The Composite OLAP-Object Data Model

Elaheh Pourabbas *
Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti"-CNR
Viale Manzoni, 30 I-00185 Roma, Italy
pourabbas@iasi.cnr.it

Arie Shoshani
Lawrence Berkeley National Laboratory, USA
Mailstop 50B-3238, 1 Cyclotron Road Berkeley, CA 94720 USA
shoshani@lbl.gov

Abstract

In this paper, we define an OLAP-Object model that *combines* the main characteristics of OLAP and Object data models in order to achieve their functionalities in a common framework. We classify three different object classes: *primitive*, *regular* and *composite*. Then, we define a query language which uses the path concept in order to facilitate data navigation and data manipulation. The main feature of the proposed language is an *anchor*. It allows us to fix dynamically an object class (primitive, regular or composite) along the paths over the OLAP-Object data model for expressing queries. The queries can be formulated on objects, composite objects and combination of both. The power of the proposed query language is investigated through multiple query examples. The semantic of different clauses and syntax of the proposed language are investigated.

1 Introduction

The OLAP data model [5] was introduced in order to manage multidimensional summary databases. Similar to the Statistical Data Model [21] it consists of three basic constructs:

1. Dimensions, where each can consist of a multi-level classification hierarchy;

2. A multidimensional object, also referred to as a "cross-product" object;

*This work was supported by a Fulbright Scholarship academic year 2004-2005 while the author was visiting Lawrence Berkeley National Laboratory, U.S.A.

3. Summary attributes, where each is associated with the multidimensional object.

For example, a database for "average-income by city, race, and sex", can be modeled in an OLAP data model, where the dimensions are *City*, *Race*, and *Sex*, the multidimensional object is the cross-product $(City) \times (Race) \times (Sex)$, and the summary attribute is the "average-income". If, in addition, cities were organized by counties, and counties were organized into states, a classification hierarchy $City \rightarrow County \rightarrow State$ can be associated with the *City* dimension. *City*, *Region*, and *Country* are category attributes. This would allow the database system to generate summarization over the classification hierarchy, such as "average-income" for counties and states.

Various representations of these concepts were proposed, including the graphical representation of Statistical Models [4], Multidimensional OLAP, or MOLAP [1] [6], Relational OLAP, or ROLAP [12] [20], and Data Cubes [2] [8] [9]. However, in many applications the elements of the dimensions as well as the elements of the classification hierarchies can be objects in their own right, such as cities and states in the example above. In such cases, these objects can have their own attributes (e.g. the mayor of a city, or the governor of a state). Furthermore, such objects may be associated with other objects (such as the hospitals in a city). This obviates the need to have concepts from the OLAP modeling domain to be combined with the familiar object-attribute-association models (such as the Entity-Relationship model or similar Object models).

In this paper, we define an OLAP-Object model that permits the combination of concepts from both domains. We classify three different object classes: *primitive*, *regular* and *composite*. We investigate the well-formed composite objects and we study their summarization semantics. Then, we define a query language which uses the path concept in order to facilitate data navigation and data manipulation. The main feature of the proposed language is an *anchor*. It allows us to fix dynamically an object class (primitive, regular or composite) along the paths over the OLAP-Object data model for expressing queries. The queries can be formulated on objects, composite objects and combination of both. The power of the proposed query language is investigated through multiple query examples. The semantic of different clauses and syntax of the proposed language are investigated.

The paper is structured as follows. The next section describes the basic constructs of the OLAP-Object model, which are enable to capture and combine the main characteristics of both data models. Section 3 introduces a graphical representation of the components of the OLAP-Object data model, which are object classes (primitive, regular, and composite) and associations (simple and classification). Section 4 discusses the well-formed issues related to the composite object classes, and Section 5 defines their summarization semantics. Section 6 shows how paths enable us to link different object classes in order to facilitate data navigation and query formulation. Section 7 proposes a query language for querying OLAP-Object data model. The syntax of the proposed language is defined in detail and for each type of the object classes the power of the

langauge is investigated through multiple query examples. Section 8 represents the extension of the proposed langauge with concepts in order to be able to answering queries that use condition caused by ellipsis in natural language (e.g., their, its, etc.) and recursive queries.

2 Basic Data Modeling Constructs

First, we adopt the basic concepts of an object-attribute-association model. These include:

- *Object Class* represents a set of individual objects, each having a globally unique identifier. An object class has a label (name). An object class must have at least one attribute. Such a class is called a *regular* object class.
- *Attribute* a property associated with an object class. Each individual object in that class determines the attribute value associated with it. Therefore, there is a functional dependency of the attribute on the object class. An attribute has a label (name).
- *Association* a way of pairing objects from two object classes. An association has a pair of cardinalities: one-to-one, one-to-many, many-to-many. An association has a label (name).

For instance, let us consider two regular object classes, called Student, and Course. The attributes of the first class are *Name* and *Age*, while the attributes of Course are *Name* and *Start-date*. An association between these classes is *enrolls*

In addition we need to have two concepts that will allow OLAP structure to be represented in the data model. The first is a primitive object classes (such as "race" or "sex") that can be used for defining dimensions in the OLAP model or classification hierarchies in the dimensions. The second is a composite object class that supports the concept of a cross-product or data cube. We define these next:

- *Primitive object class* an object class that represents a finite enumerated set of values. The values represent the identifiers of the individual objects in the class. A primitive object class has a label (name).
- *Composite object class* an object class defined over two or more object classes, where each individual object has an identifier composed of the identifiers of objects form each of the object classes. Each individual object class may belong to a hierarchical classification structure. A composite object class can have at least one attribute. A composite object class has a label (name).

Note that a composite object class can be defined over primitive object classes, over regular object classes, and over other composite object classes. In this paper, we consider that a composite object class be defined over primitive or regular object classes.

For instance, let us consider "average-income by city, race, and sex" mentioned in the previous section. This can be modeled as a composite object class named *Income* where *City*, *Race*, and *Sex* are dimensions. *City* is a regular object class and its attributes are *Name*, *Major*, while *Race*, and *Sex* are primitive object classes. The attribute of the composite object class is *average-income*.

Finally, we make a distinction between a simple association and an summarizable association. A summarizable association must be one-to-many and must pass the test of completeness and non-overlap of objects as described in [19]. Example of summarizable associations are *is-in* between City and State object classes, or *offer* between Course and Department object classes. The object classes related by a summarizable association defines a classification structure, where each object class represents a *level* of this structure and corresponds to different granularities of viewing data. In general, a hierarchy represents the relationships between domains of values. Each operation on a hierarchy can be viewed as a mapping from one domain to a smaller domain.

3 A Graphical Representation

The representation of the main components of our model, i.e. object classes and relationships are based on nodes and arcs. More specifically, primitive and regular object classes are represented by a simple node (◯), whereas a composite object class is represented by a so-called "circledtimes" ($\bigotimes$) node. As we mentioned in the previous section, a composite object class is defined by a set of primitive o regular classes. For simplicity, we call them *component object classes*. In the graphical representation, each component class is connected to the belonging composite class by a dashed line. Each of the above mentioned node is labeled to indicate the name of the classes. For a clear illustration, the attributes of each node are indicated in *italic* style text. Each component class may belong to a classification structure. A classification structure is defined by a certain number of categories

To represent a simple association or *simple relationship* between two object classes, a simple labeled arc is used. The label indicates the name of relationship and it is indicated in *italic* style text and lowercase letters. A summarizable association or *classification relationship* between two primitive or regular classes is illustrated by an oriented bold arc.

Example In Figure 1 a simple Object model is illustrated. In this example, Student, Course, Lecturer, Department are regular classes. The attributes of Student are *Name*, *Address*, *Birthday*, *Age*; Course and Lecturer have attributes *Start-date* and *Name*, respectively.. The attributes of Department are *Name*

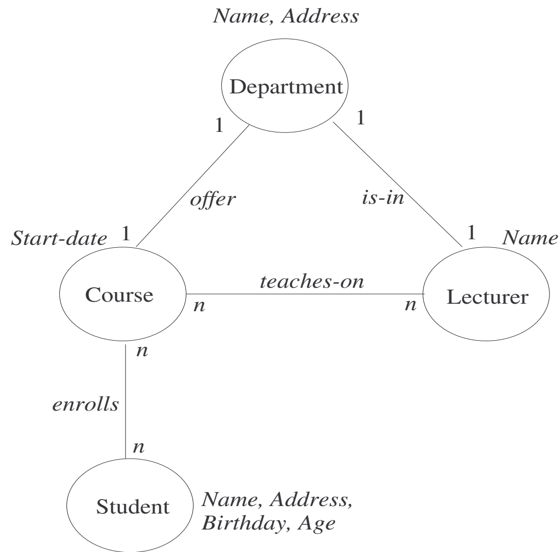


Figure 1: Example of a database with regular object classes

and *Address*. While, “enrolls”, “offer”, “teaches-on”, and “is-in” are simple relationships between each pair of the mentioned classes.

