Natural quantification and genitive case constructs in COOL: A composite object oriented declarative database language

J. Bradley

Department of Computer Science

University of Calgary

Calgary, Alberta, Canada

Abstract COOL is a composite object-oriented declarative data base language for use with N^2F relational data bases. COOL uses a construct called a genitive relation to enable unambiguous specification and quantification, both conventional and natural, of relationships. A genitive relation is used in a manner corresponding to the genitive case with noun objects in natural language. Instead of formal genitive relation name syntax, a genitive case alias can be used to promote ease of expression construction.

1.0 INTRODUCTION

The object-oriented approach to database management [2, 5, 7, 8, 9, 10, 11] evolved from the object-oriented approach used with programming languages such as C^{++} and Smalltalk, and is finding use in many application areas.

Although rich in properties, the object oriented model is quite complex and does not have the strong theoretical foundation of the conceptually simpler relational model [6] - nor is there any agreement on what exactly constitutes an object-oriented model [5]. But because of the firm foundation underlying the relational ap-

proach and its wide use, there has evolved considerable support in the data base research community for extending the relational model to enable it to support objects, both from a point of view of the structure of objects and the behaviour of objects.

1.1 Object oriented extensions to the relational model

The most widely accepted extension paradigm involves removing the requirement for normalized relations on which the conventional relational approach is based [6] giving us non nornal form or N^2F relations [1].

An N^2F relation can have collection attributes, both sets and lists. While in theory a set attribute of an unnormalized relation could have a value that is a set of tuples, it seems desirable and sufficient to restrict N^2F relations to attributes that contain a single stored atomic value, attributes that contain stored sets or lists of atomic values, structure attributes (attributes with composite type) corresponding to program defined hierarchically structured types (such as DATE or ADDRESS), and attributes whose values are not stored but are derived, where the derivation of an attribute value is via a function acting on stored values. The equivalent of structure array attributes is not allowed. It is in this sense that N^2F relations are used in this paper.

Extended versions of the conventional relational declarative languages are needed to manipulate an N^2F relational data base. The best known prototypes embodying this approach are IBM's STARBURST [11], which uses an extension of SQL, and the POSTGRES system [12], which uses an extension of QUEL.

1.2 Two types of declarative languages for data base manipulation

It is the thesis of this paper that there are two fundamentally different approaches to declarative languages. One approach is the relation-oriented approach, embodied in the languages of the conventional relational approach, such as DSL Alpha, SQL, and relational algebra and QUEL [6]. The other approach is what might be called a composite object-oriented approach. Currently, there are no implemented examples of this second type of language. This paper deals with a proposal for a language of the second type, called COOL - or Composite Object Oriented Language. COOL is soundly based on a set theoretic tuple calculus [3]. Full details of COOL cannot be given in this paper but are to be found in [4].

1.3 The project database

In the object-oriented approach, an object has a unique identity that is independent of any values it contains [5]. Such system generated unique keys are assumed in the N^2F data model used in this paper.

Our N^2F data base is assumed to allow all of the attribute types commonly fround in OO databases - but no repeating group equivalents. Many of these features can be illustrated by the database definition in Figure 1 for the project database, which concerns document management.

```
Document:  <
    doc#:        Document;
    title:       STRING;
    revised:     DATE;
    topic:       STRING;
    nchapters()  INTEGER FUNCTION;
```

```
    keyword:    SET[STRING];
    authlist:   LIST[Author];
    chaplist:   LIST[Chapter];
    activlist:  LIST[Activity];  >
   Chapter:   <
    chap#:      Chapter;
    doc#:       Document;
    ctitle:     STRING;
    npages:     INTEGER;
    ndiagram:   INTEGER;>
Create genitive relation alias
     Document.chaplist*Chapter     Document's Chapters
                                   Document's Chapter objects
      Chapter.doc#*Document         Chapter's Document
Activity:  <
    act#:       Activity;
    doc#:       Document;
    auth#:      Author;
    payment:    INTEGER; >
Author: <
    auth#:      Author;
    doclist:    LIST[Document];
    actlist:    LIST[Activity];
    pname:      STRING;
    position:   STRING;        >
```

Figure 1

The main object type is **Document**, where each object represents a document. A document can have many chapters, with each chapter represented by a **Chapter** object. An Author can author many documents and a document can be authored by many Authors. An Author is represented by a **Author** object and the object **Activity** (or author activity) enables the resulting many-to-many relationship between **Document** and **Author** objects.

The genitive relation specification in Figure 1 will be explained presently. Note that the system generated object identifier for each object type is specified in Figure 1 using the object type. Thus the object identifier **doc#** must have the type **Document**, and **chap#** the type **Chapter.**

The convention of using a relation name beginning with an upper case letter, and an attribute name beginning with a lower case letter is used throughout, with both SQL and COOL expressions.

