Implementation of Object View Query on a Relational Database[1]

Tetsuya Takahashi
Kobe Steel, Ltd.

Arthur M. Keller
Stanford University

Abstract

We present the implementation of the query function for the Penguin system. Penguin is an object-oriented database system that supports multiple object views on a relational database. It enables many applications to share a database using different object schemata. Also, users can take queries for the Penguin database in their applications, to retrieve objects on the heterogeneous data model. The query function is very powerful for manufacturing and engineering purposes, that is, connections between relations and methods defined by users in applications are available in query requests. We introduce the object model and the query execution algorithm of the Penguin system with an example for a steel plant, and show the system configuration for the implementation of the query function.

1 Introduction

In the field of manufacturing and engineering, an important requirement for database systems is the handling of complicated structures of data, such as manufacturing design data and production data.

In these applications, the data are closely related to each other, and we have to manage such relationships. It is well known fact that the relational data model is not necessarily suitable to represent such a complex schema. New types of database systems, such as object-oriented databases, are being proposed to support these applications.

On the other hand, the relational model is so widely accepted that most new database applications are based on the relational model today, because of its simple description and mathematically elegant theory. As the result, it is now almost impossible to replace all existing database applications with object-oriented ones. Therefore, we need some other approach to fill the gap between the object-oriented models and the relational models, and to migrate from present databases to next generation systems.

Many extended relational models have been studied to break the limitation of the relational model [8][9]. These models allow nested relations and can treat more complex data schema than the conventional relations. But they still have less natural representations of the real world, compared with the object-oriented model.

Some object-oriented database management systems are available today [4], and there are many attempts to utilize them in manufacturing and engineering applications. Current object-oriented DBMSes still have a lot of problems. One issue is that they do not support the view concept [11]. That is, they permit us to define only one conceptual schema for each database, and all the applications which access the same database are forced to use the same structure of object classes. Another problem is that object-oriented database often provides only limited query capability (as compared with SQL). These are strong impediments to the development of applications.

The Penguin system, developed at Stanford University, solves those problems. The Penguin system has been developed to allow the sharing data in a common relational database among multiple applications that each have their own object schemata [1] [2] [3] [5] [6] [12]. Penguin also has a powerful object query function available for applications [10]. We introduce the data models and the query algorithm of Penguin in the next section, and then describe how the query function is implemented for the Penguin system in Section 3.

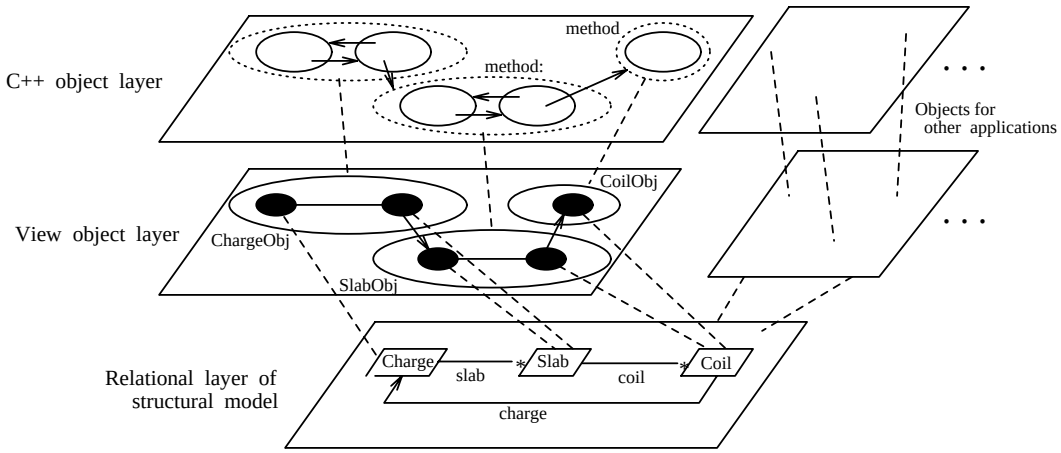

Figure 1 : Data mapping in multi-layered model

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

2 Penguin system

2.1 Data model

Penguin has a multi-layered conceptual schema. The layers are the relational layer, the view-object layer, and the C++ object layer. In Fig. 1, we illustrate this layering using an example of a database containing data on charges, slabs and coils that are intermediate products in a steel plant. A charge is a unit of molten steel produced in the steel-making process; it is cast into slabs by a continuous caster. Then coils are produced from slabs in the rolling process.

