Chapter 6

INTERROGATING SUPERVIEWS

Amihai Motro

Department of Computer Science, University of Southern California, Los Angeles, California CA 90089, USA

The concept of database is generalized to include virtual databases: databases which do not contain any data but are mapped onto other databases. Virtual databases have been in existence mainly as user-views. However virtual databases may also be of a "super-view" kind, to provide effective integration of several independent databases. This paper describes a method to construct virtual databases, which yields automatic mapping of queries. Virtual databases are generated by applying a sequence of schema transformations to other databases. The given databases serve as an initial approximation for the desired superview and each individual transformation restructures the current approximation into the next approximation. In addition, each transformation defines a mapping of queries against the new superview into the appropriate sets of queries against the previous superview. The end result is an automatic mapping of queries against the final superview into sets of queries against the initial databases. A prototype database management system that allows users to construct virtual databases and interrogate them has been developed.

1. INTRODUCTION

In general, databases are designed to accomodate specific applications, both current and anticipated. If this design methodology is followed, each of these applications is then fully satisfied, in terms of the data it requires, by the databases. Further adaptation of the database towards each application is provided by user views. Eventually, however, applications may evolve which are no longer satisfiable by this database, their data extending over two or more independent databases. Such applications cannot be provided with a single integrated view of their data.

The responsibility for such undesirable situations cannot always be placed with the designers of the database. Rather, we argue that such situations are due to limitations of the current approach to databases. According to this approach, a database is a collection of data structured in accordance with some schema. Once this schema has

been determined and the data organized, the database is "committed". This database will accomodate (with appropriate user views) all anticipated applications. Many unanticipated applications may be accomodated as well. However, such applications must be satisfiable by the data included in this database.

In this paper we advocate a different approach, which generalizes this concept of databases. We introduce the concept of a virtual database as a database that has a schema, but no data that conforms to this schema. Instead, a mapping is available from this schema into schemas of other databases. In other words, while traditional (physical) databases are schema-data pairs, virtual database are schema-mapping pairs.

For an application which cannot be satisfied by a single database, (and for which the construction of a new physical database is not justified) a virtual database should be defined. It will integrate the schemas of the relevant databases (or portions thereof) into a single global schema (a superview), that will accommodate the new application.

Obviously, user views are one type of virtual databases: rather than integrate a number of databases, they transform a single one. In other words, the problem of database integration includes user views, and the same tools and techniques developed for schema integration could be used to generate user views.

Specifically, three separate tools are involved: the traditional database specification language is augmented by a schema integration language, the processor that compiles data definition statements and generates physical databases is joined by a twin processor to compile schema integration statements and generate virtual databases and, finally, the database management system is modified to handle transactions against all databases in a uniform way.

The schema integration language we employ is actually a set of schema restructuring transformations. With a sequence of such transformations, existing schemas may be integrated to create a new superview. As each transformation restructures the current superview into a new superview, it defines a mapping from this new superview into the previous one. By composing these mappings, when the restructuring sequence ends, a mapping of the final superview into the underlying databases is available. This mapping is stored together with the superview. The database management system handles queries against physical databases in the usual way. To handle queries against virtual databases, the database management system employs the mapping to decompose each query into a set of queries against the component databases. These queries will then be resubmitted against these component databases. The answers will then be recomposed to form an answer to the original query. Consequently, the effect will be as if a physical global database is actually available. Note that virtual databases can be integrated to form new virtual databases. Therefore, query decomposition is a recursive process.

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 PERSON which appears in both databases. Such constraints are checked in 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.

We know of two proposals that relate to this kind of database integration. McLeod and Heimbigner [4] have suggested a "federated database architecture". They note the limitation of current approaches to database distribution and suggest an alternative in which databases that are physically distributed are loosely integrated in a federation: each component database indicates the portion of its structure it is willing to share with other databases, its export schema, and specifies the information it wishes to receive from other databases, its import schema. A federal schema coordinates the different contributions and requests, and provides the common basis for communication among the individual databases. Another proposal, called MULTIBASE [10] outlines an architecture for a software system to integrate heterogeneous databases. In this respect, the latter proposal adopts an approach which is closer to the approach advocated here: achieve integration by imposing a single global view on different component databases. To our knowledge, no attempt has yet been made to investigate in a formal way the various problems which integration presents.

To explore these issues a formal data model is clearly necessary. The model we adopt is the abstract data model [5]. This model combines a functional approach to data modelling (suggested by Sibley and Kershberg [8] and implemented by Shipman [7] and Buneman and Frankel [1]) with aggregation and generalization hierarchies (similar to those described by Smith and Smith [9] and incorporated into the Semantic Data Model by Hammer and McLeod [3]). Without much difficulty, the abstract data model can be used to represent schemas of other data models [6], for example the relational and the network data models. Therefore, by using mechanical schema translators, techniques developed for this model can be applied to databases constructed according to different data models.