In Figure 2, an example of graphical representation of a composite class is illustrated . The composite class *Demographic_Group* is shown by a circledtimes node, and it is defined by the component classes *Age*, *Sex*, and *City*. They are the dimensions of Demographic_Group, and are connected to the composite class by dashed arcs. *Demographic_Group* has a single attribute named “Population”. The classification relationships between pairs of regular classes, *City*, *Region*, and *Region*, *Country* are illustrated by oriented bold arcs.

4 Well-Formed Composite Object Classes

As we mentioned in the previous section, a composite object class references two or more object classes, or component object classes. Each component object class (e.g., *City*) can in turn reference its own parent class (e.g., *Region*), where each its value is associated with one and only one value at the parent level; thereby modeling many-to-one relationships. This introduces a hierarchical relationship between pairs of object classes, that is a functional dependency from one level of a hierarchy to the next level in the hierarchy. In other words, the structure induced by functional dependencies defines a partial order on instances.

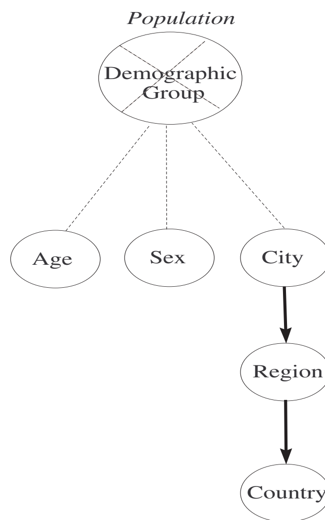
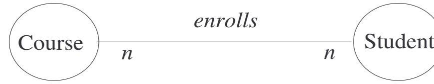


Figure 2: Example of a composite object class

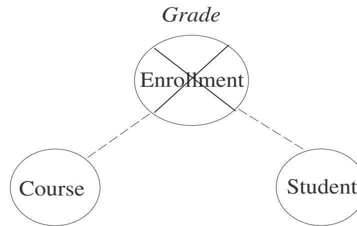
Generally, the definition of the composite objects classes is based on the idea to manage many-to-many relationships between component object classes. We start to considering a binary many-to-many relationship, which is an association between the instances of one object class with the instances of the another object class and represents that there is no functional dependency between object classes. For a clear description, let us consider the many-to-many relationship *enrolls* between object classes Student and Course. This relationship associates students and the courses they attend. If the relationship *enrolls* is not defined by any attributes then in the database schema design it can be represented by a many-to-many relationship (see Figure 3-(A)).

Otherwise, for instance, if the attribute *Grade* for the *enrolls* relationship is considered, then this many-to-many relationship cannot be taken over directly in the database design. The method which we adopt consists of to introduce a new composite object class for the relation, named *Enrollment* and associate the original object classes, i.e. Student, and Course to this composite class (see Figure 1-(B)). Note that, a composite object class without any attribute is equivalent to the simple many-to-many association between component object classes, as is illustrated in Figure 3-(A).

In Entity-Relationship models, ternary or multiple binary relationships are evaluated as part of the overall diagram and they have been analyzed in order to coexist with other relationships. Teorey [22] has discussed the functional dependencies that can be derived from various cardinalities of 1:1:1,1:1:M, 1:M:N, and M:N:P ternary relationships. In [13], specifically, a set of rules for combining ternary relationships with binary relationships to investigate the structural va-



(A)



(B)

Figure 3: Example of a binary many-to-many relationship (A) and a composite object class (B)

lidity of ternary relationships is defined. In [14] [15] the decomposition of ternary relationships with various cardinalities (1:1:1,1:1:M, 1:M:N, M:N:P) into binary relationships is discussed. These studies consider mainly the issues of embedded binary combinations in ternary relationships, their permitted combinations, and their decomposition into multiple binary relationships. We study the ternary and higher degree relationships in the context of composite object classes and we define some rules in order to guarantee their well-formed structure.

Generally, the ternary relationship of cardinality 1:1:1 represents the trivial cases, where the cross product of three object classes at extensional level is defined by only one tuple; i.e. each object class is defined by one instance. Similarly, 1:1:M cardinality represents cases where two object classes participating in the relationship are defined by one instance. This ternary relationship is characterized by a one-to-one binary relationship and two one-to-many binary relationships. The ternary relationship 1:M:N configures cases of two one-to-many and one many-to-many binary relationships. The case of M:N:P represents three many-to-many binary relationships. The relationships with cardinalities 1:1:1, 1:1:M, 1:M:N define functional dependency between object classes, while M:N:P represents that there is no functional dependency between any pair of object classes.

In our OLAP-Object model, a composite object captures the ternary and higher degree relationships, where the cardinality of the binary relationships between each pair of its component object classes is many-to-many. The fol-

lowing theorem gives the condition upon which a composite object is said to be well-formed.

Theorem 4.1 *A composite object class is well-formed if between any pair of its component classes there is no functional dependency.*

Proof: Let us consider a composite object be defined by three object classes named X, Y, and Z. Let there is at least one pair of object classes, say X, Y, between which there is a functional dependency. Let to each instance of X corresponds N instances of Y. This dependency represents the binary relationship with 1:N cardinality between X and Y, which defines a partial order on instances X and Y. Therefore, these two component classes result to be related by a hierarchical relationship. This contradicts the definition of a composite object, which is defined by two or more object classes, each of which can be associated separately with a classification hierarchy. Thus, there can not be a hierarchical relationship between any pair of component classes.

For example, if the schema of the database shown Figure 3-(B) is extended by the object class Department, then the resulted composite object is not well-formed (see Figure 4-(A)). As stated in Theorem 4.1, there is functional dependency between object classes Course and Department. The cardinality of this binary relationship is one-to-many. The well-formed composite object is shown in Figure 4-(B).

5 Summarization Semantics

The summarization over composite objects must satisfy the conditions of the *summarizability* discussed in [19]. These conditions are: disjointness of category attributes (or levels) in hierarchies; completeness in hierarchies; correct use of measure (summary attributes) with statistical functions. Disjointness implies that instances of category attributes in dimensions form disjoint subsets of the elements of a level. Completeness in hierarchies means that all the elements occur in one of the dimensions and every element is assigned to some category on the level above it in the hierarchy. Correct use of summary attributes with statistical functions depends on the type of the measure and the aggregation function, like COUNT, SUM, MIN, MAX, and AVERAGE.

The query expressed over composite objects will assume a number of semantics in the context of our query langauge, which are defined as follows.

Definition 5.1 *To summarize a category attribute over a composite object the following semantic conditions must hold:*

1. *if a dimension (or component object class) is not specified in the summary query, then the summarization must take place over all the values of its individual objects.*

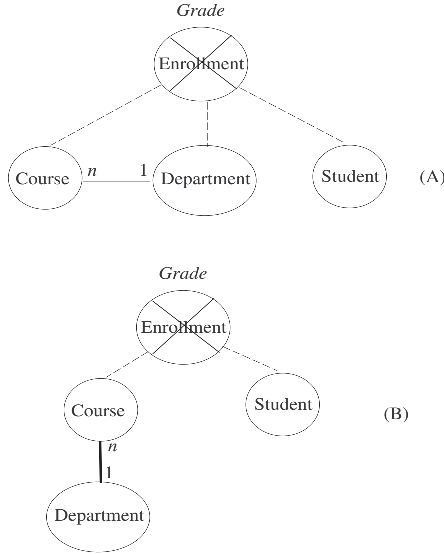


Figure 4: Example of a non well-formed (A) and a well-formed composite object class (B)

2. *if a sequence of values of a certain dimension is specified in the query, then the restriction must take place over this sequence and the remaining individual object values are skipped out. In the result of query, the composite object is defined by this sequence of values.*

3. *if a single or a range of values of a certain dimension is specified in the query, then the summarization must take place over remaining individual values, and the summary attribute of the composite object will be evaluated for this single/range of values.*

4. *if no dimension is specified in the query, then the summarization must take place over all the values of dimensions. This corresponds to achieve only one value for the summary attribute of the composite object (known as, grand-total).*

Example Let us consider the composite object shown in Figure 2. Let us consider the following query "Find city and population in $20 \div 40$ age-groups". *Population* is the summary attribute to which the aggregation functions can be applied. Note that, Sex is not specified in the query, then, according to the Definition 5.1-*(1)* the composite object Demographic_Group will be summarized over all values of Sex. Therefore, Demographic_Group is defined by City, and Age. In the same query, a sequence of values (i.e. $20 \div 40$) for Age is specified, then according to Definition 5.1-*(2)*, the result of query is Demographic_Group by City and Age, where Age is specified by this sequence.

