Two-Level Caching of Composite Object Views of Relational Databases*

Catherine Hamon Arthur M. Keller

Abstract

We describe a two-level client-side cache for composite objects mapped as views of a relational database. A semantic model, the Structural Model, is used to specify joins on the relational database that are useful for defining composite objects. The lower level of the cache contains the tuples from each relation that have already been loaded into memory. 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 level of the 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. This two-level cache is designed as part of the Penguin system, which supports multiple applications, each with its own object schema, to share data stored in a common relational database.

1 Introduction

The client-server architecture is becoming commonplace in modern computing environments that use high-speed computer network technology with workstations. In such environments, caching data on the client workstations minimizes the cost of accessing the remote database server. Client-side caching has been extensively studied for object-oriented DBMSs (OODBMS). Caching schemas use the notion of object identifier (*oid*) to store, retrieve and maintain cached objects. An object fault on the workstation causes the transfer of the requested object from the server to the client. With a page-server architecture, the objects residing on the same page are also transferred. Different client-server architectures can be defined based on how database functionalities are partitioned between the client and the server. Three client-server architectures for OODBMSs are described in [6].

*For information about the Penguin project, please write to Arthur M. Keller, Stanford University, Computer Science Dept., Stanford, CA 94305-2140, or to ark@cs.stanford.edu

In contrast to OODBMSs, the interaction between the client and the server in a relational DBMS is based on SQL queries. The server is responsible for processing SQL blocks and returning the result to the client. Tuples can be cached in the client main memory for providing a local response to the same query frequently given by a user or to another query. Two important issues are how to cache the query results and how to access the cached tuples for a local query processing. These issues are related to the issues of view and query optimization. Cached data can be seen as materialized views on which index structures can be built to retrieve the tuples of the base relations. The concept of view has been widely used for many purposes and techniques developed in view maintenance [7, 8, 18, 20] are of interest for several areas such as integrity constraint maintenance, query processing, active database, and cache refresh operations.

In this paper we consider two problems. The first one is how to efficiently cache nested tuples defined as the result of materialized views over a relational database, so that the application program or the end-user can scan the result of a view (i.e., instances of an object class) and navigate from one view to another. For example, Figure 1 shows two collections of nested tuples for the views *CourObj* and *DeptObj* defined over a relational database containing data for a university. The end-user can scan a collection of nested tuples, or access to an instance of *DeptObj* from a tuple of the department in *CourObj*.

The second problem is how to reuse the cached results of a collection of views to locally process another query. If a same query is frequently given by a user, caching the instances of a view for extended periods of times obviously avoids the extra cost of re-fetching data and nesting tuples. Let us consider now the reuse of tuples across several views. As illustrated in Figure 1, the instances of *CourObj* consist of nested tuples of the relations *Course*, *Student* and *Department*. The instances of *StudObj* consist of nested tuples of the relations *Student*, *Course* and *Grade*. These two collections of nested tuples are composed of overlapping pieces of data. A set of tuples fetched to materialize a

CourObj

number	title	Student		Department	
		ssn	major	name	chair
CS356	CLA	56	CS	CS	Klein
		57	CS		
Math2	CLB	57	CS	Math	Reno
		90	Math		
		98	Math		
CS357	CLC	57	CS	CS	Klein

DeptObj

name	chair	school	People
			name
CS	Klein	Eng.	Smith
			Harris
Math	Reno	Sci.	Harris
			Brown
			Kelly

StudObj

ssn	Course	Grade
	title	grade
56	CS356	A-
57	CS356	A-
	Math2	A-
	CS357	B
98	Math2	A-
90	Math2	B+

Figure 1: Examples of View-Objects.

view can thus be reused to wholly or partially answer another query.

We propose a two-level client caching architecture to handle these two issues for applications with heterogeneous schemas (C++, Smalltalk) sharing data stored in a relational database. The upper level of the cache uses the view-object model. This model has been proposed in [1] and extended in [16] to define on top of the relational database an object layer independent of any programming language. Based on this model, an application view can be built consisting of view-objects (i.e., an object class) organized in PART-OF and IS-A graphs. Several important features are provided through this model. First, the view-object concept combines both the notion of object class in programming languages and the database notion of view (see [25]). The extension of a view-object is a collection of nested tuples called composite objects (see Figure 1). Complex units of information can thus be built and shared across different application views. Second, an approach has been developed in [4] to preserve the information sharing feature when mapping view-objects into object classes in the programming language (see [16] for the description of the C++ linking). Third, the issue of updating data from object-oriented programming languages can be handled through the problem of updating view-objects. We use the approach described in [10, 11, 12, 14] that extends previous works on relational views. Fourth, an Object Query Language with an SQL-like syntax has been proposed in [21, 22] for querying composite object views from the application layer (e.g., a C++ application). In this paper, we show how we can benefit from the view-object model to cache nested tuples independently of any programming languages. Each application has its own view-object schema and can access data stored in a view-object cache (see Figure 2). These data are organized according to the view-object schema. The view-object cache is within each application's address space. It provides efficient browsing and navigation capabilities among the view-object instances.

