Object identification in multidatabase systems

[10/16/92]

William Kent, Rafi Ahmed, Joseph Albert, Mohammad Ketabchi, Ming-Chien Shan

**Pegasus Project, Database Technology Department,
Hewlett-Packard Laboratories, Palo Alto, California, USA**

Abstract

In the Pegasus prototype multidatabase system, the key to the problem of object identity is a differentiation between the view of a data administrator, capturing all the underlying discrepancies and their solutions, and the view of an end user or application, in which only a consistent result is visible. New language constructs with which the administrator can describe solutions to identity problems include underlying and unifying types, the Image function, and producer types.

1 INTRODUCTION

Object identity in a multidatabase system [5][2][20] can be very confusing if we aren't clear about what objects there are to identify, and how many.

The data administrator's (schema definer's) view is much more complicated than the end user's view. The data administrator is aware of discrepancies and inconsistencies in the underlying data, such as the same student occurring in different databases with different birthdates, or a bewildering array of course titles used in various schools. Semantic integration involves the construction of a coherent view for the end user with discrepancies reconciled, e.g., one birthdate per student and a unified[1] set of course titles that might be standardized at the nationwide level.

Confusion is compounded, and anomalies are introduced, by trying to insist that one student is the same object everywhere, particularly if there might be inconsistent data about the student in various databases [15]. A more consistent general model treats students in distinct databases (possibly for the same or different schools) as distinct objects, with yet another object representing the student in the end user's coherent view, together with appropriate mappings among them.

1. The usage of the terms "unified", "unifier", and "unification" in this paper is unrelated to their usage in deductive database theory.

Object identity in this context becomes a more complex three-stage phenomenon. It is necessary to (a) manage identity for the underlying objects reflecting the data in the external data sources, (b) manage identity for the objects in the end user's unified view, and (c) maintain mappings between the two. Simpler cases are accommodated by letting the objects in the end user's view be the same as in the administrator's view when that's appropriate.

Syntactic discrepancies are first eliminated by "importing" the underlying data into a common model. The model adopted for the Pegasus project at Hewlett-Packard Laboratories [1]-[4] is an extension of the functional object model supported in OSQL for the Iris research project and the OpenODB object-oriented database system [8][9].

The imported view provides uniform access to multidatabase data. This object-model view is a simple union of the data imported from the underlying sources without modifications for removing discrepancies or inconsistencies. If there are no significant anomalies, this imported view can be exposed directly to end users without further integration. Otherwise, integration problems as well as their solutions can be described within the common model, thus supporting the two viewpoints of the data administrator and the end user.

This paper develops a framework of underlying and unifying types for managing identity which are being investigated in the Pegasus project. The interdependence between object identity and object existence is developed. The work focuses on tools needed by an administrator to express solutions to identity problems, not with methodologies for discovering solutions, nor with resolution of data discrepancies. The paper deals only with object identity and not such related matters as shallow or deep equality [17].

It was our experience that issues concerning object identity in multidatabase systems frequently reduced to issues about object identity in general. Therefore Section 2 begins with a review of the basic premises assumed for object identity. Section 3 and Section 4 then describe the treatment of identity during import and integration, respectively. Future extensions are explored in Section 5, and Section 6 closes with conclusions.

2 PREMISES

2.1 Scope: What There Is To Identify

A multidatabase system is presented to its users via a single system interface in a *home* database management system (dbms), through which all data made available by *external data sources* is provided in a uniform way, together with data managed locally at the home system. It is conceptually possible to enumerate all the objects known at this interface. The interface thus establishes a scope of availability and uniqueness: identification must be unique within the set of objects known at this interface [16].

Some users might only have access to restricted subsets of the information at the interface. In particular, for the integration paradigm, an administrator might see

a large set of disparate objects, while end users only see a smaller set of integrated objects.

A crucial distinction between single and multidatabase systems is the degree to which the system manages the state of the database. In a single database, nothing happens without the knowledge of the dbms: it is truly *managing* the database. This enables the dbms to fully enforce constraints, and to be aware of the creation and destruction of objects. Unique identifiers can be assigned as needed, and referential integrity can be managed. In contrast, the home system in a multidatabase system must cope with *unmanaged* operations. Successive queries might return 1000, 900, and 1100 employees, without the system noticing any intervening database operations. The same employee might periodically disappear and reappear. The state of the database changes, and objects are created and destroyed, essentially behind the home system's back. The best the system can do is to check once in a while to take stock of the situation. In effect, objects may not be known to the home system until some time after they are externally created, and may remain known to the home system long after they are externally destroyed.

2.2 Consistency of Identity

Identity is consistent within a home system; objects which are distinct as instances of subtypes can't be the same object as one instance of a supertype. We don't adopt ambivalent viewpoints of an application domain. If the sales clerk is also the night watchman, some observers might insist that these are still two employees. We can't model the two employee objects as being the same as one person object; we need to be clear and consistent regarding the number of objects we are talking about. It doesn't matter whether the person has the same or different employee numbers, either with the same or different employers. If we model all the concepts here by saying that there are two employee objects and one person object, with some sort of correspondence among them, then Employee can't be a subtype of Person — that would mean the employee objects *are* person objects. While there may be some other correspondence between these types, they are not related as subtype and supertype. A supertype includes the union of its subtypes, in the sense of a set-theoretical union of the *same members* that exist in the subtypes. If we do want Employee to be a subtype of Person, then we must model that person as one object, even if he has two jobs — and even if he has two employee numbers!