In the above query, if city and population for *females* in $20 \div 40$ age-groups" is specified, then according to Definition 5.1-*(3)*, the value *male* of Sex is skipped out and the result Demographic_Group indicates *Population* of *female* by city and age in $20 \div 40$.

Now, let us consider the query: "Find city and population between $20 \div 40$ age-groups", this requires to aggregate the summary attribute *Population* over a range of values of Age from 20 to 40. Therefore, by Definition 5.1-*(3)*, the result indicates *Population* of age between $20 \div 40$ by city.

Finally, if only population of demographic group is specified in the above query, then Age, Sex, and City are summarized (see Definition 5.1-*(4)*) and the result indicates total population.

6 Paths over the OLAP-Object Data Model

The idea of using paths to ease accessing objects in query languages goes back to $80's$. This was given in one of the first approaches, GEM ([23]) which was based on QUEL, and in other approaches which were the extension of SQL, like OSQL [10], ORION [18], XSQL [16], O_2SQL [3], ESQL [7], etc, or similarly in functional-logic programming [11], [17].

Our data model is based on the concept of path, which in our believe is more appropriate to facilitate data navigation and query formulation over OLAP-Object databases. A path expression defines the steps to take on the path toward the objects to be retrieved in a query. This enable us to link objects without having to express explicit join conditions.

In our model, a path expression consists of one step, which generates a sequence of objects. The objects in a sequence are separated by dot notation. The sequence ends up by a colon, after which one predicate is expressed in order to be evaluated on the properties of the terminal object class. Obviously, a path can be defined by more steps. This because in the *same* path, more than one predicate can be expressed on the properties of the terminal object or the properties of any object in the sequence can be referred to by predicates. In our model, paths can be applied in one dimension or step. Therefore, a multi-steps path is split up to a conjunction of several single-step paths.

Example Let us consider the schema of database shown in Figure 1. The query "students who are enrolled in course Mathematics offered by department Electrical Engineering" is expressed in a single step as follows:

Student.Course:Name="Mathematics".Department:Name= "Electrical Engineering"

In a multi-steps path, the above query is expressed by conjunction of two paths:

Student.Course:Name="Mathematics" **AND** Student.Course.Department:Name= "Electrical Engineering"

If between two objects more than one relationship hold, and if both objects get involved in the path expression then only one relation should be specified. For instance, let us consider Figure 1, where the object Student is related to Course by an additional relationship likes. Therefore, to specify the students who are enrolled in course mathematics, the relationship enrolls is indicated between Student and Course. For a sake of explanation, this relationship is enclosed in brackets. The path is shown as follows:

Student(enrolls)Course:Name="Mathematics"

Similarly, let we ask the courses taken by lecturer in department of Electrical Engineering. The object class Course is related to Department by offer relationship, and to object Lecturer by teaches_on. Since only one relationship holds between pairs of these object classes, any reference to relationship is skipped out in the path expression as follows:

Course.Lecturer.Departemnt:Name="Electrical Engineering"

The path expressions are defined starting from a certain regular or primitive object class, and they are specified in the clause **CONDITION** of our query language construct, which will be introduced in the next section. In a composite OLAP-Object data model, the regular and/or primitive objects are the components of a composite object, from which a path over other objects is constructed. In other words, these objects binds composite objects of an OLAP data model to the objects of Object data model.

Let us consider the schema of the OLAP-Object database shown in Figure 9, which is obtained combining the schemas of databases shown in Figure 2 and Figure 6. Let us ask the number of beds in wards maternity of cities where the population female of age between $20 \div 40$ is greater than 100000. The object "City", that is a component of the composite object class "Demographic_Group", is a binding object class to Object-model shown in Figure 6. The predicate expressions are defined, respectively, over Sex (female), Age $(20 \div 40)$ and a path as follows:

City.Health Center.Ward:Name="Maternity"

Path through composite object that does not involve attribute of this object class is a regular path. For instance, let in Figure 4-(B) an object class named City be related to Student by *lives* relationship. Let us consider the query as follows: "City of students who are enroled in course Computer Science". This query does not specify *Grade* attribute. Then, the following regular path is allowed.

City.Student.Enrollment.Course:Name="Computer Science"

7 The Query Language

The main feature of the proposed query language is an *anchor*. It allows us to fix dynamically an object class (primitive, regular or composite) that is invoked from query. We call this *anchor-object class*. The condition expressions or path condition expressions can be formulated on the anchor-object class or on classes that are joint from anchor-object class through paths over the OLAP-Object data model described in Section 6. In output, the results of a query refer to the anchor-object class and are obtained from the evaluation of the condition expressions. In the next subsection, the syntax of the proposed query langauge is discussed, and the semantic of different clauses are investigated.

In query language, we need to separate the query constructs for regular object classes (including paths on conditions and output) and composite object classes. In subsection 7.2 and subsection 7.3, we describe the application of the proposed query constructs to regular and composite object classes. Then, in subsection 7.4 we show how the combination of the object query construct and composite query construct can be applied to combined OLAP-Object schemas without splitting the query itself into subqueries. In subsection 7.5, we discuss composite-composite queries. They invoke a given composite object class (C_i) in output but the evaluation of condition expressions depends on a second composite object query (C_j). The partial results from (C_j) are provided by a subquery. We investigate the power of the proposed query language through multiple query examples. Note that the queries expressed over primitive object classes (e.g., Sex) refer to their domain values (e.g., Female, Male) and the explanation of these queries is straightforward.

7.1 The Syntax

We propose a query language which is based on four constructs, named **ANCHOR**, **FROM**, **CONDITION**, and **OUTPUT**. We define the syntax of each construct. We use the symbols of Table 1 and basic structures introduced in APPENDIX A. In APPENDIX B, the syntax of primitive, regular and composite object classes is defined.

The ANCHOR clause

The ANCHOR clause takes the form
ANCHOR `<anchor-class>`

This clause contains either one composite object class or two or more primitive and regular classes. If two or more primitive and regular object classes are indicated, this represents that they belong to a composite object class. In such a case, it requires to indicate the composite object class in the FROM clause. We describe this next.

```
<anchor-class>::=             <anchor-class-simple>|'['<anchor-class-composite>']'
<anchor-class-simple>::=      <primitive class> {',' <primitive class>}
                              |<primitive class> {',' < regular class>}
                              |< regular class>{',' <primitive class>}
                              |< regular class>{','<regular class>}
<anchor-class-composite>::=   <composite class>
```

The main characteristic of this clause consists of to invoke all the properties of objects in output. This is described next in the definition of the OUTPUT clause .

The FROM clause

The FROM clause takes the form

FROM `<anchor-class-composite>`

This contains only one composite object class, and indicates that one or more components (primitive or regular classes) is/are enclosed in the ANCHOR clause.

The CONDITION clause

The CONDITION clause takes the form

CONDITION `<codition-expressions>`

Let O be the result of evaluating the ANCHOR clause. Then, the result of CONDITION clause is an object that is derived from O by eliminating all tuples for which the conditional expression does not evaluate to true. If the CONDITION clause is omitted, it indicates that the result is simply O. The syntax of <condition-expression> is reported below.

To represent conditions caused by ellipsis, we introdcue the concept of *back reference*, which is described in detail in Section 8.

```
<condition-expressions> ::= <conditions> | <path-conditions>

<conditions> ::= <atomic-condition> | <conditions> <conjunction> {<conditions>}

<atomic-condition> ::= <expression>|<expression> <conjunction> <expression>
```

```
<expression>::=<anchor-path>':'<subexp>|<back-reference-expression>
|<object class>':'  {'('<subexp> <conjunction> <subexp>')'}
|<object class>':'<attribute name><null-operator>
|<object class><range-operator> <set-of-values>
|<object class><in-operator><query-item>

<subexp>::=<attribute name><comparison-operator> <primitive value>

<back-reference-expression>::=
<anchor-path-ref>':'<attribute name>
<comparison-operator> <attribute name> '('<anchor-path-backref>')'

<conjunction> ::= AND | OR
<comparison-operator>::= '=' | '<' | '>' | '=' | '=' | '!='
<primitive-value> ::= <numeric> | "<string>"
<null-operator> ::= NULL | NOT NULL
<range-operator> ::= EQ | BETWEEN
<set-of-values> ::= <set-of-numerics> | <set-of-strings>
<set of numerics> ::= {<numeric> ',' <numeric> }
<set-of strings> ::= {<string> ',' <string>}
<in-operator> ::= IN | NOT IN

<path-conditions>::=
<path-condition> | <path-condition> { <conjunction> <path-conditions>}
<path-condition>::=<anchor-path-1-condtion> | <anchor-path-condition>

<anchor-path-1-condition>::=
<anchor-with-composite><comparison-operator><primitive value>
|<anchor-with-composite><null-operator>
|<anchor-with-composite><in-operator><query-item>

<anchor-with-composite>::=
'['<anchor-class-simple>']''('<composite class>')'':'<attribute-name>

<anchor-path-condition>::= <anchor-path>':'<subexp>
|<anchor-path>':'{'('<subexp> <conjunction> <subexp>')'}
|<anchor-path>':'<attribute name><null-operator>
|<anchor-path><in-operator><query-item>
```

```
<anchor-path> ::=
<primitive class>{'.'<primitive class>|'.'<regular class>}|
<regular class>{'.'<primitive class>|'.'<regular class>}

<anchor-path-ref> ::=
<primitive class>'('variable')'{'.'<anchor-path>}|
<regular class>'('variable')'{'.'<anchor-path>}

<anchor-path-backref> ::=
{<anchor-path>'.'}<primitive class>'('variable')'|
{<anchor-path>'.'}<regular class>'('variable')'

<variable>::= <primitive value>
```

