# OBJECT QUERIES OVER RELATIONAL DATABASES : LANGUAGE, IMPLEMENTATION, AND APPLICATIONS *


Victor M. Markowitz and Arie Shoshani

Data Management Research and Development Group
Information and Computing Sciences Division
Lawrence Berkeley Laboratory, Berkeley, CA 94720



## Abstract

*Relational database management system (DBMS) definitions and queries for complex (e.g. scientific) database applications are typically large, hard to maintain, and involve terms that obscure the semantics of the application-specific data structures and operations. Since it has been proven in practice that using an object (e.g. Entity-Relationship) model greatly simplifies database definition, one may expect a similar effect on database querying by using an object query language. In this paper we describe such a query language called the Concise Object Query Language (COQL). COQL is unique in its conciseness, in its support of inheritance, and in the capabilities it provides for defining application-specific structures. We present the COQL to SQL translation, describe its implementation on top of a commercial relational DBMS, and discuss how COQL can be used for constructing application-specific views for scientific applications.*


## 1. Introduction

Managing data of scientific database applications requires, in addition to the traditional capabilities provided by database management systems (DBMSs), mechanisms for presenting, browsing, and querying the data in a way that users would easily understand. Several new technologies, such as object-oriented DBMSs and extended relational DBMSs, promise to have an important role in supporting such mechanisms. However, in practice users often prefer to use proven commercial DBMS technology, that is, relational DBMSs, and are content to perform specialized operations (such as a spatial search, statistical summaries) outside DBMSs, in their application programs.

Applications that are complex in terms of the number of different types of objects and their relationships, need some methods and tools for describing their data structures (schemas) in terms of objects. These schemas need to eventually be translated into database definitions of relational DBMSs. Accordingly, a data model that can be directly supported by relational DBMSs must be used. We have shown in [6] that one such model is the Extended Entity-Relationship (EER) model. We present in this paper an object query language based on the EER model and a translator from this query language into SQL. Together with a previously developed EER to relational DBMS schema translator [7], the object query translator allows application programs and user interfaces to query databases in terms of objects.

The problem we address is the link between an application and its underlying relational DBMS. Often, applications require specialized object *views* of the data stored in the database. The structure of these views usually cannot be represented directly by EER objects, where some view objects correspond to multiple EER objects. For example, a biologist's view of a genomic *map* in a molecular biology application looks as shown in figure 1. Genomic maps are schematic representations of DNA fragments which correspond to regions on a chromosome; each fragment contains locations of various DNA markers called *loci*. While from a biologist's point of view a genomic map is a single (complex) object, the information

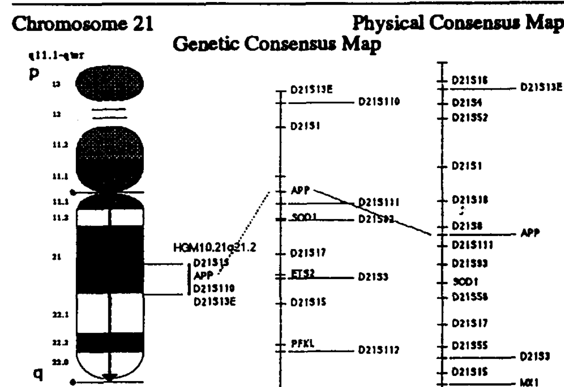


Fig. 1. Examples of Genomic Maps.


* Issued as technical report LBL-32019. This work was supported by the Applied Mathematical Sciences Research Program and the Office of Health and Environmental Research Program, of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC03-76SF00098.





U.S. Government Work Not Protected by U.S. Copyright

on a map is typically represented by several EER objects, as shown in figure 6 of section 5; in a relational database such a *map* would be represented by multiple tuples scattered among several relations. The definition of an application-specific object represented by multiple EER objects involves not only an EER structural specification, but also specifying the construction of an instance for that object from multiple instances of EER object-sets. In a relational DBMS this can be achieved by executing one or several fairly complex SQL queries. This is a complex, tedious, and error-prone process when done manually. For a molecular biology application, for example, the procedures supporting the manipulation of *maps* and related information amounted to over 5000 SQL lines [9].

Object-level query languages provide a more concise and intuitive way of specifying queries and defining application views. The design considerations for such a language, called COQL (Concise Object Query Language), its translation into SQL, and its use in a scientific application are presented in this paper.

COQL is designed for expressing queries over an object-level data model. The only assumptions made are that such a data model is capable of describing objects, their attributes, and connections between objects. The nature of these connections is not visible in COQL, so that users do not have to understand concepts such as generalization or various types of object aggregations.

Our experience with using COQL for complex database applications has shown that the complexity (in number of query lines) of COQL queries relative to equivalent SQL queries can decrease by a factor of 10. To illustrate the conciseness of COQL, consider genomic *maps* represented using the EER schema shown in figure 6 of section 5. In order to retrieve *genetic map* objects together with their *loci* and *citations*, the following COQL query can be specified:

```
OUTPUT MAP:  Name, Name OF LOCUS,
                   Title OF CITATION;
CONDITIONS MAP:  Type = "Genetic";   END
```

The query above associates every *genetic map* instance of the MAP entity-set, with its name, a set of names for LOCUS, and a set of titles for CITATION. Expressing this query in SQL for the Sybase DBMS, for example, requires a sequence of two subqueries for a total of over 30 lines of SQL, where most of the query involves join conditions.

Although our example is taken from a scientific application, we note that the need for application-specific views that reflect more closely the way users perceive the application, exists in conventional applications, as well.

