1.0 INTRODUCTION

The object-oriented approach to database management [3, 8, 13, 14, 15, 16, 18, 19, 20, 25] evolved from the object-oriented approach used with programming languages such as C^{++} and Smalltalk, and is finding use in many application areas.

A consistent feature of the object-oriented approach is that associated with every object type representation, usually called a class, is a system generated identifier attribute together with both reference-list attributes and individual reference attributes, that reference the object identifiers of related object instances.

In an object-oriented database, at the conceptual level, the relationships between object types can be one-to-many, binary many-to-many, ternary many-to-many, recursive many-to-many, and and IS-A one-to-one relationships, as with relational databases [11, 12]. These relationships are supported by individual and reference-list attributes that are included in the conceptual database definition.

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 - nor is there any agreement on what exactly constitutes an object-oriented model [9, 26]. Unfortunately it is the simplicity of the relational model that makes it awkward to use effectively with data bases that essentially describe complex objects, where users frequently need to deal with composite objects, within which is an extensive structure of contained objects (often with a hierarchy of IS-A-PART-OF one-to-many and IS-A relationships). In the relational approach each type of subobject has to be retrieved and dealt with separately [12].

But because of the firm foundation underlying the relational approach 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 [1, 2, 21]. Retaining this requirement makes for a simpler data model, but one that is probably too simple for object-oriented data bases. Removing it makes for a more complex but richer relational model on which what might be termed a comprehensive object-oriented relational approach can be based. With such a model, conventional 3NF or 5NF relations can be used if desired, so that the model would be upward compatible with the conventional relational model, which is obviously desirable; where object-orientation is desired, unnormalized, or non-first normal form (N^2F) relations need to be used.

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 [17], which uses an extension of SQL, and the POSTGRES system [23, 24], which uses an extension of QUEL. Such systems, known as extended relational systems, essentially attempt to combine the benefits of object oriented data base systems with those of the standard relational approach. Extended relational systems are best suited in circumstances where there are many different kinds of data bases (both object and other data bases) with applications that sometimes need access to more than one kind. They are thus in the mainstream of data base development. Strictly object-oriented systems, particularly those that are designed to interface efficiently with a specific object-oriented programming language, are likely to find use restricted to independent engineering (CAD/CAM) applications. In this paper we are concerned with the object-orientation aspect of extended relational systems. An N^2F data base, restricted in the sense described above, is assumed.

1.2 Declarative language approaches

There appears to be 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 [12]. The other approach is what might be called a composite object-oriented approach exemplified by Composite object-oriented Language or COOL [7], which is currently being implemented as part of the GenRel project.

COOL is a powerful language, is soundly-based on the theory of sets and relations [6] and is fundamentally object-oriented. COOL has borrowed some concepts, such a natural quantifier facilities, from older languages for relational data bases, such as the EOS predicate calculus [4], and SQL/N [5]. However COOL has many new concepts, particularly the concept of a genitive relation, that enable it to handle all kinds of composite objects.

We can initially compare and contrast the concepts behind these two approaches, that is, essentially the SQL and COOL apraoches, in the light of the three types of structures that occur in data bases; these structures are:

(a) independent entity structures, both hierarchical and network,
(b) aggregation or composite object structures, aggregated via IS-A-PART-OF relationships,
(c) generalization or IS-A type hierarchies.

1.3 Independent-entity structures

Independent-entity structures are widespread in business data bases, with independent entity types or object classes participating in one-to-many and many-to-many relationships resulting in either network structures, for example, Date's network structure Suppliers, Parts, Projects and Shipments data base [12], or a hierarchical structure such as Bank-branch (root), Teller (level 2), Customer (level 3), Savingsaccount (level 3), Checkingaccount (level 3).

It is important to understand that SQL is used with such data bases in an entire relation oriented manner. If one wants the status of suppliers in Athens that have shipped only red bolts, one uses constructs that deal with all of the Supplier tuples, all of the Shipment tuples and all of the Parts tuples, that is, one deals with, and thinks in terms of, entire relations and all of the relations involved in the retrieval.

There is little need for an object-oriented approach here and the conventional relational approach with SQL works very well. However, a language like COOL can be beneficial even here. To illustrate, compare the SQL and COOL expressions for the retrieval from Date's Supplier-Parts database:

Retrieve the status of London suppliers that have supplied only red bolts:

```
SQL:   Select S#, status
       from S
       where S# not in (select S#
                        from SP
                        where P#  not in (select P#
                                          from P
                                          where Pname = 'bolt'
                                            and color = 'red'));

COOL:  Select S#, status
      from [each] S [object]
      where for each [of its] S.Sref*SJ [objects](
                for the [one] SJ.P#*P [object] (Pname = 'bolt'
                                                  and color = 'red')
```