2.0 COOL CONCEPTS, SEMANTICS AND SYNTAX

2.1 COOL manipulation of atomic objects

To retrieve attribute values from each object instance of a given type that complies with a simple condition, the semantics and essential syntax of SQL suffices, except where a collection attribute is involved in a condition. Thus the following could specify the COOL retrieval: Get the titles of chapters with more than 10 pages:

```
select ctitle from [each] Chapter [object]
where pages > 10;
```

The expression is enriched with the non essential words, each and object, to help clarify semantics. Like SQL, COOL does not normaly use prior delared tuple or range variables. In the semantics for

the expression, the term Chapter is a default range variable ranging over **Chapter** tuples. Attributes are selected from each specific range variable value for which the where-condition holds.

2.2 Language semantics for 1:n relationships

Here we deal with the common case of retrieval of attribute values from each tuple of object type A that satisfies a condition that can involve one or more related tuples of type B, where A and B are in a 1:n relationship.

Consider the portion of the database definition in Figure 1 that includes the 1:n relationship between **Document** and **Chapter** objects. In the object **Chapter**, the attribute **chap#**, although system generated, is taken as naming the object identifier for a chapter of a document. Accordingly, the collection attribute **chaplist** in **Document**, which is a list of **chap#** values, gives a list of the object identifiers of the chapters of that document, so that the type of **chaplist** must be LIST[**Chapter**]. Furthermore, in a **Chapter** object, there is an attribute **doc#** with the type **Document**, that is, its value must be a **Document** object identifier. The attributes **chaplist** and **doc#** are reference or relationship attributes. They are used instead of the primary and foreign keys of the conventional relational approach, and precisely define the 1:n relationship between the objects **Document** and **Chapter.**

Now suppose we are dealing with a specific **Document** object. To specify a quantity of its chapters, that is, a quantity of its related **Chapter** objects that complies with a given condition, the construct needed must specify

(a) A quantifier, and

(b) The set of related **Chapter** objects

(c) The condition the specified quantity of objects must satisfy or more formally:

<quantifier><related objects><(condition)>

an expression that will have the value true or false. In the syntax of a computer language, the quantifier symbol could be any common quantifier notation, such as **for all** [**its**], or **for each** [**of** [**its**]]. In the simplest case, the condition specification would involve the attribute name, a relational operator, and a literal value, such as: (page = 10).

To specify the <related objects> term, where in English the genitive expression "chapters of document" would be used, a precise relationship specification is needed, since there could be more than one relationship between two object types. To explicitly specify the relevant object instances of the relationship the reference list **chaplist** can be used in a flexible and rich construct that specifies what we propose to call a genitive relation.

Semantically, what is needed is an unambiguous specification of the current document's **Chapter** tuples, that is, an unambiguous specification of a genitive relation. Since for a given **Document** tuple, the chaplist attribute specifies the set of identifiers of that **Document**'s chapters, any construct listing chaplist and **Chapter** can serve to unambiguously specify of the current **Document**'s **Chapter** tuples. We therefore propose the term

[Document.]chaplist*Chapter

to specify the related **Chapter** tuples of the current **Document** tuple, that is, a genitive relation. [The complete syntax for

specifying a COOL where-expression involving a genitive relation is shown in the appendix.] The use of a genitive relation is illustrated by the retrieval:

Get the document title for each database document with at least 4 chapters with more than 10 pages.

In this case the required natural quantifier is **for at least 4**:

```
select title from [each] Document [object]
where where topic = 'databases'
and for at least 4 [Document.]chaplist*Chapter [objects]
                                                (pages > 10)
```

The term chaplist*Chapter denotes the set of, that is, the derived relation holding, **Chapter** tuples that are referenced in **chaplist** in the current **Document** object. Thus a genitive relation can be looked at as the join of the list (regarded as a unary relation) **chaplist** and the relation **Chapter**, using the object identifier as the join field. The relation chaplist*Chapter clearly also is the set of child **Chapter** tuples for the **Document** tuple containing the **chaplist** value used. This set of child tuples can also be expressed using its full path name:

```
Document.chaplist*Chapter [objects].
```

It should be clearly understood that a genitive relation such as chaplist*Chapter is a relation. Since a relation name in SQL and in COOL serves as an implicit range or tuple variable, a genitive relation name also serves as an implicit range variable in COOL, and the COOL expression above must be interpreted in this sense. In the next section there is a brief discussion of genitive relations and tuple variables in the context of predicate calculus.

