INTRODUCTION

In this paper we propose an object-oriented, as opposed to entire relation oriented, tuple calculus [16, 21] for an object-oriented (but fundamentally relational) data model [9, 10], consisting of non normal form (N^2F) relations [1, 2, 19]. The motivation is a need for a set theoretic and relational foundation for the development of object-oriented, relational declarative data base languages, in the same way that normalized relations and conventional relational tuple calculus are the foundation for conventional relational declarative data base languages, such as SQL and QUEL [11, 13, 14, 15, 16].

The object-oriented approach to database management evolved from the object-oriented approach used with programming languages such as C^{++} and Smalltalk [1, 2, 9, 10, 26], and is finding use in such application areas as software engineering, document preparation and management, the computer-aided design and manufacture of engineering parts, and scientific data bases [10].

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 [10]; often also there are public, and sometimes private, stored individual and collection attributes, and public derived or function-generated attributes. Apart from the reference list and collection attributes, an instance of a class, or object instance representation, crudely corresponds to a tuple in the relational approach [16, 21], and a class corresponds to a relation.

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 IS-A one-to-one relationships [23], as with relational databases. However, unlike relational databases, where relationships are supported by the values of individual attributes, these relationships are supported by individual and reference-list attributes that are included in the conceptual database definition. Thus, when an object instance is retrieved, references to all other objects to which that object is related are immediately available, which is not the case in the relational approach, where, if relations participating in relationships with the retrieved object are not known, then the entire data base must be searched for matching foreign key values.

The reference attributes supporting a relationship in the object-oriented database definition are two-way, for example, a reference list in a parent object instance referencing related child object instances, and an individual reference in a child object instance to its parent. In addition, in the case of type hierarchies resulting from a hierarchy of IS-A relationships, each subtype inherits the properties of its parent, which, in turn, inherits the properties of its parent, and so on.

Although rich in properties, the object oriented model does not have the strong theoretical foundation of the conceptually

simpler relational model [13, 15, 16, 21] - nor is there any agreement on what exactly constitutes an object-oriented model [9], the properties listed above being simply found in many object-oriented data models. It is the sound foundation of the relational model, based on the established theory of relations, that permits the construction of relational calculus-based languages like SQL [11, 15, 20, 26]. Unfortunately it is the very 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 subobjects (often with a hierarchy of IS-A-PART-OF one-to-many relationships). In the relational approach each type of subobject has to be retrieved and dealt with separately. It is often awkward, as well, to deal with large type hierarchies resulting from IS-A relationships [21] in the relational approach.

However, the relational approach is quite flexible and can always handle, in some way, a data base consisting of complex objects with IS-A, IS-A-PART-OF, and the other common relationships. It is just that with such data bases a relational system is often more inconvenient and slower than an object-oriented system designed specifically for data bases that emphasise objects, particularly if the host programming language is object-oriented.

But because of the firm foundation underlying the relational approach, the existence of a powerful standard declarative language (SQL) for manipulation of relational data bases, a broad installation base and the fundamental flexibility of the relational approach, 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 [19, 24, 25].

A non first normal form 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, 19, 25]. 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 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. Thus 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.

Relations, objects and declarative languages

It has been argued [8] 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 [16]. The other approach is what might be called a composite object-oriented approach. Currently, the only example of the composite object oriented approach appears to be COOL, designed as a subset of (or extension to SQL) for use with an N^2F object-oriented relational data base [8]. In the composite object-oriented approach, a retrieval language expression is explicitly structured along the lines of the structure of the complex object instance being retrieved, and natural quantifiers [4] that quantify each set of related subobject instances are used to accomplish this. This permits the specification of complex object instances, involving aggregation IS-A-PART-OF and inheritance ISA relationships, that often cannot be specified with SQL.

COOL expressions make use of genitive relations [8], which will be formally defined in this paper, in order to specify relationships unambiguously. It is the reference list attributes allowed in N^2F data bases that make it possible to use genitive relations in declarative languages. The genitive relation is a quite fundamental construct, and in this paper we shall demonstate a composite object-oriented tuple calculus based on genitive relations. This genitive relational calculus is interesting in its own right, but is primarily envisaged as a foundation for genitive relational composite object oriented declarative languages, such as COOL, for

manipulating N^2F relational data bases.