Our second aim is to augment the local reuse of the cached tuples. With this in mind, we propose to add another layer in the cache that is shared by the client applications. The idea is to store at this level the tuples fetched through queries or tuple faults initiated from any of these applications. The cached tuples are this way made available for locally instantiating a view-object whatever the application it belongs to. The final result is then exported to the upper level and linked to the corresponding view-object. In this paper, we show that the Structural Model introduced in [24] allows us to handle this issue effectively. The structural model of a relational database consists of normalized relations and connections (relationships) among those relations (see Figure 2). Based on this model, tuples are cached in the form of a network whose links represent join expressions. The instantiation process of a view-object in this cache acts as a filter on the tuples and the links; the resulting instances represent actually a view over the cache network. Remote access is minimized because of the network. Figure 2 shows an overview of the two-level cache. The application layer contains the applications with heterogeneous schemas.

The execution of queries in the cache requires considerations of the issues of cache completeness and currency. Cache completeness addresses the question of whether the desired answer to a query can be obtained from data present in the cache, without the need for a network fetch from the database. Cache currency is concerned with the effect of database updates on locally cached data, that is, whether an update at the server needs to be propagated to a client cache to ensure cache consistency. The cache consistency issue in client-server DBMS architectures is examined in [6, 29, 5, 23]. In [17], we propose predicate-based descriptions of client caches to handle the two issues mentioned above for cached objects not identified by logical or physical identifiers. Predicates are obtained from WHERE clauses of SELECT-FROM-WHERE statements.

This paper is organized as follows. In Sections 2 and 3, we present respectively the Structural Model and the view-object model. The lower and the upper levels of the client cache are described in Sections 4

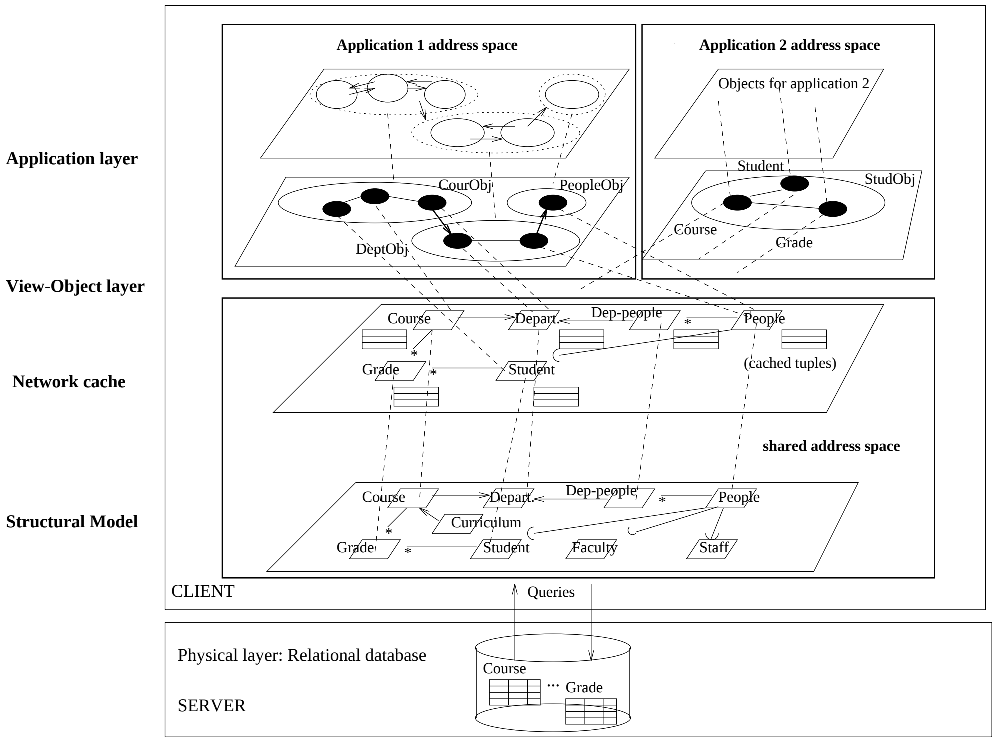

Figure 2: Two-level cache for heterogeneous applications (see Section 2 for the symbols used in this figure).

and 5 respectively. In Section 6, we show the advantages of using separated address spaces for each application and another address space shared by the client applications. We then present our conclusions.

