Evaluation and Optimization of Query Programs in an Object-Oriented and Symbolic Information System

Song Bong Yoo and Phillip C.-Y. Sheu

Abstract—**OASIS is a database programming environment that extends UNIX with the concept of persistent objects. The OASIS query languages extend conventional database query languages with procedural methods and general control statements. As the complexity of the languages makes it difficult to devise a "query optimizer" based on a universally applicable algorithm, each query in OASIS is optimized based on a collection of "basic patterns" for which each pattern is associated with a separate query optimization algorithm. The optimization techniques for a set of basic patterns consisting of iterative statements and a set of nested statements is described. The optimization techniques discussed in the paper include an extended decomposition algorithm, evaluation of multiple conditions, data dependence analysis, and optimization of queries with arbitrary nesting.**

Index Terms—**Algorithms, database, query optimization, programming, data types and methods.**

I. Introduction

SINCE the announcement of the Fifth-Generation Project, the development of the next generation database systems has received much attention. Most research in this area has been focused on extending relational databases with mathematical logic (e.g., deductive databases [9]), with object-oriented programming (e.g., object-oriented databases [20]), or with both (e.g., deductive object bases [28]). A growing interest can also be found in the subject of database programming languages [3] with a goal to strengthen the expressive power of the existing query languages and to eliminate the need of any host language on top of a database system.

We define an object-oriented database program to be a set of statements (function calls in a LISP-flavored programming language) whose execution is sequenced by a set of control constructs. These statements in general operate on a set of programming objects (i.e., variables and constants) and database objects that are classified into different types. The major difference between a programming object and a database object is that the latter is persistent. However, the contents of a programming object can be assigned to a (compatible) database object and vice versa.

From performance point of view, we feel that a major difference between a database programming system and an ordinary programming system should be that a database program needs to be evaluated by the database programming system but not by some extensions of an ordinary compiler due to the large volume of data involved. On the other hand, it would be inappropriate to loosely couple an ordinary compiler (with a pre-processor) and a database query optimizer due to the communication overhead and the lack of global considerations.

This paper studies the approach to globally optimizing the evaluation of database programs within a prototyped object-oriented database programming environment OASIS (an object-oriented and symbolic information system), which is a database system intended to extend the conventional UNIX programming environment with persistent objects, object-oriented database programming, and symbolic information management. The OASIS query languages extend conventional database query languages with procedural methods and general control statements. As the complexity of the languages makes it difficult, if not impossible, to devise a "query optimizer" based on a universally applicable algorithm, the OASIS query interpreter optimizes the performance of OASIS programs based on a collection of "basic patterns" for which each pattern is associated with a separate query optimization algorithm. Consequently, an OASIS program can be divided into a set of segments and each segment is optimized separately.

In this paper, we describe the optimization techniques for a set of basic patterns consisting of iterative statements and a set of nested statements. Such statements occur most frequently in query programs and are different from traditional nested queries (which are mainly used for the purpose of aggregation) in nature. The optimization techniques discussed in this paper include an extended decomposition algorithm, evaluation of multiple conditions, data dependence analysis, and optimization of queries with arbitrary nesting. The conventional query decomposition algorithm [35] is extended to incorporate the evaluation of procedural methods. When a series of conditional statements is included in a nested loop, these statements can be transformed into independent statements so that common subexpressions can be shared to reduce the evaluation cost. When update operations are included in nested statements, data dependencies among statements are taken into account

Manuscript received November 29, 1989; revised May 17, 1990. This work was supported in part by the Air Force Office of Scientific Research under grant number AFOSR-89-0004.

S. B. Yoo was with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907, and is now with the Department of Industrial Automation, Inha University, Incheon, Republic of Korea.

P. C.-Y. Sheu is with the Department of Electrical & Computer Engineering, Rutgers University, Piscataway, NJ 08855.

IEEE Log Number 9208079.

for proper optimization. Finally, for a general query that is an arbitrary combination of basic patterns, a global optimization strategy is discussed.

II. Previous Work

In the past, several database programming languages have been proposed and/or implemented. A partial list includes Pascal/R [26], Rigel [24], and the type system of Ingres [29]. Some database query languages can be embedded in a host programming (e.g., SQL in PL/I [1] and QUEL in C [31]). More recently, several object-oriented databases have been proposed; a partial list includes GEMSTONE [20], [7], GEM [32], Iris [8], Ariel [18], EXODUS [4], Trellis/Owl [21] and POSTGRES [25], [30]. Most of these systems have been designed to simulate semantic data models by including mechanisms such as abstract data types, procedural attributes, rules, inheritance, union type attributes, and shared subobjects. Some of them also support certain forms of database programming. For example, in POSTGRES data-access procedures are allowed to be the value of an attribute so that these procedures can be applied to obtain the value of the attribute, EXODUS extended C++ to maintain persistent classes and persistent events, and GEMSTONE extended Smalltalk with associative object retrieval.

To our knowledge, most of the above systems have been implemented with an extended language compiler and a separate database system so that accesses to persistent objects/data are optimized by the database system at the query level. Little consideration has been given to global program optimization as an ordinary compiler does.

Along another direction, extensive research has been reported on the subject of *multiple-query optimization* and evaluation of nested queries for relational databases. In general, multiple-query optimization procedures consist of two parts [22]: identifying common subexpressions and constructing a global access plan. Although detection of common subexpressions or applicability of access paths may be computationally intractable or even undecidable if a set of arbitrary subexpressions is considered [14], various approaches have been proposed [5], [6], [12]–[14], [19]. Given the information of sharing, several search heuristic algorithms have been discussed [11], [22], [23], [27]. On the other hand, optimization of nested queries in a relational database such as SQL has been discussed extensively in [15], [16], and [10], for which the major concern for nested queries has been the treatment of aggregation functions.

Multiple queries and nested transactions, in general, can be regarded as special cases of database programs. Consequently, the techniques developed in these two areas can be applied to optimize qualified segments in a database program as we will describe later.

III. Overview of an Object-Oriented and Symbolic Information System

In this section, we briefly review the essence of OASIS and introduce the schema definition for a small database as an example.

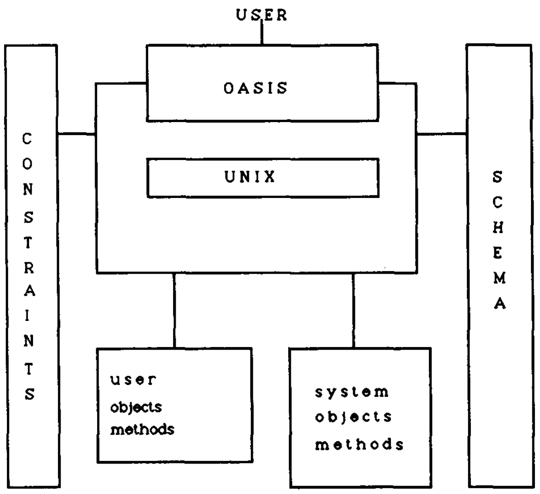


Fig. 1 OASIS Architecture.

A. The Architecture of OASIS

The overall architecture of an OASIS environment is shown in Fig. 1, which consists of a database, a knowledge base, a metaknowledge base, and an OASIS database programming language interpreter:

1) The database contains a set of methods and a set of persistent objects that are organized into classes.
2) On top of the objects and methods, there are a set of integrity constraints and a set of view definitions, stored in a textual form.
3) An OASIS database program interacts with the OASIS environment with the OASIS interpreter. A user interacts with the OASIS system with functions written in OASIS-LISP (which is an extension of LISP) or OASIS-C (which is an interpretable version of C); both types of functions can be used in an interleaved way to accomplish a complex user query.

In OASIS, five classes have been predefined: integer, float, string, symbol[1] and list, where the first four are referred to as *primitive classes.* A *list* object is a composite object with three attributes: *name*, *car* and *cdr*, where *name* stands for the name of the list, *car* refers to the first element of the list, and *cdr* refers to a list that is composed of the remaining elements in the list. A summary of the database definition language of OASIS is given in Appendix I.

A Small Database as an Example

In this section, we present a schema definition that describes a small world including various obstacles and moving cars. This example will be used for the other examples throughout this paper. The meaning of each class and attribute is self-explanatory. Note that a class (recursively) inherits the attributes and the methods of its superclass unless the class overrides them. A class can redefine a method with the same name as in its superclass and rename an attribute in its

[1] A symbol is a sequence of alphanumeric characters.

superclass (e.g., the notation TID/OID can be used to change an attribute name OID into TID). For example, each entity in the class *triangle* has TID, SIZE, COLOR, and its three nodes as its attributes.

The class hierarchy of this database is as follows:

```
< classes >
(defclass car_world (CWID int) (key(CWID))
   < subclasses >
   (defsubclass obstacles car_world (OID/CWID int) (AREA
     int) (COLOR string) (key OID))
      < subclasses >
      (defsubclass triangle obstacles (TID/OID int) (NODE1
        position) (NODE2 position)
          (NODE3 position) (key TID))
      (defsubclass rectangle obstacles (RID/OID int)
        (NODE1 position) (NODE2 position)
          (key RID))
      (defsubclass circle obstacles (CID/OIDint) (CENTER
        position) (RADIUS int)
          (key CID))

   (defsubclass operator car_world (EMP_ID/CWID int)
     (DEPT string) (CAN_DRIVE car)
          (key EMP_ID))
   (defsubclass car car_world (CAR_ID/CWID int) (YEAR
     int) (MODEL string)
          (PERMIT int) (key CAR_ID))
   (defsubclass station car_world (SID/CWID int) (PLACE
     position) (PERMIT int)
          (key SID))

< methods >
; get area of the obstacle
(defmethod (get_area int)(a obstable) (...)).

; rotate an object a by d degree
(defmethod (rotate int) (a car_world) (d int) ( ... ))

; return the distance between two objects
  a and b
(defmethod (distance int) (a car_world) (b car_world)
  ( ... ))

; return the length of the shortest path between two objects
  a and b
(defmethod (shortest_path int) (a car_world) (b car_world)
  ( ... ))

; Given two geometric object a and b, check if they are
  intersected or not

; return the overlapped rectangle; NIL if not overlapped
(defmethod (overlap rectangle) (a rectangle) (b rectangle)
(...)).

; If intersected, return the size of the intersected area;
  otherwise return 0.
(defmethod (intersect int) (a car_world) (b car_world) ( ... )).
```

B. Storage Structures

In OASIS, objects in a class are stored in the form of a relation, where each tuple corresponds to an object instance in the class. Considering this, in the remaining of this paper, we use the term "relation" and the term "class" interchangeably. An attribute value in OASIS can be a nested object and retrieval of each nested object is done directly through its key identifier (whose value is stored as the value of the attribute). For simplicity, we assume a uniform cost for referencing an attribute value.

