Journal of Systems Integration, 5, 253–274 (1995)
© 1995 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

An Interface between Object-Oriented Databases and Relational Databases

CHIN-WAN CHUNG
Department of Information and Communication Engineering, Korea Advanced Institute of Science and Technology, Seoul 130–650, Korea

Abstract. Currently relational databases are widely used, while object-oriented databases are emerging as a new generation of database technology. This paper presents a methodology to provide effective sharing of information in object-oriented databases and relational databases. The object-oriented data model is selected as a common data model to build an integrated view of the diverse databases. An object-oriented query language is used as a standard query language. A method is developed to transform a relational data definition to an equivalent object-oriented data definition and to integrate local data definitions. Two distributed query processing methods are derived. One is for general queries and the other for a special class of restricted queries. Using the methods developed, it is possible to access distributed object-oriented databases and relational databases such that the locations and the structural differences of the databases are transparent to users.

Keywords: database, interface, distributed systems, translation, integration

1. Introduction

Accessibility to heterogeneous distributed databases enables the sharing of an organization's data. An integrated view of the heterogeneous databases is required to provide the transparency of the locations and the structure of the databases to users and application.

There has been a number of systems [7, 8, 15, 16, 19, 22, 23] which aimed at achieving an interoperability of traditional database systems including hierarchical, network, and relational database systems and file systems. Many of the above heterogeneous distributed DBMS's (HDDBMS's) used the relational data model as a common data model to provide the integrated view because data definition in hierarchical or network data model can be translated to equivalent relational data definitions. Based on the relational data model, an HDDBMS called DATAPLEX [8] had been developed, which interfaced relational databases and hierarchical databases. While the Multibase [16] used a functional data model which is non-relational, its purpose was to integrate traditional database systems.

Recently, the object-oriented (OO) database [2, 3, 4, 11, 13, 14,17] has emerged and it is becoming important. The advantages of the OO database include the adequacy for non-traditional application areas such as CAD, CAM, CIM and office automation as well as rapid application development using shared code stored in the database. It is expected that the amount of OO database usage will increase in the near future. On the other hand, the relational database is currently popular and many relational database systems are being developed.

It is necessary to interface OO databases and relational databases for the sharing of diverse databases in an organization. In other words, an HDDBMS needs to provide access to OO databases in addition to traditional databases. Since the interface among the traditional databases has been developed, we investigate how to interface an OO database with a relational database. Then, one way to interface an OO database with a network or hierarchical database is a transitive interface through a relational database.

The information captured by the relational data model can be represented using the OO data model, while some features in the OO database cannot be modeled using the relational framework. These features include multimedia data types and the encapsulation of the behavior of objects. In an OO database environment, designers of parts, for example, can store and execute engineering analysis programs and manufacturing constraint tests as methods, as well as store and view pictures of parts using the image data type. The relational database management system does not directly support such features. Although the relational data model can be extended, currently the OO data model is more natural and established to represent the features and structures of diverse databases. In addition to a powerful modeling capability, the OO paradigm provides features to tailor the operations of objects to perform specific functions. For instance, an object may trigger an execution of an existing application program in a remote database system or an object may further manipulate raw data retrieved from another location.

Therefore, we select the OO data model as a common data model to build an integrated view of distributed OO databases and relational databases. Through an integrated view in the common OO data model, the users can access data in both OO databases and relational databases. Additionally, all the features of local OO databases can also be used through the integrated view because the integrated view is in the OO data model. There were other systems [1, 5] which took object-oriented approaches to access heterogeneous databases. These systems used a multidatabase approach which is to provide multidatabase applications with interfaces to access multiple data sources without having an integrated schema. Our approach is a federated approach which is to build an integrated view through schema translations to support the transparency of the locations and the structures of diverse databases.

The effects of the above two approaches to the performance are considered to be similar in the sense that the effects are relatively small compared with the overall processing time of queries. In a federated approach, it is the creation of an integrated view that is a time consuming process due to syntactic and semantic heterogenities of local databases. However, once an integrated view is derived, the use of the view is fairly simple. The view is used to find locations of the data objects referenced in a query and the local names of the data objects, and optionally to enforce an access control. Since the size of an integrated view is small enough, these operations can be efficiently performed as cached local table lookups.

In a multidatabase approach, an integrated view is not built. Instead, each user has to know the details of local databases that the user wants to access. In addition, in order to access multiple databases, a multidatabase language has to be developed to formulate a users request that cannot be expressed in a local database language. A query formulated in a multidatabase language may contain entries that require table lookups. For example, a directory must be searched to find the location of a local database X that includes a data object Y when X.Y appears in a query [18]. Therefore, a multidatabase approach involves steps similar to those in using an integrated view in a federated approach when processing a query.

From the performance test [8, 9] of DATAPLEX, the effect of a federated approach to the performance can be analyzed. The test environment involved a computer network in a building and two different types of local database systems connected by the computer network. The query response time consists of (1) the processing time at the query origin, (2) the query and data transfer time through the network, and (3) the processing time at the target database locations. The time for using an integrated view is a small fraction of (1) which also includes the time for query parsing and decomposition as well as the time for the presentation of the query result. For a small test database, the combination of (1) and (2) were 20-30% of the query response time on the average. For a large production database, the sum of (1) and (2) were about 4% of the query response time, which implies that the time for using an integrated view is less than 1% of the query response time. From the above discussions, it is expected that the effect of a multidatabase approach to the performance is also very small but nonzero. Consequently, the effect of a federated approach and that of a multidatabase approach to the performance is similar.

While the basic construct of the OO data model has been established, the standardization of the model is not complete. As we discuss our approach, we will introduce OO data model features which are useful for interfacing OO databases with relational databases, but which have not been commonly used. They include representations of different types of relationships among objects, and some aspects of the query formulation for a navigation in the objects structure. On the other hand, although there are many OO database features, we will deal with only the features which are directly related to the issues addressed in this paper.

We develop a method to translate a relational data definition to an equivalent OO data definition and to integrate local OO data definitions. A major portion of the method is the modeling of the access path among classes and relations. In addition, a method is derived to process an OO query formulated based on the integrated view. Two distributed OO query decomposition algorithms are developed.

Section 2 presents the schema translation and the schema integration for building an integrated view using the OO data model. The processing of a distributed query is described in Section 3. An OO query syntax is defined. The processing of general queries and the processing of a special class of restricted queries are explained.

2. Schema Translation and Intergration

In this section, we present a method to synthesize an integrated view from local schemas. Since the scope of access may be different for different groups of users, there can be many integrated views. The core of the schema translation and integration is the representation of access paths among entities.

An access from instances of an entity to instances of another entity can be made when there is a relationship between instances of the two entities. Different data models represent the relationship among entities in different ways.

