Penguin: Objects for Programs, Relations for Persistence

Arthur M. Keller and Gio Wiederhold
Computer Science Dept., Stanford University

September 1999

Abstract

Penguin is designed to support object-orientation for application programs while using relational databases as the persistent backing store. Objects are attractive to customers and programmers of applications because they structure information into a relevant and effective view. The use of relational databases that store of large amounts of base data for long periods of time enables Penguin to take advantage of mature solutions for sharing of information, concurrency, transactions, and recovery. We expect that application programs will be best designed with their own object schemata, so each object schema is supported as a series of views of the underlying relational databases. Penguin provides for multiple mappings to diverse object configurations, enhancing inter-application interoperation. This approach supports coexistence and sharing data among programs using relational technology with diverse application programs using object technology, as well as facilitating a migration to object technology.

1. Introduction

Penguin is an object data management system that relies on relational databases for persistent storage. Object views of the relational databases allow each application to have its own object schema rather than requiring all applications to share the same object schema [Wied86]. In this paper, we discuss the principles, architecture, and implementation of the Penguin approach to sharing persistent objects.

The primary motivation for using a database management system (DBMS) is to allow sharing of data among multiple customers and multiple applications. To support sharing among independent transactions, DBMSs have evolved services including transaction independence, persistence, and concurrency control. When a database is shared among multiple applications, these applications typically have differing requirements for data access and representation. Such differences are supported by having views, which present diverse subsets of the base data [ChamGT75].

The primary motivation for defining objects is to include sharable semantics and structure in the information. Must all applications sharing objects use the same object schema, or is it better to give each application its own object schema and somehow to integrate them? If multiple applications differ in view the needed compromise reduces the relevance and effectiveness of the object representation [AbitB91]. For instance, `customers` will have an orthogonal view of an `inventory` versus the `suppliers`.

When combining independently developed applications, we do not have the luxury of choosing a common object schema. Many legacy databases and legacy data are still being used. We must retain the investment in existing application software and databases, while building new software using the object approach. When creating a federation of heterogeneous (pre-existing) databases, we must support a degree of interoperation among these databases and their schemas. Consider also that current projects will become legacy in a few years hence, but their semantics will remain. Whatever solutions we create in shared settings must support evolution and maintenance.

Object-oriented database management systems (OODBMSs) define an object schema that is used to support all information for its applications. Typically, that object schema is designed for one initial application, and other applications that want to share the information will have their needs *grafted on* [TsicB89]. To us, it appears preferable for each application to use the object schema most convenient for it. Typical OODBMSs do not support mappings between object schemas. Encapsulation provides data independence only from outsiders, but does not support integrating new applications.

Database development was greatly influenced by the three-level data modeling architecture from the ANSI/SPARC committee [Steel80]. Their model partitions the concerns of information management into a view level, a conceptual level, and a physical level. The view level is comprised of 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. Today, in a distributed environment, there may not be a single conceptual model, but rather one for each configuration of databases used in a set of applications. The physical level describes the implementation of the database at an abstract level, including distribution, data structures, and access paths. Most RDBMSs , support the view level, and many views may be defined corresponding to various user roles. In contrast, OODBMSs typically do not support the view level; there is only one view corresponding to the conceptual level [ShilS89].

The Penguin approach builds on mature relational database management system (RDBMS) technology by creating object views. Our techniques could be used to create object views of alternate object databases, but we have not yet done this work, so it is beyond the scope of this paper.

2. Mapping from the Object Model to the Relational Model

An object schema defines data elements as well as its structure. Object classes may have single or multiple inheritance. Object instances within an object class may have internal structure, such as object classes being composed of other object classes by reference or embedding.

Let us consider a database design approach, in which multiple object schemas and schemas of legacy databases are mapped into the relational model, and then integrated into a common database schema. Figure 1 illustrates this approach.

