The Breakdown of the Information Model in Multi-Database Systems

William Kent
Pegasus Project
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, California 94303
kent@hplabs.hp.com

1. Introduction

Multi-database systems violate implicit assumptions we make about how databases model reality. We can progress by recognizing the assumptions and their violations, and then adapting. This paper describes some foundations for such an approach.

2. Information Modeling in a Database

A database tries to present a faithful model of some body of information we can loosely call "reality" (even though the information might be historical, obsolete, fictional, speculative, or even erroneous). The model is constructed from whatever sorts of materials a particular data model provides, such as relations and attributes, or entities, attributes, and relationships. In this paper, we will build models using the materials provided by functional object models such as Iris [Fi]: objects, types, and functions. Functions collectively model the notions of attributes, relationships, and operations.

While a database can't really guarantee realism in the sense of correctness, techniques such as naming and constraints help maintain plausibility — the look and feel of realism.

2.1 Identity and Naming

The most fundamental principle on which modeling rests is a one-to-one correspondence between the *proxy* objects in the database and the *entity* objects in the real world the proxies are supposed to represent. The assumption is strongest in object-based systems, a bit weaker in value-based systems. So long as there is such a tight correspondence, we even want to forget the distinction between the two. We are content to think we can create an employee named Sam, rather than having to be so picky as to request the creation of a proxy object intended to represent the real Sam. The fact that we're creating something whose "age" is considerably greater than zero at the moment of creation doesn't seem to bother us.

Naming is also essential to make modeling work. If it stands for Sam, we want to call it Sam. If we are thinking about employees, we want a type called "Employee". If we are concerned with salaries, there ought to be a function called "Salary". We can think of such types and functions in the database as themselves being proxies for their real-world counterparts.

We have long ago learned that mechanistic systems, from paper ledgers to computer systems, don't have nearly the sophisticated power of the human mind to resolve ambiguous references in context. So we have grown accustomed to assigning unique identifying codes to things, such as employee numbers, social security numbers, part numbers, department numbers, airport and city codes, etc. We accept those as serving the purpose of "naming", even though people rarely use them in conversational references to things. In spite of our best intentions, though, such identifiers can still be unreliable. Several of them might be the same (some employee number might match some part number), not all entities might have one, some entities might have several — and they might even change, meaning that we need something else to identify the thing whose identifier has changed. That's why object-oriented systems have gone one step further, generating their own unique and invariant identifiers for proxy objects in order to preserve the correspondence with the entity objects being modeled.

How do we know whether x and y refer to the same object [K1, K2]?

In a single object-oriented database (or other object-oriented system), the identity of most objects can be traced back to recognizable creation events, which leave their mark in terms of unique object identifiers. Distinct creation events yield distinct oid's. x and y refer to the same object if and only if their values are matching oid's.

Value-based systems such as relational databases don't provide quite as good a model of identity. They don't have explicit creation events as the basis for identity and existence. Primary and foreign keys provide a rough approximation: insertion of a new key value in a "primary" (entity) relation approximates the creation of a new object. The scheme is imperfect in several respects:

- Some objects may have no primary relation in which their identifiers serve as the primary key. This might be the case for such things as dependents, cities, schools, etc. A database might still contain information "about" such objects, sometimes — but not always — in unnormalized relations, such as the social security number, name and age of a dependent. It's quite difficult to associate a creation event with such objects.
- The same key might be a primary key in several tables, so that insertion into a table does not necessarily imply creation of a new object. This might arise simply from vertical partitioning (e.g., keeping employee benefits and payroll information in separate tables). It might also arise from subtypes: if data about salesmen is kept in a separate table, then inserting an employee into the table doesn't necessarily connote creation of a new object. In some cases the problem might be solved by designating one of the tables to be the real primary one, but that doesn't always work.
- Nothing prevents several primary keys from belonging to the same object. Different relations might be keyed by social security number, employee number, and military service number. There is no inherent mechanism to detect when several such identifiers identify the same person.

Thus value-based systems don't model identity as well as object-oriented systems, presenting a two-stage problem: identity first needs to be clarified within a single system, and then integrated into a multi-database environment.