The Structural Model, the view-object model and the C++ binding have been implemented in the Penguin system (see [1, 2, 4, 16, 19, 21, 22, 27, 28]).

2 Structural Model

We use a directed-graph representation of the structural model of a relational database, where nodes correspond to normalized relations and arcs to connections (see Figure 3(A)). Three types of connections are defined in the Structural Model. The *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 of an employee is owned by the record of that employee. The *reference connection* ($\rightarrow$) 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. The *subset connection* ($-\!\!\supset$) 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 parent to the child. The semantics in the structural model is used to avoid problems of updating databases through object views. Syntactic and integrity rules for each of these connections are detailed in [1, 3].

3 View-Object Model

A view-object represents a portion of the structural model hierarchically. The hierarchical structure is defined by a *template tree* whose nodes represent relations from the structural model, on which projections can be defined. The relations are composed through relational joins. Figure 3(C) shows an example of a template tree whose *pivot relation* is *Course* and that contains the nested relations *Department*, *Grade*, and *Student*. Figure 3(B) shows the *candidate tree* from which the designer builds the template tree. The template tree is rooted on the same relation *Course*, its nodes are any subset of the candidate tree's nodes, and its arcs are paths of length 1 or greater from the pivot relation to those nodes in the original candidate tree. An algorithm has been developed in [1] for the computation of a unique candidate tree rooted on a given relation.

An instance of a view-object is a composite object

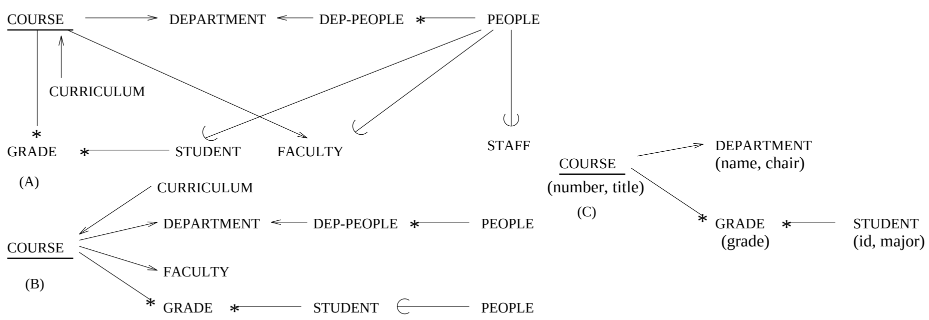

Figure 3: Examples of structural model, candidate tree and template tree.

generated from the outer join of the tuples of the corresponding relations. These tuples are nested according to the template tree. The primary key of the pivot relation permits any given instance of the view-object to be uniquely identified. The necessary joins and projections define the *data-access function* (DAF) for the view-object; they are only dependent of the object structure. It is possible that atomic attributes are either missing (e.g., the key attributes for the pivot relation) or redundant, or that relations not chosen by the designer are necessary to compute join expressions. Thus, a recomputation of the DAF information can be necessary. Using the DAF, a view-object has information for creating a SELECT-FROM-WHERE block of an SQL query at instantiation time.

A view-object is also defined by a set of references to other view-objects whose primary key is included in the source view-object. References enable the user to define PART-OF graphs of view-objects convenient for one application. Figure 2 shows a PART-OF graph for application 1 where *CourObj*, *DeptObj*, and *PeopleObj* correspond to the nodes and the arrows represent the accesses allowed among the view-objects. The user can navigate to *DeptObj* from a tuple of the department in *CourObj*. Indeed, Figure 1 shows that *CourObj* contains subtuples of the department, and includes the primary key of *DeptObj*.

4 Network Cache

A global schema based on the Structural Model defines a unique logical organization for data shared by different applications views. The lower level of the cache uses the structural model. Cached tuples are stored by relation, with arcs connecting the tuples across relations based on the connections in which their relations participate. The possible connections are ownership, reference, and subset, all of which are variations of foreign key to primary key joins. In this section, we show how the network model brings effective solutions for efficiently caching data shared by client applications and locally instantiating view-objects.