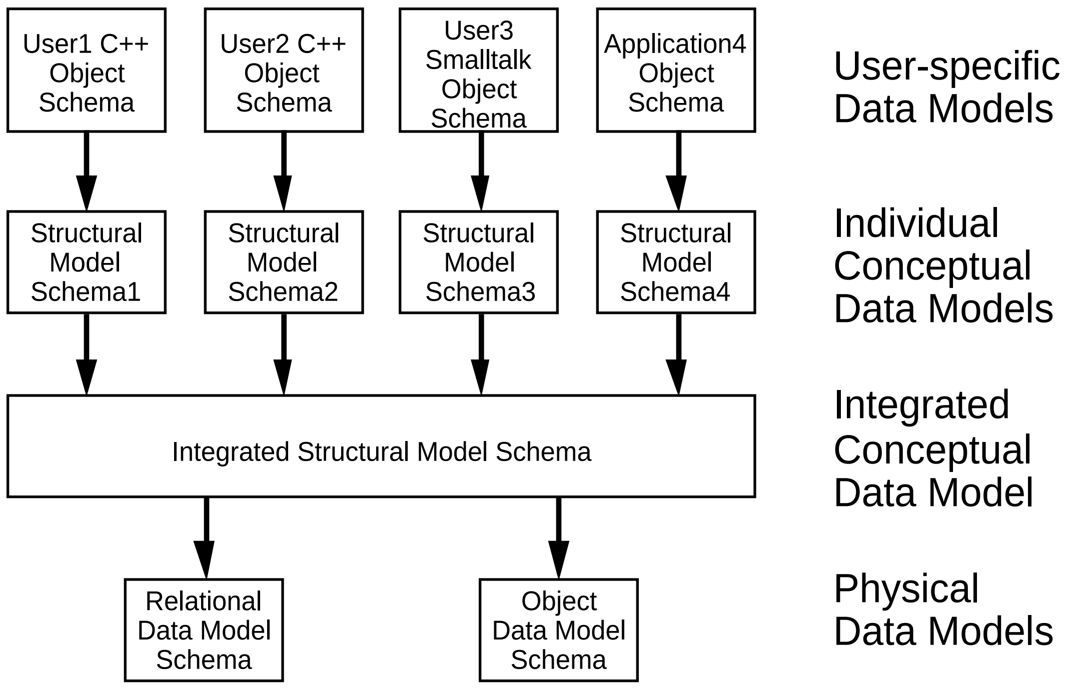
User1 C++ Object Schema
User2 C++ Object Schema
User3 Smalltalk Object Schema
Application4 Object Schema
User-specific Data Models
Structural Model Schema1
Structural Model Schema2
Structural Model Schema3
Structural Model Schema4
Individual Conceptual Data Models
Integrated Structural Model Schema
Integrated Conceptual Data Model
Relational Data Model Schema
Object Data Model Schema
Physical Data Models

Figure 1. Database Mapping and Integration.

Object schemata have semantics. Inheritance and composition hierarchies organize elements so that they are convenient for the current task. Object methods contain further semantics but these are not needed for mapping to the relational model nor for integration. Relational structures have few semantics: there is no defined hierarchy. Rather the elements are composed as needed by the application through views and queries.

To perform a mapping from object schemata to a relational model, we integrate the object hierarchies into a semantic model network, which is more general than a hierarchy. The semantic model we use is the Structural Model, which only describes systematic relationships, and does not require separate instance storage over that in the relational database [WiedE80]. Additional classes can be defined within this framework by augmenting the model, Storage of data elements from the object is accomplished by creating base relations as needed. If appropriate relations exist they can be augmented and connected.

Once the base relations are populated, instances of links can be computed from the model and the stored base relations. The Structural Model can reconstruct component object classes as well as new potential object classes by recognizing hierarchies within its network. New classes can be created as well, by and connecting them to each other and to relevant existing relations.

For each of the input object schemata, the Penguin approach converts the inheritance (IS-A) and composition (PART-OF) structures into directed graphs. To handle inheritance, horizontal and vertical partitioning are supported. Also supported is the option where (part of) an inheritance hierarchy is stored in a *universal relation* containing all attributes in the hierarchy along with an attribute indicating the type of the object instance [Ullm83]. We normalize object nodes into Third Normal Form (3NF) using standard relational database design techniques. The result is a network representation.

The Structural Model recognizes various types of relationships, formalized into 3 types of *connections*. A Structural Model is a graph where each node represents a relation and each edge represents a connection (or relationship) among two relations. This model is comparable to the Entity-Relationship (E-R) Model, but uses the relational model as a base. Connections require a feasible join path. Penguin also defines an information metric which assigns strengths to each combination of each direction of the 3 connections types. The types are

(1) *Ownership connection*, which links instances that depend on the owner. The linked instances disappear when the owner is removed. For example, `Equipment` owns its `Maintenance` records. Its cardinality is *n:1*. This connection type is also used to represent part-of semantics, as `vehicle` has `engine, wheels`, etc. Penguin considers ownership a strong relationship (0.9), and being owned weak (0.3).

(2) *Reference connection*, which links to abstract entities. The referenced entities must exist while there are references to them. For example, `Hazard` refers to `Hazard Codes`. Its cardinality is *1:n*. Reference connections are weak (0.3, 0.5).

(3) *Subset connection*, which links a generalization to a subtype. The subtype instance disappears when its generalization instance is deleted. For example, `Resource` is linked to `Equipment` (because `Equipment IS-A Resource`). Its cardinality is *1:1 partial.* The selection may be based on a predicate. Subset connections have strengths weaker than full ownership (0.75, 0.2).

The *m:n* relationship of an E-R model is modeled by pairing two structural connections, since in relational implementations the instances of *m:n* links are explicitly represented in a joining relation. For example, a relation `Equipment-location` is needed to represent such a relationship, and may use an ownership connection from `Equipment` and a reference connection to a `Location` relation [WiedE80].

Once we have a structural model for each input schema (object, relational, or legacy), we can proceed to integrate these schemata. There has been extensive research on integrating relational database schemata or E-R schemata [NavaEL86], and little work in integrating object database schemata [SaltR:97]. Using the Penguin approach, research results on integrating relational and entity-relationship schemata becomes applicable to integrating object schemata. In addition, there is even some work on integrating structural models that is directly applicable.

An integrator identifies and coalesces common elements (relations and connections). Problems occur due to semantic heterogeneity among the input schemata, such as differences in naming, value representation, schema, and data semantics. To solve these, it may be necessary to add mediating services, creating intermediate connections and relations. Such research is being undertaken at many sites, for instance [HammMG97].