A structural model is defined on top of the relational database layer to classify the relationships between the relations. Connections are defined between the relations to be joins that also provide structural semantics for the relational model. Connections are represented in a directed graph representing the structural model. Penguin currently supports three types of connection: *Ownership Connections* (represented by $—*$), *Reference Connections* (represented by $\longrightarrow$) and *Subset Connections* (represented by $—\supset$).

In the view-object layer, composite objects are defined on the structural model. For example, the objects defined in Fig. 1 correspond to the following view-objects.

ChargeObj

charge_id	carbon	sulpher	slab
			slab_id
CH131	0.040	0.015	SL321
			SL322
CH132	0.030	0.015	SL345
			SL346
			SL347
⋮	⋮	⋮	⋮

SlabObj

slab_id	length	coil
		coil_id
SL321	950.0	CO122
SL322	955.0	CO123
SL345	935.0	CO511
SL346	930.0	CO532
		CO533
SL347	920.0	CO814
⋮	⋮	⋮

CoilObj

coil_id	thickness	width	charge_id
CO111	45.0	950.0	CH541
CO194	30.0	800.0	CH417
CO123	40.5	1010.0	CH131
CO222	25.0	850.0	CH354
⋮	⋮	⋮	⋮

The 'ChargeObj' consists of the attributes of molten steel, such as the contents of carbon and sulphur, and also contains the keys of the slabs produced from the charge. In the same way, the 'SlabObj' and the 'CoilObj' are nested tuples which contain the attributes of slabs and coils produced in the steel plant.

Each application can have its own object schema, and can share the data with other applications. That is, a user can decide a desirable object schema for the respective application, and he can access the data in the relational database via the window of view-objects.

Penguin also automatically generates the C++ classes corresponding to the view-objects, which consist of the attributes in the view objects and the basic methods to access the view-objects (e.g., navigation, query, and update). Users can add their own methods in this C++ object layer, and can construct applications based on those methods defined in the application layer.

2.2 Query language

In applications of manufacturing and engineering, flexible and intelligent query functions are often required. For instance, a user in the manufacturing field may want to retrieve the data of the products that have any quality problems. But the method of evaluating quality depends on the particular manufacturing process, and often has to be modified due to the change of the customers' needs. To satisfy such various and variable requests, the database system has to support queries that contain methods that may be defined by users, such as functions to estimate the quality of the products.

As described before, Penguin supports the multi-layered data schema of view-objects on the relational database. And users can make their own methods in the C++ object layer. We have defined a query language which enables the users to use the C++ methods in the query statements which are issued in a C++ application linked with the Penguin query modules.

The Penguin query language is based on SQL, but it has some extensions to the syntax. For example, path-expressions are supported, as is common among query languages for object-oriented databases. A path-expression can contain many implicit joins defined in the structural model. Users can just use the corresponding connection names included in the view-object definitions; they do not have to know the specific definition of the joins, such as the names of relations and join attributes. Thus, the basic form of the query language is as the next example.

Consider the following query made by engineers in a steel plant when they found some problems in the surface quality of the coil CO123 and they wanted to investigate the causality between coil widths and carbon contents and coil surface quality.

Example of query: "Select the coils that are wider and inferior in surface quality than the coil CO123, and which are produced from charges whose carbon contents are less than that of CO123's. Then, show those coils and the corresponding slabs.

$$
\begin{array}{ll}
SELECT & ch2.slab,\ co2 \\
FROM & ChargeObj\ ch1\ ch2,\ CoilObj\ co1\ co2 \\
WHERE & co1.coil\_id = 'CO123' \quad AND \quad ch1.charge\_id = co1.charge\_id \\
 & AND \quad ch2.slab.SlabObj.coil\_to\_care() = co2 \quad AND \quad co1.width < co2.width \\
 & AND \quad co1.surface\_quality() > co2.surface\_quality() \quad AND \quad ch1.carbon > ch2.carbon
\end{array}
$$

For example, *slab* in the above statement is the name of the connection corresponding to the join between the relation $Charge$ and $Slab$, so $ch2.slab$ represents one of the nested tuples of the slabs contained in the charge object $ch2$. And $SlabObj$ is the reference to the slab object from the slab tuple in the charge object.

The method *coil_to_care*() defined in the C++ layer is also contained in the path-expression, and it is invoked with a slab object and returns the coil produced from the slab if it is likely to produce a coil with some problems in its quality. The $surface\_quality()$ is also a method to estimate the quality of the coil surface, which returns an integer value for the index of the quality. These methods should be variable according to the change of the customers' needs and the processes in the steel plant.

