Superviews: Virtual Integration of Multiple Databases

AMIHAI MOTRO

Abstract—An important advantage of a database system is that it provides each application with a custom view of the data. The issue addressed in this paper is how to provide such custom views to applications that access *multiple* databases. The paper describes a formal method that generates such *superviews*, in an interactive process of schema editing operations. A mapping of the superview into the individual databases is derived from the editing process, and is stored together with the superview as a *virtual* database. When this database is interrogated, the mapping is used to decompose each query into a set of queries against the individual databases, and recompose the answers to form an answer to the original query. As this process is transparent to the user, virtual databases may be regarded as a more general type of databases. A prototype database system, that allows users to construct virtual databases and interrogate them, has been developed.

Index Terms—Database, database integration, database view, multidatabase environment, query mapping, superview, virtual database.

I. Introduction

An important advantage of a database is that it is a global source of data, that can satisfy all the data requirements of a given set of applications. Instead of a multitude of separate files (that between them may have overlapping or irrelevant data), a database provides each application with a single, integrated view of all the data it requires. Eventually, however, applications may evolve that are no longer satisfiable from a single database, their data extending over two or more databases. These applications must now "shuttle" between the individual databases, storing their intermediate data in separate files. Obviously, such applications cannot benefit from this important advantage of a single global view of their data. In many respects, this multidatabase situation is similar to the multifile situation before the invention of databases. In fact, so firm is the belief that a single database should be able to satisfy all the needs of every application, that many databases management systems do not even have provisions for accessing more than one database at a time, posing further difficulties for multidatabase applications.

The obvious solution for this multiple database situation, is to consolidate them into a single new database. In practice, however, this solution is acceptable in only a few cases. Because it is costly, it is justified only when the new data requirements are considered permanent, and the number of multidatabase applications is substantial and expected to grow. Also, it may be desirable to maintain the independence of the individual databases. When some of these databases are provided by external sources, consolidation is simply impossible.

An alternative solution is to develop data manipulation languages, that can access more than one database at a time. This was the approach taken by Litwin in the design of MALPHA [1], which is an extension of ALPHA [2] to allow manipulation of *collections* of relational databases. MALPHA queries can access and combine data from different databases to form a single answer.

In this paper we advocate yet a different approach to multidatabase situations. We introduce the concept of a *virtual* database, as a database that has a schema, but no data that populate this schema. Instead, a mapping is available from this schema into schemas of other databases. In other words, while *actual* databases are <schema, data> pairs, *virtual* databases are <schema, mapping> pairs. For an application that cannot be satisfied from a single database, a virtual database should be created. It will integrate the schemas of the relevant databases (or portions thereof) into a single global schema (called a *superview*), that will accommodate the new application.

In principle, the superview approach and the multidatabase language approach are not entirely different, as each MALPHA query reflects a view that integrates the individual databases. But while the MALPHA view is pertinent to one query only (and disappears when it terminates), superviews are more permanent: they are stored in virtual databases, and may even be combined to create higher-level superviews. With respect to permanency, superviews are somewhere between the query-generated views of MALPHA and physical consolidation.

Our approach here involves two separate tools: a virtual database generator, and a virtual query processor.

The *virtual database generator* is an interactive program that generates a virtual database, from the schemas of existing databases, and statements in a schema integration language that are provided by the user. These statements define the schema of the virtual database (the superview), in terms of the schemas of the existing databases.[1] The integration statements also define a mapping from the final superview into the initial databases. When this interactive process terminates, the virtual da-

Manuscript received December 28, 1984; revised July 31, 1986. This work was supported in part by an Amoco Foundation Engineering Faculty Grant.

The author is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 8714553.

[1]Therefore, the schema integration language may be regarded as the counterpart of the *data definition language*, which is for defining actual databases.

tabase generator stores this mapping, together with the superview, as a new virtual database.

The *virtual query processor* handles queries against actual databases in the usual way. To handle queries against virtual databases, it employs the mapping to decompose each query into a set of queries against the individual databases. These queries are then resubmitted against the individual databases, and the answers are recomposed to form an answer to the original query. Consequently, the effect is as if a single global database is actually available. Note that, since virtual databases can be integrated to form new virtual databases, queries submitted to the individual databases may require further translation, until they yield queries against the actual databases.

The construction of the superview may introduce integrity constraints. For example, if two personnel databases with age information are integrated, the age values must agree for each employee that appears in both databases. Such constraints are checked at the time of query processing; if they exist and are violated, they only cause failure when an attempt is made to interpret a relevant query.

An important goal of this paper is to discuss the issues of database integration within a formal framework. Towards this goal, we develop an *abstract* model, that attempts to capture formally the principles involved in the constructing and interrogation of virtual databases, while neglecting some of the more pragmatic, albeit important, issues. The model has two parts: a database model (with access operators), and an integration language. The database model is based on the functional approach. It incorporates important database concepts, such as classes, types, domains, keys, attribute relationships, and generalization relationships. To simplify the analysis, both the access operators and the integration language are based on small sets of primitives. In practice, however, a greater variety of more powerful operators may be introduced.

A. Related Research

Two multidatabase system currently under development are Multibase [3]-[5] and ADDS [6]. Both systems are designed to integrate heterogeneous databases (Multibase will handle relational and network databases and simple files; ADDS will handle relational, network, and hierarchical databases). In both systems integration is achieved through a global schema that is mapped onto the schemas of the existing databases (Multibase uses the functional model as its unifying data model; ADDS uses an extended relational model). Users are then allowed to present queries against this global schema, and the system responds with answers composed of data retrieved from the individual databases. Thus, both Multibase and ADDS solve the multiple database problem with a virtual database that effectively integrates the individuals databases. Yet, our approach is different in several respects from the approaches taken in these other systems. To integrate the individual databases, Multibase and ADDS employ a special data definition language (DDL), in which the global schema and its relationships to the individual schemas are specified. Here, we develop a set of *schema restructuring operators*. To integrate individual databases, an appropriate sequence of these operators is applied to their schemas. In addition, both Multibase and ADDS are expected to yield substantial software systems, and in both systems integration will require expert programming. Therefore, these systems will probably be suitable for stable environments and long-range applications. In contradistinction, our approach here is to design a compact integration tool that will be a simple extension of current database systems, and will allow the creation of quick global views, in an interactive schema editing process. Partial results of our research were reported in [7] and [8].

We have already mentioned the solution of physical consolidation of multiple databases. The topic of physical restructuring of files and databases (also called data translation, or data conversion) received much attention in the 1970's, and several systems to translate data and convert utility programs were developed (for example, [9]-[14]). Although these systems perform physical restructuring, they employ translation procedures and conversion operators that are often relevant to our problem here.

A common database design technique is to obtain first individual views for the various applications, and then apply an integration procedure that will combine these partial views into a single global schema (for example, [15]-[18]). While these procedures do not address the issues of actual integration (for example, data consistency, query translation), their schema synthesis procedures are also relevant to our problem here.

Obviously, external views [19] are one type of virtual databases: rather than integrate a number of databases, they transform a single one. Therefore, the same tools and techniques developed for schema integration could be used for generating external views. In some sense, construction of a superview is the inverse process of construction of external views: given two or more logical schemas, a superview is a larger schema, that has the given schemas as external views.

B. Overview of this Paper

The next two sections are devoted to the abstract model. Section II defines the database model (and the access operators), and Section III defines the integration language. An example superview is constructed in Section IV, which also describes how the mapping from the superview into the individual databases is derived. Section V describes and demonstrates the query translation process. Section VI concludes with a brief summary and discussion of remaining issues.

II. An Abstract Model of Databases

At the basis of the database model used here is a functional approach, which has received considerable attention since first described by Sibley and Kerschberg [20] (for example, [21]-[23]). While the details of these functional models may differ, they all employ the notions of data domains and attribute functions: domains are sets of data values, functions assign the values of one domain to values of another domain as their attributes. The model defined here distinguishes between two types of functions: attribute and generalization. These correspond to the aggregation and generalization relationships proposed by Smith and Smith [24], and incorporated into the Semantic Data Model by Hammer and Mcleod [25]. Generalization relationships become necessary, when two independent schemas are to be considered together. We see several advantages in the functional approach; in particular, it overcomes some of the acknowledged limitations of the relational model and it provides a formal framework in which both the relational model and the network model may be subsumed. A brief description follows; for further details see [26].

A. The Database

A *database* is a collection $\mathfrak{D}$ of named *classes* such that

1) Two relations **att** and **gen** are defined on $\mathfrak{D}$. Their intersection is empty, and their union has irreflexive transitive closure.

2) Each class $S \in \mathfrak{D}$ has a *domain* $\mathbf{dom}(S)$ of values.

3) Each class $S \in \mathfrak{D}$ has a *type*: $\mathbf{type}(S) = \{T \mid T \mathbf{\ att\ } S\}$.

