Querying Heterogeneous Object Views of a Relational Database

Tetsuya Takahashi
Kobe Steel, Ltd. and
Stanford University

Arthur M. Keller
Stanford University

Abstract

We present the query processing algorithm for the Penguin system. Penguin supports multiple object views on a relational database, so that data may be shared by applications with heterogeneous object schemata. Penguin also offers an interface for the C++ language. We have developed a query processing algorithm for querying composite object views. The query algorithm takes a query on a composite object view, and decomposes it into partial queries on the relational database. The final query result is composed from those partial results. We also discuss an optimization for reducing the volume of temporary data created in query processing.

1 Introduction

Object-oriented programming is becoming quite prevalent for its benefits of software productivity, quality and reusability. Software developers are eager to adopt the object-oriented approach in new applications. On the other hand, the relational model introduced by Codd [7] is so widely accepted that most new database applications use relational databases today. But there are several problems with linking object-oriented programming with databases.

The first problem is the need to linking programs written in the object-oriented paradigm with legacy data in relational databases. The relational model has a simple but powerful description and its theory is mathematically elegant. Yet the normalized data structures, suggested by relational design theory, require that data be composed in order to be treated in the object-oriented model. Many extended relational models have been studied to break this limitation of the relational model [9, 19, 21], but nested relations do not alone support object-oriented programming.

The second problem is the difficulty of sharing data among applications with heterogeneous object schemata even when using a common database. Some object-oriented database management systems are available currently [14, 15, 23], but they permit us to define only one conceptual schema for each database and do not support the view concept. This means all the applications which access the same database are forced to use the same structure of the object classes. As the result, application development is quite limited and cannot achieve the full benefit of object-oriented programming. Therefore, we believe that object-oriented databases should support view-objects as described in [24], and each application should have its own specific object schema.

The third problem is that there is not yet a standard query language for object-oriented databases, and query capabilities among object-oriented database systems vary. Several query languages have been proposed [1, 2, 6, 12, 20], but most of them are also dependent on their own data models.

The Penguin system solves these problems. The Penguin system has been developed to allow multiple applications that have their own object schema share data in a common relational database [3, 4, 5, 11, 25]. We have developed query manager for the Penguin system. We discuss how a powerful object query language can be supported on top of a relational database. We briefly introduce the data models of Penguin in the next section, and then describe the query language in Section 3. Our query processing algorithm is described in Section 4. Some further issues are also discussed in Section 5.

2 Penguin system

Penguin[1] has a multi-layered conceptual schema. The layers are the relational layer, the view object layer and the C++ object layer. We illustrate this layering using an example of a database containing data on companies, departments and employees in Fig. 1. The objects have the structures of composite objects with the nested tuples as following examples.

