A C++ Binding for Penguin: a System for Data Sharing among Heterogeneous Object Models*

Arthur M. Keller Catherine Hamon

Stanford University, Computer Science Dept.
Stanford, CA 94305-2140

Abstract. The relational model supports the view concept, but relational views are limited in structure. OODBMSs do not support the view concept, so that all applications must share the same arrangement of object classes and inheritance. We describe the Penguin system and its support for the view concept. Each application can have its own arrangement of object classes and inheritance, and these are defined as views of an integrated, normalized conceptual data model, in our case the Structural Model. We define view-objects in a language-independent manner on top of the conceptual data model. These view-objects can be complex objects supporting a composite structure. We discuss the extension of Penguin to support PART-OF (reference) and IS-A graphs for composite view-objects. We also discuss the C++ binding to Penguin, where C++ code is generated for object classes corresponding to the view-objects along with basic operations on them (creation, query, navigate, browsing, and update).

1 Introduction

The relational model introduced by Codd [8] has found widespread acceptance. The model is successful because of its simple but powerful description and the ability to reorganize data upon retrieval according to the needs of the application program, rather than having to anticipate all application requirements in advance. Relational database design theory leads to the design of normalized data structures, which are mathematically elegant but can be inefficient for some applications. Object-oriented database management systems [2, 9, 19, 23] and extended relational database management systems [10, 18, 27] support a richer description of data in a manner more convenient for design and other applications. These systems either do not support the view concept, or their notion of views does not support object-orientation, such as inheritance.

One approach is to define a standard arrangement of object classes and inheritance for all applications in a given application area. For the reasons described in [33], we believe that this approach is bound to have limited success.

We believe that the "second application problem" will arise for object-oriented application development. Organizations that are convinced of the benefits of

* For information about the Penguin project, please write to Arthur M. Keller at the above address or `ark@db.stanford.edu`

object-orientation will develop a new application in a new object-oriented programming language using an object-oriented database management system as a persistent store. An arrangement of object classes and inheritance will be designed specifically for that application, and careful attention will be paid to encapsulation. This application will be successful, but it will not integrate with existing applications. A problem will arise when the organization attempts an encore, that is, when they attempt to develop a new application using the object paradigm that is to share data with the first application. The new development team will need to violate the encapsulation of the first applications arrangement of object classes in order to support the differing abstraction and navigation requirements of the new application. Thus, use of object-oriented databases will succeed the first time, but will succumb to the "second application problem" when attempting to integrate a second application.

Our approach is for the object classes, inheritance hierarchies, and encapsulation decisions to be application specific. We believe it is important to provide independence between the application view level and the conceptual level. We support data sharing among multiple applications using the object paradigm, by giving each application its own view, consisting of view-objects (object classes) and inheritance. The contribution of this paper is the extension of the object model proposed in [3] to support inheritance and reference.

The following is the structure of the remainder of this paper. We review the Structural Model we use for the integrated conceptual data model in section 2. In section 3, we present how an application view is defined, consisting of object classes with inheritance and references. In section 4, we discuss the generation of C++ code for an application view. In section 5, we describe the Penguin system and its architecture. Following this, we then describe our future plans and our conclusion.

2 Structural Model

The relational model provides a simpler model of data that facilitates database design and navigation. Database design theory (see [31], for example) shows how normalization is important for eliminating undesirable properties of a relational database design. However, the price paid for normalization is a scattering of data fragments in the entire database as the original structure is flattened. The Structural Model provides structural semantics missing from the relational model [32] and useful for the database design and view integration process. The structural model of a relational database is a formal semantic data model consisting of normalized relations and connections (representing relationships) among those relations. We use a directed-graph representation of a structural model of the database, where nodes correspond to relations and arcs to connections (see Figure 1).

Formally, a connection is specified by the connection type and a pair of source and destination relations. Three types of connections are used (see [3]); they

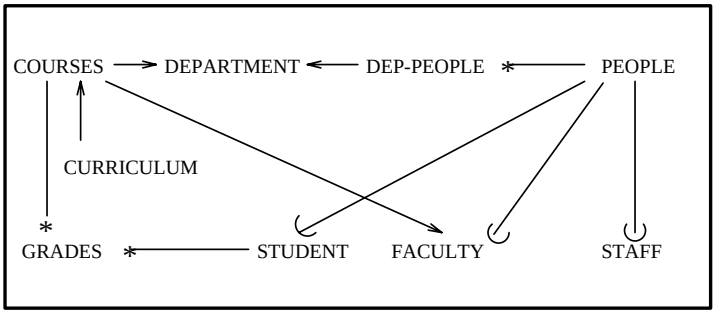

Fig. 1. Example of Structural Model.

correspond to precise integrity constraints, define the permissible cardinality of the relationships and encode the relationship semantics:

1. *Ownership connection* is a one-to-many relationship connecting a single parent tuple to zero or more child tuples dependent on the parent tuple. For example, the list of job skills for an employee is owned by the record for that employee. Two ownership connections may be used to represent an association between two entity sets, such as the grades relating students and the courses they take. We use $—*$ to represent an ownership connection.
2. *Reference connection* is a many-to-one relationship from an entity set to an abstract entity set. For example, a course references the department offering that course. We use $\rightarrow$ to represent a reference connection. Note that both ownership and reference connections have the cardinality of many-to-one, but there is a semantic difference: The skill set for an employee is really a repeating group for that employee and was given a separate relation because of normalization, while courses represent their own abstract entity set.
3. *Subset connection* is a partial one-to-one relationship connecting a parent superset relation to a child subset relation. The IS-A relationship can be modeled by a subset connection from the child to the parent. We use the symbol $—\supset$ to represent a subset connection.

Note that many-to-many relationships are not modeled directly in the structural model but can be represented using combinations of the connections.

3 Defining Application Views

View-objects in Penguin define the windows that permit users to access and update tuples stored in different or overlapping subsets of normalized relations [5]. These windows organize the data into the objects useful to an application by reassembling the attributes scattered among different normalized relations into the attributes of an object class. The Penguin implementation of the view-object concept supports object-oriented access to shared information. Each view-object

represents an object class for the programming language. They can refer to one or several existing view-objects to support a PART-OF hierarchy. They can also be organized into an inheritance graph that can support either single or multiple inheritance, as for an IS-A graph. In this section, we present the concept of view-object and describe how an application view can be built, consisting of object classes with inheritance and references.

3.1 View-Object Model

The view-object model proposed in [3] is based on the view-object concept introduced in [33] to support both abstract complex units of information and sharing of those units. In this model, a view-object is characterized by a name and a type. The name identifies the view-object, while the type specifies the hierarchical structure of view-object's instances. This structure is defined by a *template tree* whose nodes represent relations from the structural model, on which projections can be defined. The root of this tree represents a base relation called the *pivot relation* that will constitute the core component of the view-object. The remaining nodes represent the secondary relations that can be reached through different paths from the pivot relation in the graph defined by the structural model. Finally, an arc of the tree represents a path in the structural model between the relations located at the origin and the end of this arc. The semantics captured in the structural model through connections are thus kept at the view-object level.

The view-object model is based on a value-oriented approach since it uses the notions of hierarchical structures, and of set and records constructors. This model is similar to models based on the Nested Relational data model [1, 24, 25, 26] that permit the user to map complex objects to one nested relation. In the view-object model, relations are nested according to the hierarchical structure defined by the template tree. The major differences with the nested relational approach lay on the use of the Structural Model that enables us to build complex objects convenient for a specific application on top of the conceptual data model in a language-independent manner. Furthermore, a view-object model can exploit the three structural model concepts of aggregation, categorization and abstraction; integrity rules can thereby be more refined for view-objects than for nested relations [5].

We have extended the view-object model to support inheritance and reference among view-objects. The referential structure of a view-object defines the other view-objects that are reachable. Inheritance allows the user to reuse instance variables (i.e., attributes) and references from existing view-objects to design a new view-object. The result of the view-object definition contains the specification for the view-object but no actual data. The instantiation of a view-object is a separate process that fetches tuples from the underlying relational database and builds a hierarchical structure consisting of nested set of (sub)-tuples bound to the view-object. In this sense, a view-object is similar to an object class. Instantiation of a view-object will occur by a query or by navigation through

object reference (e.g., PART-OF). In the remaining part of this subsection, we present the major concepts that support the different stages of the view-object definition.

Choosing the Pivot Relation. The notion of pivot relation is central to the view-object formalism. The tuples of the pivot relation have a one-to-one correspondence with the object instances in the view-object. Hence, the primary key of the pivot relation is the semantic key of the view-object. This requires all the key attributes to be included in the projection defined on the pivot relation. The semantic key of the object permits any given instance of the view-object to be uniquely identified. An instance of a view-object is generated from the outer join of the tuples of the corresponding relations. The hierarchical structure of each instance is in accordance with the view-object template tree.

Candidate Set. When a user selects a pivot relation, an important issue is to specify the "sphere of influence" for this pivot relation, that is, to specify which relations are near enough to the view-object's pivot relation that they should be presented to the user for inclusion in the view-object being defined. In Penguin, this is done by using an information-metric function that takes into account the type of each connection traversed and their combinations in a depth-first traversal of the tree (while avoiding cycles), and limiting the path length so that it does not exceed the threshold when measured using the information-metric function. For this purpose, all paths between two relations in the structural model are considered, independently on the direction of the arcs (connections) on the paths. The relations relevant for the pivot relation R are then kept in a *candidate set* that specifies the nodes of a subgraph $G(R)$ of the directed graph G_s defined by the structural model (see Figure 2(A)). The metric-function used in Penguin is described in [3], where a formal definition of a candidate set is also given. The purpose of the metric-function is to manage the scope of information presented to the user, as well as to reduce the size of the computation in defining a view-object.