Warehouses provide high performance through physical integration and replication of data [QuaW97]. However, physical integration of the component databases is not needed, as long as access paths to data instances, direct or mediated, can be provided.

Relational Database Storage and Semantic Modeling.

The relations of the integrated structural model are ordinary 3NF relations representing entities. Actual data instances are stored as tuples in a relational database system. Connections in the structural model are relationships among those relations to which formal semantics have been attached. Connections instances are represented as matching values in the connected relations, they are not stored as instances. View instances can be created by joining sets of tuples along these connections.

We will discuss concurrency control and transaction processing below after we discuss caching for Penguin. Note that data partitioning and replication approaches to handling multiple persistent data stores is orthogonal to the issues of having an object layer above the DBMS.

Defining Object Views of Relational Database

In this section, we describe the definition of object classes (view-objects) in Penguin as object views of the underlying persistent data store [BarsW90]. The input is the integrated structural model. We will refer to the person controlling this process as the object base administrator (OBA), corresponding to the term database administrator (DBA), the definer of database models and schemas.

There are three steps involved in the definition of a view-object

(1) Choosing a *pivot relation*. The entities that define the root for the desired object class are identified by determining a *pivot* relation (R). The pivot relation will have a one-to-one correspondence between tuples in the relation and object instances in the desired object class. The key of the pivot relation becomes the *semantic key* of the object class.

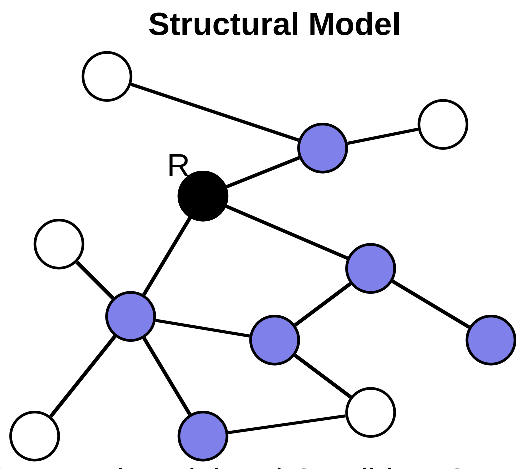

Figure 2. Structural Model and Candidate Set.

Once the pivot relation has been selected by the OBA, the Penguin system computes the closeness from the pivot to all connected relations, using the information metric values sketched above. The closeness is computed by computing the product of strength values associated with the connections along the paths. It then displays those relations that are *closer* in terms of the path than an OBA-defined threshold to the pivot relation. The *candidate set* consists of these semantically *nearby* relations. Figure 2 shows a structural model with pivot relation marked with R and the candidate set shaded. The ODB can eliminate relations containing irrelevant data from the candidate set, keeping the objects trim. Figure 3 shows the names of the relations in the candidate set.

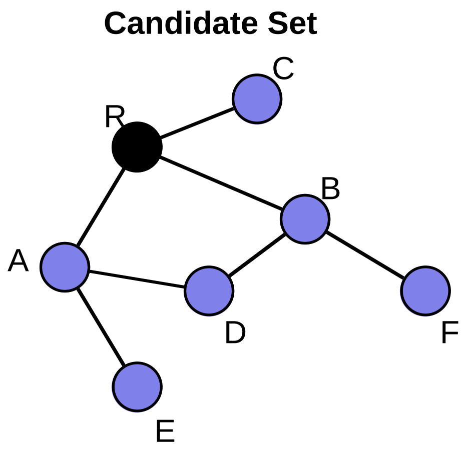

Figure 3. Candidate Set Labeled.

The Penguin system then converts the candidate set to a *candidate bag*. The candidate bag is a *covering tree* of the candidate set, that is, nodes creating cycles in the graph are replicated so that there is a copy of node for each acyclic path from the pivot relation. Edges are typically not replicated. Figure 4 shows the candidate bag.

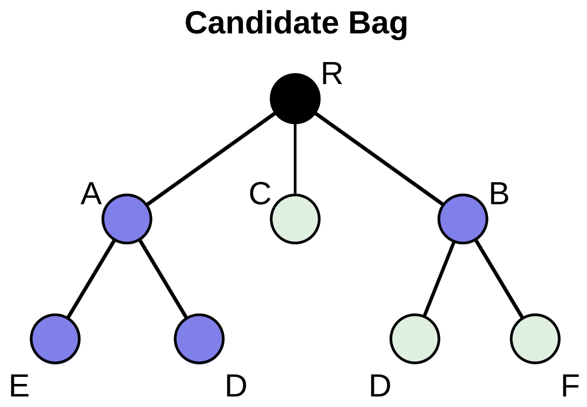

Figure 4. Candidate Bag.

(2) Choosing *instance variables*. The tree now represents the object structure, containing all attributes from the candidate bag. Now the ODB can select

or deselect variables for the object class. Attributes values can refer to object sub-classes that have their pivot relations in the candidate bag, to support PART-OF hierarchies. Attributes can be inherited from other object classes to support IS-A hierarchies. Figure 5 shows the object class.