In the OO data model, there are basically two features to represent relationships among the classes: the class hierarchy (or is-a hierarchy) [20, 21] and the class composition hierarchy (or part-of hierarchy) [3,6]. The class hierarchy represents the relationship between the superclass and the subclass. The set of instances of a class is a subset of the set of instances of its superclass. However, the class hierarchy does not specify the relationship among instances of the two classes.

When the domain of an attribute in a class is a user-defined class, the class composition hierarchy (CCH) is made between the two classes. Such attribute is called a composite attribute. The value of a composite attribute of an object can be an identifier of an object or a set of identifiers of objects of the class which is the domain of the composite attribute. Therefore, the CCH represents the relationship between the instances (i.e., objects) of two classes. Depending on the numbers of objects of one class related to objects of the other class, the relationship between the classes may be one-to-one, one-to-many, or many-to-many. We use the CCH to model the access path.

The CCH is a well-established concept of the OO data model, however different types of the CCH are not yet completely standardized. Two types of the CCH are defined in the context of building an integrated view. Further, a method is developed to identify the types in the process of translating a relational data definition to an OO data definition. One type is the independent CCH and the other the dependent CCH. The independent CCH models the part-of relationship between the objects of two classes. The existence of the objects of two classes related by the independent CCH is autonomous. For example, even if the data of an assembly is no longer needed, the data of the component parts of the assembly often need to be kept.

In case of the dependent CCH, the dependent object does not have any meaning without the owner object. The dependent CCH is similar to the parent-child relationship of the hierarchical or network data model. The characteristics of the dependent CCH to model the access paths are as follows:

C1: An owner object must exist to create a dependent object.

C2: If an owner object is deleted, its dependent objects must be deleted.

It is clear from the above that, if a class has many owner classes,(a) all owner objects must exist to create a dependent object, and (b) if any of the owner objects is deleted, the dependent objects must be deleted.

In the relational data model, the relationship between two relations is represented by the joining attributes which are the common attributes of the two relations. The access paths among the relations can be found by identifying joining attributes.

In order to provide an integrated view of heterogeneous databases in terms of the OO data model, the relational data definition must be translated to an equivalent OO data definition. Besides, the definitions of the OO databases which have relationships to relational databases need to be modified to include access paths to there lational databases.

A relation $R(a_1, a_2, \ldots, a_m)$ will be translated to a class $C(a_1, a_2, \ldots, a_m)$ where a_i is the ith attribute. This notation is to show the correspondence between the source and the target of a translation. The name of a relation and attributes may be changed in a translation. Since R is a relation in at least first normal form, C does not contain any composite attribute. Obviously, C does not include any method either. A class translated from a relation is not in a class hierarchy with any user-defined class in an OO database or classes translated from other relations. Initially, the translated class becomes a subclass of the system-defined root class (or any class which has the characteristics of the root class).

The sets of classes in local OO databases are linked by class hierarchies and CCH's. These class structures become a part of the integrated view. At this point, the integrated view consists of class structures from local OO databases and standalone classes from local relational databases. The integrated view is completed by adding CCH's to represent access paths.

Let $C(R_i)$ and $C(R_j)$ be the classes translated from relations R_i and R_j, respectively, and $C(C_i)$ and $C(C_j)$ the classes translated from classes C_i and C_j, respectively, in a local OO database. The possible access paths are between $C(R_i)$ and $C(R_j)$, between $C(R_i)$ and $C(C_j)$, and between $C(C_i)$ and $C(C_j)$.

Consider the access path between $C(R_i)$ and $C(R_j)$. The acce path in the relational database is via joining attributes. Suppose $R_i(\alpha, a_{i1}, a_{i2}, \ldots, a_{in})$ and $R_j(\alpha, a_{j1}, a_{j2}, \ldots, a_{jm})$ are two relations with the joining attribute α (it is trivial to generalize this to a set of joining attributes) and attributes a_{ix}, for $1 \leq x \leq n$, and a_{jy}, for $1 \leq y \leq m$. In creating a CCH between $C(R_i)$ and $C(R_j)$, we have to decide the direction and the type of the CCH. If the database is used by a single user or a small group of users, the users can specify the direction such that the starting class of the CCH is the class which is more frequently accessed and more often becomes the origin of the search path. However, in an environment with many users, which would be more common, rules to determine the direction have to be derived.

We use the semantics represented by the joining attribute a in relations R_i and R_j. Consider a situation where $R_i.\alpha$ is a primary key and $R_j.\alpha$ is a foreign key. In many occasions, $R_j.\alpha$ is a non-key, and R_j is a relation describing a relationship between an entity corresponding to R_i and another entity. In this case, since R_i is more frequently accessed than R_j, the direction of the CCH becomes from $C(R_i)$ to $C(R_j)$.

Although there are exceptions to the above strategy, it covers the majority of occasions. Therefore, if $R_i.\alpha$ is a key and $R_j.\alpha$ is a non-key, we make the direction of the CCH from $C(R_i)$ to $C(R_j)$. In case, both $R_i.\alpha$ and $R_j.\alpha$ are keys or both are non-keys, the direction of the CCH is arbitrary.

Suppose the direction of the CCH becomes from $C(R_i)$ to $C(R_j)$. This is accomplished by adding a composite attribute β to $C(R_i)$. The domain of β depends on the cardinality relationship between R_i and R_j. When $R_i.\alpha$ is a key (a primary key or an alternate key), it is easy to show that if $R_j.\alpha$ is a key, then the relationship between $R - i$ and R_j is one-to-one, otherwise the relationship is one-to-many. If both $R_i.\alpha$ and $R_j.\alpha$ are non-keys, the relationship is many-to-many.

When the direction of CCH is $C(R_i)$ to $C(R_j)$ (including the cases where this direction is taken arbitrarily), the cardinality relationship between R_i and R_j are as follows:

(a) If $R_i.\alpha$ is a key and $R_j.\alpha$ is a non-key, the relationship is one-to-many.

(b) If both $R_i.\alpha$ and $R_j.\alpha$ are keys, the relationship is one-to-one.

(c) If both $R_i.\alpha$ and $R_j.\alpha$ are non-keys, the relationship is many-to-many.

If the relationship is one-to-many or many-to-many, the domain of the composite attribute β in $C(R_i)$ *is the set of* $C(R_j)$. Consequently, the translation of R_i and R_j

results in two classes $C(R_i)(\alpha, a_{i1}, a_{i2}, \ldots, a_{in}, \beta$ (DOMAIN_IS SET_OF $C(R_j)$) and $C(R_j)(\alpha, a_{j1}, a_{j2}, \ldots, a_{jm})$, where the domains of attributes other than the composite attribute β are not specified for simplicity. If the relationship between R_i and R_j is one-to-one, the domain of β is $C(R_j)$. In this case, two translated classes are $C(R_i)(\alpha, a_{i1}, a_{i2}, \ldots, a_{in}$, β (DOMAIN_IS $C(R_j)$) and $C(R_j)(\alpha, a_{j1}, a_{j2}, \ldots, a_{jm})$.