The SQL expression is error prone because of the required double-negative and danger of mistakes with De Morgan's Rules. In the COOL expression words in square brackets may be omitted and are intended to aid readability. The COOL expression follows natural language structure, and has obvious meaning except for the terms S.Sref*SJ and SJ.P#*P. These terms denote genitive relations and essentially correspond to the expressions "supplier's shipments" and "shipment's parts" respectively. Unlike SQL, COOL expression structure is independent of the quantifiers involved. If we change the quantifier in the above retrieval to "for most", instead of "for all" or "for each", as in:

Retrieve the status of London suppliers that have supplied mostly red bolts:

the COOL expression structure remains unchanged:

```
COOL:  Select S#, status
      from [each] S [object]
      where for most [of its] S.Sref*SJ [objects](
                for the [one] SJ.P#*P [object] (Pname = 'bolt'
                                            and color = 'red')
```

In contrast, the SDQL expression is much more complex and quite different in structure from the previous one. The reader is urged to try it.

1.4 Aggregation or composite-object structures

The second type of structure is a composite object, or aggregate, type, where the relationships are mostly one-to-many and involve physical containment or attachment, and where the resulting structure is usually hierachical. For example, suppose that a data base involves national parks entities (object type Park). A Park instance can contain instances of ranger stations (Station), and forest areas [Forest]. A Forest can contain campgrounds (Campgr), and lakes (Lake). The data for a specific park forms a composite object instance, whose subobject instances form a hierarchical structure, involving the specific Park data, with its contained Ranger and Forest instances, with, in turn for each Forest instance, the contained Campgr and Lake instances. [A specialized variation is the bill-of-materials structure, where root and subobjects are all of the same class Part; this give rise to both hierachical explosions and implosions.]

Once again SQL is used with such a data base in an <u>entire relation</u> oriented manner. If one wants the size of Parks in California that have only forests without bears, one uses constructs that deal with all of the Park tuples and all of the Forest tuples, that is, one deals with, and thinks in terms of, entire relations and all of the relations involved in the retrieval.

However users seem to prefer to work with composite object instances when dealing with such an aggregation data base and not with Forest and Lake relations, whose tuples belong to a wide variety of Park instances. They prefer to deal with them in a composite object-oriented manner, that is, they prefer to retrieve and store composite object instances, manipulate them with programming languages, have them displayed in hierachical format, and, most importantly from a declarative language point of view, specify the retrieval and update of a composite object-instance in terms of the values in that instance. They prefer not to deal with them in the quite unnatural terms of the entire relations to which the subobjects belong, as is required with the relation-oriented approach to declarative languages. As an example, take the easily expressed request:

    Retrieve full data on each California park object instance
    where all of its forests exceed 10 square miles and have only
    campgrounds with fireplaces.

With SQL we would have to code something like:

```
sql:      select * from Park
            where loc - 'California' and
             park# in (select park# from Forest)
              and park# not in (select park# from Forest
                where area <= 10  or
                   forest# <not> in (select forest# from Campgr
                                  where fireplace = 'no')
                 or forest# not in (select forest# from Campgr));
```

Although the type of park required is natural to visualize from a point of view of a composite object, we have to think and code in relational terms of entire relations (not to mention treading carefully with De Morgan's rules), which can be error prone (the above expression contains a slight but fatal logical error, which the reader is invited to find). An alternative is a language that enables us to state our requirements in terms of the kind of composite object it is desired to retrieve, that is, from a composite object manipulation point of view, for example:

```
 Retrieve each park, in California
  where all that park's forests each has
                      (a) area > 10, and
                      (b) all that forest's campgrounds each has
                                                    a fireplace.
```

Such a language would be fundamentally object oriented and much more convenient to think in terms of - for data bases of this type. COOL has such a structure.

The COOL expression for the above retrieval mirrors the natural language expression structure:

```
  select * from [each] Park [object]
  where loc = "California"
        and for all [Park.]forestref*Forest [objects]
                          ((area > 10 and
                          for all [Forest.]campref*Campgr [objects]
                                                 (fireplace = "yes'))
```

Items in square brackets may be omitted. The expression Park.forestref*Forest objects specifies a genitive relation which is semantically equivalent to the natural language genitive case expression "a park's forests". Similarly Forest.campref*Campgr is equivalent to the expression "a forest's campgrounds". The quantifier "for all" is used exactly as in natural language and not in the convoluted manner of conventional set theoretic expressions, which has prevented its incorporation in SQL. Once again, if the quantifiers change, the structure of the COOL expression does not.

For example, suppose now we want full data on each california park object instance where all but 3 of the forests therein exceed 10 square miles and each forest has a majority of campgrounds with fireplaces. The COOL expression is:

```
 select * from [each] Park [object]
  where loc = "California"
        and for all but 3 [Park.]forestref*Forest [objects]
                         ((area > 10 and
                         for most [Forest.]campref*Campgr [objects]
                                              (fireplace = "yes')
```

The quantifiers are the natural quantifiers **for all but 3** and **for most** or **for majority of**. The composite-object oriented structure of COOL and its use of natural quantifiers simplify many query expressions involving composite objects. The above query expressed in SQL is quite difficult - and quite different in structure from the earlier one. This does not imply that there is anything wrong with SQL. It is merely that SQL is relation-oriented and is best suited to situations where it is natural to deal with whole relations. Where it is natural to deal with a single composite object instance SQL is usually inconvenient to work with, and composite-object-oriented languages like COOL are unusually convenient to work with.