Notation

Upper case letters are used for atomic attributes of relations, and upper case bold is used for names of relations. Each relation is assumed to have an atomic primary key attribute, which corresponds to the system generated unique identifier for an object type in an object-oriented data base. The primary key is underscored, and for reader convenience has the same upper case letter as the relation name. Relation schemes are implied but are not named [21]. Subscripted lower case is used for attribute values, and lower case letters are used for tuple identifiers. Thus the relation scheme [T, U, V, W] could give rise to the relation instance **T**(T,U,V,W), which may have a tuple $t(t_1, u_1, v_1, w_1)$. TX, TY or TZ denotes a tuple variable for the relation **T**. Relations may be unnormalized, in that a non-key attribute is not necessarily atomic, and may contain a set of values. Each tuple of a relation is assumed to represent an object instance, and a non atomic attribute can be a reference list attribute, that is, it may hold a set of primary keys of related objects. Reference list attribute names have two upper case characters, the last one ending in L(ist), such as LL or QL. Other non atomic attribute names have two upper case letters, the last one ending in S(et), such as PS or GS.

1 RELATIONSHIPS

We begin with a general definition of a relationship that allows us to distinguish the many different kinds of relationship. We then define relationship relations for unambiguous specification of relationships. We define sets of related objects in terms of relationship relations and introduce a convenient notation for a such sets. This leads us to the concept of a genitive relation, and allows us to develop an object-oriented tuple calculus.

1.1 Relationship definition

A definition of a relationship of any kind in an unnormalized relational data base is as follows:

> Definition 1 In an unnormalized relational data base with a set of relations [**A, B, C, ...**], there is a relationship between any arbitrary pair of relations (**A,B**) if it is possible to generate a relation **R**(A,B, ...), with A and B as minimal atomic attributes, from the set of relations by a process of relational algebraic operations involving only atomic attributes. **R** is called the relationship relation for the relationship.

Notice that attributes A and B are the primary keys of relations **A** and **B**. Each tuple of **R** relates an A entity and a B entity, and thus must define a relationship.

1.2 Classification of relationships

Relationships can be either primitive or composite. Assume that **p** and * are used for relational algebraic projection and natural join respectively [3, 16].

Definition 2 The relationship between two relations (**A,B**) from a set of relations [**A,B, C, ...**] is primitive if **R**(A,B) = $\mathbf{p}_{A,B}$(**A*B**).

Definition 3 A relationship that is not primitive is composite.

Primitive relationships are simple in concept, and will occur if the two relations involved have at least one common atomic attribute drawn on the same domain [6]. It is this common attribute that makes possible formation of **R** from a single join.

Where the relationship is composite there is no common join attribute, and the relationship is due to a chain of primitive relationships; it thus typically requires a chain of joins to form **R**. Typically we will have **R**(A,B) = $\mathbf{p}_{A,B}$(**A*H*K*L ...*B**)

The primitive relationships are:

(a) Simple 1:n relationship Here one of the relationship attributes is a primary key and the other is not. If A in **A** is the primary key attribute and A in **B** is not, then for a given **A** tuple there may exist many related **B** tuples.

(b) Co-relationship Here neither of the relationship attributes is a primary key. This relationship is common but much misunderstood and little investigated. It is often confused with a many-to-many relationship. It is implicated in the connection trap [6,7] and join dependencies [6].

(c) ISA or inheritance relationship Here both relationship attributes are primary keys. This can be considered to be a trivial 1:1 relationship, and in the relational approach it often is. However, it is non trivial if each of the participating relations participates in a different chain of relationships. In that case the relationship is directional in that one of the relations inherits the properties of the prior relations in the chain [6, 8, 10, 16].

The main composite relationships are:

(a) Composite 1:n relationship There is a composite 1:n relationship between **A** and **Z** if there is a primitive 1:n relationship between **A,B**, between **B,C**, ..., between **X,Y**, and between **Y,Z**, that is, if **A** and **Z** are connected by a chain of 1:n relationships.

