Integrating Data into Objects Using Structural Knowledge

Gio Wiederhold, Ph.D., and Arthur M. Keller, Ph.D.
Stanford University
Computer Science Dept.
Gates Building 4A
Stanford CA 94305-9040
{gio, ark}@cs.stanford.edu

Abstract

The concept of object technology has been well accepted. Using the grouping of data provided by object-oriented data structuring provides advantages in terms of efficient programming and program maintenance. However, current object-oriented systems are far from solving all the problems that exist in computing. Scalability is a particular concern. The modest acceptance of object-oriented databases is evidence of that problem. In this paper we survey the state of object data management, and current research. In particular we assesses multi-user and semantic scalability the object-oriented data paradigm.

Rather than rejecting the use of objects due to the problems discovered we describe an algebraic approach which permits the generation of data objects from relational data, based on the knowledge captured in a formal model. Now objects can be created that satisfy a variety of particular views, as long as the hierarchies represented by the views are subsumed in the network represented by the overall structural model.

In truly large systems new problems arise, namely that not only multiple views will exist, but also that the domains to be covered by the data will be autonomous and hence heterogeneous. We extend concepts used in object-based structural algebras to an algebra that can handle differences in terminology, suitable for information systems that span multiple domains. Using such a knowledge-based algebra, the domain knowledge can be partitioned for maintenance. Only the articulation points, where the integration intersects, have to be agreed upon. This to be achieved defined by matching rules which define the shared knowledge. The principal operations in the algebras are simple and provide for selection from the objects in the data space and composing them into new structures that represent the desired information.

At both levels, scaling is achieved by moving beyond hierarchical structures. The underlying concepts are based on the observation that integrated, multi-purpose data, and the knowledge that describes the data forms complex webs of information, while effective processing and its algorithms require hierarchical processing of the application problem-specific subsets.

1. Introduction

In modern software development efforts the concepts of object technology have been well accepted. Using the grouping of data and methods provided by object-oriented data structuring provides advantages in terms of efficient programming and program maintenance. However, current object-oriented systems are far from solving all the problems that exist in computing. Object configurations impose a consistency of structure that is attractive for implementation, but limits diversity. The need to enforce consistency limits scalability, since with a broadening of scope more diversity must be supported. The modest acceptance of object-oriented databases is evidence of that problem, since databases derive much of their power by serving as a communication medium among diverse users and communities.

There are two tributaries that converge to form today's stream of object technology for data management. From the software side, there is the desire to group data and associated program fragments, into larger chunks, thus raising the level of abstraction. Object classes are a natural progression, capturing the concept of abstract data types (ADTs). The origin of the second tributary is database technology, a well accepted and standardized basis for most of our information systems. Databases have dealt with large scale systems by imposing regularity and assuming semantic coherence. However, their marriage has not brought about happiness for all concerned.

The primary reason for using a database management system (DBMS) is to allow sharing of data among multiple applications or multiple users. When a database is shared among multiple applications, these applications typically have differing requirements for data access and representation. To support sharing in large-scale settings, DBMSs provide views, which subsets that are relevant to some user group, and can have specific representations and access rights. The infrastructure supports independence of execution, it includes atomicity, concurrency control, and persistence in case of failures.

Object-oriented database management systems (OODBMSs) are based on a single object schema that is used to support all data for all applications. This promotes coherence, and improves performance, but reduces independence of applications as well as scalability. Typically, an object schema is designed for one application, and other applications are grafted on, and well represented . The alternative, convening a committee to decide which object configuration is the *right* one is tedious, costly, and often naïve. Multiple views are a valid concept, for instance in logistics having a supplier, a warehouse and a consumer view will structure the objects differently, even if they all refer to same parts base. One organization's ideal hierarchy is often viewed by another organizations as an impenetrable bureaucracy.

When combining independently developed legacy applications, we do not have the luxury of choosing a common object schema. Legacy will be with us forever. We cannot pretend that applications developed now will represent what we need in the future, so that anything being built now is the legacy, good and bad, for the next generation. We must retain the investment in operational application software and databases. New applications should be able to use the best of modern technology, however. Any new system architectures must also support evolution and maintenance.