If the quantifier in the retrieval above is changed, to **for most** or **for a majority of**, for example, only the quantifier in the predicate needs be changed, as in:

```
select title from [each] Document [object]
where topic = 'database' and
for a majority of [Document.]chaplist*Chapter [objects]
                                    (pages > 10);
```

The above quantifier retrieval examples involved retrieving data from a parent object, given conditions in an associated child object, with a 1:n relation. In such expressions we used a reference list, such as chaplist, to specify the needed genitive relation. The converse case involves retrieval of a child, given conditions for the parent. In this case we use the syntax variable <reference>, which specifies a reference (such as **doc#**) to the parent entity, to costruct the relatively trivial genitive relation. This is illustrated by the retrieval:

Get the names of chapters with more than 10 pages in documents on databases.

```
select ctitle from Chapter
where (npages > 10)
   and for its one [Chapter.]doc#*Document [object]
                                    (topic = 'databases');
```

The formal COOL syntax for the expressions in this section is given in the appendix.

2.3 Use of alias genitive relation names

A genitive relation can additionally be defined in the data base definition has having an alias that is convenient to remember. Suppose we define:

Create genitive relation alias:

Document.chaplist*Chapter	Document's Chapters/ Document's Chapter objects
Chapter.doc#*Document	Chapter's Document

as in Figure 1. In that case the retrieval expressions above could be rewritten:

Get the document title for each database document with at least 4 chapters with more than 10 pages.

```
select title from [each] Document [object]
where where topic = 'databases'
and for at least 4 Document's Chapter objects (pages > 10);
```

Get the names of chapters with more than 10 pages in documents on databases.

```
select ctitle from Chapter
where (npages > 10)
and for its Chapter's Document object (topic = 'databases');
```

Note that since the genitive case construct is of fundamental imporatnce in natural languages in dealings with complex objects, it seems sensible to introduce it into computer languages for dealing with objects.

2.4 Many to many relationships

In the object-oriented approach, a many-to-many relationship can involve either two objects, in the case where there is no intersection data, or three objects, for the case where the third object type concerns intersection data. Consider now the many-to-many relationship between a document and an Author, where one or more Authors can author one or more documents.

In the case of no intersection data, the object-oriented approach simply treats the many-to-many relationships as two symmetric one parent for many children relationships. Thus the constructs for retrieving a parent given conditions involving the children, as developed in the previous section, may be used with this kind of many-to many relationship - specifically we may use the <where-expression> syntax given in the previous section in the appendix. As an example, suppose the retrieval:

Get the each documents about databases with at least two authors who are systems analysts.

This can be expressed:

```
select title from [each] Document [object]
where (topic = 'databases')
  and for at least 2 [Document.]authlist*Author [objects]
                        (position = 'systems analyst')
```

A converse retrieval would be:

Get the name each engineer who has never authored any documents about computers.

```
select pname from [each] Author [object]
where (position = 'engineer')
                  and for no doclist*Document [objects]
                                        (topic = 'computers');
```

In the case of intersection data, there are simply two symmetric 1:n relationships, referring to Figure 1, between **Document** and **Activity**, and between **Author** and **Activity**. These can be handled like normal 1:n relationships. However, some retrievals will require the use of all three objects, and therefore a nesting of

quantified expressions. The where-expression formalism given in the appendix covers unlimited nesting of quantified cross references. For example, consider the retrieval:

> Retrieve the name of each document about computers written by authors who were all systems analysts, all of whom were paid more than $100.

A further level of nesting is needed. By nesting the constructs already developed, in compliance with the <where-condition> syntax above, we get the expression:

```
select title from [each] Document [object]
where topic = 'databases'
  and for each [Document.]authlist*Activity [objects]
   (payment > 100 and for the [one] auth#*Author [object]
                                  (position = 'systems analyst'))
```

It is useful to compare this expression with the equivalent relational SQL expression, if the database were relational (i.e. all collection attributes are omitted):

```
sql:   select title from Document
       where topic = 'databases' and doc# not in
         (select doc# from Activity where
         payment not > 100 or auth# not in
                       (select auth# from Author
                        where position = 'systems analyst'));
```

The required negation of the implicit existential quantifier means that we have to negate a complex predicate, and use De Morgan's rules for negation of compound expresssions to do it correctly.

2.5 Composite objects and inheritance

COOL has facilities for defining and retrieving composite object types, that is, the equivalent of an XNF view. It also has facilities for concentrating composite object types once retrieved or defined. COOL also has extensive facilities for handling generalization structures, that is, structures involving IS-A relationships and inheritance. Furthermore, it has facilities for aggregation, with the use of aggregastion functions like count(), sum(), max(), min(), avg() and stdv(); see the term <aggregation function> in the appendix.