Candidate Bag. Once a candidate set $G(R)$ is constructed, the problem arises that some relations in the candidate set may be reached using multiple paths from the pivot relation. The *candidate bag* for R contains all the relations of R's candidate set, the multiplicity of each relation Q in the bag being the number of valid paths from R to Q in $G(R)$. Relations contained in the candidate bag can be arranged in a unique hierarchical structure $T(R)$ called the candidate tree of R (see Figure 2(B) where *people* is replicated because there are two valid paths from the pivot relation *course* to the secondary relation *people*). Using a depth-first traversal of the candidate set graph starting at the pivot relation, we replicate the nodes of the graph for each path to create a tree. This operation is related to creating a spanning tree of a graph, where each node is retained but non-essential arcs are removed; here each arc is retained and nodes are replicated in order to create a tree. Note that we may traverse an arc in either direction, so that there are six connection types of interest (the three connections defined in

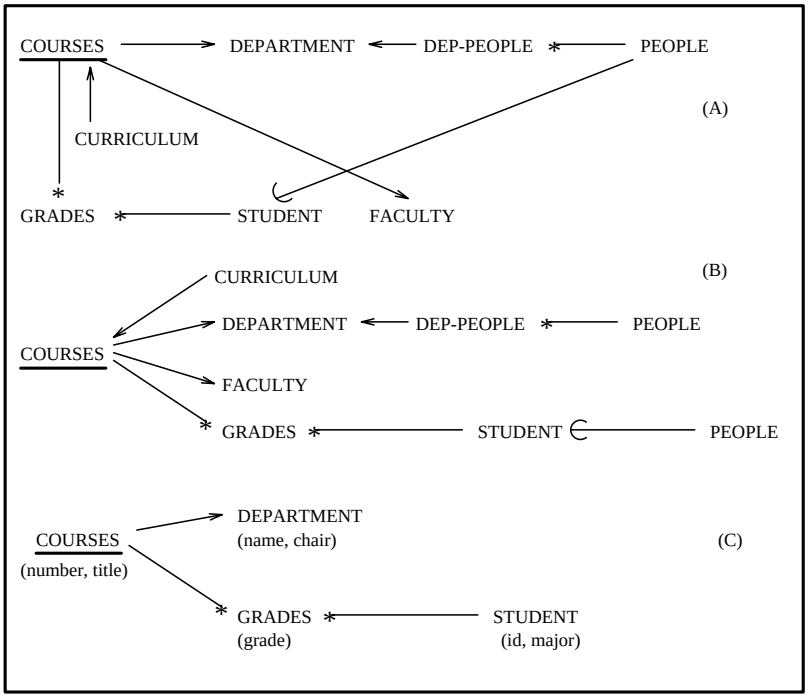

Fig. 2. Definition of a view-object: candidate set, candidate bag, and view-object.

the structural model, plus the three inverse connections obtained from traversal in the reverse direction). Thus, multiple copies of a non-pivot relation can be included in one view-object. The formal definition of a candidate bag and its transformation into a candidate tree is presented in detail in [3]. The candidate bag forms a *covering tree* of the candidate set.[2]

View-Object Template Tree. Once a pivot relation is chosen by the user, a unique candidate tree is generated by Penguin: It defines all possible configurations for a view-object template anchored on this pivot relation. The user defines a specific hierarchical structure for a view-object by selecting nodes (relations) and atomic attributes for each node (see Figure 2(C), the attributes selected for each node of the template tree are shown in parentheses). The subset candidate tree selected by the user is represented as a view-object template. However, a view-object template not only contains the relations for the object and the relationships among them, but also information on how to select and shape the view-object instances. The necessary joins and projections are only dependent on the object structure. Together, they define the *data access function* (DAF) for the object, that is computed at the view-object definition time. The DAF

[2] This name was suggested by Anne C. Elster of Cornell as the analog of a spanning tree.

contains information that permits the creation of a SELECT-FROM-WHERE block of an SQL query at the view-object instantiation time. While computing the DAF, the Penguin system checks for atomic attributes that are either missing or redundant—the atomic attributes must allow the system to identify the tuples for each relation comprising the view-object—and for relations that have not been chosen by the user but are necessary to the join expression.