Pegasus currently supports a single consistent "viewpoint" for all end users of a home system. Techniques for supporting multiple viewpoints are being considered [Section 5.3].

2.3 Object Identifiers

Although identity can be abstractly modeled in terms of an identity predicate, most object systems support identity simply via comparison of certain values, designated *object identifiers (oid's)*. A *reference* can then be defined as any expression which evaluates to an oid.

An oid must always be *unique* within the scope of a home system, i.e., two objects cannot be represented by the same oid within the system. In the strictest formulation, an oid is *immutable* (an object is represented by the same oid throughout its lifetime) and *singular* (two oid's cannot represent the same object). Two references are to the same object if and only if their oid values are equal.

Immutability can be relaxed so long as an object is represented everywhere in a given system by the same oid at any point in time. Changing an object's oid requires *system-wide oid replacement*: replacing all occurrences of the old oid in the system with the new one. This may be feasible if it is known that no oid's are stored, or if it is easy to find all the stored oid's. System-wide oid replacement is not feasible if applications may legitimately store oid's outside the control of the system. System-wide oid replacement also allows identity merging, whereby two objects become one. The system will, however, have no recollection that these were once distinct objects.

Singularity could be relaxed to allow an object to be represented by several oid's (synonyms), provided that *oid equivalence* is strictly observed: all functions must behave the same when applied to any of the equivalent oid's, and all but one must be eliminated from any context which excludes duplicates. Object equality can no longer be based on simple oid value matching, but must instead involve an equivalence predicate. This typically means that a table of synonymous oid's has to be maintained somewhere. No support is planned in Pegasus for synonymous oid's.

Literal values can be thought of as objects whose oid's contain recognizable representations of themselves. Non-literal objects typically have arbitrary oid's, assigned when the objects are created. Literals and non-literals need some consistent form of identification, since both can occur as arguments and results of functions.

Literal values in different data types often have the same representations, and may also coincide with arbitrary system-generated identifiers. An object identifier thus needs to include some sort of qualifier to differentiate the various data types and the system-generated identifiers. Hence an object identifier is logically decomposable into *prefix* and *suffix* components; no specific format or implementation is implied. The prefixes don't necessarily have to be physically encoded in the identifier, provided they can be reliably deduced from the context in which they are used.

For literals, the prefix differentiates the data types, while the suffix contains the actual representation. For non-literals, the prefix primarily establishes that these are not literals; their prefixes could be further refined to differentiate oid's generated in different databases, and also to label property-based identifiers, e.g., as used in producer types [Section 3].

Fixed-length implementations of oid's create some difficulties. It limits the set of literals which can actually be represented. It complicates the identification of extensional aggregates such as sets, whose oid's should logically include the oid's of their members. It can make it difficult to use oid's generated in other object

systems. And it can limit the values which can be used for property-based identifiers.

Object identifiers can be *system-assigned* or *algorithmic*. A system-assigned identifier is an arbitrary suffix value assigned to an object at the time it is created; uniqueness is typically guaranteed by some centralized agency in the system. The difficulty with system-assigned identifiers in a multidatabase system is that there is no such central agency guaranteeing system-wide uniqueness. Oid's originating in external systems must be mapped into oid's unique within the home system.

Algorithmic identifiers are derivable in terms of some known characteristics of the object. They do not require a creation event to trigger assignment of an identifier, although algorithmic identifiers could also be used when an object is created. For example, a unique identifier could be derived from an object's social security number if it has such a number at the time of creation and the value is not allowed to be updated. Literals essentially have algorithmic identifiers, whereby the suffixes are defined by the representations of the literal values. Most other algorithmic identifiers are property-based.

2.4 Entity And Proxy Objects

Multidatabase systems force us to recognize a difference between what we consider to be the same object and what the system is treating as the same object [15].

Databases contain *proxy* objects to represent the *entity* objects we think about. Single databases are carefully designed to maintain a one-to-one correspondence between these, so we really don't have to think much about the difference. In a given database, there's only one thing (object, tuple, record, or whatever) serving as proxy for a given employee. It can only have one birthdate and one salary, and no two of them can have the same employee number or social security number. Thus the proxy object has the look and feel of the entity object.

Not so in multiple databases. An employee might have different birthdates or salaries in different databases, and a given social security number might be paired with different employee numbers in different databases. The appropriate constraints simply aren't enforced across multiple databases. Anomalies arise if we try to insist that these are all the same object. While we may think of them as the same entity, the system is treating them as distinct proxy objects.

The administrator's job is to create a coherent view for the end user which again contains only one proxy object for each entity object. This proxy may sometimes be a new one, distinct from the others, which don't even appear in the end user's view. We need to preserve the identities of all these proxy objects, and relate them appropriately to each other.