The development of COQL is part of an evolutionary approach to the development of data management systems. Our approach entails developing techniques and tools that allow users to use existing commercial relational DBMSs, while ensuring a way to transfer to other DBMSs in the future. This is achieved by insulating the applications from the underlying DBMS using an object model. In the short term, we are developing translators from this object model to commercial relational DBMSs. In the future, we plan to implement the same object model on top of new DBMSs. This will provide a smooth evolution environment, since existing applications will continue to interact with the same object model. However, because the underlying DBMS will be more powerful, it will be possible to enrich the object model with specialized data structures and operations.

The rest of the paper is organized as follows. As background, we describe in section 2 a three-level architecture approach for supporting data management applications, and work previously done on the translation of EER schemas into relational DBMS schemas [6]. COQL is described in section 3. Section 4 describes the COQL to SQL translator, and discuss several practical issues related to the translation. An example of using COQL in supporting scientific data management applications is discussed is section 5. Section 6 contains concluding remarks and briefly discusses further plans.

## 2. Background

Typically, the languages provided by database management systems (DBMSs) for describing data structures and for querying data are at a general-purpose, lower level, of abstraction than that of the underlying applications. Database descriptions involve mainly technical terms that are foreign to application users, contain elements that have no direct counterpart in the application, and therefore obscure the semantics of the application. Moreover, DBMS databases have very large definitions (schemas). For the Sybase DBMS, for example, a database consisting of tens of tables requires thousand of lines of code for defining tables, indexes, and procedures for maintaining the integrity of data. Similarly, manipulating (i.e. retrieving and updating) data in databases involves the development of numerous procedures. Developing and maintaining database definitions and procedures is a tedious, error prone, and time consuming process.

### 2.1 A Three Level Architecture

Because of the discrepancy between the constructs provided by DBMSs and the level of abstraction required by applications, there is a need for supporting data structures and operations at two levels: an *application* level that contains the application-specific data structure definition (schema) and operations, and a *database* level that contains the application data structured according to a database definition (schema) using constructs supported by the

underlying DBMS. Consequently, the application-specific schemas and operations must be mapped into database schemas and operations, and the data retrieved from the DBMS must be converted into data structured according to the application-specific schema.

For reasons explained below, the schema, operation, and data mappings mentioned above can be simplified by introducing an intermediate object-level between the application and database levels. We have selected the *Extended Entity-Relationship* (EER) model for this object-level mainly because of its widespread use in designing relational databases.

As shown in figure 2, each level has schema, data, and operations (queries) associated with it. The intermediary EER level allows decomposing the operation and schema mappings between the application and database levels into simpler mappings between the application and EER levels, and the EER and database levels, respectively. Once general purpose translators between the EER and relational DBMS levels are provided, it is significantly easier to develop translators for application-specific views for the following reason. EER schemas and queries are specified in terms of objects and object connections, and therefore are inherently more concise, simpler to specify, and simpler to comprehend than relational DBMS schemas and queries. Accordingly, expressing application-specific data structures and operations using EER constructs and queries is significantly simpler than using relational DBMS constructs and queries. Furthermore, since developing EER schemas and queries is independent of a specific DBMS, they can be transferred to other DBMSs without affecting existing application-specific software.

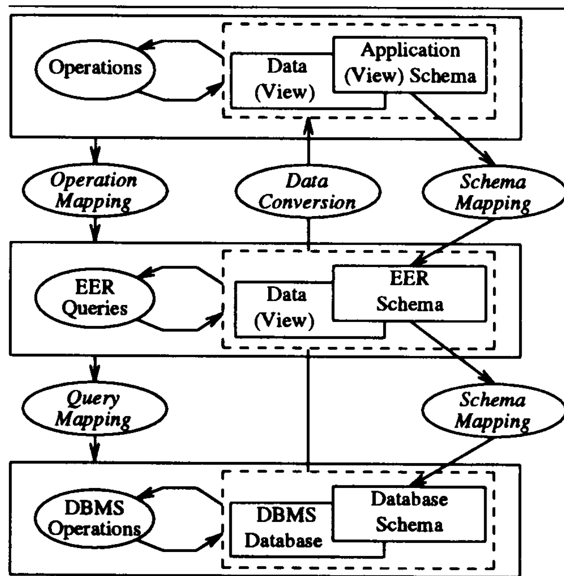


Fig. 2. Data Management System Architecture.

As mentioned above, EER schemas can be implemented over commercial relational DBMS using tools that can automatically carry out the EER to relational DBMS schema mapping [6]. In this paper, we are concerned with the specification of EER operations and their and mappings to relational DBMS queries.

## 2.2 The Extended Entity–Relationship Model

The *Extended Entity-Relationship* (EER) model is a version of the popular *Entity-Relationship* (ER) model [1]; the EER model has two additional constructs, generalization and full aggregation [6]. For the sake of brevity, we present only the terms used to describe EER constructs; detailed definitions can be found in [6] and [12].