Defining Inheritance of View-Objects. Multiple view-objects can be anchored on the same pivot relation and can have in common some nodes of the candidate tree. Using the inheritance property, view-objects anchored on the same pivot relation can be organized in a IS-A hierarchy. Inheritance in Penguin is used to define the type of the view-object. When the user chooses the pivot relation, he can inherit from existing view-objects anchored on the same pivot relation, and select one or more view-objects according to his needs. Some of these view-objects may belong to the same inheritance path; Penguin keeps in this case the most specialized view-object. In a case of multiple inheritance, a relation attribute of a tuple may appear in different inheritance sources and be defined differently. Penguin keeps the most specialized relation that will be finally inherited by the subtype.[3] The user can then define additional information specific to the new view-object. At this stage, Penguin guarantees the following:

- The subtype always contains all the relations of the parent type organized according to a similar nested structure.
- For each relation common to the two view-objects, the relation of the subtype must contain at least all the attributes of the corresponding relation of the parent type.
- The subtype always contains all the references of the parent type.

Thus, a view-object anchored on the pivot relation R can inherit relations and references defined in existing view-objects also anchored on R[4]. Additional information can be added such as new atomic attributes for the inherited relations, new relations coming from the candidate tree rooted on R, and new view-object references.

Defining References between View-Objects. A view-object anchored on R can refer to one or several view-objects whose pivot relation is contained in the candidate bag of R; it can also be itself the target of many references. Thus, the origin of a reference link is a view-object whose template tree contains a relation that serves as a pivot relation for the referenced view-object. This guarantees that, for a particular node's instance value in the referencing view-object, we

[3] Object instances created by an application program are created using the correct type. We envision that when data is fetched from the database, a test will be made to determine the correct type of the object to be instantiated. This test will be described declaratively or by a programmer-exit routine.

[4] It is also reasonable for an IS-A graph to include view-objects pivoted on a relation related by a subset connection to R.

can rapidly retrieve the corresponding instances attached to the referenced view-object and present them to the user according to their nested pattern. The user can therefore create a navigational structure convenient for an application, that is, view-object PART-OF graphs that make explicit the general data structure of the application. Navigation among view-objects is similar to navigation in the structural model or among object classes in C++.

3.2 Object Schema of an Application View

The view-object model enables users to build object schemas that are most convenient for different applications reusing the same collection of data. The semantics of shared data is captured in the structural model. View-objects represent hierarchically portions of the network described as a structural model and the relations that comprise them are composed through relational joins. For a specific application, the user defines complex object classes (view-objects) as well as IS-A and PART-OF classes graphs. The definition of an application view is language independent, because it is based on the structural model that plays the role of an integrated data model on top of the conceptual data model.

As stressed in [30], many problems come from the integration of database techniques and object-oriented programming languages. We argue that our approach must not only be seen as a means to favor data sharing among applications but also as a means of reconciling database and object-oriented approaches by keeping the best of each of them. In the following section, we present how Penguin generates language-specific code for the C++ binding to define the object classes describing the view-objects and basic operations on them (creation, navigate, browsing, and update).

4 Generating C++ Code for an Application View

C++ [28] appears to be a *de facto* standard in the object-oriented world, and a growing number of object-oriented databases offer this language to directly specify classes and methods. However, in these systems, a database is created from C++ classes, which can make the reusing of some parts of the database for other application programs difficult. The problem is that the design of an object class schema convenient for a particular application may not be convenient or easily extendable to support the requirements of future applications. This problem is particularly tricky because of the semantic richness of the object-oriented concepts and the limited experience the community has in integrated object schema from independently developed applications. The Penguin system uses another approach that consists in generating C++ classes convenient for each application, from an integrated structural model. An additional layer has thus been added to the Penguin system to make possible the automatic generation of C++ classes. These classes can be used in applications that want to access the data and then present it in the form of C++ objects.

4.1 Use of Two Layers for Mapping

In Penguin, the C++ mapping uses a two-layer approach to support sharing. In the lower layer, C++ classes are defined for each of the relations in the structural model, with the appropriate connections between classes. Each class is derived from one base relation and contains an attribute for each of the base relation's attributes in addition to an attribute for each connection or inverse connection involving the base relation. In the first case, the type of the attribute is a simple one (e.g., integer, real, char); in the second case, the type of the attribute corresponds to the name of a C++ subclass. For the relation classes, Penguin generates C++ methods responsible for instantiation, browsing and updating.

In the higher layer, C++ subclasses are created to define the view-object mapping. These subclasses make visible only those attributes and connections that the view-object preserves at its level. More precisely, to each view-object corresponds a collection of C++ subclasses, one subclass representing a relation involved in the view-object template. These C++ subclasses are actually organized into an aggregation hierarchy whose root is the subclass representing the pivot relation of the view-object; the remaining subclasses are located at the other nodes according to the view template tree defined by the user. Furthermore, the arcs of this C++ tree correspond to the connection attributes. We can note that the leaves of the tree actually represent subclasses that do not have connection attributes. Thus, there is a one-to-one mapping between the view-object template tree defined by the user and the corresponding C++ aggregation hierarchy created at the C++ layer.

