ownership of a car. Thus, the two paths are *composition_hierarchy-equivalent.*

2.2 Primitive Integration Operators

We define a set of operators for integrating object schemas. These operators can be categorized into *class restructuring* and *class integration operators*. *Class restructuring operators* are used to restructure the classes in component schemas to resolve their conflicts. After the restructuring process, *class integration operators* are then used to integrate classes. The class hierarchies and class composition hierarchies among classes in different component schemas will also be built by the *class integration operators*.

2.2.1 Class restructuring operators

Class restructuring operators may change the structure of attributes in a class and the structure of a class hierarchy. We provide seven *class restructuring operators* as follows.

1. *Refine*

   The *Refine* operator adds an attribute to a class. The added attribute is assigned a constant value for saving certain semantic information. It has the following syntax:
   Refine(*source-class*, *new-attribute*, *constant-value*).

2. *Hide*

   The *Hide* operator removes an existing attribute from a class. It has the following syntax:
   Hide(*source-class*, *hidden-attribute*).

3. *Rename*

   The *Rename* operator renames a class name or attribute name by a new name. It has the following syntax:
   Rename(*source-class*(.*source-attribute*), *new-class*(*attribute*)-*name*).

4. *Aggregate*

   The *Aggregate* operator aggregates a set of primitive attributes of a class into a complex attribute. Besides, a new virtual class is created to be the domain class of the new

complex attribute. It has the following syntax:
Aggregate(*source-class*, [*attribute-list*], *new-complex-attribute*, *new-domain-class*).

5. *Invert*

   The *Invert* operator defines a new complex attribute which is the inverse of a complex attribute in another class. It has the following syntax:
   Invert(*source-class*, *inverted-attribute*, *new-complex-attribute*).

6. *Build*

   The *Build* operator creates a new virtual class containing virtual objects satisfying a simple predicate clause on an attribute from a given class. It has the following syntax:
   Build(*source-class*, *new-class*, [*predicate clause*]).
   In the class hierarchy, *source-class* is the superclass of the *new-class* in the global schema.

7. *Demolish*

   The *Demolish* operator demolishes the subclasses of some class to a set of attributes in the new virtual class. It has the following syntax: *Demolish*(*source-class*).
   Note that the *Demolish* operator is recursive. That is, the subclasses will be *Demolished* if they also have subclasses.

2.2.2 Class integration operators

Class integration operators are used to integrate classes from different component databases. Some operators are used to build the classes of global schema by integrating two classes. The other operators can be used to build more complete class hierarchies and class composition hierarchies in the global schema. Five *class integration operators* are defined.

1. *OUnion*

   The *OUnion* operator is the *set-union* of objects in two equivalent classes. It has the following syntax: *OUnion*(*source-class1*, *source-class2*, *new-class*);
   with *new-class* being created to be the result. Only *new-class* will appear in the global

schema. The attributes of the new virtual class are the *set-union* of the attributes of the two operands.

2. *Generalize*

   The *Generalize* operator creates a common superclass of two classes. It has the following syntax: *Generalize*(*source-class1*, *source-class2*, *common-superclass*).
   The attributes of the new virtual class – *common-superclass* are the *set-intersection* of the attributes in the operand classes. The two operand classes will become two virtual subclasses under the *common-superclass*.

3. *Specialize*

   The *Specialize* operator creates the common subclass of two classes. It has the following syntax : *Specialize*(*source-class1*, *source-class2*, *common-subclass*).
   The attributes of the new virtual class – *common-subclass* are the *set-union* of the attributes in the two operand classes. There will be two virtual classes produced as the superclasses of the *common-subclass* in the global schema.

4. *Inherit*

   The *Inherit* operator builds the class hierarchy relationship of two classes. It has the following syntax: *Inherit*(*source-subclass*, *source-superclass*);
   with *source-superclass* being built as the superclass of *source-subclass*. Two virtual classes are produced in the global schema. One class corresponds to the *source-superclass* with attributes being the same as those in *source-superclass*. The other class corresponds to the *source-subclass* whose attributes are the same as the *source-subclass*. Besides, it will inherit the attributes of the *source-superclass*.

5. *Upgrade*

   The *Upgrade* operator upgrades a set of attributes in a class to a complex attribute. The domain class has to be an existing class in another component database. It has the following syntax: *Upgrade*(*source-class*, [*attribute-list*], *new-complex-attribute*, *domain-class*).