4.1 Network of Cached Tuples

Conceptually, the nodes of the cache network represent cached relations and arcs stand for connections between relations. Links among tuples of cached relations can be seen as particular instances of connections defined in the structural model. Contrary to approaches that use the notion of *oid* to construct graph-structured objects (see [9]), we use for this purpose the concepts of primary and foreign keys that allow the client to capture and manage bidirectional links among tuples stored in the cache. These links are based on the commonality of values of two subsets of attributes X_1 of R_1 and X_2 of R_2, for the relations R_1 and R_2 being connected. X_1 and X_2 have identical number of attributes and domains. Since we consider three types of connections, we have the following situations:

1. Ownership connection; R_1 owns R_2. X_1 is the key of R_1 and X_2 is included in the key of R_2. More precisely, X_2 represents a foreign key whose values match X_1 values. To one value of X_1 may thus correspond several tuples of R_2.

2. Reference connection; R_1 refers to R_2. X_2 is the primary key of R_2 and X_1 is included in the key or the non-key attributes of R_1. In this case, X_1 represents a foreign key whose values match X_2 values. Consequently, to one value of X_2 may correspond several tuples of R_1.

3. Subset connection; $R2$ is a subset of $R1$. X_1 and X_2 are respectively the key of R_1 and R_2, and to one tuple of R_2 corresponds one tuple of R_1.

The tuples of a relation can also be linked according to the value of the foreign key(s) they contain. These

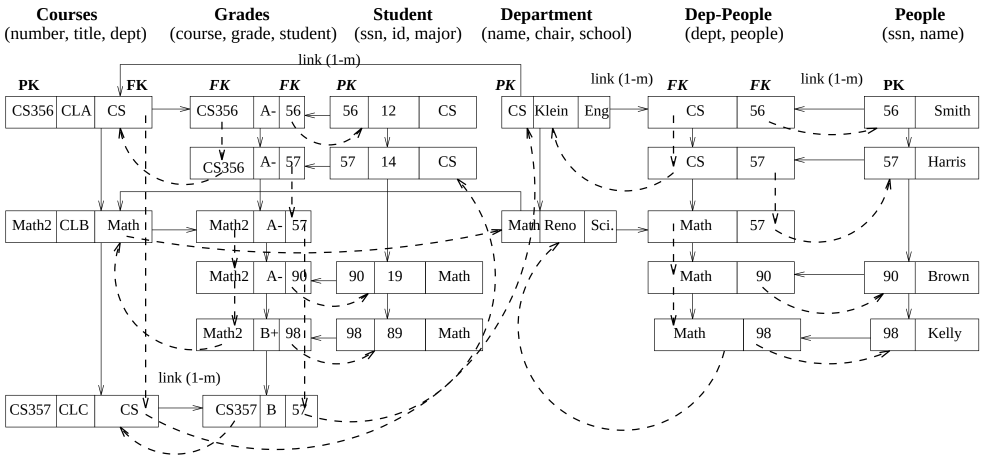

Figure 4: Rings-based network for the cache.

links are called *foreign key* links. Let us consider tuples of the relation *Course* as shown in Figure 4. Since these tuples contain two different values for the foreign key *dept*, that are *CS* and *Math*, they can be separated into two tuple lists based on these values.

A one-to-many connection (ownership or inverse reference connections) from R_1 to R_2 represents a set of potential parent-child links between tuples of R_1 and R_2. If we only represent parent-to-first-child link and last-child-to-parent link, then we can use foreign key links based on the values of X_1 to retrieve the correct list of R_2 tuples whose X_2 values match X_1 values. We can thus use X_1 values to form rings (see Chapters 6 and 7 of [26]) connecting each tuple of R_1 to the tuples of R_2 that it owns (ownership connection) or, that reference it (inverse reference connection). Since a same relation R can be at the origin of many many-to-one connections (inverse ownership and reference connections), it can contain several foreign keys referring to the key attributes of the connected relations. Consequently, tuples of a relation can be involved in many rings depending on a same foreign key or on different foreign keys. For example, *Grade*-tuples are part of three rings based on the values *CS356, Math2 and CS357* of the foreign key *course*. These tuples are also involved in four other rings based on the values *56, 57, 90, 98* of the foreign key *student*.

We use a low-level cache structure based on a ring network that captures the semantics of the structural model (see Figure 4). The cached tuples contain all the attribute values as defined in the relations: The primary and foreign keys of the cached tuples are needed to build tuple rings and the remaining attributes are potentially useful for any local view-object instantiation.

4.2 Physical Organization of the Cached Tuples

We have chosen to use hash tables and hard pointers to implement the ring network cache (see Figure 5). Tuples belonging to the same relation are stored in a table hashed on the values of the primary key. The use of these hash tables permits a direct access to cached tuples according to their primary key values, but it does not allow range queries. Our approach will work well with additional indexing structures, such as various kinds of trees, that support range queries or other access methods.