The OUTPUT clause

The OUTPUT clause takes the form

OUTPUT `<anchor-path-item-semicolonlist>`

The OUTPUT clause represents the result of evaluation of the ANCHOR, FROM, CONDITION clauses. We consider three cases for explaining anchor-path-item. They are explained as follows.

```
<anchor-path-item>::= <anchor-path-1>|<anchor-path-2>|<anchor-path-3>
```

Case 1. The anchor-path-item takes the form

```
<anchor-path-1>::=<anchor-with-composite>|<anchor-path>':'<attribute-item>

|<anchor-path>':''ID'
```

In the OUTPUT clause only the classes (primitive, regular, composite) enclosed in the ANCHOR clause are indicated. Therefore, the results are obtained from the evaluation of condition predicates on classes involved in the ANCHOR clause or the classes that are joint from these through paths. The results concern any set or all ('*' denotes all) attributes of the regular classes or classes that are derived from summarization over one or more dimensions of a composite class.

The query may ask all instances of a regular class. In this case, the 'ID' of the regular class is indicated.

Case 2. The anchor-path-item takes the form

```
<anchor-path-2>::=
<anchor-path>':'<Agg-function>'('<agg-attribute>')'|<anchor-path>':'COUNT
| '[' <anchor-class-simple> ']' '('<composite class>')'
':'  <Agg-function> '(' [<attribute name>] ')'
| '('<composite class>')' ':'  <Agg-function> '(' [<attribute name>] ')'

<agg-attribute>::=null| <attribute name>
<Agg-function>::=SUM|AVG|MIN|MAX
```

The aggregate function (except COUNT) are applied on summary attributes of classes which are involved or are joint from the classes indicated in the ANCHOR clause (e.g. SUM(Population) or Student.Course:COUNT). The COUNT function is applied to count the tuples of a given object class (e.g., Student:COUNT).

Case 3. The anchor-path-item takes the form

```
<anchor-path-3>::=<anchor-path>IN<query-item>
```

This occurs when a query can not be resolved in one step. In other words, the evaluation of a given query depends on the result of a subquery which provide the partial result in output. The complete syntax of queries is shown in APPENDIX C.

7.2 Object Query

In this case, an anchor is a regular class and the condition expressions are formulated either on anchor or on other regular classes along the paths (see Figure 5). In output the result can be anchor class itself or its properties.
In the following, we give some examples of query formulated on regular classes. They are based on the schema of database shown in Figure 1. A brief explanation for some of them is given.

Query Examples

Query 1: Students who are enrolled in courses taken by lecturer Johnson.

ANCHOR	Student
CONDITION	Student.Course.Lecturer: Name= "Johnson"
OUTPUT	Student:*

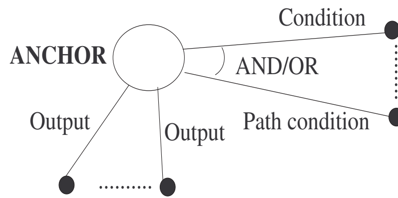


Figure 5: Illustration of Object query

Explanation: In Figure 1, any pair of classes are related to each other by one relationship. For this reason, they are omitted in the above path condition. In this query, Student is anchor class and the condition path is constructed on this class in order to fix the comparison predicate on the attribute Name of Lecturer. Since no specific property is requested from the query, in output all the attributes of Student is given.

Query 2: Students who are enrolled in courses Mathematic and Logic taken by lecturer of the department Electrical Engineering.

ANCHOR	Student
CONDITION	Student.Course: (Name= "Mathematics" **AND** Name="Logic")
	AND Student.Course.Lecturer.Department: Name= "Electrical Engineering"
OUTPUT	Student:*

Explanation: In the above query, the conjunction of path condition expressions is shown. It is also illustrated the conjunction of comparison predicates on the attribute Name of Course.

Query 3: Name and birthday of students and address of departments in which the courses enrolled by students is offered by some department.

ANCHOR	Student
CONDITION	Student.Course.Department: NOT NULL
OUTPUT	Student:(Name, Birthday); Student.Course.Department: Address

Explanation: In the CONDITION clause, the expression is evaluated to find *some* departments which offer the courses enrolled by students. According to the query language syntax, NOT NULL operator defines the quantifier *some*. In the OUTPUT clause, the output paths are constructed over anchor class Student.

Query 4: Name and birthday of students and start-date of courses enrolled by student and taken in department Electrical Engineering

ANCHOR	Student
CONDITION	Student.Course.Department:Name= "Electrical Engineering"
OUTPUT	Student:(Name, Birthday); Student.Course: Start-date

Query 5: Start-date of Courses taken in department Electrical Engineering and name and address of students enrolled in these courses

ANCHOR	Course
CONDITION	Course.Department:Name= "Electrical Engineering"
OUTPUT	Course: Start-date; Course.Student:(Name, Address)

Explanation: The anchor class is Course, and the results are defined by one property of this class and an output path.

Query 6: Name of students whose age is ≥ 21

ANCHOR	Student
CONDITION	Student: Age ≥ 21
OUTPUT	Student:Name

Query 7: Number of Courses enrolled by each student

ANCHOR	Student
CONDITION	
OUTPUT	Student.Course:COUNT

Explanation: The output path represents the count of courses for each student. This is performed by the aggregate function COUNT.

Query 8: Number of students who are enrolled in Course mathematics

ANCHOR	Student
CONDITION	Student.Course: Name="Mathematics"
OUTPUT	Student:COUNT

Explanation: In the OUTPUT clause, COUNT function is applied to anchor class Student.

Query 9: Number of students enrolled in course Mathematics started on 25 May offered by the department Electrical Engineering

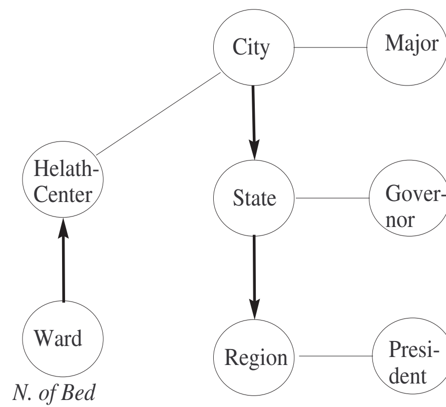


Figure 6: Schema of a database with regular object classes

ANCHOR	Course
CONDITION	Course: (Name="Mathematics" **AND** Start-date="25 May")
	AND Course.Department: Name="Electrical Engineering"
OUTPUT	Course.Student:COUNT

Query 10: In each course, give average age of students

ANCHOR	Course
CONDITION	
OUTPUT	Course.Student: AVG(Age)

Explanation: In the OUTPUT clause, the aggregate function AVG is applied to the attribute Age of the regular class Student joint through path from the anchor class Course.

The following queries are formulated on the schema of the Object database shown in Figure 6.