After the *Upgrade* process, the *source-class* is modified to a virtual class with the same attributes except the *attribute-list* being replaced by *new-complex-attribute*.

One purpose of the class integration operations is to integrate the objects in different databases, which represent the same real-world entity, to a single object for answering queries against the global schema. However, the same real-world entity may be stored in different object databases with incompatible object identifiers. [3] proposes a mechanism to identify the objects representing the same entity, called *isomeric objects*, among classes in different databases, and to integrate them into a single object. In this paper, we assume that the isomeric objects have been identified. For the global query processing, we assign a global object identifier (GOID) to each object in the multiple database system. The isomeric objects are assigned the same GOID.

2.3 Integration Rules

Integration rules are the major part that guides the integrator to do the actual schema integration. According to the specified corresponding assertions, there are three kinds of situations where the classes have to be integrated or certain relationships built between them. In the following, three situations will be discussed, followed by an overall application of the integration rules to an example. The principle of our integration strategy is to keep the data of component databases retrievable from the global schema without losing information. Besides, we may derive more informative query answers from the multiple databases due to schema integration.

2.3.1 Integration rules for class-equivalent classes

When classes **X1** and **X2** are *class-equivalent*, they can be integrated into a single virtual class. In the integration process, the first four steps resolve some possible conflicts among these two classes, and the last step builds the integrated virtual class. These five steps are repeated for all class-equivalent assertions between the component schemas. The integration

rules are applied to classes in the top-down order of the class hierarchies.

Step 1 is used to resolve the conflicts of class hierarchies among classes **X1** and **X2**.
[**step 1**] Consider when **X1** or **X2** has subclasses.
Use *Demolish* or *Build* operators to build the one-to-one *class-equivalent* correspondence between direct subclasses of **X1** and **X2**, we do nothing at **step 1**.

Step 2 is used to resolve the conflicts of attribute structures among classes **X1** and **X2**.
[**step 2**] Consider when there are *attribute_set-equivalent* assertions between **X1** and **X2**. If the assertion is between a complex attribute and a set of primitive attributes, the set of primitive attributes are *Aggregated*. The aggregated new class and the domain class of the complex attribute are specified to be *class-equivalent*. Otherwise, both sets of primitive attributes are *Aggregated*. Two aggregated new classes are specified to be *class-equivalent*.

Step 3 is used to resolve the conflicts of composition hierarchies between classes **X1** and **X2**.
[**step3**] Consider when there are *composition_hierarchy-equivalent* assertions between the composition hierarchies of **X1** and **X2**. In this situation, the complex attributes which form the equivalent composition hierarchies should be *Inverted*.

[**step 4**] Consider when **X1** and **X2** are *explicit class-equivalent*. In this situation, both **X1** and **X2** have to be *Refined*. The values of the new attributes specify which database an object comes from after the integration. Thus, both attributes are assigned a constant value.

[**step 5**] Processes the integration of **X1** and **X2**. We use *OUnion* operator to integrate these two classes. The virtual objects in the integrated class may contain more information if they denote the same real-world entities.

2.3.2 Integration rules for class-correspondent classes

When classes **X1** and **X2** are *class-correspondent*, we have to build a class hierarchy for these two classes. Some conflicts in the structures of the attributes have to be resolved too.

Step 1 is used to resolve the conflicts of attribute structures among classes **X1** and **X2**.
[**step 1**] Consider when there are *attribute_set-equivalent* assertions between **X1** and **X2**. This step is the same as **step 2** in Subsection 3.3.1.

Step 2 builds a class hierarchy structure for classes **X1** and **X2**.
[**step 2**] Consider the different kinds of *class-correspondent* assertions.
[**step 2-1**] If **X1** and **X2** are *class_containment-correspondence*, *Inherit* operator is used to build a class hierarchy relationship between them.
[**step 2-2**] If **X1** and **X2** are *class_overlap-correspondence*, *Generalize* and *Specialize* operators are used to build the common superclass and subclass of both classes, respectively.
[**step 2-3**] Otherwise, **X1** and **X2** are *class_disjointness-correspondence*. We use *Generalize* operator to build the common superclass of these two classes.

These two steps are repeated for all the specified *class-correspondent* classes to build the structure of class hierarchies in the global schema.