Thus, the language supports the various layers of data models, and offers users the power of describing the query request through all the data layers.

2.3 Query execution algorithm

2.3.1 Basic query procedure

As shown in the previous section, a query request contains requirements for all three layers of the data schema. So the query processor has to decompose the total query into the parts that can be treated in the relational SQL and the other parts containing the methods to be processed in Penguin or the application. The procedure for query processing consists of the following steps, and the more details of these steps are shown below.

1. Create a query graph to describe the whole query request.
2. Analyze the query graph to decompose the total query into partial queries.
3. Decide the sequence of calculating the partial queries, and generate the procedure of composing the final result.
4. Execute the query procedure, and instantiate a view object for the result.

2.3.2 Query graph and decomposition

At first, the query statement issued by a user is analyzed, and a query graph is generated as in Fig. 2. The vertices in the graph represent relations and the hyperedges correspond to the joins or other predicates. The query graph contains all the hidden relations and joins involved by the path-expressions. Naturally, the other predicates that explicitly appear in the where clause are also part of the graph.

The edges illustrated by the arrows are the implicit joins using methods. For example, the method *coil_to_care*() is regarded as a join from the slab relation to the coil relation. The method $surface\_quality()$ is treated as a join from the coil relation to a virtual relation "integer". Thus, the graph represents the full specification of the query request.

Those processes in the query graph have to be divided into parts so that each of them is delivered to the appropriate layer of processor. For this purpose, the query graph is modified to decompose the query into the partial queries. The parts of the graph that contain methods are separated from the other parts, with new instances of the relations connected by the equijoins.

Fig. 3 shows the result of the decomposition. The method join parts B, C, and D are separated from the other part of the graph. New instances for $R3$, $R4$, and $R5$ are created and the equijoins for them are also added to the graph so that the final query result can be calculated. Only A is a partial query consisting of relational operations. Thus, the areas enclosed with bold lines in Fig. 4 are the partial queries resulted from the decomposition, and the predicates enclosed with bold broken lines are the joins for the composition.

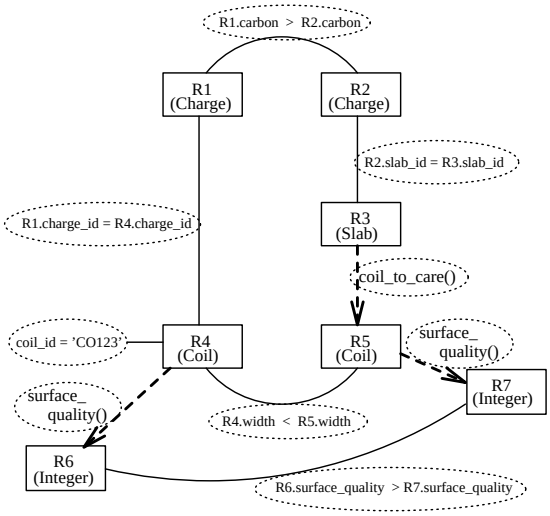

Figure 2 : Query Graph

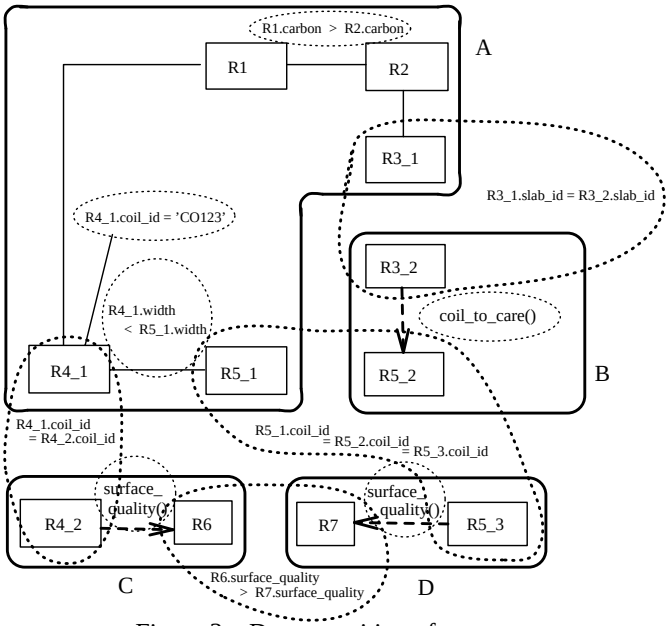

Figure 3 : Decomposition of query

2.3.3 Optimizatnion

It is true that partial queries can be independently calculated, but that approach to query execution will create large temporary relations for the partial results. So Penguin has a query optimization algorithm to reduce the size of the temporary data.