The data model and the restructuring transformations were reported in [5]. Since they are at the basis of our methods, a summary is given in the next two sections. We then proceed to define a simple query language for this data model. For each restructuring transformation and for each query type we determine the query translation involved. Given a sequence of restructuring transformations, a mapping of the final superview into the initial schemas is derived. This mapping and the single-step translations obtained earlier provide us with a recursive query translation algorithm. We conclude with a brief discussion of the results.

2. THE DATA MODEL

At the basis of the data model used here is a functional approach. This approach employs notions of data domains and attribute functions: domains are sets of data objects, functions assign objects of one domain to objects of another domain as their attributes.

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 T att S, there is a function f_{ST} : dom(S) -> dom(T),

4. for every two classes S,T, such that T gen S, 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 whioh 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.

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

SS# att FACULTY,	OFFICE att FACULTY,
SS# att STUDENT,	SCHOOL att STUDENT,
SS# att PERSON,	S_NAME att SCHOOL,
NAME att FACULTY,	OFFICE att SCHOOL,
NAME att STUDENT,	PERSON att FACULTY,
NAME att PERSON,	PERSON att 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 olass S in D is
type(S) = {T | T 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 object in 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 $\{T_1,\ldots,T_n\} \subseteq$ type(S). $(T_1,\ldots,T_n)$ key S if

$f : dom(S) \rightarrow dom(T_1) \times \ldots \times dom(T_n)$

$f(x) = (f_{ST_1}(x),\ldots,f_{ST_n}(x))$

is an injection.

Thus, the classes $(T_1,\ldots,T_n)$ constitute a key to class S, if a combination $(x_1,\ldots,x_n)$ 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. The key relation will sometimes be denoted by underlining the participating classes in each type. For example, SS# key FACULTY is denoted by (SS#,NAME,OFFICE).

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 => S att R, $f_{RS} = i_{RT}.f_{TS}$,
   S gen T, T gen R => S gen R, $i_{RS} = i_{RT}.i_{TS}$,

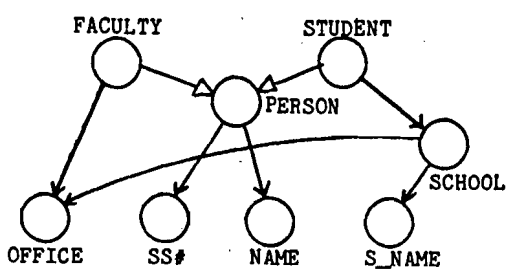
3. $\forall x,y \in dom(S)$:
   $(\forall T \in type(S): f_{ST}(x)=f_{ST}(y)) \Rightarrow x=y$.

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. Inheritance of attributes over generalizations and 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:

- Every class is guaranteed a key,

- Classes related by generalization have the same key.

In many cases it is convenient to represent a database schema by a graph. 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 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:



We shall refer to this model as the abstract data model (ADM). The abstract data model is related to both the relational and the network data models. By allowing only attribute relationships with single

level hierarchies (i.e. classes are either primitive or aggregates of primitive classes) the abstract data model is reduced to the well known relational model. In other words, abstract schemas introduce hierarchies into "flat" relational schemas. Any abstract schema that incorporates only attribute relationships can be converted to a network schema, by inverting the direction of tbe attribute functions (replacing them with one-to-many relationships). Consequently, the abstract data model can express network structures, with the additional benefit of providing a formal framework.

3. THE RESTRUCTURING LANGUAGE

The restructuring language is 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 schemas. We begin by introducing three operators (meet, join and fold) to 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 such a generalization is determined by the properties of their keys. As an example, consider the classes FACULTY = (SS#,NAME,OFFICE) and STUDENT = (SS#,NAME,SCHOOL). Since FACULTY and STUDENT have a common key (SS#), they have a common generalization: the class PERSON = (SS#,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. Denote the common attributes by $R_1,\ldots,R_n$ and the attributes that distinguish S and T by $P_1,\ldots,P_l$ and $Q_1,\ldots,Q_m$, respectively. The transformation meet S and T, is performed by adding a new class, the meet of S and T, denoted by $S \wedge T$, and the relationships $S \wedge T$ gen S, $S \wedge T$ gen T and R_i att $S \wedge T$ $(i=1,\ldots,n)$. The type of $S \wedge T$ is therefore given by $type(S \wedge T) = type(S) \cap type(T)$. The new class is populated with the union of the domains of S and T: $dom(S \wedge T) = dom(S) \cup dom(T)$. The functions from dom(S) and dom(T) into $dom(S \wedge T)$ are defined as identities. The functions from $dom(S \wedge T)$ into the domains of $R_1,\ldots,R_n$ 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 $f_1,\ldots,f_n$ and $g_1,\ldots,g_n$ the attribute functions from S and T, respectively into $R_1,\ldots,R_n$. Let $K = \{R_1,\ldots,R_k\}$. Define functions f and g as follows:

$$f : dom(S) \rightarrow dom(R_1) \times \ldots \times dom(R_k)$$
$$f(x) = (f_1(x),\ldots,f_k(x))$$

$$g : dom(T) \rightarrow dom(R_1) \times \ldots \times dom(R_k)$$
$$g(x) = (g_1(x),\ldots,g_k(x)))$$

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

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

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. Consider again the classes FACULTY = (SS#,NAME,OFFICE) and STUDENT = (SS#,NAME,SCHOOL). The join of STUDENT and FACULTY is the class ASSISTANT = (SS#,NAME,SCHOOL,OFFICE); ASSISTANT is generalized by both FACULTY and STUDENT and its domain includes all those which are 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 $S \vee T$, and the relationships S gen $S \vee T$, T gen $S \vee T$, R_i att $S \vee T$ $(i=1,\ldots,n)$, P_i att $S \vee T$ $(i=1,\ldots,l)$ and Q_i att $S \vee T$ $(i=1,\ldots,m)$. The type of $S \vee T$ is therefore given by type($S \vee T$) = type(S) $\cup$ type(T). The domain of $S \vee T$ is dom($S \vee T$) = dom(S) $\cap$ dom(T). The injections from dom($S \vee T$) into dom(S) and dom(T) are defined as identities. The functions from dom($S \vee T$) into the domains of $R_1,\ldots,R_n$, $P_1,\ldots,P_l$ and $Q_1,\ldots,Q_m$ 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 <not-applicable> 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 <not-applicable> value for objects in dom(T) but not in the image of this injection). Using the same injection, functions 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. meet, join and fold are operators that manipulate the generalization hierarchy of the databases. The next two primitives (aggregate and telescope) manipulate 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)=($T_1,\ldots,T_m,T_{m+1},\ldots,T_n$). The transformation aggregate ($T_1,\ldots,T_m$) of S is performed by adding a new class T and the relationships T att S. Also every relationship T_i att S is replaced with T_i att T. The domain of T is populated with new objects that are tuples of objects from the aggregated domains, that are realized by objects of S: dom(T) = $\{(f_{ST_1}(x),\ldots,f_{ST_m}(x)) \mid x \in dom(S)\}$. The function that supports the relationship between S and T is defined by $f_{ST}(x) = (f_{ST_1}(x),\ldots,f_{ST_m}(x))$. The functions from dom(T) into dom(T_i) ($i=1,\ldots,m$) 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) = ($T_1,\ldots,T_n$), 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 T_i att T with T_i att S. The functions that support the new attribute relationships are simple compositions: $f_{ST_i}(x) = f_{TT_i}(f_{ST}(x))$ ($i=1,\ldots,n$).

With aggregate and telescope, a class may be relocated on the schema. aggregate may also be used to bring a schema into a normal form, in which the non-key attributes of each class are fully dependent on the key. A situation common to loosely integrated schemas is that one class includes an attribute which is also the key attribute of another class. As an example, consider classes ACCOUNT = (ACC#,NAME,BALANCE) and TRANSACTION = (TRANS#,ACC#,AMOUNT).

ACC# is an attribute of TRANSACTION and a key of ACCOUNT. Since each TRANSACTION has its own ACOUNT, rather than an ACC#, by modifying TRANSACTION to (TRANS#,ACOUNT,AMOUNT) a more accurate semantic description is obtained. This transformation, called retraction, may be obtained through a sequence of primitive transformations. (Note that if ACC# were a key of both ACCOUNT and TRANSACTION, a meet of these two classes would have been more appropriate, as there is evidence that they 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 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 the same 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: dom(P)={x}. 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) into dom(P).

3.7 Delete

To remove portions of the database which are not relevant to the application the primitive delete may be used. 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.

4. THE QUERY LANGUAGE

For query language we prefer a small set of basic retrieval operators. This set may then be used as the target of translation of higher level query languages. Because they offer versatility and portability, low level operators are useful in other areas where translation is involved: low level operators have to be implemented only once for every environment, every high level language is then translated into these operators. In our case this approach also facilitates the development of a query mapping technique.

The set of retrieval operators for ADM consists of only two operators. Given the name of any class, a Domain query simply returns the objects in the domain of this class. Assume a basic situation with two classes S and T, where T att S (T may be either an immediate attribute of S or there could be a third class R such that T att R and R gen S). Given an object $s \in dom(S)$, a Function query returns the object $t \in dom(T)$, which is assigned to the object s for the attribute T.