[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

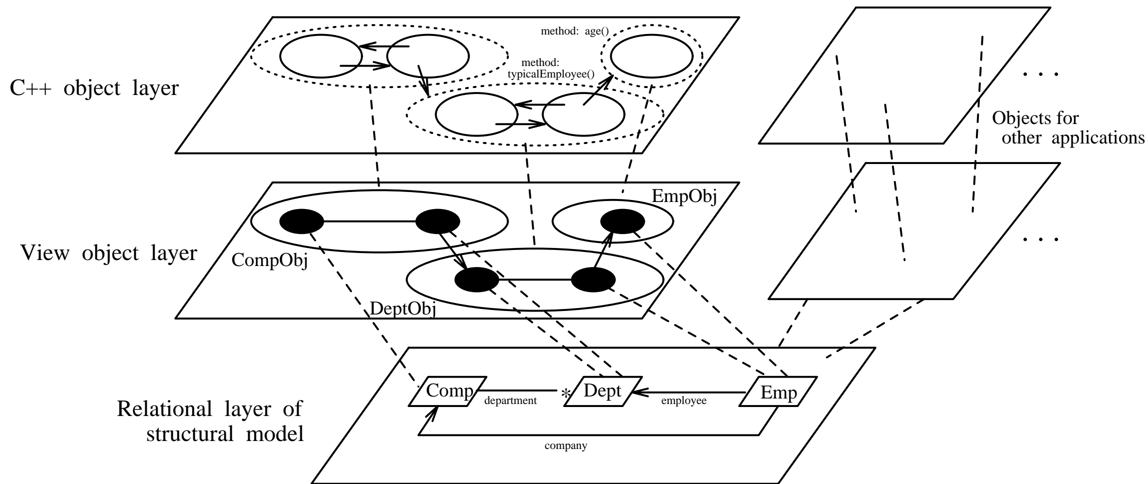

Figure 1: Data mapping in multi-layered model.

CompObj

cname	sales	department
		dname
California Steel	2,800	Development
		Personnel
Kobe Electronics	1,200	Sales
		Support
		Technical
⋮	⋮	⋮

DeptObj

cname	dname	location	employee
			ename
California Steel	Development	Palo Alto	John
			Mary
			Steve
California Steel	Personnel	San Jose	Tom
			Peter
			George
⋮	⋮	⋮	⋮

EmpObj

ename	cname	dname	salary	birth
John	California Steel	Development	38,000	1962
Tom	California Steel	Personnel	32,000	1968
Jim	Kobe Electronics	Sales	43,000	1955
⋮	⋮	⋮	⋮	⋮

A company object *(CompObj)* consists of the attributes of a company and the set of its departments. A department object *(DeptObj)* consists of a department and a set of its employees. An employee object *(EmpObj)* consists of information about an employee. We briefly describe how such objects are defined in the Penguin multi-layered schema. The details can be found in [3].

Penguin defines a structural model on top of the relational database layer to classify the relationships between the relations. The relations contained in the structural model are normalized based on the design theory of the relational database. And connections are defined between the relations, which are basically joins but also provide structural semantics for the relational model. Connections are represented in the directed-graph for the structural model. Penguin system currently supports three types of connection, *Ownership Connections* (represented by $—*$), *Reference Connections* (represented by $\longrightarrow$) and *Subset Connections* (represented by $—\supset$). The semantics in the structural model is used to avoid problems of updating databases through object views [5].

View-objects are composite objects defined on the structural model in a language-independent object layer. Each application can have its own object schema, and can share the data with other applications. That is, Penguin system organizes the composite objects from the normalized relations, based on the view-object definition which is designed specifically to each application. Thus, 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.

The view-objects support navigation through the composite object structures. One type of the navigation is accessing data via connections which link the sub-tuples contained in a view-object with the primary tuple. This type of the navigation corresponds to the data access inner view-object. The other type of the navigation is the reference to other view-object whose semantic keys are included in the source view-object. For example, *CompObj* contains the nested tuples of the department, and includes the semantic keys of *DeptObj*. A user can navigate to *DeptObj* from a tuple of the department in *CompObj*.

Penguin also generates the C++ classes corresponding to the view-objects. Fig. 2 shows a part of the C++ code automatically generated for the example application shown above. Those C++ classes consist of the members corresponding to the attributes in the view objects and the basic methods (member functions), which are available to access the view-objects (e.g., navigation, query, and update). Users can add

```
class  Comp {
  protected:
               //  members for attributes
    char  cname[ 32 ];
    int  sales;
               //members for connections
    Dept  *  department ;    // Ownership
  public:
    get_relation_name() { return( "Comp" ); }
        . . .
};

class  CompObj : public  Comp {
  public:
               //  member functions to access data
    char  *  get_cname();
    int  get_sales();
               // member functions to navigate
    CompObj_department  *  get_department();
        . . .
};
                  .
                  .
                  .
```

Figure 2: C++ code for class definition.

their own methods to these classes, and can construct applications based on those methods defined in the application layer. We selected the C++ as the first application language to implement because it is a de facto standard for object-oriented programming. But the implementation of the view-object layer is independent of the application languages, and other choices will be possible.