2.2 Constraints

Constraints — either explicitly stated or implicit in data structures — are also crucial to maintaining the realism of the model. Since two real employees can't have the same employee number, we don't let two proxies in a database have the same employee number. That's typically enforced by a unique index whose scope is all employees. Also, since a real employee only has one birthday, we don't let a proxy in a database have more than one. That's typically enforced implicitly in the data structure. When we assert a birthday for Sam, it overwrites any previous birthday that might have been recorded for him, so he still has only one recorded birthday.

2.3 Certitude

Simple databases seem certain about their information — even if it's wrong. If you ask for Dick's birthday, you generally get an answer in no uncertain terms, reinforcing the look and feel of realism.

2.4 Stability

Although there is always some update going on, and occasionally some schema modification, a database is relatively stable. There are rarely major population swings in the database content, and the user has a relatively clear notion of what portion of reality is being modeled.

3. The Breakdown in Multi-Database Systems

Any one database does a pretty fair job of maintaining the realism of its model. We know what to expect of it, and we know to expect different things of different databases. We don't expect the parts database to know about employee benefits. We expect a database in another country to spell the Salary function differently, and to give results in a different currency.

Things start to break down, though, when we deal with several databases at a time [Ba, Da, Sh]. To begin with, we ask our system designers to make that transparent to us, so that we don't have to know or care which databases we're dealing with, or how many, at any point in time.

We still want to be presented with the illusion of a single database, a single faithful model of reality.

That's where most of the problems begin. Too many of the techniques that preserve the realism of a single database's model of reality fail across multiple databases, and we are currently trying to figure out how to fix that.

3.1 Identity and Naming

Above and beyond the aforementioned problems of value-based systems, multi-database systems present us with the problems of multiple creation events and uncoordinated object identifiers.

Just to keep things clear, let's agree that each act of creation creates a distinct proxy object, so that proxies created in different databases are necessarily different. If we "create" the employee Sam in several databases, we have created several proxies.

That immediately breaks down the fundamental modeling assumption, the one-to-one correspondence between proxies and entities. We might have different proxies in the system representing the same entity — and they might not behave consistently. The fundamental problem of multi-database systems is to restore consistency of the model, i.e., to provide an environment in which there appears to be one proxy per entity.

We can no longer ignore the difference between proxies and entities quite so casually, though we'd like to provide a system in which end users can. "Same proxy?" and "Same entity?" are no longer the same question; the answer to one can be different from the other. Let's use $x=y$ when x and y refer to the same proxy, and $x\equiv y$ when they refer to the same entity. The first should imply the second, but not *vice versa*.

Creating a proxy no longer corresponds uniquely to creating an entity, and we even need to rethink what "creation" means here. It's a little harder to pretend that we are "creating" a new Sam each time. While we *are* creating a new proxy, we are only *introducing* an entity that wasn't previously known to that particular database.

How shall we know that two proxies represent the same entity, that $x\equiv y$ even though $x\neq y$? That's the $64,000 question. To begin with, the question can only arise in a context where both proxies are known.

A simple but tedious solution is to have someone tell us, either when a proxy is created or at some later time, that one proxy represents the same entity as another, e.g., that the Sams we've created in several databases really represent the same thing. We would keep tables of such correspondences around, and look them up every time we need to evaluate $x\equiv y$ when $x\neq y$.

Though such a solution must be available as a last resort, it's not very nice. Our job is to make life easier than that for the user whenever we can. Which means we should try to infer $x\equiv y$ from available information whenever we can — which is possible in various degrees, and with various levels of confidence. It's easiest when there's a suitable identifying property, such as employee number. Two employee proxies represent the same entity if they have the same employee number (assuming we can detect identity of the employee number functions). This method is reliable to the extent that the property is a reliable identifier: unique (no two employees have the same one), singular (no employee has two of them), total (each employee has one), and stable (the same one always identifies the same employee).

More complex situations arise when there are several such identifying properties, or properties which are almost reliable. For instance, some people might have social security numbers and some might have employee numbers; some might have both, and some neither. Two databases might show the same social security number assigned to people having different employee numbers. An even more complex approach is to try to establish equivalence with some level of confidence based on similarities.