Definition 4: Let S and T be two classes and T att S. The Abstract Data Model retrieval operators are:

Domain: (S) = dom(S)
Function: $T(S=s) = \{t \mid t=f_{ST}(s)\}$.

T and S are called the target and source attribute, respectively, of Function query. Normally, Function queries return singleton sets. There may be, however, four irregular cases:

1. If T is not an attribute of S, the query is rejected at the schema level. In this case the special set <not-applicable> is returned.

2. If T is an attribute of S, but the object supplied is not a member of the domain of S, the query is rejected at the data level. In this case the special set <not-found> is returned.

3. If T is an attribute of S and s is a member of the domain of S, but the value that should be assigned to it is undefined, it is a case of incomplete information. In this case the special set <not-available> is returned.

4. If the result is not a singleton, but a larger set, we have an inconsistency. In this case the special set <not-consistent> is returned. While such occurrences are excluded by the very nature of our data model, they become possible when independent databases are integrated into a superview.

This analysis of "null values" is summarized in the following definition:

$$T(S=s) = \begin{cases} \text{<not-applicable>}, & \text{if } T \notin \text{type}(S) \\ \text{<not-found>}, & \text{if } s \notin \text{dom}(S) \\ \text{<not-available>}, & \text{if } f_{ST}(s) \text{ is undefined} \\ \text{<not-consistent>}, & \text{if } |\{f_{ST}(s)\}| > 1 \\ \{f_{ST}(s)\}, & \text{otherwise} \end{cases}$$

The first four cases in this definition all result from some kind of irregularity. However, these null values differ in the information they provide. This is evident when two Function query answers have to be combined. Assume a query regarding the AGE of a certain PERSON is submitted to two databases. If the two answers are identical AGE values, they are combined to that common value. If, however, the values are different, the combined answer is <not-consistent>. Clearly, the combination of a legitimate AGE value with <not-found> or with <not-available> should be the AGE value. Similarly, <not-found> and <not-available> should be combined into <not-available>, which is more informative. Finally, <not-applicable> is the least informative answer and should be ignored in favor of any other answer. Assume a and b are two different singletons, the definition of this answer-combining operation, denoted $\underline{\vee}$, is tabulated below.

$\underline{\vee}$	<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

Sometimes it may be important that the target class of a query is applicable to its source class in <u>both</u> databases. We will then use a slightly different answer-combining operation, denoted $\overline{\wedge}$, in which <not-applicable> overrides all other answers:

$\overline{\wedge}$	<n-ap>	<n-fd>	<n-av>	<n-co>	a	b
<n-ap>	<n-ap>	<n-ap>	<n-ap>	<n-ap>	<n-ap>	<n-ap>
<n-fd>	<n-ap>	<n-fd>	<n-av>	<n-co>	a	b
<n-av>	<n-ap>	<n-av>	<n-av>	<n-co>	a	b
<n-co>	<n-ap>	<n-co>	<n-co>	<n-co>	<n-co>	<n-co>
a	<n-ap>	a	a	<n-co>	a	<n-co>
b	<n-ap>	b	b	<n-co>	<n-co>	b

As an example of how this basic set of retrieval operators may be used to implement high level queries, consider this database:

```
STUDENT att ENROLLMENT
COURSE att ENROLLMENT
GRADE  att ENROLLMENT
SS#  att STUDENT
NAME  att STUDENT
```

Assume we want a list of the students who received the grade 'A' in the course 'CS100'. In high level query languages, such as DAPLEX or FQL, we may issue:

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

or:

```
!ENROLLMENT.|([[GRADE,'A'].EQ,[COURSE,'CS100'].EQ].AND).*STUDENT.*NAME
```

Using an ALGOL-like host language, these queries are translated into a procedure which accesses the database only with Function and Domain queries:

```
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.
```

To increase efficiency, it may be worthwhile to add other retrieval operators. For example, consider the operator that accepts an object t from dom(T) and returns all the objects $\{s_1,\ldots,s_n\}$ in dom(S), that have t as their value for attribute T. Such an operator will allow the previous query to be handled much more efficiently: with two retrievals, the set of all ENROLMENTS with GRADE='A' and the set of all ENROLMENTs with COURSE='CS100' are obtained. Their intersection provides the set of all ENROLMENTs, whose STUDENT NAMEs should be printed. However, to simplify our query translation techniques a minimal set of operators is preferred.

5. QUERY TRANSLATION OVER PRIMITIVE RESTRUCTURING

Every restructuring primitive transforms a given schema into a new schema. Interrogation of the new schema is accomplished by translating each query against this schema into the appropriate set of queries against the schema that existed prior to the transformation, and then combining the answers to form an answer to the original query. Clearly, only queries that involve new (or modified) classes require translation; all other queries may be submitted to the previous schema without any modifications.

