Constructing Superviews

Amihai Motro and Peter Buneman

Department of Computer and Information Science,
Moore School, University of Pennsylvania,
Philadelphia, Pa. 19104.

Abstract

A method is described for integrating two or more databases into a conceptual "superview", through a set of schema transformations. Such integration may be useful when it is required to produce a unified view of two databases while preserving their physical independence. Each transformation defines a mapping of queries against the superview into the appropriate set of queries against the underlying databases and imposes a constraint that is checked when the query is evaluated. A program that interactively aids the user in constructing the superview and that performs this query transformation is being developed.

1 Introduction

Even with the increased use of large and complex databases, it often happens that the information required for a specific application, or set of applications, extends over two or more physically independent databases. The writing of such applications is considerably simplified if the databases appear to the program as a single integrated database. However, the cost of performing any physical restructuring may be prohibitive and may impose unnecessary constraints on the structure and content of the database as it is viewed by the original users.

We describe a method that will perform a virtual merge of existing independent databases, that presents the user with a larger conceptual structure that may be queried and possibly updated without compromising the independence of the existing databases. This process is in some sense the inverse of constructing "user-views" or "external schemas". Given two or more logical schemas, what larger schema has these schemas as user views? We shall call this larger schema a superview and the purpose of this paper is to describe a set of formal schema transformations for the construction and manipulation of superviews.

There are a number of proposals that relate to this kind of database merging. McLeod and Heimbigner [1] have suggested a "federated database architecture". They note the limitation of current approaches to database distribution and suggest an alternative in which databases are physically distributed but externally represented by a central logical schema, that is derived from the schemas of the component databases. This federal schema is used to specify information that may be shared by the various components and to formalize communication among the individual databases. Another proposal, based on a functional data model by Shipman [2], suggests that a global schema could be built for several databases and provides a method for defining a function in the global schema from a set of functions defined in the component databases. To our knowledge, no attempt has yet been made to provide any general set of schema restructuring tools with which the global schema may be created and from which the appropriate mappings of access paths may be deduced.

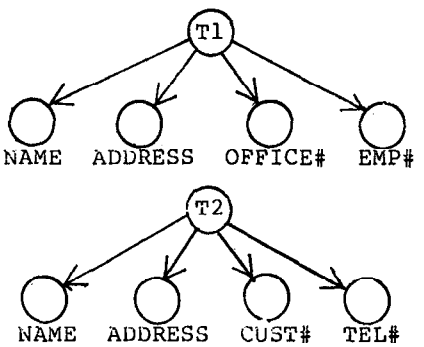
The transformations described in this paper have been incorporated into a program that, under interactive control, constructs the superview. It is an interesting property of these transformations, that once some low level identifications have been made, much of the higher level merging can proceed automatically. A second component of this program transforms queries against the the superview into queries against the component schemas, and checks integrity constraints that may have been introduced in the construction of the superview. In this system, no physical

constraints are placed on the component databases. The constraints, if they exist and are violated, will only cause failure when an attempt is made to interpret a query. We would hope that such a system might prove useful when short-term restructuring is needed for the extraction of specific data aggregates. For the preservation of long-term superviews, it is probably more natural and efficient, as is suggested by the Federated DBMS architecture, to enforce these constraints permanently.

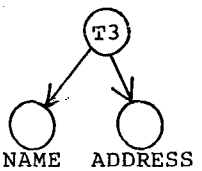
In order to define a set of schema transformations, a formal data model is clearly necessary. The model we shall describe has close affinities to the Sematic Data Model defined by Hammer and McLeod [3], the aggregation/generalization hierarchies of Smith and Smith[4,5], the functional approach suggested by Sibley and Kershberg [6] and the functional model used by Shipman [2]. The next section is devoted to a description of our model. However, we see little difficulty in extending the technique to these related models.

2 The Abstract Data Model

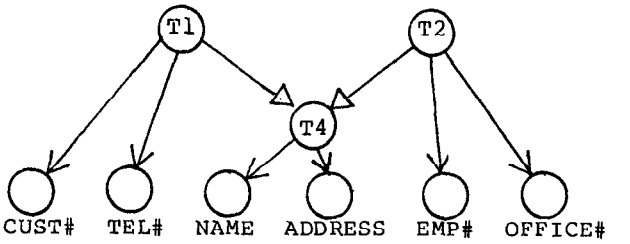
To illustrate the problems in constructing a superview consider an attempt to merge two (very simple) databases:



The arrows in this diagram indicate an attribute relationship, for example that NAME is an attribute of T1. In merging these two databases we may be seeking just the common attributes:



or we may require that a database that represents both the common attributes and the attributes that distinguish the two databases:



where we have introduced a new relationship (indicated by ——▷). This is a subtype relationship: T1 is a subtype of T4, and as such inherits the attributes of T4, that is, both NAME and ADDRESS are attributes of T1. These two relationships, attribute and subtype, are precisely those of aggregate and generalization proposed by Smith and Smith [4,5]. The simple, but important, point to be made is that while the original database schemas did not include subtype relationships, it may be necessary to introduce them in order to produce an accurate description of the combined data.