4) Each class $S \in \mathfrak{D}$ has a *key*, which is a subset of its type: $K \subseteq \mathbf{type}(S)$, $K \mathbf{\ key\ } S$.

5) For every two classes $S, T \in \mathfrak{D}$ such that $T \mathbf{\ att\ } S$, there is a function

$$f_{ST}: \mathbf{dom}(S) \rightarrow \mathbf{dom}(T).$$

6) For every two classes $S, T \in \mathfrak{D}$ such that $T \mathbf{\ gen\ } S$, there is an injection

$$i_{ST}: \mathbf{dom}(S) \rightarrow \mathbf{dom}(T).$$

7) For every two classes $S, T \in \mathfrak{D}$:

$$S \mathbf{\ att\ } T,\ T \mathbf{\ gen\ } R \Rightarrow S \mathbf{\ att\ } R,\ f_{RS} = i_{RT} \circ f_{TS}$$

$$S \mathbf{\ gen\ } T,\ T \mathbf{\ gen\ } R \Rightarrow S \mathbf{\ gen\ } R,\ i_{RS} = i_{RT} \circ i_{TS}$$

8) If $(T_1, \cdots, T_k) \mathbf{\ key\ } S$ and $f_{ST_i}: S \rightarrow T_i\ (i = 1, \cdots, k)$ are the functions that support that **att** relationships, then

$$\begin{cases} f: \mathbf{dom}(S) \rightarrow \mathbf{dom}(T_1) \times \cdots \times \mathbf{dom}(T_k) \\ f(x) = (f_{ST_1}(x), \cdots, f_{ST_k}(x)) \end{cases}$$

is an injection.

9) For every two classes $S, T \in \mathfrak{D}$:

$$K \mathbf{\ key\ } S,\ S \mathbf{\ gen\ } T \Rightarrow K \mathbf{\ key\ } T$$

The first statement establishes among the classes of $\mathfrak{D}$ an attribute relationship, by which one class becomes an attribute of another class, and a generalization relationship, by which one class becomes a generalization of another class. It requires that the same class if not both an attribute and a generalization of another class, and that there is no chain of related classes (by either **att** or **gen**), that begins and ends in the same class. The next three statements associate with each class of $\mathfrak{D}$ a domain, a type, and a key. The type is defined as the set of all the "outgoing" attributes; the key is a designated subset of the type. Classes that do not have any attributes are called *primitive* classes. Thus, primitive classes have empty types.

The fifth and the sixth statements establish the actual relationships between the values in the domains of related classes: the **att** relationships are supported with functions, and the **gen** relationships are supported with one-to-one functions. For example, if NAME and PERSON are two database classes such that NAME **att** PERSON, then there is a function from **dom**(PERSON) into **dom**(NAME), that assigns each person a name. And, if STUDENT is another database class such that PERSON **gen** STUDENT, then there is an injection from **dom**(STUDENT) into **dom**(PERSON), that identifies each student as a unique person.

The seventh statement establishes the *inheritance* of attributes over generalizations, and the *transitivity* of generalizations. As an example, consider again the relationships NAME **att** PERSON and PERSON **gen** STUDENT. The inheritance requirement assures that also NAME **att** STUDENT, and that the name of a student is the same as the name of the person, who is a generalization of this student. Similarly, consider the relationships PERSON **gen** STUDENT and STUDENT **gen** PHD-STUDENT. The transitivity requirement assures that also PERSON **gen** PHD-STUDENT, and that a Ph.D. student is generalized to the same person, whether by a direct generalization, or through an intermediate generalization class.

The last two statements establish properties of keys. For many applications it is necessary that every value in a domain is uniquely identifiable by a combination of its primitive attributes. This is especially important for the purpose of integrating two different databases, so that when their two populations are consolidated, identical values can be recognized as such. Keys also convey important semantic information about their classes. Two classes that are keyed on the same attributes, can be assumed to be populated with "comparable" values. We shall use this information in the schema operations that integrate classes from different databases. The eighth statement requires that keys are supported by injections. For examples, if the classes $(T_1, \cdots, T_k)$ are the key of S, then a combination $(x_1, \cdots, x_k)$ of values from these classes determines at most one value of S. Assuming that every person has a different Personal Identification Number, the class PIN is a possible key for PERSON. The ninth statement guarantees that a key of a given class is also a key of every class that is generalized by the given class. For example, if PIN **key** PERSON and PERSON **gen** STUDENT, then also PIN **key** STUDENT. Note that the function from STUDENT to PIN is the composition of two injections, and is therefore an injection.

Keys are not necessarily simple (constituting a single class), or primitive. However, by composing keys, every value of a nonprimitive class can be identified by a combination of primitive values. Keys are not necessarily unique, and a class may have several different keys (however, for simplicity, our examples will feature only primary keys).

For example, consider the classes FACULTY, STUDENT, PERSON, PIN, NAME, RANK, and GPA with the **att** and **gen** relationships

PIN **att** FACULTY	NAME **att** STUDENT
PIN **att** STUDENT	NAME **att** PERSON
PIN **att** PERSON	GPA **att** STUDENT
RANK **att** FACULTY	PERSON **gen** FACULTY
NAME **att** FACULTY	PERSON **gen** STUDENT

In this example, PIN, NAME, RANK, and GPA are primitive classes. FACULTY, STUDENT, and PERSON are nonprimitive. The nonempty types are

$$\mathbf{type}(\text{FACULTY}) = (\text{PIN}, \text{NAME}, \text{RANK})$$
$$\mathbf{type}(\text{STUDENT}) = (\text{PIN}, \text{NAME}, \text{GPA})$$
$$\mathbf{type}(\text{PERSON}) = (\text{PIN}, \text{NAME})$$

Assume that persons are identified by a Personal Identification Number. The key relationships in this example are

PIN **key** PERSON

PIN **key** STUDENT

PIN **key** FACULTY

For brevity of notation, we shall use $S = (\underline{T}_1, \cdots, \underline{T}_k, T_{k+1}, \cdots, T_n)$ to describe a situation where $\mathbf{type}(S) = (T_1, \cdots, T_n)$ and $(T_1, \cdots, T_k)$ **key** S. Graphic representations of database schemas are very helpful. The following representation will be used in this paper. Each database class is represented by a node. If T **att** S, there is a directed edge from node S to node T: $S \rightarrow T$. If T **gen** S, there is a directed edge (with a double arrowhead) from node S to node T: $S \twoheadrightarrow T$. However, if T **gen** R and S **att** T, then S **att** R is suppressed in the graphic representation. Similarly, if S **gen** T and T **gen** R, then S **gen** R is suppressed (these are the inheritance and transitivity discussed above. For conciseness they will also be suppressed in all future specifications of databases). Graphs that represent databases do not have cycles or parallel edges. The graphic representation of the above example is given in Fig. 1.

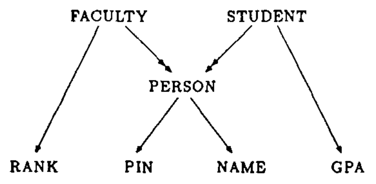

Fig. 1. Database on Faculty and Students

B. The Access Operators

We distinguish between a query language, in which retrieval requests are formulated, and access operators, that actually retrieve data from the database. For a query language, an algebraic language similar to DAPLEX [21], will be suitable. The following query, to list the names of students who received the grade A in the course CS100, gives a flavor of DAPLEX (the schema of this database is shown in Fig. 2):

```
for each ENROLLMENT
    such that COURSE(ENROLLMENT) = 'CS100'
    and GRADE(ENROLLMENT) = 'A'
print NAME(STUDENT(ENROLLMENT)).
```

We assume that such a query language is implemented with only a small set of primitive access operators. The set of access operators considered here consists of three operators. Given the name of any class, a *domain* operator simply returns the values in the domain of this class. Let T **att** S. Given a value $s \in \mathbf{dom}(S)$, a *function* operator returns the value $t \in \mathbf{dom}(T)$, that is assigned to the value s for the attribute T. Given a value $t \in \mathbf{dom}(T)$, an *inverse* operator returns all the values $\{s_1, \cdots, s_n\}$ in $\mathbf{dom}(S)$, that have t as their value for attribute T. The notation for the access operations is as follows:

domain:	$\{S\} = \mathbf{dom}(S)$
function:	$T(S = s) = f_{ST}(s)$
inverse:	$\{S(T = t)\} = f_{ST}^{-1}(t)$

The previous DAPLEX query can now be translated into the following Algol-like procedure, that accesses the database with *domain* and *function* operators,

```
for each x in {ENROLLMENT} do
begin
    if COURSE(ENROLLMENT = x) = 'CS100'
    and GRADE(ENROLLMENT = x) = 'A'
    then do
    begin
        y: = STUDENT(ENROLLMENT = x);
        print NAME(STUDENT = y)
    end
end.
```

