**An extended relational data model and SQL subset with support for containment, composite structures and inheritance**

*J.Bradley*
*Department of Computer Science*
*University of Calgary*
*Calgary, Alberta, Canada*

***ABSTRACT*** An extended relational data base model and SQL subset is described. The data model is fully relational. It allows unnormalized relations only where attributes are sets or lists of atomic attributes. There is support for both inheritance and composite entities. A composite structure can be stored by decomposition into sets of tuples stored in relations but can also be stored virtually as a view. Composite structures, including OOPL objects, can be transferred directly between database system and program variables. Manipulation languages are an extended SQL and a separate new SQL language subset for efficient manipulation of containment structures, whose manipulation with SQL is error-prone. The language subset has a predicate calculus expression structure, but allows the use of genitive relations to model containment relationships and facilitate application of natural quantifiers to containment sets. Range variables are allowed and can range over genitive relations as well as base tables.

***KEY WORDS*** Containment, database, declarative language, genitive relation, quantifier, query language, relation, predicate calculus

## INTRODUCTION

This paper concerns the design of an experimental extended relational database system called ComposeR. This relational database system embodies an extended relational data model that can be manipulated by SQL and by a separate and new subset of SQL This SQL subset, called COOL (Containment oriented Language) has non-SQL features for dealing with containment structures, composite entity instances and inheritance supertype-subtype hierarchies. COOL is designed for use with extended-relational data bases, that is, data bases with an underlying data model that is relational but which allows for extensions of the classic Normal Form relations, to permit support of, among other extensions, list attributes, set attributes, virtual attributes, list relations, bag relations, supertype and subtype relations, and user-written functions.

Extended relational database systems [SRL+90, Kim92] are being researched because the conventional relational approach [Mai83, Dat95], does not handle databases representing complex composite entities efficiently. This has become a problem since data bases representing complex entities have become increasingly important in recent years, particularly in engineering, scientific, financial and econometric applications.

The need for efficient manipulation of complex-entity databases is being filled, to some extent, by object-oriented programming language (OOPL) data base systems or ODBMSs [ABD+89, Ban93, Bro91, Cat91, US90, ZM89]. However ODBMSs have evolved from the need to persistently store the class instances or objects used in OOPLs rather than records or tuples of relations (rows of tables). The OOPL object or class instance, which is essentially a structure variable whose members can be accessed only via associated functions, was motivated by the need for efficient construction of complex software systems with extensible and reusable modules (objects). The OO paradym involves use of highly structured programs consisting of objects that communicate with each other via messages (function calls to the associated functions). These objects are also based on object types or classes that can be reused in different systems, both directly and within derived objects. ODBMSs are commonly used to persistently store objects or class instances, used in a specific programming language, such as C++. For this reason, ODBMSs tend to interface only with one specific OOPL and to do so using the OOPL constructs in a near seamless manner; in other words, they tend to use the OOPL as the database manipulation language. This is appealing to the OOPL programmer, resulting in greater programmer productivity and fewer programming language-database system interface errors. Some important ODBMSs are Gemstone [KL89c, BOS91], O2 [BBB+88, LRV89, Deu91] and Orion [KL89b, Kim90].

Nevertheless, despite success in applications involving OOPLs, the initial apparent narrowness of the ODBMS approach has distinct disadvantages where the database needs to be shared among a wide variety of users with different programming languages, some OO and some not, as well as users who want to execute direct queries. It is in shareability and ability to carry out direct queries that the relational approach excels. This has given rise to research into extended relational systems and thus two competing trends. On one hand ODBMS vendors and researchers are continually attempting to broaden the appeal of their systems by allowing for more than one programming language, abeit more than one OOPL, as well as by introducing OOPL-flavored query languages with some ressemblence

to SQL such as OSQL in ODMG-93 [Cat93, Kim94]. In constrast proponents of the relational approach are extending their systems (extended relational database systems (ERDMS)) with two main goals. One is to make the ERDMS more compatible with the persistent object storage and retrieval needs of OOPL programmers. The other is to enable it to deal more efficiently with complex and other specialized entities while retaining the broad sharability and power of conventional relational systems, as well as upward compatibility.

An ERDMS is sometimes called an object relational database system. However this term is misleading since it tends to imply that an object-relational data base system is an object-oriented data base system; this cannot be the case since it is not designed to hold arbitrary class instances, as is the case with an ODBMS. ComposeR is not an ODBMS, although, as we shall see it can store some quite complex programming language objects (or class instances). ComposeR is designed as an ERDMS.

Extended relational data models are extensions of the conventional relational data model devoloped by Codd and others [Cod81, Dat95]. According to Codd, any data model is a collection of allowed data object types, a collection of required integrity rules, and a collection of allowed operators. Briefly, in the conventional relational model, the object data types are normalized relations and the allowed operations are those of conventional relational algebra; any expression in SQL can be reduced to a relational algebra routine.