In each subclass, simple attributes and connections are inherited from the class representing the entire relation. However, a subclass represents the information the user needs. For this purpose, *get* and *set* methods are defined for those attributes only; they permit the corresponding data to be accessed or updated. These methods are automatically generated by the Penguin system. C++ classes can be seen simply as structures generated for users to use in their programs. The user can specialize C++ classes and customize them for example by adding new attributes and new methods.

4.2 The Mapping Process

Since there are two levels of mapping, there are actually two mapping processes that occur. The first is the mapping of the semantic model onto C++ classes. This mapping creates a collection of classes that point to one another as needed to realize the connections between the relations. The mapping is done on a relation-by-relation basis and the class name corresponds to the relation name. Let us consider the structural model of Figure 1. We can get from this model the C++ classes named *courses*, *department*, *people*, *curriculum*, *grades*, *student*, *faculty*, and *staff*. For example, the class *courses* will be defined as shown in Figure 3(A).

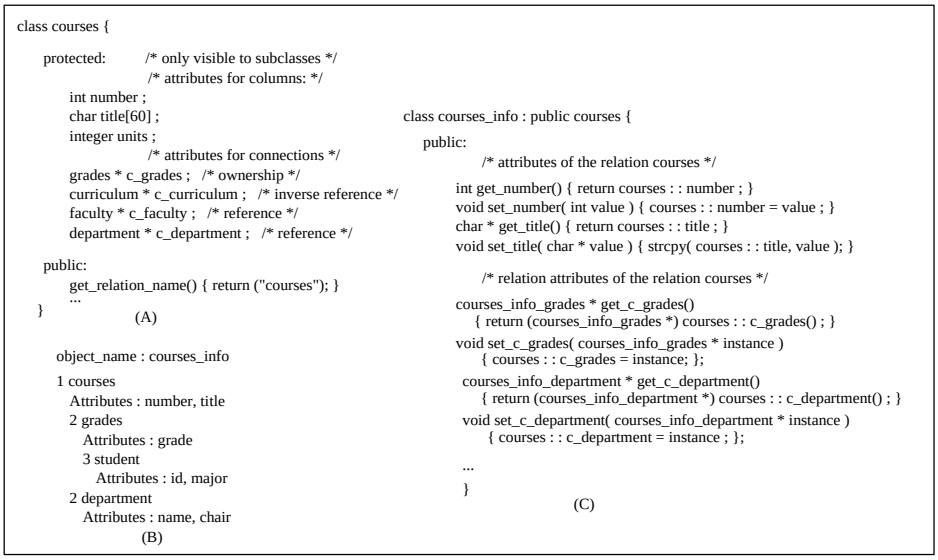

Fig. 3. C++ mapping for the relation *courses* and the view-object *courses-info*.

The second mapping process goes from the view-objects onto C++ classes that are subclasses of the classes defined for the semantic model. There is a starting point for each view-object: its root (i.e., its pivot relation). The class name corresponds to the view-object name. The view-object of Figure 2 has its template depicted in Figure 3(B). To each relation included in this view-object corresponds a C++ class whose name consists of the name of the view-object and the name of the relation. Each C++ class specifies *set* and *get* methods for the atomic and relation attributes that are visible in the corresponding relation. For example, Figure 3(C) shows part of the C++ specification for the view-object *courses-info*. Note that the *get* and *set* methods of the class *courses-info-grades* apply to the atomic attribute *grade* and the relation attribute *student*. Finally, the classes *courses-info-department* and *courses-info-student* access and update only atomic attributes that are respectively *name*, *chair*, and *id*, *major*.

4.3 C++ Level for Navigation, Query, and Update

The functions offered by Penguin to create, browse, and update view-object instances can be invoked by the C++ methods automatically generated during the mapping process. The methods *first*, *next*, *previous*, and *last* deal with the browsing of each view-object's instance. The methods *insert*, *delete*, and *replace* enable the user to insert, delete, or replace values that correspond to relations instances; they take into account update rules specific to each view-object and specified on it.

The *instantiate* method invokes the Penguin functions responsible for re-

trieving all the instances of a view-object. An instance of the view-object is then copied in the C++ objects organized according to the aggregate hierarchy defined by the instance of the C++ subclass. These C++ objects point to (sub)tuples values of the current view-object instance. Each call of a browsing operation returns a new view-object instance that replaces the previous one in the C++ structure.