(b) Many-to-many or n:m relationship This relationship occurs between **A** and **Z** if there exists a single relation **M**, such that

there is either a primitive or composite 1:n relationship between **A** and **M**, and also a primitive or composite 1:n relationship between **Z** and **M**. If **A** and **Z** are the same relation, the relationship is cyclic or recursive [5].

Note that other, more obscure, composite relationships are a possibility, according to definition 3. In the object-oriented approach, it is only 1:n relationships, many-to-many and ternary relationships derived from two primitive 1:n relationships, ISA relationships (and occasionally co-relationships) that are commonly defined at the conceptual level, by means of collection, or reference list, attributes. Other composite relationships are dealt with in terms of these basic relationships. However other composite relationships could be supported by means of reference list attributes in an N^2F data base.

Definition of unnormalised (N^2F) relational data structure

Definition 4. A set of relations [**A, B, C, ...**] may be said to have an (object-oriented) N^2F unnormalized relational data structure if:

(a) For any arbitrary pair of related relations (**A,B**) with the relationship relation **R**(A,B, ...), with A and B as minimal atomic attributes, there may exist, in relation **A**, a collection, or reference list, attributes, where if LL is such an attribute, for any value a_n of A, LL is the set [RX.B: RX.A = a_n], and there may exist in relation **B**, a collection, or reference list, attribute, such that if QL is such an attribute, for any value b_n of B, QL is the set [RX.A: RX.B = b_n].

(b) There may exist, in any relation, set collection attributes, where such an attribute may have a set of atomic literals as its value.

Clearly, according to the above definition, if the relationship between **A** and **B** were 1:n, then **A**.LL would be a collection attribute referencing a set of related **B** objects, and **B**.QL would be an atomic attribute referencing a single **A** object. If the relationship between **A** and **B** were many-to-many, then **A**.LL would be a collection attribute referencing a set of related **B** objects, and **B**.QL would also be a collection attribute referencing a set of related **A** objects.

But notice that the definition of object-oriented unnormalized relational data structure above does not exactly correspond to what is commonly thought of as an object-oriented data structure. The definition is more flexible, since LL and QL are capabable of being used with any relationship, primitive or composite, and if composite, no matter how obsure.

2 RELATIONSHIPS AND QUANTIFICATION

2.1 Quantifiers

A quantifier is used in natural language to specify a quantity of objects for which a condition holds. However, in conventional mathematical logic quantifier use is restricted to specification of a quantity of members of a set for which some stated property is true. The quantifiers used in logic are further restricted to the existential quantifier and the universal quantifier [12, 17, 18, 22]. The quantity specified by the existential quantifier is at least one of the members of the set; the quantity specified by the universal quantifier is all of the members of the set.

In addition to the universal and existential quantifiers there is a large number of other quantifiers, or natural quantifiers, that are used in everyday language with quite precise meanings [4]. Examples are: for all but one, for one and all, for at least 2, and for a majority of. We shall use the symbolism E(quantity) for quantifiers that specify a predetermined numerical quantity, and the symbolism U() for other quantifiers. In this symbolism U() is the univeral quantifier, and E(>=1) is the existential quantifier. E(>U()/2) is the quantifier for a majority of, U(-2) is the quantifier for all but 2, and E(<=4) is the quantifier for at most 4.

2.1 Sets of related tuples and genitive relations

A set of related tuples may be precisely defined as follows:

> Definition 5 Suppose we have two relations **A**(A,...), **B**(B, ...) related in any relationship for which there is a relationship relation **R**(A,B,...). The set of **B** tuples that are related to any **A** tuple a(a_n,..) in this relationship is the set:
>
> [BX ∈ **B**: E(>=1) RX ∈ **R** (RX.B = BX.B and RX.A = a_n)]
> = [a_n:**R**:**B**] (1)

A convenient notation for the set of **B** tuples related to an **A** tuple a_n for the relationship defined by the relationship relation **R** is [a_n:**R**:**B**]. Where the **A** tuple is identified by the value of a tuple variable AX, the set of related **B** tuples can be denoted by [AX:**R**:**B**]. We call such a set of related tuples, regardless of the type of underlying relationship, a genitive relation [8].