IV. A Database Programming Language and Basic Patterns

As described earlier, an OASIS database program interacts with the OASIS environment with the OASIS interpreter. A user interacts with the OASIS system with functions written in OASIS-LISP (which is an extension of LISP) or OASIS-C (which is an interpretable version of C); both types of functions can be used in an interleaved way to accomplish a complex user query. In the rest of this paper, we shall concentrate on OASIS_LISP. The term "query" will be used interchangeably with the term "query program." A summary of programming constructs in OASIS-LISP is given in Appendix II.

An outstanding feature of OASIS-LISP that may affect the evaluation significantly is that procedural method and control structures are allowed to be used in a query. In the following, we briefly explain the following two features.

Methods in Queries: Methods are customized procedures associated with object classes. Including methods usually saves additional programming effort by calling methods within a query program. For instance, the following query retrieves a list of triangles that intersect with a rectangle and the areas of the two objects are the same.

```
(forall v1 in triangle
  (forall v2 in rectangle
    (cond (and (ieq v1.AREA v2.AREA)
        (intersect v1 v2))
      (retrieve v1.TID))))
```

Without the method *intersect* in the above example, we may need an additional program segment checking the intersection of two rectangles.

Control Structures: In OASIS-LISP, various control structures such as conditional statements and iteration loops are included to enhance the scope of traditional query languages; these include *while, do* and *forall,* where a *while* or *do* statement iterates until a condition fails or the iteration variable reaches a preset limit and a *forall* statement iterates for each instance of a given set. One example usage of iterations is to realize a transitive closure. One can find all the connections between two nodes using the transitive closure of connections.

In OASIS-LISP, most interesting relational (or set-oriented) operations can be programmed using *forall* and *cond* statements. Consequently, the basic patterns of statements can be classified according to the *forall* and *cond* statements in a query.

A. Canonical **FORALL-COND** *Statements*

A canonical query consists of a set of successively nested *forall* statements and a *cond* statement in the innermost loop. It is in the following form:

```
(forall ··· in $R_1$
  ···
    (forall ··· in $R_k$
      (cond (F action))···)
```

When the innermost *action* is a *retrieve* statement, this is equivalent to a relational query as follows (written in QUEL):

```
RANGE OF $v_1$ IS $R_1$
⋮
RANGE OF $v_i$ IS $R_i$
RETRIEVE
  WHERE condition
```

Because only arithmetic comparisons and aggregation methods are supported in a relational database, including procedural methods causes some problems to traditional relational query optimization strategies. In optimizing a query program, a canonical query can be considered as a basic pattern. When a query program is processed, it should be transformed into a canonical query if possible. For example, consider the following query that includes a *cond* statement in the middle of a set of successively nested *forall* statements.

```
(forall $v_1$ in $R_1$
  (forall $v_2$ in $R_2$
    (cond $F_1$
      (forall $v_3$ in $R_3$
        (cond ($F_2$ action)))))
```

Using the commutative law between selections and cross products [33], the above query can be transformed into a canonical query as

```
(forall $v_1$ in $R_1$
  (forall $v_2$ in $R_2$
    (forall $v_3$ in $R_3$
      (cond ((and $F_1$ $F_2$) action))))).
```

B. Nested Statements in Successive **FORALL***STATEMENTS*

Various statements can be nested in one or more successively nested *forall* statements. The basic patterns of nested queries can be classified as follows:

1) A General **COND** *Statement (TYPE-GENERAL_COND):* If we extend a canonical query with a general *cond* statement, we can obtain a query of the form:

```
(forall ··· in $R_1$
  ···
    (forall ··· in $R_k$
      (cond ($F_1$ $action_1$)
        ···
        ($F_n$ $action_n$)···).
```

where the generalized *cond* statement should be interpreted as

```
(IF $F_1$ THEN DO $action_1$)
(ELSE IF $F_2$ THEN DO $action_2$)
···
(ELSE IF $F_n$ THEN DO $action_n$).
```

2) Multiple **COND** *Statements (TYPE-MULTIPLE_COND):* If multiple *cond* statements are included inside of a set of successively nested *forall* statements, we would obtain a query of the form

```
(forall ··· in $R_1$
  ···
    (forall ··· in $R_k$
      (cond ($F_1$ $action_1$))
        ···
      (cond ($F_n$ $action_n$)···).
```

Semantically, the *cond* statements in the above query should be processed sequentially for each instance of variable bindings (i.e., each tuple in the cross products of relations $R_1, \cdots, R_k$).

3) Nested **FORALL** *Statements (TYPE-NESTED_FORALL):* When a *forall* statement is present with other statements (e.g., other *forall* or *cond* statements) at the same level of nesting, it cannot be included in the outer *forall* statements. For example, consider the following query:

```
(forall ··· in $R_1$
  ···
    (forall ··· in $R_k$
      (forall ··· in R
        (retrieve ···))
      (cond (F action)···).
```

In this example, (*forall* ··· *in R* (*retrieve* ...)) is an individual *forall* statement rather than a part of a successively nested *forall* statements. Typical nested queries in a relational language can be considered as instances of this type.

4) Assorted Statements (TYPE-ASSORTED): In general, an arbitrary combination of the statements available in OASIS-LISP can be nested. Besides *forall*, *cond* and *retrieve*, a statement in OASIS-LISP could be a method that may be a data manipulation statement such as *append*, *delete*, and *replace*. In this situation, optimization of a nested statement may be affected by the presence of other statements. In particular, when data manipulation statements are present along with other statements (e.g., *cond* and nested *forall*) inside of a set of successively nested *forall* statements, data dependences should be analyzed for proper optimization. As an example, consider the following query:

```
(forall ··· in $R_1$
  ···
    (forall ··· in $R_k$
      (cond ($F_1$ $action_1$))
      (replace $R_i$.A (plus $R_i$.A 10))
      (cond ($F_2$ $action_2$)))).
```

Because the attribute value $R_i \cdot A$, where $1 \leq i \leq k$, are updated after the first *cond* statement in each iteration, F_2 should be evaluated according to the updated values.

In fact, a canonical query or a nested statement of type GENERAL_COND, MULTIPLE_COND or NESTED_FORALL is a special case of a type-ASSORTED statement. Given a type-ASSORTED statement, those special cases should be considered first.

C. General Multilevel Statements (TYPE-GENERAL)

We define a level of nesting to be set of a successively nested *forall* statements and the body of the innermost *forall* statement, where the body of the innermost *forall* statement can include another level of nesting recursively. Generally, a nested query may consist of multiple levels of nesting, where each level of nesting would be one of the basic patterns defined above (i.e, canonical queries and nested statements of types GENERAL_COND, MULTIPLE_COND, NESTED_FORALL, and ASSORTED).

V. Processing A Canonical Query

In the previous section, we showed a canonical query is semantically equivalent to a relational query. However, including procedural methods introduces problems to conventional relational query optimization techniques.

In [28], we described a query reordering algorithm to optimize such queries, assuming a nested-loop approach is taken. For a large database, however, an optimal nested-loop algorithm could be inefficient when the number of variables involved in a query is large. Realizing this, a nonlinear search approach based on query decomposition is taken in OASIS. The definitions of the query decomposition algorithm [35] and connection graphs are summarized in Appendix III. The main idea of the decomposition algorithm can be summarized as follows:

1) Perform lower-cost operations first, i.e., in the order of selection, equi-join, general-joins, and Cartesian product.
2) Keep the temporary relations small by selecting small relations first and disconnecting the graph if possible.

In order to evaluate a canonical query written in OASIS-LISP, the original query decomposition algorithm should be modified because procedural methods were not considered. While conditions in the original query decomposition algorithm can be considered as logical methods in OASIS (see Appendix II), a conjunct in OASIS-LISP can be a method of any type. In the following, we discuss how to decrease the number of method calls when the query decomposition algorithm is followed.

In a connection graph, a procedural method can be denoted by a rectangular node, where its input arguments and output arguments are represented by input arcs and output arcs, respectively. As an example, an extended connection graph for the following query is shown in Fig. 2.

```
(forall car in car
  (forall sta in station
    (forall opr in operator
    (forall mgr in manager
      (cond (and (ieq car.PERMIT sta.PERMIT)
        (and (seq car.MODEL opr.CAN_DRIVE)
        (and (seq opr.DEPT mgr.DEPT)
        (and (shortest_path car sta PATH)
        (ile PATH 15)))))
      (retrieve all)))))
```

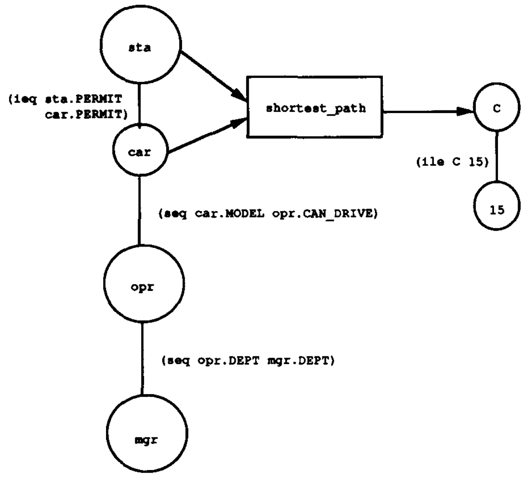


Fig. 2. An extended connection graph.

In OASIS, procedural methods are procedures that can include either simple mappings or very expensive computations (e.g., matrix computations). If a procedural method has only one input argument (we define the domain of the argument relation to be the *input relation* to the method), the method will be executed for each tuple in the relation. In this case, evaluation of the method does not change the number of tuples in the input relation. It attaches the values of output arguments to each tuple in the input relation. If the input values of a procedural method come from more than one relation and the relations are not connected by joins, the method should be evaluated for all possible combinations (of instantiations) for its input variables (i.e., all the tuples from the Cartesian product of the input relations).