We begin with the primitive meet. meet introduces a new class, which may be involved in two queries: a Domain query of this class and a Function query with this class as source attribute. Consider the previous example. The query (PERSON) is answered with the union of the answers to the queries (STUDENT) and (FACULTY). To answer the query NAME(PERSON=x) we submit both NAME(STUDENT=x) and NAME(FACULTY=x) and combine the answers. The combined answer could be either a valid name or <not-found> or <not-available> or <not-consistent>. Similarly, to answer OFFICE(PERSON=x) we submit OFFICE(STUDENT=x) and OFFICE(FACULTY=x). Since OFFICE is not an attribute of STUDENT, the answer to the first query is <not-applicable>. As meet classes have the intersection of attributes of their creators, OFFICE cannot be an attribute of PERSON and the combined answer should be <not-applicable>. The answer-combining operation that is appropriate here is $\bigwedge$.

Similarly, only two queries are affected by the primitive join. A Domain query, such as (ASSISTANT), is answered with the intersection of the answers to (STUDENT) and (FACULTY). The query NAME(ASSISTANT=x) is answered by a combination of the answers to NAME(STUDENT=x) and NAME(FACULTY=x). The query OFFICE(ASSISTANT=x) is translated into OFFICE(STUDENT=x) and OFFICE(FACULTY=x). Again, OFFICE(STUDENT=x) is <not-applicable>. As join classes have the union of the attributes of their creators, in this case the <not-applicable> value should be ignored. The answer-combining operation $\bigvee$ accomplishes that.

Assume the schema PERSON = (SS#,NAME), STUDENT = (SS#,NAME,SCHOOL) and PERSON gen STUDENT is restructured by the transformation fold STUDENT into PERSON. Denote the new class by PERSON'. There are three possible queries that involve PERSON'. The domain of PERSON' is simply the domain of PERSON. Function queries, however, are slightly more complicated. Queries in which PERSON' is source attribute must be mapped into "alternative" queries: SCHOOL(PERSON'=x) is mapped into SCHOOL(PERSON), which returns <not-applicable>, and SCHOOL(STUDENT=x). The combination $\bigvee$ of both answers produces the correct result. Queries in which PERSON' is target are handled in the same way: to answer PERSON'(ENROLLMENT=x) we submit both PERSON(ENROLLMENT=x) and STUDENT(ENROLLMENT=x) and take the combination $\bigvee$ of the answers.

Let ENROLLMENT = (SS#,NAME,SCHOOL,COURSE,GRADE) and assume we aggregate (SS#,NAME,SCHOOL) into a new class STUDENT. Again, there are three queries to consider. The Domain query (STUDENT) is answered by {(SS#(ENROLLMENT=x),NAME(ENROLLMENT=x),SCHOOL(ENROLLMENT=x)) | x $\in$

(ENROLLMENT)}. The Function query STUDENT(ENROLLMENT=x) is translated into a tuple of queries: (SS#(ENROLLMENT=x),NAME(ENROLLMENT=x), SCHOOL(ENROLLMENT=x)). The answer to a query which mentions STUDENT as source, such as NAME(STUDENT=x), is contained in the argument itself.

Consider now the transformation <u>delete</u>. For example, let SHIPMENT = (<u>S#</u>,DATE,SUPPLIER,CHARGE) and assume SUPPLIER is deleted, resulting in SHIPMENT' = (<u>S#</u>,DATE,CHARGE). The query (SHIPMENT') is answered by (SHIPMENT) and all function queries in which SHIPMENT' is target are translated into queries in which SHIPMENT is target. However, to translate queries that mention SHIPMENT' as source attribute, the target attribute has to be examined: if it is SUPPLIER, the answer is <not-applicable>, otherwise the query is passed forward after changing the source to SHIPMENT.

We define query translations for the other restructuring primitives in a similar way. The complete set of translations required by primitive transformations is summarized below. In each case Q is the name of the class introduced or modified by the transformation and R is any related class. Wherever possible, a Function query S(T=x) is abbreviated S(T).

- <u>meet</u> S <u>and</u> T
  - (Q) = (S) $\cup$ (T)
  - Q(R) = <not-applicable>
  - R(Q) = R(S) $\overline{\wedge}$ R(T)

- <u>join</u> S <u>and</u> T
  - (Q) = (S) $\cap$ (T)
  - Q(R) = <not-applicable>
  - R(Q) = R(S) $\underline{\vee}$ R(T).