Things like employees — and people, ships, cars, stars, departments, companies — are naturally discrete, giving rise to *consistent* populations in different databases. That is, though different databases might contain different instances, possibly overlapping, they all seem to contain subsets of the same coherent set.

A more extreme problem arises when we don't exactly have even the same sorts of populations in the different databases, giving rise to a form of domain mismatch. Some things are relatively amorphous, being arbitrarily partitioned in different ways in different databases. The jobs in one database might include secretaries and file clerks, while another database has administrative assistants, typists, and receptionists. Colors in one database might include red, pink, white, and blue, while another might have coral, carmine, scarlet, blue, and aqua. The same might happen with chemical compounds, medicines, illnesses, skills, school subjects, courses, news categories, and many other things. Populations of such things can be *inconsistent* across databases, not being subsets of one coherent set. Correspondences among such instances in different databases are not simple or obvious.

The problem here is how to show a consistent set of proxy objects to an end user, and how these and all the underlying proxies should be identified and related to each other. In a typical solution, the administrator creates an arbitrary set of instances for the end-user's view, and then define mappings to these from the underlying objects.

2.5 How Objects Become Known

The common paradigm for generating object identifiers is to have them assigned by the system at the time an object is created. This implies that the system knows when an object is created — which isn't always the case. Literals, for example, are never created; there are no creation events at which the system assigns identifiers to individual literals. One could hypothesize a mythical genesis at which all literals were created, but such mythical creation events serve little practical purpose. We thus have two modes by which objects become known to the system: they either have an *eternal* existence, or they are *created* by explicit creation requests to the system. Eternal objects have algorithmic identifiers. The identity of a created object is inherently tied up with its creation event: each creation event creates a distinct object, distinct from any other object. The system should be prepared to provide an identifier unique to the particular creation event, though under certain conditions such an identifier could be based on other information known about the object at the time of its creation.

A third mode of existence which has received little attention in single systems becomes important in multidatabase systems. An *ephemeral object* exists only while some associated rule is satisfied. Such objects could exist in discontinuous time periods as the rule periodically becomes true or false. Hence a distinct creation event cannot be associated with an ephemeral object, and its identifier must necessarily be algorithmic, based on information related to the existence rule.

Constructed aggregates are an example of ephemeral objects. If there is a set constructor {}, then the set {x,y} exists exactly whenever the objects x and y exist. A mention of such a set does not constitute creation; like literals, such objects may be mentioned many times without any definite sense that one such mention is a creation event. In theory, since the identity of such a set is determined by its instances, the algorithmic identifier for the set could be defined as the concatena-

tion of identifiers of its members in some canonical order. In practice, length limitations on identifiers make it difficult to correctly implement such semantics.

Other forms of ephemeral objects could be useful in single database systems. For example, an audit object might exist just while a department's expenditures exceeded its budget. Its identifier could be derived syntactically from the department's identifier. Similarly, the set of skill objects could be defined to correspond to whatever the current values are for a Skills attributes of employees; assigning a previously unknown skill to an employee could automatically create the corresponding object. The identifier is likely to be derived from the name of the skill.

Objects imported from external data sources in a multidatabase system will generally be ephemeral objects. For example, a definition might specify that an employee object exists for each employee number occurring in one or more columns of some underlying relational database [Section 3].

3 IMPORTING OBJECTS: PRODUCER TYPES

The first stage of multidatabase access in Pegasus *imports* each external data source by mapping its schema semantically into a Pegasus schema. This first level of mapping provides access to multidatabase data via a common data model. This object-model view is a simple union of the data imported from the underlying sources without modifications for removing discrepancies or inconsistencies. If there are no significant anomalies (i.e., there is still just one proxy object for each entity object), then this imported view can be exposed directly to end users without further integration.

Existence and identity of imported objects is established by means of producer types. (Producer types are also used in the integration phase [Section 4.2].)

A *producer type* defines the existence of its instances according to a rule based on some literal-valued property. (Oid's in an object-oriented external data source are treated as literals from the viewpoint of the home system; they are not recognized as oid's of the home system.) The rule, called a *producer expression,* defines a possibly time-varying set of property values called a *producer set*. The producer type has one instance for each value in the producer set, but the values in the producer set are not themselves the instances of the producer type. For example, the producer expression may yield a set of employee numbers as its producer set, but the instances of the producer type would be employee objects, not employee numbers.

A producer type differs from a derived type in that a derived type selects its instances from *existing* instances based on some criterion, while a producer type actually *generates* its instances.

If a given value disappears from the producer set as conditions change, the corresponding instance of the producer type no longer exists. If the value reappears in the producer set, the same object resumes its existence in the producer type. The property chosen as the basis of the producer rule should be one which is considered

stable (invariant) for the entities being identified. If this property value changes, this will appear to be a different object to the Pegasus system.