2. Creating Objects

Rather than rejecting the use of objects due to the problems discovered we have developed an algebraic approach which permits the generation of data objects from relational data, based on the knowledge captured in a formal Entity-Relationship model, the Structural Model. The advantage is that now objects can be created that satisfy a variety of particular views, as long as the hierarchies represented by the views are subsumed in the network represented by the overall structural model. We now present the concepts, architecture, and implementation of a novel approach to sharing persistent objects.

PENGUIN [Barsalou *et al*., 1991] is an object data management system that relies primarily on relational databases for persistent storage. It supports multiple object configurations based on the underlying data [Wiederhold, 1986]. Such *object views* of relational databases allow each application to have its own object schema rather than requiring all applications to share the same object schema.

2.1 Concepts

The underlying architectural concepts we use derive from the three-level data modeling architecture from the ANSI/SPARC committee [ANSI, 1977]. They consisted of the view level provided to the customer, the conceptual level, and the physical level. The view allows multiple user data models, one for each application, that cover only part of the database and provide for logical data independence. The conceptual level describes a common data model for the entire database and provides for physical data independence. In a distributed environment, it is possible for there not to be a single conceptual model, but one for each database or group of databases. The physical level describes the abstract implementation of the database, including distribution, data structures, and access paths. Relational DBMSs (RDBMS) typically strongly support the view level, and many views may be defined on the conceptual level. However, RDBMSs often are quite restrictive on the

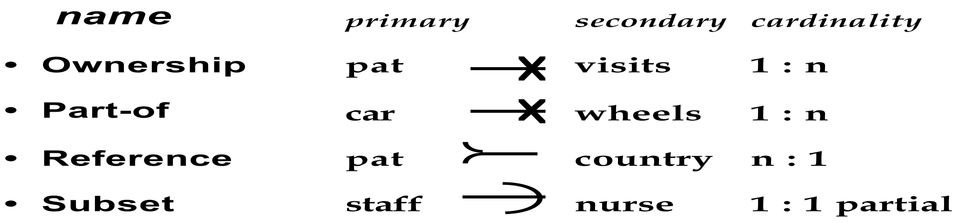

Figure 1: Summary of Structural Model Connections

physical level, largely because of limited semantic capability and to support simple implementations with high performance. In contrast, OODBMSs typically do not support the view level; there is only one view corresponding to the conceptual level. OODBMSs do support a rich variety of structures at the physical level, with much support for programmer control.

2.2 Semantics on Relational Databases

In the design phase of relational databases much semantic and structural information is collected, and there are quite a number of design tools available, mostly based on the Entity-Relationship (E-R) model [Chen:76]. Common modern tools use the IDEF models. Relationships in the basic E-R model are classified by their cardinality: 1:1, 1:n, and m:n. Processes for the integration of relational databases have been widely published, starting with a structural model defined by [ElMasri and Wiederhold, 1979]. If integration of OODBMSs is to be performed, and object schemas are available, we can also use their models. They contain more semantic information, specifically references among objects and structure within object classes as Part-of and ownership.

In our approach we use the structural model to represent the semantics needed to cover both object requirements and relational sources. A Structural Model is a graph where each node represents a relation and each edge represents connection (or relationship) between two relations. These connections define feasible and persistent join paths. There are relational tuples and 4 types of connections in the Structural Model. The connections are shown in Figure 1, but since the ownership and Part-of connections have identical semantics in object models, we will subsequently ignore their differences.