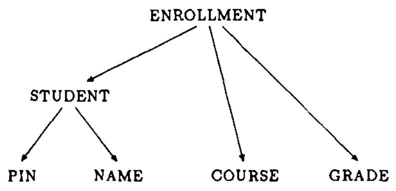

Fig. 2. Database on Enrollments

Or by using *inverse* operators:

```
for each x in {ENROLLMENT(COURSE='CS100')}
intersect {ENROLLMENT(GRAE='A')} do
begin
    y: =STUDENT(ENROLLMENT=x);
    print NAME(STUDENT=y)
    end.
```

III. The Integration Language

In this section we describe a small set of operators that integrate or modify database schemas. With each operator there is an associated set of constraints that must be satisfied by the values that populate the individual schemas. In the current implementation these constraints are checked only upon interrogation. We begin by introducing three primitive operators (**meet**, **join**, and **fold**), that manipulate the generalization hierarchy, a renaming operator (**rename**) for changing the names of classes, and two composite operators (**combine** and **connect**) that can be implemented with primitive operators.[2]

A. Meet

The **meet** operator produces a common generalization of two classes. A common generalization is possible only when the two classes have a common key. As an example, consider again the classes FACULTY = (PIN, NAME, RANK) and STUDENT = (PIN, NAME, GPA). The **meet** of FACULTY and STUDENT is the class PERSON = (PIN, NAME). Its domain includes all the values that are either in FACULTY or in STUDENT. This operation is based on the existence of a common key PIN.

Formally, assume that S and T are nonprimitive classes not related by **gen.** Assume there exists $K \subseteq \mathbf{type}(S) \cap \mathbf{type}(T)$ that maintains K **key** S and K **key** T. The operator **meet** S **and** T **into** U adds a new class U, the **meet** of S and T, and the relationships U **gen** S, U **gen** T, and R_i **att** U ($i = 1, \cdots, n$). The type of U is therefore given by $\mathbf{type}(U) = \mathbf{type}(S) \cap \mathbf{type}(T)$. The new class is populated with the union of the domains of S and T: $\mathbf{dom}(U) = \mathbf{dom}(S) \cup \mathbf{dom}(T)$. A graphic representation of **meet** is shown in Fig. 3. In this figure, the common attributes are represented by $R_1, \cdots, R_n$. The attributes that distinguish S and T are represented by $P_1, \cdots, P_l$ and $Q_1, \cdots, Q_m$, respectively. The injections from $\mathbf{dom}(S)$ and $\mathbf{dom}(T)$ into $\mathbf{dom}(U)$ are defined as identities. The functions from $\mathbf{dom}(U)$ into the domains of $R_1, \cdots, R_n$ are defined to preserve inheritance (i.e., values of both $\mathbf{dom}(S)$ and $\mathbf{dom}(T)$ are mapped into the same values in $\mathbf{dom}(R_i)$, either directly or through $\mathbf{dom}(U)$). The latter functions require a *consistency constraint*: values in $\mathbf{dom}(S)$ or $\mathbf{dom}(T)$ that have the same key, must agree over their shared attributes. Formally, denote by $f_1, \cdots, f_n$ and $g_1, \cdots, g_n$ the attribute functions from S and T, respectively, into $R_1, \cdots, R_n$. Let $K = (R_1, \cdots, R_k)$. Define functions f and g as follows:

$$\begin{cases} f: \mathbf{dom}(S) \rightarrow \mathbf{dom}(R_1) \times \cdots \times \mathbf{dom}(R_k) \\ f(x) = (f_1(x), \cdots, f_k(x)) \end{cases}$$

$$\begin{cases} g: \mathbf{dom}(T) \rightarrow \mathbf{dom}(R_1) \times \cdots \times \mathbf{dom}(R_k) \\ g(x) = (g_1(x), \cdots, g_k(x)) \end{cases}$$

Then, $\forall y \in f(\mathbf{dom}(S)) \cap g(\mathbf{dom}(T))$:

$$f_i(f^{-1}(y)) = g_i(g^{-1}(y)) \; (i = k+1, \cdots, n)$$

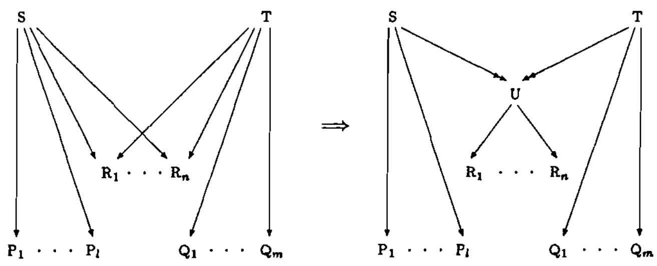

Fig. 3. The **meet** Operator

B. Join

meet creates a class, whose type is the intersection of both types, and whose domain is the union of both domain. Another class that may be created under the same circumstances is the dual class, whose type is the *union* of both types, and whose domain is the *intersection* of both domains. As an example, consider again the classes FACULTY and STUDENT. Their **meet** is the class PERSON, whose domain includes all those that are *either* FACULTY or STUDENT. The **join** of STUDENT and FACULTY is the class ASSISTANT = (PIN, NAME, GPA, RANK). Its domain includes all those that are *both* FACULTY or STUDENT. While PERSON generalizes both FACULTY and STUDENT, ASSISTANT is generalized by both FACULTY and STUDENT.

Formally, assume S and T maintain the same conditions as before. The operator **join** S **and** T **into** U adds a new class U, the **join** of S and T, and the relationships S **gen** U, T **gen** U, R_i **att** U ($i = 1, \cdots, n$), P_i **att** U ($i = 1, \cdots, l$) and Q_i **att** U ($i = 1, \cdots, m$). The type of U is therefore given by $\mathbf{type}(U) = \mathbf{type}(S) \cup \mathbf{type}(T)$. The domain of U is $\mathbf{dom}(U) = \mathbf{dom}(S) \cap \mathbf{dom}(T)$. A graphic representation of **join** is shown in Fig. 4. The

[2]**meet** and **join** are named for their Boolean algebra counterparts. In particular, **join** should not be confused with the similarity named relational algebra operator.

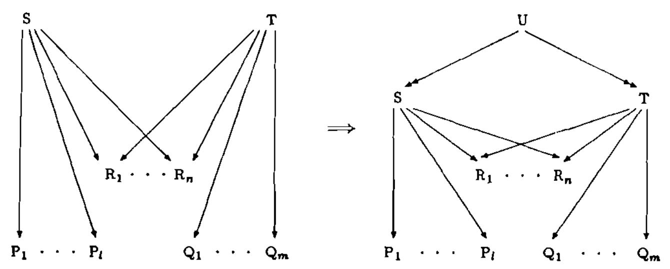

Fig. 4. The **join** Operator

injections from **dom**(U) into **dom**(S) and **dom**(T) are defined as identities. The functions from **dom**(U) into the domains of $R_1, \cdots, R_n$, $P_1, \cdots, \cdots, P_l$, and $Q_1, \cdots, Q_m$ are defined to preserve inheritance. Again, the same consistency constraint is required.

C. Fold

meet and **join** add a generalization class. **fold** allows a generalization class to absorb a more specific class. With **fold,** the class STUDENT may be absorbed by the more general class PERSON, with the distinguishing STUDENT attributes carried over to PERSON. Nonstudents are assigned a special *null* value, called [*not-applicable*], for these attributes.[3]

Formally, assume S and T are two nonprimitive classes such that T **gen** S. The operator **fold** S **into** T removes the class S and replaces it with T in all relationships. A graphic representation of **fold** is shown in Fig. 5. Functions and injections that had **dom**(S) as their domain are modified to have **dom**(T) as their new domain, using the previous injection from **dom**(S) into **dom**(T) (and [*not-applicable*] for values in **dom**(T), but not in the image of this injection). Using the same injection, functions and injections that had **dom**(S) as their range, are modified to have **dom**(T) as their new range.

D. Rename

In the course of schema integration, it will sometimes be necessary to *rename* a class. The operator **rename** S **to** T assigns the new name T to the class S.

E. Combine and Connect

With **meet**, the similarity between two related classes may be expressed. When two classes have *identical* types, complete overlap may be achieved with a combination of **meet** and two **folds.** This sequence will be abbreviated **combine.** Formally, assume S and T are nonprimitive classes not related by **gen,** and **type**(S) = **type**(T). The composite operator **combine** S **and** T **into** U is defined as follows:

meet S **and** T **into** U
fold S **into** U
fold T **into** U

When the type of one class is *contained* in the type of

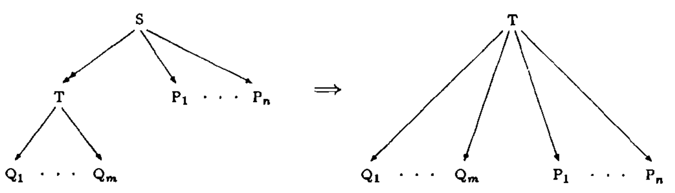

Fig. 5. The **fold** Operator

another class, a **meet** followed by a **fold** can make the former a generalization of the latter. This sequence will be abbreviated **connect.** Formally, assume S and T are nonprimitive classes not related by **gen, type**(T) $\subseteq$ **type**(S). The composite operator **connect** S **to** T is defined as follows:

meet S **and** T **into** U
fold T **into** U
rename U **to** T

Note that, while **combine** and **connect** were described as composite operators, in future discussions they will be treated as primitive operators. In addition, the operator **combine** is extended to apply also to two *primitive* classes. Every two primitive classes S and T may be combined into a new primitive class U. The domain of U is the union of the domains of S and T, and functions that previously had either domain for their range, are modified to have the new domain for their range.

meet, join, fold, combine, and **connect** are operators that manipulate the *generalization* hierarchy of databases. The next two operators, **aggregate** and **telescope,** allow modifications to the *attribute* hierarchy.