Extensions to the conventional relational model tend to draw on the concepts of the semantic models [HR87]. The most important semantic model is the Entity Relationship model [Che76] and enhanced versions. Others are the Functional Data model [SHI81], and the Semantic Data model [HM81]. Semantic data models were directed at the goal of capturing more meaning in the data model, so that semantic models have a higher level of abstraction than the relational model. The Entity-Relationship (ER) model allows for entities, which can be modelled by relations, and relationships or associations between entities [Ull88]. The ComposeR data model can be viewed as an ER model on a relational foundation.

Some important extended relational models that have been developed are the Relational Object Model [SS90, SS91], the POSTGRES data model [Cat91, SR86, SK91, Sto87], IBM's Starburst data model [Cat91, LLPS91, LH90] and Kim's Unified Relational and Object-oriented data model [Kim92]. Starburst supports a rich type system, enhanced performance features, encapsulation of behaviour with data, identifiers for stored entites, large structured complex entitiess, and a declarative language that is an extension of SQL. An extension of SQL, called SQL/X, is used as the declarative language in Kim's Unified Relational and Object-oriented model [Kim92]. The ISO draft standard, commonly referred to as SQL3 [Mel94], is an extended relational data model that allows for storage of relations in which a tuple is an instance of an abstract data type.

The ComposeR data model is mostly similar to those of other extended relational systems, and is discussed later; what is novel is the features of the declarative language COOL. However, although the language is separately from SQL, it is not intended as a stand-alone language. Rather the intention is that its features be seamlesslessly incorporated into an extended SQL.

**Motivation**

The fundamental motivation for the features of COOL is the desire to eliminate from predicate calculus, and the SQL derived from it, the error-causing and time-consuming contraint of entire relation-quantification in all cases where the quantification involves a containment entity in the broadest sense. This statement requires some explanation.

When we refer to a *containment entity* we can mean an entity that physically contains one or more other entities within it, that is, physical containment, as with a building containing offices or a park containing lakes. But it is also very useful to include such cases as (a) *physical attachment,* as in the integrated circuits attached to a circuit board, or pins attached to an integrated circuit, (b) *necessary proximity,* as in moons orbiting a planet or passengers on a ship and (c) *generational proximity* as in the children biologically generated from a parent, or parts made by a manufacturer, (d) *posessional or control proximity* is in the books possessed by a person, or bananas possessed by a primate.

As an example of the problems that occur with whole relation quantification in the case of containment enties, consider the following simple relational database:

*Planet (pname, diam)*
*Moon(mname, pname, diam, dist, color)*
*Lunarcrater(cname, mname, diam)*

with range variables:

*Range Planets P, Moons M, Lunarcraters L*

From an entity-relationship viewpoint, for one planet there can be zero or more moons orbiting the planet, and for one moon zero or more lunar craters on that moon. Suppose we wish to specify:

*The name of each planet of diameter less than 10,000 miles whose moons are all less than 100 miles in diameter and less than 1000 miles from the planet's surface.*

In conventional predicate calculus there are only two ways to specify this, either (a) using the universal quantifier *for all* with a Horn clause, or (b) using a negated existential quantifier and a negated search condition, a double negative, as in:

*(a) { P.pname: P.diam < 10,000*
*and for all M(P.pname != M.pname or M.diam < 100 and M.dist <1000)}*
*(b) {P.Pname: P.diam <10000*
*and not exists M(P.pname = M.pname and M.diam!<100 or M.dist !<1000)}*

Because the need for a Horn clause with the universal quantifier can mystify many users and is also error-prone, in deriving SQL from predicate calculus, the SQL designers omitted the universal quantifier from SQL, thus requiring users to rely on the negated existential quantifier SQL equivalent to handle universal quantification:

```
Select P.pname from Planet P
 where P.diam < 10000 and not exists (select *
                                      from Moon M
                                      where M.pname = P.pname
                                  and M.diam. >= 100 or M.dist >=1000)
```

Unfortunately, this negated existential quantifier version too is error prone, in four important ways:

(a) *Need for De Morgan's Rules with compound negated search conditions.* Since it is always necessary to negate the search condition following the quantifier, this means application of De Morgan's Rules if the condition is compound. Thus we must negate the search condition

*not (M.diam < 100 and M.dist <1000)*

and write it in the form:

*M.diam. >= 100 or M.dist >=1000)*

(b) *Misinterpretation of universal quantifier semantics.* It is quite possible that the user will misinterperte the meaning of the mathematical universal quantifier used in predicate calcululus and consequently the meaning of its negated existential quantifier equivalent. This error can have fatal consequences in military, aeronautical and medical applications. For example, in the retrieval above, assuming the database is limited to the known planets, we would expect retrieval of Mars, which has two small moons. But most users would not expect the result actually retrieved, which would be Mars, Venus and Mercury, even though the latter two planets have no moons. This result is due to the correct semantics of the mathematical universal quantifier, which strictly means that where x ranges over set X, the expression *for all x (<condition>)* is true if either the condition is true for every member of X or if X is empty, as does the negated existential quantifier equivalent *not exists x (not <condition>).*