Query 11: Name of cities and Health-Center in the Western region of USA

ANCHOR	City
CONDITION	City.Region:Name="Western"
	AND City.State.Region.Country:Name="USA"
OUTPUT	City:Name, City.Health-Center:Name

Query 12: N.of Beds of ward maternity in each city of USA

ANCHOR	City
CONDITION	City.Health-Center.Ward:Name="Maternity"
	AND City.State.Region.Country:Name="USA"
OUTPUT	City:Name, City.Health-Center.Ward:N.ofBed

Query 13: Total N.of Beds of ward maternity in each city of USA

ANCHOR	City
CONDITION	City.Health-Center.Ward:Name="Maternity"
	AND City.State.Region.Country:Name="USA"
OUTPUT	City.Health-Center.Ward:SUM(N.ofBed)

Query 14: Find ward of maternity with maximum N.ofbeds in each city of USA

ANCHOR	City
CONDITION	City.Health-Center.Ward:Name="Maternity"
	AND City.State.Region.Country:Name="USA"
OUTPUT	City:Name, City.Health-Center.Ward:MAX(N.ofBed)

Query 15: Find ward of maternity with maximum N.ofbeds in each city of state California

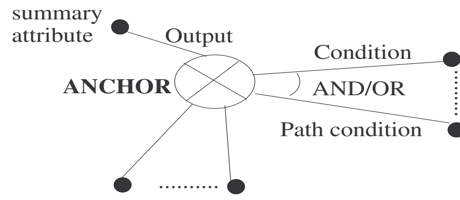
ANCHOR	City
CONDITION	City.Health-Center.Ward:Name="Maternity"
	AND City.State:Name="California"
OUTPUT	City:Name, City.Health-Center.Ward:MAX(N.ofBed)

Query 16: Find ward of maternity with maximum N.ofbeds in each region

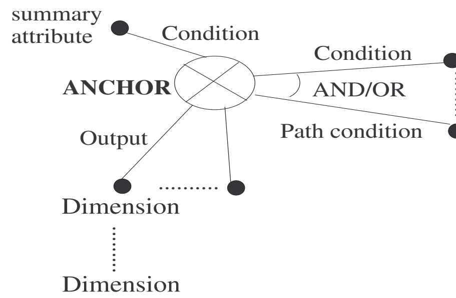
ANCHOR	Region
CONDITION	Region.State.City.Health-Center.Ward:Name="Maternity"
OUTPUT	Region:Name, Region.State.City.Health-Center.Ward:MAX(N.ofBed)

7.3 Composite Query

Two types of queries are considered. In Figure 7, they are indicated by Type I and Type II. Queries Type I ask in output the summary attribute and the condition expressions are defined on dimensions. Queries Type II ask in output



(Type I)



(Type II)

Figure 7: Illustration of Composite query

dimensions and the condition expressions are defined on summary attributes. In the following, we illustrate these types of queries by some examples and we describe in detail their characteristics. The following queries are formulated on the schema of OLAP database shown in Figure 2.

Query 17: Find region by age and population for females in $20 \div 40$ age-groups in USA regions

ANCHOR	Region, Age
FROM	Demographic_Group
CONDITION	Age **EQ** {20,40}
	AND Region.Country:Name="USA" **AND** Sex="Female"
OUTPUT	[Region, Age](Demographic_Group): Population

Explanation: This is a Type I query. It asks Population by Region and Age with certain conditions. The Region and Age are anchor classes and they are two dimensions of the composite class Demographic_Group. The result gives the summary attribute Population by anchor classes. Since the anchor classes

are components of the composite class Demographic Group, the latter class is indicated in the FROM clause, and the summary attribute (Population) is calculated for anchor classes and indicated in the OUTPUT clause. In this query, the tuples are restricted to a range of values on dimension Age, which varies from "20" up to "40", and indicated in the CONDITION clause by the **EQ** operator. Similarly, the dimension Sex is restricted to "Female".

Query 18: Find region and population for females between $20 \div 40$ in USA regions

ANCHOR	Region
FROM	Demographic_Group
CONDITION	Age **BETWEEN** {20,40} **AND** Sex="Female"
	AND Region.Country:Name="USA"
OUTPUT	[Region](Demographic_Group): Population

Explanation: Query 18 is formulated on Demographic_Group but only by dimension Region. Similar to Query 12, the values of the dimension Age is restricted to a range, and indicated by **BETWEEN** operator in the CONDITION clause. This differs from the previous query because the summary attribute should be aggregated in this range. Therefore, Demographic_Group is computed according to the conditional predicates and the dimensions Age and Sex are summarized.

Query 19: Find region with population $> 10M$ for females between $20 \div 40$ in USA regions

ANCHOR	Region
FROM	Demographic_Group
CONDITION	Age **BETWEEN** {20,40}
	AND Region.Country:Name="USA"
	AND [Region](Demographic_Group):Population $> 10M$
OUTPUT	[Region](Demographic_Group): Population

Query 20: Find region by age where female population is > 100000 and output the population of such regions.
(a)

ANCHOR	Region, Age
FROM	Demographic_Group
CONDITION	Sex="Female" **AND** [Region,Age](Demographic_Group):Population > 100000
OUTPUT	[Region, Age](Demographic_Group): Population

Explanation: In Query 19 and Query 20, a condition expression over summary measure (Population) is defined.

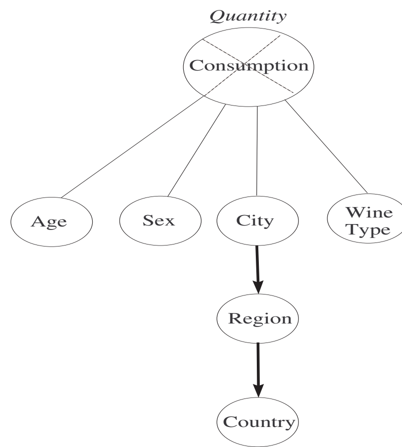


Figure 8: Schema of a composite class

Let us consider Query 20, and let the query asks the total population per region. Then, the predicate in the OUTPUT clause will be modified as follows:

ANCHOR	Region, Age
FROM	Demographic_Group
CONDITION	Sex="Female" **AND** [Region,Age](Demographic_Group):Population > 100000
OUTPUT	[Region](Demographic_Group): SUM(Population)

An example ofType II query expressed on the schema of the above mentioned composite class is indicated as follows.

Query 21: Find region by age where female population is > 100000 and output the population per city in these regions.

ANCHOR	Region, Age
FROM	Demographic_Group
CONDITION	Sex="Female" **AND** [Region,Age](Demographic_Group):Population > 100000
OUTPUT	Region.City: Population

Now let us consider Figure 8. This shows the schema of the composite class Consumption, which gives the *Quantity* of consumption of wine by Age, Sex, City, and Wine Type. These four classes are the dimensions of Consumption and the dimension City belongs to a classification hierarchy defined by three levels; i.e. City, Region, and Country. The following queries are given.
Query 22: Consumption of wine of female between age $20 \div 40$ in the cities of USA

ANCHOR	City
FROM	Consumption
CONDITION	Age **BETWEEN** {20,40} **AND** Sex="Female"
	AND City.Region.Country:Name="USA"
OUTPUT	[City](Consumption): Quantity

Query 23: Find Wine_Type by city consumed by female between age $20 \div 40$ where the consumption exceeds 100 gallons

ANCHOR	City, Wine_Type
FROM	Consumption
CONDITION	Age **BETWEEN** {20,40} **AND** Sex="Female"
	AND [City, Wine_Type](Consumption): Quantity > 100
OUTPUT	[City, Wine_Type](Consumption): Quantity

The above query with an additional condition on Wine_Type dimension is "Find quantity of Wine_Type Merlot consumed by city by female between age $20 \div 40$ where the consumption exceeds 100 gallons formulated as follows:

ANCHOR	City
FROM	Consumption
CONDITION	Age **EQ** {20,40} **AND** Sex="Female"
	AND [City](Consumption): Quantity > 100
	AND Wine_Type="Merlot"
OUTPUT	[City](Consumption): Quantity

Query 24 Let us consider the query "Find countries by Wine_Type where the consumption of male is >500 gallon and output the quantity of consumption per region in these countries" formulated as follows:

ANCHOR	Country, Wine_Type
FROM	Consumption
CONDITION	Sex="Male" **AND** [Country, Wine_Type](Consumption): Quantity > 500
OUTPUT	Country.Region:Quantity

Query 25: Total population

ANCHOR	*
FROM	Demographic_Group
OUTPUT	Demographic_Group:Population

Query 26: Population and average income by city

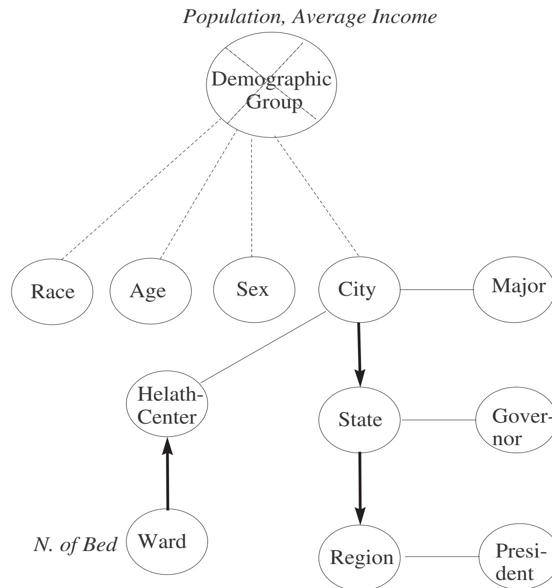


Figure 9: Example of composite-object class

ANCHOR	City
FROM	Demographic_Group
CONDITION	
OUTPUT	[City](Demographic_Group):Population; [City](Demographic_Group): Average-Income

7.4 Composite-Object Query

As we mentioned before, composite-object queries represent a class of queries that are formulated simultaneously on OLAP and Object data models. In this section, we show the proposed query language is powerful to express composite-object queries using the constructs discussed previously. For a clear explanation, we refer to the schema of the OLAP-Object database shown in Figure 9. Let us consider the following queries.