The algorithm is similar to the well-known decomposition method of query processing [7] [13]. An advantage of the decomposition approach is that during composition we do not have to use the original relations for the calculation of the partial queries. We can use the results of other partial queries instead. Because we join the partial results together, only the tuples of the partial results that participate in the answer need to be computed. This optimization is akin to semijoin reduction. Therefore, by using the results of partial queries, we can avoid unnecessary generation of tuples in temporary relations.

Another advantage is that not all the attributes are required for composition. The attributes requested in the select clause are necessary, and we also need ones used for the subsequent joins and the object instantiation process. We have to include only those attributes in the temporary results.

The query optimizer decides the sequence of calculating partial queries, and selects the attributes that are contained in the temporary relations. We use a heuristic approach, and some of these heuristics follow.

- If there is a method part with the source relation that is not joined with any other uncalculated partial query, calculate it soon because there is no possibility to make its result smaller.
- If a partial query calculates the relation joined with the source relation of an uncalculated method part, calculate it earlier than that method part if possible, to reduce the size of the source relation for the method join.
- If no uncalculated method part remains, the rest of the calculation should be entirely performed at the same time, to make the best use of the optimizer of the relational DBMS. That is, no more temporary relations are created if the rest of the computation can be entirely computed by the relational DBMS.

These rules can be applied to the example query to decide the sequence of calculating the partial queries. The partial query A is calculated first based on the second rule, because A contains the relation $R3\_1$, $R4\_1$, and $R5\_1$, which can be used for the source relations of the partial queries B, C, and D. Then, B and C have the source relation, which are not joined with other uncalculated parts any more. Hence, they are calculated before D by the first rule. After the last part D is calculated, there remains no partial query and the last rule applies. Finally, the rest of the query—only the join parts in this case—is performed. The query procedure for this example follows.

$$R(A) = \pi_{\substack{R3.slab\_id,\\ R4.coil\_id,\\ R5.coil\_id}} \sigma_{\substack{R1.charge\_id = R4.charge\_id \\ \wedge\ R4.coil\_id = 'CO123' \\ \wedge\ R1.carbon > R2.carbon \\ \wedge\ R2.slab\_id = R3.slab\_id \\ \wedge\ R4.width < R5.width}} (R1 \times R2 \times R3 \times R4 \times R5) \quad (1)$$

$$R(B) = \phi(\ R(A).R3,\ coil\_to\_care(),\ R5, \{\ R3.slab\_id,\ R5.coil\_id\ \}\ ) \quad (2)$$

$$R(C) = \phi(\ R(A).R4,\ surface\_quality(),\ R6,\ \{\ R4.coil\_id,\ R6.surface\_quality\ \}\ ) \quad (3)$$

$$R(D) = \phi(\ R(B).R5,\ surface\_quality(),\ R7,\ \{\ R5.coil\_id,\ R7.surface\_quality\ \}\ ) \quad (4)$$

$$RESULT = \pi_{\substack{R3.slab\_id,\\ R5.coil\_id}} (R(A) \bowtie R(B) \bowtie R(C) \bowtie_{\substack{R5\_2.coil\_id = R5\_3.coil\_id \\ \wedge\ R6.surface\_quality > R7.surface\_quality}} R(D)) \quad (5)$$

The $R(X)$ represents the temporary relation for X. The operations π, σ, $\times$ and $\bowtie$ are the relational operators projection, selection, Cartesian product, and join, respectively. The $R(A)$ and the final result consist of only those operations, so they are calculated by the relational database using its optimizer. The $\phi(\ X,\ f,\ Y,\ \{a1, a2, \ldots\}\ )$ represents the join of the source relation X and the target relation Y via the method f, and only the attributes $a1, a2, \ldots$ are created in the result of the join. For instance, the $R(C)$ is calculated invoking the method *coil_to_quality*() to every object which has its semantic key in the $R4$ part of the $R(A)$. Thus, the $R(C)$ is calculated from $R(A)$ which is expected to be a small relation containing only the data of coil 'CO123' in $R4$. The $R(B)$ and the $R(D)$ are also calculated using the results of other partial queries which can be smaller than the original relations. You can also see that the unnecessary attributes are projected out from the temporary results.