At the basis of the data model used here is a functional approach, first described by Sibley and Kershberg [6]. This approach employs the notions of data domains and attribute functions: domains are sets of data objects, functions assign the objects of one domain to objects of another domain as their attributes. The version of the functional data model that we describe here bears a close relationship to functional models that have been described in other contexts [2,7,8]. While the details may differ, we see no real difficulty in modifying the techniques described here to work against these other models. 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 [9].

Assume a collection D of named classes such that

(1) each class S has a domain dom(S) of objects,
(2) two relations att and gen are defined on D,
(3) for every two classes S,T, such that S att T, there is a function f_{ST} : dom(S) -> dom(T),
(4) for every two classes S,T, such that S gen T, there is an injection i_{ST} : dom(S) -> dom(T).

The collection D of classes (with their associated domains) incorporates two types of relationships. The att relationship, by which one class becomes an attribute of another class, is supported by functions. The gen relationship, by which one class becomes a generalization of another class, is supported by one-to-one functions.

As an example consider the classes FACULTY, STUDENT, PERSON, SS#, NAME, OFFICE, SCHOOL, S_NAME, with the relationships

SS# att FACULTY,
SS# att STUDENT,
SS# att PERSON,
NAME att FACULTY,
NAME att STUDENT,
NAME att PERSON,
OFFICE att FACULTY,
SCHOOL att STUDENT,
S_NAME att SCHOOL,
OFFICE att SCHOOL,
PERSON gen FACULTY,
PERSON gen STUDENT.

Note that, each FACULTY must have exactly one NAME and be exactly one PERSON, but while several different members of FACULTY may have the same NAME, they each must be a different PERSON.

Classes that do not have any attributes are called primitive classes. Their members are primitive objects. In the example SS#, NAME, OFFICE and S_NAME are primitive classes. FACULTY, STUDENT, PERSON and SCHOOL are non-primitive.

Besides domains, classes also have types. Types are derived from the relation att.

Definition 1: The type of a given class S in D is

$$type(s) = \{T \mid T \text{ att } S\}.$$

Clearly, primitive classes have empty types. The non-empty types in the above example are:

type(FACULTY) = (SS# NAME OFFICE),
type(STUDENT) = (SS# NAME SCHOOL),
type(PERSON) = (SS# NAME),
type(SCHOOL) = (S_NAME OFFICE).

For many applications it is necessary that every member of a domain is uniquely identifiable by a combination of its primitive attributes. This is especially important for the purpose of merging two different databases, so that when their two populations are consolidated, identical objects can be recognized as such. We must therefore identify a key relationship between classes.

Definition 2: Assume $\{T1,\ldots,Tn\} \subseteq type(S)$. (T1 ... Tn) key S if

$$\begin{cases} f : dom(S) \to dom(T1) \times \ldots \times dom(Tn) \\ f(x) = (f_{ST1}(x),\ldots,f_{STn}(x)) \end{cases}$$

is an injection.

Thus, the classes (T1 ... Tn) constitute a key to class S, if a combination (x1 ... xn) of objects from these classes determines at most one object of S. Assuming every person has a different Social Security number, the key relationships in the above example are:

SS# key FACULTY,
SS# key STUDENT,
SS# key PERSON,
S_NAME key SCHOOL.

Keys are not necessarily primitive or simple (constituting a single class), as the above example might suggest. A class ENROLLMENT may be introduced, which is keyed on the combination of a non-primitive class COURSE and the non-primitive class STUDENT. By composing keys each non-primitive object can be identified by a combination of primitive objects.

To become a proper database, a few more requirements are imposed on the structure defined so far. They are stated in the following definition.