Oid's are generated for instances of producer types. Oid's generated for instances of two producer types will be distinct, even if the same value occurs in their producer sets, unless the producer types are specified to share oid forms. This might arise, for example, if it is known that one producer type is a subtype of another producer type, perhaps because they have been imported from relations for which an inclusion dependency is known [4]. In that case, a common value occurring in both producer sets will correspond to the same oid, i.e., an object which is an instance of both types. Thus it makes sense for producer types which share oid forms to be based on the same property.

A plausible implementation which captures these semantics would construct an oid by concatenating a *suffix* value from the producer set with a *prefix* unique to the producer type; overlapping producer types sharing an oid form would use the same prefix.

A typical producer type for import might be defined as

```
CREATE PRODUCER TYPE EDB.Student
FROM EDB
PRODUCING BY SQL(SELECT StudID FROM Students)
    FUNCTIONS (
    StudentID Integer AS IDENTIFIER;
    SSNum Integer AS MAP TO Students.SSNo;);
```

The type is named EDB.Student, and is imported from a database named EDB. The producer expression

```
SELECT StudID FROM Students
```

is specified in SQL for this mapping. It defines the producer set of character strings, consisting of values in the StudID attribute of a Students relation in the external database, evaluated whenever this type is referenced. There will be one instance of EDB.Student for each distinct value in this set.

The *identifying function* StudentID (which needn't have the same name as the external attribute), takes as argument the oid of one EDB.Student and returns the value of his StudentID. That value is a member of the producer set. Under the plausible implementation, StudentID(x) could be evaluated by simply extracting the suffix value from the oid of x.

SSNum illustrates the mapping of other functions to the external data source [3][4].

4 INTEGRATION

Further integration is required whenever instances of imported types are not semantically distinct, i.e., there may be more than one proxy object representing the same entity object. The purposes of integration are to provide end users with a view containing (1) only one proxy object for each entity, and (2) data associated

with such objects, after reconciling discrepancies that may exist in the underlying data. This paper deals only with the first aspect.

The principle mechanism is the *Image* function, which maps one or more proxy objects into a single proxy object, all intended to represent the same entity. The intent is that the end user only sees image objects. Formally, objects x and y are *semantically equivalent* if Image(x)=Image(y). The image might be one of x or y, or a totally different object. The default assumption is that there are no identity anomalies, each object is its own image, and so Image(x)=x is the default definition of the function.

The corresponding notion at the type level is that of *underlying* and *unifying* types, with the intent that only unifying types are visible in the end user's view. Every type has exactly one unifying type, given by *Unifier(t)*, being itself by default: Unifier(t)=t. An underlying type is a type which is not its own unifying type: Unifier(t)≠t.

Whenever one type is specified as the unifying type for another, the Image function may optionally be redefined (overloaded) between them. If x is an instance of t, then Image(x) is an instance of the unifying type of t: $x \in t \Rightarrow Image(x) \in Unifier(t)$.

Figure 1 shows Student as a unifying type with WDB.Student and EDB.Student as underlying types.

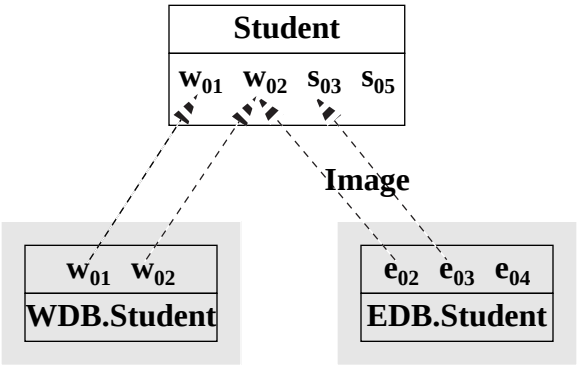

Figure 1. Underlying and Unifying Types.

When the instances of one or more of the underlying types are semantically distinct among themselves, the oid's of their instances could be allowed to occur in the unifying type, as illustrated for WDB.Student. The Image function need not be redefined for these underlying types. In this case the instances of such an underlying type would be visible to the end user, even if the underlying type itself is hidden.

In some cases, Image(x) may be null, as with e_{04} in Figure 1. (If Image(x) becomes null, it does *not* reacquire Image(x)=x as a default value.) When not null, if x is an instance of several underlying types on which Image is redefined, then Image(x) must return the same value in each case. Thus if Image was redefined on EDB.Student and also on EDB.Teacher, and some object was an instance of both, it must have the same image in both cases.

The inverse of the Image function generally yields a set, being all the things having the same image. There might be no inverse, i.e., an instance of a unifying type might not be the image of any instance of any underlying type, as with s_{05} in Figure 1. The Image function need not be unique-valued within a type. Two distinct instances of the *same* type may have the same image, e.g., two colors (coral and crimson) might map into the same unifying color (red), or two jobs (typist and file clerk) might map into the same unifying job (secretary), or two courses at the same university might map into the same unifying course.

The Image function presumes that the mapping is independent of the context in which it is used. This assumption doesn't always hold. The conversion from letter grades to numeric grades might be different for graduate and undergraduate courses. Whether a typist is to be viewed as a secretary or administrative assistant might depend on length of time in the job. Such cases are not handled by the general Image function, but within the specific function in which they are used, such as the functions which return a student's grade in a course, or an employee's job.