2.3.3 Integration rules for attribute_set - class-equivalent

When attribute set **S** = { a_1, a_2, ..., a_n } and class **X** are *attribute_set - class-equivalent*, we try to upgrade the attribute set **S** to get more information from **X**. However, the purpose can be satisfied only in some situations. When each set of values for **S** corresponds to an object in **X**, the attribute set **S** can be *Upgraded* to a complex attribute and linked to class **X**. Otherwise, the set of attributes in **S** have to be *Aggregated*. The aggregated new class and class **X** are specified to be *class-equivalent*.

2.3.4 The overall application of different integration rules

When applying the different integration rules to integrate component schemas, the following order is followed:

- Integration rules for *attribute_set - class_equivalent*
- Integration rules for *class-equivalent*
- Integration rules for *class-correspondent*
- Put in those classes without corresponding assertions

Now, we consider an example to describe the overall processing of schema integration. Figure 1 shows the schemas of two databases: database1 and database2. They are databases in two different schools, used to store the personal information.

First, the corresponding assertions among these two schemas are specified. The *class-equivalent* classes and their *attribute-equivalent* or *attribute_set-equivalent* attributes are specified in Table 1 to Table 4.

Besides, **Person@DB1**.car and **Car@DB2**.owner are *composition_hierarchy-equivalent*. In addition, **Student@DB1**.department and {**CS-Student@DB2**, **EE-Student@DB2**} are *attribute - class_set-equivalent*. **Employee@DB1**.position and {**Faculty@DB2**, **Staff@DB2**} are *attribute - class_set-equivalent*, too.

Then, according to the corresponding assertions specified, the integration rules are applied.

- Apply integration rules for *class-equivalent* classes:

  1. Integrate **Person@DB1** and **Person@DB2**:
     Upgrade(**Person@DB1**, [blood-type], blood, **Blood@DB2**) (step 2)
     Aggregate(**Person@DB2**, [city, street, no], address, **Address@DB2**) (step 2)
     Invert(**Car@DB1**, **Person@DB1**.car, owner) (step 3)
     Invert(**Person@DB2**, **Car@DB2**.owner, car) (step 3)

Refine(**Person@DB1**, school, "NCTU") (step 4)

Refine(**Person@DB2**, school, "NTHU") (step 4)

OUnion(**Person@DB1**, **Person@DB2**, **Person**) (step 5)

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

   OUnion(**Address@DB1**, **Address@DB2**, **Address**) (step 5)

3. Integrate **Car@DB1** and **Car@DB2**:

   OUnion(**Car@DB1**, **Car@DB2**, **Person**) (step 5)

4. Integrate **Student@DB1** and **Student@DB2**:

   The *division characteristic* of **Student@DB1** is *degree* and the *division characteristic* of **Student@DB2** is *department*. We select *department* as the *division characteristic* of **Student** in the global schema.

   Demolish(**Student@DB1**) (step 1-2)

   Build(**Student@DB1**, **CS-Student@DB1**, [department="CS"]) (step 1)

   Build(**Student@DB1**, **EE-Student@DB1**, [department="EE"]) (step 1)

   OUnion(**Student@DB1**, **Student@DB2**, **Student**) (step 5)

5. Integrate **Employee@DB1** and **Employee@DB2**:

   The *division characteristic* of **Employee@DB2** is *position*.

   Build(**Employee@DB1**, **Faculty@DB1**, [position="faculty"]) (step 1)

   Build(**Employee@DB1**, **Staff@DB1**, [position="staff"]) (step 1)

   OUnion(**Employee@DB1**, **Employee@DB2**, **Employee**) (step 5)

6. Integrate **CS-Student@DB1** and **CS-Student@DB2**:

   OUnion(**CS-Student@DB1**, **CS-Student@DB2**, **CS-Student**) (step 5)

7. Integrate **EE-Student@DB1** and **EE-Student@DB2**:

   OUnion(**EE-Student@DB1**, **EE-Student@DB2**, **EE-Student**) (step 5)

8. Integrate **Faculty@DB1** and **Faculty@DB2**:

   OUnion(**Faculty@DB1**, **Faculty@DB2**, **Faculty**) (step 5)

9. Integrate **Staff@DB1** and **Staff@DB2**:

   OUnion(**Staff@DB1**, **Staff@DB2**, **Staff**) (step 5)

- Put in those classes without corresponding assertions:

  Blood@DB2 and **Course@DB2** are put into the global schema.

The resultant global schema is shown in Figure 2.

2.4 Mapping between Global Schema and Component Schemas

Our mapping strategy uses the *mapping table* to save the mapping information with the classes in component schemas. Each virtual class produced in the process of schema integration has a corresponding *mapping table*. As the processing of schema integration, the contents of the *mapping tables* are modified to respond to the result of every restructuring or integration operation. When schema integration is complete, the information in the *mapping tables* for every class in the global schema is stored in the DD/D.