3 Query language

As described in the previous section, Penguin supports the multi-layered data schema of view-objects on the relational database. But users want to describe their query requests using the object schema, without having to understand the relational schema. To satisfy this desire, we have defined the Penguin query language that operates on composite objects, which Penguin must translate into query requests on the underlying relational database.

SQL is the standard query language for the relational databases. For ease of learning, we have designed a query language that is based on SQL, but with extensions to the syntax to apply to objects.

We support path-expressions, 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, and do not have to know the specific definition of the joins, such as the names of relations and join attributes.

If the query is issued in a C++ application linked with the Penguin query manager, methods in the C++ objects can be also contained in the expressions. Those methods should be defined in C++ object classes by the user, which would be invoked to an instance of the class and would also return an C++ object corresponding to a view-object.

Thus, the basic form of the query language is as follows.

```
<query-statement>   ::=  SELECT <select-list>
                         FROM <from-spec>
                         [ WHERE <boolean> ]
<select-list>   ::=  <path-expression>
                     [ { , <path-expression> } ... ]
<from-spec>   ::=  <variable-declare>
                   [ { , <variable-declare> } ... ]
<boolean>   ::=  <predicate>
                 | <boolean> AND <predicate>
<variable-declare>   ::=  <object-name>
                          <variable> [ <variable> ... ]
<path-expression>   ::=  <path-sequence>
                         [ . <path-terminator> ]
<path-sequence>   ::=  <variable> [ { . <path> } ... ]
<path-terminator>   ::=  <attribute> | <atom-method>
<path>   ::=  <connection>
              | <reference> | <object-method>
```

Many predicates and operators defined in the standard SQL, as listed below, are available in the where clause.

- =, <>, <, >, <=, >=
- +, −, *, /
- *LIKE*, *IS NULL*, *IS NOT NULL*

The predicates can also contain path-expressions. We do not support disjunction, nor nested queries.

Consider the following example.

Query: "Select the departments whose typical employee is younger than Tom, but his salary is more than twice of Tom's, even though the sales of his company is less than that of Tom's company. Show those departments and the names of their typical employees."

$$
\begin{array}{ll}
SELECT & c2.department,\ e2.ename \\
FROM & CompObj\ c1\ c2,\ \ EmpObj\ e1\ e2 \\
WHERE & e1.ename = \text{'}Tom\text{'} \\
 & AND\ \ c1.cname = e1.cname \\
 & AND\ \ c2.department.DeptObj \\
 & \qquad\qquad .typicalEmployee() = e2 \\
 & AND\ \ e2.salary > e1.salary * 2 \\
 & AND\ \ e2.age() < e1.age() \\
 & AND\ \ c2.sales < c1.sales
\end{array}
$$

This query takes advantage of the navigations defined in the view objects. For example, *department* in the above statement is the name of the connection corresponding to the join between the relation *Comp* and *Dept*, so *c2.department* represents one of the nested tuple of the department contained in the company object *c*2. And *DeptObj* in the where clause is the reference to the department object from the department tuple in the company object.

The method *typicalEmployee*() is also contained in the path-expression, which is invoked to a department object and returns a typical employee from the department object. This method is defined in the C++ application specifically by the users. The *age*() is also a method, which is defined on the employee object to calculate the age of the employee from the birthday. This method returns an atomic value of integer. Such an atomic-method can be used only to terminate

the path-expression. 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.

4 Query execution

We show the basic algorithm of the query execution in this section. As shown in the previous section, a query request made by a user contains requirements for all the three layers of the data schema. So the query processor has to decompose a query request into processing that is performed at the various levels from the relational database level up to the C++ programming level. That is, the total query is divided and categorized 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 of query processing consists of the next steps.

1. Analyze the path-expressions to specify:
   - relations
   - joins
   - attributes
   - methods

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

We describe some details of those steps in the following part of this section.