1. Tuples, as in the relational model are lists of attributes or items that co-occurs. In the E-R model these are shown as 1:1 relationships, but often omitted from explicit denotation to reduce excessive detail, and simply referred to as data records.
2. Ownership connection, which links instances that depend on the owner. The subsidiary instances depend on the owner and disappear when the owner is removed. In the E-R model these are one subset of 1:n relationships. For example, Ships owns its Voyage records.
3. Reference connection, which links to abstract entities. The referenced entities must exist while there are references to them. In the E-R model these are the other subset of 1:n relationships, or conceptually better: m:1 relationships. For example, Ship refers to Type.
4. Subset connection, which links a generalization to a subtype. In the E-R model these are not made explicit, but also appear as a subcategory of ownership, however IDEF allows their notation and recognizes their cardinality as 1:n-partial. As in ownership, the subtype instance disappears when its generalization instance is deleted. For example, docks for Goods is linked to Docks (because Goods IS-A Dock).

These connections are only model concepts, they need not to be explicitly represented. They are summarized in Figure 1, using an example from a healthcare domain.

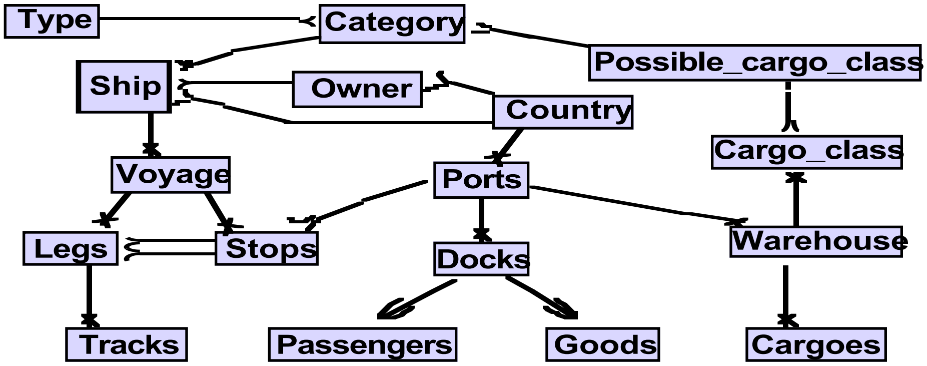

Figure 2: Example: Ships Database Structural Model

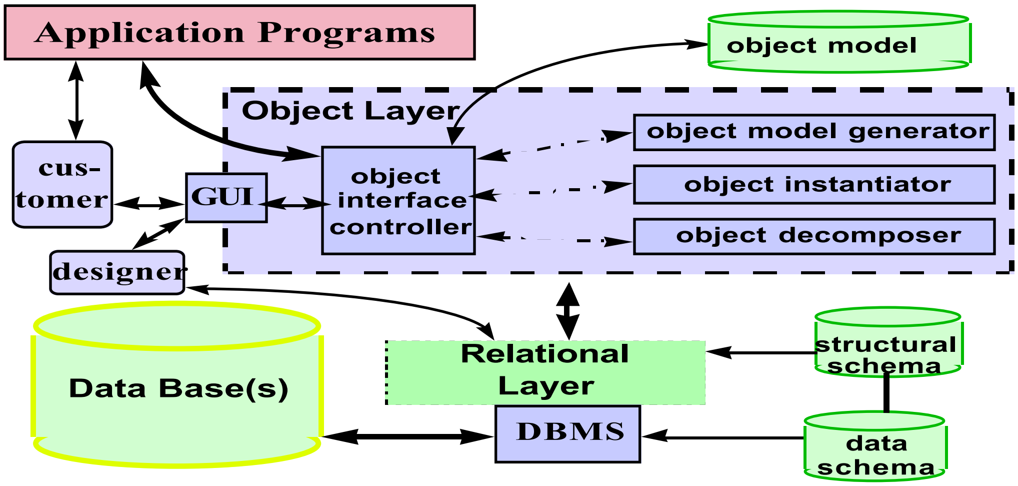

Figure 3: Penguin Architecture

Instances of connections are links among data objects, existing whenever the data values match. In an RDBMS they are instantiated by join and select operations. Processing can be speeded up if they are materialized, as in an OODBMS.