An EER description of a database application is called an EER schema, and can be represented graphically in a diagrammatic form as shown in figure 3. An EER schema consists of *entity–sets* (e.g. EMPLOYEE, PROJECT), and of entity-set associations, called *relationship–sets* (e.g. WORK, LOCATED). We refer commonly to entity-sets and relationship-sets as *object-sets*. Individual object-set instances are qualified by *attributes* (e.g. Name, Salary). Entity-sets, relationship-sets, and attributes are represented diagrammatically in an EER schema by rectangle, diamond, and ellipse shaped vertices, respectively; relationship-sets are connected by arcs to the object-sets they associate, and entity-sets and relationship-sets are connected by arcs to their attributes. *Generalization* is an abstraction mechanism that allows viewing a set of *specialization* entity-sets (e.g. MANAGER, SECRETARY) as a single *generic* entity-set (e.g. EMPLOYEE). A specialization entity-set (e.g. MANAGER) *inherits* the attributes and relationship involvements of all its generic entity-sets (e.g. MANAGER *inherits* Age, Salary, WORK, and ASSIGNED_TO from EMPLOYEE). Generalization is represented diagrammatically in an EER schema by arcs labeled *ISA*, connecting specialization entity-sets to generic entity-sets.

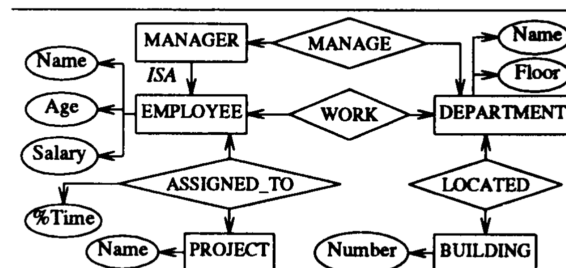


Fig. 3. An Extended Entity-Relationship Schema Example.

## 2.3 Translating EER into Relational Schemas

Our approach to translating EER schemas into relational schemas [6], involves separating the translation process into several independent stages, such as normalization, assigning names to relational attributes, merging relation-schemes, etc. The translation procedure developed in [6] has been implemented as part of a *Schema Definition and Translation* (***SDT***) tool [7].

***SDT*** works in two main stages. In the first stage, ***SDT*** maps EER schemas into normalized DBMS-independent relational schemas consisting of definitions for relations, keys, and declarative referential integrity constraints. In this stage interchangeable procedures for assigning names to relational attributes and merging relations can be applied. In the second stage, ***SDT*** generates schema definitions for specific DBMS; for DBMSs that have procedural referential integrity mechanisms (e.g. *triggers* in Sybase 4.0), ***SDT*** generates in this stage the appropriate insert, delete, and update procedures for maintaining the referential integrity constraints. In addition, ***SDT*** generates a *metadatabase* that contains information on the EER schema, the generated relational schema, and the mapping between these two schemas. This metadatabase is implemented using the underlying DBMS, and is used by the COQL query translator.

***SDT*** targets several relational DBMSs: DB2, Sybase 4.0, Ingres 6.3, Informix 4.0, and Oracle 6.0. ***SDT*** has been used for the development of databases for scientific applications at various laboratories worldwide. ***SDT***'s automatic mapping from concise EER specifications to more verbose DBMS definitions is faster and less error-prone than developing DBMS definitions directly. For example, the EER schema for a molecular biology application, was about 150 lines long, while its Sybase schema definition, including trigger procedures, was about 3500 lines long [9].

# 3. A Concise Object Query Language

Object data models are commonly used in practice for relational database design. For querying databases, however, users often have to interact directly with relational DBMSs. A natural extension of using an object data model for designing databases is to provide a query language also based on the object data model. In this section we describe such a language, the *Concise Object Query Language* (COQL). The translation of COQL queries into SQL queries is discussed in the next section.

Since we have selected the EER model as our object data model, we have considered initially adopting one of the ER or EER query languages proposed in the literature (e.g. [2, 3, 5, 11]). Although COQL has features that are common to these languages, it is unique in its support of inheritance, its conciseness, and by allowing attributes associated with certain object-sets to be viewed as attributes of other object-sets. The latter is essential for supporting the construction of application-specific views. These features are described below. The full syntactic definition for COQL is given in [8]. The examples in this section are based on the EER schema shown in figure 3.

## 3.1 Main Features

We begin with a simple example that illustrates the conciseness and the main components of a COQL query. Consider a request for finding the names of departments that have managers earning more than 75K, along with the names of the employees working in these departments. The intention of this request is to restrict departments to those that have managers earning more than 75K, and view the names of employees as (*foreign*) attributes of departments, which entails considering departments *regardless* of whether they do or do not have employees. This request is expressed as follows in COQL:

**OUTPUT** DEPARTMENT: Name, Name **OF** EMPLOYEE;
**CONDITIONS** MANAGER: Salary > 75,000; **END**

The query above is concise and quite close to its English formulation. Note that the same construct is used for (local attribute) Name, (inherited attribute) Salary, and (foreign attribute) Name of EMPLOYEE. The latter is treated like any other attribute of DEPARTMENT, that is, if a DEPARTMENT object is not associated with an EMPLOYEE, then a null EMPLOYEE Name is associated with that DEPARTMENT.

Unambiguous connections between object-sets appearing in a COQL query do not need to be specified. The ability to express queries without explicit connections contributes to their conciseness. When the connections are ambiguous, several strategies can be used for selecting the intended connections, or the user can include an explicit connection statement in the query. In the query above, for example, there are two possible connections between MANAGER and DEPARTMENT and therefore one of them must be selected. Before being translated into SQL, the connections between the object-sets appearing in the COQL query are either fully specified by the user or are derived by the system. The **CONNECTIONS** statement for the query above is:

**CONNECTIONS** MANAGER MANAGE DEPARTMENT;
EMPLOYEE WORK DEPARTMENT;

COQL is characterized by the separation of its **OUTPUT**, **CONDITIONS**, and **CONNECTIONS** parts; the order of these parts is arbitrary. The main reason for this separation is clarity and conciseness. Embedding conditions with object-set connection statements, as usually done in other ER and EER query languages (as well as SQL), can be cumbersome and confusing. Quite often object-set connectivity can be inferred without ambiguity,

and then the user can specify a query consisting of conditions and output statements only.