One of our main objectives is to efficiently implement one-to-many links among cached tuples to rapidly retrieve the list of tuples of a relation R_2 according to the key value of a relation R_1, R_1 and R_2 being connected by a one-to-many connection. One way to achieve this goal is to explicitly build for each primary key value of R_1 the corresponding list of R_2-tuples. Note that these lists do not contain real tuples but rather pointers to the real tuples accessible from the hash table where they are stored. For example, Figure 5 illustrates how tuples of *Course* are stored in the hash table of this relation and referenced from the hash table of *Department*, *Department* and *Course* being connected by an inverse reference connection (one-to-many cardinality). Following successive one-to-many links amounts to following hard pointers among tuples. For example, Figure 5 shows that from a particular tuple of *Department* we can access real tuples of *Course* that in turn give access to real tuples of *Grade*.

As for many-to-one links (multiple children for one parent), we have chosen not to implement them by us-

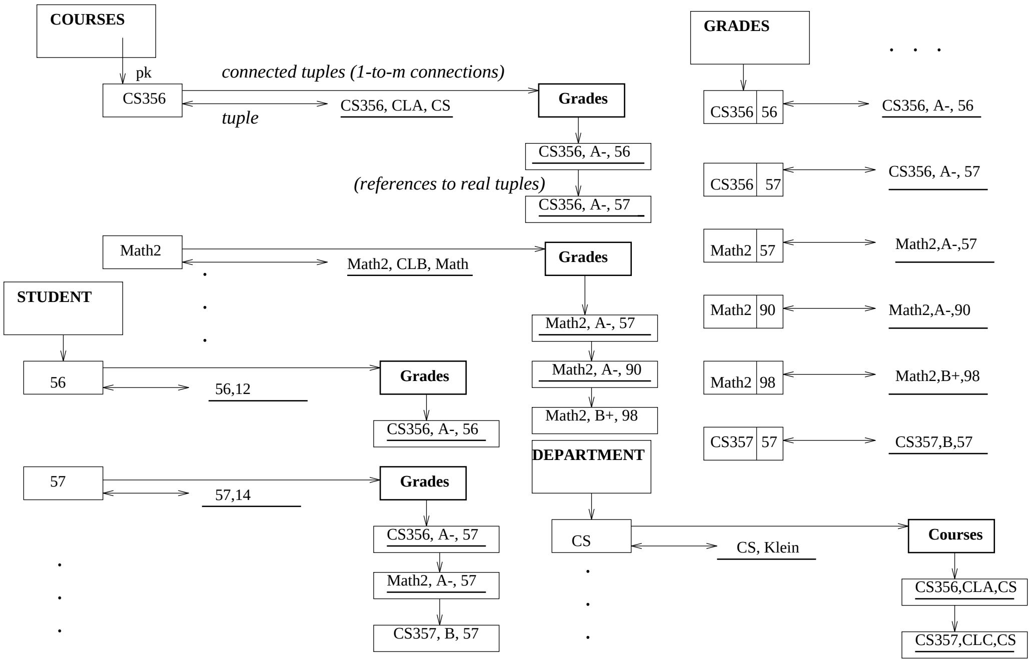

Figure 5: Network cache implementation.

ing hard pointers. We simply extract the foreign key value of the child tuple to retrieve the primary key value of the parent tuple and consequently the corresponding hash table entry. Note that from one child tuple we can at most get one tuple of the parent relation; more exactly one tuple in the case of an inverse ownership connection; zero or one tuple in the case of a reference connection.

4.3 Maintenance of the tuple network

Relations in a template tree can be connected by arcs that represent paths of length greater than 1 in the structural model. To maintain a tuple network that matches the structural model, the client may need to fetch additional attribute values upon instantiation of a view-object, that is, data not specified in the DAF information of this view-object (see Section 3). The SELECT-clause of the query must indeed contain those attributes that permit links to be established among the cached tuples according to joins based on connections. The examples of the Figures 6(B, C) show that the attribute values of the missing relation B are needed to establish connections among A-tuples and C-tuples since A and C do not have common attributes due to the type of the connections between the relations A, B and C. Note that all the attributes of the tuples to select will be imported in the network cache.