The m:n relationship of the E-R model is indirectly represented in the structural model, namely as a pairing of one of the other connections. This permits finer semantic mappings by distinguishing the 6 cases. For instance Suppliers of parts and consumers of parts. Furthermore, in implementation there has to be physical relation or other representation to define the actual linkages (Suppliers-consumers), since a join operation cannot determine the specific linkages. Figure 2 shows an integrated model, where the m:n relationship of a ship's Category and Cargo_class is materialized by the Possible_cargo_class relation. It should be noted that there are also more complex relationships, all decomposed into binary connections.

2.3 Penguin Architecture

To support the operations of object retrieval and storage the Penguin system provides object instantiation, using data from the relations in the supporting databases, and object decomposition, distributing updates to those relations. In order to perform these tasks an object schema must be made available, which relates a users' view-objects with the relations and its structural model. The overall architecture is sketched in Figure 3. All customer interactions are controlled via

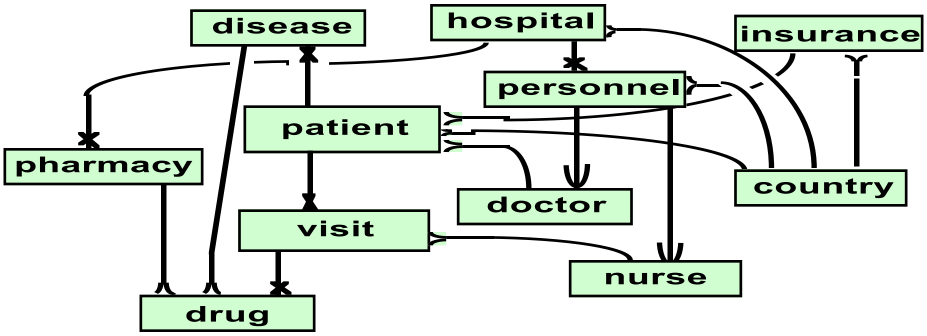

Figure 4: Structural Model in Healthcare.

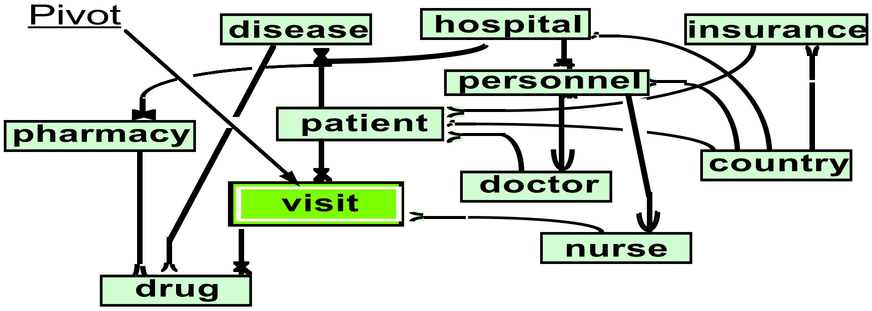

Figure 5: Creating an Object View

graphical user interface, but during execution data flows directly from application through the controller to the relational database. The controller selects which of the object modules is needed, and the modules retrieve or store information in the object model as appropriate.

Before objects can be retrieved or restored, this object model has to be populated. This is the task of the designer, who will be using the object generator.

2.4 Object Mode and Generation

The strength of the relational model is its formal algebra. The Structural Model connections are also manipulated by the same algebra. We use this capability to define object classes (view-objects) as object views of the underlying databases. There are four steps
involved in this definition, which uses as input the integrated structural model. We use now the example of Figure 4 which is associated with the schema of the relational database. The designer is aided by tools provided as part of the object model generator.

Step 1. Choosing a pivot relation. We choose a pivot relation which has a one-to-one correspondence between tuples in the relation and object instances in the desired object class. The key of the relation becomes the root of the object class.

The Penguin system displays the relations closer than a user-defined threshold to the pivot relation. The candidate set consists of these "nearby" relations. Figure 5 shows a structural model with pivot relation marked and the linkages to the candidate marked darker than the others. The weak connections and their dependent relations are omitted. In large model only a small fraction of the information is displayed. The designer can then mark which other relations
are to be omitted, and the remaining relations become the input for object model creation.