COQL supports both attribute and relationship *inheritance*. Thus, for entity-sets (e.g. MANAGER) that are specializations of generic entity-sets (e.g. EMPLOYEE), the attributes (e.g. Age) and relationship-sets (e.g. WORK) of the generic entity-sets are treated as if they are directly associated with (i.e are *inherited* by) the specialization entity-sets. This capability simplifies the expression of COQL queries and improves their conciseness.

## 3.2 Basic COQL Constructs

Object-sets are referenced in a COQL query using their names as specified in the underlying EER schema. A COQL **OUTPUT** or **CONDITIONS** statement consists of an object-set name *header* followed by ':' and the *body* of the statement. If the same object-set name appears several times in a COQL query, then it is interpreted as the same reference to (the same instantiation of) an object-set. In order to distinguish between multiple references to the same object-set in the underlying schema, object-set *aliases*, consisting of object-set names suffixed with *reference-labels*, can be used. For example, object-set EMPLOYEE can be referenced in a query using its name, that is, EMPLOYEE, or an alias, such as EMPLOYEE.1 or EMPLOYEE.NEW. An example of two distinct query references to the same object-set in the underlying schema is shown in the query below requesting the lists of employees for departments employing Jim Smith:

**OUTPUT** DEPARTMENT: Name, Name **OF** EMPLOYEE;
**CONDITIONS** EMPLOYEE.1: Name = "Jim Smith"; **END**

We refer below to object-set names and aliases appearing in COQL queries as *query object-sets*.

COQL **CONDITIONS** statements have a form similar to analogous constructs in other query languages. A **CONDITIONS** statement whose *header* refers to object-set $O_i$ can involve local and inherited attributes of $O_i$. **CONDITIONS** statements are formulas consisting of and/or compositions of atomic comparisons of the form $A\,\theta\, val$ or $A\,\theta\, B$ , where $A$ and $B$ are local or inherited attributes, $\theta$ is a comparison operator (such as '=', '>', **MATCH**, etc), and *val* is a list of atomic values. For example, departments with a floor number different than 2 and 4, and with names containing strings 'CS' or 'EE', are denoted by the following condition ('!=' denotes not equal, '%' denotes any string): DEPARTMENT: Floor != 2, 4 **AND** Name **MATCH** "%CS%", "%EE%".

In a COQL query, compositions of atomic comparisons which involve the same attribute in their left-hand-side can be abbreviated. For example, 'Age > 35 **AND** Age < 55' can be expressed in the following abbreviated form: 'Age [ > 35 **AND** < 55 ]'.

For describing further the COQL constructs, we use below a query example that expresses a request for finding the names and salaries of managers older than 35 and younger than 55, managing departments located in building 3, along with the names of these departments and the names of the employees who are assigned to the "DB" project and who work in these departments:

**OUTPUT** MANAGER: Name, Salary;
DEPARTMENT: Name, Name **OF** EMPLOYEE;
**CONDITIONS** MANAGER: Age [ > 35 **AND** < 55 ] ;
BUILDING: Number = 3;
PROJECT: Name = "DB";
**CONNECTIONS** MANAGER MANAGE DEPARTMENT;
DEPARTMENT LOCATED BUILDING;
EMPLOYEE ASSIGNED_TO PROJECT;
EMPLOYEE WORK DEPARTMENT; **END**

The query object-sets appearing in headers of **OUTPUT** statements determine the object structure of (i.e., the type of objects that are included in) the query result, and therefore are said to be the *primary* query object-sets. Conversely, non-primary (or *auxiliary*) query object-sets do not affect the object structure of the query result. *Auxiliary* query object-sets can appear in the body of **OUTPUT** statements or in headers of **CONDITION** statements.

In a COQL query, a query object-set can be associated with the local and inherited attributes of the object-set it references in the underlying schema. A primary query object-set can be also associated with (*foreign*) attributes of (i.e., that are associated with) auxiliary query object-sets. In the query above, for example, attribute Name of (auxiliary query object-set) EMPLOYEE is associated as a foreign attribute with (primary query object-set) DEPARTMENT; similarly, attribute Number of (auxiliary query object-set) BUILDING is associated as a foreign attribute with (primary query object-set) DEPARTMENT. Auxiliary query object-sets that appear in the body of **OUTPUT** statements can be associated, in turn, with *foreign* attributes of auxiliary query object-sets that appear in headers of **CONDITION** statements. For example, in the query above attribute Name of (auxiliary query object-set) PROJECT is associated as a foreign attribute with (auxiliary query object-set) EMPLOYEE.

The association of query object-sets with foreign attributes is detailed using the COQL **CONNECTIONS** statement that consists of a collection of sequences (*paths*) of query object-sets. For example, the associations of attribute Number of BUILDING with DEPARTMENT, of attribute Name of PROJECT with EMPLOYEE, and of attribute Name of EMPLOYEE with DEPARTMENT, are expressed using the last three paths of the **CONNECTIONS** statement in the query above. Identical query object-sets in different paths of a **CONNECTIONS** statement are interpreted as

identical references to the same object-set in the underlying schema; adjacent query object-sets must refer to object-sets that are related in the underlying schema either directly or via an inherited relationship-set.

The association of a foreign attribute, $A$, with a query object-set, $O_i$, in a COQL query can be:

1. *optional*, which entails considering the objects of $O_i$ *regardless* of the existence of an associated value for $A$;
2. *mandatory*, which entails considering the objects of $O_i$ only if they have an associated non-null value for $A$.