Definition 3: A collection D of classes with relations, domains, functions and injections (as in 1-4 above) is a database if:

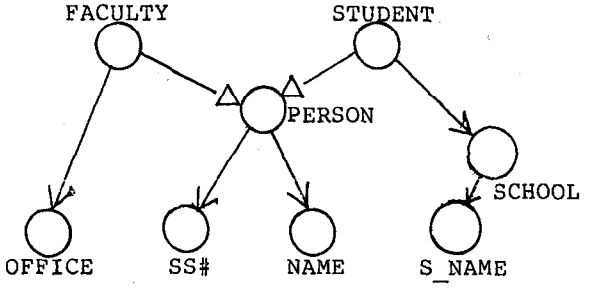
(1) the intersection of att and gen is empty and the union has irreflexive transitive closure,
(2) S att T, T gen R $\Rightarrow$
S att R, $f_{RS} = i_{RT} \circ f_{TS}$,
S gen T, T gen R $\Rightarrow$
S gen R, $i_{RS} = i_{RT} \circ i_{TS}$,
(3) $\forall x,y \in dom(S)$:
$(\forall T \in type(S): f_{ST}(x)=f_{ST}(y)) \Rightarrow x=y.$

The first condition guarantees that one class is 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 inheritance of attributes over generalizations and the transitivity of generalizations are assured by the second condition. The last condition states that no two objects in a domain have the same values for all their attributes; members of each domain are distinguishable by at least one attribute. The underlying justification is that this enforces a more accurate semantic specification; if it is necessary to distinguish between such objects, an appropriate attribute should be present. A consequence of the last condition is that each class in the database can always be assigned one key, the trivial key comprising the entire type. Also, if S gen T, then inheritance guarantees that every attribute of S is also an attribute of T. In particular, a key of S is composed of attributes of T. Because the composition of injections is an injection, the key of S is also a key of T. These consequences are summarized in the following statement:

(1) Every class is guaranteed a key,
(2) Classes related by generalization have the same key.

We have been using a graphic representation of a database schema. Each database class is represented by a node. If T att S, there is a directed arc from node S to node T:  . If T gen S, there is a directed arrow from node S to node T: (an edge is either an arc or an arrow). However, if T gen R and S att T, then S att R is suppressed in the

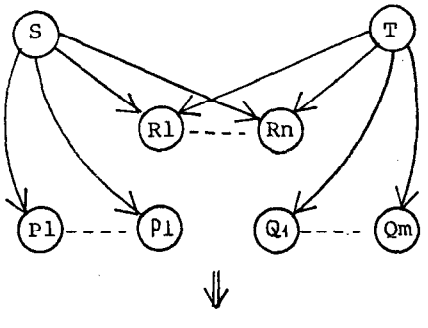
graphic representation. Similarly, if S gen T and T gen R, then S gen R is suppressed (these are the inheritance and transitivity discussed above. For conciseness, these relationships will also be suppressed in all future specifications of databases). Graphs that represent databases do not have cycles or parallel edges. The graphic representation of the above example is:

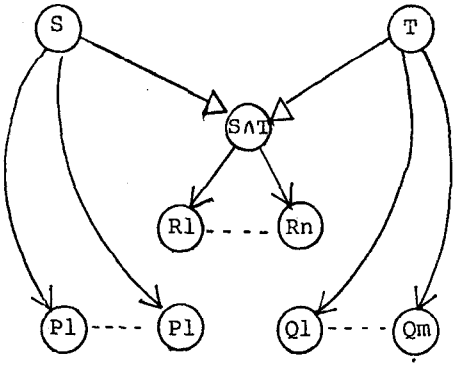


3 Restructuring Primitives

In this section we describe a small set of restructuring transformations that merge or modify database schemas. With each operator there is an associated set of constraints that must be satisfied by the objects that populate the component schemas. In the current implementation these constraints are checked only upon interrogation. We begin by introducing three primitives (meet, join and fold) that manipulate the generalization hierarchy.

3.1 Meet. The meet operator produces a common generalization of two classes, if such a generalization may be found. The existence of a generalization is determined by the properties of their keys. The example that introduces Section 2 shows how meet is applied to an employee (T1) and a customer (T2) to produce person (T4). This operation is based on the existence of a common key (NAME). Formally, assume that S and T are non-primitive classes not related by gen. Assume there exists $K \subseteq type(S) \cap type(T)$ that maintains K key S and K key T. The transformation meet S and T is performed by adding a new class, the meet of S and T, denoted by S∧T, and the relationships S∧T gen S, S∧T gen T and Ri att S∧T (i=1,....,n). The type of S∧T is therefore given by type(S∧T) = type(S) ∩ type(T). The graphic representation of meet is





In this figure the common attributes (i.e. type(S) ∩ type(T)) are represented by R1,...,Rn. The attributes that distinguish S and T are represented by P1,...,Pl and Q1,...,Qm, respectively. The new class is populated with the union of the domains of S and T: dom(S∧T) = dom(S) ∪ dom(T). The injections from dom(S) and dom(T) into dom(S∧T) are defined as identities. The functions from dom(S∧T) into the domains of R1,...,Rn are defined to preserve inheritance. The latter functions require a consistency constraint: objects in dom(S) or dom(T) that have the same key, must agree over their other shared attributes. Formally, denote by f1,...,fn and g1,...,gn the attribute functions from S and T, respectively, into R1,...,Rn. Let $K = \{R1,...,Rk\}$. Define functions f and g as follows:

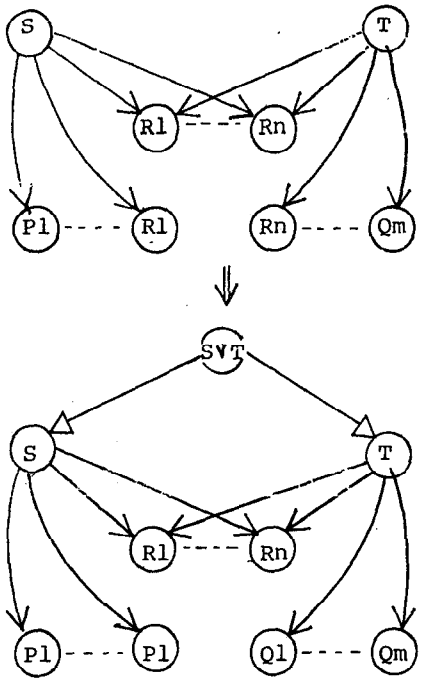
$$\begin{cases} f : dom(S) \to dom(R1) \times ... \times dom(Rk) \\ f(x) = (f1(x),...,fk(x)) \end{cases}$$

$$\begin{cases} g : dom(T) \to dom(R1) \times ... \times dom(Rk) \\ g(x) = (g1(x),...,gk(x)) \end{cases}$$

Then $\forall y \in f(dom(S)) \cap g(dom(T))$:

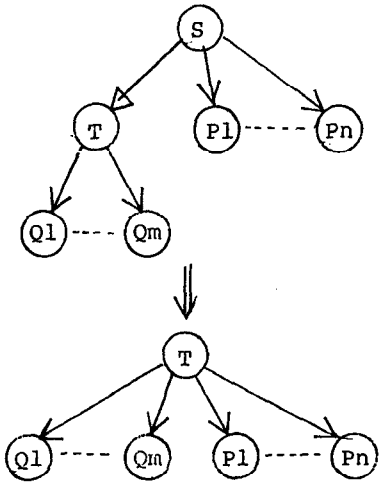
$$fi(f^{-1}(y)) = gi(g^{-1}(y)),\ i=k+1,...,n.$$

3.2 Join. meet generates a class whose type is the intersection of both types and whose domain is the union of both domains. 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 = (SS# NAME OFFICE) and STUDENT = (SS# NAME SCHOOL). The meet of FACULTY and STUDENT is the class PERSON = (SS# NAME). Its domain includes all those which are either FACULTY or STUDENT. PERSON generalizes both FACULTY and STUDENT. The join of STUDENT and FACULTY is the class ASSISTANT = (SS# NAME SCHOOL OFFICE). Its domain includes all those which are both FACULTY and STUDENT. ASSISTANT is generalized by both FACULTY and STUDENT. Formally, assume S and T maintain the same conditions as before. The transformation join S and T is performed by adding a new class, the join of S and T, denoted by S∨T, and the relationships S∨T gen S, S∨T gen T, Ri att S∨T (i=1,...,n), Pi att S∨T (i=1,...,l) and Qi att S∨T (i=1,...,m). The type of S∨T is therefore given by type(S∨T) = type(S) ∪ type(T). The graphic representation of join is



The domain of S∨T is dom(S∨T) = dom(S)∩dom(T). The injections from dom(S∨T) into dom(S) and dom(T) are defined as identities. The functions from dom(S∨T) into the domains of R1,...,Rn, P1,...,P1 and Q1,...,Qm are defined to preserve inheritance. Again, the same consistency constraint is required.

3.3 Fold. meet and join add a generalization. fold removes a subtype. With fold a subtype class STUDENT may be folded into the more general class PERSON, with the distinguishing STUDENT attributes carried over to PERSON (adding special "undefined" values for non-STUDENTs). Formally, assume S and T are two non-primitive classes such that T gen S. The transformation fold S into T is performed by removing the class S and replacing it with T in all relationships.