F. Aggregate

The operator **aggregate** creates an intermediate class, between a given class and a designated subset of its attributes. Formally, assume S is a nonprimitive class, **type**(S) = $(T_1, \cdots, T_m, T_{m+1}, \cdots, T_n)$. The operator **aggregate**$(T_1, \cdots, T_m)$ **of** S **into** T adds a new class T and the relationship T **att** S. Also, every relationship T_i **att** S is replaced with T_i **att** T $(i = 1, \cdots, m)$. A graphic representation of **aggregate** is shown in Fig. 6. The domain of T is populated with new values, that are tuples of values from the aggregated domains: **dom**(T) = $\{(f_{ST_1}, \cdots, f_{ST_m}) \mid s \in S\}$. The function that supports the relationship between S and T is defined by $f_{ST}(x) = (f_{ST_1}(x), \cdots, f_{ST_m}(x))$. The functions from **dom**(T) onto **dom**(T_i) $(i = 1, \cdots, m)$ are simple projections.

G. Telescope

While **aggregate** extends the attribute hierarchy, **telescope** performs the inverse: it removes a class by assigning its attributes directly to its ancestor class. Formally, assume T is a nonprimitive class, **type**(T) = $(T_1, \cdots, T_n)$, and is an attribute of only one class S. The operator **telescope** T **into** S removes the class T and the relationship T **att** S, replacing the relationships T_i **att** T with T_i **att** S $(i = 1, \cdots, n)$. A graphic representation of **telescope** is shown in Fig. 7. The functions that sup-

[3][*not-applicable*], and other null values, are discussed further in Section V.

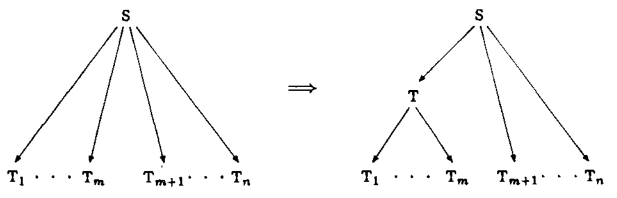

Fig. 6. The **aggregate** Operator

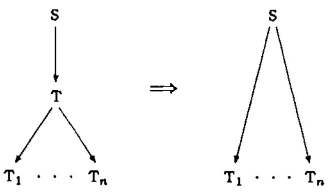

Fig. 7. The **telescope** Operator

port the new attribute relationships are simple compositions: $f_{ST_i}(x) = f_{TT_i}(f_{ST}(x))$ $(i = 1, \cdots, n)$.

With **aggregate** and **telescope,** an attribute may be *relocated* on the schema. Consider FACULTY = (NAME, RANK, DEPARTMENT) and DEPARTMENT = (D-NAME, COLLEGE, ADDRESS). By telescoping DEPARTMENT into FACULTY, and then aggregating D-NAME and COLLEGE back into DEPARTMENT, the attribute ADDRESS is relocated from DEPARTMENT to FACULTY.

aggregate may be used to *normalize* the schema into a form, in which the non-key attributes of each class are fully dependent on the key. If a class exists with some attributes that are dependent on a subset of its key, these attributes, together with the subkey, are aggregated into an interim class. For example, consider the class ENROLLMENT = (STUDENT-NO, STUDENT-NAME, COURSE-NO, COURSE-TITLE, GRADE). The key of ENROLLMENT is both STUDENT-NO and COURSE-NO, but only GRADE depends on both; STUDENT-NAME depends only on STUDENT-NO, and COURSE-TITLE depends only on COURSE-NO. Using **aggregate** twice, the following normal form schema may be obtained: STUDENT = (STUDENT-NO, STUDENT-NAME), COURSE = (COURSE-NO, COURSE-TITLE), and ENROLLMENT = (STUDENT, COURSE, GRADE).

As a third example, consider the classes ACCOUNT = (ACC-NO, NAME, BALANCE) and TRANSACTION = (TRANS-NO, ACC-NO, AMOUNT). By aggregating ACC-NO into an interim class under TRANSACTION, which is then combined with ACCOUNT (both have a common key ACC-NO), a *re-traction* of TRANSACTION to ACCOUNT is obtained: AC-COUNT is left unchanged, but TRANSACTION is modified to TRANSACTION = (TRANS-NO, ACCOUNT, AMOUNT), which is more accurate, since each transaction has its own account, rather than its own account number. Like normalization, retraction may be applied wherever possible to improve the representation. (Note that if ACC-NO were a key of both ACCOUNT and TRANSACTION, a **meet** of these two classes would have been more appropriate, since there is evidence that these classes have strong semantic similarity.)

All the previous operators merely transformed given structures to comparable structures. The last two operators (**add** and **delete**) are different, in that they allow current structures to be extended or reduced.

H. Add

In general, the addition of a new class (with its attribute relationships) to an existing database should be considered an *augmentation* of this database by another database, and not a *restructuring* operation. In many cases, however, a given class has an attribute which is implied, but not specified. For example, a class CAR in the database of a Ford car dealer may not include the attribute MAKE. Adding this attribute (with a single value 'Ford' for all cars) does not qualify as augmentation by another database, but will prove important when databases of different car dealers have to be integrated. Formally, assume S is a nonprimitive class. The operator **add** $P(x)$ **to** S adds a primitive class P with a singleton domain $\{x\}$, and a relationship P **att** S with a *constant function* from **dom**(S) onto **dom**(P).

Whenever identical structures from two databases are combined, loss of information may result. Consider two library databases, both with a class BOOK = (BOOK-NO, TITLE, AUTHOR). If these classes are combined, the information on where each book is shelved would be lost. Using **add,** this implied knowledge can be added to each class as a new attribute LIBRARY, yielding: BOOK = (BOOK-NO, LIBRARY, TITLE, AUTHOR). These classes can now be combined safely. Note that the attribute LIBRARY was also added to the key of BOOK (this is done with a special declarative statement **add** T **to key of** S). Without this addition, the two classes would not comply with the integrity constraint required by **meet.**

I. Delete

To remove portions of the database which are not relevant to the application, a **delete** operator is defined. Assume S is a nonprimitive class, and T **att** S, but $T \notin$ **key**(S). The operator **delete** T **from** S removes the relationship T **att** S. If T is no longer an attribute of any other class, it too is removed together with all its out-going relationships. Each of its attibutes is in turn examined, to see if it is still an attribute of any other class, and so on. If T is part of the key of S, then its deletion would have serious semantic implications. In particular, values in the domain of the new class S, that were previously differentiated only by their key value, could no longer be identified. For example, deleting COURSE-NO from ENROLLMENT = (COURSE-NO, STUDENT-NO, GRADE) creates (STUDENT-NO, GRADE), a class whose meaning is unclear.

In general, **aggregate, telescope, add,** and **delete** may be used to iron-out structural differences between two given schemas, so that better overlapping may be achieved.

IV. Constructing Virtual Databases

A virtual database is constructed by editing the schemas of individual databases (actual or virtual), with the oper-

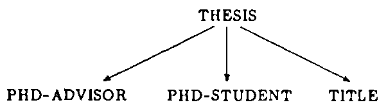

Fig. 8. Database on PhD Theses

ators described in the previous section. When this process terminates, two products are available: a superview (the schema of the virtual database), and a mapping of the superview into the initial schemas (the "data" of the virtual database). The derivation of the superview and the mapping are described in this section.

A. Constructing the Superview

The purpose of the integration process is to identify similar structures in the individual schemas. It is convenient to think of the initial schemas as a single (disconnected) schema; each integration step then *restructures* the current version of the schema into the next version.

The integration process requires knowledge of the meaning of the different classes. For example, two primitive classes from two databases may be combined, only if they model the same real world concept. Thus, two classes describing Personal Identification Numbers may probably be combined, but the employee number in two different organizations may reflect two independent sequencings without any global meaning. The latter case does not create problems, unless these classes participate in keys; in this case, identical values (i.e., the same employee in both organizations) would not be recognized as such. We illustrate the integration technique with an example.