The problem is that in everyday language the quantifier for all is usually taken to mean English language quantifier *for one and for all,* so that the search condition is expected to be true for one moon and for all moons. The correct predicate calculus and SQL expressions to avoid retrieving planets with no moons is :

*{ P.pname: P.diam < 10,000*
*and for all M(P.pname != M.pname or M.diam < 100 and M.dist <1000)*
*and exists M(P.pname = M.pname)}*

```
Select P.pname from Planet P
where P.diam < 10000 and not exists (select *
                                     from Moon M
                                     where M.pname = P.pname
                                   and M.diam. >= 100 or M.dist >=1000)
  and exists (select * from Moon M
              where M.pname = P.pname)
```

(c) *Propagation of De Morgan's Rules in nested expressions* If there is additional nesting of a quantifier expression, De Morgan's rules propagate down to it too, in a manner that even expert SQL programmers find very difficult. For example, suppose that we have a retrieval involving the third relation Lunarcrater:

*Get the name of each planet of diameter less than 10,000 miles all of whose moons are less than 100 miles in diameter, are less than 100 miles from the planetary surface and have all their craters less than 10 miles in diameter.*

The SQL is:

```
Select P.pname from Planet P
where P.diam < 10000 and  not exists (select * from Moon  M
                                      where M.pname = P.pname
 and M.diam >= 100 or M.dist >=1000 or exists (select * from Lunarcrater L
                                               where L.mname = M.mname
                                               and  L.diam >= 10)
```

Notice that even though the inner select-block involves a universal quantifer expression normally requiring a negated existential quantifier, De Morgan's rules flowing from the condition negation of the containing select-block require that in this case we do not negate the quantifer but do negate its condition *L.diam < 10.*

This is bad enough, but we must also ask if we are really retrieving what we want. We have two univeral quantifer for all expressions, one nested within the other. But it could be that one or both of the quantifiers really should be *for one and all.* This give us four posible SQL expressions to carry out the retrieval, (a) one for outer *for all* and inner *for all*, which is the above expression, (b) one for outer *for all* and inner *for one and all,* (c) one for outer *for one and all* and inner *for all* and (d) one for outer *for one and all* and inner *for one and all.* These latter three each require more complex individual SQL expressions than the one above, whose intricacies we leave it to the reader to ponder.

(d) *Failure to understand subset quantification.* This difficulty has to do with quantification of a subset of a set of contained entities, for example the condition *all the planet's blue moons are within 100,000 miles,* as constrasted with *all of the planet's moons are within 100,000 miles.* This will be discussed in Section 1.5.

In the case of planets and moons, and similar cases, these difficulties all arise from the requirement in conventional predicate calculus that every tuple in the Moon relation be quantified in a predicate calculus expression. There is nothing fundamentally wrong with this. It is one way of taking containment relationships into account. But it is not the only way. The other way, which we call the natural quantifier approach to expressing containment relationships, is the way used in natural language, the way that corresponds to how most people think about containment entities, and the way that handles the relationships inherent in containment entities as economically as possible.

Nature's way, as evolved in natural languages, involves the use of a large array of quantifiers. Each of these natural quantifiers has a simple, precise and unambiguous meaning, which is lacking in the mathematical universal quantifier. Indeed, it is worth noting that it is not possible to construct an English-language quantifier phrase for non technical users that explicitly and correctly defines the meaning of the mathematical quantifier *for all* used in predicate calculus. The best the author can do is *for all of the members of a set whether or not the set is empty*, or *for all of a quantity, whether or not the quantity is zero.* Furthermore the author is convinced that since nature has seen fit to evolve language constructs for dealing with containment entities in this alternative

(natural language) manner, this alternative manner must be best suited to ensure the survival of its users. Clearly if nature ever did evolve a language in the distant past that used the conventional predicate calculus approach to that part of natural language concerning containment entities, that language (and perhaps also its proponents) did not survive.

The solution to the problem is to modify predicate calculcus to allow for containment relationships. Briefly the solution involves allowing for containment relationships defined in terms of the containment entities, as follows:

*Planet (pname, diam)*
*Moon(mname, pname, diam, dist, color)*
*Range Planet P; Moon M*
*Containment tuple set:*
*[P::Moon] = {M: M.pname = P.pname}*
*Range [P:Moon] m*

*[P::Moon]* is (arbitrary) notation for the *containment tuple set* for the containment relationship between Planet and Moon, and is a variable contents set of tuples; *[P::Moon]* is thus a variable relation, equal to the set of Moon tuples related to a specific P tuple, which can vary. Natural language has an an equivalent construct, namely P's Moons, or Planet's Moons. This containment tuple set, also called a *genitive relation,* would be equivalent to the *path expression* of some ODMSs, for example as in OQL[Kim 94].