Query 27: Find name of governors of states where average income of population is > 20000

ANCHOR	State
FROM	Demographic_Group
CONDITION	[State](Demographic_Group):Average-Income> 20000
OUTPUT	State.Governor:Name

Explanation: The query asks the name of the governors of states. This invokes an object class from the schema of the Object model, while the condition expression is expressed over the composite object class Demographic_Group from the OLAP data model.

Query 28: N.of Beds of ward maternity of the cities where the population of female of age between 20 ÷ 40 is > 1500000

ANCHOR	City
TO	Demographic_Group
CONDITION	Age **BETWEEN** {20 40}
AND	Sex="Female" **AND** [City](Demographic_Group): Population > 1500000
AND	City.Health-Center.Ward:Name="Maternity"
OUTPUT	City.Health-Center.Ward:N.ofBed

Now, let us consider the schema of an OLAP-Object data model shown in Figure 10. The OLAP data model is characterized by the composite object Loan defined by the object classes (or dimensions) Date, Borrower, and Book. The primitive object class Date is a level of the classification hierarchy $Date \rightarrow Month \rightarrow Year$. The Object model defines Borrower], City, Author and Book object classes. Type-of-Loan are Staff-loan, Faculty-loan, and Common-loan.
Query 29: All borrowers that have loaned the book "Databases" and live in San Francisco

ANCHOR	Borrower
FROM	Loan
CONDITION	Book:Title="Databases" **AND** Borrower.City:Name"San Francisco"
OUTPUT	Borrower:ID

Explanation: This is a simple composite-object query, where the conditions are expressed over both databases.

Query 30: Type of Loan of book "Databases" and the name of authors

ANCHOR	Book
FROM	Loan
CONDITION	Book:Title="Databases"
OUTPUT	[Book](Loan):Type-of-Loan; Book.Author:Name

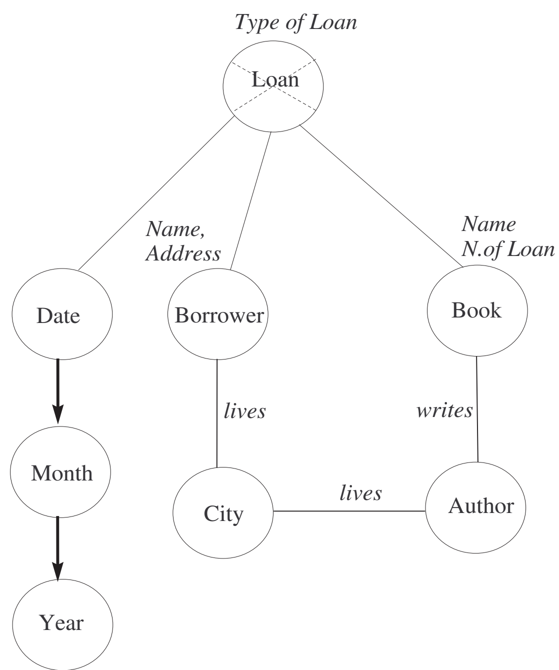


Figure 10: Example of composite-object classes

Explanation: The results in OUTPUT are evaluated over composite object class (Loan) and regular object class (Book).

Query 31: Authors of Books which are loaned more than 20 times in 2003 and loaned from faculty

ANCHOR	Book
FROM	Loan
CONDITION	Date.Month.Year="2003" **AND** [Book](Loan):Type-of-Loan="Faculty-loan" **AND** Book.N.ofLoan> 20
OUTPUT	Book.Author:Name

Query 32: All the Books which are loaned in San Francisco in september

ANCHOR	Book
FROM	Loan
CONDITION	Date.Month="September" **AND** [Book](Loan):N.ofLoan> 20 **AND** Borrower.City:Name="San Francisco"
OUTPUT	Book:ID

7.5 Composite-Composite Query

These queries are formulated on multiple OLAP databases. The composite construct can still be applied to answer queries, but for their complex composition the whole query is subdivided in query-subquery. In this way, the subquery fragment is resolved first and the result is used to give the final answer. In the following, we illustrate some query examples that refer to the schema shown in Figure 11.
Query 33: Consumption of wine in cities by sex where sales is > 10000$ in 2001

Query1

ANCHOR	City
FROM	Sales
CONDITION	Year="2001" **AND** [City](Sales):Amount> 10000
OUTPUT	[City](Sales):Amount

ANCHOR	City, Sex
FROM	Consumption
CONDITION	Year="2001" **AND** City **IN** Query1
OUTPUT	[City,Sex](Consumption):Quantity

Explanation: Query 1 gives the list of city where sales of wine is greater than 10000$ in 2001. This query is formulted on the composite object Sales. Then, the consumption of wine is calcutated for each city retrieved from Query 1.

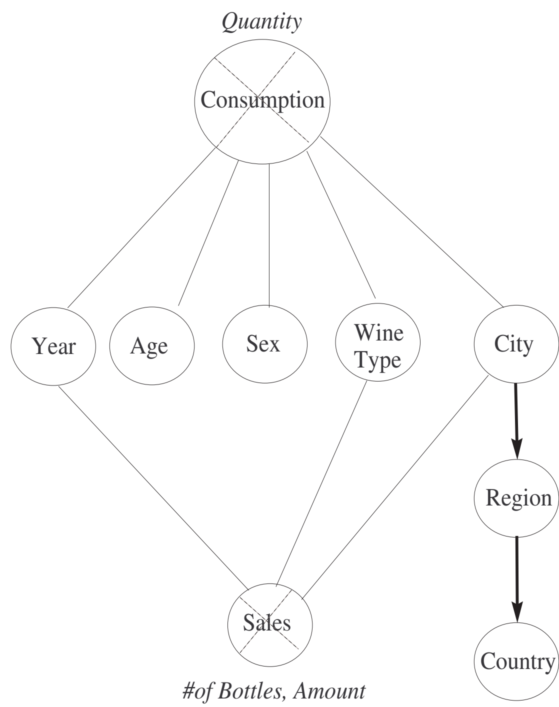


Figure 11: Example of composite-composite classes

In the following, similar queries are expressed. Their explanations are straightforward.

Query 34: Consumption of wine in states where in its cities sales is > 10000$ in 2001

ANCHOR	State
FROM	Consumption
CONDITION	Year="2001" **AND** State.City **IN** Query1
OUTPUT	[State](Consumption):Quantity

Explanation: Query 1 in this query corresponds to Query 1 in the previous query.

Query 33: Consumption of wine by age and sex in 2003 in USA where in the same year the sales of wine is > 100000 bottles

Query1

ANCHOR	Sales
FROM	
CONDITION	Year="2003" **AND** City.Region.Country="USA"
	AND (Sales):#ofBottles> 10000
OUTPUT	(Sales):#ofBottles

ANCHOR	Age, Sex
FROM	Consumption
CONDITION	Year="2003" **AND** Sales **IN** Query1
	AND City.Region.Country="USA"
OUTPUT	[Age,Sex](Consumption):Quantity

Query 34: Consumption of wine by city, age and sex in 2003 in USA where in the same year the sales of wine in each city is ≥ 100000$

Query1

ANCHOR	City
FROM	Sales
CONDITION	Year="2003" **AND** City.Region.Country="USA"
	AND [City](Sales):#ofBottles> 10000
OUTPUT	[City](Sales):#ofBottles

ANCHOR	City, Age, Sex
FROM	Consumption
CONDITION	Year="2003" **AND** City **IN** Query1
	AND City.Region.Country="USA"
OUTPUT	[City,Age,Sex](Consumption):Quantity

Query 35: Sales of red wine in the regions of USA, where the consumption of male in 2003 is $\geq$ 300 gallon

Query1

ANCHOR	Region
FROM	Consumption
CONDITION	Year="2003" **AND** Region.Country="USA"
	AND Sex="Male"
	AND [Region](Consumption):Quantity ≥ 300
OUTPUT	[Region](Consumption):Quantity

ANCHOR	Region
FROM	Sales
CONDITION	Region **IN** Query1
	AND Year="2003"
	AND Wine_Type="Red"
OUTPUT	[Region](Sales):(#ofBottles, Amount)

8 Extended Concepts

8.1 Back Reference

In this section, we consider queries which express condition caused by ellipsis in natural language (e.g., their, its, etc.). This class of queries can be answered by a specialized version of paths discussed in Section 6. It is called *back reference* path. This represents that two condition expressions should be compared with each other. Let us consider Figure 10, and ask the following query.

Query 36: Give the name of books, of which the authors live in the same city of borrower.