Assume the previous database on faculty and students, as described in Fig. 1, and a second database, that describes the different Ph.D. theses currently under development. Each thesis is a combination of a Ph.D. student, a faculty advisor (both identified by PIN), and a title (that identifies the thesis). This database has four classes and three attribute relationships, as follows (a graphic representation is shown in Fig. 8):

TITLE **att** THESIS
PHD-ADVISOR **att** THESIS
PHD-ADVISOR **att** THESIS

The integration of these databases will be done in three steps. The first group of statements is aimed at restructuring the second database to appear more like the first database (the resulting schema is shown in Fig. 9):

1) **aggregate** (PHD-ADVISOR) **of** THESIS **into** *T*
2) **rename** PHD-ADVISOR **to** PIN1
3) **rename** *T* **to** PHD-ADVISOR
4) **aggregate** (PHD-STUDENT) **of** THESIS **into** *T*
5) **rename** PHD-STUDENT **to** PIN2
6) **rename** *T* **to** PHD-STUDENT
7) **combine** PIN1 **and** PIN2 **into** PIN'
8) **meet** PHD-ADVISOR **and** PHD-STUDENT **into** PERSON'
9) **add** NAME'([*not-available*]) **to** PERSON'

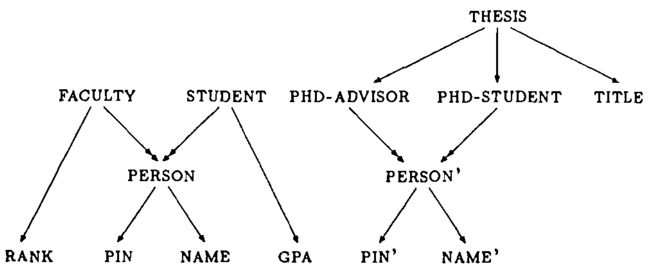

Fig. 9. After the First Group of Statements

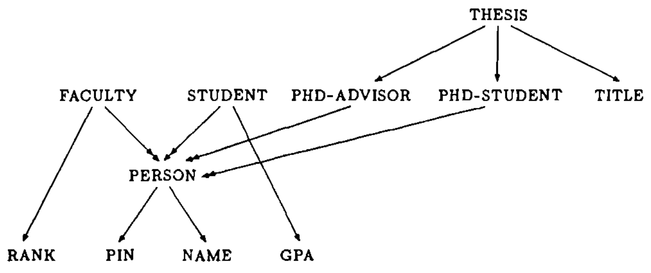

Fig. 10. After the Second Group of Statements

The value [*not-available*] is a null value that is used whenever an attribute is applicable, but the particular value that applies is not known.[4] Next, we link the two databases at the primitive level, and then combine their two PERSON classes (the resulting schema is shown in Fig. 10):

1) **combine** PIN **and** PIN' **into** PIN
2) **combine** NAME **and** NAME' **into** NAME
3) **combine** PERSON **and** PERSON' **into** PERSON

Finally, we insert FACULTY and STUDENT as generalizations of PHD-ADVISOR and PHD-STUDENT, respectively (the final schema is shown in Fig. 11):

1) **connect** PHD-ADVISOR **to** FACULTY
2) **connect** PHD-ADVISOR **to** STUDENT

Note that, as PHD-STUDENT does not have any attributes in addition to those of STUDENT, it is possible to add to each of the constant attribute LEVEL, with [*not-available*] for STUDENT and 'phd' for PHD-STUDENT, and them fold them into one class (similarly for PHD-ADVISOR and FACULTY).

B. Constructing the Mapping

The mapping of the superview into the initial databases may be constructed in various ways. Our method is to associate with each class of the final superview an expression that denotes the *origins* of this class, in terms of classes of the initial databases. This expression is obtained incrementally during the integration process, as each operator updates the expressions associated with the classes it modifies. The expressions associated with the classes of the initial databases are the names of the classes themselves. The expressions associated with the classes of the final superview constitute the mapping.

[4]Null values are discussed in Section V.

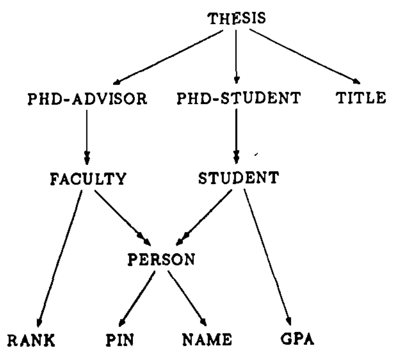

Fig. 11. The Final Superview

Assume an integration process in which k integration operators had already been applied. Let R, S, and T be classes with associated expressions α, β, and γ, respectively. Assume the next integration step is **meet** S **and** T **into** Q. The expression associated with the new class Q is $(\alpha \wedge \beta)_{k+1}$. Assume the next integration step is **delete** R **from** Q. The expression associated with Q is updated to $((\alpha \wedge \beta)_{k+1} - \gamma)_{k+2}$. And so on. The complete notation is shown below.

operation	updated class	associated expression
meet S **and** T **into** Q	Q	$\alpha \wedge \beta$
join S **and** T **into** Q	Q	$\alpha \vee \beta$
fold S **into** T	T	$\beta \leftarrow \alpha$
rename S **to** T	T	α
combine S **and** T **into** Q	Q	$\alpha \circ \beta$
connect S **to** T	Q	$\alpha \downarrow \beta$
aggregate $(T_1, \ldots, T_n)$ **of** S **into** T	T	$(\beta_1, \ldots, \beta_n) \cdot \alpha$
telescope T **into** S	S	$\alpha : \beta$
add $T(x)$ **to** S	S	$\alpha + T_x$
delete T **from** S	S	$\alpha - \beta$

As an example, assume the classes SUPPLIER and CUSTOMER are first generalized to ASSOCIATE, and then the attribute TEL-NO is deleted from ASSOCIATE. The expression associated with the final class ASSOCIATE is $((\text{SUPPLIER} \wedge \text{CUSTOMER}))_1 - \text{TEL-NO})_2$.

V. Query Translation

Our translation strategy is as follows. When an access operator is submitted to a superview, it is translated over each of the integration operators that were involved in the derivation of the superview classes that it addresses. The order of the translations is the reverse of the order in which the integration operators were applied, as indicated by the expressions associated with these classes. This process yields access operators that can be submitted to the actual databases. The answers are then passed back in the reverse direction, until an answer to the original access operator is formed. This recursive process of query decomposition and answer recomposition involves only *primitive translations*: translations of access operators over single integration operators.

A. Combining Answers to Access Operators

Each primitive translation decomposes a given access operator into one or more access operators; eventually, their answers will be recomposed to provide an answer to the given access operator. Answers to *domain* operators will usually be combined with standard set operations. The recomposition of answers to *function* and *inverse* operators is discussed below.

When a *function* operator $T(S = s)$ is submitted to a database, several situations may occur:

1) T is not an attribute of S.

2) T is an attribute of S, but s is not in $\mathbf{dom}(S)$.

3) T is an attribute of S, and s is in $\mathbf{dom}(S)$, but, for some reason, the value $f_{ST}(s)$ is undefined.

4) T is an attribute of S, s is in $\mathbf{dom}(S)$, and the value $f_{ST}(s)$ is defined.

Thus, each successive situation is more "successful" than the preceding one. To handle these situations, we extend the definition of the *function* operator, to include three kinds of *null* values, called [*not-applicable*], [*not-found*], and [*not-available*], as follows:

$$T(S = s) = \begin{cases} [\textit{not-applicable}], & \text{if } T \notin \mathbf{type}(S) \\ [\textit{not-found}], & \text{if } s \notin \mathbf{dom}(S) \\ [\textit{not-available}], & \text{if } f_{ST}(s) \text{ is undefined} \\ f_{ST}(s), & \text{otherwise.} \end{cases}$$

Recall that the null values [*not-applicable*] and [*not-available*] were introduced earlier as values of attribute functions, following certain restructuring operations. Therefore, $f_{ST}(s)$ may evaluate to a null value. Also, if s is a null value, we define $T(S = s) = s$.

Assume now a simple case when this operator is submitted to two independent databases. Each answer could be any of the above four alternatives. When the answers are identical, it is obvious how they should be combined. When the answers are different, we adopt the answer that is more "successful." As an example, assume a query regarding the AGE of a certain PERSON is submitted to two databases. When one answer is an actual AGE value, and the other answer is [*not-found*], then the combined answer is the AGE value; [*not-available*] and [*not-found*] are combined to [*not-available*], and so on. Finally, there is the case when the two databases return two non-null AGE values that are different. This is a case of inconsistency, and a fourth null value, [*not-consistent*], is returned. Assume a and b are two different values. The definition of this answer-combining operation, called *best-value* and denoted $\vee$, is tabulated below.