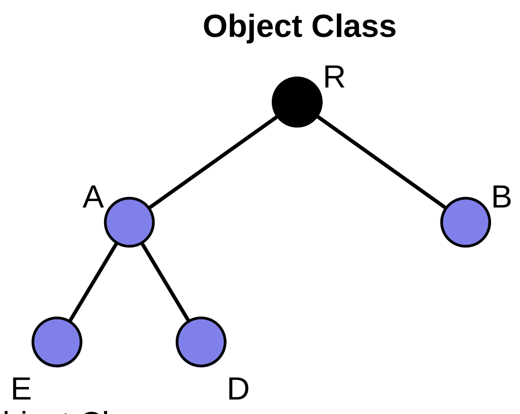

Figure 5. Object Class

(3) Specify *inheritance*. Next the ODB can indicate the object class(es) pivoted on the same relation from which this new object class should inherit. 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.) This approach is compiler independent. Figure 6 illustrates how a new object class inherits from object classes R1 and R2. Attributes in relations A and E are inherited from R1. A reverence to an object class pivoted on B is inherited from R2. Any new instance variables from relation D are added.

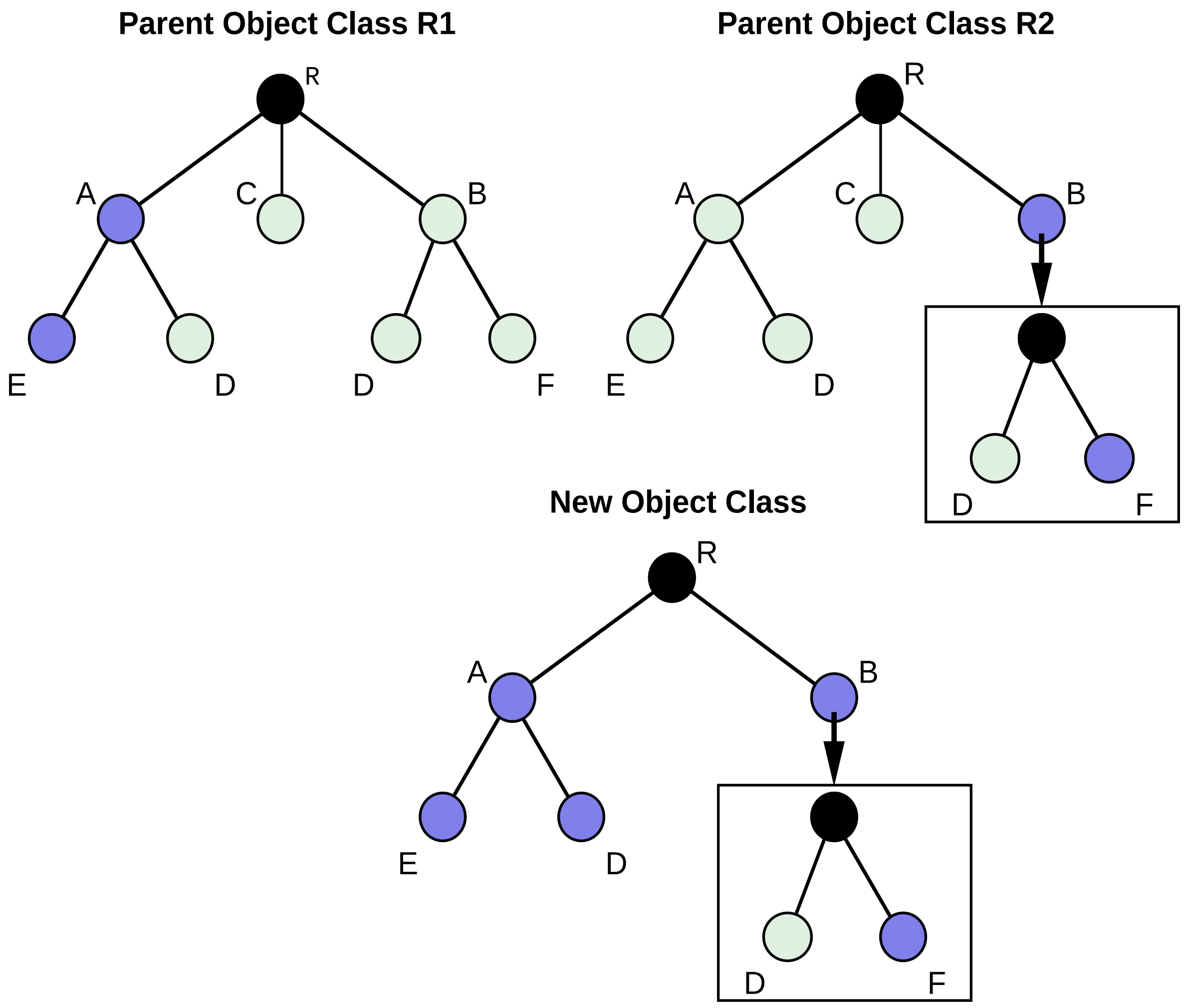

Figure 6. Inheritance Example.