The optional or mandatory type of the association of a foreign attribute $A$ with a query object-set $O_i$ is determined as follows: if $A$ appears in the body of the **OUTPUT** statement whose header contains $O_i$, then the association of $A$ with $O_i$ is considered *optional*; otherwise (if $A$ appears in a **CONDITIONS** statement) the association of $A$ with $O_i$ is considered *mandatory*.

Note that while the optional type of association expresses the usual way of associating attributes with objects, the mandatory type of association is implied by the requirement of having non-null values for attributes involved in condition formulas. In the query above, for example, DEPARTMENT has (i) a mandatory association with (foreign attribute) Number of BUILDING (i.e. only departments that are located in building 3 are included in the result), and (ii) an optional association with (foreign attribute) Name of EMPLOYEE (i.e. departments are included in the result *regardless* of them having or not having employees).

In a COQL query all primary query object-sets have a mutual *mandatory* association. The mandatory type of this association entails including in the query result an object of an object-set referenced by a primary query object-set only if it is associated with an object from each object-set referenced by the other primary query object-sets. For example, in the query above the mandatory association of primary query object-sets DEPARTMENT and MANAGER entails including in the query result only managers that manage some department and departments that have managers. The detailed association of primary query object-sets is also specified using the **CONNECTIONS** statement, as illustrated by the first path of the **CONNECTIONS** statement in the query above.

Ambiguous object-set connections are not allowed in COQL queries. In the query above, for example, without explicit connections it would not be clear that PROJECT is related to EMPLOYEE. Unambiguous object-set connections, however, do not have to be fully specified in COQL. For example, the connection DEPARTMENT LOCATED BUILDING in the query above, could be specified only as DEPARTMENT BUILDING. Object-set connections must be fully specified in a COQL query (by users and/or via an inference process) prior to its translation into SQL.

COQL allows the specification of aggregate functions (**COUNT**, **AVG**, **MIN**, **MAX**, **SUM**) in output statements. Such functions define *derived* attributes associated with the corresponding primary query object-set. For example, the following output statement:

**OUTPUT** DEPARTMENT: Name, **COUNT** EMPLOYEE;

associates every DEPARTMENT with the number of EMPLOYEEs in the DEPARTMENT. A similar construct can be used in a condition statement, but has not been implemented yet. For example, the following condition statement: DEPARTMENT: **AVG** Age **OF** EMPLOYEE > 30; denotes departments in which the average age of employees is greater than 30.

### 3.3 Semantics of COQL Queries

A COQL query is interpreted as follows:

1. first, evaluate the mandatory associations of query object-sets with foreign attributes appearing in **CONDITION** statements;
2. next, apply selections according to formulas expressed in the **CONDITION** statements;
3. evaluate optional associations of primary query object-sets with foreign attributes that appear in **OUTPUT** statements;
4. evaluate aggregate functions, and associate the resulting aggregate (derived) attributes with the corresponding query object-sets;
5. finally, evaluate the mandatory associations of primary query object-sets.

For example, consider the query above. First, the mandatory associations of DEPARTMENT with (foreign attribute) Number of BUILDING and of EMPLOYEE with (foreign attribute) Name of PROJECT are evaluated. As a result, only departments that are located in some building and employees that are assigned to some project are considered for inclusion in the query result. Next, the selections expressed by the condition statements are applied; thus, MANAGER is restricted to the subset of managers that are older than 35 and younger than 55, EMPLOYEE is restricted to the subset of employees assigned to project "DB", and DEPARTMENT is restricted to the subset of departments located in building number 3. Then, the optional association of DEPARTMENT with (foreign attribute) Name of EMPLOYEE is evaluated, and, accordingly, departments are considered for inclusion in the query result regardless of whether they are associated with (one or several names of) employees or not. Finally, the mandatory association of DEPARTMENT with MANAGER is evaluated. Consequently, only departments having managers and only managers managing some department will appear in the result of the query.

## 4. Translating COQL into SQL

In this section we briefly describe the COQL query editor, present the main features of the COQL translator, and discuss some characteristics of translating COQL queries into Sybase SQL queries.

We have developed a query editor for specifying COQL queries [10] and a COQL translator for mapping COQL queries into queries in the SQL dialect specific to the underlying relational DBMS [8]. The COQL editor and translator use a *metadatabase* which contains information on the EER schema, the underlying DBMS schema, and the mapping between the EER and DBMS schemas. The metadatabase is essential for inferring the connections (paths) between the objects specified in a COQL query, and for providing the information on the correspondence of EER and DBMS constructs. As mentioned earlier, this metadatabase is automatically generated by ***SDT*** during its EER schema translation process [7].

The COQL query editor supports the graphical specification of concise COQL queries in terms of objects, attributes, operators, and values. Its purpose is to guide the user through the various stages of query specification. As mentioned in the previous section, COQL queries can be simplified by omitting unambiguous object connections. The connections that are not specified in the COQL query are inferred by the COQL query editor using the information stored in the metadatabase on the structure of the underlying EER schema. The result of this inference process is a COQL query with a complete connection statement. The COQL query editor is described in [10]. A COQL translator which targets the Sybase DBMS has been fully implemented, and is briefly described below.