Then the Penguin system converts the candidate set to a candidate bag. This conversion is needed since proper relational models omit all redundancies in their conversion to Boyce-Codd normal form,

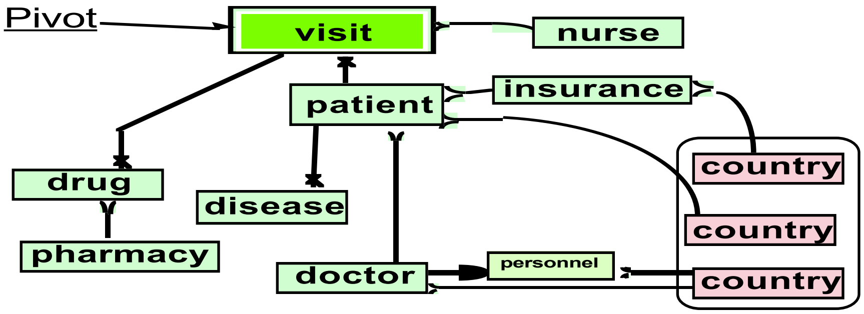

Figure 6: Object Structuring

whereas object models, describing instances require redundancy. For instance, in our example country appears once in the relational model, but in an object model the patient's country, the country of the patient's insurance carrier, and the doctor's country all differ. We have already omitted from our model the nurses' country and country where the hospital is located, since the application did not require this information. We can now also omit unneeded intermediate nodes, in our example the personnel relation. Figure 6 shows the resulting bag. It provides a covering tree of the candidate set, since now nodes are replicated so that there is a copy of a node for each acyclic path from the pivot relation. Edges are typically not replicated.

Step 2. Choosing instance variables. Typically not all available attributes are needed for a specific application and its object class. The designer is now prompted to make that selection and prune the class definition.

An application will typically use multiple object classes. If these classes are also to be supported from source databases Steps 1 and 2 will be repeated.

Step 3 Specify inheritance. Penguin shows the object classes which use pivots from on the same relation from which this new object class will inherit values. This approach can support multiple inheritance if the programming language does. For example, in C++ we instantiate the object instance of the right type and populate it with the data from the database.

Now the object model is complete. However since view update is inherently ambiguous, Penguin must now resolve potential ambiguities with the designer, so that the customer and the application can proceed without errors or run-time interactions.

Step 4. Resolving update ambiguities. Whenever operations only any information subset are performed there is the likelihood that dependent, but invisible information in the full information base must be changed. Since the full set is not visible to the customer or the application, but is available to the designer, the appropriate place to resolve any uncertainties on how to perform the database update is at design time.

Penguin must hence now present the designer with all possible ambiguities, and let the designer select the desired method. An expert system [Davidson, 1984] can help with that task, since typically the updates that involves least disturbance with the remainder of the information is appropriate. For instance, correcting the nurse entry for a visit should not mean that any other information about that nurse should be deleted from database, even though the nurse is no longer seen in this patient object, or perhaps any object accessed by the application.

Most issues are automatically resolvable, given reasonable heuristics, and any remaining ones are shown to the designer, who can then indicate which choice to take [Barsalou *et al.*, 1991].

Now retrieval and storage operations can commence.

3. Performance

The disadvantage of creating view-objects dynamically is that the additional layering has performance implications, so that the speedup expected from object-oriented databases versus relational databases, due to their hierarchical object storage, cannot be fully realized.

3.1 Object Size and Transmission

However, in integrated and distributed applications, the objects being transmitted will be focused and much smaller than objects stored in an OODBMS. Large systems tend to have multiple objectives, and, if directly implemented in an object formalism must manage and transport much excess and redundant data to their applications. Penguin objects are smaller due to the pruning of attributes from the base. This can offset some, but certainly not all of the performance hit when creating objects from an RDBMS.

We have also investigated improved methods of managing the transmissions, and found that having the serve transmit selected and projected subsets of the source relations, and performing the object assembly at the client can have significant benefits [Lee and Wiederhold, 1994].