As discussed earlier, if **A** and **B** also constitute a formal object-oriented data structure, then **A** will have a non-atomic set or reference attribute LL, such that for any value a_n of A, LL is the set [RX.B: RX.A = a_n]. It follows from Definition 5 that for any value a_n of A, it is also true that LL = [BX.B: BX C [a_n:**R**:**B**]], or more simply, since [a_n:**R**:**B**] is a subset of R, it is also a relation, so that the collection attribute LL in an **A** tuple with A vaslue a_n is the projection on B given by:

$$[AX.LL : AX.A = a_n] = p_B([a_n{:}\mathbf{R}{:}\mathbf{B}]) \qquad (2)$$

Expression (2) has an important inverse:

$$[a_n{:}\mathbf{R}{:}\mathbf{B}] = [[AX.LL *_B \mathbf{B}] : AX.A = a_n] \qquad (3)$$

which is the expression for the genitive relation that gives the **B** tuples related to the **A** tuple identified by a_n. It follows that the expression for the genitive relation that gives the **B** tuples related to the **A** tuple identified by AX is

$$[AX{:}\mathbf{R}{:}\mathbf{B}] = [[AX.LL *_B \mathbf{B}] : AX.A = AX.A]$$

which reduces to:

$$[AX{:}\mathbf{R}{:}\mathbf{B}] = [AX.LL *_B \mathbf{B}] \qquad (4a)$$

If it be understood that the join is on the identifier attribute **B** of **B** we get a concise and useful notation (and the one used in COOL) for a genitive relation:

$$[AX{:}\mathbf{R}{:}\mathbf{B}] = [AX.LL{*}\mathbf{B}] \qquad (4b)$$

The utility of the notation AX.LL*B for the genitive relation is that it does not involve the underlying relationship relation **R**, which is never given and must always be computed, but involves instead the reference list attribute for the relationship, which is normally always given in an object-oriented data base for important relationships (but not for all possible relationships).

3 GENITIVE RELATIONAL TUPLE CALCULUS EXPRESSIONS

3.1 Tuple variables with genitive relations for 1:n relationships

Suppose an N_2F data base with two relations:

A(A, C, D, E, LL)
B(A, B, H, J)

in a 1:n relationship, where the underscore denotes a primary key, or otherwise uniquely generated identifier, and LL is a reference list attribute listing the primary keys of related **B** tuples. It is possible to construct a predicate calculus using conventional techniques plus genitive relations and natural quantification. Suppose we define:

AX: tuple variable for relation **A**
BX: tuple variable for relation **B**
LY: tuple variable for genitive relation AX.LL*B

This means that for each distinct tuple variable instance AX there exists a genitive relation AX.LL*B, any tuple of which can be the

value of tuple variable LY.

Suppose we are dealing with simple composite objects each instance of which consists of an **A** tuple and zero or more related **B** tuples. Suppose now we want to express the set of D and E values from each object instance for which C is c2 and for which all related **B** tuples have the J value j1. The conventional predicate calculus expression, which is entire-relation oriented, would be:

```
(AX.D, AX.E) : AX.C = c2 and
               U(0) BX (BX.J = j1 or AX.A not = BX.A)        (5)
```

Here U() denotes the universal quantifier. The version using the genitive relation implicit in the situation is:

```
(AX.D, AX.E) : AX.C = c2 and U(0) LY (LY.J = j1)             (6a)
```

Unfortunately, readability of such a concise expression is impaired by the need to know the tuple variable definitions, particularly for tuple variables, such as LY, that range over the tuples of a genitive relation defined by the value of another tuple variable, such as AX. An equivalent but less concise expression requires that tuple variables be defined within the expression:

```
(AX.D, AX.E) : E(>=1) AX ϵ A (AX.C = c2
        and U(0) LY ϵ AX.LL*B (LY.J = j1))                   (6b)
```

Here E(>=1) denotes the conventional existential quantifier. And for the sake of completeness:

```
AX.LL*B = [AX:R:B]
        = [BX ϵ B: E(>=1) RX ϵ R (RX.B = BX.B and RX.A = AX.A)]

where R(A,B) = p_A,B(A*B).                                   (7)
```

which is a formal way of stating that the value of the reference list attribute LL that purports to support the 1:n relationship between **A** and **B** must be consistent with the relationship as defined by the relationship relation.