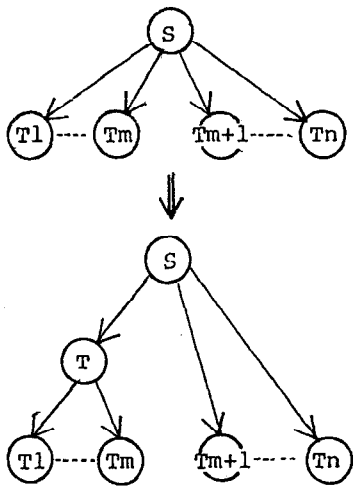


Functions and injections that had dom(S) as their domain are modified to have dom(T) as their new domain, using the previous injection from dom(S) into dom(T) (and an "undefined" value for objects in dom(T) but not in the image of this injection). Using the same injection, functions and injections that had dom(S) as their range are modified to have dom(T) as their new range.

meet is the principal operator. With meet the similarity between two semantically related classes, which are not identical, may be expressed. If the type of one class contains the type of the other class, meet produces a situation suitable for folding. After fold is applied, one class becomes a generalization of the other. Consider the previous class STUDENT and GRAD_STUDENT =(SS# NAME SCHOOL DEGREE). The result of meet is a new class STUDENT' = (SS# NAME SCHOOL) whose domain is the union of both domains. STUDENT is then folded into STUDENT'. In the end STUDENT' and GRAD_STUDENT are connected via a generalization. If the two classes happen to have identical types, then after the application of meet, fold may be applied twice. In the end the two classes are combined into one class whose domain is the union of both domains. In the last two situations, the same results may be achieved by performing a join, followed by one or two folds.

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

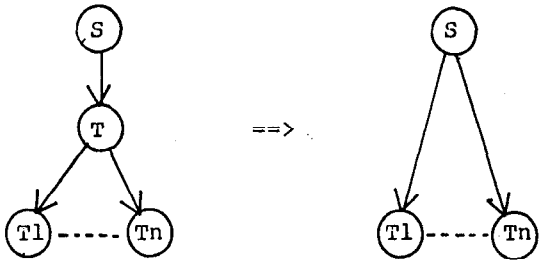
3.4 Aggregate. The attribute hierarchy may be extended by aggregating a subset of the attributes of a given class into a separate class, which then becomes an attribute of the original class. Formally, assume S is a non-primitive class, type(S) = (T1 ... Tm Tm+1 ... Tn). The transformation aggregate (T1 ... Tm) into T of S is performed by adding a new class T and the relationship T att S. Also every relationship Ti att S is replaced with Ti att T. Graphically,



The domain of T is populated with new objects, that are all the possible tuples

of objects from the aggregated domains: dom(T) = dom(T1) x ... x dom(Tm). The function that supports the relationship between S and T is defined by $f_{ST}(x) = (f_{ST1}(x),...,f_{STm}(x))$. The functions from dom(T) onto dom(Ti) (i=1,...,m) are simple projections.

3.5 Telescope. While aggregate extends the attribute hierarchy, telescope performs the inverse: it removes a class by assigning its attributes directly to its ancestor. Formally, assume T is a non-primitive class, type(T) = (T1 ... Tn), which is an attribute of only one class S. The transformation telescope T into S is performed by removing the class T and the relationship T att S, and replacing the relationships Ti att T with Ti att S. Graphically,



The functions that support the new attribute relationships are simple compositions: $f_{STi}(x) = f_{TTi}(f_{ST}(x))$ (i=1,...,n).

With aggregate and telescope, a class may be relocated on the schema. Consider the classes PATIENT = (SS# NAME AGE FAMILY) and FAMILY = (ADDRESS SIZE DOCTOR). By telescoping FAMILY into PATIENT and then aggregating ADDRESS and SIZE into FAMILY, the attribute DOCTOR is relocated from FAMILY to PATIENT. Relocation can take place in both directions.

aggregate may be used to bring a schema to a normal form, in which the non-key attributes of each class are fully dependent on the key. If a class exists with some attributes which 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 ORDER = (PART# PART_NAME SUPPLIER# SUPPLIER_NAME QUANTITY). The key of ORDER is both PART# and SUPPLIER#, but only QUANTITY depends on both; PART_NAME depends only on PART# and SUPPLIER_NAME depends only on SUPPLIER#. Therefore this class is not in normal form. Using aggregate twice, the following schema may be obtained: PART = (PART# PART_NAME), SUPPLIER = (SUPPLIER# SUPPLIER_NAME) and ORDER = (PART SUPPLIER QUANTITY). The new schema is in normal form.

As a third example, consider the classes ACCOUNT = (ACC# NAME BALANCE) and TRANSACTION = (TRANS# ACC# AMOUNT). Consider the following four operations. First ACC# is aggregated into an interim class ACCOUNT' which becomes an attribute of TRANSACTION. Having a common key ACC#, ACCOUNT and ACCOUNT' are then generalized by ACCOUNT''. Finally ACCOUNT and ACCOUNT' are each folded into ACCOUNT'' (which is then renamed ACCOUNT). The final result is a retraction of TRANSACTION from ACC# to ACCOUNT: ACCOUNT is left unchanged, but TRANSACTION is modified to TRANSACTION = (TRANS# ACCOUNT AMOUNT), which is more semantically accurate, since each TRANSACTION has its own ACCOUNT, rather than an ACC#. Like normalization, retraction may be applied wherever possible to obtain a "better" representation of the schema. (Note that if ACC# 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 are semantically "comparable").