3.2 Caching Issues

Penguin obtains performance in navigation by using a cache. Navigating a relational database tuple-at-a-time is very inefficient. In contrast, issuing a query to a relational database and then loading the result into a cache will provide much more efficient navigation. We first discuss the organization of the cache, and then we discuss concurrency control and transactions.

Penguin uses a two-level cache [Hamon and Keller, 1995]. The lower level of the cache is a network representation corresponding to the structural model. The upper level of the cache corresponds to the object classes of the user object schema. There is a virtual level of the cache that is language-specific.

The lower level of the cache contains a network that matches the structural model. The relations are linked together as in the structural model. Tuples in relations are linked together according to the joins based on connections. We use "semantic pointer swizzling" to turn semantic keys (foreign key to primary key) references into pointers in memory. Data in this level of the cache are stored non-redundantly. The lower level of the cache is shared among all applications on the same computer [Keller and Basu, 1996].

The upper level of the cache corresponds to the object classes of the user object schema. Data is stored in hierarchical form according to the object classes of the application. References to other object instances are also swizzled. Data in the cache is stored redundantly if necessary. The object cache is for a single application and lives in the application's address space.

3.3 Methods and Penguin

Objects can have arbitrary methods attached, while RDBMSs only support retrieval and tuple insertion and update. Most OODBMSs simplify dealing with arbitrary methods by restricting applications to one program language, often C++. Penguin generates three types of methods automatically based on declarative descriptions of the object classes. Navigation methods are generated for navigating among object instances using object class references. Query methods are generated that support path expressions on each object class. Update methods are generated that support changing a cached object, making the change persistent, and committing or aborting changes, i.e., transactions. These are by far the most common methods in practical use

Objects created by Penguin can also employ methods defined by the user. The object classes defined by Penguin are object classes that obey the normal inheritance mechanism of the object programming language (e.g., C++), so user object methods are inherited correctly. Unfortunately, user methods defined for one application's object schema are not applicable to another application's object schema, even if they share the same data. They can hence not be shared through current Penguin mechanisms.

3.4 Summary

The Penguin approach builds on relational databases by creating object views. Scalability, access to legacy data, and sharability is greatly enhanced. Performance is addressed Our techniques could be used to create object views of object databases, but we have not yet done this work, so it is beyond the scope of this presentation. The work on Penguin has had industrial spinoffs [Keller *et al*., 1993].

Conceptually, this work has become a basis for further work on intermediary services, leading to more general three-level architectures, where there are clients, *mediators*, and servers [Wiederhold *et al*., 1992], [Wiederhold and Genesereth, 1997]. Object representations are now being created in Digital Libraries using bibliographic resources [Wiederhold, 1995D], for decision support, manufacturing, and for logistics [Wiederhold, 1996]. Since most of these systems do not have a structural model to interpret, the object model is typically created manually, although tools to simplify its development are appearing [Papakonstantinou *et al*., 96].

4. Domains

While Penguin provides an intermediary service which can deal with multiple databases, it does not deals with inconsistencies among them databases that are common when the database are developed independently, by mutually autonomous groups [Wiederhold, 1995O]. Resolving such heterogeneities is difficult for systems, even where it easy for machines. There are many forms of heterogeneity: platforms, operating systems, data representation, transmission standards, scope of domains, and terminological inconsistency [Agarwal *et al.*, 1995]. These challenges are being addressed, with the latter issues, that deal with semantics of information, still being the weakest link.

4.1 Scope Differences

Frequently data from one source does not match in scope data from another source. For instance, budgets are prepared on a monthly basis, while pay periods are weekly. Given formal definitions of such differences we can address the problem of correctly performing relational database query operations. One approach is to define virtual attributes, and partial values to resolve scope differences and to map conflicting attribute values to common domains. An extended relational algebra can now resolve domain mismatch and to perform operations over information from mismatched domains [DeMichiel, 1989]. Sometimes the results must convey uncertainty, common when humans interact, but unexpected when computers are involved. This work enables information derived from multiple sources to be presented in an integrated form. For instance, when days and months are used as the basis for collected data, then their integration and reporting can never be precise in terms of weeks or months, only in terms of quarter years.