Notice that, according to the definition for **R**, expression (6b) is valid for the 1:n relationship assumed with expression (7). But if the relationship had been a many-to-many relationship, where both **A** and **B** are 1:n related to **M**, then expression (6b) would still apply if the reference list LL in **A** is valid for that relationship, that is, if:

```
AX.LL*B = [AX:R:B]
        = [BX ϵ B: E(>=1) RX ϵ R (RX.B = BX.B and RX.A = AX.A)]
where  R = p_A,B(A*M*B) instead of R(A,B) = p_A,B(A*B).
```

But expression (7) will apply in the general case, no matter what relationship is involved, provided the reference list LL involved is consistent with the general expression for **R**:

$$\mathbf{R} = p_{A,B}(\mathbf{A*H*K*L} \ldots \mathbf{*B})$$

3.2 Natural quantifiers

If we were to change the quantifier in the above query from **for all** to **for all but 3**, a conventional expression is impossible without some construct for counting tuples. A version using the genitive relation merely requires that the quantifier in expression 6a be changed:

```
(AX.D, AX.E) : AX.C = c2 and U(-3) LY (LY.J = j1)          (8)
```

Furthermore the expression structure in 6a can be used with all the natural quantifiers. As a further example of the use of a natural quantifier, expression 9 specifies those **A** tuples where C is c2 and most, or a majority, of related **B** tuples have a J value of j1 is:

```
(AX.D, AX.E) : AX.C = c2 and E(>)(U()/2) LY (LY.J = j1)     (9)
```

Here E(>)(U()/2) denotes the natural quantifier (exists more than half of all).

The tuple calculus presented here allows for unlimited nesting of quantified clauses. The essential syntax for a genitive relational tuple calculus expression is:

```
<tuple calculus expression>:=
     <TX1>.<attribute1>, <TX1>.<attribute2>, ...:
               <attribute-condition> [<op> <quantifier-condition>]

<quantifier-condition>:=
           <quantifier-condition> <op> <quantifier-condition>

<quantifier-condition>:=
  <quantifier> <<TX2> C <genitive relation> (<attribute-condition>
                                    [<op> <quantifier condition>])
```

<attribute-condition> is a logical expression constructed from simple conditions involving the attributes of a tuple variable;

<op> is logical AND/OR

TX1, TX2 are tuple variables.

As an example of an expression with nesting, suppose we have relations **A** and **B** participating in a 1:n relationship supported by the reference list **A**.BL and relations **B** and **C** participating in a 1:n relationship supported by a reference list **B**.CL:

A(A, C, D, E, BL)
B(A, B, H, J, CL)
C(C S T)

Suppose we define:

AX: tuple variable for relation **A**
BX: tuple variable for relation **B**
LY: tuple variable for genitive relation AX.BL*B
CX: tuple variable for relation **C**
LZ: tuple variable for genitive relation LY.CL*C

Now consider the set theoretic expression for the C and E attribute values from each **A** tuple with D = 4 such that a majority of related **B** tuples have both J = 6 and at least 3 related **C** tuples with T = 8.
This would be written:

```
[AX.C, AX.E : D = 4 and E(>U()/2) LY (LY.J = 6 and
                          E(>=3) LZ (LZ.T = 8))]        (10)
```

Notice that a tuple calculus expression based on genitive relations is intrinsically object oriented. In constructing the expression the writer focusses on a composite object. In the above expression the writer focusses on a specific **A** tuple first, then on a quantity of related **B** tuples, and then for each such **B** tuples, the writer focusses on a quantity of related **C** tuples. This is why the tuple calculus is useful as a basis for constructing a natural quantifier object oriented declarative language, such as COOL, for N^2F relational data bases.