To answer this query, city of author should be compared with city of borrower. Therefore, two condition expressions are needed. For a correct formulation of query, we should retrieve books from the first expression such that on *these books* the second expression can be evaluated. A solution to this is to create a reference for Book and use this in both above mentioned expressions. The query is answered by a back reference path as follows.

Book(x).Author.City:Name=Name(City.Borrower.Book(x)

Note that the relationship between Borrower and Book is Loan. The term Book(x) in both sides of condition expression serves as the linking variable. The

back reference is appropriate to express condition expressions on the values of a class (Name of City) which should be matched over two different paths.

ANCHOR Book
FROM Loan
CONDITION Book(x).Author.City:Name=Name(City.Borrower.Book(x))
OUTPUT Book:Name

The syntax of back reference is indicated below.

```
<anchor-path-ref> ::=
<primitive class>'('variable')'{'.'<anchor-path>}|
<regular class>'('variable')'{'.'<anchor-path>}

<anchor-path-backref> ::=
{<anchor-path>'.'}<primitive class>'('variable')'|
{<anchor-path>'.'}<regular class>'('variable')'

<variable>::= <primitive value>
```

The back reference construct is useful to formulate *recursive* queries, where a class is related to itself by a simple association. For instance, a regular class called Person is related to itself by Friend relationship. Let ask the following query on this schema:

Query 37: Find people who have at least one friend with the same first name, and return full name of person and his friend.

This query is easily formulated as follows:

ANCHOR Person
CONDITION Person(x):Friend-Name=Friend-Name(Person(y)(friend)Person(y))
OUTPUT Person(x):(First-Name,Last-Name);Person(y):(First-Name,Last-Name)

8.2 ISA

The proposed query language supports the subclass-superclass relationships in OLAP-Object data model. A subclass is a specialization of its superclass and inherits all the attributes associated with its super-classes. Then, a regular object class O_i isa O_j means that the instances of O_i are also instances of O_j.

In the proposed query language, *isa* is not considered as a "distinct" association between regular object classes. In other words, if a regular object class O_i is a specialization of regular object class O_j and it is invoked by a query, then the query is evaluated on its super-class O_j.

For instance, let us consider *Worker_Student* be a subclass of *Student* in Figure 1, and a query asks the name and type of work of students enrolled in course Mathematics. According to the query syntax, the anchor class is *Worker_Student*, but the path condition expression is formulated on *Student*; that is Student.Course:Name="Mathematics". Similarly, the output is defined on *Student* (see below).

ANCHOR	Worker_student ISA Student
CONDITION	Student.Course: Name="Mathematics"
OUTPUT	Student:COUNT

The syntax of anchor class is extended to enclose isa association as follows.

```
<anchor-class>::= <anchor-class-simple>
|'['<anchor-class-composite>']'

<anchor-class-simple>::= <primitive class> {',' <primitive class>}
 |<primitive class> {',' < regular class>}
 |< regular class>{',' <primitive class>}
 |< regular class>{','<regular class>}
|<regular class> ISA <generic-regular class> {','<anchor-class-simple>}

<anchor-class-composite>::= <composite class>
<generic-regular class>::= <regualr class>
```

9 Conclusions

In this paper, we proposed a data model that combines the main characteristics of OLAP and Object data models in order to achieve their functionalities in a common framework. We classified three different object classes: primitive, regular and composite. We investigated the well-formed composite objects and we studied their summarization semantics. Then, we defined a query language which uses the path concept in order to facilitate data navigation and data manipulation. The main feature of the proposed language is an anchor. It allows us to fix dynamically an object class (primitive, regular or composite) along the paths over the OLAP-Object data model for expressing queries. The power of the proposed query language was investigated through multiple query examples. The semantic of different clauses and syntax of the proposed language were investigated.

References

[1] Arbor Software Corporation. *Arbor Essbase.* http://www.arbosoft.com/essbase.html, 1996.

[2] Agrawal, R., Gupta, A., Sarawagi, S.: Modeling Multidimensional Databases. Proceedings of the 13th International Conference on Data Engineering - ICDE'97, pp. 232–243, 1997.

[3] Bancilhon, F., Cluet, S., Delobel, C.: A Query Language for an ObjectOriented Database System. In 2nd International Worshop on Database Programming Languages (DBPL), pp. 301–322, 1989.

[4] Chan, P., Shoshani, A.: SUBJECT: A Directory Driven System for Organizing and Accessing Large Statistical Databases. Conference on Very Large Data Bases, pp. 553–563, 1981.

[5] Codd, E. F., Codd, S.B., Salley, C.T.: Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT mandate. Technical report 1993.

[6] Colliat, G.: OLAP, Relational, and Multidimensional Database Systems. ACM Sigmod Record, 25(3):64–69, 1996.

[7] Gardarin, G., Valduriez, P. ESQL2: An Object-Oriented SQL with F-Logic Semantics , IEEE International Conference on Data Engineering, Phoenix, pp. 320–327, 1992.

[8] Gray, J., Bosworth, A., Layman,A., Pirahesh, H.: Data cube: a Relational Aggregation Operator Generalizing Group-by, Cross-tabs and Subtotals. Proceedings of 12th IEEE International Conference on Data Engineering, pp. 152–159, 1996.

[9] Gyssens, M., Lakshmanan, L.V.S.: A Foundation for Multidimensional Databases. Conference on Very Large Data Bases - VLDB'97, pp. 106–115, 1997.

[10] Fishman, D., Beech D., Cate, H., Chow, E., et al.: IRIS: An Object-Oriented Database Management System, ACM Transaction on Information Systems, 5(1), 48–69, 1987.

[11] Frohn, J., Lausen, G., Uphoff, H.: Access to Objects by Path Expressions and Rules. Proceedings of 20th International Conference on Very Large Databases-VLDB, pp. 273–284, 1994.

[12] MicroStrategy, Inc. *MicroStategy's 4.0 Prodct Line.* http://www.strategy.com, 1997.

[13] Il-Yeol Song, Jones T. H., and Park, E. K.: Binary relationship imposition rules on ternary relationships in ER modeling. Proceedings of the second international conference on Information and knowledge management, CIKM '93, Washington, D.C., United States, ACM Press, pp. 57–66, 1993.

[14] Il-Yeol Song, Jones, T.H. : Ternary relationship decomposition strategies based on binary imposition rules. 11TH International Conference on Data Engineering, pp. 485–492, 1995.

[15] Jones, T. H., Il-Yeol Song: Analysis of Binary/Ternary Cardinality Combinations in Entity-Relationship Modeling. Data and Knowledge Engineering, 19(1): 39-64 (1996).

[16] Kifer, M., Lausen, G., Wu, J.:Querying Object-Oriented Databases. In Proceedings of the ACM SIGMOD international conference on Management of data, pp. 393–403, 1992.

[17] Kifer, M., Lausen, G., Wu, J.: Logical Foundations of Object-Oriented and Frame-Based Languages, Journal of the ACM, 42, 1995.

[18] Kim, W.: A Model of Queries for Object-Oriented Databases. In Proceedings of the International Conference on Very Large Databases, pp. 423–432, 1989.

[19] Lenz, H.-J., Shoshani, A.: Summarizability in OLAP and Statistical Data Bases. In Proceedings of Ninth International Conference on Scientific and Statistical Database Management-SSDBM, Ioannidis. Y. E. and Hansen D. M. (Eds.), August 11-13, 132-143, Olympia, Washington, USA, IEEE Computer Society Press, 1997.

[20] Red Brick systems, Inc. *Red Brick Warehouse 5.0.*. http://www.redbrick.com, 1997.

[21] Shoshani, A.: OLAP and Statistical Databases: Similarities and Differences. Proceedings of 16th ACM Symposium on Principles of Database Systems, pp. 185–196, 1997.

[22] Teorey, T.J.: Database Modeling ad Design. The Entity Relationship Approach. Morgan-Kauffman, 1990.

[23] Zaniolo, Z: The Database language GEM. In Proceedins of the ACM SIGMOD Conference on Management of Data. pp. 207–218, 1983.

Table 1: Symbols used in the BNF notation

<table>
<tr><th>Symbols</th><th>Semantic</th></tr>
<tr><td>::=</td><td>Indicates the equivalence of the left hand side
of a statement to the right hand side of the statement.</td></tr>
<tr><td>|</td><td>Indicates an alternative.</td></tr>
<tr><td><></td><td>Encloses schema construct.</td></tr>
<tr><td>{ }</td><td>Encloses an optional construct that can appear zero o more times.</td></tr>
<tr><td>[ ]</td><td>Encloses an optional construct that can appear no more than once.</td></tr>
<tr><td>' '</td><td>Encloses terminals.</td></tr>
<tr><td>...</td><td>Encloses continuation.</td></tr>
<tr><td>“abc-semicolonlist”</td><td>It represents a sequence of one or more “abc” in which each
pair of adjacent “abc” is separated by a semicolon.</td></tr>
</table>

APPENDIX A

This appendix includes the table of symbols and the basic structures used to define the syntax of the proposed query language.