4.1 Query graph

To decompose a query request, we create a query graph. At first, the statement is analyzed and the implicit relations and joins are extracted from the path-expressions as in Fig. 3. In the tree of Fig. 3, all the relations and joins are revealed, including those which have been hidden behind the path-expressions.

Fig. 4 shows the query graph for the example query. The vertices 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.

Note the edges illustrated by the arrows in the figure, which are the implicit joins using methods. For example, the method *typicalEmployee*() is regarded as a join from the company relation to the department relation. The method *age*() is treated as a join from the employee relation to a virtual relation "integer". Those joins via methods are directional as indicated by arrows. This means such joins can be calculated only from the source relations, and the attributes of the destination relations are added to the result of the join.

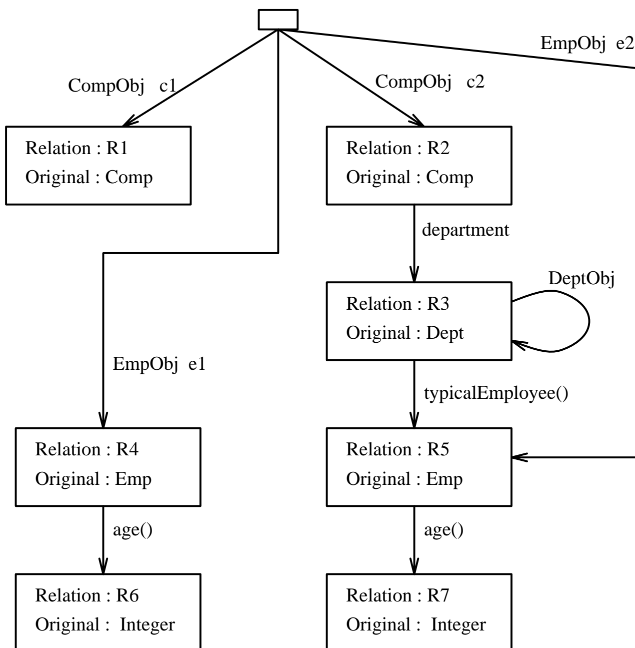

Figure 3: Analysis of path-expressions.

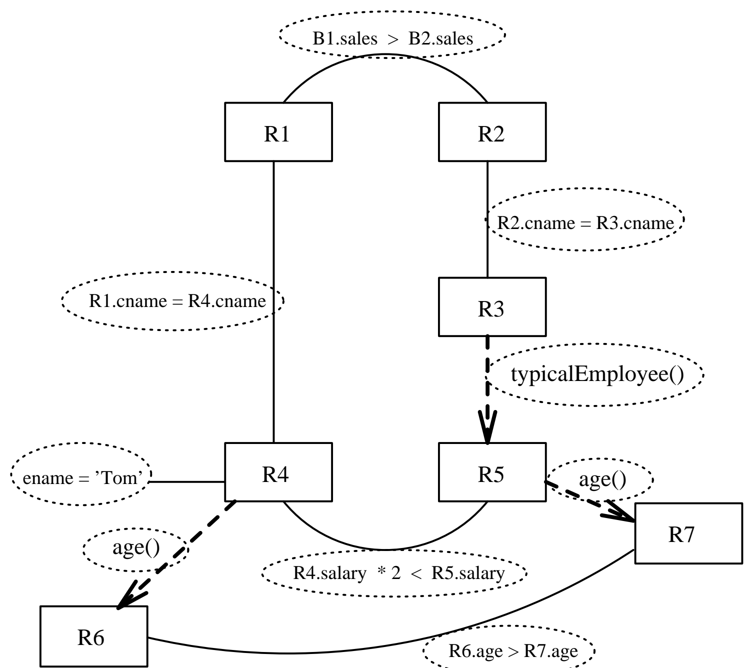

Figure 4: Query graph.