- <u>fold</u> S <u>into</u> T
  - (Q) = (T)
  - Q(R) = T(R) $\underline{\vee}$ S(R)
  - R(Q) = R(T) $\underline{\vee}$ R(S)

- <u>aggregate</u> $(T_1,\ldots,T_n)$ <u>of</u> S
  - (Q) = $\{(T_1(x),\ldots,T_n(x)) \mid x \in (S)\}$
  - Q(R) = if R=S then $(T_1(S),\ldots,T_n(S))$ else <not-applicable>
  - R(Q=$(x_1,\ldots,x_n)$) = if R=T_i then $\{x_i\}$ else <not-applicable>

- <u>telescope</u> T <u>into</u> S
  - (Q) = (S)
  - Q(R) = S(R)
  - R(Q) = R(T(S)) $\underline{\vee}$ R(S)

- add T_t to S
  (Q) = (S)
  Q(R) = S(R)
  R(Q) = if R≠T then R(S) else
  if x ∈ (S) then {t} else <not-found>

- delete T from S
  (Q) = (S)
  Q(R) = S(R)
  R(Q) = if R=T then <not-applicable> else R(S)

Equipped with these restructuring primitives and the query translations they impose, we now proceed to describe how virtual databases are generated and interrogated.

6. VIRTUAL DATABASES AND THEIR INTERROGATION

A virtual database consists of a schema and mapping. The schema is the result of an integration process and the mapping is a by-product of this process.

An integration process 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 tbe same real world entity. Thus, two classes describing Social Security numbers may probably be combined, but the employee number in two different organizations may indicate two independent sequencings, without any global significance. The latter does not create problems, unless these classes participate in keys: identical objects (i.e. the same employee in both organizations) could then 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.

Assume this integration technique is applied to databases D_1 and D_2. Denote by V_0 the initial schema (which is actually the disconnected schemas of D_1 and D_2). Let $\theta_1,\ldots,\theta_n$ be the sequence of restructuring primitives applied to V_0. Let V_i be the schema obtained from the application of θ_i to V_{i-1}. The final superview is V_n.

The overall database mapping from V_n into D_1 and D_2 can be preserved in different forms. One method is to choose the class names of V_i to reflect the transformation θi. For example, after aggregate $(T_1,\ldots,T_n)$ of S the new class is named $(T_1,\ldots,T_n)$*S (other names are left unchanged), after telescope T into S the class S is renamed S:T, after add T_t to S the class S is renamed S+T_t, after delete T from S the class S is renamed S-T, etc. Since a class that is assigned a name in a current transformation, may still be renamed in future

transformations, as the restructuring progresses names become more and more compound. For example, assume SUPPLIER and CUSTOMER were generalized to express the class of business acquaintances, from which the attribute TEL# was later deleted. The final name of this class will be (SUPPLIER $\wedge$ CUSTOMER)-TEL#. Consequently, each class name in the final schema records the origins of this class. In other words, the final superview incorporates (in the names of its classes) the mapping to the original schemas.

Consider a query which is submitted to V_n. To translate this query, we take a step by step approach. First we translate it to a set of queries that should be directed at schema V_{n-1}. These queries are in turn translated into queries which accomodate V_{n-2}. The process continues, until a set of queries is obtained, that accomodates the initial schema V_0. These final queries are actually submitted to the original databases D_1 and D_2. The answers are passed back in the reverse direction. In each step they are combined to form answers to their initiating queries from the previous step, until an answer is formed to the original query. This recursive process of query decomposition and answer recomposition involves only primitive translations of the type treated in Section 5. Actually, many of the query decomposition steps leave a particular query unchanged, as the corresponding restructuring steps did not affect classes involved in this query. Therefore, rather than the complete restructuring sequence directing the query translation process, the "local history", as recorded in each class name, provides us with a sub-sequence of $\{\Theta_i\}$, which is sufficient to translate this query.

As an example, assume two independent libraries maintain their collections in two slightly different databases

Database$_1$: BOOK = (<u>BOOK#</u>,COVER,NO_COPIES)
COVER = (TITLE,AUTHOR)
Database$_2$: BOOK = (<u>BOOK#</u>,TITLE,SOURCE)
SOURCE = (PUBLISHER,PRICE)

and assume the following superview is desired

Superview: BOOK = (<u>BOOK#</u>,COVER)
COVER = (TITLE,AUTHOR,PUBLISHER)