The COQL translator maps a COQL query with a complete connection statement into one or several SQL queries in several steps listed below:

1. Abbreviated condition statements are expanded.
2. An *object-connectivity* graph that drives the COQL translation process is derived from the COQL query.
3. The attribute and relationship inheritance is resolved by expanding the object-connectivity graph.
4. The relational interpretation (*outer* or *regular* join) of the associations of query object-sets with foreign attributes or with other query object-sets is established, according to the mandatory or optional type of these associations.
5. Using (semantic) information from the EER schema, some outer joins are reduced to regular joins.
6. An SQL query generation plan is developed with the goal of minimizing the number of SQL subqueries; this plan depends on the specific SQL capabilities supported by the target DBMS.
7. The expanded COQL query is translated into the SQL dialect of the underlying relational DBMS.

Steps 1 through 5 do not depend on a specific DBMS, while steps 6 and 7 are specific to the target DBMS. We will illustrate below these steps using the COQL query presented in section 3.2. The COQL query translation process is described in detail in [8].

The first step of the COQL query translation consists of expanding abbreviated COQL conditions into SQL-like conditions. For example, the condition associated with MANAGER in the COQL query of section 3.2 is expanded into MANAGER: Age > 30 **AND** Age < 55.

The second step of the COQL query translation consists of constructing an ***object-connectivity graph*** for the COQL query; the edges of this graph represent pairs of adjacent query object-sets appearing in the **CONNECTIONS** clause of the COQL query, where each vertex represents a distinct query object-set appearing in the clause. For example, the object-connectivity graph for the COQL query of section 3.2 is shown in figure 4(i).

Inheritance is resolved by expanding the object-connectivity graph as follows: if a query object-set, $O_i$, appears in the query associated with inherited attribute $A$, where $A$ is the attribute of generic object-set $O_j$, then a vertex representing a (new) query object-set referencing $O_j$ is added to the graph, and is connected to the vertex representing $O_i$. For example, because MANAGER is associated in the COQL query of section 3.2 with inherited attribute Age, the object-connectivity graph shown in figure 4(i) is expanded with a vertex representing an alias for EMPLOYEE, EMPLOYEE.1, as shown in figure 4(ii). Note that an alias is required here in order to distinguish the new reference to EMPLOYEE from the already existing one.

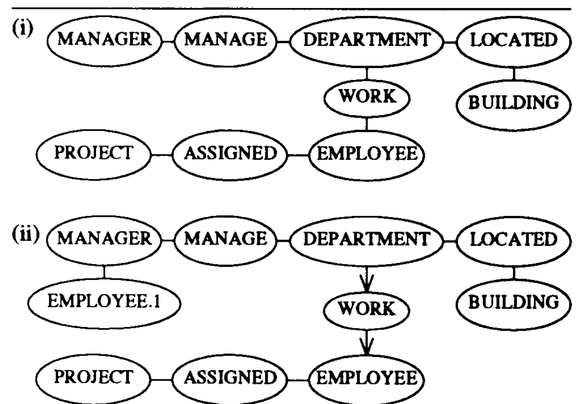


Fig. 4. Object-Connectivity Graphs for Query Example.

The next step of the translation process deals with the relational interpretation of the associations of query object-sets with foreign attributes, and the mutual associations of primary query object-sets. Let attribute $A$ of query object-set $O_m$ be associated as a foreign attribute with query object-set $O_1$. Then the **CONNECTIONS** statement contains a path involving both $O_1$ and $O_m$. If the association of $A$ with $O_1$ is optional (resp. mandatory) then for every pair of adjacent query object-sets, $O_{k-1}$ and $O_k$, $1<k\leq m$, on the path between $O_1$ and $O_m$, the relation representing in the underlying database the object-set referenced by $O_{k-1}$ is left outer-joined (resp. regularly joined) with the relation representing the object-set referenced by $O_k$. In the object-connectivity graph, the edge representing the connection of $O_{k-1}$ and $O_k$ is directed (resp. undirected) from the vertex representing $O_{k-1}$ to the vertex representing $O_k$. Associations of primary query object-sets are interpreted in a similar way to mandatory associations of foreign attributes, using regular joins.

In the COQL query of section 3.2, for example, only the path from DEPARTMENT to EMPLOYEE corresponds to an optional association of query object-sets with foreign attributes. Accordingly, in the object-connectivity graph the edges from DEPARTMENT to WORK and from WORK to EMPLOYEE are directed as shown in figure 4(ii).

The next step of the query translation is a *semantic optimization* process. Using semantic information from the EER schema, the relational join interpretation for some query object-set pairs is changed (reduced) from outer join to regular join whenever the regular and outer joins are equivalent. For the COQL query of section 3.2, for example, the left outer join of the relations representing query object-sets WORK and EMPLOYEE is equivalent to their regular join because WORK refers to a relationship-set and therefore (by definition in the EER model, and because of the referential integrity constraints in the relational database) there are no values in the joined columns of the WORK relation without a corresponding value in the joined columns of the EMPLOYEE relation. Consequently, the corresponding edge in the object-connectivity graph is changed from a directed into an undirected edge. Thus, the edge from WORK to EMPLOYEE in the object-connectivity graph of figure 4(ii), is changed from a directed into an undirected edge.

The order in which the joins involved in the interpretation of a COQL query are evaluated is determined by the semantics of the COQL query, as defined in section 3.3. For example, for the expanded COQL query of section 3.2 (after semantic optimization) the joins corresponding to paths (1) DEPARTMENT LOCATED BUILDING and (2) EMPLOYEE ASSIGNED_TO PROJECT must be evaluated before the joins corresponding to path (3) EMPLOYEE WORK DEPARTMENT, which, in turn, must be evaluated before the joins corresponding to path (4) EMPLOYEE.1 MANAGER MANAGE DEPARTMENT.