The missing relations of a template tree are nodes of the corresponding candidate tree that do not appear between the pivot relation and the leaves of the template tree. These relations and those contained in the template tree define a subtree of the candidate tree called *complete template tree* (see Figure 6). The execution of a SELECT query on the database yields in the network cache a collection of tuples to be organized according to the hierarchy of the complete template, redundant tuples being eliminated. The complete template tree provides information about the type of the connections existing between the relations involved in the query. Using this information, the client creates or updates the appropriate tuple lists (one-to-many links) in the cache. New entries may have to be created to lead to tuple lists. For example, consider the instantiation of the view-object of Figure 6(C). Suppose that the corresponding query result is not contained in the network cache. The resulting tuples are instances of relations A, B, C and E. We need to fetch B-tuples from the database although B is not contained in the template tree. Indeed, B-tuples ensure the creation of navigational paths between A-tuples and C-tuples. Furthermore, since we want these new tuples to be organized according to the complete template that matches a portion of the structural model, we need also to create new entries for the hash table of the relation D included in the ownership path that goes from A to E. Entries of this hash table

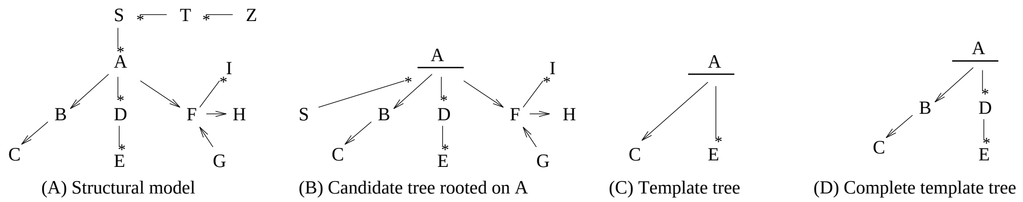

Figure 6: Missing nodes in a template tree.

will be computed from the resulting E-tuples that contain the primary key values of D-tuples. These entries will only lead to E-tuple lists.

When storing new tuples in the cache, we have to be careful that all required links are created to support future instantiations of view-objects and navigation among tuples. Consider the previous example and Figure 6(A). The client may have to create new entries for the relation F and build or update A-tuple lists. In effect, an A-tuple contains the primary key value of an F-tuple; we can thus compute the corresponding entries in the hash table of the relation F and associate them with the appropriate A-tuples. The same goes for the relation S connected to the relation A by an ownership connection. Note that this process does not go beyond the relations reachable from F due to the type of the connection between the relations A and F. However, it is possible to go beyond the relation S and associate A-tuples to both Z-tuples and T-tuples since the primary key of the relation A contains the primary key of relations S, Z and T. But even in that case, we do not go beyond the relation S. Indeed, we do not allow the access of A-tuples from relations Z and T that are not directly connected to A (the network cache matches this way the structural model). Subsequent queries can later return in the cache S-tuples, D-tuples and F-tuples whose entries exist in the cache, and only lead to tuple lists.

We have discussed the advantages of maintaining a network cache matching the structural model. This model avoids the creation and the maintenance of links representing paths of length greater than 1 in the structural model. The cost incurred by their creation can be very high as illustrated in the previous example. Our approach permits tuples fetched from queries to be stored in the form of a network so that data is organized conveniently for view-object instantiation in the cache. This avoids the costly operation of restructuring the data into the appropriate hierarchical form. Furthermore, when the network cache partially answers a query, we ensure that the appropriate links will be created between the cached data and the new data (the missing data).

5 View-Object Cache

In this section, we show how the network cache is used for view-object instantiation to fill the view-object cache. *Browsing* links are created among cached copies of (sub)tuples, and the final data structure is bound to the appropriate view-object. These copies guarantee that the updates of application instances are visible only in the application views where they occur, until commit time.

5.1 View-Objects Instantiation in the Cache

The view-object instantiation starts when all the required information is stored in the cache, except possibly when instantiation is a result of object faulting through navigation among view-objects (see Section 5.2). Performing the instantiation process involves navigating the network cache and visiting the relevant nodes in an order that matches a depth-first traversal of the complete tree. First, the tuples of the view-object's pivot relation are selected. Then, the access to the remaining tuples can be done through one-to-many links (hard pointers that lead to tuple lists), or many-to-one links. Note that in the case of many-to-one links, foreign key values included in the current tuples serve to compute the entries of the connected tuples. Otherwise, a one-to-many link leads to a tuple list and, depending on the predicates of the query, a more refined selection can be done. During this process, copies of the (sub)tuples of the template's relations are created and organized by the use of *child* links (first child links) and/or *sibling* links (see Figure 7). A child link (or a sibling link) between two tuples may correspond to a path of length 1 (any of the three connections of the structural model or their inverse) or greater than 1. Access from children to parent tuples is ensured by *parent* links. In case of ownership or inverse reference connections (one-to-many cardinality), the parent tuple can own multiple child tuples, or can be referenced by more than one referencing child tuple. A *next* link is used to group tuples of a same relation (template's node) having the same