In general, two things have to be defined for a unifying type: how its instances are maintained, and how underlying types map into it. Its instances might either be maintained independently or defined by algorithm. In the first case the unifying type is defined as an ordinary type [Section 4.1]; the second case is supported by an alternate usage of producer types [Section 4.2].

4.1 Independently Maintained Unifying Types

One example of an independently maintained unifying type arises if a university is only concerned with data about its own students, but wants to integrate data about them from various other universities they may have attended. The underlying types would reflect data imported from the databases of other universities, as well as the locally maintained data about their own students. The unifying type would have its instances defined as the local student body. Students in the underlying types which did not attend this university would not have any image.

Another example might arise with respect to courses. The unifying courses might be predefined as the local university's own courses, or a national standard such as the ACM/IEEE Computer Science curriculum. The courses of the underlying university databases then have to be mapped into these. A course given by a university might have no image if it had no counterpart in the standard curriculum. Conversely, if a course in the standard curriculum was not offered by any university, then it would not be the image of any underlying object.

There are three modes by which the various underlying types might map into the unifying type via the Image function:

- Instances might preserve their identity, retaining the default definition Image(x)=x. This would happen with the students of the local university, or when the courses of the local university are the same as the unifying courses (or a subset thereof).

- There might be an algorithmic mapping which can be specified as a derived function. Underlying students might map into unifying students having the same social security number. Underlying courses might map into unifying courses which use the same textbook. The algorithm might be incomplete, yielding no image for some underlying objects.
- Manually maintained mappings, under which the image of an underlying object must be explicitly asserted. This can be assisted by various tools and interactive dialog managers.

These are illustrated in the following example:

```
ADD UNDERLYING TYPES WDB.Student, EDB.Student, NDB.Student, SDB.Student
UNDER UNIFYING TYPE Student
(EDB.Student.Image AS STORED)
(NDB.Student.Image(x) AS OSQL SELECT s FOR EACH Student s WHERE SSN(s)=SSN(x))
(SDB.Student.Image(x) AS OSQL SELECT s FOR EACH Student s WHERE PPN(s)=PPN(x));
```

This command establishes WDB.Student, EDB.Student, NDB.Student, and SDB.Student as underlying types of the unifying type Student. These types all existed previously. The Image function is redefined for the last three.

Since Image is not redefined for WDB.Student, the Image function retains its previous definition for instances of WDB.Student. This would usually be the default definition Image(x)=x, allowing the instances of WDB.Student to become instances of Student. In this case, although WDB.Student is hidden from the end user as an underlying type, its instances remain visible. In fact, WDB.Student behaves like a subtype, in the sense that anything which becomes an instance of this underlying type becomes an instance of the unifying type.

Since Image is redefined as a stored function for EDB.Student, the image of an x in EDB.Student must be asserted; it will be undefined (null) until a corresponding instance of Student is assigned. The Image function for NDB.Student is defined on the basis of social security numbers (SSN). The image of an instance of NDB.Student is that instance of Student having the same social security number; there might be none. Similarly, the image of an instance of SDB.Student is that instance of Student having the same passport number (PPN); there might be none.

4.2 Algorithmically Maintained Unifying Types: Producer Types Again

Another form of producer type can be used when the existence of instances of the unifying type can be defined by an algorithm, e.g., there is to be exactly one student object for each distinct social security number held by any student in any of the underlying types. This is illustrated in the following simple example:

```
CREATE PRODUCER TYPE Student
UNIFYING EDB.Student, WDB.Student, NDB.Student
FUNCTIONS (
    SSNum Integer AS IDENTIFIER );
```

This form requires that each of the underlying types also has SSNum as an identifier function. This command implicitly defines the producer rule for the Student

type and also the Image mapping for each of the underlying types. The implicit producer rule is

```
…PRODUCING BY OSQL(
SELECT SSNum(x) FOR EACH EDB.Student x
UNION
SELECT SSNum(x) FOR EACH WDB.Student x
UNION
SELECT SSNum(x) FOR EACH NDB.Student x;)
```

The Image function naturally maps each underlying student into the unifying student having the same social security number. For each underlying type db.Student, Image is effectively redefined as:

```
CREATE FUNCTION Image(db.Student x)→Student y AS OSQL
SELECT y WHERE SSNum(y)=SSNum(x);
```

4.3 Hidden and Visible Things

Everything is visible to the administrator. The intent is that only unifying types and image objects are visible to the end user (or application), with the underlying things being hidden. The initial state before the introduction of any unifying types is that everything is visible to the end user as well, with Image(x)=x and Unifier(t)=t.

Such intentions would be supported by a subschema mechanism, which is not currently provided in Pegasus. When it is provided, these intentions could still be overridden if the administrator chooses to explicitly design a subschema differently for some special purpose.

To be more precise, an *object* x is *hidden* if Image(x)≠x, including the case where Image(x) is null. A *type* t is *hidden* if it is an underlying type or if any instance of t is hidden. Thus any subtype or supertype of a hidden type is hidden, and so is any other type not disjoint from a hidden type.