Before a procedural method is evaluated, all the input relations of the method should have been instantiated or dissected because the values of input arguments are needed for evaluation. In the decomposition algorithm, evaluation of procedural methods should be included among dissections because instantiations can always be done without increasing the number of tuples in relations. During dissections, we can reduce the number of method calls by considering the effects of a dissection on an input node (i.e., a node that represents an input relation) or on a node that is connected to an input node of the method.

When two relation nodes n and m are connected by a join edge, we define the reduction factor r_n^m associated with the edge to be

$$r_n^m = \frac{|m \text{ join } n|}{|n|}.$$

Similarly, the reduction factor r_m^n can be defined as

$$r_m^n = \frac{|m \text{ join } n|}{|m|}.$$

If an input node in for a procedural method is a candidate for the next dissection, we can compute the reduction factors for all the nodes $con_1, \cdots, con_k$ which are connected to in. If the value of any $r_{in}^{con_i}$, where $1 \le i \le k$, is less than

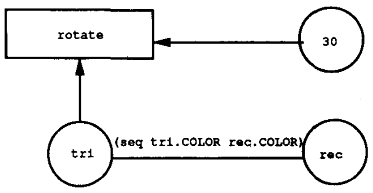


Fig. 3. A connection graph including a procedural method.

one, dissecting con_j can reduce the number of method calls. Similarly, if a candidate node $cand$ for the next dissection is connected to an input node in for a method to be evaluated, we compute the reduction factor r_{in}^{cand}. If r_{in}^{cand} is greater than one, the input node in should be dissected first to decrease the number of method calls. As an example, consider the connection graph shown in Fig. 3, assuming the cardinalities of the involved relations are given as

|*triangle*|=100
|*rectangle*|=100
|*triangle join rectangle*|=500.

According to the decomposition algorithm, ignoring the method $rotate$, either rec or tri can be selected for dissection. Assuming rec is selected, because the node rec is directly connected to the input node tri of the method $rotate$, we compute the reduction factor r_{tri}^{rec} before rec is dissected:

$$r_{tri}^{rec} = \frac{|triangle\ join\ rectangle|}{|triangle|} = 5.$$

This implies that if the join is performed, the method $rotate$ should be evaluated for each tuple in the result of the join, and the cardinality of which is five times as large as that of the relation *triangle*. Considering this, the node *tri* should be dissected first, and, in this case, the number of method calls is 100.

If a method has more than one input relation, then any connection between the input relations should be considered before the method evaluation. When two input relations for a method are connected directly by an edge, evaluation of the edge may decrease the number of method calls because the method will be evaluated for only those pairs of tuples that satisfy the condition on the edge. This is a slight modification to the dissection operation because the method calls will be deferred until after the edge is evaluated. When two input relations for a method are connected indirectly (i.e., they are connected through some other nodes), we can compare the cardinalities of the resulting relation produced by the connections and the Cartesian product of the two input relations to choose a smaller one. If the former is chosen, evaluation of the method should be deferred until all the involved edges are evaluated.

Considering the above, the modified query decomposition algorithm can be summarized as follows:

Algorithm 1—Modified Query Decomposition Algorithm

Input: A connection graph for a canonical query.
Output: An access plan.

1) Do all instantiations. Here a method is instantiated (executed) if all of its input arguments have been instantiated.
2) Select a relation node n for dissection based on the original query decomposition algorithm.
   2.1) If n is an input relation for a method to be evaluated, compute the reduction factors with respect to all the nodes $con_1, \cdots, con_k$ connected to n. If any reduction factor $r_n^{con_i}$, where $1 \leq i \leq k$, is less than one, select the node con_j, which produces the minimal $r_n^{con_j}$ for dissection.
   2.2) If the node n is connected to an input node in of a method to be evaluated, compute the reduction factor r_{in}^{n}. If r_{in}^{n} is greater than one, select the node in for the next dissection.
3) Perform dissection on the selected node n'. If at this point all the input nodes to a method are only connected through the method, evaluate the method for all the tuples in the Cartesian product of the input nodes.
4) Repeat steps 1–3 recursively until no edge remains.
5) Return the Cartesian product of the relations saved so far.□

Example 1: Consider the connection graph shown in Fig. 2. According to the original decomposition algorithm, either opr or car can be selected for dissection. Assume r_{car}^{opr} is less than one, where

$$r_{car}^{operator} = \frac{|car\ join\ operator|}{|car|}.$$

According to step 2.1) of Algorithm 1, we dissect *opr* first. After the dissection, the relations car and $manager$ can be instantiated. Subsequently, the join between $operator$ and $manager$ is computed. Because the method $shortest\_path$ has two input nodes and they are connected directly, it should be evaluated for each tuple in the result of the join between the two relations car and $station$. Finally, all the car-$station$ pairs with the shortest distance less than or equal to 15 are computed, and the Cartesian product of such pairs and all the $operator$-$manager$ pairs computed earlier are returned as the result. □

The modifications introduced in Algorithm 1 may change the basic order of dissections in the original decomposition algorithm only when the change can reduce the number of method calls. In addition, when a node to be dissected is one of the input nodes of a method, evaluation of the method is deferred until all the input arguments are instantiated.

VI. Processing a Type-GENERAL_COND or Type-MULTIPLE_COND Query

A type-GENERAL_COND or type-MULTIPLE_COND query may contain multiple conditional clauses (i.e., conditions and associated actions). Basically these conditional clauses can be evaluated sequentially by nested iterations. However, if the order of evaluation is not relevant to the meaning of the query (e.g., read-only queries in general), for each conditional clause, we can derive a relation whose tuples satisfy the condition using relational operations. Subsequently, evaluation of multiple conditions can be optimized by considering common subexpressions [15] [16]. A heuristic approach to optimize multiple conditions will be presented later by modifying the decomposition algorithm.

A. Multiple Conditions

We recite the general syntax of type-GENERAL_COND queries as follows:

$(forall \cdots in\ R_1$

$\cdots$

$(forall \cdots in\ R_k$

$(cond\ (F_1\ action_1)$

$\cdots$

$(F_n\ action_n) \cdots)$

The simplest approach to processing a query of the above form would be to take a Cartesian product of all the relations involved in the *forall* statements, then test each condition in turn. Assuming that $|R_i| = n_i$ (for $1 \leq k$), the cost of evaluating such a query would be of the order $O(n_1 \times \cdots \times n_k)$. However, if we collect only those tuples that will be used for the actions in the conditional clauses, performing the Cartesian product of all the involved relations can be avoided.

As mentioned earlier, the *cond* statement in the above is semantically equivalent to an IF-THEN-ELSE statement. Among the tuples in the Cartesian product of $R_1, \cdots, R_k$, only those which satisfy at least one of the conditions will be executed. In terms of relational algebra [33], one of the actions is executed for the following candidate relation:

$$R_{cand} = \sigma_F(R_1 \times \cdots \times R_k)$$

where $F = F_1 \vee \cdots \vee F_n$. The disjunction of conditions can be expanded using *unions*, that is

$$R_{cand} = \sigma_{F_1}(R_1 \times \cdots \times R_k) \cup \cdots \cup \sigma_{F_n}(R_1 \times \cdots \times R_k).$$

Similarly, consider a type-MULTIPLE_COND query in the following form:

$(forall \cdots in\ R_1$

$\cdots$

$(forall \cdots in\ R_k$

$(cond\ (F_1\ action_1))$

$\cdots$

$(cond\ (F_n\ action_n) \cdots)$

In this case, a candidate relation for each *cond* statement can be computed. The candidate relation R_{cand_i} for $action_i$, $1 \leq i \leq n$, contains only those tuples that will be actually used for the execution of $action_i, 1 \leq i \leq n$:

$$R_{cand_i} = \sigma_{F_i}(R_1 \times \cdots \times R_k).$$

The evaluation of R_{cand} consists of relational operations and procedural methods, and it can be optimized by Algorithm 1. The evaluation cost can be reduced by considering the sharing of common expressions among R_{cand_i}, $1 \leq i \leq n$. In the next subsection, Algorithm 1 is further modified to evaluate multiple conditions.

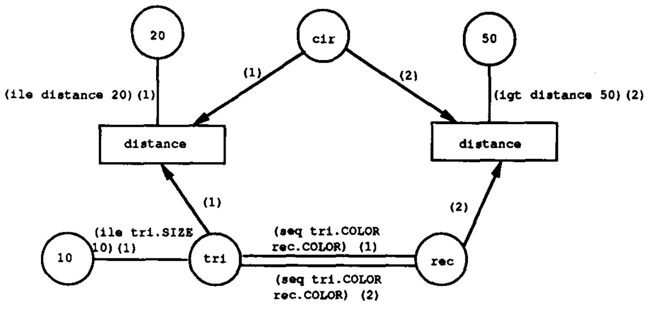


Fig. 4. A connection graph for multiple conditions.

B. Processing Multiple Conditions

As discussed in [6], a connection graph can be extended as follows to represent multiple conditions. First, one candidate relation should be returned as the answer for each condition. Second, the priority in selecting the nodes should be changed because some instantiations may prevent later sharings.

Now, given a node for each relation R_i, $1 \leq i \leq n$, a conjunct in each condition can be added as an edge. The label of each edge is added with the condition number from which it comes. For example, considering the following query, its connection graph is shown in Fig. 4.

C_1 : (*and* (*ile tri.* AREA *10*)

(*and* (*seq tri.* COLOR *rec.COLOR*)))

C_2 : (*and* (*seq tri.*COLOR *rec.COLOR*)

(*igt* (*distance rec cir*) *50*))

According to [6], in processing an extended connection graph, each condition is evaluated separately unless some sharing is possible. The basic two operations (i.e., instantiation and dissection) were modified as follows:

1) After an instantiation has been performed on a relation, the edge and the node corresponding to the constant are deleted, and the relation node is turned into a small node. If two conditions are identical, only one small node is created.
2) When the join conditions are different between two nodes, one dissection will be done for each condition and a separate set of constant nodes is created. For two conditions that are identical, only one set of constant nodes is produced.