Basic Structures

<digit>::= 0|1|...|9

<lower-case-letter>::= a|b|...|z

<upper-case-letter>::= A|B|...|Z

<comparison-symbol>::= =| < | > | = | =

<assignment-symbol>::= :=

<sign>::= + | -

<special-symbol>::=! | | $ | % | & | @ |# | (|) | *| + | - | _

<letter>::= <lower-case-letter> |<upper-case -letter>

<boolean-value>::= TRUE | FALSE

<unsigned-integer>::= <digit> <digit>

<integer>::= [<sign>] <unsigned-integer>

<real-number>::= [<integer>].<unsigned-integer> [(E|e) <sign> <unsigned-integer>]

<numeric>::= <integer> | <real-number>

```
<character>::= '<letter>' | '<digit>' | '<special-symbol>'| ' '

<string>::= '<character> <character>'

<case-string>::= <lower-case-string> | <upper-case-string>

<lower-case-string>::= <lower-case-letter> | <lower-case-letter> <letter>
| <digit> | '-'

<upper-case-string>::= <upper-case-letter> | <upper-case-letter> <letter>
| <digit> | '-'

<constant>::= <numeric> | <string> | <character>
```

Types

```
<types>::= <base-type> | <template-type> | <user-defined-type>

<base-type> ::= <char-type> | <boolean-type> | <integer-type> | <floating-type>
| <string-type>

<char-type> ::= CHAR

<boolean-type> ::= BOOLEAN

<integer-type> ::= <signed-integer> | <unsigned-integer>

<signed-integer> ::= <signed-long-integer> | <signed-short-integer>

<signed-long-integer> ::= LONG

<signed-short-integer> ::= SHORT

<unsigned-integer> ::= <unsigned-long-integer> |<unsigned-short-integer>

<unsigned-long-integer> ::= UNSIGNED LONG

<unsigned-short-integer> ::= UNSIGNED SHORT

<floating-type> ::= FLOAT | DOUBLE

<string-type> ::= STRING

<template-type> ::= <set-type> | <list-type>| <array-type>

<set-type> ::= SET '<' <types> '>'
```

<list-type> ::= LIST '<' <types> '>'

<array-type> ::= <array-tag> '<' <postitive-integer> ',' <types> '>'
| <array-tag> '<' <types> '>'

<array-tag> ::= ARRAY

<postitive-integer> ::= [+] <unsigned-integer>

<user-defined-type>::= <user-defined-type-name> <simple-type>

<user-defined-type-name> ::= <case-string>

Attributes

<object-attributes> ::= <object-attribute> {<object-attribute }

<object-attribute>::= <simple-attribute> |<tuple-attribute>

<simple-attribute> ::=
ATTRIBUTE <simple-attribute-name> ':'<single-or-set-or-list><class-type>

<simple-attribute-name> ::= <attribute-name>

<attribute-name> ::= <case-string>

<tuple-attribute>:=
ATTRIBUTE <tuple-attribute-name> ':' <single-or-set-or-list><component-classes-type>

<tuple-attribute-name> ::= <attribute-name>

<single-or-set-or-list> ::= <set-of> | <list-of> | <null>

<component-classes-type> ::= <class-type> ','<class-type>

<class-type> <types>

APPENDIX B

Object Classes

<object class> ::=<primitive object class> | <regular object class>
| <composite object class>

Primitive object class

<primitive object class> ::= PRIMITIVE OBJECT CLASS <object class name>
<attibute list>

<primitive object class name> ::= <class name>

<class name> ::= <case-string>

<attibute list> ::= <null>

Regular object class

<regular object class> ::= REGULAR OBJECT CLASS <object class name>
<regular object identifier> <attibute list>

<regular object class name> ::= <class name>

<class name> ::= <case-string>

<attibute list> ::= <object-attibutes>

Composite Object class

<composite object class>::= COMPOSITE OBJECT CLASS <object class name>
<object classes> <attibute list>

<object class name>::= <class name>

<class name>::= <case-string>

<object classes>::= <object class> {',' <object class>} — <object-class-category>
',' <object class> | <object-class-category> {',' <object class-category>}

<object-class-category>::=<regular class> '→'<regular class> {'→'<regular
class>}
<attibute list>::= NULL | <object attibutes>

APPENDIX C

Query Syntax

```
<query-item> ::= <anchor-statement><from-statement> <condition-paths-statement>
<output-statement>


<anchor-statement> ::= ANCHOR <anchor-class>
<anchor-class>::= <anchor-class-simple>
|’[’<anchor-class-composite>’]’

<anchor-class-simple>::= <primitive class> {’,’ <primitive class>}
 |<primitive class> {’,’ < regular class>}
 |< regular class>{’,’ <primitive class>}
 |< regular class>{’,’<regular class>}
|<regular class> ISA <generic-regular class> {’,’<anchor-class-simple>}

<anchor-class-composite>::= <composite class>
<generic-regular class>::= <regualr class>

<from-statement>::=FROM <anchor-class-composite>


<condition-paths-statement> ::= CONDITION <condition-expressions>
<condition-expressions> ::= <conditions> | <path-conditions>|<null>

<conditions> ::= <atomic-condition> | <conditions> <conjunction> {<conditions>}

<atomic-condition> ::= <expression>|<expression> <conjunction> <expression>

<expression>::=<anchor-path>’:’<subexp>|<back-reference-expression>
|<object class>’:’  {’(’<subexp> <conjunction> <subexp>’)’}
|<object class>’:’<attribute name><null-operator>
|<object class><range-operator> <set-of-values>
|<object class><in-operator><query-item>

<subexp>::=<attribute name><comparison-operator> <primitive value>

<back-reference-expression>::=
<anchor-path-ref>’:’<attribute name>
<comparison-operator> <attribute name> ’(’<anchor-path-backref>’)’
```

```
<conjunction> ::= AND | OR
<comparison-operator>::= '=' | '<' | '>' | '=' | '=' | '!='
<primitive-value> ::= <numeric> | "<string>"
<null-operator> ::= NULL | NOT NULL
<range-operator> ::= EQ | BETWEEN
<set-of-values> ::= <set-of-numerics> | <set-of-strings>
<set of numerics> ::= {<numeric> ',' <numeric> }
<set-of strings> ::= {<string> ',' <string>}
<in-operator> ::= IN | NOT IN

<path-conditions>::=
<path-condition> | {<path-condition> <conjunction> <path-conditions>}
<path-condition>::=<anchor-path-1-condtion> | <anchor-path-condition>

<anchor-path-1-condition>::=
<anchor-with-composite><comparison-operator><primitive value>
|<anchor-with-composite><null-operator>
|<anchor-with-composite><in-operator><query-item>

<anchor-with-composite>::=
'['<anchor-class-simple>']''('<composite class>')'':'<attribute-name>

<anchor-path-condition> ::=< anchor − path >':'< subexp >
|<anchor-path>':'{'('<subexp> <conjunction> <subexp>')'}
|<anchor-path>':'<attribute name><null-operator>
|<anchor-path><in-operator><query-item>

<anchor-path>::=
<primitive class>{'.'<primitive class>|'.'<regular class>}|
<regular class>{'.'<primitive class>|'.'<regular class>}

<anchor-path-ref> ::=
<primitive class>'('variable')'{'.'<anchor-path>}|
<regular class>'('variable')'{'.'<anchor-path>}

<anchor-path-backref> ::=
{<anchor-path>'.'}<primitive class>'('variable')'|
{<anchor-path>'.'}<regular class>'('variable')'

<variable>::= <primitive value>


<output-statement>::=OUTPUT <anchor-path-item-semicolonlist>
<anchor-path-item>::= <anchor-path-1>|<anchor-path-2>|<anchor-path-3>
```

```
<anchor-path-1>::=<anchor-with-composite>|<anchor-path>':'<attribute-item>
|<anchor-path>':''ID'
<anchor-with-composite>::='['<anchor-class-simple>']'
'('<composite class>')'':'<attribute-name>
<anchor-path> ::=
<primitive class>{'.'<primitive class>|'.'<regular class>}|
<regular class>{'.'<primitive class>|'.'<regular class>}
<attribute-item>::=<attribute-name>|'*'
<attribute-name>::=
'('<attribute name>','<attribute name>{','<attribute name>}')'
```

```
<anchor-path-2>::=
<anchor-path>':'<Agg-function>'('<agg-attribute>')'|<anchor-path>':'COUNT
|'['<anchor-class-simple>']' '('<composite class>')
':'<Agg-function>'('[<attribute name>]')'
<agg-attribute>::=null| <attribute name>
<Agg-function>::=SUM|AVG|MIN|MAX
```

```
<anchor-path-3>::=<anchor-path> IN <query-item>
```