3.3 An object-oriented declarative data base language

One experimental data base language that can be based on the above genitive relational tuple calculus is COOL. COOL has been described in detail elsewhere [8]. The tuple calculus basis for COOL can be seen from the COOL expression for expression (9) above:

```
Select * from [each] A [object]
where D = 4 and
      for most of its [related] [A.]BL*B [objects] (E = 6)     (11)
```

The words in square brackets may be omitted. Here the reference attribute BL, that determines the relationship in the object-oriented approach, is used to specify a genitive relationship for the implcit tuple variable **A**, which is also a relation name. As in SQL, in COOL, relation names serve as implicit tuple variables. But strictly <**A** object> is the tuple variable, and not **A**. Since the genitive relation name **A**.BL*B is a relation name, it too serves as an implicit tuple variable, whose values are the same as some <**B** object> tuple variable values.

As an example of how expressions are nested in COOL, expression [10] is written as:

```
Select C, E from [each] A [object]
where D = 4 and
   for most of its [related] [A.]BL*B [objects]
```

(J = 6 and for at least 3 [**B.**]CL.**C** [objects] (T = 8)) (12)

Just as SQL is based on predicate calculus [18, 21], with relation names serving as implicit tuple variables, so too, COOL is based on a predicate calculus that allows genitive relations, but with both relation names and derived genitive relation names serving as implicit tuple variables [8]. This must never be forgotten when constructing and interpreting COOL expressions and designing reduction systems for them.

3.4 Set expressions

Suppose an N_2F data base with two relations:

A(A, C, D, KS, E, BL)
B(A, B, H, VS, J)

in a 1:n relationship, where KS and VS are set attributes. The genitive relational calculus can handle such set attributes, and predicates involving sets. As before, we define:

AX: tuple variable for relation **A**
BX: tuple variable for relation **B**
LY: tuple variable for genitive relation AX.BL*B

Set inclusion conditions are handled conventionally. For example:

(AX.D, AX.E) : (k_2, k_3, k_5) $\subseteq$ AX.KS (13)

The expression specifies D and E attribute values from each tuple whose set KS attribute contains the specified set of literal values.

Even when not dealing with collection (set) attributes, there is still a general need for set theoretic constructs with relationships in object-oriented manipulation languages, which we now address. For example, the retrieval:

Retrieve the D and E attribute values from each **A** tuple for which C has value c2 and for which the J attribute values in related **B** tuples have at least the values j1, j4, j7 provided the H attribute value in those related **B** tuples is equal to h3.

This quite complex retrieval can be expressed quite concisely using the genitive relation for the relationshipo between **A** and **B**:

(AX.D, AX.E) : AX.C = c2 and
(j1, j4, j7) $\subseteq$ (LY.J: LY.H = h3) (14a)

If the tuple variables are defined within the expression, it can be written:

(AX.D, AX.E) : E(>=1) AX $\in$ **A** (AX.C = c2
and E(>=1) LY $\in$ AX.BL*B ((j1, j4, j7) $\subseteq$ (LY.J: LY.H = h3)) (14b)

The syntax given earlier needs to be expanded to cover set conditions, as follows:

```
<attribute condition> := <atomic set> <set compare op> <atomic set>
<atomic set> := <set of literal values> /
                <set collection attribute> /
                <tuple calculus expression>
```

The <tuple calculus expression> term should involve only one attribute.

3.5 Composite objects

No conventional retrieval language the author is aware of does justice to retrieval and definition of composite objects. 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 [19]; 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 composite view, that is, of the selected object. A composite object, or 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 a2, 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 object instance 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 com-