This problem applies to meta objects (types and functions) as well as to ordinary objects, reflected in the problems of schema integration. Not only do we have to figure out whether two proxies represent the same person, we also have to figure out whether two types are the same, or two functions are the same. Before we can observe that Sam has different birthdays in two databases, how did we figure out that two "birthday" functions are the same thing? They may or may not have the same names, and they may or may not match in other properties. Just what

would it mean to say two things are the "same function" if they have different names, or different argument or result types, or if they return different result values for the "same" argument? What does it mean to say two types are the same if they have different instances or subtypes or supertypes, or if they have different functions defined on them?

Beyond these semantic issues are representational problems involving object identifiers generated by autonomous databases. Different oid's might be assigned to the "same" object, and the same oid might be assigned to "different" objects. Equally annoying, the oid formats might be different, so they cannot be freely intermixed.

3.2 Constraints

We currently don't know how to enforce constraints across autonomous databases. We are fuzzy as to whether constraints apply to proxies or to entities. Paradoxes arise if we do or don't assume that two proxies represent the same entity. In Figure 1, if the proxy objects $@_1dick$ and $@_2richard$ represent the same thing, it has two birthdays. If they are different things, then we have two things with the same employee number.

Hidden constraints, which we didn't even realize were implicit in single databases, emerge to plague us in multi-database systems. In a single system, only one person has a given employee number and only one has a given social security number, so we don't bother to say that persons having the same employee number must have the same social security number. This rule can be violated in multi-database systems.

Some examples in two databases are shown in Figure 1 ($@_1xxx$ denotes an oid from the first database).

3.3 Certitude

With integrated databases, you have a greater likelihood of getting a wishy-washy list of possibilities: "Well, one source gives Dick's birthday as so-and-so, but another lists it as ..." This again underscores the shift in our perception of a database from a mirror of reality to a body of knowledge — sometimes uncertain.

$Employee_1 = \{@_1tom, @_1dick\}$	$Employee_2 = \{@_2richard, @_2harry\}$
$EmpNum_1: Employee_1 \rightarrow Integer$ *[single,unique]* $EmpNum_1(@_1tom) = 111111$ $EmpNum_1(@_1dick) = 222222$	$EmpNum_2: Employee_2 \rightarrow Integer$ *[single,unique]* $EmpNum_2(@_2harry) = 333333$ $EmpNum_2(@_2richard) = 222222$
$Birthday_1: Employee_1 \rightarrow Date$ *[single]* $Birthday_1(@_1tom) = 4/4/44$ $Birthday_1(@_1dick) = 5/5/55$	$Birthday_2: Employee_2 \rightarrow Date$ *[single]* $Birthday_2(@_2harry) = 6/6/66$ $Birthday_2(@_2richard) = 4/5/67$
$SSNum_1: Employee_1 \rightarrow Integer$ *[single,unique]* $SSNum_1(@_1tom) = 123456789$ $SSNum_1(@_1dick) = 987654321$	$SSNum_2: Employee_2 \rightarrow Char$ *[single,unique]* $SSNum_2(@_2harry) = '012345678'$ $SSNum_2(@_2richard) = '987654312'$
$Manager_1: Employee_1 \rightarrow Employee_1$ *[single]* $Manager_1(@_1tom) = @_1dick$	$Address_2: Employee_2 \rightarrow Char$ *[single]* $Address_2(@_2richard) = '77\ Main\ St'$

Figure 1. Discrepancies between two databases.

3.4 Stability

A query to list all employees can return enormously different populations, depending on which databases are attached at the moment. Though logically equivalent to insertion and deletion, the quantitative difference is large enough to feel like a qualitative difference. We need to prepare the user for that. It's part of the behavior he has to expect, hence part of the semantics of the system. In effect, the view of reality may expand or contract quite dramatically. While we should try to explain it in semantic rather than implementation terms, we can't hide it behind the veil of transparency.

4. SOLUTION APPROACHES

A number of techniques are being investigated in the Pegasus project of the Database Technology Department at Hewlett-Packard Laboratories.

4.1 Spheres of Knowledge

A database is not so much a faithful model of reality as a repository of "knowledge" about reality. Like people, different databases may "know" different things about the same situation.

End users and administrators need different views. End users should still see a consistent model, like a single database, without seeing underlying problems. Administrators need a broader view. They need to see the underlying databases and their problems, and they need tools to provide nicer views for end users.