3 Implementation of query function

3.1 Penguin architecture

Fig. 4 shows the simplified diagram of the system configuration. The main part of the Penguin system consists of the *Structural Model Agent*, the *Template Generation Agent*, the *Object Instantiation Agent*, and the *Instantiation Navigation Agent* [6].

The *Structural Model Agent* is responsible for managing the connections between relations, which are defined by users. With the support of this agent, a user can built an integrated database of normalized relations with semantics modeled by connections.

The *Template Generation Agent* manages the definition of view objects defined on the structural model. Users can select the connections and attributes to be included in the view-objects using this agent. The view-object definition is specific to each application, so every application can use its own object schema, while sharing the common integrated structural model with other applications.

The *Object Instantiation Agent* creates object instances from the relational database based on the view-object definition. Based on requests from applications, this agent constructs the nested tuples for the view-objects, generating

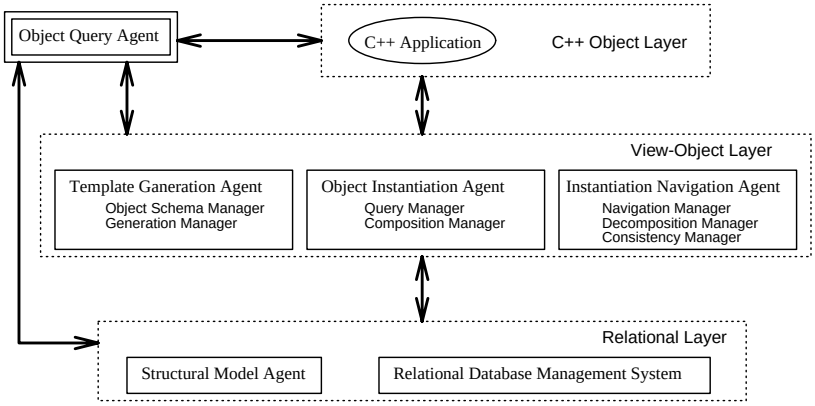

Figure 4 : Architecture of Penguin system.

SQL statements to issue for the underlying relational database. By this process, the network of objects is created in the client cache so that the following navigation is processed efficiently.

The *Instantiation Navigation Agent* controls the process of browsing the database through view objects. It also controls the update/insert/delete behavior for the database through the view-object, to maintain the restrictions among the data based on the semantics of the connections. That is, this agent avoids the problems of updating through views [3].

Under the control of these modules, applications can access the database using their own proper object definitions. Penguin also generates C++ classes corresponding to the view objects, which serve as application interfaces to access the database. The basic methods to access, update, and navigate the database are offered by the generated classes. When a user uses the C++ objects, the data are automatically mapped on those objects from the database. Therefore, there is no cost for application developers to fetch data in their programs.

The application interface also supports the query function, which offers users the capability to utilize the query execution process shown in Section 2.3. More details about the implementation of the query function are described below.

3.2 Linkage of query function with application

Fig. 5 presents the module diagram of the *Object Query Agent* in the Penguin system. It performs queries issued by users communicating with the main part of Penguin and applications.

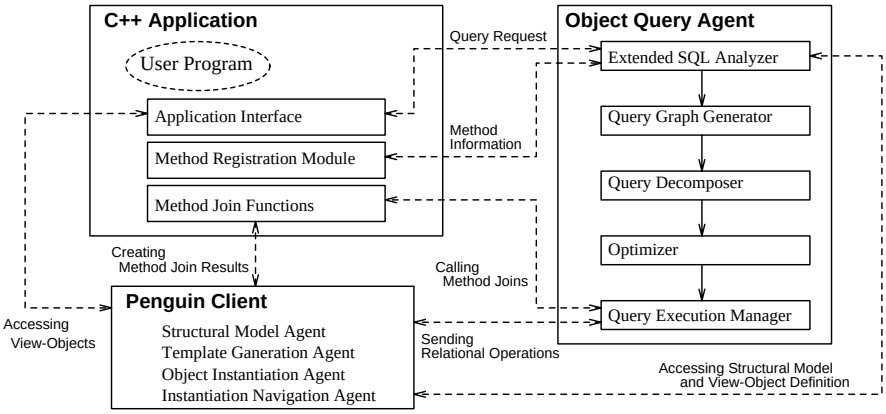

Figure 5 : Implementation of object query system

When a user uses the query function, two kinds of special modules are linked to his application program, besides the linkage of the regular application interface.