Using this construct we can construct a tuple calculus expression for the first query above without the need for a Horn clause;

*{P.pname: P.diam < 10,000*
*and for all m (m.diam < 100 and m.dist <1000)}*

Unlike M in the earlier version, which ranges over the entire relation Moon, the range variable m ranges only over the Moon tuples related to or contained in P, that is, only over P's moons. If we interpret *for all* conventionally as the universal quantifier of mathematics, the expression will retrieve Mars, Venus and Mercury. To make sure we specify only planets with moons, we could use the natural quantifier *for one and all:*

*{P.pname: P.diam < 10,000*
*and for one and all m (m.diam < 100 and m.dist <1000)}*

This clearly solves the first two difficulties discussed above, the need for negated condition or Horn clauses, and also the second problem of the unintuitive meaning of the mathematical universal quantifier. It also solves the problem of flow through to a further nested quantifier expression. For example the second retrieval above can be expressed as

*(a) {P.pname: P.diam < 10,000*
*and for all m (m.diam < 100 and m.dist <1000*
*and for all c (c.diam < 10))}*

Here c must be defined as ranging over the containment set *(M::Lunarcrater)*. Here we have no negations, no Horn clauses and no need for De Morgan acrobatics, and we can replace *for all* by f*or one and all* where required by more precise retrieval semantics.

But there is an additional bonus from doing things this way. We are no longer restricted to the universal and existential quantifiers. We can use any of the large array of natural quantifiers without having to alter the expression structure. For example, the semantics of:

*{P.pname: P.diam < 10,000*
*and for all but one m (m.diam < 100 and m.dist <1000*
*and for a majority of c (c.diam < 10))}*

should be obvious to the reader . The expression *for all but one m (<condition>)* is a logical expresion that is true if every single m tuple except one obeys the condition. The expression *for a majority of c (<condition>)* is true if more than half of all the c tuples obeys the condition.

In passing it should be pointed out that a restricted quantifier *all* has been introduced into SQL. This quantifier is very retricted in its use, is error prone, and is not the equivalent of the universal quantifier of conventional predicate calculus, as is the case for the existential quantifier *exists* in SQL. For these reasons, which space does not permit us to discuss in detail, the author believes that the the incorporation of this *all* quantifier in SQL was a serious mistake, and that it's use should be avoided.

In this paper we discuss COOL and its implementation at length. COOL is SQL-like sublanguage directly derived from the modified predicate calculus outlined above, instead of conventional predicate calculus from which SQL is derived. Our ultimate goal is incorporation of COOL features into SQL with SQL-compatible syntax in a seamless manner. A later paper will show how this can be done.

## 1. COOL and Extended relational algebra.

### 1.1 Summary of the ComposeR data model

COOL can be used to manipulate both data representations of simple entities and composite entities. In ComposeR simple entities are modeled as tuples of relations, a collection of such entities being modeled as a relation. A relation may have the usual atomic attributes, or an attribute that is a set or list of atomic attributes, or virtual or derived attributes represented by functions whose arguments are other attributes. A relation may not have an attribute that is a set of tuples. Thus a relation may be unnormalized to a restricted degree. An attribute may also be an aggregation of simpler attributes, for example Date and Address. In addition, the tuples of a relation can be defined as instances of an abstract data type, provided the types within the ADT do not involve sets of structures or repeating groups of aggregates.

If an entity type is a subtype of another supertype entity, the corresponding subtype relation can be defined as inheriting the attributes and other properties of the corresponding supertype relation. Thus if we have relation *R(a, b, c(), d())*, we can define a relation *S (v, w()) isa R*, so that S inherits the properties of R and has attributes *S(a, b, c(), d(), v, w())* and participates in the relationships of R.

A composite entity can be modelled only as a collection of tuples from relations that model related entities. However there is no relation structure allowed in ComposeR that can model a composite entity. For example, a planet with its moons would be an example of a fairly simple composite entity. An engine with all its subcomponents and components etc would a complex composite entity. In a ComposeR database the component tuples of a composite entity representation must be stored in their respective relations.

Simple entities can be related in one-to-many and many-to-many relationships, both non-recursive and recursive; there will also be ISA-generalization implicit relationships between base and derived relations (as with R and S above). Entities in a one-to many (1:n) relationship are referred to as *parent* and *child* entities. Relationships are enabled by reference lists of unique tuple identifiers. Relationships can also be enabled by the convenional method of matching attribute values between tuples. ComposeR also supports a type of relation called a genitive relation, which will be discussed presently.

The ComposeR data model supports 1:n and n:m relationships, as well as recursive 1:n and n:m relationships, by means of genitive relations. It also supports composite entity representations, this support being (a) the capability to retrieve composite entities (each composite entity consisting of a collection of tuples of different types) and deliver them to matching programming language data structures (complex structure variables, but also OO class instances) (b) to create composite entity views, and (c) to accept a composite entity from a programming language data structure and store it (virtually) in a view as well as decomposing it into its component tuples for actual insertion into the data base relations or tables. This feature will be described later.