5 Penguin Architecture

Since generality and portability are major objectives, the C programming language and the SQL language have been used to implement the Penguin system. The current implementation runs on UNIX with data stored in Ingres. Penguin also has been ported to Oracle. Penguin was originally written on VAX VMS using DEC's RDB.[5] Currently, this system has a four-layer architecture:

- The physical layer. Data are stored in relations in a relational database. This layer handles the storage requirements for Penguin.
- The relational layer. This layer is concerned with the semantic model that augments the relational model by adding connections between relations.
- The object layer. It corresponds to the view-object model. Penguin provides functions for creating, instantiating, updating, and browsing view-objects.
- The C++ layer. C++ classes are automatically generated by the Penguin system to provide a programmatic interface on top of Penguin for creating, instantiating, updating and browsing view-objects. This C++ mapping preserves the information-sharing features of Penguin.

The software components of the Penguin system have been defined in a modular way and are organized around six *agents*. The *call interface agent* creates a message-passing paradigm that provides interagent communications. The interaction between an application using Penguin (such as the generated C++ interface code), and the Penguin system is controlled by an *application interface agent*. In this section, we present the role of the four other agents and describe the module responsible for the C++ application generation.

5.1 Structural Model Agent

The *structural model agent* is responsible for the creation, the destruction and the display of connections between relations. The ownership, reference, and subset connections enable the user to capture all the potential links existing between data. This implies that a view-object template generated from the structural model defines a hierarchical structure that is complete for this view.

[5] Detailed documentation of the Penguin system is in [6]. A description of the theory of Penguin is contained in [5, 3]. Some other papers on the Penguin project are also in the bibliography [4, 20, 34, 35].

5.2 Template Generation Agent

This agent is responsible for the creation of a view-object template and the definition of the update, insert, and delete operations of the view. A template is defined as described in Section 3, generated from a candidate tree rooted on a pivot relation. Since the user can include any number of attributes for each relation in a view-object, this agent has to test for attributes that are either missing or redundant. The joins and projections that are necessary to get the view-object instances are dependent on the view-object structure and lead to the construction of the DAF for the view-object.

5.3 Object Instantiation Agent

This agent performs several tasks with the purpose of instantiating a view-object. It offers the user an Object Query Language [29] that has an SQL-like syntax and ensures the translation of a user object query into a relational SQL. Since the target of an object query is a view-object, the generated SQL query is based on the DAF of this view-object eventually augmented by additional predicates. This agent handles building the SQL query and passes it down to the Penguin database where the query is executed. The result is a collection of flattened tuples that are then organized into a hierarchical, nested, traversable structure according to the template tree. The resulting view-object instances are cached in main memory and bound to the corresponding view-object. The restructuring process, based on the approach developed in [21, 22], eliminates duplicate subtuples and null subtuples.

In the current Penguin system, each time a view-object is instantiated, the relevant instances are retrieved from the remote database and replace the old instances. By keeping the resulting tuples in the local memory for an extended period of time, we could reuse the cached tuples to locally answer queries and thereby avoid database accesses. To do that, we have defined a two-level client-side cache for composite objects mapped as views of a relational database[11]. The lower level of the cache contains the tuples from each relation that have been loaded into memory already. These tuples are linked together from relation to relation according to the joins of the structural model. This level of the cache is shared among all applications using the data on this client. The higher level of the cache contains composed objects of data extracted from the lower level cache. This cache uses the object schema of a single application, and the data is copied from the lower level cache for convenient access by the application.

5.4 Instantiation Navigation Agent

This agent is responsible for view-object instances browsing and updating. The browsing operations (*first, last, next, previous*) use the hierarchical structure of the nested sets of cached tuples. Navigation among view-object instances exploits the reference links defined between the view-objects. This agent also offers functions to perform instance modifications (delete, insert, and update).

It checks for consistency whenever a modification of a view-object instance is done. The integrity rules of semantic connections and the update rules specific to each view-object control the impact a user can have on the shared data of the underlying relations. The approach chosen to handle updating through view-objects [5] extends previous work on relational views [12, 13, 14, 16].

5.5 C++ Application Interface Generator

This module generates a C++ application interface from the structural model and an application view as described in section 4. A generic class *Root* common to the relation classes is created to specify generic methods that can be redefined at a lower level. A view-object is represented by an aggregate hierarchy of C++ classes whose root refers to the view-object's pivot relation.