The type of a CCH between two classes in a local OO database can be specified during the database design to capture the semantic relationship of the two classes. However, the semantic relationship among relations in the relational database is represented in terms of attributes. Therefore, we have to use the semantics captured in attributes to decide the type of a CCH created in the process of schema translation and integration.

The type of the CCH between $C(R_i)$ and $C(R_j)$ is determined by two integrity constraints [12] of the relational database. The entity integrity is that no component of a primary key value may be null, whereas the referential integrity is that the value of a foreign key must be either null or equal to a value of the primary key. The following theorem provides the conditions to determine the type of a CCH using the two integrity constraints.

THEOREM 1 *The relationship between $C(R_i)$ and $C(R_j)$ is dependent iff $R_i.\alpha$ is the primary key and $R_j.\alpha$ is a component of the primary key of R_j.*

Proof: ($\leftarrow$) Since $R_i.\alpha$ is the primary key, $R_j.\alpha$ is a foreign key. By the referential integrity, the value of $R_j.\alpha$ is either null or the same as a value of $R_i.\alpha$. Since $R_j.\alpha$ is a component of the primary key of R_j, from the entity integrity, the value of $R_j.\alpha$ is equal to a value of $R_i.\alpha$. Therefore, the characteristics of the dependent CCH, C1 and C2 are satisfied.

($\rightarrow$) Suppose $R_i.\alpha$ is not the primary key or $R_j.\alpha$ is not a component of the primary key of R_j. (i) If $R_i.\alpha$ is not the primary key, $R_j.\alpha$ may or may not be a foreign key. If $R_j.\alpha$ is a foreign key, its corresponding primary key is $R_k.\alpha$ for $k \neq i$. (ii) If $R_j.\alpha$ is not a component of the primary key of R_j, the value of $R_j.\alpha$ may be null. For both (i) and (ii), there can be the value of $R_j.\alpha$ which is not in the set of values of $R_i.\alpha$. This implies that the characteristics C1 and C2 are not satisfied.

In summary, there are six different cases excluding the symmetric cases, and the creation of a CCH for each case is shown in Table 1. ■

When a global query references both a relational database and an OOdatabase, a translator must generate a relational query to access the relational database. The relational query retrieves the values of attributes from the relational database. Therefore, the access path between $C(R_i)$ and $C(C_j)$ exists only if there is a common attribute in R_i and C_j. Thus, the access path between $C(R_i)$ and $C(C_j)$ can be handled in the same way as the modeling of the access path between $C(R_i)$ and $C(R_j)$.

Consider an access path between $C(C_i)$ and $C(C_j)$, where C_i and C_j are classes in different local OO databases. A composite attribute will be added to $C(C_i)$ or $C(C_j)$ to create a CCH. This type of access path is similar to the access path in a local OO database. However, there is one important difference in terms of actual data storage. The local OO database is populated according to the local OO data definition. The values of composite

Table 1. The Creation of CCH's for Six Different Cases

Case	$R_i.\alpha$	$R_j.\alpha$	CCH Direction	CCH Type	Domain of β
1	Primary key	Alternate key	Arbitrary	Independent	$C(R_j)$
2	Primary key	Non-key & part of primary key	$C(R_i)$ to $C(R_j)$	Dependent	SET_OF $C(R_j)$
3	Primary key	Non-key & not part of primary key	$C(R_i)$ to $C(R_j)$	Independent	SET_OF $C(R_j)$
4	Alternate key	Alternate key	Arbitrary	Independent	$C(R_i)$ or $C(R_j)$
5	Alternate key	Non-key	$C(R_i) to C(R_j)$	Independent	SET_OF $C(R_j)$
6	Non-key	Non-key	Arbitrary	Independent	SET_OF $C(R_i)$ or SET_OF $C(R_j)$

attributes in a local OO data definition are stored in the local OO database. On the other hand, the composite attributes added in the global view merely specify the access paths.

It is not desirable to identify, store and maintain the values of the added composite attribute. In particular, these values must be stored separately from the local OO database so that the values are invisible to users of the local database. Suppose there is a common attribute between the two classes. An effective way to handle the values of an added composite attribute is to compute the values when it is needed using the values of the common attribute. For this reason, we only consider the access path between $C(C_i)$ and $C(C_j)$ when there is a common attribute in C_i and C_j.

Therefore, the six cases for creating CCH can be applied to model all possible types of access paths: between relations, between a relation and a class, and between classes. The following example shows the translation of local data definitions incorporating the creation of access paths. The result is an integrated view.

Example 1. There is a relational database at Location 1 consisting of three relations given below.

PART (P#, P_NAME, MATERIAL, PRICE, DS#)
CUSTOMER (C#, C_NAME, LOCATION, GRADE, BUSINESS_TYPE)
SUPPLY (P#, C#, QUANTITY)

An OO database is at Location 2 with three classes and two CCH's.

DESIGN (DS#, DESIGNER, DS_DATE, MANUFAC_COST, DRG(DOMAIN_IS
        SET_OF DRAWING))
DRAWING (DR#, PICTURE (DOMAIN_IS IMAGE))
IMAGE ()

We assume that a design consists of a set of unique drawings, and therefore the two CCH's are dependent. The methods that may be in classes are irrelevant. Figure 1 shows

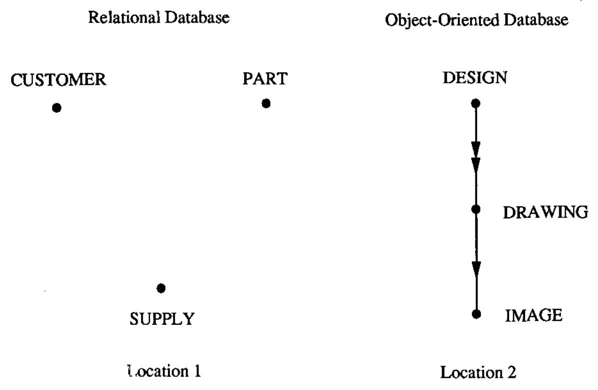

Figure 1. Two local database schemas.

the two local databases. A single-arrow on a CCH indicates the direction of one-to-one relationship, whereas a double-arrow one-to-many relationship.

PART.P# is the primary key of PART, and SUPPLY.P# is a non-key and a component of the primary key SUPPLY.P#, SUPPLY.C#. Therefore, the relationship between PART and SUPPLY is one-to-many and a dependent CCH is created from C(PART) to C(SUPPLY). Similarly, a dependent CCH representing one-to-many relationship is created from C(CUSTOMER) to C(SUPPLY).