One module linked to the application is the method registration. The user has to include the method registration part in his application, to declare which methods are allowed to be used in queries. This module informs the *Object Query Agent* about the information of the available methods. From the received information, the *Object Query Agent* knows the name of the method, the type of the object to which the method is issued, and the type of the object resulted from the method. Thus, the *Object Query Agent* can recognize the methods in the query statement made by users, and can control the process of generating and decomposing the query graph.

The method join functions are also linked to the program, which are included in the user's program if the user chooses the methods as in the method registration part. A method join function is invoked when the *Object Query Agent* sends a request of method join in the process of the query execution. Then, the called function issues the corresponding method to the objects instantiated from the source relation, and the result of joining the source and destination relation is generated. Thus, the result of the method join is created as a temporary relation by the method join function.

The query function is one of the C++ functions contained in the application interface. When a user calls the query function with the extended SQL statement as shown in Section 2.2. the query request is sent to the *Object Query Agent*. It performs the requested query in the way described in Section 2.3. The request in the extended SQL is analyzed to generate the query graph, then the query is decomposed into the partial queries, and the optimizer decides the procedure to calculate the query.

The *Object Query Agent* controls the process of the calculation. As the procedure generated by the optimizer contains relational operations and method joins, the *Object Query Agent* sends the relational operation parts to the underlying database, and requests to perform the method join functions are sent to the application. Finally, the view object for the query result is instantiated, which can be accessed using the C++ object interface in the application.

As shown in this section, the *Object Query Agent* makes use of the heterogeneous and multi-layered object schema supported by Penguin, and can perform the powerful and flexible queries which contain the connections defined in the structural model and the methods in the application.

4 Conclusion

We have introduced a query system for the Penguin system and have described its implementation. The query system shown in this paper is used to support object views of a relational database, which enables many applications to share data with their own object schemata. The query system supports the multiple layers of the schema, and users can make use of the relational DBMS and C++ methods. We have also shown the system configuration for implementation of the query function and have described how queries are executed in our system. For future work, we plan to experiment with approaches to query performance such as making better use of the cache. We believe those activities will help us build more efficient and powerful applications in engineering and manufacturing.

Acknowledgements

This work was performed as part of the Penguin project at Stanford University, and other members of the project provided valuable feedback and advice.

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.

We also thank Marianne Siroker for her assistance in the preparation of this paper.

References

[1] T.Barsalou. *View Objects for Relational Databases.* Ph.D. dissertation, Stanford University, 1990.

[2] T.Barsalou and G.Wiederhold. "Complex Objects for Relational Databases." *Computer Aided Design*, Vol.22, No.8, pp. 458–468, 1990.

[3] T.Barsalou, N.Siambela, A.M.Keller, G.Wiederhold. "Updating Relational Databases through Object-Based Views." *Proc. ACM SIGMOD*, pp. 248–257, 1991.

[4] M.J.Carey, D.J.DeWitt and S.L.Vandenberg. "A Data Model and Query Language for EXODUS." Tech. Rep. Univ. Wisconsin, CS-TR-734, 1987.

[5] C.Hamon and A.M.Keller. "Two-Level Caching of Composite Object Views of Relational Databases." Submitted for publication, 1993.

[6] 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*, 1993.

[7] B.C.Ooi and R.Sacks-Davis. "Query Optimization in an Extended DBMS in Foundations of Data Organization and Algorithm." (eds. W.Litwin and H.J.Schek) *3th Int. Conf. FODO Proc.*, pp. 48–63, Springer-Verlag, 1989.

[8] M.A.Roth and H.F.Korth. "The design of $\neg NF$ Relational Databases into Nested Normal Form." Tech. Rep. Univ. Texas at Austin, TR-85-19, 1986.

[9] M.Stonebraker and M.Hirohama. "Extendability in POSTGRES." *Database Engineering*, Vol.6, p.76-83, 1987.

[10] T.Takahashi and A.M.Keller. "Querying Heterogeneous Object Views of a Relational Database." *Proc. Int. Symp. on Next Generation Database Systems and Their Applications*, pp. 34–41, 1993.

[11] G.Wiederhold. "Views, Objects and Databases." *IEEE Computer*, Vol.19, No.12, 1986.

[12] G.Wiederhold, T.Barsalou and S.Chaudhuri. "Managing Objects in Relational Framework." Stanford Univ. Tech. Rep., CS-89-1245, 1989.

[13] E.Wong and K.Youssefi. "Decomposition — A Strategy for Query Processing." *ACM Trans. on Database Systems*, Vol.1, No.3, pp. 223–241, 1976.