posite 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
```

The genitive relational tuple calculus can be used to specify composite objects and also to specify the concentration of such composite objects. An example will illustrate the essence of the approach.

Suppose the we have the relations:

A(A, D, E, BL, CL)
B(A, B, H, J, XL, YL)
C(A, C, S, T)
X(B, X, V, W)
Y(B, Y, P, Q)

where we define:

AX: tuple variable for relation **A**
BX: tuple variable for relation **B**
LY: tuple variable for genitive relation AX.BL***B**
CX: tuple variable for relation **C**
LZ: tuple variable for genitive relation AX.CL***C**
XX: tuple variable for relation **X**
MX: tuple variable for genitive relation LY.XL***X**
YY: tuple variable for relation **Y**
MY: tuple variable for genitive relation LY.YL***Y**

(1) Using the following selection conditions we want to specify the selection of a set C1 of composite object instances, or XNF view, such that each composite object instance consists of an **A** object with related **B**, **C**, **X**, and **Y** objects with the following properties:

* An A object has E value e3.
* At least 2 of its related B objects have H value h1.
* Most of its related C objects have T value t2
* All of the X objects related to a B object in the composite object have W value w2.
* Exactly 3 of the Y objects related to a B object in the composite object have P value p2.

(2) From each composite object instance in C1, where possible, we want to specify a concentrated object instance, concentrated using the following concentration conditions

* Only A, and D attributes (from **A**) are included
* Only B, H and J attributes from each related **B** object are included provided all its related **X** objects have V value v1.
* All **C** attributes are excluded.
* Only B and W attributes from **X** are included.
* Only Y and P attributes from a **Y** object instance are included provided the Q attribute has value q4.

This set of concentrated composite object instances, or concentration, is to be named CC1.

This cannot be specified in conventional SQL, nor even the extended SQL of Starburst, although COOL could manage it. In practice, in engineering, composite objects with many hierarchical levels have to be managed. The operation is specified in genitive relational tuple calculus with a composite object selection step followed by a concentration step. No substantial new syntax is needed. To extract the necessary composite object instances or XNF view, we specify

```
C1 =
      [AX.A, AX.D, AX.E, AX.BL, AX.CL
       LY.A, LY.B, LY.H, LY.J, LY.XL, LY.YL
       LZ.A, LZ.C, LZ.S, LZ.T
       MX.B, MX.X, MX.V, MX.W
       MY.B, MY.Y, MY.P, MY.Q    :
      AX.E = e3 and E(>=2) LY (LY.H = h1 and U(0) MX (MX.W = w2)
                                          and E(3) MY (MY.P = p2) )
                 and E(>U(0)/2) LZ (LZ.T = t2)  ]
```

This operation is selecting each hierarchy type, with an **A** tuple root, from the data base, provided the hierarchy type complies with the conditions stated.

To specify the concentrated object instances CC1, we use the tuple variables defined above, with the stated understanding that these will be applied only to **A**, **B**, **C**, **X**, and **Y** tuples occurring within the composite object C1:

```
CC1 =
  [C1.AX.A, C1.AX.D
   C1.LY.B, C1.LY.H, C1.LY.J : U(0) C1.MX (C1.MX.V = v1)
   C1.MX.B, C1.MX.W
   C1.MY.Y, C1.MY.P : C1.MY.Q = q4]
```

A further concentration of CC1 could also be specified.

The above expressions demonstrate the importance of being able to specify the set of tuples related to a specific parent tuple in a direct and concise manner using the genitive relation construct and natural quantifiers. Without this direct specification capability, it would not be possible to specify any arbitrary concentrated composite object type. Although there are only three 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 ob-

jects the genitive relational tuple calculus is powerful and versatile - as can be languages based on it, such as COOL, and more so than any proposed SQL extensions in the literature. The reader is also invited to attempt specification of CC1 using SQL - it simply cannot be done.

Language constructs for handling inheritance

No additional language constructs are needed for manipulation of inheritance. Suppose we have:

A(A, D, E, BL, CL)
B(B, H, J, AL1)
C(C, S, T, FL, AL2)
F(C, F, V, W)

where C is drawn on the same domain as A, such that a **C** object inherits the properties of a parent **A** object. Note also that the relationship between **A** and **B** is many-to-many. AL1 and AL2 are list reference attributes referring to **A** tuples.
[To follow the discussion, it might help to consider an **A**-tuple as describing a document of any kind, and a **B** tuple as a document author, so that a document can have many authors and an author can write many documents; a **C** tuple could describe a particular type of document, such as a computer program, so that a **C** tuple inherits the properties of a related **A** tuple; an **F** tuple could describe a program execution, so that a particular program can have many executions.]
Suppose also we define:

AX: tuple variable for relation **A**
LV: tuple variable for genitive relation AX.CL*C
LW: tuple variable for genitive relation LY.CL*C
BX: tuple variable for relation **B**
LY: tuple variable for genitive relation BX.AL1*A
LX: tuple variable for genitive relation LV.FL*F
LZ: tuple variable for genitive relation LW.FL*F

As a simple example of the use of inheritance, consider the request:

> Specify the B and J values for each **B** tuple related to a **C** tuple with at least two related **F** tuples with W value w1.

This must necessarily involve the fact the a C tuple inherits the properties of its parent **A** tuple which in this case is related to one or more **B** tuples. Put differently, between **A** and **B** there is a many-to-many relationship, and therefore, via inheritance, also between **B** and **C**, which is the relationship required for the request. The request can be concisely expressed as:

```
[BX.B, BX.J: E(>=1) LY (E(1) LW (E(>=2) LZ (LZ.W = w1)]
```

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 each **C** object that has a related **F** object with W value w1.

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

```
C2 =
          [AX.A, AX.D, AX.E, AX.BL, AX.CL,
           LV.A, LV.S, LV.T, LV.FL, LV.AL2 :
            E(>=1) LX (LX.W = w1)  ]
```

4 CONCLUSIONS

The expressive power of a genitive relational tuple calculus, as applied to an object-oriented N2F relational data base, has been demonstrated. The genitive relation construct in the calculus allows for quantification of relationships by means of any of the natural quantifiers, and for concise specification of quite complex objects. In contrast, conventional tuple calculus expressions as applied to relations are limited primarily by a lack of object-orientation, and, secondarily, by a consequent quite rigid convention that quantifiers quantify only entire relations by means of only existential and universal quantifiers. Declarative data base languages, such as SQL, that are based on conventional tuple calculus consequently also suffer from this lack of object-orientation.

Just as conventional tuple calculus with normalized relational data bases is the basis for SQL, genitive relational tuple calculus can be used as a theoretical foundation for such languages as COOL (composite object-oriented language), a recently developed [8] genitive relational object-oriented natural quantifier data base language for use with N2F data bases as a subset of SQL.

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. Aho, A. V., Beeri, C., and Ullman, J. D. The theory of joins in relational data bases, ACM Trans on Data Base Sys., 4(3), 1979, 297-314.

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

5. Bradley, J. Recursive relationships and natural-quantifier set theoretic expression techniques, Computer Journal, to appear, 1992.

6. Bradley, J. Polygonal join dependencies, closed co-relationship chains, and the connection trap in relational data bases, Computer Journal, 31(2), 1988, 141-46.

7. Bradley, J. Co-relationships, levels of significance, and the source of the connection trap in relational data bases, Computer Journal, to appear, 1992.

8. Bradley, J. Genitive relations and the composite-object oriented language COOL for object support in N^2F relational data bases, Res. Report 92/482/20, Dept. of Computer Sci., Univ. of Calgary, 1992, to be published.

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. Chamberlin, D.D., et al. SEQUEL 2: A unified approach to data definition, manipulation and control, IBM J. Res. & Dev., 20(6), 1976, 560-575.

12. Chang, C. L. DEDUCE-2: Further investigations of deduction in relational data bases, in "Logic and Databases", Plenum Press, 1978 (Editors: H. Gallaire, J. Minker).

13. Codd, E. F. Relational completeness of data base sublanguages. In "Database Systems", Courant Computer Science Symposia, Vol. 6, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

14. Codd, E. F. Extending the relational data base model to capture more meaning, ACM Trans. on Database Sys., 4(4), 1979, 397-434.

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

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

17. Gallaire, H., Minker, J. and Nicholas, J. M. Logic and data bases: A deductive approach, ACM Comput. Surveys, 16(2), 1984, 153-185.

18. Hilbert, D. and Ackerman, W. Principles of Mathematical Logic, Chelsea Publishing Co., New York, 1950.

19. 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.

20. Luk., W.S. and Kloster, S. ELFS: English language from SQL, ACM Trans. on Database Sys., 11(4) 1986, 447-472.

21. Maier, D. Theory of Relational Databases, Computer Science Press, Potomac, Maryland, 1983.

22. Rybinski, H. On first-order-logic databases, ACM Trans. on Database Sys., 12(3), 1987, 325-349.

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

24. 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.

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

26. Ullman, D. Implementation of logical query languages for data bases, ACM Trans. on Database Sys., 10(3), 1985, 289-321.

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