Two major parts in the *mapping tables* are *class mapping records* and *attribute mapping records*. The *class mapping record* consists of three fields, as shown in Table 5. They are **class name**, **class type** and **expression**. **Class name** is the name of the virtual class having the *mapping table*. There are two kinds of **class types** which denote the constituent characteristic of the virtual class. One is *simple*, which denotes that the virtual objects in the virtual class correspond to the objects in a class in certain component schema. The other one is *multiple*, denoting that the virtual objects in the virtual class are produced from integrating the objects in two *simple* classes. In addition, **expression** field records the classes in component schemas and operations which construct the virtual class.

According to the **class type** in the *class mapping record*, there are two kinds of *attribute mapping records*. They are shown in Table 6 and Table 7, respectively. The *attribute mapping record* in Table 6 is applied when the **class type** in the *class mapping record* is *simple*. In Table 6, *attribute name* denotes an attribute in the virtual class. Attributes in the virtual class may result from different operations. Thus, there are eight different kinds of *attribute types* to denote the constituent characteristic of the attribute. The *attribute types* and their

meanings are listed below:

[**s**] : the attribute is originally in the **s**ource constituent classes;

[**r**] : the attribute is produced from **r**efine operator;

[**n**] : the attribute is re**n**amed;

[**a**] : the attribute is produced from **a**ggregate operator;

[**i**] : the attribute is produced from **i**nvert operator;

[**b**] : the attribute is used for **b**uild operator;

[**d**] : the attribute is produced from **d**emolish operator;

[**o**] : the attribute comes from **o**ther classes.

For different kinds of *attribute types*, the **parameters** are needed to denote the parameters which the attribute is composed of. The other form of *attribute mapping record* shown in Table 7 is applied when the **class type** in the *class mapping record* is *multiple*. In Table 7, *attribute name* denotes an attribute in the virtual class. Each set of **a-name**, **a-type**, and **parameters** is called an *attribute mapping field*. Since this kind of virtual class is produced from integrating two *simple* classes, each *attribute mapping field* corresponds to an *attribute mapping record* in one constituent *simple* class. Each row in the second kind of *attribute mapping record* stores the attribute mapping information for the semantic equivalent attributes in two constituent classes.

Before the schema integration, the *mapping tables* are produced for every class **X** in each component database **I**. The initial form of each *mapping table* is shown in Table 8. Notice that there is one *attribute mapping record* in the *mapping table* for each attribute (including inherited one) in class **X**.

During the process of schema integration, the *mapping tables* are modified to respond to the result of different operations. For the example in Subsection 2.3.4, the mapping tables for classes **Person** in the two component databases are described as follows. The *mapping table* for class **Person** in database **DB1** after *Upgrade* and *Refine* operations is shown in Table 9. In addition, the *mapping table* for class **Person** in database **DB2** after *Aggregate*, *Inverse*, and *Refine* operations is shown in Table 10.

After classes **Person** in the two databases are *OUnioned*, a new *mapping table* for class **Person** in the global schema is created as Table 11. The two *mapping tables* listed in Table 9 and Table 10 are removed. The *mapping table* shown in Table 11 is stored in the DD/D for class **Person** in the global schema.

3 Global Query Processing

In our schema integration, we allow vertical integration (i.e., class inheritance hierarchy) and horizontal integration (i.e., class composition hierarchy), and consider various schema conflicts. These complicate the global query processing. For example, a path expression specified against the global schema may involve more than one class from different sites. There are no actual links among these classes. Join-like actions have to be performed to simulate the links among these classes.

Our global query processing starts with constructing a query execution plan. In a query execution plan, the global query is decomposed to subqueries, each formed as a job and dispatched to the associated site for execution. All these jobs are coordinated and the partial results integrated to form the final result. A Flow Control Language (FCL) is proposed in the following to specify the query execution plan. Query optimization techniques are also studied in order to construct an optimized query execution plan.

Moreover, the concept of global object identifier (GOID) is employed to deal with object integration from different component databases.

3.1 Flow Control Language

The structure of FCL is shown in Figure 3, and described in the following.

The **JobId** is assigned by the system as an identifier and used to coordinate with other jobs. A job is sent to the site specified in the **To** field. Its execution is deferred when jobs specified in the **Wait** field have not completed at that site. It means the partial results from the jobs in the **Wait** field are required for the execution of the deferred job. When all of these jobs have completed, the subquery specified in the **QueryBody** field is performed.