$\vee$	$[n-ap]$	$[n-fd]$	$[n-av]$	$[n-co]$	a	b
$[n-ap]$	$[n-ap]$	$[n-fd]$	$[n-av]$	$[n-co]$	a	b
$[n-fd]$	$[n-fd]$	$[n-fd]$	$[n-av]$	$[n-co]$	a	b
$[n-av]$	$[n-av]$	$[n-av]$	$[n-av]$	$[n-co]$	a	b
$[n-co]$	$[n-co]$	$[n-co]$	$[n-co]$	$[n-co]$	$[n-co]$	$[n-co]$
a	a	a	a	$[n-co]$	a	$[n-co]$
b	b	b	b	$[n-co]$	$[n-co]$	b

Consider now an *inverse* operator $\{S(T = t)\}$ is submitted to a database. The are three possible situations:

1) T is not an attribute of S.
2) T is an attribute of S, but t is not in **dom**(T).
3) T is attribute of S, and t is in **dom**(T).

Again, each successive situation is more "successful" than the preceding one. We extend the definition of the *inverse* operator, as follows:

$$\{S(T = t)\} = \begin{cases} [\textit{not-applicable}], & \text{if } T \notin \mathbf{type}(S) \\ [\textit{not-found}], & \text{if } t \notin \mathbf{dom}(T) \\ f_{ST}^{-1}(t), & \text{otherwise.} \end{cases}$$

Note that $f_{ST}^{-1}(t)$ may be the empty set (the empty set is not a null value). If t is a null value, we define $\{S(T = t)\} = t$.

Answers to *inverse* operators are combined as follows. If both answers are sets, then the answer is their union. Otherwise, the answer that is more "successful" is adopted. Assume A and B are two different sets of values. The definition of this answer-combining operation, called *best-set* and denoted $\sqcup$, is tabulated below.

$\sqcup$	$[n-ap]$	$[n-fd]$	A	B
$[n-ap]$	$[n-ap]$	$[n-fd]$	A	B
$[n-fd]$	$[n-fd]$	$[n-fd]$	A	B
A	A	A	A	$A \cup B$
B	B	B	$A \cup B$	B

B. Primitive Translation Rules

We have described three access operators and ten integration operators. Consequently, there are thirty different cases (*rules*) of primitive translation. We shall describe only some of them here. For a complete analysis see [26].

To explain the primitive translations, we assume that a database is given, and is restructured with a single integration step into a new database. In translating access operators that are submitted to the new database, only operators that involve classes that were affected by the restructuring need to be considered; other operators may be passed to the old database without modification.

We begin by describing the translations necessary after the restructuring operation **meet** S **and** T **into** Q. Let R be any attribute of Q. In the new database, three access operations may need translation: a *domain* operator on Q, a *function* operator from Q into R, and an *inverse* operator from R into Q. The domain of Q is constructed from the union of the domains of S and T. To translate $R(Q = x)$, we submit $R(S = x)$ and $R(T = x)$ to the old database. Recall that each answer could be either an actual value, or one of the null values: [*not-applicable*], [*not-found*], [*not-available*]. If the answers are different actual values, they are combined to [*not-consistent*]; otherwise, they are combined to the more successful answer. To translate $\{Q(R = y)\}$, we submit $\{S(R = y)\}$ and $\{T(R = y)\}$. If both answers are sets, they are combined to their union; otherwise, they are combined to the more successful answer. These translations are summarized in the following rules:

$$\{Q\} \equiv \{S\} \cup \{T\}$$

$$R(Q = x) \equiv R(S = x) \vee R(T = x)$$
$$\{Q(R = y)\} \equiv \{S(R = y)\} \sqcup \{T(R = y)\}$$

For example, assume PERSON is the **meet** of STUDENT and FACULTY. The answer to the *domain* operator {PERSON} is the union of the answers to {STUDENT} and {FACULTY}; the answer to the *function* operator NAME(PERSON $= x$) is the *best-value* of the answers to NAME(STUDENT $= x$) and NAME(FACULTY $= x$); and the answer to *inverse* operator {PERSON(NAME $= y$)} is the *best-set* of the answers to {STUDENT(NAME $= y$)} and {FACULTY(NAME $= y$)}.

The translations required after **combine** S **and** T **into** Q are very similar to those described for **meet**, except that here it is possible for the new class Q also to be the *range* of an attribute function. Let P be any class in the new database, such that Q **att** P. The *function* operator $Q(P = x)$ and the *inverse* operator $\{P(Q = y)\}$ are translated as follows:

$$Q(P = x) \equiv S(P = x) \vee T(P = x)$$
$$\{P(Q = y)\} \equiv \{P(S = y)\} \sqcup \{P(T = y)\}$$

As an example, assume the class BOOK-NO-1 (which is an attribute of the class BOOK-1) and the class BOOK-NO-2 (which is an attribute of the class BOOK-2) are combined into the class BOOK-NO (which is now an attribute of both BOOK-1 and BOOK-2). The *inverse* operator {BOOK-1(BOOK-NO $= y$)} is answered with {BOOK-1(BOOK-NO-1 $= y$)} $\sqcup$ {BOOK-1(BOOK-NO-2 $= y$)}.

Next, we describe the translations necessary after the restructuring operation **telescope** T **into** S. For clarity, let Q denote the class S in the new database. Let P and R be any classes in the new database, such that Q **att** P and R **att** Q. In the new database, five access operations may need translation: a *domain* operator on Q, a *function* operator from P into Q, an *inverse* operator from Q into P, a *function* operator from Q into R, and an *inverse* operator from R into Q. Because the domain of a class does not change after **telescope**, the first three operators are translated into similar operators with the old class S. The translation of the other two operators depends on whether R was a direct attribute of S. If so, these operators are translated into similar operators with the old class S. Otherwise, if R was an attribute of T, these operators are translated in two steps. The *function* operator $R(Q = x)$ is translated as follows:

$$z := T(S = x)$$
$$R(T = z)$$

And the *inverse* operator $\{Q(R = y)\}$:

$$Z := \{T(R = y)\}$$
$$\sqcup_{z \in Z} \{Q(T = z)\}$$

These translations are abbreviated $R(T(S = x))$ and $\{S(T(R = y))\}$, respectively. Note that, instead of *testing* whether R was an attribute of S or an attribute of T, it is possible to attempt *both* translations. For example, to translate the *function* operator $R(Q = x)$, both $R(S =$

x) and $R(T(S = x))$ are submitted. As one of these operators must evaluate to [*not-applicable*], the other answer will be adopted. Similarly for the *inverse* operator. Hence the following translation rules:

$$R(Q = x) = R(S = x) \vee R(T(S = x))$$
$$\{Q(R = y)\} = \{R(S = y)\} \sqcup \{S(T(R = y))\}$$

As an example, consider ENROLLMENT = (STUDENT, COURSE, GRADE) and STUDENT = (PIN, NAME, GPA), and assume STUDENT is telescoped into ENROLLMENT, yielding the class ENROLLMENT'. To translate the *function* operator NAME(ENROLLMENT' = x), both NAME(ENROLLMENT = x) and NAME(STUDENT(ENROLLMENT' = x)) are submitted. As the former will return [*not-applicable*], the answer to the latter will be adopted.

Finally, we examine the translations necessary after **add** $T(t)$ **to** S. Again, let Q denote the class S in the new database, and let P and R be any classes in the new database, such that Q **att** P and R **att** Q. Again, five access operations may need translation: a *domain* operator on Q, a *function* operator from P into Q, an *inverse* operator from Q and P, a *function* operator from Q into R, and an *inverse* operator from R into Q. Again, because the domain of a class does not change after **add**, the first three operators are translated into similar operators with the old class S. The translation of the other two operators depends on whether R is the new attribute T. If not, these operators are translated into similar operators with the old class S. Otherwise, $R = T$, and

$$R(Q = x) \equiv \text{if } x \in \{S\} \text{ then } t \text{ else } [\textit{not-found}]$$
$$\{Q(R = t)\} \equiv \{S\}$$

C. The Global Translation Procedure

To translate an access operator that is submitted to a superview, the primitive translations rules are incorporated into a *global translation procedure*. To simpli the description of this procedure, we shall assume there is a separate translation procedure for each access operator. Each translation procedure consists of a *driving* routine and one *decomposition-recomposition* routine for each translation rule. In practice, many routines are identified.

For global translation, all the primitive translation rules are transcribed, to replace *names* of classes with the appropriate *expressions*. For example, Let α, β, and γ denote expressions. The primitive translation rules of **meet** are transcribed to

$$\{\alpha \wedge \beta\} \equiv \{\alpha\} \cup \{\beta\}$$
$$\gamma(\alpha \wedge \beta = x) \equiv \gamma(\alpha = x) \vee \gamma(\beta = x)$$
$$\{\alpha \wedge \beta(\gamma = y)\} \equiv \{\alpha(\gamma = y)\} \sqcup \{\beta(\gamma = y)\}$$

Similarly, the given access operator is transcribed to replace its classes with their associated expressions. An access operator will specify either one expression (a *domain* operator) or two expressions (a *function* or an *inverse* operator). If there are two expressions, the "later" expression would be the subject of the next translation.[5]

The process begins in the driving routine. It analyzes the subject expression in the access operator, identifies the translation rule that applies, and calls the appropriate decomposition-recomposition routine. The decomposition-recomposition routine determines the access operators necessary for the translation, calls the driving routine for each, and recomposes the answers to an answer that it finally returns to the driving routine. Therefore, global translation is a recursive process.