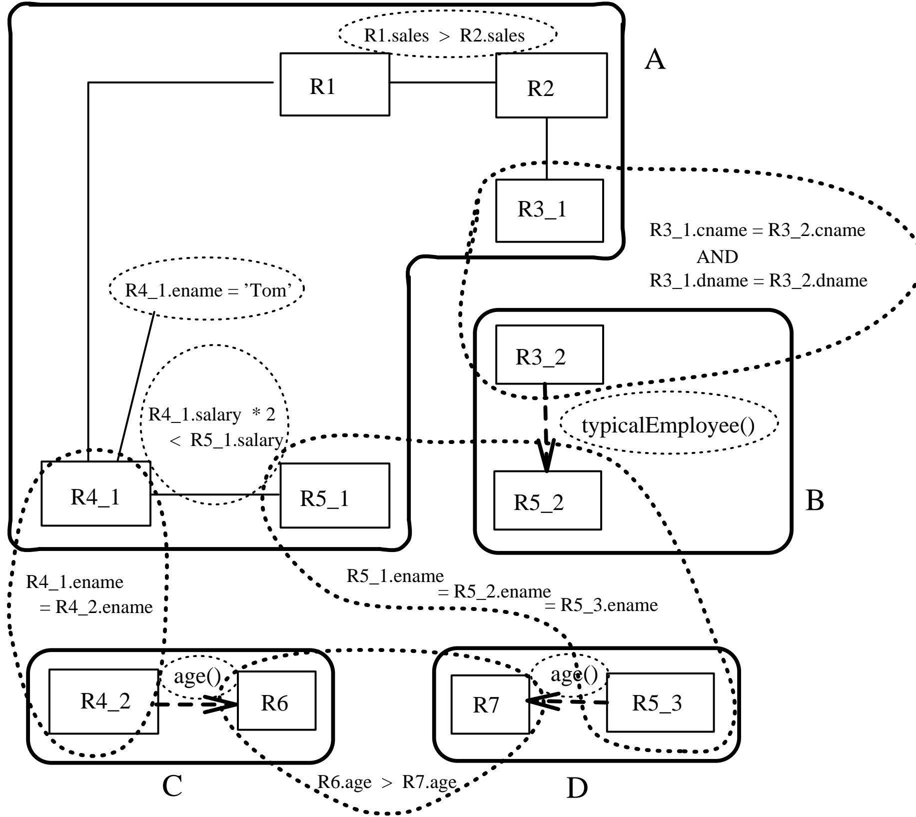

Figure 5: Decomposition of query.

Thus, the graph represents the full specification of the query request.

4.2 Decomposition and composition

The query graph contains all the operations from the relational level through the C++ object level. Those processes 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.

The algorithm of the decomposition is as follows.

1. Separate every method join as a partial query, modifying the graph in the following way.
   (a) If the method join shares regular relations, but virtual one, with other predicates, create an instance of every shared relation.
   (b) Add equijoins which connects the new instances and their original ones.
2. Make partial queries which consist of the still jointed nodes and the predicates occupied by those nodes.

Based on this algorithm, we can create the partial queries as A through D in Fig. 5.

The parts B, C and D are separated in Step.1. New instances for $R3, R4$ and $R5$ are created in Step.1a, and the equijoins for them are also added to the graph in Step.1b. No instances are to be created for the virtual relations $R6$ and $R7$, and the predicates involving such virtual relations are always used only in the final composition process. Only one part A is made in Step.2, which is a set of the relations and the connecting predicates. We do not divide such part as consists of only relational operations any more, because the underlining relational DBMS is responsible for the optimization of the relational expression. Thus, the areas enclosed with bold lines in Fig. 5 are the partial queries resulted from the decomposition, and the predicates enclosed with bold broken lines are the joins for the composition.

The implementation of the method parts (i.e., parts B through D) depends on the application language, and they are calculated in the C++ object layer. On the other hand, the other part can be performed in the relational layer. And the final result is composed by taking the joins of all the partial results. From the composed relation of the final result, the view object and the C++ object (i.e., *DeptObj*) are instantiated by arranging the nested tuples. 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. We will next discuss an approach to optimization to avoid this problem.