The additional action (such as scale convertion) specified in the **Do** field is executed after the subquery has been executed. An example of FCL specification will be given later.

3.2 Procedure for Global Query Processing

In our query model, a global query is composed of three clauses: select, from and where clauses (as in XSQL [16]). The format is shown as follows:

select $< target >$

from $< range >$

where $< predicate >$

The *target* indicates what objects the user wants to retrieve, and *predicate* specifies the qualification of these objects. The *range* informs the query processor of the classes involved in the query. When a global query is issued, we process it in the following steps:
[Step 1] According to the *range*, we examine the global mapping table to find the involved classes in local databases. The global query is decomposed into subqueries whose *range* are classes in a single component database. The select and where clauses in the subqueries remain the same as the global query in this step.
[Step 2] If a *refined* attribute appears in the *target*, we get the value from the global data dictionary. If it appears in the *predicate*, we get the value of this attribute from the global data dictionary, and evaluate the associated predicates. Suppose P_r denotes the associated predicates, P_o the other predicates, and P the original predicate (i.e., $P = P_r \theta P_o$, where θ is a Boolean operator). There are four conditions after P_r is evaluated:

1. P_r is true and $\theta = and$, then P_r can be removed.

2. P_r is true and $\theta = or$, then P is true no matter what P_o is. We can modify the *predicate* to $True$ in this condition.

3. P_r is false and $\theta = and$, then P is false no matter what P_o is. We can return an empty result for this subquery without executing it at local site.

4. P_r is false and $\theta = or$, then P_r can be removed.

[Step 3] For each subquery S_i, we remove the attributes from the *target* and *predicate*, which do not belong to the classes in the *range*, and we add an attribute (named oid) into the *target* for retrieving the oid's of the qualified objects of this subquery. Each subquery can then be executed at the associated site. Since not all attributes in the orginal *target* and *predicate* appears in the subquery, the result objects of each subquery may lack some *target* attributes and may need further qualification. We further process these partial results as follows.

1. The local oid's in each partial result are mapped to GOID's by looking up the local oid-GOID mapping table.

2. Combine the GOID's of the partial results by the Boolean operators which connect the predicates in the subqueries into the global *predicate*. The results are the objects which qualify the global *predicate*.

   For example, assume the *predicate* of a global query is "A>10 *and* B=8," where attribute A comes from site 1 and B from site 2. Thus, we have two subqueries S_1 and S_2 executed at site 1 and site 2, respectively. Results of S_1 are qualified by "A>10" and S_2 by "B=8." We perform *and* operation on the GOID's of the two partial results. The results are the objects which qualify "A>10 *and* B=8."

3. A template is created to form the query answer. It consists of all target attributes and a GOID attribute which is used to match the objects from the partial results. We put the partial results from 2. into the template object by object. If the GOID of the object exists in the template, we just fill the absent attribute values. If the object cannot be found in the template, we create a new record for this object and store the data of this object in the template. Notice that all the data should be converted to the data type and scale of the global sehema before filled into the template. Finally, after all partial results are filled into the template, the template is sent to the query site as the final result.

3.3 Example

In the following, we give an example to illustrate our procedure for global query processing. Consider query Q: "Retrieve the employee's name and salary who works for NTHU or whose salary is greater than 30000," as shown in Figure 4(a). By Step 1, Q is decomposed into Q1 and Q2 as in Figure 4(b). By Step 2, Q1's *predicate* is reduced to X.salary>30000 and Q2's to *True*. This is because school is a *refine* attribute with value "NCTU" in DB1 and "NTHU" in DB2. Now consider the *target*. Q2's salary is removed from the *target* because the salary attribute is not in class Employee of DB2. The modified subqueries are shown in Figure 4(c). Finally, we create three jobs in FCL as shown in Figure 5 to specify the execution plan for this query. Job1 and job2 are dispatched to DB1 and DB2, respectively, to retrieve data. Job3 waits for the partial results from job1 and job2 to form the final result. Notice that both predicates of Q1' and Q2' are equivalent to the global predicate, we just merge their partial results by UNION operation as the final result.

4 Conclusion and Future Work

To provide an interface for users to retrieve data from multiple object databases, we present the strategy to integrate local object schemas into a global object schema and deal with the global query processing against the global schema.