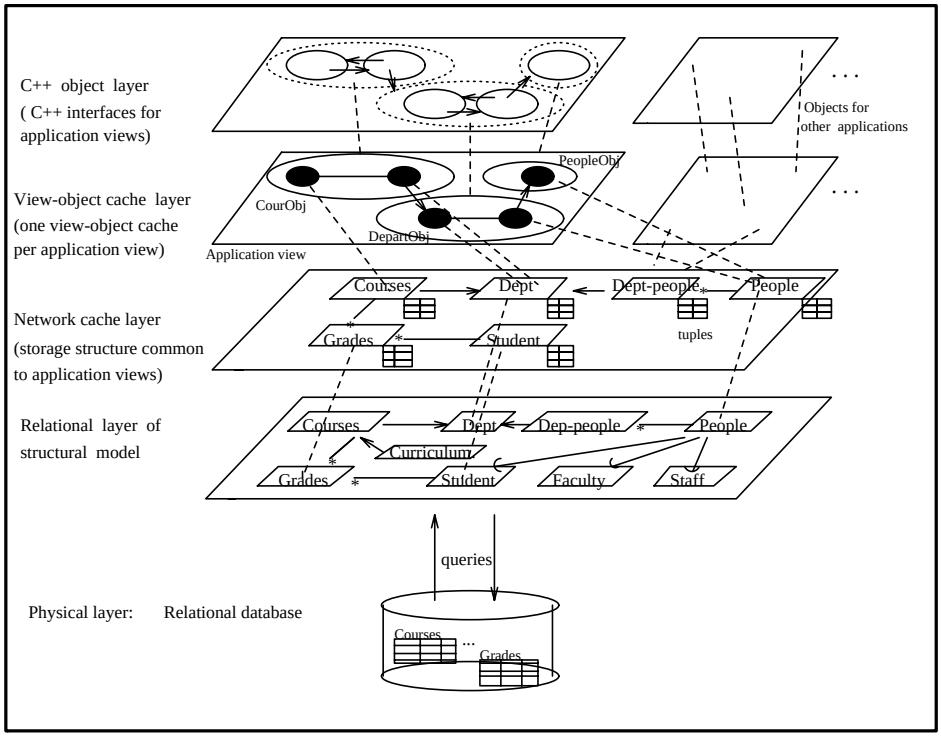

Fig. 4. The different layers of the Penguin system.

The C++ classes define the basic behavior (methods) of view-objects, that enables the user to create, browse and update view-objects instances. In this last case, C++ methods invoke Penguin functions that propagate changes in the cache and on the shared database. Cache consistency maintenance on the

different client sides and client notification are part of ongoing research. Figure 4 shows an overview of the different layers in the Penguin system along these lines.

6 Ongoing and Future Work

We are currently using Penguin to demonstrate how independently developed applications with differing object schemas can be integrated. We have taken two applications with C++ object schemas, converted each of these schemas into a structural model, integrated the structural models, created a relational database to support these applications, and are now defining the object classes and porting the applications to use Penguin through these object classes.

We are also investigating new caching techniques using a predicate-based approach. This approach is important when objects are cached based on their contents, rather than by object ID. Our approach allows clients to cache data bound into the form convenient to the application, while minimizing communication with the server and supporting concurrency control. Our approach will allow queries to be executed locally if all data is available. As part of this research, we are investigating query languages that can be executed locally or decomposed into parts executable by a relational database to obtain data not available locally.

7 Conclusion

In this paper, we have explained how two fundamental concepts of the Penguin system aim at promoting data sharing between multiple object schemas and a relational model. Our approach allows the co-existence of legacy applications and databases using the relational model and new applications written using the object-oriented approach. Furthermore, our approach is compatible with object-view integration, a demonstration of which is in progress.

The first concept is the Structural Model that uses the concepts of aggregation, categorization and abstraction to explicitly express semantics relating normalized data. These connections specify precise integrity rules that are used to control the effects of view-object updates on the base relations. Both connections and inverse connections are considered when view-objects are created from the structural model. They are used to define the correct hierarchical arrangement of view-object's instances. The View-Object Model constitutes the second main component that allows the user to define application views consisting of view-objects organized into PART-OF and IS-A graphs. View-objects support the definition and the manipulation of complex data structures as well as data sharing between different applications.

Penguin generates C++ code defining the C++ structure and the basic behavior of an application view. This C++ mapping maintains the information sharing feature of Penguin and offers a high-level interface for data creation, browsing and update.

8 Acknowledgements

This effort was supported in part by the Microelectronics Manufacturing Science and Technology project as a subcontract to Texas Instruments on DARPA contract number F33615-88-C-5448 task number 9. The work of Catherine Hamon is supported by a postdoctoral fellowship supported by the French government.

The work reported herein was implemented by numerous students and visitors. Amelia Carlson implemented the generation of base C++ classes. Leonid Karasik implemented the specification of composite Penguin classes with inheritance and references, and generation of C++ class definitions. Tetsuya Takahashi defined and implemented the query system for the C++ binding. Tahir Ahmad, Larry Safran, and Srinivasan Venkatesan implemented other parts of the C++ binding for Penguin. Earlier development of the Penguin system includes work by Gio Wiederhold, Kincho Law, Thierry Barsalou, Niki Siambela, David Zingmond, Harvinder Singh, and Byung-Suk Lee, among others. We also thank Marianne Siroker for her assistance in the preparation of this paper.