There are the usual system provided aggregation functions (such as *count(), sum(), avg(),* and so on); user defined functions can also be used with conditions in COOL and SQL expressions (for example the function overlap() might be used to test for overlap of one crater with another in a retrieval condition). The implemented version of COOL described in this paper can manipulate 1:n, m:n, recursive 1:n and recursive n:m relationships.

### 1.2 Overview of COOL and genitive relations

As an example of a ComposeR database, consider a database for simple entities Planet, Moon, and Lunarcrater where a planet can have many moons, and a moon many lunar craters. In addition we might want to have the specialization entity Ringplanet where a Ringplanet is a Planet, and a Ringplanet has many rings (Ring relation).

*Planet (pname, diam, volume(), moonlist)*
*Ringplanet (include superentity Planet attributes, nrings, ringlist) isa Planet*
*Ring( r#, pname, radius, width, color)*
*Moon(mname, pname, diam, dist, craterlist)*
*Lunarcrater(cname, mname, diam, rim, area())*

Thus a composite entity could be Venus, with no rings and no moons, or it could be Mars, alongs with its two moons and their respective craters, or it could be Saturn along with its rings and with its moons and their respective craters

In the implemented system *pname, r#, mname* and *cname* would be primary keys, accessible by the user. In addition, a user-inaccessible unique tuple identifier is generated for each tuple in the data base. The optional reference list attribute *moonlist* in a Planet tuple will contain a list of the identifiers of related or "contained" Moon tuples, to help enable the 1:n relationship; similarly the optional attribute *craterlist* in a Moon tuple will contain a list of identifiers of related Lunarcrater tuples. The value of a reference list attribute like *moonlist* cannot be accessed by the user, although the name can be used to specify a genitive relation, as described presently. The flavor of COOL, compared with that of SQL, is similar to the modified predicate calculus above and is exemplified in the following query:

*Get the name of each planet of diameter less than 10,000 miles whose moons are all less than 100 miles in diameter, and all with craters less than 10 miles in diameter.*

The COOL expression is:

```
Select P.pname from Planet P where P.diam < 10000
        and for all P's Moons M (M. diam < 100 and
                for all M's Lunarcraters C (C.diam < 10))
```

The range variables P and M can be omitted and the relation name Planet and genitive relation name Planet's Moons used as implicit range variables:

```
Select pname from Planet where diam < 10000
        and for all Planets's Moons (diam < 100 and
                for all Moon's Lunarcraters (diam < 10))
```

A more basic version uses the formal genitive relation name *Planet.moonlist*Moon* instead of the genitive relation name alias *Planet's Moons,* and similarly uses the formal name *Moon.craterlist*Lunarcrater* instead of the alias *Moon's Lunarcraters.*

Note that the inheritance of a planet's properties by a ringed planet will be supported by the system, so that both tuples of the relation Planet and Ringplanet will be considered for retrieval, and not just Planet as stated in the query. Thus both Planet and Ringplanet tuples can be retrieved.

In the above retrieval both quantifiers were *for all.* As with the modified predicate calculus above, the whole range of natural quantifers is available for use with COOL expressions. For example, suppose we alter quantifiers in the above query giving the query:

*Get the name of each planet of diameter less than 10,000 miles one and all of whose moons are less than 100 miles in diameter and have a majority of craters less than 10 miles in diameter.*

The COOL expression requires only quantifier modification:

```
Select pname from Planet where diam < 10000
        and for one and all Planets's Moons (diam < 100 and
                for majority of Moon's Lunarcraters (diam < 10))
```

A quite different and much more complex structure is required for the corresponding SQL expression.

All of the natural quantifiers of the English language, can in theory be used with COOL. The prototype implementation of COOL described in this paper permits14 quantifiers including the universal and existential quantifiers of conventional predicate calculus.

### 1.3 COOL and the Genitive Relation Concept

The concept of a *genitive relation,* corresponding to the containment set in the modified perdicate calculus above, and also to the genitive case construct in natural languages, is fundamental to COOL. In a COOL expression, the use of a genitive relation makes it possible to refer unambiguously to a set of child tuples related to a specific parent in a 1:n relationship (or to a set of tuples of any relation P that are related to a specific tuple of relation Q, with a many-to-many relationship between P and Q entities). For each instance of a given entity type, in order to specify a quantified cross reference (or *xref)* involving a specific quantity (specified by the quantifier) of related entity instances that satisfy a condition, COOL uses the syntax construct:

*<xref>:: <quantifier><related-entities><(condition>)*

Here <quantifier> denotes any natural quantifier, and <related-entities> a genitive relation. The genitive relation defines a specific relationship between two object classes, since there could be more than one relationship.