We provide simple corresponding assertions for a DBA to specify the semantic correspondences between component schemas. Then, a set of primitive integration operators are defined to modify or integrate the component schemas. Besides, integration rules are formulated to guide the integration process. Our strategy resolves conflicts in attribute structures or class hierarchies in component schemas, provides schema transparency, keeps the data in component databases retrievable without losing information, and allows queries to get more information from the integrated schema. Moreover, *mapping tables* are produced in the process of schema integration to store the mapping information between the global schema and component schemas for global query processing.

A procedure for global query processing against the integrated global schema was pre-

sented as well as a flow control language to specify the execution. The technique of GOID is used for the integration of partial results. The predicate reduction skill is addressed when the predicat involves *refined* attributes.

The *composition_hierarchy-equivalent* is a noticeable structure conflict among component schemas. The integration rules for the general composition hierarchy equivalence and other more complex conflicts among component schemas are to be studied. In addition, restrictions can be enforced in the schema integration process to provide a global schema which allows update operations. We have discussed a procedure for global query processing. However, based on the distribution of data involved in the select and where clauses, various FCL specifications are possible. We are currently investigating several optimization techniques in order to achieve the best FCL execution.

References

[1] C. Batini, M. Lenzerini, and S.B. Navathe. "A comparative analysis of methodologies for database schema integration". *ACM Computing Surveys*, pages 323–364, 1986.

[2] Y. Breitbart, P.L. Olson, and G.R. Thompson. "Data integration in a distributed heterogeneous database system". In *Proc. of IEEE Data Engineering*, 1986.

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

[4] B. Czejdo, M. Rusinkiewicz, and D.W. Embley. "An approach to schema integration and query formulation in federated database systems". In *Proc. of IEEE Data Engineering*, pages 477–484, 1987.

[5] B. Czejdo and M. Tarylar. "Integration of Database Systems Using an Object-oriented Approach". In *Proc. of IEEE Interoperability in Multidatabase Systems*, pages 30–37, 1991.

[6] S.M. Deen, R.R. Amin, and M.C. Taylor. "Data integration in distributed databases". *IEEE Transactions on Software Engineering*, pages 860–864, 1987.

[7] L.G. DeMichiel. "Resolving database incompatibility: an approach to performing relational operations over mismatched domains". *IEEE Transaction on Knowledge and Data Engineering*, pages 485–493, 1989.

[8] U. Dayal and H.Y. Hwang. "View definition and generalization for database integration in a multidatabase system". *IEEE Transactions on Software Engineering*, pages 628–644, 1984.

[9] R. Elmasri and S. Navathe. "Object integration in logical database design". In *Proc. of IEEE Data Engineering*, pages 426–433, 1984.

[10] W. Gotthard, P.C. Lockemann, and A. Neufeld. "System-guilded view integration for object-oriented databases". *IEEE Transaction on Knowledge and Data Engineering*, pages 382–398, 1992.

[11] S. Hayne and S. Ram. "Multi-User view integration system (MUVIS) : An expert system for view integration". In *Proc. of IEEE Data Engineering*, pages 402–409, 1990.

[12] P. Johannesson. "Schema transformations as an aid in view integration". *Stockholm Univ. Working Paper*, 1992.

[13] B. Paul Jenq, Darrell Woelk, Won Kim, and Wan-Lik Lee. "Query Processing in Distributed ORION". *MCC Technical Report Number: ACA-ST-035-89*, pages 1–26, Jan 1989.

[14] M. Kaul, K. Drosten, and E.J. Neuhold. "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views". In *Proc. of IEEE Data Engineering*, pages 2–10, 1990.

[15] W. Kent. "Solving domain mismatch and schema mismatch problems with an object-oriented database programming language". In *International Conf. on Very Large Data Bases*, pages 147–160, 1991.

[16] Michael Kifer, Won Kim, and Yehoshua Sagiv. "Querying Object-Oriented Databases". In *Proc. of ACM SIGMOD*, pages 393–402, 1992.

[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*, pages 449–463, 1989.

[18] M.V. Mannino and W. Effelsberg. "Matching techniques in global schema design". In *Proc. of IEEE Data Engineering*, pages 418–425, 1984.

[19] A. Motro. "Superviews : Virtual integration of multiple databases". *IEEE Transactions on Software Engineering*, pages 785–798, 1987.

[20] E.A. Rundensteiner and L. Bic. "Set operations in object-based data models". *IEEE Transaction on Knowledge and Data Engineering*, pages 382–398, 1992.