The concepts behind genitive relations and the use of COOL for elementary retrieval expressions with composite objects have been presented in detail in [7]. In this paper, COOL concepts and facilities for defining and concentrating complex objects are presented.

1.5 Generalization or IS-A type hierarchies

The third kind of data base structure involves generalization, that is, IS-A relationships and inheritance. Before the invention of classes and object-oriented programming languages, such structures were commonly dealt with (and still are) in COBOL and PL/1 file processing, using variable-length record files.

The classic variable-length record file processing example involves a file containing records about ships, where each record represents a ship. Suppose for now that just two distinct kinds of ship are represented, tankers and freighters, so that there are two kinds of record in the file, each of different length and format. But because all records deal with ships, the initial fields, such as shipname, tonnage, captain, would be common to all records. Other fields, oil capacity, number of tanks, and so on, would apply only to the tanker records, whereas container capacity, maximum container size, and so, on would apply only to the freighter records. Thus the program would need two types of record structure to deal with this file, one for ships that are freighters and one for ships that are tankers. However, within each record structure, normally at the beginning, is the set of attributes common to all types of ship, one of which will indicate the type of ship, enabling the record to be dealt with, that is, assigned to the correct structure variable, when read from a file.

The object-oriented programming approach to such ship objects would be to define three classes (structures with private attributes) as opposed to two record structures in COBOL or PL/1. We define a Ship class, together with a Freighter class that inherits all the attributes of a Ship class, and also a Tanker class that inherits all the attribute of the Ship class. Leaving aside for the moment considerations of private attributes and methods, a (structure) program variable will be used to represent a specific freighter instance and a different variable to represent a tanker instance, each of which variables will also explicity, via inheritance, represent a ship instance. The contents of these variables can be made persistent and stored as part of a data base, if an object oriented data base system is available, the stored structure being the equivalent of a variable-length record file. This is essentially not any different, at least in principle, from the COBOL or PL/1 approach, where there is a structure variable for tanker records and another one for freighter records. What is quite different in the object-oriented programming approach is that some of the attributes of a class can be private, and others are derived attributes, the result of applying a function (or method) to other, possibly private, attributes. However, this is best treated as an additional or extension feature of earlier ideas.

The relational approach to generalization structures is to define a relation for the root type (for example a SHIP relation) and each subtype (Tanker or Freighter) and for each of their subtypes (for example a freighter could be a Bulkcarrier or a Containership), and so on. Suppose we have the type hierachy Ship, which can be Tanker or Freighter, which can be Bulk carrier or Containership. In the relational approach there would likely be five relations, one for each of the Ship, Tanker, Freighter, Bulkcarrier, Containership relations. There is a severe problem with this approach. As an example, suppose we want all data about the container ship Lentia. In SQL we would specify:

```
sql:     select *
         from Ship, Freighter, Containership
           where Ship.ship# = Freighter.ship#
            and Freighter.ship# = Containership#
            and Ship.shipname = 'Lentia'.
```

We concatenate or join all Containership tuples with their matching Freighter tuples and all the resulting tuples with their matching Ship tuples (using the Ship identifier attribute ship#) to give the full tuples for all containerships represented in the data base. Then we select the record for shipname Lentia. However, this presupposes that we knew in advance that the Lentia was a containership. Suppose we had not known. In that case we would have had to search all of the subtype hierarchy looking for a match, which is awkward to do in SQL, especially if the hierarchy is large. There are several alternatives, but the simplest, in this case, is search the full concatenated types for all tankers (join of Ship and Tanker), for all bulk carriers (join of Ship, Freighter, Bulkcarrier) and for all containerships (join of Ship, Freighter and Containership), which requires three separate SQL expressions. It is clear, however, that it can be awkward to handle type hierarchies with SQL in a conventional relational data base.

In this paper we present COOL techniques, using an N^2F relational data base, that enable such retrievals to be expressed more conveniently. These techniques depend on techniques for definition and concentration of composite objects, as discussed in the previous section.

1.6 COOL and object manipulation

In the rest of this paper we show how the composite object-oriented language, COOL, for object-oriented N^2F relational data bases can be used with both aggregation data base structures and generalization IS-A type hierarchies but with other structures as well. For this reason the project data base used to demonstrate COOL concepts in this paper is not solely either aggregation or generalization structure oriented, but is well diversified with respect to possible structures.