Consider the 1:n relationship above between Planet and Moon entities. The value of the reference attribute (or reference list) *moonlist* in a Planet instance (modelled as a tuple) that defines the relationship is the set of identifiers of the Moon instances that are contained in (belong to) the specific Planet instance. In this case the COOL syntax to formally specify the genitive relation is *Planet.moonlist*Moon.* This denotes a join of the set of values within the list attribute *Planet.moonlist* with the relation Moon, giving a set of Moon tuples that are related to the specific Planet instance.

Thus a genitive relation is a set of tuples that are related to a specific tuple, where the corresonding English expression would use the genitive case, either *the planet's moons,* or *the moons of the planet.* In COOL the genitive relation is specified as

*<relation-name>.<identifier-list>*<relation-name>*

or *<range-variable>.<identifier-list>*<relation-name>*

or with a user-defined convenient alias. An alias similar to the English language genitive case construct that uses the common apostrophe s construction, as in *Planet's Moons,* is probably most convenient.

In the implementation reported here both the formal genitive relation specification and the *apostrophe s* construct are permitted. Other aliases, could easily be implemented as well, but were not. In general an alias for a formal genitive relation name is best specified as part of the conceptual database definition, for example:

*Planet.moonlist*Moon alias Planet's Moons*
*alias Moons of the Planet*

If no formal genitive relation name can be specified because no reference list attribute like moon list has been defined in the database definition, in general a genitive relation alias can still be defined using an SQL expression as follows:

*Instance Planet P;*

*Create genitive relation (Select M.* from Moon M*
*where M.pname = P.pname)* *alias* *Planet's Moons*

Since a genitive relation is a relation, it can serve as a range variable in COOL, in the same manner as relation names serve as range variables in SQL. The names of relations can also serve as range varibles in COOL. It is worth pointing out that the path expression concept used with relationships with the OO query language OQL in the ODMG-93 standard is similar to, but not the same as, the genitive relation concept [Atw93, Atw94, Cat93, Kim94].

When dealing with 1:n relationships there are two basic kinds of genitive case, that concerning children related to a parent, described above, and that concerning the parent of a child. For example, if we have entity *ship* and entity *passenger*, a *ship's passengers* or *passengers of a ship* is the first kind, and a *passenger's ship* or the *ship of a passenger* is the second. As an example of the second with COOL, consider the retrieval:

*Get full details of each moon where the diameter is larger than 2000 miles and the moon's planet has a diameter less than 5000 miles.*

Select * from Moon where (Moon.diameter > 2000 and
for the Moon's Planet (Planet.diam < 5000))

The quantifier is *for the [one]* and the genitive relation alias is *Moon's Planet,* the formal genitive relation name being *Moon.pname*Planet,* which could have been used instead. The syntax for the formal genitive relation name employs the foreign key *pname* instead of a list attribute (whose value is a list of child tuple identifiers), since there can be only one parent for a given child.

*Many-to-many relationships* are handled similarly. Suppose we extend the database above to include the entities space probe and planetary visit. This gives rise to a many-to-many relationship between probe and planet since a probe can visit many planets and a planet can be visited by many probes. Intersection data for the relationship is in the relation Visit. As usual the many-to-many relationship can be treated as a pair of 1:n relationships, between Planet and Visit and between Probe and Visit.

*Planet (pname, diam, volume(), moonlist, visitlist, probelist)*
*Ringplanet (include superentity Planet attributes, nrings, ringlist) isa Planet*
*Ring( r#, pname, radius, width, color)*
*Moon(mname, pname, diam, dist, craterlist)*
*Lunarcrater(cname, mname, diam, rim, area())*

*Probe (p#, pname, year, cost, visitlist, planetlist)*
*Visit (pname, p#, type, year)*

Consider:

*Get full details of each planet where the diameter is larger than 3000 miles and a majority of the visits prior to 1995 were by probes costing more than $1,000 million.*

Using informal genitive relation names:

```
Select * from Planet where (diam > 3000  and
        for majority of Planet's Visits (year < 1995
                and for the Visit's Probe (cost > 1000)))
```

Planet's Visits can be defined as an alias for the formal genitive relation name Planet.visitlist*Visit.

If the intersection data (eg year < 1995) is not relevant, then we can use the genitive relation existing directly between planet and probe as in:

```
Select * from Planet where (diam > 3000  and
        for majority of Planet's  Probes (cost > 1000))
```

whose semantics should be obvious. However, so far only genitive relations for 1:n relationships have ben implemented. Planet's Probes could be defined in the database definition as an alias for the formal genitive relation name Planet.probelist*Probe. Alternatively it could be defined in the database definition using an SQL expression:

*Instance Planet P;*
*Create genitive relation (Select Pb.* from Probe Pb*
*where Pb.p# in (select V.p# from Visit V*
*where V.pname = P.pname))*
*alias Planet's Probes*

[It is worth pointing out here that analysis shows that it is is possible to use SQL to define spurious relations that are not genitive relations, so that an integrity constraint has to be placed on the the SQL expression within a create genitive relation statement. The essence of this constraint is that if we wish to define a genitive relation Y's Xs, then the SQL expression must retrieve X tuples and it must be possible to relate a single Y tuple to each of the X tuples retrieved through a sequence of joins.]