For example, consider a *function* operator $\alpha(\beta = x)$, where β is the subject expression, and assume that $\beta = \beta_1 : \beta_2$ (i.e., the class described by β was formed in a **telescope** operation). The driving routine calls the decomposition-recomposition routine that handles this particular rule. The latter routine decomposes this operator into $\alpha(\beta_1 = x)$ and $\alpha(\beta_2(\beta_1 = x))$, and calls the driving routine for each of these two operators. When the results arrive, they are combined with *best-value*, and this answer is returned to the driving routine. If α were the subject expression in the given access operator, and, for example, $\alpha = \alpha_1 + \alpha_2$ (i.e., the class described by α was formed in an **add** operation, after the **telescope** operation), then one of the **add** decomposition-recomposition routines would have been called. The latter routine would have translated $\alpha_1 + \alpha_2(\beta = x)$ into $\alpha_1(\beta = x)$, and would have submitted it to the driving routine.

D. An Example

As an example of a translation process, assume two bibliographic databases. One database stores articles, as follows:

A-TITLE **att** ARTICLE

A-AUTHOR **att** ARTICLE

JOURNAL **att** ARTICLE

VOL-NO **att** ARTICLE

A-TITLE **key** ARTICLE

The other database stores books:

B-TITLE **att** BOOK

B-AUTHOR **att** BOOK

PUBLISHER **att** BOOK

YEAR **att** BOOK

B-TITLE **key** BOOK

Articles and books are both items of literature, each with a title and an author. To integrate these databases we 1)

[5]Recall that expressions reflect the integration steps that were applied, and the *positions* of these integration steps in the complete sequence of integration operations. Thus, given two expressions, it is possible to determine which was modified last.

combine the primitive classes A-TITLE and B-TITLE into TITLE, 2) combine the primitive classes A-AUTHOR and B-AUTHOR into AUTHOR, and 3) generalize BOOK and ARTICLE into a common class ITEM. The final superview is as follows (recall that inherited attribute and key relationships are omitted):

TITLE **att** ITEM

AUTHOR **att** ITEM

JOURNAL **att** ARTICLE

VOL-NO **att** ARTICLE

PUBLISHER **att** BOOK

YEAR **att** BOOK

TITLE **key** ITEM

ITEM **gen** ARTICLE

ITEM **gen** BOOK

The mapping of this superview into the actual databases is represented by the following expressions (primitive expressions are omitted):

ITEM:	$(\text{ARTICLE} \wedge \text{BOOK})_3$
TITLE:	$(\text{A-TITLE} \circ \text{B-TITLE})_1$
AUTHOR:	$(\text{A-AUTHOR} \circ \text{B-AUTHOR})_2$

Consider now a query against this virtual database about all the titles of items authored by one 'Charl A. Tan'. A procedure that executes this query is

```
for each x in {ITEM(AUTHOR='Charl A. Tan')} do
  print TITLE(ITEM=x).
```

First, the *inverse* operator {ITEM(AUTHOR = 'Charl A. Tan')} must be translated into access operators that will be submitted to the actual databases. As the expression associated with ITEM reflects a later integration step, it is handled first:

$$
\begin{aligned}
&\{\text{ITEM(AUTHOR = 'Charl A. Tan')}\}\\
=\;& \{\text{ARTICLE(AUTHOR = 'Charl A. Tan')}\}\\
& \sqcup\\
& \{\text{BOOK(AUTHOR = 'Charl A. Tan')}\}\\
=\;& \{\text{ARTICLE(A-AUTHOR = 'Charl A. Tan')}\}\\
& \sqcup\\
& \{\text{ARTICLE(B-AUTHOR = 'Charl A. Tan')}\}\\
& \sqcup\\
& \{\text{BOOK(A-AUTHOR = 'Charl A. Tan')}\}\\
& \sqcup\\
& \{\text{BOOK(B-AUTHOR = 'Charl A. Tan')}\}
\end{aligned}
$$

These four requests are submitted to the actual databases. The second and the third would return [*not-applicable*], which will be ignored by the respective *best-set* operation:

$$
\begin{aligned}
=\;& \{\text{ARTICLE(A-AUTHOR = 'Charl A. Tan')}\}\\
& \sqcup\\
& [\textit{not applicable}]\\
& \sqcup\\
& [\textit{not-applicable}]\\
& \sqcup\\
& \{\text{BOOK(B-AUTHOR = 'Charl A. Tan')}\}\\
=\;& \{\text{ARTICLE(A-AUTHOR = 'Charl A. Tan')}\}\\
& \sqcup\\
& \{\text{BOOK(B-AUTHOR = 'Charl A. Tan')}\}
\end{aligned}
$$

For each value x in the result, the *function* operator TITLE(ITEM = x) is now applied. This operator is translated as follows:

$$
\begin{aligned}
&\text{TITLE(ITEM} = x)\\
=\;& \text{TITLE(ARTICLE} = x)\\
& \vee\\
& \text{TITLE(BOOK} = x)\\
=\;& \text{A-TITLE(ARTICLE} = x) \vee \text{B-TITLE(ARTICLE} = x)\\
& \vee\\
& \text{A-TITLE(BOOK} = x) \vee \text{B-TITLE(BOOK} = x)\\
=\;& \text{A-TITLE(ARTICLE} = x) \vee [\textit{not-applicable}]\\
& \vee\\
& [\textit{not-applicable}] \vee \text{B-TITLE(BOOK} = x)\\
=\;& \text{A-TITLE(ARTICLE} = x)\\
& \vee\\
& \text{B-TITLE(BOOK} = x)
\end{aligned}
$$

Thus, the final answer is a list of titles of articles and books. Note that, if a certain manuscript appears as both an article and a book, it will be listed only once.

It important to realize that all nonprimitive database values are communicated outside their databases as *external representations* (i.e., structured keys). In our example, the values x (delivered by the individual databases in response to the first round of requests, and used by the superview in the second round of requests) are actually titles of articles and books.

VI. Conclusion

We have described a method for the effective integration of multiple databases, through the use of virtual databases. The schema of a virtual database is a *superview* which is obtained in a process of schema editing; the data of a virtual database is a *mapping* which is derived from the editing process. The database system employs this mapping in a query translation process, which is designed to make the existence of virtual databases transparent.

Our virtual databases store mappings instead of conventional descriptive data. Indeed, these mappings may be regarded as a *more general* form of data. Furthermore, one many define data as any process that produces descriptions, and databases as repositories of such *data processes*. A data process may be simple data, but may also involve retrieval from other databases, as well as computation. An example of a database system that allows a data type that corresponds to retrieval commands, can be found in [27].

A. Experimentation

To establish the feasibility of the methods described in this paper, and to obtain feedback on their promise, a small prototype of a virtual database system was implemented (using the programming language Lisp), and examples, similar to those described here, were tested. The system consists of several components. They allow the user to: 1) define a new actual database, 2) load a new actual database, 3) query an actual database, 4) define a new virtual database, and 5) query a virtual database. In general, the prototype achieved its purpose, as it demonstrated feasibility, and exposed several inadequacies (later corrected) in the definitions of the operators and in the translation procedures.

Admittedly, the different components of our model (the database model, the access language, and the integration language) are all of modest detail. This abstract approach enabled us to concentrate on the basic problems and treat them in a formal way. In an actual system, various pragmatic issues must be considered, and additional features must be added. It is intended that the principles described here would guide this larger implementation. Three important issues that still remain to be investigated are discussed below.

B. Update of Virtual Databases

We have focused our attention on the construction and interrogation of virtual databases, and the possibility of *updating* virtual databases has not been considered. While this issue requires additional research, we estimate that most virtual updates may be performed satisfactorily, and we sketch here a possible approach for incorporating update operations into our model.

In general, our approach for translating update operators parallels the approach described in Section V for translating access operators. First, a set of update operators should be adopted. Then, primitive translation rules would be determined for each possible situation. Finally, the primitive rules would be incorporated into a global translation procedure. We illustrate this approach by means of two simple examples.

Assume T **att** S, and consider this update operator to *modify* the current functional value $T(S = x)$ to y:

$$T(S = x) \leftarrow y$$

As an example, consider an update request to modify NAME(PERSON $= x$) to y. This request would simply be translated into two parallel requests to update *both* STUDENT(PERSON $= x$) and FACULTY(PERSON $= x$) to this new value. The original update request would be considered successful, if *at least one* of the actual updates completes successfully.