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

3.6 Add. In general, the addition of a new class and the specification of its attribute relationships to existing classes is actually an augmentation of the current database by another database and therefore may not be considered as 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 a 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 merged. Formally, assume S is a non-primitive class. Let P be a new primitive class with a single object domain. The transformation add P(x) to S is performed by adding the class P and the 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 the class BOOK = (BOOK# TITLE AUTHOR). The combined class contains the unified collection of books. However, the information on where each book is shelved is lost. With the help of add, this implied knowledge can be added to each class BOOK before they are combined. The combined class BOOK = (BOOK# TITLE AUTHOR LIBRARY) includes the source library for each book. Note that the class LIBRARY must now be included as part of the key of BOOK. Hence add may require that the new attribute is added to the key.

3.7 Delete. To remove portions of the database which are not relevant to the application the delete primitive may be used. In other words, the delete operator enables user-views. Assume S is a non-primitive class and T att S. The transformation delete T from S is performed by removing 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 attributes 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 has serious semantic implications: in the domain of the new class S objects that were previously differentiated only by their key value, are now identified. For example, the deletion of COURSE# from ENROLLMENT = (COURSE# SS# GRADE) generates (SS# GRADE), a class whose meaning is unclear.

In general, aggregate, telescope, add and delete may be used to iron-out structural differences between the two candidate databases, so that better overlapping is achieved.

4 The Merge Technique

The merge technique consists of an initial step, followed by a sequence of restructuring primitives.

In the initial step the user relates the two independent databases by pairing primitive classes. Each pair associates a primitive class in one database with a primitive class in the other database. Each pair is then combined into one primitive class with a unified domain.

To be correct, the primitive classes in each pair should model the same real world entity. Thus, two classes describing the Social Security number may probably be associated, but the employee number in two different organizations may indicate two independent sequencings, which do not have any global meaning. The latter does not create problems, unless these classes participate in keys. Identical objects (i.e. the same employee in both organizations) could remain separate.

Once these initial associations have been supplied, the two databases are connected to become one. From here on the process is that of restructuring, with the purpose of identifying similar structures. We demonstrate this technique by means of an example.

Assume an organization with two independent databases. One describes the assignment of employees to projects. The other gives details on the different projects. An employee may participate in several projects, but each project has a unique employee to manage it. The definition of the first database is as follows (for a graphical representation see Figure 1):

EMPLOYEE att ASSIGNMENT,
PROJECT att ASSIGNMENT,
PERSON gen EMPLOYEE,
JOB att EMPLOYEE,
DEPARTMENT att EMPLOYEE,
SS# att PERSON,
NAME att PERSON,
ADDRESS att PERSON,
JOB_CODE att JOB,
JOB_DESC att JOB,
D_NAME att DEPARTMENT,
OFFICE att DEPARTMENT,
ROOM# att OFFICE,
PHONE# att OFFICE.

By making DEPARTMENT an attribute of EMPLOYEE, each EMPLOYEE is constrained to one DEPARTMENT. The possibility for one EMPLOYEE to participate in several PROJECTs is expressed by relating them through ASSIGNMENT, which is then keyed on both. The keys are:

(EMPLOYEE PROJECT) key ASSIGNMENT,
SS# key EMPLOYEE,
SS# key PERSON,
JOB_CODE key JOB,
D_NAME key DEPARTMENT,
ROOM# key OFFICE.

The second database is much less detailed; each project is described by its project number, its manager and the total budget (see Figure 2):

P# att PROJECT,
MANAGER att PROJECT,
BUDGET att PROJECT.

The only key relationship is P# key PROJECT. Note that PROJECT in the first database actually refers to project numbers, while MANAGER in the second database contains only Social Security numbers.

We now issue the restructuring requests to merge these two databases. The initial step consists of two associations only (the result is shown in Figure 3):

(1) PROJECT and P# are combined into P#,
(2) SS# and MANAGER are combined into SS#.

Our first goal is to make PROJECT an attribute of ASSIGNMENT. This retraction is performed in four primitive steps (the result is shown in Figure 4):

(1) aggregate (P#) into PROJECT' of ASSIGNMENT,
(2) meet PROJECT and PROJECT' (new class is T),
(3) fold PROJECT' into T,
(4) fold PROJECT into T (rename T to PROJECT).