1.7 The project database

In the object-oriented approach, an object has a unique identity that is independent of any values it contains [10]. An object normally has associated attributes, and, as in the relational approach, one of these attributes, or a group of them, may be regarded as a primary key. However, because every object has a unique identity, an object need not have a primary key. Instead, the database system will generate a unique object identifier, which may or may not be accessible by the user, depending on the database system employed. Such system generated unique keys are assumed in the N^2F data model used in this paper.

As well as a possible primary key attribute, an object may have either simple attribute types, such as a quantity or a name, whose type allows either literal numeric or alphnumeric values, or structure attribute, that is, composite attribute types, such as the type ADDRESS, indicating a composite value, coresponding to the value of a C structure variable (that is not an array). As in O^2, ORION and POSTGRES [3, 16, 14, 24], an element of a composite type can be specified using a path expression with a nested dot notation, as in address.street.number. A composite type may not be a set, that is, an embedded relation, so that the equivalent of repeating groups of conventional file processing are not allowed in the N^2F relations considered here.

An object type may also have collection attributes, such as sets or lists, for example, the set of keywords in a document, or a list of object identifiers to support a relationship. As well as conventional stored or persistent attribute object types, an object type may have computed object types, computed by means of functions, which can use related object instances, as in Cactis [], to derive a result. Thus for an object type **Sphere**, attributes **radius** and **center** could be stored, whereas other non stored but retrievable attributes could be **volume**() and **surface-area**(), and **number-of-attachments**(), this last attribute being computed from the number of related object instances in the **Attachments** object type. If the stored attributes are all private, only computed attributes are available. For purposes of this paper, all attributes are assumed to be public. The inclusion of functions as attributes is merely a convenience. Obviously, they can be done without by defining views that contain the results of retrieval operations, but often at the expense of more complex query expressions.

Our N^2F data base is assumed to allow all of the attribute types described above - but, as mentioned, no repeating group equivalents. Many of these points can be illustrated by the database definition in Figure 1 and also later in Figure 3 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[Person];
    chaplist:    LIST[Chapter];>
    nchapters(d) =
                count (select * from Chapter
                       where for the related doc#*Document object
                                     doc# = d)
   Chapter:   <
    chap#:       Chapter;
    doc#:        Document;
    ctitle:      STRING;
    npages:      INTEGER;
    ndiagram:    INTEGER;>

Person: <
    pers#:       Person;
    doclist:     LIST[Document];
    pname:       STRING;
    position:    STRING;        >

Program: <
    prog#:       Document
    lang:        STRING;
    lines:       integer
    runlist:     LIST[Run];   >

Run: <
    run#:        Run;
    prog#        Program;
    machine:     STRING;
    rundate:     DATE; >
```

Figure 1

With the exception of cyclic or recursive relationships, the database in Figure 1 contains the common types of relationships that are encountered in object-oriented databases. It is a modified N^2F version of a database used by Cattell in a discussion of object-oriented databases [10]. It has 1:n relationships as well as a binary many-to-many and a subtype (IS-A) relationship, this last relationship permitting utilization of the inheritance concept [22].

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. The number of chapters in a document is not stored, but is derived by the function nchapter(), shown defined as the result of a COOL operation, which will become clear later. [There is no advantage in using COOL here to define nchapters(), however. It could equally well be defined using an SQL expression as:

```
     nchapters(d) = select count(*) from Chapter
                    where doc# in (select doc# from Document
                                   where doc# = d)          ]
```

A person can author many documents and a document can be authored by many persons. A person is represented by a **Person** object and the object **Authact** (or author activity) enables the resulting many-to-many relationship between **Document** and **Person** objects.

The object **Program** represents a computer program. Since a program is a kind of document, the set of unique (system generated) object identifiers (**prog#** values) in **Program** objects are to be found among the set of unique object identifiers (**doc#** values) in **Document**. Thus the object types **Document** and **Program** form a subtype hierarchy, the relationship between the two being an IS-A relationship. A **Run** object represents an execution of a program. Since a program can be executed many times there is a 1:n relationship between **Program** and **Run**.

Note the significance of the IS-A relationship in the database. Because of this relationship, a **Program** object inherits not only the attributes of the corresponding **Document** but also each relationship in which **Document** participates. Thus we can have both legitimate requests involving attribute inheritance, such as:

"What are the titles of programs executed more than 100 times?"

and legitimate requests involving relationship inheritance:

"Who are the authors of C programs that have never been executed?"

The names of the attributes in Figure 1 were chosen to make the semantics self-apparent. Where this is not so, more detailed discussion later in the paper should clarify matters. 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#** has 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 BASIC COOL AND GENITIVE RELATION CONCEPTS

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 [7].

2.1 Genitive relations and 1:n relationships

A genitive relation is needed 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 or n:m relationship.

Consider the 1:n relationship between **Document** and **Chapter** objects (Figure 1). 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 "document's chapters" 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 is called 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 use 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 Appendix 1.] 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].
```

[Note that chaplist* is essentially the syntactic equivalent of the apostrophe s construct of English for the genitive case, so that the above is semantically equivalent to the English language expression "document's chapter objects".]