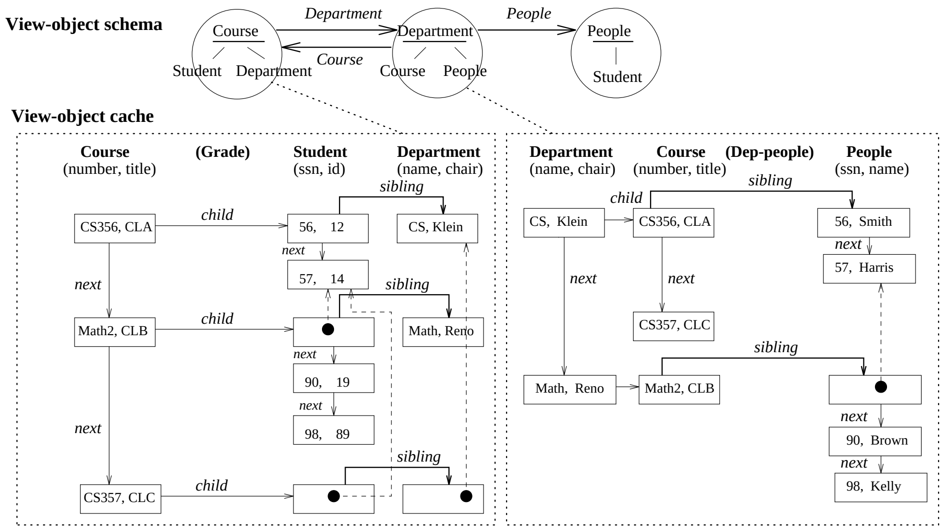

Figure 7: View-object cache implementation.

parent tuple. A tuple can therefore lead to different sets of tuples through child and sibling links.

5.2 Browsing and Navigation

At the application layer, each client application accesses the data stored in its own view-object cache. Objects contain pointers to the cached data. The choice of the pointers at this layer avoids the problem of propagating updates to the view-object cache and vice-versa. The links among the (sub)tuples of the template's nodes allow an efficient browsing operation over the instances of a view-object. Navigation among view-objects is based on PART-OF graphs (see Section 3). Navigation among view-objects is similar to navigation in the structural model. We have chosen to dynamically implement the navigational capability. Remember that the origin of a reference link is a view-object ω_1 whose template tree contains a relation R that serves as a pivot relation for the referenced view-object ω_2. As a result, for a particular node's instance value in the referencing view-object, we can retrieve the corresponding instances attached to the referenced view-object and present them to the user according to their nested pattern. The user may wish to select one or several instances of ω_1 to retrieve the corresponding instances of ω_2. R-tuples selected in ω_1 are the starting points of a navigation in the network cache, the aim being to reach the remaining tuples so as to build the proper instances of ω_2. To do that, a first solution is to ensure that the cache contains the required information, eventually fetch the missing information through a query, and then perform the instantiation process in the cache. Another solution is to follow links in the network cache until a tuple fault (i.e., a cache miss) occurs. In this case, the cache description as introduced in [17] will permit the client to know which part of the data has to be fetched from the database. We have chosen this last solution because it avoids the cost of a useless scan of the cache description.

6 Architectural Issues of the Two-Level Caching

We propose an architecture where each application has its own private address space including its own view-object cache. This cache contains copies of data stored in the network cache, which is shared by all applications running on the same workstation. Browsing and navigation are performed directly in the application's address space where the information is organized according to the data structures defined by the object schema of the application. Modification operations remain visible only in view-object cache until commit time. At commit time, the network cache is updated and other applications sharing the data need to be notified. The update of a client network cache may make the associated view-object caches inconsistent. Consistency of the levels of the two-level cache and of the various clients and server are beyond the scope of this paper. However, we have developed algorithms for this problem in our ongoing work.

The network cache is part of a separated address space that contains data fetched through queries initiated by the various applications. These data are organized into a network matching the structural model. Navigation based on this network is performed upon view-object instantiation. Since the client applications may request overlapping sets of data, we can expect to augment the potential re-use of data stored in the network cache and minimize remote access. Copies of some of these data are transferred to an application's address space upon view-object instantiation, or when update propagations are performed to ensure the view-object cache consistency. But transfer of data from a view-object cache to the network cache is only done at transaction commit. Therefore, the network cache cannot be corrupted by programming errors in applications. The application's address space and the address space shared by all client applications are illustrated in Figure 2.