When the above two types of basic operations are executed, the execution cost can be reduced by sharing common subexpressions among conditions. However, in order to achieve the maximum saving, a complete search with a combinatorial complexity is necessary [14]. In most cases, a heuristic approach with a relatively small search space would be viable. In evaluating multiple conditions heuristically, the most important thing is to defer the instantiations properly. If two conditions share a join and one of these two includes a selection on one of the relations (see Fig. 5(a) for an example), the common join should be executed first. If two conditions share a join and each of them has a different selection on one of the relations (see Fig. 5(b) for an example), the two relations have to be evaluated separately. However, as shown in Fig. 5(c), two selection conditions could be comparable (i.e., one

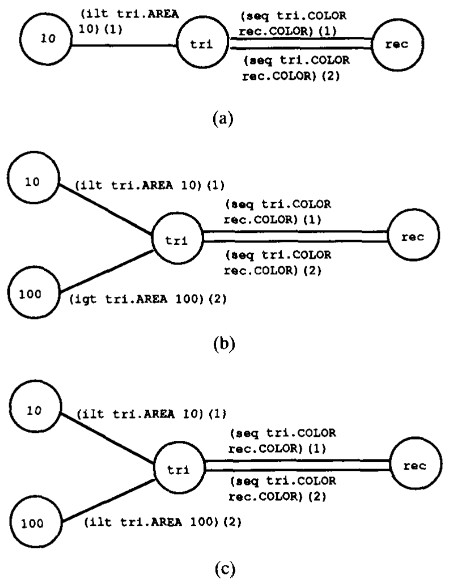


Fig. 5. Examples of ordering common expressions.

condition subsumes the other). In other words, the result of one selection (tri.AREA < 100) is a super set of the result of the other selection (tri.AREA < 10). In this situation, the more general selection (tri.AREA < 100) can be executed first followed by the join operation. The other selection can be executed based on the result of the join.

When methods are considered, the order of evaluation with sharing of subexpressions as described above may conflict the procedure described in the previous section. In other words, evaluating common subexpressions may increase the number of method calls. To avoid this, we can dissect an input argument of a method if the cardinality of the input node is known to be increased (by computing the reduction factors) after including it in a common subexpression. Considering the above, a modified decomposition algorithm that considers the evaluation of multiple conditions and methods can be summarized as follows:

Algorithm 2—Query Decomposition Algorithm for Multiple Conditions

Input: A connection graph for multiple conditions.
Output: An access plan.

1) Do instantiations on relation nodes that are not incident upon any common edge. Here a method is instantiated only if all of its input relations have been instantiated.
2) Select a node n for dissection in the following order of preference:
   1.1) Select a node that is incident on a common join edge between two relation nodes for selection. If the node is connected to an input node in of a method and the reduction factor r^n_{in} is greater than one, dissect in instead of n. If the node is connected to any constant with a selection edge, do the following:
      1.1.1) If the selection is common to the set of conditions that share the common join edge, execute the instantiation before the node is dissected.
      1.1.2) If the node is connected to a different constant for each of the conditions that share the common join edge and these selections are comparable, execute only the most general selection before the node is dissected. The other selections are done immediately following the dissection.
   1.2) Select a node that is incident on a common join edge between a relation node (i.e., itself) and a method for dissection. This choice can be superceded by the same considerations listed in step 3 of Algorithm 1. If the node is connected to any constant with a selection edge, do the following:
      1.2.1) If the selection is common to the set of conditions that share the common edge, execute the instantiation before the node is dissected.
      1.2.2) If the node is connected to a different constant for each of the conditions that share the common edge and these selections are comparable, execute only the most general selection before the node is dissected. The other selections are done immediately following the dissection.
   1.3) If no node can be selected in the above, select a node for dissection according to steps 2 and 3 of Algorithm 1.
3) Do steps 1–2 recursively until no edge remains.
4) Return the Cartesian product of the relations saved so far for each condition. □

Example 2: Consider the multiple queries shown in Fig. 4. Initially, no instantiation is possible since none of them is shared. According to step 1.1 of Algorithm 2, the node tri is be selected for dissection in Step 1 (This is arbitrary, the node rec can be selected as well.) Assuming that the reduction factor between tri and rec is 5, rec is dissected instead of tri. The method $distance$ with input node rec can be evaluated in Step 2. In Step 3, the common join with condition (*seq* tri.COLOR *rec.COLOR*) is performed. Subsequently, the selection (*ile* tri.AREA 10) is be executed before the evaluation of the method $distance$. □

The algorithm for multiple queries described in this subsection considers procedural methods in sharing of common subexpressions among multiple conditions. One major modification made to the decomposition algorithm presented in [6] is the change of the order of preference in selecting operations. Clearly, this algorithm does not generate a globally optimal access plan while the methods in [27] and [22] do. The procedure generated by this algorithm could be noticeably expensive compared with the case in which each common subexpression is extremely high while it can be avoided if each condition is processed separately. In order to prevent this situation, we could include some checking procedure prior to the evaluation of each common subexpression. For example, we can estimate the cardinality of the result relation for a common subexpression and take alternatives if it is too large.

C. Program Transformation

Once each $R_{cand_i}, 1 \leq i \leq n$, is obtained, a query of type-GENERAL_COND given earlier in this section can be transformed into the following:

```
(forall $v_1$ in $R_{cand_1}$
  $action_1$)
(forall $v_2$ in ($R_{cand_2}$ - $R_{cand_1}$)
  $action_2$)
⋮
(forall $v_n$ in ($R_{cand_n}$ - $R_{cand_2}$ - ⋯ - $R_{cand_n - 1}$)
  $action_n$)
```

Using a temporary relation, a number of subtraction operations can be saved in the above query:

```
(forall $v_1$ in $R_{cand_1}$
  $action_1$)
(forall $v_2$ in ($R_{cand_2} - R_{cand_1}$)
  $action_2$)
(set temp ($R_{cand_1} \cup R_{cand_2}$))
(forall $v_3$ in ($R_{cand_3}$ − temp)
  $action_3$)
(set temp (temp ∪ $R_{cand_3}$))
⋮
(set temp (temp ∪ $R_{cand_n - 1}$))
(forall $v_n$ in ($R_{cand_n}$ − temp)
  $action_n$)
```

Similarly, a query of type-MULTIPLE_COND can be transformed into the following:

```
(forall $v_1$ in $R_{cand_1}$
  $action_1$)
(forall $v_2$ in $R_{cand_2}$
$action_2$)
⋮
(forall $v_n$ in $R_{cand_n}$
  $action_n$)
```

The evaluation cost of the query can thus be reduced by considering common-subexpressions in the transformed query.

VII. Processing a TYPE-NESTED_FORALL Query

When a *forall* statement or a set of successively nested *forall* statements is present with other statements in the body of another set of successively nested *forall* statements, a query is not canonical and a different optimization technique is needed. In OASIS, a type-NESTED_FORALL query is transformed to a canonical query if it is equivalent to a traditional nested query [15]. If a type-NESTED_FORALL query cannot be transformed into a canonical one, the nested *forall* statement can be optimized by avoiding repeated processing of invariant computations inside of the statement. In this subsection, we describe an approach to detecting loop invariants inside of a nested *forall* statement. To start, given a nested *forall* statement within another nested *forall* statement, we define its associated *inside loops* to be all the *forall* statements included in the statement and its associated

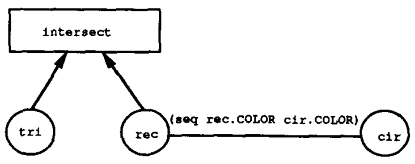


Fig. 6. A connection graph.

outside loops to be all the *forall* statements that iterate on the nested *forall* statement. As an example, consider the following query:

```
(forall $v_1$ in triangle
  (forall $v_2$ in rectangle
    (forall $v_3$ in circle
      (cond ((and (intersect $v_1$ $v_2$)
          (seq $v_2$.COLOR $v_3$.COLOR))
        (retrieve $v_3$.AREA into Temp)))
  (cond (ieq $v_1$.AREA imax" " AREA)
      (retrieve $v_1$.TID)))))).
```

We can derive the following:

$$forallstatement = (forallv_2intriangle \cdots)$$

$$insideloops = \{(forallv_2inrectangle \cdots), (forallv_3incircle \cdots)\}$$
$$outsideloops = \{(forallv_1intriangle \cdots)\}.$$

Given a nested *forall* statement, a connection graph can be constructed for its associated inside loops. The connection graph for the above query is shown in Fig. 6. Note that as before, we use arrows to represent input arguments for procedural methods.

Given a connection graph, we traverse it from all the nodes corresponding to the relations included in the outside loops (e.g., *triangle* in this example). In the traversal, we follow all incident arrows and edges on a current node, where arrows are traversed only in the forward direction. After the traversal, we can collect all the edges that have not been passed as loop invariants, and these edges can be precomputed outside of the nested *forall* statement only once. In this example, we start the traversal with the node *tri*. The traversal soon terminates at the node *intersect*; thus leave the edge between *rec* and *cir* not traversed. Consequently, the join between *tri* and *rec* is considered loop invariant and can be precomputed and saved. As a result, the above query can be transformed into the following:

```
(forall $v_1$ in triangle
  (forall $v_2$ in new
    (cond (intersect $v_1$ $v_2$.RECTANGLE)
      (retrieve $v_2$.AREA into Temp)))
  (cond (ieq $v_1$.AREA (imax Temp "" AREA))
    (retrieve $v_1$.TID)))
```

where the temporary relation *new* can be computed as

```
(forall $v_3$ in rectangle
  (forall $v_4$ in circle
    (cond (seq $v_3$.COLOR $v_4$.COLOR)
      (retrieve $v_3$ and $v_4$.AREA into new)))).
```

VIII. Processing a Type-ASSORTED Query

When a number of statements are arbitrarily combined and nested in a set of successive *forall* statements, evaluation or optimization of a nested statement may be affected by other nested statements. In this section, update statements such as *append, delete* and *replace* are considered.