Next we create a subtype of EMPLOYEE, to be called MANAGER, which will replace SS# as an attribute of PROJECT. To achieve this we aggregate the SS# of PROJECTs into an interim MANAGER, and assign to it the (undefined) attributes NAME, ADDRESS, JOB and DEPARTMENT:

(5) aggregate (SS#) into MANAGER of PROJECT,
(6) add NAME(NIL), ADDRESS(NIL), JOB(NIL), DEPARTMENT(NIL) to MANAGER.

Now, since EMPLOYEE and MANAGER have the same key, meet and fold may be applied, resulting in EMPLOYEE becoming a generalization of MANAGER (the final database is shown in Figure 5):

(7) meet EMPLOYEE and MANAGER (new class is T),
(8) fold EMPLOYEE into T (rename T to EMPLOYEE).

A total of two initial associations and eight restructuring primitives were needed to arrive at the final database. Under a simple assumption, this merge technique can be improved, so that a large part of the restructuring may be inferred from the given databases. This assumption states that the type of a class (i.e its set of attributes) incorporates all the necessary characterizations. It follows from this "complete semantics" assumption, that every two classes with the same type may safely be combined (given that consistency is maintained), or else each class should have included a distinguishing attribute. Thus, all meet and fold transformations may be executed automatically, without user initiation.

In addition, retractions and normalizations are always semantically correct. Therefore, they too may be inferred from the databases. The new merge technique will use these inferences in the following way. After the initial associations, all inferred transformations will be applied. Once the process stops, the user may "revive" it with additional restructuring requests. These, in turn, may cause more inferred transformations to take place, and so on. Using this technique in the previous example, out of the eight transformations only two (5 and 6) must be initiated from outside. This technique may still be used, even if the assumption of "complete semantics" is not always correct. Instead of applying the inferred transformations, they are only suggested for user approval. Thus, database merging is seen as an interactive process with a helpful system, in which the given schemas are "edited" to a satisfactory structure.

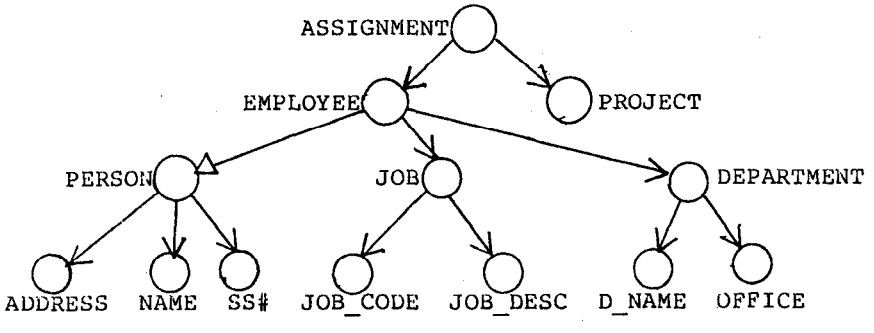


Figure 1

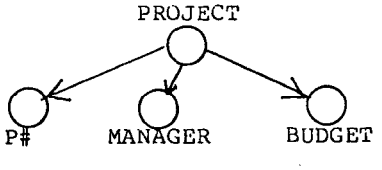


Figure 2

References

[1] D.McLeod and D.Heimbigner; A Federated Architecture for Database Systems; Proceedings of AFIPS National Computer Conference, Anaheim, California, 1980.

[2] D.W.Shipman; The Functional Data Model and the Data Language DAPLEX; Proceedings of ACM-SIGMOD International Conference on Management of Data, Boston, Massachussetts, 1979 (to appear in ACM-TODS).

[3] M.Hammer and D.McLeod; A Semantic Data Model: A Modelling Mechanism for Data Base Applications; Proceedings of ACM-SIGMOD International Conference on Management of Data, Austin, Texas, 1978.

[4] J.M.Smith and D.C.P.Smith; Database Abstractions: Aggregation; Communications of the ACM, Vol.20, No.6, June 1977.

[5] J.M.Smith and D.C.P.Smith; Database Abstractions: Aggregation and Generalization; ACM Transactions on Database Systems, Vol.2, No.2, June 1977.

[6] E.H.Sibley and L.Kershberg; Data Architecture and Data Model Considerations; Proceedings of AFIPS NAtional Computer Conference, Dallas, Texas, 1979.

[7] B.C.Housel, V.E.Waddle and S.B.Yao; The Functional Dependency Model for Logical Database Design; Proceedings of the Fifth International Conference on Very Large Data Bases, Rio De Janeiro, Brazil, 1979.