References

1. S. Abiteboul and Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *Journal of Computer and System Sciences*, December 1986.
2. T. Andrews and C. Andrews. Combining Language and Database Advances in an Object-Oriented Development Environment. *Proceedings of OOPSLA*, Orlando, Florida, 1987.
3. T. Barsalou. *View Objects for Relational Databases.* Ph.D. dissertation, Stanford University, March 1990, technical report STAN-CS-90-1310.
4. T. Barsalou and G. Wiederhold. Complex Objects For Relational Databases. *Computer Aided Design*, Vol. 22 No.8, Buttersworth, Great Britain, October 1990.
5. T. Barsalou, N. Siambela, A. M. Keller, G. Wiederhold. Updating Relational Databases through Object-Based Views. *ACM SIGMOD*, Denver, May 1991.
6. A. Carlson. *Penguin System Internal Maintenance Specifications.* Unpublished document, October 1992.
7. R. Cattell. *Object Data Management: Object Oriented and Extended Relational Systems.* Addison-Wesley, 1991.
8. E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13(6), June 1970.
9. O. Deux. The Story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.
10. L. Haas, W. Chang, G. Lohman, J. McPherson, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.
11. C. Hamon and A. M. Keller. Two-Level Caching of Composite Object Views of Relational Databases. Submitted for publication, 1993.
12. A. M. Keller. *Updating Relational Databases Through Views.* Ph.D. dissertation, Stanford University, February 1985, technical report STAN-CS-85-1040.
13. A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Computer*, 19(1), January 1986.

14. A. M. Keller. Choosing a View Update Translator by Dialog at View Definition Time. *12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986.
15. A. M. Keller. Unifying Database and Programming Language Concepts Using the Object Model, (extended abstract). *Int. Workshop on Object-Oriented Database Systems*, IEEE Computer Society, Pacific Grove, CA, September 1986.
16. A. M. Keller and L. Harvey. *A Prototype View Update Translation Facility.* Report TR-87-45, Dept. of Computer Sciences, Univ. of Texas at Austin, December 1987.
17. A.M. Keller, R. Jensen, S. Agarwal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. *ACM SIGMOD*, 1993.
18. J. Kiernan, C. de Maindreville, and E. Simon. The Design and Implementation of an Extensible Deductive Database System. *SIGMOD Record*, September 1989.
19. W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.
20. K. H. Law, G. Wiederhold, T. Barsalou, N. Sambela, W. Sujansky, and D. Zingmond. Managing Design Objects in a Sharable Relational Framework. *ASME meeting*, Boston, August 1990.
21. B. S. Lee and G. Wiederhold. *Outer Joins and Filters for Instantiating Objects from Relational Databases through Views.* Center for Integrated Facilities Engineering (CIFE), Stanford University, Technical Report 30, May 1990.
22. B. S. Lee. *Efficiency in Instantiating Objects from Relational Databases Through Views.* Ph.D. dissertation, Stanford University, December 1990, technical report STAN-CS-90-1346.
23. D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. *Proceedings of OOPSLA*, Portland, Oregon, 1986.
24. Z.M. Ozsoyoglu and L.Y. Yuan. A New Normal Form for Nested Relations. *ACM Transactions on Database Systems*, 12(1), 1987.
25. P. Pistor and F. Andersen. Designing a General NFNF Data Model with an SQL-Type Language Interface. *Twelfth International Conference on VLDB*, Kyoto, Japan, 1986.
26. M. A. Roth, H. F. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM TODS*, 13(4), December 1988.
27. M. Stonebraker. Object Management in Postgres Using Procedures. *On Object-Oriented Database Systems*, Springer-Verlag, 1991.
28. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.
29. T. Takahashi and A.M. Keller. Querying Heterogeneous Object Views of a Relational Database. Submitted for publication, 1993.
30. D.C. Tsichritzis, T. Bogh. Fitting Round Objects into Square Databases. *OOPSLA*, New Orleans, 1989.
31. J. D. Ullman. *Principles of Database and Knowledge-Base Systems. Volume 1: Classical Database Systems*, Computer Science Press, 1988.
32. G. Wiederhold and R. ElMasri. The Structural Model for Database Design. *In Entity-Relationship Approach to System Analysis and Design*, North-Holland, 1980.
33. G. Wiederhold. Views, Objects and Databases. *IEEE Computer*, 19(12), 1986.
34. G. Wiederhold, T. Barsalou, and S. Chaudhuri. *Managing Objects in a Relational Framework.* Stanford Technical report CS-89-1245, January 1989, Stanford University.
35. G. Wiederhold, T. Barsalou, B. S. Lee, N. Siambela, and W. Sujansky. Use of Relational Storage and a Semantic Model to Generate Objects: The PENGUIN Project. *Database '91: Merging Policy, Standards and Technology*, The Armed Forces Communications and Electronics Association, Fairfax VA, June 1991.