4.3 Optimization

Our approach to optimization is similar to the well-known decomposition method of query processing [16, 22, 26].

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. 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, the 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.

$$R(A) = \pi_{\substack{R3.cname, \\ R3.dname, \\ R4.ename, \\ R5.ename}} \sigma_{\substack{R1.cname = R4.cname \wedge \\ R4.ename = \text{“}Tom\text{''} \wedge \\ R1.sales > R2.sales \wedge \\ R2.cname = R3.cname \wedge \\ R5.salary > R4.salary * 2}} (R1 \times R2 \times R3 \times R4 \times R5)$$

$$R(B) = \phi(\ R(A).R3,\ typicalEmployee(),\ R5,\ \{\ R3.cname,\ R3.dname,\ R5.ename\ \}\ )$$

$$R(C) = \phi(\ R(A).R4,\ age(),\ R6,\ \{\ R4.ename,\ R6.age\ \}\ )$$

$$R(D) = \phi(\ R(B).R5,\ age(),\ R7,\ \{\ R5.ename,\ R7.age\ \}\ )$$

$$RESULT = \pi_{\substack{R3.cname, \\ R3.dname, \\ R5.ename}} (R(A) \bowtie R(B) \bowtie R(C) \bowtie_{\substack{R5_2.ename = R5_3.ename \\ \wedge\ R6.age > R7.age}} R(D))$$

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. We have followed the notation of [22]. 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 $age()$ 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 Tom's data 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.

5 Further discussion

5.1 Criteria for optimization

In our optimization approach, we have focussed on the size of the temporary data generated in the process of the query execution. One of the reasons for this choice is that the performance index should be independent from the implementation of the underlining relational database. Very general criteria are required, because we do not customize our system to one particular relational DBMS and will not go into the optimization performed in the relational level.

In general, the increase of the data gives rise to the frequent page fetching in relational databases, and raises the communication cost in a distributed system. Therefore, The query processing which minimize the size of the temporary data gives a fairly good solution in the most cases, even though it is not necessarily the best solution.

We have shown only heuristic rules for the optimization, but there are many ideas on the metrics to estimate the query performance predicting the size of the temporary data. The discussion on the metrics for the query optimization is beyond the scope of this paper.

5.2 Mismatch in language and data

It is pointed out that some impedance mismatch exists between the languages and the data models in object-oriented databases [8]. As mentioned in Section 2, Penguin system generates the C++ objects on which the data of the internal database are automatically mapped, and user can access, update and navigate the database using the methods offered by the C++ classes. Therefore, it is not costly for users to fetch data in their applications. Also, the language-independent view-objects alleviate the impedance mismatch since they support translation to the object paradigm.

But another type of serious mismatch can occur between languages and data models when users issue ad hoc queries, an issue relating to the lack of the closure property in object-oriented databases [1]. For example, the result of the query shown in the previous section has a compatible structure with the pre-defined view-object *DeptObj*, which contains the data of the department and the nested tuple of the employees. Hence, the view-object and the C++ object of *DeptObj* can be created for the query result, and the user can access the result just in the same way as accessing the pre-defined objects. But if a user makes the next request, some problems arise.

Query: "Select every pair of employees, where both persons work in the same department."

```
SELECT   d1.employee, d2.employee
FROM     DeptObj  d1 d2
WHERE    d1 = d2
```

In this case, the user is likely to request the following view-object for the query result.

ename	colleague ename
Tom	Peter
	George
John	Mary
	Steve
	Jim
⋮	⋮

But no compatible pre-defined object type is found for this result. To cope with such a situation, a new view is to be dynamically generated. A conceptual connection *colleague* is also introduced in this new view-object. The connection *colleague* is not found in the original schema neither, but this connection can be actually constructed on the original structural model using two sets of *employee* connection with a hidden relation *Dept* (that is, Emp $\longrightarrow$ Dept $\longleftarrow$ Emp). Some ideas of the connection with such missing nodes will be found in [10].