There is a common attribute DS# between the class DESIGN and the relation PART. We assume that there is one design per part. In the class DESIGN, the values of DS# in different design objects are distinct. Although the object identifier of the OO database takes the role of the primary key of the relational database, we will informally call DESIGN.DS# the primary key in the sense that the value of DS# of a design object is unique. Since DESIGN.DS# is the primary key and PART.DS# is an alternate key, there is an independent CCH between C(DESIGN) and C(PART) with an arbitrary direction and the relationship between DESIGN and PART is one-to-one. We select the direction to be from C(DESIGN) to C(PART). The independent CCH implies that even when the design information is deleted, the part information should be kept. The six translated classes with added composite attributes are listed below.

```
C(DESIGN) (DS#, DESIGNER, DS_DATE, MANUFAC_COST, DRG
            (DOMAIN_IS SET_OF C(DRAWING)), PA(DOMAIN_IS
            C(PART)))
C(DRAWING)(DR#, PICTURE (DOMAIN_IS C(IMAGE)))
C(IMAGE) ()
C(PART) (P#, P_NAME, MATERIAL, PRICE, DS#, SY (DOMAIN_IS
         SET_OF C(SUPPLY)))
C(CUSTOMER) (C#, C_NAME, LOCATION, GRADE, BUSINESS_TYPE,
              SY (DOMAIN_IS SET_OF C(SUPPLY)))
C(SUPPLY) (P#, C#, QUANTITY)
```

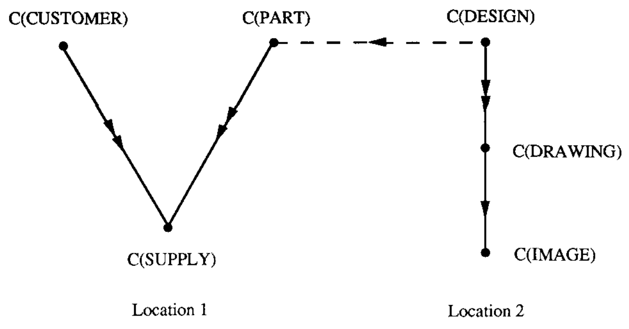

Figure 2. An integrated view in object-oriented data definition.

The integrated view in the OO data definition is depicted in Figure 2. The solid line specifies the dependent CCH and the dotted line the independent CCH.

3. Distributed Query Processing

Our objective is to provide an OO data model as a common data model and an OO query language as a standard query language. In this environment, a global query in an OO query language is formulated based on the integrated view discussed in Section 2. The global query must be decomposed into local queries by locations of referenced data. Then a local OO query is translated to an equivalent relational query when the referenced local database is a relational database. Finally, the local results are merged to form a single response to the original query.

3.1. Query Syntax

A major issue is that currently the OO query language is far from being standardized. There are many different types of OO query languages. Some of them are experimental and others used by commercial OODBMS's. These languages differ very much in functions and syntax. Therefore, rather than using a particular OO query language, we consider certain basic functions of the OO query language and concentrate on unique problems in processing those functions in a heterogeneous database environment.

The features of a relational query language are supported by basic operations selection, projection, equijoin and various aggregations. Especially the equijoin is used to access from one relation to another. Among the operations, the aggregations (e.g., sorting, averaging) are performed on raw data after they are retrieved by other operations. The OO query language should contain the features corresponding to the relational selection, projection, equijoin and aggregations. Since the aggregations in a distributed OO query should also be performed at the query origin after local results are retrieved, the part of the query containing the aggregations should be separated out and processed at one location. Therefore, we do not have to consider aggregations in distributed query processing.

The processing of selection and projection is similar in an OO database and a relational database. That is, there is one-to-one correspondence in translating an OO query to a relational query with respect to these operations. A major difference between the OO database and the relational database is that the operation corresponding to the relational equijoin is processed quite differently in the OO database. As discussed in Section 2, the relationship between the entities is modeled by the CCH in the OO database. Therefore, the OO query language feature corresponding to the relational equijoin is the navigation through the CCH. Since the inequality join is very seldom used, a join implies an equijoin for the rest of this paper.

The relational join is performed by comparing the values of joining attributes, whereas the navigation through the CCH is by using the object identifiers (OID's) stored under a composite attribute. To be as general as the relational query language, the OO query language should be able to represent the access path between two classes regardless of the direction of a CCH.

The navigation can be expressed using the composite attribute. The navigation through a CCH following the direction of the CCH is called a forward navigation. Given the starting class of a navigation, the forward navigation only requires a composite attribute of the class because the composite attribute uniquely determines its domain class. The domain class becomes the ending class of the navigation. However, the name of a composite attribute is not unique. Therefore, the backward navigation requires the ending class name in addition to its composite attribute whose domain is the starting class. This is illustrated in the coming example.

From the above discussions, the SQL-like syntax of the OO query containing basic features is as follows:

```
SELECT target attributes
FROM classes containing target attributes
WHERE qualification
```

The qualification contains selection terms, composite attributes, parentheses and Boolean operators AND, OR, NOT. This syntax is by no means intended to suggest an operational query language, but to show a method for processing queries involving basic operations.

For example, in the integrated schema in Figure 2, consider a request "Find the location of the grade A customer who receives more than 500 plastic parts." The corresponding query is as follows:

```
SELECT LOCATION
FROM C(CUSTOMER)
WHERE GRADE = 'A'
    AND (SY
            QUANTITY > 500
        AND (C(PART).SY
                MATERIAL = 'plastic'))
```

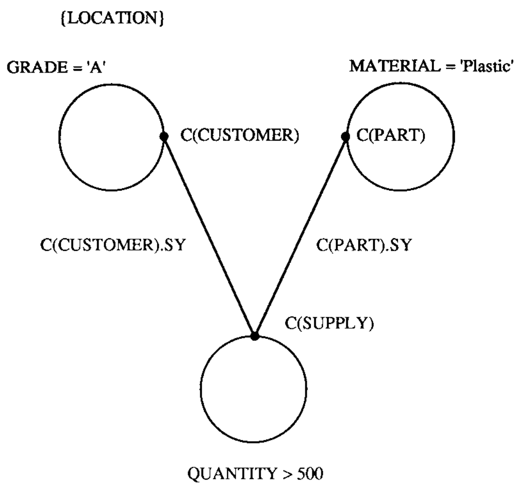

Figure 3. The query graph of an example query.

SY is the composite attribute in C(CUSTOMER) representing the forward navigation through the CCH from C(CUSTOMER) to C(SUPPLY). C(PART).SY represents the backward navigation through the CCH from C(PART) to C(SUPPLY).