Updating Base Relations from Objects

We have also developed methods that validate potential updates from the defined object types [BSKW91]. Here the ODB is informed of all possible update ambiguities, which typically involve understanding relations beyond those included in the object view [BlakCL89]. The ODB selects one of the alternatives, which choice is then to be recorded in the object schema [Kell86]. End-users are then not faced with update ambiguities at execution type. These methods have not been integrated into Penguin systems at this time, since implementations have focused on distributed resources and read-only object retrieval [LWBSSZ90]. All updates then originated at the sources. When the ODB has completed the definition of a Penguin object schema it can be made available to users and their applications. We have only created C++ object definitions, although the process is language independent.

Heterogeneous Data Sources

This data mapping is straightforward if the persistent store's schema is described by the structural model. However, we must handle legacy data as well. We handle legacy data by describing it as a structural model and wrapping the source to provide adequate access [Hamm*97]. The result is included as one of the constituent schemata input to the integration phase.

Wrapping source objects is relatively simple. Internal object semantics focus on ownership connections, while linkages between objects are commonly references. Multiple inheritance could require *m:n* relationships. Any Penguin objects which match one-to-one should be able to be copied directly from the sources, but we have not developed such a shortcut. Instead applications may prefer to retain direct access, which is not a problem if update is constrained to one of the access mechanisms.

Methods

Penguin generates basic methods for fetch, store, and update 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 change transactions.

Databases also support updates to sets of tuples and relations. Penguin could support a such an approach, which will allow changes to be made to multiple objects, and the corresponding changes could be made as one SQL statement to the base relations. Providing a method which can update more than object at a time would require generation of additional C++ code.

Penguin also supports 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.

Concurrency Control.

We need to coordinate concurrency control and transaction processing of the object layer and conceptual layer with the persistent data store. Thus transactions of the object layer are based on transactions of the persistent data store. Concurrency control in the object layer is based on concurrency control on the underlying data in the persistent data store. For example, the object layer can implement optimistic concurrency control on objects in an in-memory cache. When the object layer transaction commits, the transaction is validated, and a (distributed) transaction is invoked to commit the changes to the (multiple) underlying persistent data store(s).

3. Performance Issues

The conversion of relational base data into objects incurs a performance penalty. There are two aspects to reduced performance versus direct use of an OODBMS. First of all, the single application focus of the object paradigm allows substantial user control at the physical level [Wied87]. An RDBMS, focused on sharing, must compromise, and tries to overcome that compromise by extensive query optimization. Since Penguin supports sharing, that aspect is intrinsic.

The second aspect the additional cost of dynamic object creation. Here, many well-known techniques can be adapted, of which caching is the primary one. Since we expect to operate in a client-server environment, some data-shipping improvements can also make a significant difference [DeliR92].

Caching Structure

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 provides for much more efficient navigation [FranCL93]. Penguin improves performance in navigation by using a cache [WangR91]. We first discuss the organization of the cache, and then we discuss concurrency control and transactions.

Penguin uses a two-level cache. 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 ODB defined object schema. In a server-client architecture the caches would by physically distributed. There is also a virtual level of the cache that is language-specific.

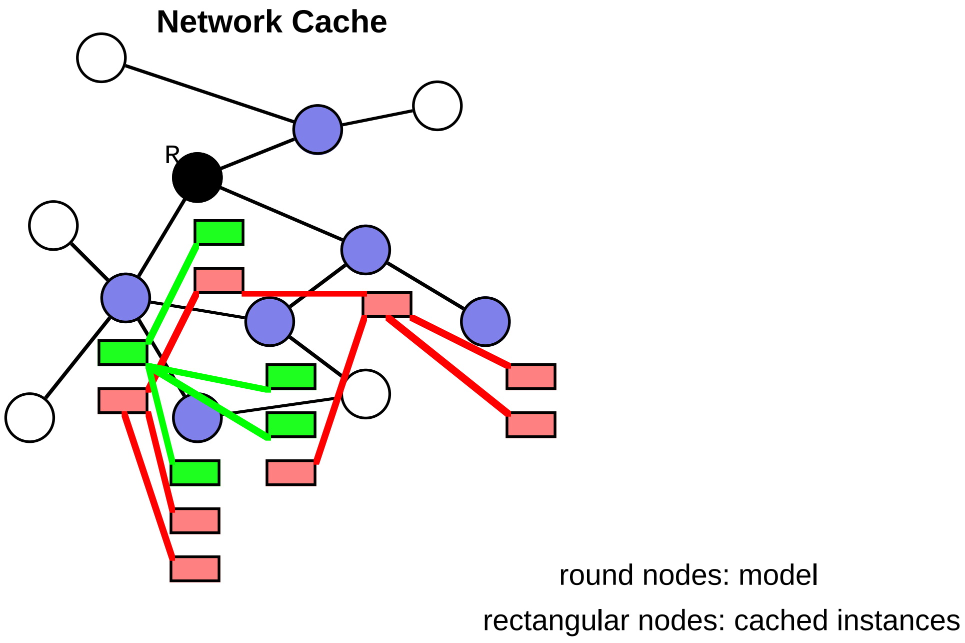

Figure 7. Network Cache.

The lower level of the cache contains a network representation that matches the structural model. In the cache tuples are linked together according to the joins based on the connections of their relations. We use *semantic pointer swizzling* to turn semantic key references (foreign key to primary key) into pointers in memory [WhitD92]. 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. Figure 7 illustrates the network level of the cache.