4.2 Terminological Differences

To deal with the differences among meaning of data, we use the term domain. When sources are truly autonomous, one cannot assume that a term means the same thing in a different domains. Even within the U.S. military terms differ among the services. Efforts are being made to enforce consistency, by developing DoD-wide data models, but these efforts are so massive, that by the time one segment is defined, other segments have changed and become obsolete. Such large *ontologies* often require compromises, reducing their precision. This work is further stymied by the movement to COTS-supplied material. Enforcing consistency onto and among industry is yet less likely. The terminology changes as fast as new products and services are developed. Companies that operate globally cannot even reach internal consistency.

We are hence commencing research to deal with semantic inconsistencies, based on the assumption that even when information is intended to be reusable, they cannot be expected to be globally consistent. We are envisaging an algebra over these terminologies, and the algebra will be knowledge driven [Wiederhold, 1994]. It would be as futile, of course, to try to describe inconsistencies globally as it is to define the base terminologies. But for sharing one only has to focus on the articulation points, where there is an intersection. There will be many intersections, of course, each with their distinct articulation knowledge. Using an intersection operation permits focusing on critical linkages. Figure 7 illustrates the

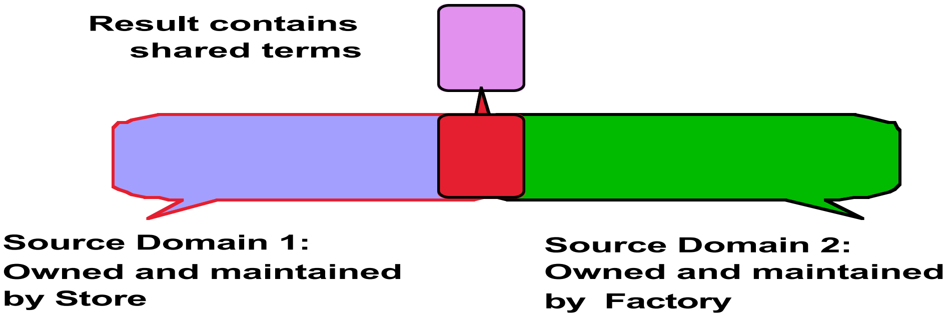
Result contains shared terms
Source Domain 1: Owned and maintained by Store
Source Domain 2: Owned and maintained by Factory

Figure 7: Intersection among Ontologies

concept when a store purchases items from a factory.

Now distinct information sources do not have to collaborate directly; their collaboration is managed by specialists in joining distinct knowledge bases. The base resources remain available and unchanged to the applications for in-depth processing, using the linkages established during the process of determining the articulations. Their use is then equivalent to the delegation of detailed domain tasks to specialists.

5. Conclusion

The tasks of moving from widely distributed data, methods and paradigms is a difficult one, and will never be completely solved. Many experiments are being performed. Frequently ad-hoc methods are applied to overcome some problem, but these often lead to new problems and costly maintenance. In this presentation we have covered past and future work on integration, focusing on approaches based on algebraic concepts. The motivation for the use of algebras is that now operations become composable, an essential aspect if scalability of approaches is to be achieved.

We have learned to deal formally with the differences of relational and object-oriented approaches. We can use the benefits of relational storage and sharability with the more focused services that object-orientation provides. This limitation in the physical level of RDBMSs gives a significant boost to the use of objects. Unless OODBMSs provide increased algebraic and view support, they will find it increasingly difficult to compete with enhanced RDBMSs.

Dealing more generally with distributed methods is a complementary challenge. We do not have the same capabilities to describe programming paradigms that we have for data structure. We still believe that progress here is possible by moving to higher levels of abstraction for programs, and have started research in that direction, exploiting the availability of a variety of client-server protocols [Wiederhold *et al.*, 92].

References

[Agarwal *et al.*, 1995] Shailesh Agarwal, Arthur M. Keller, Krishna Saraswat, Gio Wiederhold: “Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases”; *Int. Conf. on Data Engineering*, Taipei, Taiwan, March 1995.