This query can be transformed to the query graph shown in Figure 3 with classes as nodes and qualification terms as edges. The list of target attributes from a class is specified in braces. It is observed from the query graph that there can be many different orders in which the operations of the query are processed. An efficient ordering of operations is generated based on the data distribution and the availability of backward pointers for CCH's and it is the role of the query optimization.

The target attribute and the selection term in the OO query syntax correspond to those in SQL, respectively. The composite attribute in the OO query syntax corresponds to the join term in SQL. Therefore, the translation of an OO query to an SQL query is straightforward. We will focus on the distributed query decomposition.

3.2. *General Queries*

An OO query can be decomposed into a set of conjunctive queries by eliminating NOT's and OR's in the qualification. NOT's are eliminated by using DeMorgan's law and negating relational operators in selection terms. OR's are eliminated by transforming the qualification into a disjunctive normal form. Each conjunctive term becomes a subquery and the result of the original query is the union of the results of the conjunctive subqueries. For example, consider the following query for the schema in Figure 2:

```
SELECT DESIGNER
FROM C(DESIGN)
WHERE DS_DATE > 901231
    AND (PA
            MATERIAL = 'aluminum'
        OR (SY
                QUANTITY > 1000
            AND QUANTITY < 2000))
```

Since the domain of C(DESIGN).PA is C(PART) and the domain of C(PART).SY is C(SUPPLY), the qualification is equivalent to

```
DS_DATE > 901231 AND
(C(PART).MATERIAL = 'aluminum' OR
(C(SUPPLY).QUANTITY > 1000 AND C(SUPPLY).QUANTITY < 2000))
```

Let W = DS_DATE > 901231,

X = C(PART).MATERIAL = 'aluminum',
Y = C(SUPPLY).QUANTITY > 1000, and
Z = C(SUPPLY).QUANTITY < 2000.

Then the qualification is

W AND (X OR (Y AND Z)) = (W AND X) OR (W AND Y AND Z)

The two conjunctive subqueries are:

```
SELECT DESIGNER
FROM C(DESIGN)
WHERE DS_DATE > 901231
    AND (PA
        MATERIAL = 'aluminum')
SELECT DESIGNER
FROM C(DESIGN)
WHERE DS_DATE > 901231
    AND (PA
        (SY
            QUANTITY > 1000
        AND QUANTITY < 2000))
```

For a distributed conjunctive query, we present two query decomposition algorithms, ALGORITHM_D1 and ALGORITHM_D2. ALGORITHM_D1 is for general queries and ALGORITHM_D2 is for a restricted class of queries.

Suppose E_i and E_j are two entities at different locations. E_i (or E_j) may be a relation or a class. From the discussion in Section 2, the CCH between $C(E_i)$ and $C(E_j)$ in the integrated schema implies that there is a joining attribute a between $C(E_i)$ and $C(E_j)$. Therefore, the navigation between $C(E_i)$ and $C(E_j)$ is equivalent to the join $C(E_i).\alpha = C(E_j).\alpha$.

ALGORITHM_D1 requires a final merging of results of local queries using relational joins. The relational join can be processed by (a) using a relational DBMS in a distributed database system, (b) using an OODBMS in which a join operation has been implemented, or (c) implementing a join operation in a distributed query manager.

ALGORITHM_D1 is as follows:

ALGORITHM_D1.

1. Transform a query to a query graph Q with classes as nodes and qualification terms and composite attributes as edges. Specify the set of target attributes T_i for each location i.

2. Convert composite attributes between two nodes at different locations to join terms.

3. Delete edges connecting nodes at different locations and save the set E of labels on deleted edges. Let Q_i denote the subgraph of Q corresponding to the location i. Suppose the label of a deleted edge is $C_j.\alpha = C_k.\alpha$, $C_j \in Q_j$, and $C_k \in Q_k$. Update the set of target attributes T_j and T_k such that $T_j \leftarrow T_j \cup \{\alpha\}$ and $T_k \leftarrow T_k \cup \{\alpha\}$.

4. Transform each subgraph Q_i back to a local query and assign the result of the query to a temporary result $TEMP_i$.

5. Formulate the final merging query such that

   (a) the target list of the merging query is the same as that of the original query,

   (b) FROM list is the list of $TEMP_i$'s, and

   (c) the qualification is the conjunction of edge labels saved in E with a class name from the location i replaced by $TEMP_i$.

We give an example to show the steps of ALGORITHM_D1 below.

Example 2. The schema in Figure 2 is used with the locations of classes changed as follows:

Location 1: C(CUSTOMER)

Location 2: C(SUPPLY), C(PART)

Location 3: C(DESIGN), C(DRAWING), C(IMAGE)

Consider a user request: For designs of ceramic parts designed after 1985 such that the parts were supplied less than 500 units to customers in California, find the manufacturing costs of the designs and the types of business of the customers.

Let C be the class C(CUSTOMER), S be C(SUPPLY), P be C(PART), and D be C(DESIGN). The OO query formulation of the request is as follows:

```
SELECT MANUFAC_COST, BUSINESS_TYPE
FROM D,C
WHERE DS_DATE > 851231
     AND (PA
          MATERIAL = 'ceramic'
     AND (SY
          QUANTITY < 500
           AND (C.SY
               LOCATION = 'California')))
```

The steps of query decomposition is as follows:

1. The query graph Q is shown in Figure 4. ϕ denotes an empty set.

2. The joining attribute between C and S is $C\#$. Since C is at Location 1 and S is at Location 2, $C.SY$ is converted to the join term $C.C\# = S.C\#$. Similarly, $D.PA$ is converted to $P.DS\# = D.DS\#$.

3. The query graph Q is decomposed into subgraphs Q_1, Q_2 and Q_3 as depicted in Figure 5.

   $E = \{C.C\# = S.C\#, P.DS\# = D.DS\#\}$

   $T1 \leftarrow T1 \cup \{C\#\} = \{BUSINESS\_TYPE, C\#\}$

   $T2 \leftarrow T2 \cup \{C\#\} \cup \{DS\#\} = \{C\#, DS\#\}$

   $T3 \leftarrow T3 \cup \{DS\#\} = \{DS\#, MANUFAC\_COST\}$

4. Subgraphs are transformed to local queries.

```
TEMP1 = SELECT BUSINESS_TYPE, C#
        FROM C
        WHERE LOCATION = 'California'
TEMP2 = SELECT C#, DS#
        FROM P,S
        WHERE MATERIAL = 'ceramic'
              AND (SY
              QUANTITY < 500)
TEMP3 = SELECT DS#, MANUFAC_COST
        FROM D
        WHERE DS_DATE > 851231
```