This genitive relation specification technique, which makes it possible to unamibguously refer to a set of child tuples in a 1:n relationship, and makes it possible to treat this set as a relation, is fundamental to COOL and is rich in possibilities.

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.

The above quantifier retrieval example 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');
```

Cross reference expressions, involving a quantifier, a genitive relation and a condition, can be nested, in which case the condition will itself include another cross reference expression. This is detailed in the COOL syntax in Appendix 1, and was illustrated with the retrievals earlier with the Parks data base.

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
```

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.

3.0 COOL AND COMPOSITE OBJECTS

3.1 Language constructs for retrieving composite objects

No conventional retrieval language the author is aware of does justice to retrieval and definition of composite objects. The most ambitious, to date, appears be be the XNF (extended normal form) views of Starburst. The problem, which is not comprehensively addressed in conventional relational languages, is that there are two quite different types of operations involved, each with distinct conditions. One operation and associated set of conditions is for selection of the full composite object stored in the data base from a set of such objects, that is, essentially an XNF view; the other operation and associated set of conditions is for selection of any of the large number of possible reduced versions, or concentrations, of the (XNF) composite view, that is, of the selected object. An XNF view is not stored, although its definition is, but the tuples that compose it are stored only in the storage form of their respective base tables.

For example suppose we have the object type aggregation hierarchy: object type A is parent of object types B and C; in turn object type B is parent of object types X and Y. Suppose A1 and A2 are instances of object type A, with B1, B2, B3 ... instances of B, and so on. Suppose then that the data base has two composite objects A1 and B2, with hierarchical structures laid out as:

```
                             A1
     B1                 B2                    C1   C2   C3
 X1 X2 X3    Y1     X4 X5    Y2 Y4


                             A2
     B3                   B4                    B5         C4   C6
X7 X8    Y5 Y6 Y7     X10    Y8 Y9     X12 X14 X17   Y12
```

A set of selected objects instances like these A1 and A2 composite object instances can be defined as an XNF view. In the simplest case the XNF view would contain only a single object instance, such as A2, although typically several object instances will be contained. For example, we can have a selection operation with associated conditions that selects composite structure A2, and rejects A1. These conditions can involve A, B, C, X and Y entities and can involve natural quantification. But having selected A2 (and perhaps some other composite objects as well), from each such composite object instance we can now create a large number of derived or concentrated composite objects based on the original composite object instance selected, using quite different conditions to sculpt a desired concentrated object instance or concentration instance.

Some concentrated objects instances derived from A2 might be:

```
      A2                                           A2
      B4           C4                              B5
 X10       Y8 Y9                                   Y12
```

In the conventional relational approach the tuples needed to construct the composite object A2 would be stored as part of 5NF relations A, B, X, Y and C:

```
A2      B3      X7         Y5          C4
 .      B4      X8         Y6          C6
 .      B5      X10        Y7           .
 .       .      X12        Y8           .
 .       .      X14        Y9           .
 .       .      X17        Y12          .
 .       .       .          .           .

 A       B       X          Y           C
```

With conventional relational technology, each of these 17 tuples might be delivered separately to the host program using a FETCH [12] or equivalent command, leaving it up to the applications program to populate a hierarchical structure variable. This is often inconvenient.

Alternatively, with conventional SQL, a composite object could be retrieved as sets of diverse tuple concatenations, with one set of tuples for each ultimate or lowest-level child object with all its ancestors in the hierarchy:

```
A2   B3   X7
A2   B3   X8
A2   B4   X10
A2   B5   X12
A2   B5   X14
A2   B5   X17

A2   B3     Y5
A2   B3     Y6
A2   B3     Y7
A2   B4     Y8
A2   B4     Y9
A2   B5     Y12

A2     C4
A2     C6
```

In the example above 3 distinct SQL expressions would be needed, one for each type of tuple. The tuples would be delivered to a program variable one at a time, and would involve duplication. This too is often inconvenient.

In the object-oriented extended relational approach the tuples going to construct the composite object A2 will be stored only as part of the N^2F relations A, B, X, Y and C, whose tuples include reference lists that may include duplicates:

```
A2 [B3,B4,B5][C4,C6]    B3[X7,X8][Y5,Y6,Y7]      X7    Y5          C4
 .                      B4[X10][Y8,Y9]           X8    Y6          C6
 .                      B5[X12,X14,X17][Y12]     X10   Y7           .
 .                      .                       X12  Y8          .
 .                      .                       X14  Y9          .
 .                      .                       X17  Y12         .
 .                      .                        .    .          .
                 A                       B        X    Y          C
```

Since object oriented programming languages, and older languages like C, PL/1 and COBOL, deal with composite objects as hierarchical structure variables, a composite object would be best delivered as a single hierarchically structured record:

A2 B3 X7 X8 Y5 Y6 Y7 B4 X10 Y8 Y9 B5 X12 X14 X17 X12 C4 C6

or, in more easily visualized terms

```
A2
  B3
    X7 X8
     Y5 Y6 Y7
  B4
    X10
     Y8 Y9
  B5 X12 X14 X17 X12
   C4
   C6
```

whose structure would match that of a previously declared structured variable.