Evaluation Of Assorted Statements

A statement in OASIS could be a *forall* statement, a *cond* statement, a method, or an update statement. Even though a method can stand alone as a statement, in most cases methods are included in a *forall* or a *cond* statement as a part of its condition or its action.

Adjacent *forall* or *cond* statements can usually be evaluated with the consideration of common subexpression sharing unless update operations are included in one of these statements. If no update operation is included, connection graphs for these statements can be merged into one and Algorithm 2 can be applied to evaluate this graph. For example, the following type-ASSORTED query can be considered as two subqueries, and the merged connection graph for this query is shown in Fig. 7(a).

(*forall* v_1 *in triangle*
  (*forall* v_2 *in rectangle*
    (*forall* v_3 *in circle*
      (*cond* ((*and* (*seq* v_1.*COLOR* v_2.*COLOR*)
        (*ieq* v_2.AREA v_3.AREA)))
      (*retrieve* v_3.AREA)))
  (*cond* (*seq* v_1.COLOR v_2.COLOR)
    (*retrieve* v_1.TID))))

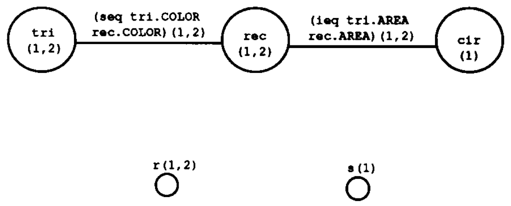
can be decomposed into two subqureies:
Statement 1:

(*forall* v_1 *in triangle*
  (*forall* v_2 *in rectangle*
    (*forall* v_3 *in circle*
      (*cond* ((*and* (*seq* v_1.*COLOR* v_2.*COLOR*)
        (*ieq* v_2*AREA* v_3*AREA*)))
      (*retrieve* v_3.AREA)))))
Statement 2:

(*forall* v_1 *in triangle*
  (*forall* v_2 *in rectangle*
    (*cond* (*seq* v_1.*COLOR* v_2.*COLOR*)
      (*retrieve* v_1.TID))))

The merged connection graph for this query is shown in Fig. 7(a).

Note that in the connection graph shown in Fig. 7(a), each relation node is marked with the statements in which it is iterated. In this example, relations *triangle* and *rectangle* are iterated in both statements, but relation *circle* is iterated only in statement 1. Given a merged connection graph, Algorithm 2 can be applied to process it until no edge remains. A candidate relation for each statement i can be generated by a Cartesian product of all the disjointed nodes marked with i.



tri (1,2)
(seq tri.COLOR rec.COLOR) (1,2)
rec (1,2)
(ieq tri.AREA rec.AREA) (1,2)
cir (1)
r(1,2)
s(1)

Fig. 7. A merged connection graph for a type-ASSORTED query.

The connection graph shown in Fig. 7(a) can be processed by dissecting the node *rec* first since it disconnects the graph and it is incident on a common join edge. After this operation, two nodes $r(1,2)$ and $s(1)$ remain as shown in Fig. 7 (b), where the node $r(1,2)$ represents the result of the join between the relation *triangle* and a tuple t in the relation *rectangle*, and (1,2) denotes that this node belongs to statements 1 and 2. Similarly, $s(1)$ stands for the results of the join between relation *circle* and a tuple t in *rectangle*, and it belongs to statement 1. As a result, the candidate relations for statements 1 and 2 can be generated as

$$R_{cand_1} = \cup_{t \text{ in } r(1,2)}(r(1,2) \times s(1))$$
$$R_{cand_2} = \cup_{t \text{ in } r(1,2)}(r(1,2)).$$

Evaluation of n consecutive statements by Algorithm 2 is viable when the summation of the cardinalities of all the candidate relations (i.e., $\sum_{i=1}^{n} |R_{cand_i}|$) is less than the cardinality of the Cartesian product of all the relations involved and there is a reasonable amount of sharing. If at least one statement runs on all the combinations of tuples (i.e., the Cartesian product) of the relations involved, applying Algorithm 2 cannot reduce the evaluation cost compared to the simple nested iterative approach. A *cond* statement with a condition that is always true or a method that stands alone as a statement are examples. Given a sequence of statements nested in a set of successively nested *forall* statements, estimating $\sum_{i=1}^{n} |R_{cand_i}|$ and the amount of sharing prior to applying Algorithm 2 is necessary.

B. Update Operations in Query Programs

Evaluation of multiple statements is more restricted if update operations (i.e., *append, delete,* and *replace*) are involved. When update operations are included in a series of statements, some relations could be modified by a statement. Consequently, any statement that follows it and uses the modified relations should wait until the modification is done before it is evaluated. This is an example of *datadependency* [34] [17], and we say that the latter statement *dependson* the former. Generally, when there are data dependencies among nested statements, a statement cannot be evaluated separately using the candidate relations described in the previous subsection. For example, consider the following query:

(*forall* v_1 *in triangle*
  (*forall* v_2 *in rectangle*
    (*forall* v_3 *in circle*
      (*cond* ((*and* (*seq* v_1.*COLOR* v_3.*COLOR*)
        (*ieq* v_2.AREA v_3.AREA)))

(*replace* v_3.AREA (*iadd* v_3.AREA 10))))
(*cond* ((*and* (*intersect* v_1 v_3)
(*ieq* v_2.AREA v_3.AREA)))
(*retrieve* (*list* v_1.TID v_2.RID v_3.CID))))).

We have two statements nested (in parallel) in a set of nested *forall* statements:

S_1 : (*cond* ((*and* (*eq* v_1.COLOR v_3.COLOR)
(*ieq* v_2.AREA v_3.AREA)))
(*replace* v_3.AREA (*iadd* v_3.AREA 10))))
S_2 : (*cond* ((*and* (*intersect* v_1, v_3)
(*ieq* v_2.AREA v_3.AREA)))
(*retrieve* (*list* v_1.TID v_2.RID v_3.CID)))

In the example, S_1 updates the relation *circle*, and S_2 reads the value of the relation. The execution of S_2 should wait until S_1 is executed in each iteration. Because the relation *circle* might be updated again in the next iteration, the execution of S_2 cannot be postponed to the next iteration either. As a result, the nested statements S_1 and S_2 should be evaluated sequentially in each iteration of the *forall* loops.

We define the *read_set* and the *write_set* for each statement as follows. The read_set of a statement is a set of relations whose contents are read by the statement. Similarly, the write_set of a statement is a set of relations whose contents are modified by the statement. This can be rewritten as

$$read\_set(S_i) \triangleq \{R| \text{ the contents of } R \text{ are read by } S_i\}$$
$$write\_set(S_i) \triangleq \{R| \text{ the contents of } R \text{ are modified by } S_i\}$$

Given a sequence of nested statements $S_1, \cdots, S_n$ in a set of nested *forall* loops, they can be numbered so that $i < j$ if S_i comes before S_j in the sequence. The procedure of finding the data dependency among a set of nested statements can be summarized as follows.

1) Derive *read_set* and *write_set* for each statement $S_i, 1 \leq i \leq n$.
2) We have a dependency pair (S_i, S_j) if
   i) $i < j$
   ii) $write\_set(S_i) \cap read\_set(S_j) \neq \emptyset$
   iii) $write\_set(S_i) \cap read\_set(S_j) \cap write\_set(S_k) = \emptyset$, for all $i < k < j$.
   In each dependency pair (S_i, S_j), S_j depends on S_i.

Example 3: For the example query in this subsection, we can derive the following:

$$read\_set(S_1) = \{triangle, rectangle, circle\}$$
$$write\_set(S_1) = \{circle\}$$
$$read\_set(S_2) = \{triangle, rectangle, circle\}$$
$$write\_set(S_2) = \{\}.$$

Through a dependency analysis, (S_1, S_2) is found to be a dependency pair because the set $\{R|R \in write\_set(S_1) \cap read\_set(S_2)\}$ (which is $\{circle\}$) is not empty. □

Given a type-ASSORTED query that includes update operations, we search for any data dependency among nested statements as described above. If no data dependency is included in the query, it can be evaluated as described in the previous subsection. Otherwise, the query should be evaluated by nested iterations.

IX. Processing a Type-General Query

A type-GENERAL query may contain multiple levels of nesting, where each level of nesting would be one of the basic patterns, i.e., canonical, GENERAL_COND, MULTIPLE_COND, NESTED_FORALL, or ASSORTED. Given a type-GENERAL query, first it can be optimized globally by moving some computations to outer levels of nesting from inside. Then, it can be processed by applying the optimization techniques discussed earlier in this paper.

A. Global Optimization

In a query with multilevels of nesting, computation in a nested loop is supposed to be evaluated for each iteration of an outer loop. In this situation, if some computations in a nested loop could be executed in an outer loop without changing the results, the overall evaluation cost can be reduced.

Detection of loop-invariants has been discussed earlier in this paper. Loop-invariants of a loop denote those computations whose results do not depend on the iteration variables of the loop. In some cases, even those computations that are not loop-invariants can be removed from a loop if necessary variables are saved properly. Generally, we can postpone any computation until its results are used by some others. The reason for postponing computations is that the amount of postponed computations may be reduced after they have gone through database operations such as joins and selections.

To optimize a query globally, levels are introduced in a connection graph for queries with multiple levels of nesting. Levels of nesting can be numbered by setting the outermost level to level 1 and increasing the number by one as it goes down to inner loops. Two adjacent levels are divided by a dotted line in a connection graph. A connection graph with the notation of levels for the following nested query is shown in Fig. 8.

(*forall* v_1 in *obstacle*
(*forall* v_2 *in rectangle*
(*forall* v_3 *in rectangle*
(*cond* ((*and* (*setq INTSEC intersect* (overlap v_2, v_3)
(*ias* (*get_area INTSEC*) *s*))
(*retrieve* v_3.AREA *S into Templ*))
(*cond* (*ieq* v_2.AREA *imax*(*Templ " " AREA*)))
(*retrieve* v_2.TID *Templ.S into Temp2*))
(*cond* (*and* (*include Temp2*.TID v_1) ((*igt Temp2*.S 100)))
(*retrieve Temp2*.TID)))))).