[ANSI, 1977] D. Tsichritzis and A. Klug (eds.): *ANSI/X3/SPARC DBMS Framework*, AFIPS Press, 1977.

[Barsalou *et al.*, 1991] Thierry Barsalou, Niki Siambela, Arthur M. Keller, and Gio Wiederhold: “Updating Relational Databases Through Object-based Views”; ACM-SIGMOD 91, Boulder CO, May 1991, pages 248-257.

[Chen, 1976] Peter P.S. Chen: “The Entity-Relationship Model–Toward a Unified View of Data”; *ACM Trans. on Database Systems*, Vol. 1, No. 1, March 1976, pages 9-36.

[Davidson, 1984] Jim Davidson: “A Natural Language Interface for Performing Database Updates”; *Proc. IEEE Data Engineering Conference*; April 1984, Los Angeles CA.

[DeMichiel, 1989] Linda G. DeMichiel: “Performing Operations over Mismatched Domains”; *Proceedings of the Fifth International Conference on Data Engineering*, 1989.

[ElMasri and Wiederhold, 1979] Ramez ElMasri and G. Wiederhold: “Database Model Integration Using the Structural Model”; *Proceedings of the ACM-SIGMOD Conference*, Bernstein(ed.), Boston MA, June 1979, pp.191-198.

[Hamon and Keller, 1995] Catherine Hamon and Arthur M. Keller: “Two-Level Caching of Composite Object Views of Relational Databases”; *Int. Conf. on Data Engineering*, Taipei, Taiwan, March 1995.

[Keller, *et al.*, 1993] Arthur M. Keller, Richard Jensen, Shailesh Agarwal: “Persistence Software: Bridging Object-Oriented Programming and Relational Databases”; *ACM SIGMOD*, International Conference on Management of Data, May 1993.

[Keller and Basu 1996] Arthur M. Keller and Julie Basu. “A Predicate-based Caching Scheme for Client-Server Database Architectures”; *VLDB Journal*, Vol. 5, No. 1, January 1996, pages 35-47.

[Lee and Wiederhold 1994] Byung Suk Lee and Gio Wiederhold: “Efficiently Instantiating View-objects from Remote Relational Databases”; *The VLDB Journal*, Vol.3 No.3, July 1994, pages 289-323.

[Papakonstantinou *et al.*, 1996] Y. Papakonstantinou, S. Abiteboul, H. Garcia-Molina. “Object Fusion in Mediator Systems”; *Proc. VLDB 96*, Morgan Kaufman 1996.

[Wiederhold, 1986] Gio Wiederhold: “Views, Objects, and Databases”; *IEEE Computer Magazine*, Vol.19 no.12, December 1986, pages 37-44; reprinted in Diettrich, Dayal, Buchmann (eds.) *Object oriented-Systems*, Springer Verlag, 1988.

[Wiederhold et al., 1992] Gio Wiederhold, Peter Wegner, and Stefano Ceri: “Towards Megaprogramming”; *Comm. ACM*, November 1992, pages 89-99.

[Wiederhold 1994] Gio Wiederhold: “An Algebra for Ontology Composition”; *Proc. of 1994 Monterey Workshop on Formal Methods*, Sept. 1994, U.S. Naval Postgraduate School, Monterey CA, pages 56-61.

[Wiederhold, 1995D] Gio Wiederhold: “Digital Libraries, Value, and Productivity”; *Comm. ACM*, Vol.38 No.4, April 1995, pages 85-96

[Wiederhold, 1995O] Gio Wiederhold: “Objects and Domains for Managing Medical Knowledge”; *Methods of Information in Medicine*, Schattauer Verlag, Vol.34, No.1, pages 1-7, March 1995. .

[Wiederhold, 1996] Gio Wiederhold (editor): *Intelligent Integration of Information*; Kluwer Academic Publishers, Boston MA, July 1996.

[Wiederhold and Genesereth, 1997] Gio Wiederhold and Michael Genesereth: “The Conceptual Basis for Mediation Services”; to appear in IEEE Expert, 1997