### 1.4 Comparisons and genitive relations

COOL permits us to compare a genitive relation with another relation. This is particularly useful with certain retrievals involving many-to-many relationships. For example, in the many-to-many relationship case of parts and suppliers[Dat95], retrievals involving extracting *each supplier that supplies all parts in the database [or vice versa]* require the condition that a suppliers parts (the genitive relation) be equal to all the parts in the database (another relation).

To illustrate using the database above, consider:

*Get the planets in excess of 10,000 miles diameter that have been visited by all of probes (listed in the data base)*

```
Select * from Planet P
 where P.diam > 10000 and
  P's Probes  = Probe
```

A COOL rule is that only genitive relations can be quantified, that is, we can use logical quantification only where we have a condition q GR (<condition>). However arithmetic

quantification of any relation, where only a count of a quantity of members of a set of tuples is involved, as in q R, is allowed. In a relation comparison, *all of* is the default quantifier. Thus changing the last line above to *P's Probes = all of Probe* would make no difference, but if we change the above query to

*Get the planets in excess of 10,000 miles diameter that have been visited by a majority of probes (listed in the data base)*

```
Select * from Planet P
        where P.diam > 10000 and
         P's Probes  = majority of Probe
```

Here the quantifier *for majority of [all]* simply specifies an arithmetic quantity of the tuples in Probe. The expression *majority of Probe* is not a logical condition. Readers who are curious about the expressive power of COOL constructs should try to write the above expression in SQL.

### 1.5 Composite genitive relations

Just as we can have composite genitive cases in natural language we can have composite genitive relations in COOL. Suppose we introduce a final entity type Solsystem or solar system in the database above

*Solsystem (s#, dist, planetlist)*

(and an imagined time when we know about other solar systems), with an additional s# foreign key attribute in Planet:

*Planet (pname, s#, diam, volume(), moonlist, visitlist, probelist)*

If a solar system can have many planets, each of which can have many moons, then both a *solar system's moons* and a *moon's solar system* are examples of the *composite genitive case,* for the composite 1:n relationship between Solsystem and Moon. Similarly we can construct *composite genitive relations*. Consider the retrieval and COOL equivalent:

*Get details of each solar system within 100 light years in which one and all of the moons exceed 100 miles in diameter*:

```
Get * from Solsystem where (dist < 100
        and for one and all Solsystem's Moons (diam > 100))
```

The composite genitive relation used above is *Solsystem's Moons,* and the formal composite genitive relation corresponding to it is

*Solsystem.planetlist*Planet.moonlist*Moon*

which, for a specific Solsystem tuple, denotes a join of planetlist attribute with the relation Planet to give the set of planets belonging to that solar system. with a further join of the moonlist attribute in each of those planet tuples with the relation Moon to give the set of moons in that specific solar system. The genitive relation can also be specifed in SQL with

*Instance Solsystem S;*

*Create genitive relation (Select M.* from Moon M*
*where M.pname in (select P.pname from Planet P*
*where P.p# = S.s#))*
*alias Solsystem's Moons*

If entity instance A is 1:n related to many B instances, which in turn are 1:n related to many C instances, the composite genitive relation name for the genitive relation between A and C has the formal syntax structure:

*A.Blist*B.Clist*C,* or, at a minimum

*Blist*Clist*C*

Alternatively, we can define the obvious alias *A's Cs.*

Of course it is often possible to avoid a composite genitive relation or genitive case by using the component non-composite ones, for example by using both *Solsystems's Planets* and *Planet's Moons* instead of just *Solsystem's Moons.* Using these and taking the 1:n relations one at a time in proper hierarchical sequence the COOL expression above can be rewritten as:

```
Select * from Solsystem where (dist < 100
        and for one and all Solsystems's Planets (
            for one and all Planet's Moons (diam > 100)))
```

This would seem to indicate that composite genitive relations (and cases) are unnecessary since we apparently can always decompose into the constituent non composite genitive relations (and cases). Although this is so sometimes, it is not true in general, because the use of two non-composite genitive relations instead of a single composite one requires the use of two quanitifiers instead of one, and with some quantifiers, we cannot know which two to use. For example, when we are referring to all the moons of a solar system, we are clearly referring to all of the planets of the solar system and all of the moons belonging to each of those planets. But if we are referring to the majority of the moons of a solar system, we are not necessarily referring to the majority of the planets and the majority of the moons belonging to each planet of the majority. A minority of the planets in a solar system might hold the majority of the moons, which is the case in our Solsystem. Thus in some circumstances we simply cannot do without the composite genitive relation (and case, which is why we must have it in English). Hence, with the expression:

```
Get * from Solsystem where (dist < 100
            and for majority of Solsystem's Moons (diam > 100))
```

with composite genitive relation *Solsystem's Moons,* we cannot replace it with an expression of the form

```
Get * from Solsystem where (dist < 100
  and q1 Solsystem's Planets (
            q2 Planet's Moons (diam > 100)))
```

that used non-composite genitive relations q1 and q2, since we do not know what quantifiers to use for q1 and q2. This problem is also there when SQL is used. It gives rise

to complex and error-prone SQL retrieval expressions (the reader might try the above retrieval in SQL).