In Fig. 8, the dotted arrows designate direct copies between levels and they do not represent any computation. In this example, the derived attribute S in level 3 is not used until the outermost query (i.e., level 1) is evaluated. We can notice that the number of calls for the method *get_area* can be reduced significantly if we defer the evaluation until level 1. To avoid this, the input argument $INSEC$ should be saved in temporary relations.

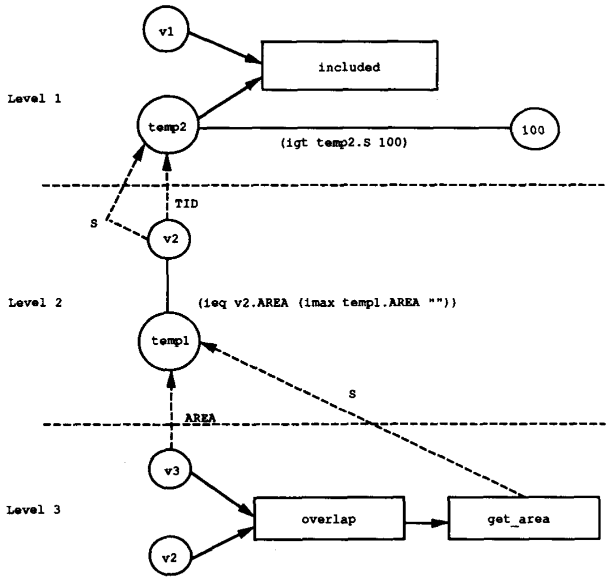


Fig. 8. A connection graph for a nested query.

In order to generalize this observation, we need a simple data-flow analysis [2]. Computations in a query program can be divided into two types: relational operations (e.g., join and selection) and procedural methods. For each operation C, we define the following:

USE[C] = the set of variables whose values are used by C.
GEN[C] = the set of variables whose values are generated by C.

For a relational operation, the set *USE* includes all the relations and constants that are involved in the operation and the set *GEN* includes the result relation. For a procedural method, the set *USE* includes all the input arguments and the set *GEN* includes all the output arguments. For each level L_i, *GEN*$[L_i]$ includes the iteration variables that can be "used" in L_i and all the variables available from all the temporary relations in L_i. According to the above, *GEN*$[L_i]$ includes all the iteration variables for level L_j, where $1 \leq j \leq i$.

To analyze the data flows in each level L_i, we first derive the *USE* and *GEN* sets for all the operations included. For each *GEN* in L_i, search all *USE* sets in L_i to check if any variable in it is used in that level. If all the variables in *GEN*$[C]$ are not used, the evaluation of C can be moved to the next higher level, i.e., L_{i-1}. The details of this procedure are described in the next algorithm. Because it is expensive to save all the input relations for relational operations, only procedural methods are considered in the next algorithm.

Algorithm 3—Global Optimization

Input: A connection graph for a type-GENERAL query.
Output: An optimized type-GENERAL query.
Suppose the given query has n levels of nesting. For each level L_i, where i = from n to 1, do the following:

1) Derive $GEN[L_i]$ and GEN and USE for each operation in L_i.
2) For each procedural method M_j, search all USEs to check any variable in it is used or not. If all the variables in $GEN[M_j]$ are not used at the level, do the following:
   2.1) If any variable in $USE[M_j]$ is an iteration variable, go to the beginning of Step 2 and consider M_{j+1}.
   2.2) Save all the variables in $USE[M_j]$ into a temporary relation and move M_j into the next level L_{i-1} with proper connections between members of $USE[M_j]$ and $GEN[C_j]$.
3) Repeat Step 2 until no method can be moved.

□

Example 4 Consider the connection graph shown in Fig. 8. At level 3, we can derive the following:

$GEN[L_3] = \{ v_3, v_2 \}$
$USE[overlap] = \{ v_3, v_2 \}$
$GEN[overlap] = \{ INTSEC \}$
$USE[get\_area] = \{ INTSEC \}$
$GEN[get\_area] = \{ S \}$.

With a simple search, we can find that the variable in $GEN[get\_area]$ (i.e., S) is not used by any other computations in L_3. Because $USE[get\_area]$ (i.e., $INTSEC$) is not an iteration variable (i.e., v_3 and v_2), the method get_area can be moved into level 2 by saving $INTSEC$ into a temporary relation. Without get_area, the next iteration of Step 2 in Algorithm 3 will find that $GEN[intersect]$ (i.e., $INTSEC$) is not used, and therefore it can be moved up again. In this example, the method $intersect$ cannot be moved because its USE includes some iteration variables.

After applying Algorithm 3 to levels 2 and 1, the connection graph is changed as shown in Fig. 9 and the corresponding query program is as follows:

```
(forall $v_1$ in obstacle
  (forall $v_2$ in rectangle
    (forall $v_3$ in rectangle
      (cond ((setq INTSEC overlap $v_2$ $v_3$))
        (retrieve $v_3$.AREA INTSEC into Templ)))
      (cond (ieq $v_2$.AREA (imax Templ "" AREA)))
        (retrieve $v_2$.TID Templ.INTSEC into Temp2))
    (cond ((and (included Temp2.TID $v_1$)
      (and ((setq S (get_area Temp2.INTSEC)
        (igt S 100))))
      (retrieve Temp2.TID)))
```

□

In the previous example, the number of calls to the method get_area is reduced because the temporary relations have been operated by a set of selections before the method get_area is evaluated in the modified program. Once a type-GENERAL query is optimized by the above algorithm, each level of nesting can be optimized further by applying an appropriate technique discussed earlier. In the next subsection, we describe the general procedure to evaluate a type-GENERAL query.

X. Procedure of Evaluating a TYPE-GENERAL Query

Given a type-GENERAL query, it can be globally optimized first as discussed in the previous subsection. Further optimiza-

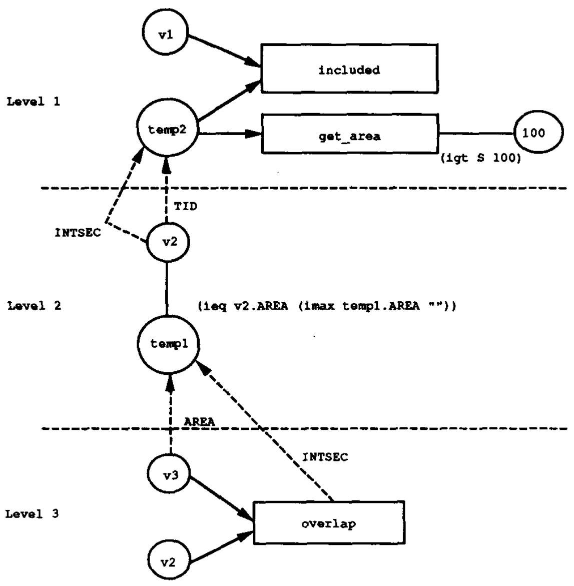


Fig. 9. An optimized connection graph.

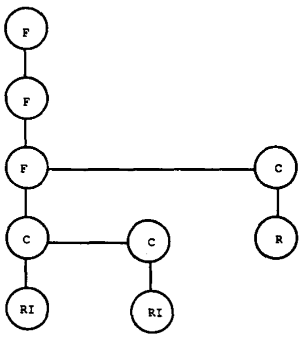


Fig. 10. A query graph.

tion and evaluation is based on the optimization techniques for basic patterns. One way to represent the structure of a given query is to use a query graph.

A query graph is a binary tree, where each node denotes a statement. Each node can have up to two children: a left child and a right child, where the left child represents the first statement among its nested statements and the right child represents the remaining statements in the same level of nesting. For example, the query graph for the transformed query in the previous subsection is shown in Fig. 10. In this query graph, nodes are labeled by F, I, R, and RI to represent statements of types *forall*, *cond*, *retrieve* and *retrieve_into*, respectively.

The optimization techniques discussed in this paper can be divided into two types:

1) *Type A:* Techniques that transform a given query syntactically. Techniques of this type are those developed for type-NESTED_FORALL queries (i.e., detecting loop-invariants and transforming them into canonical queries).
2) *Type B:* Techniques that speed up the evaluation of a given query. Techniques of this type are those developed for canonical queries and for types GENERAL_COND, MULTIPLE_COND and ASSORTED.

Techniques of type A may change the structure of a given query or move some computations into upper levels. For this reason, techniques of type A should be applied prior to those of type B. Also, they should be applied in a bottom-up fashion (i.e., from the lower levels to the upper levels) so that any change in the structure of the query can be propagated to the upper levels.

Techniques of type B can include some modifications to the statements, but those modifications are internal and do not affect the overall structure of a given query. Techniques of type B are applied in a top-down fashion.

In summary, a general procedure for processing a type-GENERAL query is described in the following algorithm.

Algorithm 4—Optimization of a Type-General Query

Input: A query graph for A type-GENERAL query.
Output: An optimized evaluation procedure.

1) Apply Algorithm 3 to the query graph for global optimization.
2) Search the query graph in the post-order (i.e., search the graph recursively in the order of left child, right child, and parent) and identify basic patterns in it.
3) Optimize all type-NESTED_FORALL in a bottom-up fashion.
4) Evaluate the optimized query in a top-down fashion. For each level, apply appropriate optimization techniques. □

XI. Conclusion

In this paper, several approaches have been described to optimize queries written in an object-oriented database programming language. The expressive power of this language makes it very complicated to optimize, as several basic patterns can be combined in an arbitrary fashion. Our approach starts with an extended query decomposition algorithm (Algorithm 1) that is capable of handling procedural methods in relational queries, in which the major concern has been placed on deferring the evaluation of such methods. This algorithm has been further extended (Algorithm 2) to consider the sharing of common subexpressions when multiple conditions are present in a nested *forall* loop based on the ideas presented in [6]. To optimize the evaluation of more general queries, a global query transformation approach that divides a query into multiple nesting levels is provided (Algorithm 4). A query, after transformed, is then evaluated inside out, with the consideration of loop invariants and data dependencies (Algorithm 3), according to the techniques developed for the basic query patterns. Compared with the existing work, it is our feeling that the scope of the queries discussed in this paper is much wider than that of conventional queries as complex control structures and procedures are allowed to be included in a query (in fact, a query program). The problem of optimal evaluation is therefore more complicated than the traditional query optimization problem for nested or multiple queries.