5. For edge labels in E, C is replaced by $TEMP_1$, S and P by $TEMP_2$, and D by

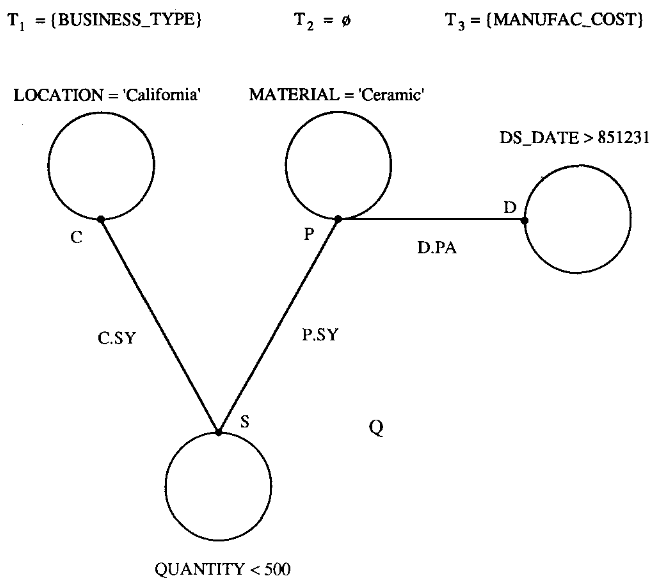

Figure 4. The query graph Q of a distributed query.

$TEMP_3$. The merging query is as below.

```
SELECT MANUFAC_COST.BUSINESS_TYPE
FROM $TEMP_1, TEMP_2, TEMP_3$
WHERE $TEMP_1.C\# = TEMP_2.C\#$
   AND $TEMP_2.DS\# = TEMP_3.DS\#$
```

3.3. *Restricted Queries*

ALGORITHM_D2 is for a query the target attributes of which are from one class and the query graph of which is acyclic. An important subset of such queries is the set of queries which retrieve identifiers of selected objects from a class so that messages can be sent to the objects. Since some OO query languages support only this type of queries, we present ALGORITHM_D2.

The OID-based retrieval is a unique characteristics of the OO database. In the relational database, the query selects data based on values of attributes and returns values. In other words, the access is always value-based. In the OO database, in addition to the value-based access, the query can return OID's and an access to a specific object is done by sending a message to the OID of the object. Therefore, the access can be value-based or OID-based. For example, to find the manufacturing cost of a design released on January 15, 1991 by John, the following query is issued to the OO database shown in Example 1:

```
VAR1 = SELECT OID
         FROM DESIGN
         WHERE DESIGNER = 'John'
             AND DS_DATE = 910115
```

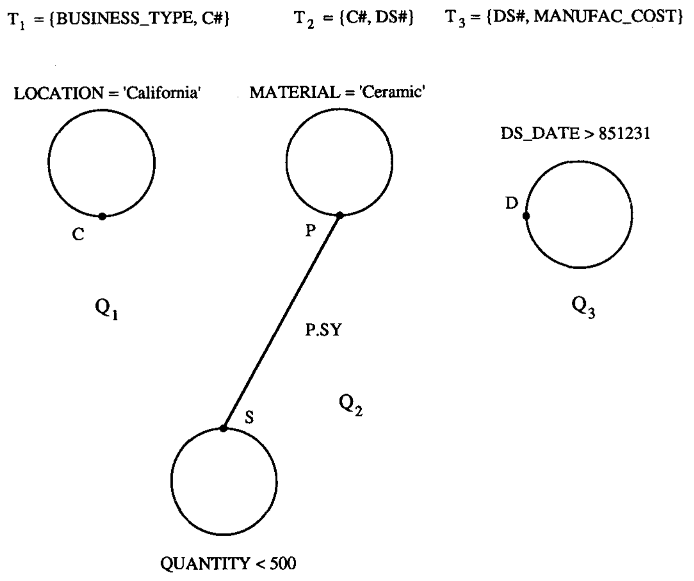

Figure 5. Subgraphs Q_1, Q_2 and Q_3.

The query returns the OID of a desired object in the variable $VAR1$. Then, a message can be sent to the OID in $VAR1$ as follows:
VAR2 = MANUFAC_COST (VAR1)

It is assumed that there is a method which retrieves the value of an attribute and that its name is identical to the name of the attribute. Now, the variable VAR2 contains the manufacturing cost of the design.

The OID-based retrieval is effective for large and complex objects because it is not practical to return a large number of such objects. In the above query formulation, the OID is treated as an attribute. If such a query is issued to a class corresponding to a relation, the primary key value of the relation is retrieved. The message to a relational database using the primary key value is translated to an SQL query with the message name as a target attribute and the primary key value in the qualification.

One implementation issue worth considering is an alternative way of handling selected objects and processing messages to the objects. For efficient message processing, it is desirable to store selected objects near to a user (or an application) especially if there is an OODBMS at the user location. This is similar to the storage management of an OODBMS which stores objects selected by a query in a buffer.

Suppose OODBMS's (or necessary object manipulation functions) are available at most of the locations to handle OO data manipulations against temporary objects. For a locations without an OODBMS, an OODBMS at a different location must be assigned. The OODBMS at the user location or the assigned OODBMS is called the designated OODBMS. Consider a query accessing a class $C(E)$ to retrieve the OID's of objects of $C(E)$. If (a) E is a class and the location of E is different from that of the designated OODBMS or (b) E is a relation, temporary objects corresponding to the selected objects or tuples are created in the private database by the designated OODBMS. A private database is accessed only by a user and it can be erased at the end of the user session. The OID's assigned to the temporary objects are returned and messages can be sent to these objects using the OID's.

ALGORITHM_D2 is based on the fact that the set of queries restricted for ALGORITHM_D2 can be completely processed by semijoins instead of intersite joins. Therefore, ALGORITHM_D2 does not require the relational join for merging local results. ALGORITHM_D2 is described below.

ALGORITHM_D2.

1. Transform a query to a query graph Q with classes as nodes and qualification terms and composite attributes as edges. Specify a set of target attributes T_i for each location i.

2. Convert the edges connecting nodes at different locations to dotted edges and convert composite attributes corresponding to the edges to joining attributes. Let Q_i denote the subgraph at location i.

3. Initialize a counter $j = 0$. Let Q_t denote the subgraph whose target attributes are the target attributes of the query. Process the following until the query graph becomes null:

   (a) Increment j to $j + 1$. Select the subgraph Q_f the farthest, in terms of the number of dotted edges, from Q_t. Break the tie in the distance arbitrarily.

   (b) Delete the dotted edge connected to Q_f. Suppose the label of the deleted edge between nodes C_m and C_n is α, where $C_n \in Q_f$. Update the set of target attributes T_f such that $T_f \leftarrow T_f \cup \{\alpha\}$. Add an edge to C_m with a label 'α IN LISTj', where IN is an operator testing the membership.