DBMSs such as Sybase 4.0 and Oracle 6.0 do not provide a way of enforcing the join precedence mentioned above, and therefore several SQL subqueries need to be generated. Moreover, systems such as Sybase 4.0 impose certain restrictions on combining outer and regular joins, and these restrictions have also an effect on the number of SQL subqueries generated. In order to minimize the number of SQL subqueries generated, an SQL-generation plan is developed. This plan is based on (1) associativity rules for outer and regular joins, and (2) restrictions on combining outer and regular joins. Regarding associativity, a left outer join followed by a regular join and a left outer join followed by a right outer join are not associative (see [4]). For the COQL query of section 3.2, for example, instead of four subqueries corresponding to the four paths above, associativity of regular joins followed by left outer joins allows generating one subquery corresponding to (composite) path EMPLOYEE.1 MANAGER MANAGE DEPARTMENT LOCATED BUILDING combined with path DEPARTMENT WORK, and another subquery corresponding to path WORK EMPLOYEE ASSIGNED_TO PROJECT. Associativity rules alone are not enough for developing an SQL-generation plan because certain associative combinations of outer joins are not allowed in some DBMSs. For example, Sybase 4.0 does not allow a relation to be involved in the inner and outer part of two left outer joins [12], although such a combination of outer joins is associative.

The last step of the translation consists of generating a sequence of SQL subqueries in the SQL dialect of the target relational DBMS, following the previously developed SQL-generation plan. The details of this process for Sybase are described in [8].

## 5. Application Example

In this section we discuss the usefulness of the EER schema and COQL translators in developing data management systems for scientific applications. For illustration, we use the *Chromosome Information System* (***CIS***) developed at Lawrence Berkeley Laboratory [9].

***CIS*** is intended to support molecular biology projects that are involved in constructing genomic *maps* such as those shown in figure 1. ***CIS*** provides biologists with facilities for storing, manipulating, and displaying such maps. The architecture of ***CIS*** follows the three-level architecture shown in figure 2, that is, consists of (1) mechanisms supporting application-specific data display and operations, (2) a database implemented using the Sybase relational DBMS, and (3) an intermediate level based on the EER model. The structure of the ***CIS*** database has been described using an EER schema that contains over 30 different object-sets representing *maps, loci,*

*probes*, bibliographic *citations*, etc., and their associations. A part of a simplified version of this schema is shown in figure 6. This EER schema has been subsequently translated using ***SDT*** into a Sybase database definition.

At the application level, *map* and other application-specific objects are characterized by complex attributes. For example, *map* objects have *name*, *type*, sets of *loci*, *citations*, etc. At this level, objects can be viewed as attributes of other objects. The relationships between application-specific objects are implied when an object (e.g. *loci*) is used as an attribute of another object (e.g. *map*). Application-specific views are presented to ***CIS*** users in windows (one window for each object-set) where object instances are displayed using sets of '*tag: value*' pairs. This application view can be characterized as follows: the view consists of a single *primary* object-set, and a set of attributes that may be single or multi-valued. Primary object-sets correspond to an object-set in the EER schema. The attributes in the view correspond to local attributes of the EER object-sets, inherited EER attributes, or attributes of other (*auxiliary*) object-sets.

Consider, for example, the EER schema shown in figure 6. At the application level *maps* are viewed as characterized by their name, type, as well as the names of *loci* and *probes* associated with *maps*, and titles of *citations* associated with *maps*. Note that this view reflects the way biologists perceive genomic *maps*. For them it is irrelevant that some attributes are inherited and others are associated with other object-sets. Furthermore, they are not aware of, and do not care about, the way these views are supported by the underlying DBMS.

The main operations supported by ***CIS*** at the application level are:

1. select and display all instances of an object-set;
2. select and display object instances that satisfy certain conditions;
3. browse through selected instances of an object-set;
4. for a given object instance, expand an attribute value; if this value identifies another object, then this operation entails displaying (by opening a new window) that object; this operation can be repeated recursively.

In the initial version of ***CIS***, a library of over 5000 lines of hand-coded SQL procedures were used for assembling application-level objects from the Sybase relations representing the EER object-sets. Implementing these procedures has been a tedious and error-prone process, especially as schema changes inevitably occurred. We show below how COQL and the COQL translator are currently used in order to replace the hand-coded procedures, and thus increase the productivity and accuracy of the interface development process.

An application-level object-set can be defined using a COQL query in a way similar to the view mechanism of relational databases. Here, however, the view mechanism is applied to the EER schema in order to specify composite application-level object-sets. For example, the following COQL query is used to assemble the instances of *maps* as required by the application-level interface:

```
OUTPUT MAP: Name, Type, Name OF LOCUS,
                      Title OF CITATION;
CONNECTIONS MAP CONSISTS_OF LOCUS;
            MAP REFERENCED_IN CITATION; END
```

Executing the SQL queries generated by the COQL translator for this COQL query provides all *map* instances required by the first interface operation mentioned above.