[21] M. Rusinkiewicz, R. Elmasri, and B. Czejdo. "ONMIBASE: Design and Implementation of a Multidatabase System". *Distributed Processing Technical Committee Newsletter, IEEE CS*, 1988.

[22] Elke A. Rundensteiner. "MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Datanases". In *International Conf. on Very Large Data Bases*, pages 187–198, 1992.

[23] A. Sheth, J. Larson, A. Cornelio, and S. Navathe. "A tool for integrating conceptual schemas and user views". In *Proc. of IEEE Data Engineering*, pages 176–183, 1988.

[24] S. Spaccapietra, C. Parent, and Y. Dupont. "Model independent assertions for integration of heterogeneous schemas". In *International Conf. on Very Large Data Bases*, pages 81–126, 1992.

class-equivalent (explicit)	**Person@DB1**	**Person@DB2**
division characteristic	career	career
attribute-equivalent	ss#	ss-no
	name	name
attribute_set -equivalent	{blood-type}	{blood}
	{address}	{city, street, no}

Table 1: The *corresponding assertions* between classes **Person@DB1** and **Person@DB2**

class-equivalent (implicit)	**Car@DB1**	**Car@DB2**
attribute-equivalent	license-no	car-no

Table 2: The *corresponding assertions* between classes **Car@DB1** and **Car@DB2**

class-equivalent (explicit)	**Student@DB1**	**Student@DB2**
division characteristic	degree	department
attribute-equivalent	s-no	stu-no

Table 3: The *corresponding assertions* between classes **Student@DB1** and **Student@DB2**

class-equivalent (explicit)	**Employee@DB1**	**Employee@DB2**
division characteristic		position
attribute-equivalent	e-no	e-no

Table 4: The *corresponding assertions* between classes **Employee@DB1** and **Employee@DB2**

class name	**class type**	**expression**

Table 5: The *class mapping record*

attribute name	**attribute type**	**parameters**
...	...	...

Table 6: The *attribute mapping record* for *simple* **class type**

attribute name	**a-name**	**a-type**	**parameters**	**a-name**	**a-type**	**parameters**
...	...	...	...	...	...	...

Table 7: The *attribute mapping record* for *multiple* **class type**

X@DB**I**	*simple*	**X**@DB**I**
attribute	[**s**]	
...	...	

Table 8: The initial mapping table for each class **X** in database **I**

Person@DB1	*simple*	**Person@DB1**
ss#	[**s**]	
name	[**s**]	
blood	[**u**]	[blood-type], **Blood@DB2**
address	[**s**]	
car	[**s**]	
school	[**r**]	"NCTU"

Table 9: The *mapping table* for class **Person** in database **DB1** after *Upgrade* and *Refine* operations

Person@DB2	*simple*	**Person@DB2**
ss-no	[**s**]	
name	[**s**]	
blood	[**s**]	
address	[**a**]	[city, street, no], **create Address@DB2**
car	[**i**]	**Car**.owner**@DB2**
school	[**r**]	"NTHU"

Table 10: The *mapping table* for class **Person** in database **DB2** after *Aggregate*, *Invert*, and *Refine* operations

Person	*multiple*			**ounion**[**Person@DB1**,**Person@DB2**]		
ss#	ss#	[**s**]		ss-no	[**s**]	
name	name	[**s**]		name	[**s**]	
blood	blood	[**u**]	[blood-type], **Blood@DB2**	blood	[**s**]	
address	address	[**s**]		address	[**a**]	[city, street, no], **create Address@DB2**
car	car	[**s**]		car	[**i**]	**Car**.owner**@DB2**
school	school	[**r**]	"NCTU"	school	[**r**]	"NTHU"