A *sphere of knowledge* is an abstraction of one of these databases or views. It might in fact be a subset of the information in a database, or a combination of several databases, or perhaps be some other sort of information source. Every object exists, and every function is executed, with respect to a sphere.

We model a multi-database system as a system of such spheres (Figure 2). Spheres s_1 and s_2 correspond to underlying databases to be integrated. In the example, they contain four proxy objects intended to represent three entity objects (along with much other information not shown). Each of these two spheres is internally consistent within itself, as described in Section 2. Sphere s^* is an integrated view seen by an end user. It too is internally consistent, showing three proxies representing three entities, together with consistent information about them. $\bar{s}$ is a super-sphere seen by an administrator. He sees all the proxies and all the inconsistencies. In the sphere $\bar{s}$, the administrator can define the functions that map the sub-spheres s_1 and s_2 into the consistent sphere s^* for the end user [K3]. Such mappings have to establish correspondences for meta-objects (types and functions) as well as ordinary objects.

Sometimes the best we can do in the integrated view is to tell the end user that we have different information from different sources, and let him make the best of it. Sometimes he'd prefer to know which databases are or aren't attached at the moment, so that he knows the scope of reality currently being presented. Much as we would like attachment to be transparent, it can't always be. It is quite obvious whether something

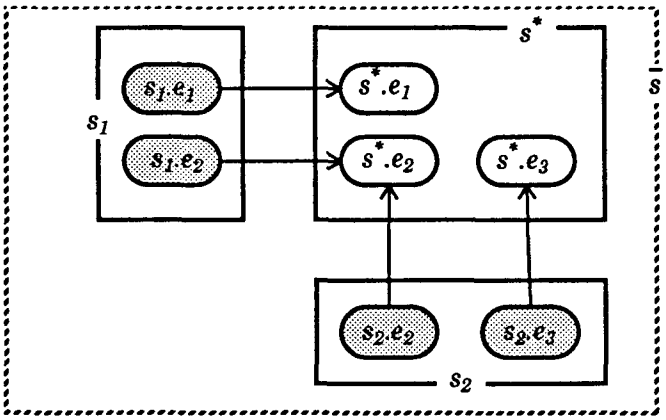

Figure 2. Spheres of knowledge.

is attached or not: certain things are either known or not. At one moment the database knows Sam's salary; at the next, it doesn't even know about Sam. Thus it may make sense for the user to know, and sometimes even control, which underlying spheres are attached at some particular time.

4.2 Integration Techniques

Other integration techniques being pursued in the Pegasus project are beyond the scope of the present paper. They are described in [Ah,K1,K3,Kr]. Of the problems described above, the ones currently receiving most attention are in the area of identity and naming, e.g.,

- Mechanisms for identifying objects and managing object identifiers for non-object-oriented data sources.
- Mechanisms required to define and assert equivalence among proxy objects in different databases.
- Investigation of the conditions under which supertypes do or don't provide an appropriate integrating mechanism.
- Various name resolution mechanisms for multi-database name spaces.

Many other things being investigated in Pegasus do not bear directly on the problems described herein, e.g., query decomposition and optimization, domain mismatch, and schema mismatch.

5. References

[Ah] R. Ahmed, P. DeSmedt, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M.-C. Shan, "Pegasus: A System for Seamless Integration of Heterogeneous Information Sources", Proc. IEEE COMPCON, March 1991, San Francisco, Calif.

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

[Da] Umeshwar Dayal and Hai-Yann Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System", IEE Trans. Software Engrg, Vol SE-10 No 6, Nov. 1984.

[Fi] D.H. Fishman, et al, "Overview of the Iris DBMS", *Object-Oriented Concepts, Databases, and Applications*, Kim and Lochovsky, editors, Addison-Wesley, 1989.

[K1] W. Kent, "The Entity Join", Proc. Fifth Intl. Conf. on Very Large Data Bases, Oct. 3-5, 1979, Rio de Janeiro, Brazil.

[K2] William Kent, "A Rigorous Model of Object Reference, Identity, and Existence", Journal of Object-Oriented Programming 4(3) June 1991 pp. 28-38.

[K3] 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.

[Kr] 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.

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