A discussion of these facilities is necessarly lengthy and is to be found in a separate paper [4]. It is sufficient for the purposes of this paper to ensure the reader that they exist and that COOL is indeed a language that can manipulate composite objects.

2.6 Updating

This paper is not concerned with languages for N^2F relational data base updating, which is a topic to be addressed in a later paper.

3.0 SUMMARY

Language constructs for tuple selection and composite object formation have been presented for COOL, an object-manipulation language designed for manipulation of an N^2F relational data base. This language makes extensive use of a construct we have called a genitive relation. Genitive relations are used to manipulate quantified relationships between objects by means of expressions that specify only components of the object involved. This is in contrast to SQL, which manipulates relationships between objects in a set-theoretic manner that requires specification of which sets of objects are involved and which are not involved.

COOL is not proposed as a replacement for SQL, but as a subset of SQL, for use where the data base has a distinct object-orientation. Currently, there are no implemented examples of an object-manipulation type data base language. However, GenRel, a project at the Space Information Science Laboratory at the Uniersity of Calgary to develop a prototype composite object-oriented N^2F genitive relational base system, will embody both SQL and COOL.

Appendix

```
<where-expression>:=
        <condition> [<<op> <quantifed xreference>> ...]
<quantified xreference>:= <quantifier><genitive
                                    relation>[(<where expression>)]
<genitive relation>:=
       [related][<referencing object>.]<reference attribute>*
             <referenced object>[object[s]]
<op>:= AND/OR
<reference attribute>:= <reference>/
                        <reference list>
 <condition>:= <literal> <comparison op> <returned aggregate>  /
          <returned aggregate> <comparison op> <returned aggregate>
<returned aggregate> :=
    <aggregation function>(select <attribute>
                            from [each] <tuple set> [object]
                            [where <where-expression>])       /
    (select <aggregation function>(<attribute>)
                            from [each] <tuple set> [object]
                            [where <where-expression>]        /
```

```
<tuple set>:= <object name>/<genitive relation>
<genitive relation>:=
      [related][<referencing object>.]<reference attribute>*
            <referenced object>[object[s]]
<comparison op>:= > / > / [not] = / >= / <=
<condition> := <atomic set> <set compare op> <atomic set>
<atomic set> := <set of literal values> /
                (select <attribute>
                 from [each] <tuple set> [object]
                           [where <where-expression>])       /
<tuple set>:= <object name>/<genitive relation>
<genitive relation>:=
      [related][<referencing object>.]<reference attribute>*
            <referenced object>[object[s]]
<set compare op> := [not] contains
```

REFERENCES

1. Abiteboul, S., and N Bidoit. Non first normal form relations to represent hierarchically organized data. In Proceeding of the ACM PODS Conference, Waterloo, Ont. Canada, 1984.

2. Bancilhon, F., et al. The design and implementation of O_2, an object-oriented DBMS, in "Advances in Object Oriented Database Systems," K. R. Dittrich, ed., Computer Science Lecture Notes 334, Springer Verlag, New York, 1988.

3. Bradley, J. A genitive relational tuple calculus for an N^2F object-oriented relational data model. Research Report 92/488/26, University of Calgary, 1992. To be published.

4. Bradley J. COOL concepts and semantics for definition, concentration and manipulation of composite objects in an N^2F relational data base. Research Report University of Calgary, 1993. To be published.

5. Cardenas, A. F., McLeod, D. "Research Foundations in Object-Oriented and Semantic Databases," Prentice Hall, Englewood Cliffs, New Jersey, 1990.

6. Codd, E. F. Relational databases, a practical design for productivity, CACM, 25(2), 1982, 109-117.

7. Hudson, S.E. and King, R. Cactis: A self-adaptive, concurrent implementation of an object-oreiented data base management system. ACM Trans. on Database Systems, 14(3), 1989, pp 291-323.

8. Kim, W. et al. Features of the ORION object-oriented DBMS, in "Object-Oriented Concepts, Databases and Applications," W. Kim, F.H. Lochovsky, Eds., Addison-Wesley, Reading, Mass, 1988.

9. Lamb, C. Landis, G., Orenstein, J. and D. Weinreb, The ObjectStore database system, CACM, 34(10), 1991, 51-63.

10. Lecluse, C., Richard, P., and F. Velez. O_2, an object-oriented data model, Proc. ACM SIGMOD Conference, 1989.

11. Lohman, G. M., Lindsay, B., Pirahesh, H., and Schiefer, K. B. Extensions to Starburst: Objects, types, functions and rules. CACM, 34(10), 1991, pp 95-109.

12. Stonebraker, M., Kemnitz, G., The POSTGRES next-generation data base system, CACM, 34(10), 1991, pp. 78-92.