COOL can be used to select composite objects, that is, the equivalent of an XNF view, and concentrate and retrieve composite objects for display or delivery to such structure variables. An example will illustrate the essence of the COOL approach. Suppose the following:

(1) We want to select each composite object consisting of a document with authors and chapters using the following composite object selection conditions:
   * It is a document about data base machines
   * At least two of its authors are engineers
   * The average chapter length exceeds 20 pages.

We intend to give the name C1 to this set of composite object instance or XNF view.

(2) From each composite object instance in C1, where possible, we want to deliver to a host program a concentrated object instance, concentrated using the following concentration conditions
   * Only authors who are secretaries are included.
   * Only chapters with less than 10 pages are included, with omission of the number of diagrams attribute value.

This cannot be specified in conventional SQL, nor even the extended SQL of Starburst, even though it is not very complicated; in practice composite objects with many hierarchical levels have to be managed. The operation can be specified in COOL with a composite object selection step followed by a concentration step. To extract the necessary composite object instances or XNF view:

```
 select * from Document object
                where Document.topic = 'database machines'
                and for at least 2 of its authact*Person objects
                                 (position = 'engineer')
                 and 20 < avg(select npages
                        from Document.chaplist*Chapter),
        * from [related] Document.chaplist*Chapter objects,
        * from [related] Document.authlist*Person objects;
```

The where-condition specifies which composite object instances, in terms of hierarchy type instances, are to be selected. The from-expressions specify which subobject instances of the composite object are to be part of the selection. As is conventional with SQL, an asterisk denotes all attributes.

A composite object so defined may be made into a composite object view, by means a create composite object command, which functions like a conventional create view command. Thus the composite object retrieved above could be placed in the composite object (view) called C1, as follows:

```
create [each instance of] composite object [view] C1 as
 select * from Document object
                where Document.topic = 'database machines'
                and for at least 2 of its authact*Person objects
                                 (position = 'computer engineer')
                 and 20 < avg(select npages
                        from Document.chaplist*Chapter),
        * from [related] Document.chaplist*Chapter objects,
        * from [related] Document.authlist*Person objects;
```

To specify the concentrated object type:

```
     select * from C1.Document object,
            * from [related] C1.Document.Authlist*Person objects
                   where position = 'secretary',
            title, npages
               from [related] C1.Document.chaplist*Chapter objects
                   where npages < 10;
```

If the concentrated composite object needs to be defined as an additional view C11, we can use the create

```
     create [each instance of] composite object [view] C11 as
     select *  from C1.Document object,
            *  from [related] C1.Document.Authlist*Person objects
                   where position = 'secretary',
            title, npages
```

```
               from [related] C1.Document.chaplist*Chapter objects
                   where npages < 10;
```

The select and create commands to retrieve and create a view for a composite object are the same as those needed to concentrate a composite object. A further concentration of C11 could be created by a further application of the the create command. The asterisk before 'from' has the conventional meaning of all attribute values. Thus the expression

```
           *  from C1.Document object
```

specifies all the attributes in the root Document tuple in an instance of the set of composite object instances C1 extracted from the data base by the 'create composite object C1' expression.

The create composite object command is quite flexible, and it is possible to both define and concentrate a composite object with a single command. This is not recommended, however, since the two operations are conceptually quite different. To create the concentrated composite object C11 in a single command we would write:

```
create [each instance of] composite object [view] C11 as
 select * from Document object
               where Document.topic = 'database machines'
               and for at least 2 of its authact*Person objects
                                (position = 'engineer')
                and 20 < avg(select npages
                      from Document.chaplist*Chapter),
        title, npages
          from [related] Document.chaplist*Chapter objects,
                where npages < 10,
        * from [related] Document.authlist*Person objects;
                where position = 'secretary'
```

The above expressions further demonstrate the importance of being able to specify the set of tuples related to a specific parent tuple in a direct and concise manner, in addition to the indirect manner of SQL. Without this direct specification capability, it would not be possible to specify any arbitrary concentrated composite object type. Although there are only two levels in the above example, the specification could continue to any arbitrary number of levels. It should be apparent that as a tool for the extraction and concentration of complex composite objects COOL is really powerful and versatile - more so than any proposed SQL extensions in the literature. The reader is also invited to attempt specification of C11 using SQL - it simply cannot be done.