To arrive at this superview, the following actions may be taken. In database$_1$, NO_COPIES is deleted from BOOK and a new class PUBLISHER is added to COVER. In database$_2$, PRICE is deleted from SOURCE and then SOURCE is telescoped into BOOK, AUTHOR is added to BOOK and then TITLE, AUTHOR and PUBLISHER are aggregated into COVER. (All additions are with <not-available> values). The two databases now have identical structures. They are overlapped in a bottom-up process that combines identical classes. Two identical classes T_1 and T_2 may be combined with <u>meet</u> followed by two <u>folds</u>. However, for brevity, we consider it a single operation <u>combine</u> T_1 <u>with</u> T_2 (the combined class is denoted $T_1.T_2$). First, primitive classes from the two databases with identical names (BOOK#, TITLE, AUTHOR and PUBLISHER) are combined. Next, the two

classes COVER are combined and, finally, the two classes BOOK are combined. Using intermediate names B_1, B_2, C_1 and C_2, the origins of the superview classes are:

```
B1 ::   BOOK1-NO_COPIES1
B2 ::   (BOOK2:(SOURCE2-PRICE2))+AUTHOR2
C1 ::   COVER1+PUBLISHER1
C2 ::   (TITLE2,AUTHOR2,PUBLISHER2)*B2
BOOK# ::        BOOK#1 .BOOK#2
TITLE ::        TITLE1.TITLE2
AUTHOR ::       AUTHOR1.AUTHOR2
PUBLISHER ::    PUBLISHER1.PUBLISHER2
COVER ::        C1.C2
BOOK :: B1.B2
```

where class names where suffixed by either 1 or 2 to identify their database. Assume we want the AUTHOR of a particular BOOK b. We submit to the superview the query AUTHOR(COVER(BOOK=b)). It is translated as follows:

```
  COVER(BOOK=b)
= COVER((B1.B2)=b)
= COVER(B1=b) ⊻ COVER(B2=b)
= (C1.C2)(B1=b) ⊻ (C1.C2)(B2=b)
= C1(B1=b) ⊻ C2(B1=b) ⊻ C1(B2=b) ⊻ C2(B2=b)
= C1(B1=b) ⊻ <not-applicable> ⊻ <not-applicable> ⊻ C2(B2=b)
= C1(B1=b) ⊻ C2(B2=b)
= (COVER1+PUBLISHER1)((BOOK1-NO_COPIES1)=b) ⊻
  ((TITLE2,AUTHOR2,PUBLISHER2)*B2)(B2=b)
= COVER1((BOOK1-NO_COPIES1)=b) ⊻
  (TITLE2(B2=b),AUTHOR2(B2=b),PUBLISHER2(B2=b))
= COVER1(BOOK1=b) ⊻
  (TITLE2(((BOOK2:(SOURCE2-PRICE2))+AUTHOR2)=b),
   AUTHOR2(((BOOK2:(SOURCE2-PRICE2))+AUTHOR2)=b),
   PUBLISHER2(((BOOK2:(SOURCE2-PRICE2))+AUTHOR2)=b))
= COVER1(BOOK1=b) ⊻
  (TITLE2((BOOK2:(SOURCE2-PRICE2))=b),
   {if b ∈ (BOOK2:(SOURCE2-PRICE2)) then <not-available> else
   <not-found>},
   PUBLISHER2((BOOK2:(SOURCE2-PRICE2))=b))
= COVER1(BOOK1=b) ⊻
  (TITLE2(SOURCE2(BOOK2=b)) ⊻ TITLE2(BOOK2=b),
   {if b ∈ (BOOK2) then <not-available> else <not found>},
   PUBLISHER2(SOURCE2(BOOK2=b)) ⊻ PUBLISHER2(BOOK2=b))
= COVER1(BOOK1=b) ⊻
  (<not-applicable> ⊻ TITLE2(BOOK2=b),
   {if b ∈ (BOOK2) then <not-available> else <not-found>},
   PUBLISHER2(SOURCE2(BOOK2=b)) ⊻ <not-applicable>)
= COVER1(BOOK1=b) ⊻
  (TITLE2(BOOK2=b),
   {if b ∈ (BOOK2) then <not-available> else <not-found>},
   PUBLISHER2(SOURCE2(BOOK2=b)))
```

The first query is submitted to $Database_1$: let c_1 be the COVER returned. The expression that follows involves several queries that are submitted to $Database_2$: let t_2 and p_2 be the TITLE and PUBLISHER returned. Then

$$
\begin{aligned}
& COVER(BOOK=b) \\
= \; & c_1 \underline{\vee} \{\text{if } b \in (BOOK_2) \text{ then } (t_2, \langle\text{not-available}\rangle, p_2) \text{ else } \langle\text{not-found}\rangle\} \\
= \; & c_1 \underline{\vee} c_2 \\
= \; & c
\end{aligned}
$$