Note that, as illustrated earlier for WDB.Student, the instances of a hidden type could themselves remain visible. It is not implied that the instances of a hidden type are necessarily hidden.

If a type t containing any hidden instances corresponds to a concept which is significance to end users, then a corresponding unifying type t' should be defined to contain the corresponding image objects. Thus, if the distinction between EStudents and WStudents is semantically meaningful, then the schema might take the form shown in Figure 1.

The Image function itself is hidden. Any other function having any hidden types in its signature is also hidden.

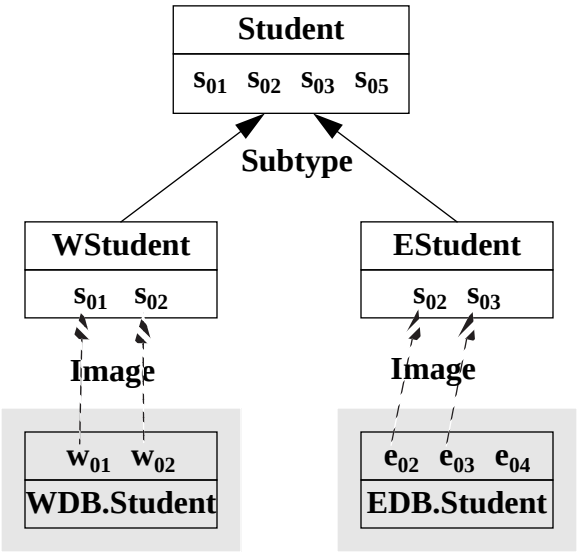

Figure 2. Underlying and Unifying Types, With Subtypes.

5 EXTENSIONS

5.1 Referential Integrity

The semantic principle of referential integrity (in terms of the OSQL functional object model [8][9]): a function should only be applicable to existing objects, and should only return existing objects in its results.

Referential integrity is difficult to support for assertable (stored) functions which have producer types as their argument or result type. Strictly speaking, whenever such a function is invoked, it would first be necessary to check the defining rules to insure that the arguments and results still exist. Anything short of this will result in objects remaining known and visible in the home system after they have been deleted from external data sources [16].

This problem is under investigation. Some possible avenues include periodic polling of the external data sources, or the introduction of monitoring facilities at the external data sources to notify the home system of changes. A more likely near-term solution is a pragmatic adjustment of the semantic model rendering knowledge in the home system somewhat independent of knowledge in the external sources.

5.2 Schema Mismatch

This approach can be extended to deal with certain forms of schema mismatch [14][19] by allowing the underlying or unifying types to be sets of types or functions rather then sets of ordinary objects. These types would then be subtypes of Type or Function in an OSQL schema.

5.3 Multiple Viewpoints

The Pegasus design currently provides one unified viewpoint for all users of a home system, though they could have different subsets of this viewpoint. Multiple viewpoints might be supported in two ways. First, multiple unifying viewpoints over the same underlying data could be supported by allowing the viewpoint to be a second parameter in the Image and Unifier functions. Thus Image(x,v) and Unifier(t,v) would define the unified view under viewpoint v.

Second, the current design presumes only a single level of unification, so that

Image(Image(x))=Image(x),
Unifier(Unifier(t))=Unifier(t).

This constraint could be relaxed in the future.

5.4 Multiple Correlating Functions

A *correlating function* f may be used to determine when underlying objects are semantically equivalent, having the property

$f(x_1) \approx f(x_2) \Rightarrow Image(x_1)=Image(x_2)$.

E.g., things having the same social security number have the same image.

We use $x \approx y$ to signify that x and y are equal and not null. If f is multi-valued, e.g., people may have several social security numbers, then $f(x_1) \approx f(x_2)$ holds if any non-null value in $f(x_1)$ equals any non-null value in $f(x_2)$; it does not hold if either is empty. Thus two persons have the same image if any of their social security numbers match.

We assume that an underlying object passes its correlating function value on to its image object (possibly by a form of upward inheritance [3]), so that

f(Image(x))=f(x) if f(x) not null.

E.g., the social security number of an underlying object becomes the social security number of its image.

If f is multi-valued, we should have $f(Image(x)) \supseteq f(x)$.

A correlating function need not be unique-valued within a single underlying type, since several distinct instances can have the same image.

An ideal correlating function is also:

- Stable: an object always has the same value of the correlating function.
- Total: no null values for any instances of the argument type.

Stability can be achieved if the correlating function is *initializable* (not updatable after initialization) and if the argument type is an *intrinsic type*, i.e., instances cannot be added or removed other than by creation or deletion. Otherwise an object could acquire such a property value some time after the object is created, or lose it before the object is destroyed. Stability can be relaxed if the system supports *system-wide oid replacement* whenever an oid changes: replacing all occurrences

of the old oid in the system with the new one. This is more feasible if it is known that little or no data is stored using the oid.

Totality can be relaxed if auxiliary means of identification are available, as discussed below.