The general syntax for the create composite object command- for extracting a composite object (with or without concentration) from the data base is simple in principle, but repetitive:

```
 create [each instance of]
             composite object [view] <composite-object-name> as
   select <attribute-list> from <root-object> [object]
    [where <where-expression>]
          [<attribute-list> from <genitive relation>
```

```
                    [where <where expression>]
          [<attribute-list> from <genitive relation>
                    [where <where expression>]
          [<attribute-list> from
                                                ...]]]
     <genitive relation>:=
        [related][<referencing object>.]<reference attribute>*
             <referenced object>[object[s]]
```

The term <where-expression> has the syntax shown in Appendix 1, and identical semantics. In any case where a where-expression follows a genitive relation specification then concentration is being carried out in addition to initial selection or definition of a composite object represented in the data base.

A simple Fetch command could be used to transfer composite objects, one by one, to a host program structure variable, or in a batch, to a host program structure array.

3.2 COOL expressions with a composite object view

COOL retrieval expressions can be used with a composite object view. There is a difference between COOL with conventional views and composite object views. With a conventional view COOL is used as with base tables. For example, suppose the view, defined with either an SQL predicate or a COOL predicate, where each tuple contains a document title, revision date and topic for documents with more than 50 pages:

```
SQL: Create view Macrodoc
     as select  title, date, topic
        from Document
        where 50 < (select sum(npages)
                    from Chapter
                    where Document.doc# = Chapter.doc#);

COOL: Create view Macrodoc
     as select  title, date, topic
        from [each] Document [object]
        where 50 < (select sum(npages)
                    from Document.chaplist*Chapter [objects])
```

The retrieval: list the titles of documents about computers, can thus be written as:

```
      select title
      from Macrodoc
      where topic = "computers";
```

The above expression is both correct COOL and correct SQL.

In contrast, consider the retrieval from the concentrated composite object C11, as defined earlier:

Get the title of each document that has at least 3 chapters with less than 6 pages:

```
select title from each C11.Document object
where for at least 3 C11.Document.chaplist*C11.Chapter [objects]
                                         (npages < 6);
```

Here the relation names must be prefixed with the composite object name, since only tuples from those relations that occur in C11 are to be considered. The expression

```
select title from each Document object
where for at least 3 Document.chaplist*Chapter [objects]
                                         (npages < 6);
```

would in general produce a different result, since tuples not in C11 would be used.

3.3 Language constructs for handling inheritance with generalization or IS-A type hierarchies

No additional language constructs are needed for manipulation of inheritance. For example, consider the retrieval:

   Get the names of authors of C programs that have never executed on machine m42.

Between **Person** and **Document** there is a many-to-many relationship, and therefore, via inheritance, also between **Person** and **Program**, which is the relationship required for the retrieval. Furthermore, the LIST attribute **personlist** in **Person** holds object identifiers of **Program** objects, since these are also object identifiers of **Document** objects. The retrieval can therefore be easily expressed as follows:

```
select pname from Person
where for at least 1 of its [related] doclist*Program objects
(lang = 'C' and for none of its [related] runlist*Run objects
                                            (machine = 'm42'))
```

All retrieval expressions involving retrieval of a single object type and inheritance can thus be handled in this way. A different approach is needed when we want to retrieve an object instance together with inherited attributes. For example, the retrieval:

      Get full information on C programs that have been run on machine M42.

        Here we want a **Program** tuple concatenated to its parent **Document** tuple, depending on conditions applying to the **Program** tuples and other related tuples. This is merely a particular case of composite object retrieval, as discussed earlier

```
select * from Program object
         where for at least 1 of its Program.runlist*Run objects
                                            (machine = 'm42'),
       * from [related] Program.prog#*Document object;
```

or if we wish to treat it as a composite object view:

```
create [each instance of] composite object view P1 as
select * from Program object
         where for at least 1 of its Program.runlist*Run objects
                                             (machine = 'm42')),
       * from [related] Program.prog#*Document object;
```

Recall that earlier we discussed inheritance in the light of both variable length files and difficulties with SQL. We had the example of the Ship, Tanker, Freighter, Containership, and Bulkcarrier relations, an IS-A type hierarchy with Ship as the root. We saw that certain types of information are not easily retrieved from such type hierarchies by SQL; for example, we saw that it could require up to three SQL retrievals to retrieve full data about the ship called Lentia, and more if the hierarchy were larger, if we did not know beforehand what kind of ship it was.

But suppose these relations were N^2F relations, where each tuple has reference list attributes to tuples at the next level down. For example, suppose Ship tuples contained the system generated identifier ship#, Tanker tuples the identifier tanker#, freighter tuples the identifier freigter#, and so on.

The above retrieval can now be carried out with COOL by creating a composite object type:

```
create [each instance of] composite object [view] C1 as
 select * from Document object

Create [each instance of] composite object [view] S1 as
         select * from Ship
                  where Ship.shipname = 'Lentia',
               * from Ship.ship#*Tanker,
               * from Ship.ship#*Freighter,
               * from Freighter.freighter#*Bulkcarrier,
               * from Freighter.freighter#*Containership;
```

If we assume that shipnames are unique, then the object will have only one instance. If the condition clause had been Ship.weight > 10000, instead, for example, the composite object would have many instances, of the kind Ship concatenated to Tanker, of the kind Ship concatenated to Freighter and Bulkcarrier, and of the kind Ship concatenated to Freighter and Containership. And if desired, such an object type could be concentrated, using the concentration command described earlier.