7 Conclusions

In this paper, we have proposed a two-level client-side cache for applications with heterogeneous schemas sharing data stored in a relational database. Effective browsing and navigation operations over nested tuples are possible in the address spaces defined by the upper level of the cache that are application dependent. Indeed, in each of these address spaces, data are organized differently according to the access needs of the application specified in the view-object schema. Tuples fetched from queries initiated from any client applications are stored in the lower level of the cache and organized into a minimal network based on the structural model (i.e. only links representing paths of length 1 in the structural model are created). With this schema, remote access can be minimized since cached data can be reused across several client applications in the address space defined by the network cache. Also, the cost of restructuring tuples into a view-object template is minimal since composite objects represent hierarchical views of the tuple network. Moreover, we have shown that the navigation in the network cache can be done without checking first that the full result is cached. In the worst case, a cache miss occurs and requires access to the missing information on the underlying database. Future work will aim at proving the feasibility of this approach through an experimental study of the two-level caching schema using the client cache description introduced in [17].

8 Acknowledgements

This effort was supported in part by the Microelectronics Manufacturing Science and Technology project as a subcontract to Texas Instruments on ARPA contract number F33615-88-C-5448 task number 9, and the Center for Integrated Systems. The work of Catherine Hamon was supported by a postdoctoral fellowship supported by the French government. Many people participated over the years in the design and implementation of the Penguin system. We appreciate the participation or feedback from Julie Basu, Kai Huang, Larry Safran, Tetsuya Takahashi, and Srinivasan Venkatesan in the development of these ideas. Marianne Siroker helped with the preparation of this paper.

References

[1] T. Barsalou. *View-Objects for Relational Databases.* Ph.D. dissertation, Stanford University, March 1990, Technical Report STAN-CS-90-1310.

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

[3] T. Barsalou, N. Siambela, A. M. Keller, G. Wiederhold. Updating Relational Databases through Object-Based Views. *ACM SIGMOD Int. Conf. on Management of Data, Denver, CO, May 1991.*

[4] A. Carlson. *Penguin System Internal Maintenance Specifications.* Unpublished document, October 1992.

[5] M.J. Carey, M.J. Franklin, M. Livny and E.J. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. *ACM SIGMOD Int. Conf. on Management of Data*, Denver, CO, May 1991.

[6] 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.

[7] A. Gupta, I.S. Mumick, V.S. Subrahmanian. Maintaining Views Incrementally. *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., May 1993.

[8] C.S. Jensen, L. Mark, N. Roussopoulos, T. Sellis. *Using Caching, Cache Indexing, and Differential Techniques to Efficiently Support Transac-*

tion Time. CS-TR-2413, University of Maryland, February 1990.

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

[10] A. M. Keller. *Updating Relational Databases Through Views.* Ph.D. dissertation, Stanford University, February 1985, Technical Report STAN-CS-85-1040.

[11] A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Computer*, 19(1), January 1986.

[12] 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.

[13] 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.

[14] A. M. Keller and L. Harvey. *A Prototype View Update Translation Facility.* Report TR-87-45, Dept. of Computer Sciences, University of Texas at Austin, December 1987.

[15] A.M. Keller, R. Jensen, S. Agarwal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. *ACM SIGMOD, Int. Conf. on Management of Data*, Washington, D.C., May 1993.

[16] 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.

[17] A.M. Keller and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *Parallel and Distributed Information Systems*, Austin, September 1994.

[18] P.A. Larson, H.Z. Yang. *Computing Queries from Derived Relations: Theoretical Foundation.* Research Report CS-87-35, University of Waterloo, August 1987.

[19] K. H. Law, G. Wiederhold, T. Barsalou, N. Siambela, W. Sujansky, and D. Zingmond. Managing Design Objects in a Sharable Relational Framework. *ASME meeting*, Boston, August 1990.

[20] N. Roussopoulos. *The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis.* CS-TR-2193, University of Maryland, February 1989.

[21] T. Takahashi and A. M. Keller. Querying Heterogeneous Object Views of a Relational Database. *Int. Symp. on Next Generation Database Systems and their Applications (NDA) 93*, Fukuoka, Japan, September 1993.

[22] T. Takahashi and A. M. Keller. Implementation of Object View Query on a Relational Database. *Data and Knowledge Systems for Manufacturing and Engineering*, Hong Kong, May 1994.

[23] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. *ACM SIGMOD Int. Conf. on Management of Data*, Denver, CO, May 1991.

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

[25] G. Wiederhold. Views, Objects and Databases. *IEEE Computer*, 19(12), 1986.

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

[27] G. Wiederhold, T. Barsalou, and S. Chaudhuri. *Managing Objects in a Relational Framework.* Stanford Technical report CS-89-1245, January 1989, Stanford University.

[28] 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.

[29] K. Wilkinson and Marie-Anne Neimat. Maintaining Consistency of Client Cached Data. *16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990.