Algorithmic unification of underlying instances (i.e., mapping to a common image) has thus far been based on a single correlating function such as social security number assumed common to all the underlying types. This is not always feasible. Students in one pair of underlying types may be correlatable by social security numbers, while those in another pair may only be correlatable by employee numbers, or by military service numbers, or by passport numbers [11][6].

Of course, it is necessary to know whether a student in one underlying type is the same as any student in any other underlying type, but it doesn't follow that all the correlating functions must be defined on all the underlying types. In effect, if SSN(x)=SSN(y) and PPN(y)=PPN(z), then x and z should have the same image, even if SSN is not defined for z and PPN is not defined for x.

Let f_i be a set of correlating functions which are all defined on a unifying type, and each is defined (overloaded) on one or more underlying types. Two such functions are *coordinated* if any one of the following conditions holds:

- There is some conversion (possibly a stored table) mapping the values of one into the values of another.
- They are both defined on the same underlying type. (Any f_i is coordinated with itself.)
- There exists another function which is coordinated with each of these.

Such coordination is necessary in order to infer, for example, that the student with a certain social security number is the same as the student with a certain passport number.

If more than one of the above conditions holds, e.g., there is more than one underlying type on which f_i and f_j are both defined, then they might be *inconsistent*. A social security number and passport number belonging to one student in one underlying type might belong to different students in another underlying type.

The argument types of two coordinated functions are *potentially coordinated* types. Full coordination among types may still fail in either of the following cases:

- The correlating functions are not total, i.e., they don't have values for all instances of their argument types.
- A coordinating linkage does not exist on the instance level.

  For example, suppose SSN is defined on underlying types t_s and t_{sp}, while PPN is defined on t_p and t_{sp}. Correlation between an x_s in t_s and an x_p in t_p could not be established if there simply didn't exist any x in t_{sp} having the corresponding values of SSN and PPN. In some sense, this is a failure to be total in the inverse direction: Image(x_s) has no pre-image in t_{sp}.

For simplicity, let a correlating function f_i be null-valued for any underlying type on which it is not defined. If t_u is the unifying type, then the Image function for any of the underlying types can be loosely characterized as

Image(x) = Select t_u u where $f_1(u) \approx f_1(x)$ AND … AND $f_n(u) \approx f_n(x)$.

The image of x is that u in t_u which matches on the non-null values of all correlating functions. Sometimes Image(x) may be undefined, perhaps because x is a new object in the underlying types. There may also be anomalies due to inconsistencies in the correlating functions.

A more precise definition is expressed in the following algorithm for finding Image(x):

1. Construct the set

   Z = Select t_u u where $f_1(u) \approx f_1(x)$ OR … OR $f_n(u) \approx f_n(x)$;

   Z contains the objects in t_u which have the same (non-null) value as x for at least one correlating function.

2. If Z has exactly one element, then that is Image(x).

3. If Z contains more than one element, we have an identification conflict. Distinct elements in t_u have correlating function values belonging to the same underlying x. For example, SSN(x) and PPN(x) belong to different image objects. The safest thing is to tell the user to make a correction.

4. Z might be empty because x doesn't have values for any correlating function, or because x has values for a “complementary” set of correlating functions. For example, x might only have a social security number, while existing objects in t_u only have passport numbers, their social security numbers being null.

   If Z is empty, then x might be mapped to an existing element of t_u, or a new image element may have to be created. Existing elements u in t_u to which x might be mapped are those for which there exists a correlating function f_i such that $f_i(x)$ is not null but $f_i(u)$ is null. (E.g., if x has a social security number, then any u with null social security number is a candidate image.) If there is no such u, then a new element has to be created in t_u as the image of x. If one or more such elements do exist in t_u, then user intervention is required to choose one of them or to decide that a new element should be created. User decisions would be based on the values of other functions.

   When there is no correlating function f_i with non-null $f_i(x)$, then the image mapping must be arbitrarily asserted (stored). This is the general case if we do not have any correlating functions on which to base the unifying mappings.

In all cases, u should acquire all the non-null correlating function values of x. For example, if u was chosen to be Image(x) by matching SSN and it has a null PPN, it should acquire the PPN of x if it has one. The values may also need to be propagated back into other underlying objects. All pre-images of u should acquire its

values of SSN and PPN if those functions are defined on their types. Note that, due to autonomy, if some x which maps into this u has no social security number, we can't force it to have one — at least not in the external database. On the other hand, it might make sense to propagate back down if the underlying types are not imported, or if data about imported objects is being maintained in the home system.

The preceding algorithm is generally applicable to independently maintained unifying types [Section 4.1]. Defining the unifying type as a producer type [Section 4.2] is difficult when more than one correlating function is required. The producer rule could be based on the concatenation of all the correlating functions, but complications arise when some values are null, or can change from non-null to null.

6 CONCLUSIONS

In Pegasus, the key to the problem of object identity in a multidatabase system is to differentiate between the view of a data administrator, capturing all the discrepancies and their solutions, and the view of an end user, in which only a consistent result is visible. There are thus three aspects:

- The existence and identity of underlying objects representing the possibly discrepant information in the external data sources.
- The existence and identity of unifying objects presenting a coherent view to end users.
- Mappings between the two.