[8] P.Buneman and R.E.Frankel; FQL - A Functional Query Language; Proceedings of ACM-SIGMOD International Conference on the Management of Data, Boston, Massachussetts, 1979.

[9] A.Motro and P.Buneman; Automatically Merging Databases; Proceedings of COMPCON Fall 80 - Distributed Computing, The 21st IEEE Computer Society International Conference, Washington, D.C., 1980.

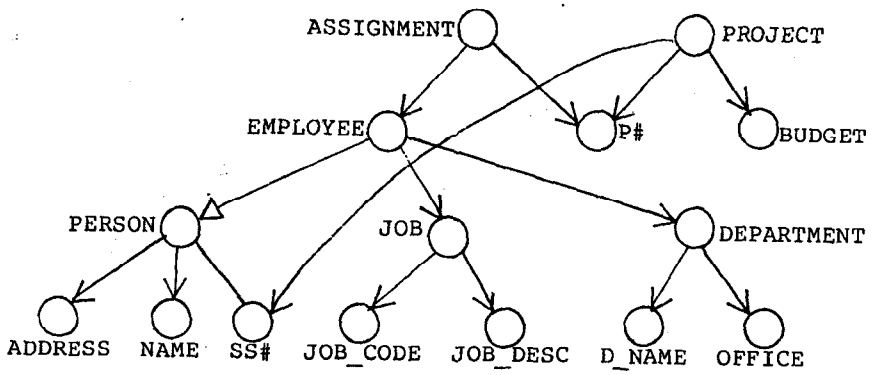


Figure 3

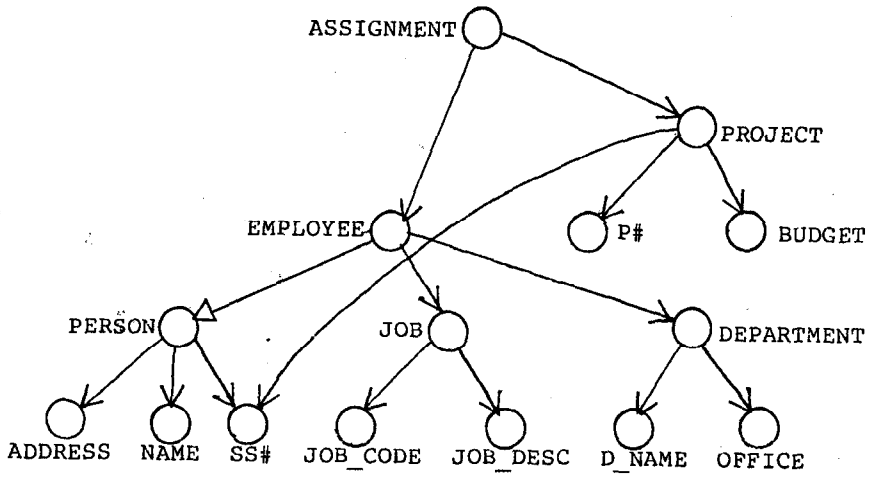


Figure 4

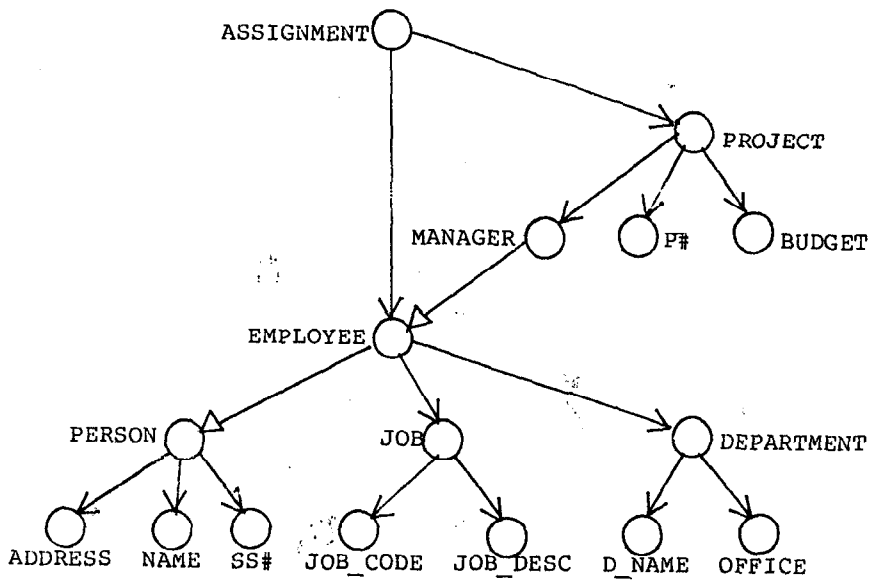


Figure 5