The problem is that the C++ does not support such a dynamic class definition mechanism, and the system cannot create C++ class definition for the ad hoc query result at run time. As shown in C++ code before, navigations correspond to pointers and member functions to access those pointers which are defined before programming, and they cannot be changed after the applications are set to run. Therefore, connections generated by ad hoc queries cannot be implemented in the same way as regular pointers. This limitation results in the situation that the user can access the data in the relational level and view-object level of interface but cannot use C++ objects. We leave this problem for further study by object-oriented programming language designers.

6 Conclusion

We have described an approach to query processing for the Penguin system. The query system shown in this paper is used to support object views of a relational database, which enables many applications to share data, each with its own object schema. The language supports the multiple layers of the schema, and users can make use of the relational DBMS and C++ methods. We have also described how the queries are optimized. The optimization is based on the decomposition approach, and the heuristics are introduced to reduce the size of the temporary data in the query process. In future work, more discussion will be done on the issues we have shown in the last section, and we also plan to experiment with approaches to query performance such as making better use of the cache. We believe those activities will help us to build more efficient and powerful applications in the object-oriented programming.

7 Acknowledgements

This work was performed as part of the Penguin project at Stanford University, and 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 and Ariadne Johnson-Slavenburg for her assistance in the preparation of this paper.

References

[1] A.M.Alashqur, S.Y.W.Su and H.Lam. *OQL: A Query Language for Manipulating Object-oriented Database.* Proc. 15th Conf. on Very Large Data Bases, p.433-442, 1989.

[2] J.Banerjee, W.Kim, K.C.Kim and H.F.Korth. *Queries in Object-Oriented Databases.* Proc. 4th Int. Conf. on Data Engineering, 1988.

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

[4] T.Barsalou and G.Wiederhold. *Complex Objects for Relational Databases.* Computer Aided Design, Vol.22, No.8, p.458-468, 1990.

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

[6] 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.

[7] E.F.Codd. *A Relational Model of Data for Large Shared Data Banks.* Communications of ACM, Vol.13, No.6, p.377-387, 1970.

[8] G.Copeland and D.Maier. *Making Smalltalk a Database System.* Proc. ACM-SIGMOD Conf., p.316-325, 1984.

[9] P.Dadam, K.Kuespert, F.Anderson, H.Blanken, R.Erbe, J.Guenauer, V.Lum, P.Pistor and G.Walch. *A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies.* Proc. ACM SIGMOD, p.356-364, 1986.

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

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

[12] W.Kim. *A Model of Queries for Object-Oriented Databases.* MCC Tech. Rep. ACA-ST-365-88, 1988.

[13] H.F.Korth and M.A.Roth. *Query Languages for Nested Relational Databases.* Tech. Rep. Univ.Texas at Austin TR-87-45, 1987.

[14] Object Design, Inc. *ObjectStore User Guide.* 1991.

[15] ONTOS, Inc. *ONTOS DB 2.2 Developer's Guide.* 1992.

[16] 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., p.48-63, Springer-Verlag, 1989.

[17] P.Pistor and F.Anderson. *Designing a Generalized NF2 Model with an SQL-type Language Interface.* Proc. 12th Conf. on Very Large Data Bases, p.433-442, 1986.

[18] M.A.Roth, H.F.Korth and D.S.Batory. *SQL/NF: A Query Language for non-NF Relational Databases.* Tech. Rep. Univ.Texas at Austin TR-85-19, 1985.

[19] 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.

[20] L.Rowe and M.Stonebraker. *The POSTGRES Data Model.* Proc. 13th Conf. on Very Large Data Bases, p.433-442, 1989.

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

[22] J.Ullman. *Principles of Database and Knowledge-base Systems.* Vol.2, p.633-733, Computer Science Press, 1989.

[23] VERSANT Object Technology. *VERSANT System Manual.* 1992.

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

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

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