   (c) Convert Q_f to a local query and assign its result to LISTj. Delete Q_f.

4. LISTj is the result of the query.

The following example illustrates ALGORITHM_D2.

Example 3. The schema and the data distribution are the same as those in Example 2. We use the query in Example 2 except the changed target attribute as follows:

```
SELECT OID
FROM D
WHERE DS_DATE > 851231
    AND (PA
            MATERIAL = 'ceramic'
        AND (SY
            QUANTITY < 500
        AND (C.SY
                LOCATION = 'California')))
```

Each step of ALGORITHM_D2 is described below.

1. The query graph Q is the same as that shown in Figure 4 except that the sets of target attributes are: $T_1 = \emptyset$, $T_2 = \emptyset$ and $T_3 = \{\text{OID}\}$, where $\emptyset$ denotes an empty set.

2. The edges with labels $C.SY$ and $D.PA$ are converted to dotted edges with joining attributes. The updated query graph is shown in Figure 6.

3. Set $j = 0$. Q_3 is the subgraph with $T_3 = \{\text{OID}\}$.

   (a) Set $j = 1$. Q_1 is the farthest from Q_3.

   (b) Delete the dotted edge with the label C#.

   $$T_1 \leftarrow T_1 \cup \{C\#\} = \emptyset \cup \{C\#\} = \{C\#\}.$$

   Add an edge to S with a label 'C# IN LIST_1'.

   (c) Q_1 is transformed to a local query at Location 1 as follows:

```
LIST1 = SELECT C#
        FROM   C
        WHERE LOCATION = 'California'
```

   Delete Q_1. The updated query graph is shown in Figure 7.

   (a) Set $j = 2$. Q_2 is the farthest from Q_3.

   (b) Delete the dotted edge with the label DS#.

   $$T_2 \leftarrow T_2 \cup \{DS\#\} = \emptyset \cup \{DS\#\} = \{DS\#\}.$$

   Add an edge to D with a label 'DS# IN LIST_2'.

   (c) Q_2 is transformed to a local query at Location 2 as follows:

```
LIST2 = SELECT DS#
         FROM P
         WHERE MATERIAL = 'ceramic'
                  AND (SY
                           QUANTITY < 500
                       AND C# IN LIST1)
```

   Delete Q_2. The updated query graph is shown in Figure 8.

   (a) Set $j = 3$. Q_3 is the only remaining subgraph.

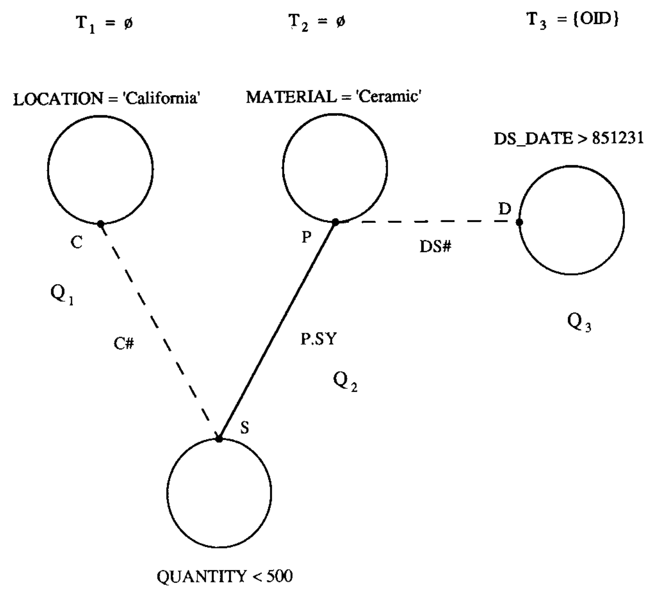

Figure 6. The updated query graph.

(b) There is no dotted edge connected to Q_3.

(c) Q_3 is transformed to a local query at Location 3 as follows:

```
LIST3 = SELECT OID
        FROM   D
        WHERE DS_DATE > 851231
            AND DS# IN LIST2
```

Delete Q_3.

4. Since the query graph is null, $LIST_3$ contains the query result.

In Example 3, the values of C# in $LIST_1$ need to be moved from Location 1 to Location 2. These values are used to reduce C(SUPPLY) locally in a local query transformed from Q_2. An intersite semijoin operation consists of such a data move and a local data reduction. The semijoin strategy can be very effective for reducing intersite data communication in processing a distributed query [9]. While the sequence of semijoins generated by ALGORITHM_D2 may not be optimal in terms of the amount of reduction in intersite data traffic, the sequence produces a minimal number of semijoins to process a distributed query.

It can be easily shown that the processings of the query in Example 3 using ALGORITHM_D1 and ALGORITHM_D2 produce the same result. If ALGORITHM_D1 is used to process the query, the local queries and the merging query are as follows:

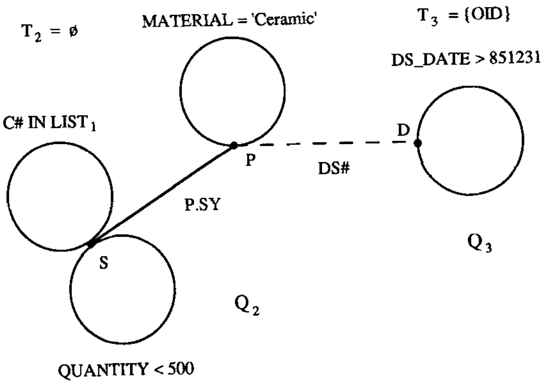

Figure 7. The updated query graph with two subgraphs.

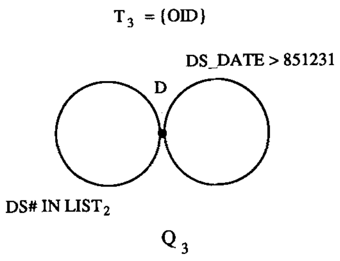

Figure 8. The updated query graph with one subgraph.