3.4 Updating

There is a difficult problem as far as updating composite objects is concerned. Change, deletion or insertion updating of a composite object retrieved from the data base without concentration does not give rise to any special difficulties, since the tuples from which it is constructed all have a unique identifier and belong to identifiable relations. However, when we try to update a concentration of such a composite object, it is entirely another matter. In many cases it will not be possible to identify the specific tuples on which the structure is based, so that an updated concentration cannot easily be stored. Essentially a concentrated object is a composite view, that is, a structure constructed from views of individual relations. Since a view of a relation cannot always be updated, neither can a much more complex concentration of a composte object. A trivial, but workable, solution would be to simply prohibit update of concentrations. However that would normally be too drastic. What is possible as far as updating concentrations is concerned will be considered in a later paper.

4.0 SUMMARY

There are two fundamentally different approaches to data base declarative languages. One approach is the entire relation-oriented approach, embodied in the languages of the conventional relational approach, such as DSL Alpha, SQL, and relational algebra. The other approach is a composite object manipulation approach, an example being COOL, an object-manipulation language designed for manipulation of an N^2F relational data base.

An overview of that part of COOL that is necessary for manipulating N^2F relations and relationships between them has been presented. Full details of this part of COOL are presented elswhere [7].

In this paper details of COOL language constructs have been presented for construction, retrieval, concentration and further manipulation composite object views of arbitrary complexity, with language constructs that are restricted to objects involved in the composite object being manipulated. It is also shown how COOL can handle generalization or IS-A type hierarchies.

COOL is proposed as an extension to SQL, for use where the data base has a distinct object-orientation. Currently, there are no implemented examples of a composite object-oriented genitive relational data base language that allows the use of natural quantifiers. However, GenRel, a project at the Space Information Science Laboratory at the University of Calgary to develop a prototype composite object-oriented N^2F genitive relational data base system, will embody both SQL and COOL.

APPENDIX 1 Essential COOL syntax

Note: As an aid to understanding, some productions are repeated.

PART 1 Quantifiers and cross references to related relations

```
<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>:= <condition-a>/<condition-s>
```

PART 2 Conditions that involve specific atomic values

```
<condition-a>:= <attributename> <comparison op> <attributename>/
                <attributename> <comparison op> <literal>/
                <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] = / >= / <=
```

PART 3 Conditions that involve sets of values

```
<condition-s> := <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., Hall, R. IFO, a formal semantic data base model, ACM Trans. on Database Systems, 12 (4), 1987.

2. 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.

3. 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.

4. Bradley, J. An extended owner-coupled set data model and predicate calculus for data base management, ACM Trans. on Data Base Systems, **3**(4), 1978, p. 385-415.

5. Bradley, J. SQL/N and attribute/relation associations implicit in functional dependencies, Int. J. Computer & Information Science 12(20), 1983.

6. 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.

7. Bradley J. COOL: A composite object oriented language for N^2F object-oriented relational data bases. Research Report 93/512/17 University of Calgary, 1993. To be published.

8. Bret, R., et al. The Gemstone Data Management System, in "Object-Oriented Concepts, Databases and Applications, W. Kim, F.H. Lochovsky, Eds., Addison-Wesley, Reading, Mass, 1988.
Butterworth, P. Otis, A, and J. Stein. The GemStone object database management system, CACM 34(10), 1991, pp. 65-77.

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

10. Cattel, R. G. G. "Object Data Management", Addison Wesley, 1991.

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

12. Date, C. J. "Introduction to Database Systems, 5th ed., Addison Wesley, Reading, Mass., 1990.

13. 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.

14. 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.

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

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

17. 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.

18. Object Design. ObjectStore Reference Manual, Object Design, Inc., Burlington, Mass., 1990

19. Objectivity. Objectivity Database Reference Manual, Objectivity Inc., Menlo Park, California, 1990.

20. Ontologic. ONTOS Reference Manual, Ontologic Inc., Billerica, Mass., 1989.

21. Osborne, S., and T. Heaven. The design of a relational systems with abstract data types as domains, ACM Trans. on Database Systems, 11(3), 1986 pp. 357-73.

22. Smith, M, and Smith, J. Database abstractions: Aggregation and generalization, ACM Trans. on Database Systems 2(2), 1977 pp 105-133.

23. Stonebraker, M., Anton, J, and E. Hanson. Extending a data base system with procedures, ACM Trans. on Database Systems, 12(3), 1987, pp 350-376.

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

25. Versant Object Technology. Versant Reference Manual, Versant Object Technology Inc., Menlo Park, California, 1990.

26. Wiederhold, G. Views, objects and databases, IEEE Comput., 19(12), 1986, 37-44.