Table 11: The *mapping table for class* **Person** *in the global schema*

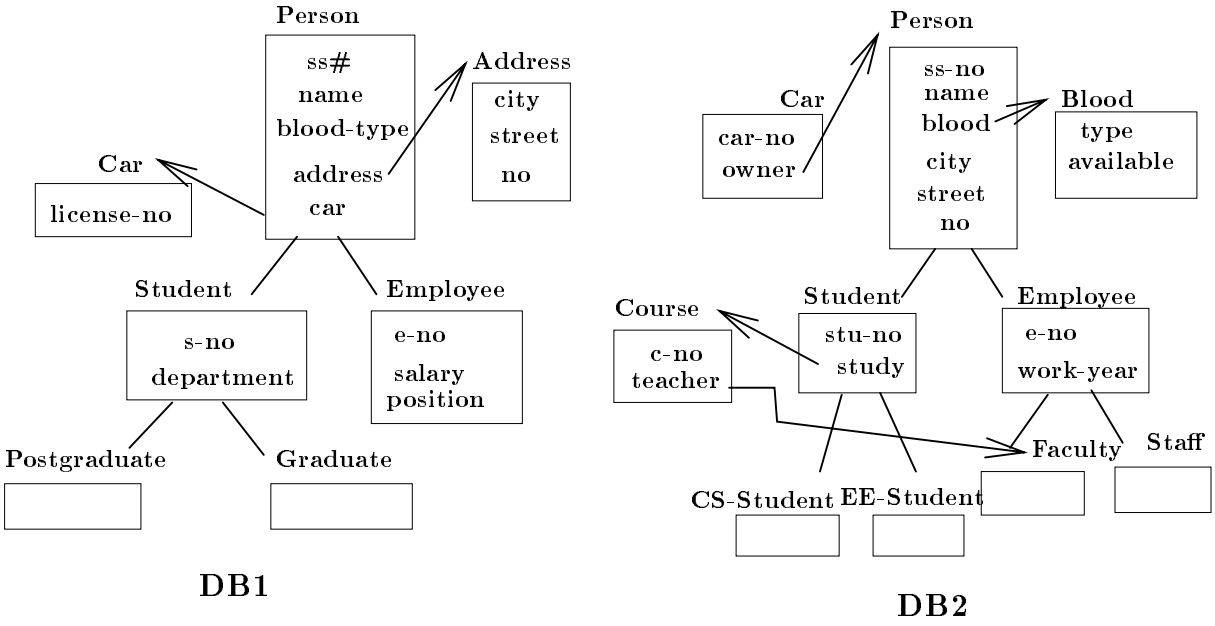

Figure 1: The component schemas in database1 and database2

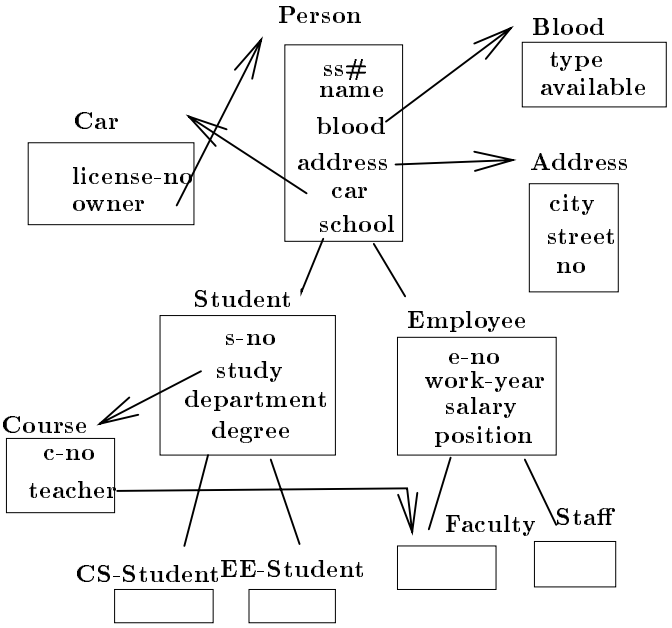

Figure 2: The integrated global schema

JobId: < *job id* >

To: < *site name* >

Wait: < *job clauses* >

QueryBody: < *subquery* >

Do: < *expressions of system functions* >

ReturnTo: < *sitenames* >

Figure 3: Structure of FCL

```
Q:  select   X.name, X.salary
    from     Employee X
    where    X.school = "NTHU" or X.salary>30000
```

(a)

```
Q1: select   X.name,X.salary                    Q2: select   X.name,X.salary
    from     Employee X                             from     Employee X
    where    X.school="NTHU" or X.salary>30000      where    X.school="NTHU" or X.salary>30000
```

(b)

```
Q1':select   X.name,X.salary          Q2': select   X.name
    from     Employee X                      from     Employee X
    where    X.salary>30000
```

(c)

Figure 4: Query modification for the example

JobId: job1
To: DB1
QueryBody: Q1'
ReturnTo: DB1

JobId: job2
To: DB1
QueryBody: Q2'
ReturnTo: DB1

JobId: job3
To: DB1
Wait: job1;job2
Do: Merge(job1,job2,Union)
ReturnTo: QuerySite

Figure 5: An FCL specification for the example