In this paper, we have assumed that each class is implemented as a relation. This implies that complex objects cannot be retrieved without multiple searches, which could be expensive compared to other storage structures such as variable-length records. In addition, we have not particularly differentiated programming objects from database objects; more research need to be done as the techniques that can be applied to evaluate these two types of objects may be different.

Appendix I

The OASIS Schema Definition Language

In OASIS, a statement to define a new class (of objects) is presented as
(defclass <class-id> (<attribute-id> <class-id> [width])....
(<attribute-id> <class-id> [width]) (key <attribute-id>))
where

a) Each argument of the statement defines an attribute, where *<attribute-id>* designates the name of the attribute, *<class-id>* designates the domain of the attribute, and *width* designates the width of the attribute (if it is primitive). If an attribute is not primitive, then the key of some object of the domain class would be used as the value of the attribute.
b) The specification (*key <attribute-id>*) assigns one of the attributes to be the key attribute of the class. Consequently, the values of the key attribute of the objects in a class need to be distinctive.

Other statements related to schema definitions are summarized as follows:

1) A statement to define a new method is presented as:
(defmethod (<method-id> <class-id>) ((<argument-id> <class-id>)...(<argument-id> <class-id>)) body
where
   a) *<method-id>* designates the name of the method to be defined.
   b) The first *<class-id>* after *<method-id>* is the class to which the returned object of the method belongs.
   c) Each argument of the statement defines an argument, where *<argument-id>* designates the name of the argument and *<class-id>* designates the domain of the argument.
   d) An argument or a variable in a method declaration could be a pointer; in this case an asterisk "*" has to precede it.
   e) body designates the body of the method, see Section II-C.
2) The following are the statements to remove a class and a method, respectively:
(delclass <class-id>)
(delmethod <method-id>).
3) The statement to define a class to be a subclass of another class is:
(defsubclass <class-id> <class-id>(<attribute-id> <class-id> [width]).... (<attribute-id><class-id> [width]) (key <attribute-id>))
where the first *<class-id>* designates the name of the subclass and the second *<class-id>* designates the name of the superclass. The attribute specifiers identify the additional attributes for the objects in the subclass.

Currently, the following methods are built in and provided by the system. We summarize them in the format to be called in an OASIS-LISP function or method. They may as well be called in an OASIS-C function or method in the following form:
method(argument, . . . ,argument)

Logical Methods

a) *(igta b), (ige a b), (ilta b), (ile a b), (ieq a b), and (ine a b)*: which is true if integer a *is greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal* to integer b, respectively.
b) *(fgt a b), (fge a b), (flt a b), (fle a b), (feq a b), and (fne a b):* which is true if float number a is greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to float number b, respectively.
c) *(seq a b), (sne a b):* which is true if string a is *equal to* and *not equal* to string b, respectively.
d) *(leq a b), (lne a b):* which is true if list a is *equal to* and *not equal to* list b, respectively.

General Methods

a) *(ias a b)* assigns the value of integer a to integer b.
b) *(fas a b)* assigns the value of float a to float b.
c) *(sas a b)* assigns the value of string a to string b.
d) *(iadd a b), (isub a b), (imult a b),* and *(idiv a b)* returns the result of adding, subtracting, multiplying, and dividing two integers a and b, respectively.
e) *(faad a b), (fsub a b), (finult a b),* and *(fdiv a b)* returns the result of adding, subtracting, multiplying, and dividing two float numbers a and b, respectively.
f) *(cons a b)* constructs a list whose car is a and whose cdr is b.
g) *(las a b)* assigns the value of list a to list b.
h) *(car a), (cdr a), (list a b), eval(a)* and other standard LISP function.

Aggregation Methods

a) *(imax r s a)* returns the maximal value of attribute a (whose domain is integer) of the objects in the set s of class r. If the method is intended to be performed on the basic set (see Section II-C), then s is " " (the empty string).
b) *(imin r s a)* returns the minimal value of attribute a (whose domain is integer) of the objects in the set s of class r. If the method is intended to be performed on the basic set, then s is " " (the empty string).
c) *(fmax r s a)* returns the maximal value of attribute a (whose domain is real) of the objects in the set s of class r. If the

method is intended to be performed on the basic set, then s is " " (the empry string).

d) *(fmin r s a)* returns the minimal value of attribute a (whose domain is float) of the objects in the set s of class r. If the method is intended to be performed on the basic set, then s is " " (the empty string).

e) *(smin r s a)* returns the maximal value of attribute a (whose domain is string) of the objects in the sets of class r. If the method is intended to be performed on the basic set, then s is " " (the empty string).

f) *(smin r s a)* returns the minimal value of attribute a (whose domain is string) of the objects in the set s of class r. If the method is intended to be performed on the basic set, then s is " " (the empty string).

g) *(iavg r s a)* returns the average value of attribute a (whose domain is integer) of the objects in the set s of class r. If the method is intended to be performed on the basic set, then s is " " (the empty string).

h) *(favg r s a)* returns the average value of attribute a (whose domain is float) of the objects in the set s of class r. If the method is intended to be performed on the basic set, then s is " " (the empty string).

i) *(countu r s)* returns the number of unique objects in the set s of class r. If the method is intended to be performed on the basic set, then s is " " (the empty string).

j) *(sort a s b)* sorts a set s of class a according to its attribute b (in increasing order). If the method is intended to be performed on the basic set, then s is " " (the empty string).

k) *(removed a s)* removes any duplicated objects from the set s of class a. If the method is intended to be performed on the basic set, then s is " " (the empty string).

Appendix II

Programming in Oasis

Once a database schema has been defined, we can create sets of objects for each class; we can also pose queries (which are functions in OASIS) to the system to retrieve, append, delete or update the objects in the database. In the following, we shall assume that once a class has been defined, a basic set for that class is also created. Unless otherwise specified (see below), all database operations are per- formed on the objects in these basic sets. For simplicity, we shall only discuss the programming constructs in OASIS-LISP. Functions in OASIS-C could be declared in a general format as in OASIS-LISP with slight changes, except that currently type declarations are not allowed in an OASIS-C function when it is being interpreted.

A query to OASIS is presented as a function in the following format:

(defun (<function-id> <type-id>) ((<argument-id> <class-id>).... (<argument-id> <class-id>)) ((<variable-id> <class-id>)... (<variable-id> <class-id>)) body)

where *body* is a sequence of statements presented in the following form:

statement statement... statement.

In specifying the body of a function (or a method), we can access an attribute *<attribute-id>* of a variable *<variable-id>* by specifying *<variable-id>.<attribute-id>*.[2] If the attribute itself is another object, we can access any of its attributes (say, *<attribute-id'>*) by specifying *<variable- id>.<attribute-id>.<artribute-id>*. The same rule can be applied as long as an attribute (of an object) is an object. We call such objects as secondary variables. Note that if desirable, a user can specify an arbitrary *<set-id>* after a *<class-id>* (in this case a colon is required to separate the two identifiers). If a set identifier is specified, it designates a set of objects whose structures are defined by the corresponding class. Referring to the previous format, a statement is composed of the function name of the statement and its associated arguments, where an argunient could be a constant, an object variable, or a secondary variable. A statement can be one of the following (where a statement in the form of c), d), e), or f) is referred to as a basic statement):

a) *(deftype <type-id> (<attribute-id> <type-id> [width])....(<attribute-id> <type-id> [width]))*
which defines a new (nonpersistent) type of objects.

b) The statement to define a type to be a (nonpersistent) subtype of another (nonpersistent) type is
(defsubtype <type-id> <type-id> (<attribute-id> <type-id> [width]).... (<attribute-id><type-id> [width]) <attribute-id>))
where the first *<type-id>* designates the name of the subtype and the second *<type-id>* designates the name of the supertype. The attribute specifiers identify the additional attributes for the objects in the subclass.

c) *(retrieve argument argument [into <class-id>])*, which returns a list that is composed of the values of the arguments (in that sequence) or saves it into a class.

d) *(append <class-id> [<set-id>] <variable-id>)*, which appends the value of variable *<variable-id>* to the set *<set-id>* whose members are in the class *<class-id>*. If *<set-id>* is not specified, then the values of the variable are appended to the basic class itself.

e) *(delete <class-id> [<set-id>] <variable-id>)*, which deletes any object from the set *<set-id>* whose members are in the class *<class-id>* and whose value matches the value of variable *<variable-id>*. If *<set-id>* is not specified, then the object is deleted from the basic class itself.

f) *(replace <class-id> [<set-id>] <variable-id1> <variable-id2>)*, which replaces any object whose value matches the value of the first variable by the value of the second variable in the set *<set-id>* of the class *<class-id>*. If *<set-id>* is not specified, then the object is replaced from the basic class itself.

g) Any method or function with its arguments.

h) *(forall <variable-id> in <class/type-id>[:<setid>] statement...statement)*

i) *(cond (qualification statement....statement)*
(qualification statement....statement)

...

(qualification statement....statement)

[2]In case *<variable-id>* is a pointer, then this notation is replaced by *<variable-id>*→*<attribute-id>*. The same rule applies to the following discussion.

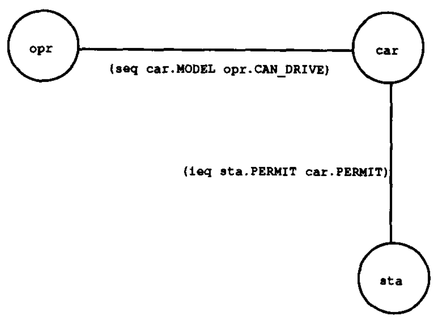


Fig. 11. A connection graph.

j) *(while qualification statement....statement))*
Here, a qualification is defined recursively as follows.
   a) A logical method/function with its arguments (where each argument could be a constant, a variable, or a secondary variable) is a *qualification.*
   b) If a is a qualification, then (a) is a qualification.
   c) If a and b are qualifications then *(and a b)*, *(or a b)*, *(not a)* and *(not b)* are qualifications.

k) *(return <variable-id>)*
which returns the value of *<variable-id>*.

l) *((var (<variable-id> <class/type-id>)... (<variable-id> <class/type-id>)) statement...statement)*
which declares a set of temporary variables before the statements are executed; the scope of the temporary variables is striclly local.

m) *(!statement)*
where the "!" sign prevents the statement from being transformed.

n) *(break)*
which breaks from the nearest nested forall loops.