Consider now this operator to *insert* a new value x into the domain of S:

$$S \leftarrow x$$

This operator also inserts [*not-available*] as the values x has for each attribute in **type**(S). These initial null values may then be modified with the previous operator. Consider now a request to insert a new value into the domain of a *virtual* class, such as PERSON. The new value would be inserted into the domains of both STUDENT and FACULTY, with [*not-available*] values for the attributes of PERSON, and [*not-applicable*] values for the distinguishing attributes of STUDENT and FACULTY. The interpretation of these null values should be that the corresponding "student" or "faculty" is not a valid member of this domain, only an "occupant." Future queries on STUDENT or FACULTY should ignore any "occupants" of these domains. Future queries on PERSON will be answered correctly.

C. Data Incompatability

The first step of an integration process is to combine pairs of primitive classes from the two databases. Because the **combine** operator generates a class whose domain is the *union* of the domains of the given classes, it can only be applied to classes that are *entirely compatible* in their semantics and in the format of their values.

In practice, in may be desirable to connect two databases over primitive classes that are not entirely compatible. There are various kinds of incompatibility of primitive classes, and we give here three examples: 1) Both databases include a class GRADE; but while in one database its values are numbers in the range 0–100, in the other database its values are letters in the range {A,B,C,D,F}. 2) Both databases include a class ZIP; but while in one database its values are five-digit zip codes, in the other database its values are extended nine-digit zip codes. 3) In one database there is a class FULL-NAME whose values are combinations of first and last names, while in the other database this information is stored in two separate classes FIRST-NAME and LAST-NAME.

Many kinds of incompatibility may be handled by generalizing **combine** to a more powerful operator. While the domain of the unifying class created by **combine** is the union of the given domains, the new operator would allow the creation of a unifying class, whose domain would be *mapped* onto the domains of the given classes. For example, in the case of GRADE, any of the two domains could be selected as the unifying domain, and mappings would be defined from this domain onto the two individual domains. When a query about a particular grade is evaluated in both databases, one database will return a letter grade, and the other will return a number grade. These values would first be converted according to the mappings, and then combined with the *best-value* operator.

D. Integration of Heterogeneous Databases

One of the assumptions in this paper was that all databases are organized according to the same database model. In practice, however, the databases that need to be integrated may be organized according to different database models.

This problem of integrating *heterogeneous databases* is considerably more complex. The solution we propose is to transform the heterogeneous case to the homogeneous case, by translating the schemas of all the existing databases into our database model (a similar approach was taken in Multibase).

As already mentioned, our functional database model can express the *principal structures* of both the relational model (i.e., relations) and the network model (i.e., CODASYL sets), as well as some others. Thus, in principle, it should be possible to translate databases from these models into the functional model (for more details, see [26]). In practice, considering the numerous features of actual database models, the conversion of databases from one model to another is a complex task of engineering.

References

[1] W. Litwin, "MALPHA: A relational multidatabase manipulation language," in *Proc. IEEE Comput. Soc. First Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 24-27, 1984, pp. 86-93.

[2] E. F. Codd, "A database sublanguage based on the relational calculus," in *Proc. ACM-SIGFIDET Workshop Data Description, Access and Control*, San Diego, CA, Nov. 1971, pp. 35-68.

[3] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. W. T. Lin, and E. Wong, "Multibase—Integrating heterogeneous distributed database systems," in *Proc. AFIPS Nat. Comput. Conf.*, Chicago, IL, May 4-7, pp. 487-499.

[4] T. A. Landers and R. L. Rosenberg, "An overview of multibase," in *Distributed Databases*, H. J. Schneider, Ed. Amsterdam, The Netherlands: North-Holland, 1982.

[5] U. Dayal and H. Hwang, "View definition and generalization for database integration of a multidatabase system," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 628-644, Nov. 1984.

[6] Y. Breitbart, P. L. Olson, and G. R. Thompson, "Database integration in a distributed heterogeneous database system," in *Proc. Second Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 5-7, 1986, pp. 301-310.

[7] A. Motro and P. Buneman, "Constructing superviews," in *Proc. ACM-SIGMOD Int. Conf. Management of Data*, Ann Arbor, MI, Apr. 29-May 1, 1981, pp. 56-64.

[8] A. Motro, "Interrogating superviews," in *Proc. ICOD-2, Second Int. Conf. Databases*, Cambridge, England, Aug. 30-Sept. 3, 1983, pp. 107-126.

[9] A. G. Merten and J. P. Fry, "A data description language approach to file translation," in *Proc. ACM SIGFIDET Workshop Data Description, Access and Control*, Ann Arbor, MI, May 1974.

[10] J. A. Ramirez, N. A. Rin, and N. S. Prywes, "Automatic conversion of data conversion programs using a data description language," in *Proc. ACM-SIGFIDET Workshop Data Description, Access and Control*, Ann Arbor, MI, May 1974.

[11] B. C. Housel, "A unified approach to program and data conversion," in *Proc. Third Int. Conf. Very Large Data Bases*, Tokyo, Japan, Oct. 6-8, 1977, pp. 327-335.

[12] R. W. Taylor, J. P. Fry, B. Schneiderman, D. C. P. Smith, and S. Y. W. Su, "Database program conversion: A framework for research," in *Proc. Fifth Int. Conf. Very Large Data Bases*, Rio de Janeiro, Brazil, Oct. 3-5, pp. 299-312.

[13] N. C. Shu, B. C. Housel, and V. Y. Lum, "CONVERT: A high level translation definition language for data conversion," *Commun. ACM*, vol. 18, no. 10, pp. 557-567, Oct. 1975.

[14] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum, "EXPRESS: A data extraction, processing and restructuring system," *ACM Trans. Database Syst.*, vol. 2, no. 2, pp. 134-174, June 1977.

[15] R. ElMasri and G. Wiederhold, "Data model integration using the structural model," in *Proc. ACM-SIGMOD Int. Conf. Management of Data*, Boston, MA, May 29-June 1, 1979, pp. 319-326.

[16] S. Navathe and S. Gadgil, "A methodology for view integration in logical database design," in *Proc. Eighth Int. Conf. Very Large Data Bases*, Mexico City, Mexico, Sept. 8-10, 1982, pp. 142-152.

[17] R. ElMasri and S. Navathe, "Object integration in logical database design," in *Proc. IEEE Comput. Soc. First Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 24-27, 1984, pp. 426-433.

[18] M. V. Mannino, "Matching techniques in global schema design," in *Proc. IEEE Comput. Soc. First Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 24-27, 1984, pp. 418-425.

[19] J. D. Ullman, *Principles of Database Systems*. Rockville, MD: Computer Science Press, 1982, pp. 6-8.

[20] E. H. Sibley and L. Kerschberg, "Data architecture and data model considerations," in *Proc. AFIPS Nat. Comput. Conf.*, Dallas, TX, June 13-16, 1977, pp. 85-96.

[21] D. W. Shipman, "The functional data model and the data language DAPLEX," *ACM Trans. Database Syst.*, vol. 6, no. 1, pp. 140-173, Mar. 1981.

[22] P. Buneman and R. E. Frankel, "FQL-A functional query language," in *Proc. ACM-SIGMOD Int. Conf. Management of Data*, Boston, MA, May 29-June 1, 1979, pp. 52-57.

[23] S. B. Yao, V. E. Waddle, and B. C. Housel, "View modeling and integration using the function data model," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 544-553, Nov. 1982.

[24] J. M. Smith and D. C. P. Smith, "Database abstractions: Aggregation and generalization," *ACM Trans. Database Syst.*, vol. 2, no. 2, pp. 105-133, June 1977.

[25] M. Hammer and D. McLeod, "Database description with SDM: A semantic database model," *ACM Trans. Database Syst.*, vol. 6, no. 3, pp. 351-386, Sept. 1981.

[26] A. Motro, "Virtual merging of databases," Ph.D. dissertation, Dep. Comput. and Inform. Sci., Univ. Pennsylvania, 1981.

[27] M. Stonebraker, E. Anderson, E. Hanson, and B. Rubinstein, "QUEL as a data type," in *Proc. ACM-SIGMOD Int. Conf. Management of Data*, Boston, MA, June 18-21, 1984, pp. 208-214.

Amihai Motro received the B.Sc. degree in mathematical sciences from Tel Aviv University, Tel Aviv, Israel, in 1972, the M.Sc. degree in computer science from the Hebrew University in Jerusalem, Israel, in 1976, and the Ph.D. degree in computer and information science from the University of Pennsylvania, Philadelphia, in 1981.

Since 1981 he has been an Assistant Professor in the Department of Computer Science at the University of Southern California, Los Angeles. His main research area is data management; in particular intelligent user interfaces to databases, knowledgeable data management, and integration of databases. He is also interested in operating systems, and has worked for several years as a systems programmer.

Dr. Motro is a member of the Association for Computing Machinery and the IEEE Computer Society.