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 object cache is stored redundantly if necessary. The object cache is for a single application client and lives in the application's address space. Figure 8 illustrates the object cache.

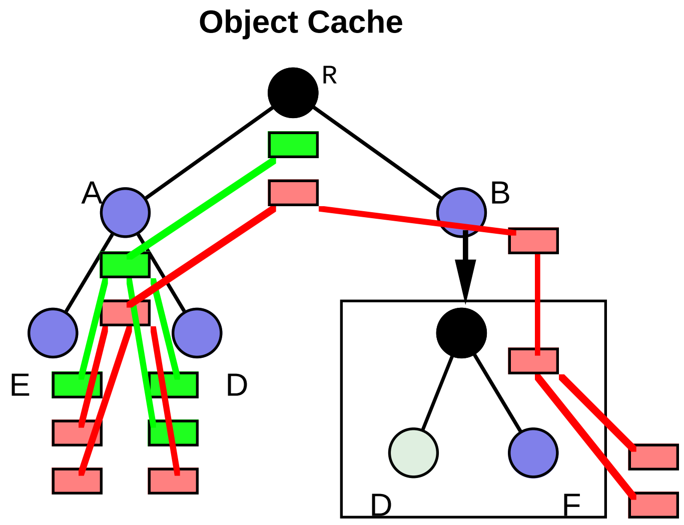

Figure 8. Object Cache.

The virtual level of the cache is language-specific. For C++, we need to specify the type of each object instance when there is multiple inheritance [Stro86]. To make an object accessible in C++, Penguin copies the data, if necessary, from the source data or the network cache into the object cache and determines the type based on the data contents. To determine the matching application type, a user-supplied method is required that refers to the data contents. Penguin then creates an object instance of the correct type and links the C++ object root to the corresponding data in the object cache [KellH93]. Figure 9 illustrates the virtual level of the cache. The ovals represent the C++ object of the correct type, which refers to the data in the object cache. The virtual cache is maintained in the application's address space.

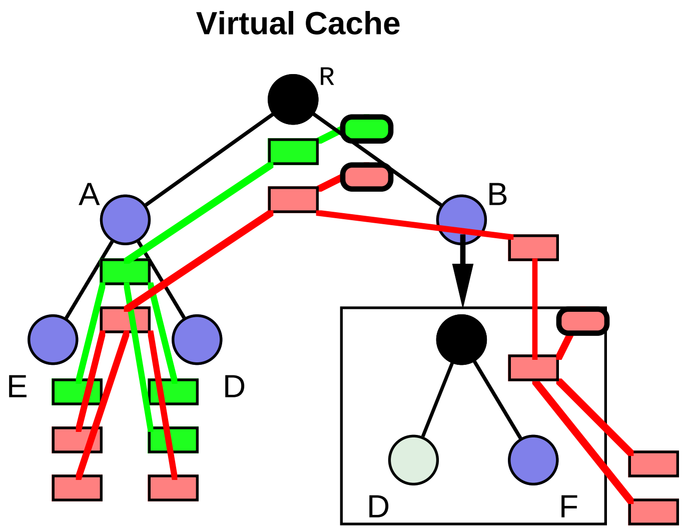

Figure 9. Virtual Cache.

Cache Management

To improve performance, Penguins loads the cache using relational, set-based queries and then navigates the cached data [DeWi*90]. Data in the cache is reused when issuing queries through the technique of predicate-based cache descriptions [KellB96]. These descriptions address two questions. First, do we know whether the desired query is entirely contained in the cache? Second, do we know whether we have the latest data in our cache [GuptMSD93]?

If we know that the desired query is entirely contained within the cache, then the Penguin client can avoid going to the persistent data store for data. We use a *conservative cache description*. That is, everything in the description is found in the cache, but the description can omit objects that are in the cache. If an object really in the cache is omitted from the conservative cache description, its processing will be non-optimal but not wrong. The conservative cache description is based on queries used to load the cache. It is used instead of the exact cache description because it can be simpler to use, although it requires some extra effort to maintain this cache description in a simple form. Queries contained in the cache can be processed (in memory) locally. Such processing requires that local indexes be built on-the-fly. That is, server data indexes are not used, so contention is reduced among multiple clients. If the query is not entirely contained in the cache, then the query must go to the server (persistent data store). If only part of the query is contained in the cache, the query may be trimmed to omit the part already cached if trimming speeds up server query processing or data transmission.

To support serializability, it is important to know that the cache contains the latest data [GrayR93]. If a client is the only client performing updates (or perhaps only to this part of the data), then the client has the latest data. The client can lock data at the server to ensure that no other client updates the data. If we choose to use optimistic concurrency control, when a transaction commits the updates are propagated to the server (the persistent store). The server uses a *liberal cache description* to determine which clients to notify. The description includes everything in a client's cache, but it can include objects not actually found in the cache. If an updated object is in the liberal cache description but not actually in the cache, processing will be non-optimal but not wrong. The liberal cache description is based on queries used to load the cache, and notifications of cache cleanup or flush.