Appendix III

Query Decomposiiion Algorithm

In [33], the query decomposition algorithm is explained using a connection graph (see Fig. 11). Suppose that we have the query

$$\pi_r(\sigma_F(R_1 \times \cdots \times R_k))$$

where σ stands for *selection* and π for *projection*. The condition F is in the conjunctive form C where a conjunct C_i is one of the following:

1) $R_1.A$ *comparison_op constant,*
2) $R_i.A$ *comparison_op* $R_j.B$.

Here, R_i and R_j, are relations and A and B are attributes. The *conparison_op* could be $=, \geq, \leq, >, <$ or $\neq$.

A connection graph G can be consttucted by creating a node for each relation R_i, $1 \leq i \leq k$ and a node for each occurrence of a constant in any of the C_j's, $I \leq j \leq k$. Two occurrences of the same constant are given two distinct nodes. Corresponding to each conjunct c_j, an edge can be created which con- nects two nodes that are related by the conjunct, and it is labelled by C_j. In a connection graph, there are two kinds of edges: join edges and selection edges. While a join edge connect two nodes correspond- ing to two relations that are to be joined, a selection edge connects two nodes which correspond to a relation and a constant. Clearly a selection edge and a join edge correspond to a conjunct of type 1) and that of type 2), respectively.

For example, the connection graph for the following query is shown in Fig. 12.

(forall car in car
  (forall sta in station
    (forall opr in operator
      (cond (and (ieq car.PERMIT sta.PERMIT)
          (seq car.MODEL opr.CAN_DRIVE))
        (retrieve all)))))

In the connection graph, we call an edge in the form $(A = B)$ a *simple* edge. A node is defined to be *small* when it corresponds to the result of an *instantiation* operation or to the result of a *dissection* operation, where these two types of operations are defined as follows:

a) An instantiation executes a selection on a relation. After an instantiation has been performed on two nodes (one corresponds to a constant and the other corresponds to a relation), the connecting edge and the node corresponding to constant is deleted and the other node becomes a small node.

b) A dissection on a relation node n runs through the tuples t in the relation corresponding to the node and replaces the node with a set of constant nodes with the attribute values of each t. Now the constant nodes and the edges can be eliminated by consequent selection operators. When more than one edge is incident on n, for each tuple t, the Cartesian product of the relations derived for each edge is the result. Subsequenlly, the union of the results with respect to each constant node is the answer to the original connection graph.
With the previous definitions, the query decomposition algorithm can be summarized as follows [33]:

1) Do all installations.
2) Select a node and perform dissection in the following preference.
   1.1) Among the nodes whose incident edges are all simple.
      1.1.1) A node representing a small relation.
      1.1.2) A node whose elimination will disconnect the graph.
      1.1.3) Any remaining node.
   1.2) Among the other nodes.
      1.1.1) A node representing a small relation.
      1.1.2) A node whose elimination will disconnect the graph.
      1.1.3) Any remaining node.

REFERENCES

[1] M. Astrahan, *et al.*, "System R: Relational approach to database management," *ACM Trans. Database Syst.,* vol. 1, no. 2, pp. 97–137, June 1976.

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[3] M. Atkinson and O. Buneman, "Types and persistence in database programming languages," *ACM Computing Surveys,* vol. 19, no. 2, pp. 105–190, June 1987.

[4] M. Carey, D. DeWitt, J. Richardson, and E. Sheikta, "Object and file management in the EXODUS extensible d atabase system," in *Proc. 12th Int. Conf. VLDB,* Kyoto, Japan, Aug. 1986.

[5] U. Chakravarthy and J. Minker, "Processing multiple queries in database systems," *Database Eng,,* vol. 1, 1982.

[6] ——, "Multiple query processing in deductive databases using query graphs," in *Proc. 12th Int. Conf. VLDB,* Kyoto, Aug. 1986, pp. 384–391.

[7] G. Copeland and D. Maier, "Making Smalltalk a database system," in *Proc. SIGMOD,* ACM, New York, 1984, pp. 316–325.

[8] D. Fish, *et al.*, "Iris: An object-oriented database management system," *ACM Trans. Office Inform. Syst.,* vol. 5, pp. 48–69, Jan. 1987.

[9] H. Gallaire, J. Minker, and J. Nicolas, "Logic and databases: A deductive approach," *ACM Computing Surveys,* vol. 16, no. 2, pp. 153–185, June 1984.

[10] R. Ganski and H. Wong, "Optimization of nested SQL queries revisited," in *Proc. ACM SIGMOD,* May 1987, pp. 23–33.

[11] J. Grant and J. Minker, "On optimizing the evaluation of a set of expressions," Tech. Rep. TR-916, Univ. Maryland, College Park, MD, July 1980.

[12] ——, "Optimization in deductive and conventional relational database systems," in *Advances in Database Theory, Vol. 1,* H. Gallaire, J. Minker and J. Nicolas, Eds. New York: Plenum, 1981.

[13] M. Jarke and J. Koch "Query optimization in database systems," *ACM Computing Survey,* vol. 16, June 1984.

[14] M. Jarke, "Common subexpression isolation in multiple query optimization," in *Query Processing in Database Systems,* W. Kim, D.Reiner and D. Batory, Eds. New York: Springer-Verlag, 1984.

[15] W. Kim, "On optimizing an SQL-like nested query," *ACM Trans. Database Syst.,* vol. 7, no. 3, pp. 443–469, Sept. 1982.

[16] ——, "Global optimization of relational queries: A first step," in *Query Processing in Database Systems,* W. Kim, D. Reiner, and D. Batory, Eds. New York: Springer-Verlag, 1984.

[17] D. Kuck *et al.*, "Dependence graphs and compiler optimizations," in *Proc. 8th ACM Symp. Principles of Programming Languages,* 1981, pp. 207–218.

[18] R. MacGregor, "Ariel—A semantic front-end to relational DBMS's," *Proc. VLDB,* 1985, pp. 305–315.

[19] G. Marque-Pucheu, J. Martin-Gallausiaux, and G. Jomier, "Interfacing Prolog and relational data base management systems," in *New Applications of Data Bases,* G. Gardarin and E. Gelenbe, Eds., Academic, 1984.

[20] D. Maier and J. Stein, "Development of an object-oriented DBMS," in *OOPSLA '86 Proc.,* 1986, pp. 472–481.

[21] O'Brien, P., Bullis, B., and Schaffert, C., "Persistent and shared objects in trellis/owl," in *Proc. 1986 IEEE Workshop on Object-oriented Database Syst.*

[22] J. Park and A. Segev, "Using common subexpressions to optimize multiple queries," in *Proc. 4th Int. Conf. Data Eng.,* Feb. 1988

[23] J. Park, T. Teorey, and S. Lafortune, "A knowledge-based approach to multiple query processing," *Data and Knowledge Engineering,* Vol. 3. New York: North-Holland, 1989.

[24] L. Rowe and K. Shoens, "Data abstraction, views and updates in RIGEL," in *Proc. ACM SIGMOD Int. Conf. Management Data,* ACM, New York, pp. 71–81.

[25] L. Rowe and M. Stonebraker, "The POSTGRES data model," in *Proc. VLDB,* 1987, pp. 83–96.

[26] J. Schmidt, "Some high level language constructs for data of type relation," *ACM Trans. Database Syst.,* vol. 2, no. 3, pp. 247–261, Sept. 1977.

[27] T. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.,* vol. 13, no. 1., Mar. 1988.

[28] P.C-Y. Sheu, R. L. Kashyap, and S. Yoo, "Query optimization in object-oriented knowledge bases," *Int. J. Data and Knowledge Eng.,* Mar. 1989.

[29] M. Stonebraker, E. Anderson, E. Hanson and B. Rubenstein, "QUEL as a data type," *ACM SIGMOD Int. Conf. Management Data,* Boston, MA, June 1984.

[30] M. Stonebraker, E. Hanson, and C. Hong, "The design of the POSTGRES rules system," in *Proc. Data Eng.,* 1987, pp. 365–374.

[31] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM Trans. Database Syst.,* vol. 1, no. 3, pp. 189–222, Sept. 1976.

[32] S. Tsur and C. Zaniolo, "An implementation of GEM—Supporting a semantic data model on a relational back-end," in *Proc. SIGMOD,* 1984, pp. 286–295.

[33] J. Ullman, *Principles of Database Systems,.* Reading, MA: Computer Science, 1982.

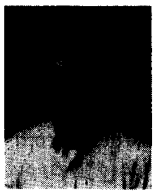
[34] M. Wolfe and U. Banerjee, "Data dependence and its application to parallel processing," *Int. J. Parallel Programming,* vol. 16, no. 2, pp. 137–178, 1987.

[35] E. Wong and K. Youssefi, "Decomposition—A strategy for query processing," *ACM Trans. Database Syst.,* vol. 1, no. 3, Sept., 1976, pp. 223–241.

Sang Bong Yoo received the B.S. degree in control and instrumentation engineering in 1982 from Seoul National University, Seoul, Korea, the M.S. degree in electrical engineering in 1986 from the University of Arizona, Tucson, AZ, and the Ph.D. degree in electrical engineering in 1990 from the Purdue University, West Lafayette, IN.

He is currently an Assistant Professor of the Department of Industrial Automation at Inha University, Republic of Korea. His research interests include knowledge and database management systems, software engineering, parallel processing, and artificial intelligence.

He is a member of Eta Kappa Nu.



Phillip C.-Y. Sheu received the M.S. and Ph.D. in electrical engineering and computer science from the University of California, Berkeley.

He is an Associate Professor at Rutgers University's Department of Electrical and Computer Engineering. Before joining Rutgers, he was an Assistant Professor at Purdue University. Before joining Purdue, he worked as a computer scientist for Systems Control Technology, Inc., Palo Alto, CA. He was also a product planning engineer for Advanced Micro Devices, Inc., Sunnyvale, CA, prior to that. His research interests include artificial intelligence, knowledge engineering, databases, CAD/CAM, computer architecture, and distributed systems.