The selection of *map* instances that satisfy certain conditions requires modifying the COQL query above by adding a condition statement, such as:

```
CONDITIONS MAP: Type = "Cytogenetic";
```

Each application-level object-set is associated with a COQL query such as that shown above. Expanding an attribute value of a selected object instance, where the attribute value identifies another object, $X$, implies executing a modified version of the COQL query associated with the object-set containing $X$. For example, suppose that *citations* are associated with the following COQL query:

```
OUTPUT      CITATION: Title, Name OF AUTHOR;
CONNECTIONS CITATION WRITTEN_BY AUTHOR; END
```

and that for a selected *map* instance a particular citation, $X$, is selected for expansion. This expansion requires executing a modified version of the COQL query above, namely a query with the following added condition:

```
CONDITIONS  CITATION: Title = "X";
```

As illustrated above, application-level interfaces for browsing, querying, and navigating between application-level object-sets can be supported by simple modifications of pre-defined COQL queries. Note that in the example above each application-specific view involves only one

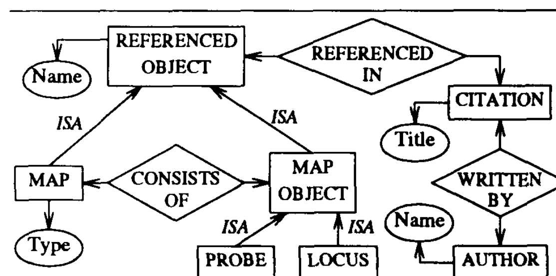


Fig. 6. An EER Schema for Genomic Maps.

*primary* (in COQL terms) object-set. In general, COQL can support the definition of application views involving multiple *primary* object-sets.

We have shown above how COQL can be used to support both the definition of application-specific views, and operations on such views. This approach reduces significantly the complexity of developing interfaces to applications supported by an underlying relational DBMS.

## 6. Concluding Remarks

We have described the *Concise Object Query Language* (COQL). Although COQL has features that are common to other object languages, it is unique in its conciseness and in allowing attributes associated with certain (*auxiliary*) objects to be viewed as attributes of other (*primary*) objects. We have described the translation process of COQL into SQL queries, and discussed its implementation for the Sybase DBMS. We have illustrated the construction of application-specific views for scientific applications using COQL and its translator.

Several practical considerations have guided the design of the initial version of COQL. The complexity of implementing the COQL translator both influenced and restricted the capabilities of this version of COQL. For example, only conjunctive object-set associations can be currently expressed in COQL and attributes of different object-sets cannot appear in the same condition statement. Nevertheless, COQL is powerful enough to express selection conditions, output, and aggregate functions. We plan to gradually extend COQL. As a first extension, we are currently incorporating update capabilities into the next version of COQL and COQL translator.

We are working on further optimizing the SQL queries generated by the COQL translator by detecting extraneous joins, that is, joins that can be removed from the generated SQL queries while preserving the semantics of the original COQL queries.

An important issue not addressed in this paper is dealing with results of COQL queries. While COQL allows the specification of complex application-specific object-sets, the result of a COQL query is currently returned in an unnormalized relational form. A more appropriate structuring of the data needs to be explored, and conversion procedures into such structures must be developed.

Finally, we have not fully explored performance issues related to the use of the COQL translator in data management systems. The COQL translation introduces an overhead (several seconds) that might be unacceptable in an environment where short response time is essential. In such environments the COQL translator can be used off line to generate parameterized stored SQL procedures, that can then be invoked at run time. Various alternatives for improving performance are currently examined.

**Acknowledgements.** We want to thank Ernest Szeto for his outstanding implementation of the COQL editor and COQL translator.

## References


[1] P.P. Chen, "The Entity-Relationship Model- Towards a Unified View of Data", *ACM Trans. on Database Systems* **1**,1 (March 1976), pp. 9-36.

[2] B. Czejdo, R. Elmasri, D.W. Embley, and M. Rusinkiewicz, "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model", *IEEE Computer* **23**, 3 (March 1990), pp. 26-36.

[3] R. Elmasri and G. Wiederhold, "GORDAS: A Formal High-Level Query Language for the Entity-Relationship Model", *Entity-Relationship Approach to Information Modeling and Analysis*, ER-Institute, 1981, pp. 49-72.

[4] C. Galindo-Jegaria and A. Rosenthal, "How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin", Proc. of the *8th Int. Conf. on Data Engineering*, pp. 402-409, 1992.

[5] V.M. Markowitz and Y. Raz, "ERROL: An Entity-Relationship Role Oriented Query Language", *Entity-Relationship Approach to Software Engineering*, North-Holland, 1983, pp. 329-345.

[6] V.M. Markowitz and A. Shoshani, "Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach", *ACM Trans. on Database Systems*, **17**, 3 (Sep. 1992), pp. 423-464.

[7] V.M. Markowitz and W. Fang, "SDT 4.1. Reference Manual", Lawrence Berkeley Laboratory Technical Report LBL-27843, 1991.

[8] V.M. Markowitz and E. Szeto, "The COQL Translator. Design Document and Reference Manual", Lawrence Berkeley Laboratory Technical Report LBL-31451, 1991.

[9] V.M. Markowitz, S. Lewis, J. McCarthy, F. Olken, and M. Zorn, "Data Management for Genomic Mapping Applications: A Case Study", Proc. of the *6th Int. Conf. on Scientific and Statistical Database Management*, June 1992.

[10] V.M. Markowitz, A. Shoshani, and E. Szeto, "The COQL Graphical Editor. Reference Manual", Lawrence Berkeley Laboratory Technical Report LBL-33218, 1993.

[11] A. Shoshani, "CABLE: A Language Based on the Entity-Relationship Model", Lawrence Berkeley Laboratory Technical Report LBL-22033, 1978.

[12] Sybase, Inc., "Transact-SQL User's Guide", Release 4.0, Emeryville, California, Oct. 1989.

[13] T.J. Teorey, D. Yang, and J.P. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", *Computing Surveys* **18**,2 (June 1986), pp. 197-222.