New language constructs with which the administrator can describe solutions to identity problems include underlying and unifying types, the Image function, and producer types.

7 REFERENCES

[1] R. Ahmed and A. Rafii, “Relational Schema Mapping and Query Translation in Pegasus”, Workshop on Multidatabases and Semantic Interoperability, Tulsa, OK, Nov. 1990.

[2] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M.C. Shan, “Pegasus Heterogeneous Multidatabase System”, IEEE Computer, December 1991.

[3] Rafi Ahmed, Joseph Albert, Weimin Du, William Kent, Mohammad Ketabchi, Ravi Krishnamurthy, Witold Litwin, Ming-Chien Shan, “An Overview of Pegasus”, HPL-92-12, Hewlett-Packard Laboratories, October 1992.

[4] Rafi Ahmed, Joseph Albert, William Kent, Mohammad Ketabchi, Ming-Chien Shan, “Automatic Importation of Relational Schemas in Pegasus”, HPL-92-13, Hewlett-Packard Laboratories, October 1992.

[5] C. Batini, M. Lenzerini, and S.B. Navathe, “A Comparative Analysis of Methodologies for Database Schema Integration”, ACM Computing Surveys 18(4), Dec. 1986.

[6] Abhirup Chatterjee and Arie Segev, “Rule Based Joins in Heterogeneous Databases”, LBL Technical Report 30754, Lawrence Berkeley Laboratory, l992.

[7] Jan Chomicki and Witold Litwin, “Declarativeness of OO Multidatabase Mappings”, in preparation.

[8] Fishman, D.H. et al, “Iris: An Object-Oriented Database Management System”, ACM Transactions on Office Information Systems, Volume 5 Number 1, January 1987.

[9] Dan Fishman, et al, “Overview of the Iris DBMS”, *Object-Oriented Concepts, Databases, and Applications*, Kim and Lochovsky, eds, Addison-Wesley, 1989.

[10] Sandra Heiler and Barbara Blaustein, “Generating and Manipulating Identifiers for Heterogeneous, Distributed Objects”, Proc Third Int'l Workshop on Persistent Object Systems, 10-13 Jan 1989, Newcastle, Australia.

[11] W. Kent, “The Entity Join”, Proc. Fifth Intl. Conf. on Very Large Data Bases, Oct. 3-5, 1979, Rio de Janeiro, Brazil.

[12] William Kent, “The Many Forms of a Single Fact”, Proc. IEEE COMPCON, Feb. 27-Mar. 3, 1989, San Francisco, Calif.

[13] William Kent, “A Rigorous Model of Object Reference, Identity, and Existence”, Journal of Object-Oriented Programming 4(3), June 1991, pp. 28-38.

[14] William Kent, “Solving Domain Mismatch and Schema Mismatch Problems With an Object-Oriented Database Programming Language”, Proc. 17th Intl. Conf. on Very Large Data Bases, Sept. 3-6, 1991, Barcelona, Spain.

[15] William Kent, “The Breakdown of the Information Model in Multi-Database Systems”, SIGMOD Record 20(4) Dec. 1991.

[16] William Kent, Mohammad Ketabchi, Ravi Krishnamurthy, Witold Litwin, Ming-Chien Shan, “On Scoping, Naming, and Overloading in Heterogeneous OODBMS”, in preparation.

[17] Setrag Khoshafian and George Copeland, “Object Identity”, Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Portland, Oregon, 1986.

[18] Won Kim and Jungyun Seo, “Classifying Schematic and Data Heterogeneity in Multidatabase Systems”, IEEE Computer, December 1991.

[19] Ravi Krishnamurthy, Witold Litwin and William Kent, “Language Features for Interoperability of Databases with Schematic Discrepancies”, Proc ACM SIGMOD Int'l Conf on Mgmt of Data, Denver, Colorado, May 29-31 1991.

[20] Amit P. Sheth and James A. Larson, “Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases”, ACM Computing Surveys 22(3), Sept. 1990.

Object identification in multidatabase systems

William Kent, Rafi Ahmed, Joseph Albert, Mohammad Ketabchi, Ming-Chien Shan

**Pegasus Project, Database Technology Department,
Hewlett-Packard Laboratories, Palo Alto, California, USA**

Abstract

In the Pegasus prototype multidatabase system, the key to the problem of object identity is a differentiation between the view of a data administrator, capturing all the underlying discrepancies and their solutions, and the view of an end user or application, in which only a consistent result is visible. New language constructs with which the administrator can describe solutions to identity problems include underlying and unifying types, the Image function, and producer types.

Notes to HP Labs Technical Publications Department

1. This paper will be presented at the IFIP TC2.6 DS-5 Conference on Semantics of Interoperable Databases, Lorne, Victoria, Australia, November 16-20, 1992. It will appear in the conference proceedings. We have already sent them a copy of the copyright release.
2. This paper is a revision of HPL-92-68, May 1992. Besides a thorough revision of the text, we have also added more authors.