Local Queries:

```
TEMP₁ = SELECT C#
        FROM   C
        WHERE LOCATION = 'California'
TEMP₂ = SELECT C#,DS#
        FROM   P,S
        WHERE MATERIAL = 'ceramic'
              AND (SY
              QUANTITY < 500)
TEMP₃ = SELECT DS#,OID
        FROM   D
        WHERE DS_DATE > 851231
```

Merging Query:

```
SELECT OID
FROM TEMP1, TEMP2, TEMP3
WHERE TEMP1.C# = TEMP2.C#
    AND TEMP2.DS# = TEMP3.DS#
```

Let $E_1[\alpha = \alpha > E_2$ denote a semijoin from E_1 to E_2 on attribute a and E'_2 be the result of the semijoin. The decomposition by ALGORITHM_D2 is equivalent to a sequence of semijoins $\mathrm{TEMP}_1[\mathrm{C\#} = \mathrm{C\#} > \mathrm{TEMP}_2$, $\mathrm{TEMP}'_2[\mathrm{DS\#} = \mathrm{DS\#} > \mathrm{TEMP}_3$, with the query result $R_2 = \mathrm{TEMP}'_3\,[\mathrm{OID}]$. However, for a restricted query, a join can be replaced by a semijoin. Let $E_1[\alpha = \alpha]E_2$ denote a join between E_1 and E_2 on attribute α and R_1 denote the result of the merging query generated by ALGORITHM_D1. Then

$$\begin{aligned} R1 &= ((\mathrm{TEMP}_1[\mathrm{C\#} = \mathrm{C\#}]\mathrm{TEMP}_2)[\mathrm{DS\#} = \mathrm{DS\#}]\mathrm{TEMP}_3)[\mathrm{OID}] \\ &= ((\mathrm{TEMP}_1[\mathrm{C\#} = \mathrm{C\#} > \mathrm{TEMP}_2)[\mathrm{DS\#} = \mathrm{DS\#}]\mathrm{TEMP}_3)[\mathrm{OID}] \\ &= (\mathrm{TEMP}'_2[\mathrm{DS\#} = \mathrm{DS\#} > \mathrm{TEMP}_3)[\mathrm{OID}] \\ &= \mathrm{TEMP}'_3[\mathrm{OID}] \\ &= R_2 \end{aligned}$$

4. Conclusions

As the amount of information in relational databases and object-oriented databases grows in organizations, an effective sharing of these heterogeneous distributed databases becomes important. An integrated view of the databases and a standard query language are necessary to provide users with location and structure transparent access to the databases. The object-oriented data model was selected as a common data model because the object-oriented data model has advantages over the relational data model in representing the integrated view. An object-oriented query syntax which contains basic data access features was defined to explain the distributed query processing.

A method was developed to build an integrated view from local object-oriented data definitions and relational data definitions. The object-oriented data model and the relational data model represent the relationship between entities differently. Therefore, the key issue in building an integrated view is the modeling of access paths among classes and relations. The class composition hierarchy of the object-oriented data model was used to model the access paths.

Two query decomposition algorithms were developed to decompose a distributed query into a set of local queries. One is for general queries and the other is for restricted queries. A special subset of the restricted queries consists of queries which retrieve object identifiers of selected objects. This type of queries is unique because messages can be sent to the selected objects using their object identifiers. The decomposition of the restricted queries is based on the efficient processing of such queries using semijoin operations. We plan to implement the query decomposition algorithms, first for restricted queries and then for general queries.

5. Acknowledgements

The author wishes to thank anonymous reviewers for helpful comments and suggestions. The major portion of the research presented in this paper was conducted while the author was at General Motors Research Laboratories. This research was partially supported by the grant from '93 Multimedia Information Systems Platform Development Project of the Center for Artificial Intelligence Research.

References

1. R. Amed et al., "The Pegasus heterogeneous multidatabase system." *Computer* 24(12), pp. 19–27, Dec. 1991.
2. M. Atkinson et al., "The object-oriented database system manifesto," in *Proc. of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, pp. 40–57, 1989.
3. J. Banerjee et al., "Data model issues for object-oriented applications." *ACM Trans. Office Inform. Syst.* 5(1), pp. 3–26, Jan. 1987.
4. E. Bertino and L. Martino, "Object-oriented database management systems: concepts and issues," *Computer* 24(4), pp. 33–47, Apr. 1991.
5. E. Bertino et al., "Integration of heterogeneous applications through an object- oriented interface." *Information Systems* 14(5), pp. 407–420, 1989.
6. B. Bobrow and M. Stefik, *The LOOPS Manual*, Xerox PARC, Palo Alto, Calif., 1983.
7. O. Bukhres et al., "Interbase: an execution environment for heterogeneous software systems." *Computer* 26(8), pp. 57–69, Aug. 1993.
8. C. Chung, "DATAPLEX: An access to heterogeneous distributed databases." *Commun. ACM* 33(1), pp. 70–80, Jan. 1990. (*Corrigendum* 33(4), p. 459, Apr. 1990).
9. C. Chung and K. McCloskey, "A DATAPLEX prototype: an interface between two heterogeneous distributed databases," Research Report CS–545, General Motors Research Laboratories, Jul. 1987.
10. C. Chung and K. Irani, "An optimization of queries in distributed database systems," *J. Parall. Distrib. Comput.* 3(2), pp. 137–157, June 1986.
11. S. Cluet et al., "RELOOP, an algebra based query language for an object-oriented database management system." *Data & Knowledge Engineering* 5(4), pp. 333–351, Oct. 1990.
12. C. Date, *An Introduction to Database Systems*, 4th Ed., vol. 1. Addison-Wesley, Reading, Mass., 1986.
13. O. Deux et al., "The O2 system," *Commun. ACM* 34(10), pp. 34–48, Oct. 1991.
14. A. Hurson, S. Pakzad, and J. Cheng, "Object-oriented database management systems: evolution and performance issues," *Computer* 26(2), pp. 48–60, Feb. 1993.
15. IEEE, Special issue on distributed database systems, *Proc. of the IEEE* 75(5), May 1987.
16. T. Landers and R. Rosenberg, "An overview of Multibase." *Distributed Databases*, North-Holland, pp. 153–184, 1982.
17. C. Lecluse and R. Richard, "Modeling complex structure in object-oriented databases," in *Proc. of the 9th ACM Conference on Principles of Database Systems*, Philadelphia, Pennsylvania, pp. 360–368, 1989.
18. W. Litwin, L. Mark, and N. Roussopoulos, "Interoperability of multiple autonomous databases." *ACM Comput. Surv.* 22(3), pp. 267–293, Sep. 1990.
19. M. Rusinkiewicz, "Heterogeneous databases: Towards a federation of autonomous systems." in *Proc. of the Fall Joint Computer Conference*, Dallas, Tex., pp. 751–752, Oct. 1987.
20. J. Schmolze and T. Lipkis, "Classification in the KL-ONE knowledge representation system," in *Proc. of the 8th International Joint Conference on Artificial Intelligence*, Karlsruhe, Germany, pp. 330–332, 1983.
21. B. Shriver and B. Wegner (Ed.), *Research Directions in Object-Oriented Programming*. The MIT Press, Cambridge, Mass., 1987.
22. M. Templeton et al., "Mermaid: A front-end to distributed heterogeneous databases," in *Proc. of the IEEE* 75(5), pp. 695–708, May 1987.
23. G. Thomas et al., "Heterogeneous distributed database systems for production use." *ACM Comput. Surv.* 22(3), pp. 237–266, Sep. 1990.