To obtain the AUTHOR requested in tbe original query, the follow-up query AUTHOR(COVER=c) is submitted to the superview. A similar translation process decomposes this query into $AUTHOR_1(COVER_1=c_1)$ which is submitted to $Database_1$, and $AUTHOR_2(C_2=c_2)$, whose value is the second component of c_2. Finally, the answer to the original query is:

$$
\begin{aligned}
& AUTHOR(COVER(BOOK=b)) \\
= \; & AUTHOR_1(COVER_1=c_1) \underline{\vee} \\
& \{\text{if } b \in (BOOK_2) \text{ then } \langle\text{not-available}\rangle \text{ else } \langle\text{not-found}\rangle\} \\
= \; & AUTHOR_1(COVER_1(BOOK_1=b)) \underline{\vee} \\
& \{\text{if } b \in (BOOK_2) \text{ then } \langle\text{not-available}\rangle \text{ else } \langle\text{not-found}\rangle\}
\end{aligned}
$$

Obviously, if AUTHOR a_1 is returned from $Database_1$, it will become the answer. However, if BOOK b is not in $Database_1$ but appears in $Database_2$, the integrated answer will be <not-available>, indicating the book exists, but the author name is not available. By comparison, if we present the original query to $Database_1$, the answer will be <not-found> and if we present it to $Database_2$, the answer will be <not-applicable>. Clearly, the superview answer is more informative, indicating proper integration has taken place.

7. CONCLUSION

A prototype database management system that allows the definition and interrogation of virtual databases was implemented and cases similar to the previous example have been tested. As that example demonstrated, there are considerable opportunities for query optimization. For example, many sub-queries may be evaluated to <not-applicable> at an early stage rather than wait until final "physical queries" are obtained (for brevity, this was actually done for $C_1(B_2=b)$). In particular, it is desirable to minimize "shuttling" between databases, as this contributes heavily to the time required for query processing. Note that, as long as query processing is conducted within a database, objects are identified by some internal representation. When an object has to be communicated back to the superview, it must be done by an external representation, such as a structured key. In the example, to obtain AUTHOR(COVER(BOOK=b)), $Database_1$ was first interrogated on $COVER_1(BOOK_1=b)$, producing the answer c_1, and then on $AUTHOR_1(COVER_1=c_1)$, producing the answer a_1. Since c_1 is not a primitive object it must be communicated back and

forth through the appropriate combination of primitive objects. With proper optimization techniques, the need to communicate c_1 outside $Database_1$ may be eliminated.

While the methods described here assume a particular data model, we have shown elsewhere [6] how relational and network schemas can be translated into abstract schemas. With such mechanical schema translators, our methods can be used to integrate databases which were constructed according to different data models.

We have shown how database restructuring statements are used to obtain a mapping from the virtual database onto the physical database. We estimate that these statements may also be used to generate the appropriate transfer programs that will actually construct a new physical database, while the schema of the original database is retained as a view. In other words, given a physical database D, virtual restructuring produced a virtual database D' and a mapping from D' into D; view-retaining restructuring will produce a physical database D' and a virtual database D with a mapping from D into D'. While virtual restructuring allows users to define new views, and write new applications for these views, without any modifications to existing structures, view-retaining restructuring will enable databases to be physically restructured, without voiding old applications. An effort in this direction, albeit using different means, was described in [2].

REFERENCES

[1] P. Buneman and R.E. Frankel. FQL - a Functional Query Language. In Proceedings of ACM-SIGMOD International Conference on the Management of Data, pages 52-57. Boston, Massachussetts, 1979.

[2] R. Gerritsen and H.L. Morgan. Dynamic Restructuring of Databases with Generation Data Structures. In Proceedings of ACM Annual Conference, pages 281-286. Houston, Texas, 1976.

[3] M. Hammer and D. McLeod. A Semantic Data Model: a Modelling Mechanism for Database Applications. In Proceedings of ACM-SIGMOD International Conference Management of Data, pages 26-35. Austin, Texas, 1978.

[4] D. McLeod and D. Heimbigner. A Federated Architecture for Database Systems. In Proceedings of AFIPS National Computer Conference, pages 283-289. Anaheim, California, 1980.

[5] A. Motro and P. Buneman. Constructing Superviews. In Proceedings of ACM-SIGMOD International Conference on Management of Data, pages 56-64. Ann Arbor, Michigan, 1981.

[6] A. Motro. Virtual Merging of Databases. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1981.

[7] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. ACM Transactions on Database Systems 6(1):140-173, March 1981.

[8] E.H. Sibley and L. Kershberg. Data Architecture and Data Model Considerations. In Proceedings of AFIPS National Computer Conference. Pages 85-96. Dallas, Texas, 1977.

[9] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. ACM Transactions on Database Systems 2(2):105-133, June 1977.

[10] J.M. Smith et al. MULTIBASE - Integrating Heterogeneous Distributed Database Systems. In Proceedings of AFIPS National Computer Conference. pages 487-499. Chicago, Illinois, 1981.