Data transfer

In object-oriented client-server systems the overhead required for data transmission from server to client and the control of such transmissions is significant, As users of protocols as CORBA have noticed, fetching and updating data from many small objects kept at a remote object store is costly. Larger objects also incur costs due to pointer swizzling [CFLS91]. The Penguin approach can move the relational, value-based linkage references to the client, and perform all swizzling at the client.

We have analyzed three data transfer alternatives: objects, views, and relation fragments, and concluded that for a wide range of situations relation fragments are best [LeeW94]. Relation fragments are the selected and projected subsets of the base relations needed to construct the object view. They avoid the redundancy in relational views when owner tuples are replicated by the join operations which create views. While foreign and base keys are replicated, this cost is close to the information needed for swizzling. At the same time, the number if instances is much smaller, determined by the complexity of the query, than the number of object instances typically needed, which is determined by the cardinality of the pivot. A low cardinality greatly reduces transmission overhead.

4. Future Directions

The concepts developed in Penguin lead to many further alternatives. Object-oriented access is direct and fast. The generality of an RDBMSs requires more accesses and processing, reducing performance, which gives a significant incentive to using an OODBMSs. But our navigation in the structural model generalizes to navigation among object classes, allowing object-oriented storage as well. For instance, if one object model dominates, then mapping costs can be reduced by storage of the data a primary object database, while keeping the integrated structural model available for mapping to secondary object configurations, or even back to relations. A mapping to object instances would require that the object store supports a powerful query language, such as that of ODMG-93 [Catt91].

As RDBMSs move to hybrid configurations, they should evolve to provide more control over an increased variety of data structures. Then the performance advantage over OODBMSs will diminish. Unless OODBMSs provide increased view support, they will find it increasingly difficult to compete with enhanced RDBMSs.

The Penguin approach does not allow persistent object-identifiers. Cross references are hence always resolved with content-based queries. It is unclear to what extent this is a liability and should be addressed [KatoM92].

As with all independently defined objects, methods defined for one application's object schema are not applicable to another application's object schema, even if they share the same data. To increase sharing to more general methods than the access methods provided in Penguin, we propose that these methods be described declaratively. When the data schemata are integrated, the declarative method descriptions should also be integrated. When object classes are defined, the declarative method descriptions should be carried with them. Then, user methods could be generated. This process would mitigate maintenance costs incurred by sharing data among multiple applications. However, we have not explored these ideas in detail.

The interaction of storage granularity, data transfer approaches, and caching is complex and will be subject to ongoing analysis as the relationships of latency, bandwidth and user patterns changes. The development of data warehouses encourages much larger views to be transmitted for analysis, although much of the processing may be statistical, so that object concepts are less relevant, while data transpositions (i.e., storage by columns of tables) will again become important [Bato79]. Similar mapping notions as used in Penguin may be applicable.

5. Conclusion

The Penguin system supports multiple object views for multiple applications sharing data. All data is currently stored in a relational DBMS, but our approach extends to storing data also in an object DBMS. Our approach is based on a formal model of object views on relational databases. We propose an approach to object schema integration that takes advantage of the large body of work on relational and entity-relationship schema integration.

There is an operational prototype for Penguin. Some of the concepts have been developed and tested outside of that prototype. Furthermore, some of the ideas used in Penguin are reflected in commercial products currently on the market [KellJA93].

Acknowledgements

The Penguin project was the thesis work of Thierry Barsalou at the Stanford Section for Medical Informatics. Subsequent developments addressed issues in Civil Engineering and Integrated Circuit Manufacturing. Julie Basu is developed the caching and evaluated the caching algorithms with Meikel Poess while supported by Oracle Corporation. Many other students at Stanford contributed to these projects. The effort leading to this paper 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, and the Center for Integrated Systems. Earlier work leading to these concepts was supported by The National Library of Medicine (NLM) and by ARPA funding for Knowledge-based Management Systems (KBMS) (N039-84-C-0211).

References

[AbitB91] S.Abiteboul and A.Bonner: Objects and Views; *ACM SIGMOD Conf. on the Management of Data*, Boulder, May1991.

[BarsW90] T. Barsalou and G. Wiederhold: Complex Objects for Relational Databases; *Computer Aided Design*, Vol. 22 No. 8, Buttersworth, Great Britain, October 1990.

[BSKW91] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold: Updating Relational Databases through Object-Based Views*; ACM SIGMOD*, Denver, May 1991.

[Bato79] D.S. Batory: On Searching Transposed Files; ACM Transactions on Database Systems, Dec. 1979, vol.4 No.4. pages 531-544.

[BlakCL89] J.A. Blakeley, N. Coburn, and P. Larson: Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates; *ACM Transactions on Database Systems*, Vol. 14, No. 3, 1989, 369-400.

[CFLS91] M. Carey, M. Franklin, M. Livny, and E. Shekita: Data Caching Tradeoffs in Client-Server DBMS Architecture; *ACM SIGMOD Int. Conf. on Management of Data*, Denver, CO, May 1991, 357-366.

[Catt91] R. Cattell: *Object Data Management: Object Oriented and Extended Relational Systems*; Addison-Wesley, 1991.

[ChamGT75] D.D. Chamberlin, J.N. Gray, and I.L. Traiger,I.L.: Views, Authorization, and Locking in a Relational Data Base System, *Proc. of the National Computer Conference 1975*, Vol.44, AFIPS Press, pages 425-430.

[DeliR92] A. Delis and N. Roussopoulos: Performance and Scalability of Client-Server Database Architectures; *18th Int. Conf. on Very Large Data Bases*, Vancouver, British Columbia, Canada, 1992, 610-623.

[DeWi*90] D. J. DeWitt, D. Maier, P. Futtersack, and F. Velez: A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems; *16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990.

[FranCL93] M.J. Franklin, M.J. Carey, and M. Livny: Local Disk Caching for Client-Server Database Systems*; 19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993, 641-654.

[GuptMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian: Maintaining Views Incrementally; *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., May 1993, 157-166.

[GrayR93] J. Gray and A. Reuter: *Transaction Processing: Concepts and Techniques;* Morgan Kaufmann Publishers, 1993.

[Hamm*97] J. Hammer, M. Breunig, H. Garcia-Molina, S.Nestorov, V. Vassalos, R. Yerneni: Template-Based Wrappers in the TSIMMIS System; in Proceedings of the 26th SIGMOD International Conference on Management of Data, Tucson, Arizona, May 1997.

[HammMG97] J. Hammer , J. McHugh , H. Garcia-Molina: Semistructured Data: The TSIMMIS Experience; in *Proceedings of the First East-European Workshop on Advances in Databases and Information Systems*-ADBIS '97, St. Petersburg, Russia, September 1997.

[HoubG97] Geert-Jan Houben, Frank Dignum: Integrating Information for Organized Work; *Proceedings of the 4th Workshop KRDB-97*, Athens, Greece, August 1997, F. Baader, MA. Jeusfeld, and W. Nutt (eds.), RWTH Aachen, Aachen, Germany.

[KatoM92] K. Kato and T. Masuda: Persistent Caching: An Implementation Technique for Complex Objects with Object Identity; *IEEE Transactions on Software Engineering*, July 1992.

[Kell86] 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, Morgan Kaufman, pubs.

[KellJA93] A.M. Keller, R. Jensen, S. Agarwal: Persistence Software: Bridging Object-Oriented Programming and Relational Databases; *ACM SIGMOD, International Conference on Management of Data*, May 1993.

[KellH93] A.M. Keller and C. Hamon: A C++ Binding for Penguin: a System for Data Sharing among Heterogeneous Object Models; *4th Int. Conf. Foundations of Data Organization and Algorithms*, Evanston, October 1993.

[KellB 94] A.M. Keller and J. Basu: A Predicate-based Caching Scheme for Client-Server Database Architectures; The VLDB Journal, Vol.5 No.1, Jan 1996, pp.35-47

[LWBSSZ90] 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.

[LeeW94] B.S. Lee and G. Wiederhold: Efficiently Instantiating View-objects from Remote Relational Databases; *The VLDB Journal*, Vol.3 No.3, July 1994, pages 289-323.

[NavaEL86] S.B. Navathe, R.El Masri, and J.A. Larson: Integrating User Views in Database Design; IEEE Computer, Jan. 1986, Vol. 19 No.1, pages 50-62.

[QuaW97] D. Quass , J. Widom: On-Line Warehouse View Maintenance for Batch Updates; Proc. ACM SIGMOD '97, June 1997.

[SaltR:97] Felix Saltor, Elena Rodriguez: On Intelligent Access to Heterogeneous Information; *Proceedings of the 4th Workshop KRDB-97*, Athens, Greece, August 1997, F. Baader, MA. Jeusfeld, and W. Nutt (eds.), RWTH Aachen, Aachen, Germany. F

[ShilS89] J.J. Shilling and P.F. Sweeney: Three Steps to Views: Extending the Object-Oriented Paradigm; OOPSLA 89, pages 353-361.

[Steel80] T.B. Steel jr.: Status Report on ISO/TC97/SC5/WG3--Data Base Management Systems; Proceedings of VLDB 6, Oct. 1980, Morgan Kaufman pubs., pages 321-325.

[Stro86] B. Stroustrup: *The C++ Programming Language;* Addison-Wesley, 1986.

[TsicB89] D.C. Tsichritzis and T. Bogh: Fitting Round Objects into Square Databases; *OOPSLA 1989*, New Orleans.

[Ullm88] J. D. Ullman: *Principles of Database and Knowledge-base Systems, Volume 1: Classical Database Systems;* Computer Science Press, 1988.

[WangR91] Y. Wang and L.A. Rowe: Cache Consistency and Concurrency Control in a Client-Server DBMS Architecture.; *ACM SIGMOD Int. Conf. on Management of Data*, Denver, CO, May 1991, 367-376.

[WhitD92] S.J. White and D.J. DeWitt: A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies; Proc. VLDB 18, Vancouver , Morgan Kaufman pubs., Aug. 1992.

[WiedE80] G. Wiederhold and R. ElMasri: The Structural Model for Database Design; *Entity-Relationship Approach to System Analysis and Design*, North-Holland, 1980.

[Wied86] G. Wiederhold: Views, Objects and Databases; *IEEE Computer*, Vol.19 No.2, 1986.

[Wied87] G. Wiederhold: *File Organization for Database Design*; McGraw-Hill, 1987.

-- o --