### *1.6 Subgrouped genitive relations.*

In natural language the use of quantifiers is not restricted to quantification of the set of related objects. Consider, for example, the expression: *those planets for which all of the moons exceed 500 miles in diameter*. Here we are specifying, for a given planet, a quantity (all) of related moons that obey the condition that the diameter exceed 500 miles. Now consider the query:

*Get the name of each planet exceeding 10,000 miles in diameter for which all of the moons more distant than 100,000 miles exceed 100 miles in diameter.*

This is an example of quantification of a related subgroup. For a given planet we are specifying the quantity of moons over 100,000 miles distant that exceed 100 miles in diameter. We are not quantifying the set of child moon entity instances related to a given parent planet instance, but a subset of the set of related moons, those over 100,000 miles distant, related to a given parent planet. Where such subset quantification is involved, even experienced SQL users, such as graduate students, nearly always go wrong; in SQL they correctly use the negated existential quantifier but frequently, with mistaken sophistication, also use negation of all child relation conditions using De Morgan's rules, and construct:

```
Select pname from Planet P
where diam > 10000
and not exists (select * from Moon M
                    where P.pname = M.pname and
                  M.dist !> 100000 or M.diam !> 100)
```

The last line is subtly wrong. The condition should be (M.dist > 100000 and M.diam !> 100) to allow for quantification of only a subset of the moons of any planet. SQL has nothing to cause the user to watch out for this trap. COOL draws the user's attention to the existence of subset quantification with the existence of subgrouped genitive relations. The above retrieval is expressed:

```
Select pname form Planet where diam  >10000 and
                    for all Planet's (dist > 100000) Moons ( diam > 100)
```

Here we use a subgrouped genitive relation alias

*Planet's (dist > 100000) Moons*

The formal genitive relation expression here would be

*Planet.moonlist*(Moon(dist > 100000))*

which is implying that the subgrouped genitive relation is derived from the join:

*moonlist*(Select * from Moon where dist > 100000)*

The entire range of natural quantifiers can be used with subgrouped genitive relations.

Composite subgrouped genitive relations are also possible in COOL, as they are in natural language, for example as in:

```
Select  * from Solsystem where (dist < 100) and
for a majority of Solsytem's (dist> 100000) Moons (diam > 100)
```

### 1. 7 Composite entities

COOL allows for retrieval of composite entities. An example of an instance of a composite entity would be a single ringplanet with its set of rings, and with its set of moons, each moon with its set of craters. A different but similar composite entity instance would be a ringplanet with its set of rings and its set of moons (but no craters). In both cases a planet entity is the root entity of the composite entity, the composite entity having a tree structure. In order to retrieve a composite entity in general we need to specify four aspects of the structure to be retrieved:

(a) Specify the retrieval condition for the root. This we can do with the kind of COOL expression facilities discussed above.

(b) Specify the subentities of the root that are required to construct the composite entity. This requires an additional syntax.

(c) Specify a named view in which the retrieved set of composite entities can be stored (virtually) ; the view should be derivable from the relations of the database.

(d) If the retrieval is embedded in a programming language, a set of suitably structured programming language structure variables or class instance variables must be specified to receive the set of composite structures retrieved. The author believes it is best if the name of each type of relation making up the composite structure is also retrieved and placed before each set of tuples of that type, as well as the number of tuples of each type as they occur.

As example suppose we have a relation Rname that is the parent of both relations Aname and Bname, and that each composite entity instance will consist of a root Rname tuple such as R2 and a set of Aname tuples such as A2, A7, A9 and a set of Bname tuples such as B1 and B6. Then we would retrieve a set of structures like the following

*Rname R2 Aname 3 A2 A7 A9 Bname 2 B1 B6*

*Rname R6 Aname 2 A1 A3 Bname 3 B3 B4 B8 etc*

and there should be a programming language structure variable or class instance variable (object) prepared to accept them.

All of the above requirements are are taken into account in the syntax of a COOL expression for retrieval of a composite entity. They are exemplified by the structure of the COOL expression for the following composite enity retrieval :

*Get each ringplanet with its rings and its moons and their craters, for each ringplanet with a diameter greater than 60,000 miles with at least 3 moons more distant than 300,000 miles from the planet's surface*

```
[Create view V1 as]
 select composite [into S]
        R.* from Ringplanet R
                                  where R.diam > 60000
                                  and for >=3 R's Moons M (M.dist > 300000),
        M.* from R's Moons M
                        ( L* from M's Lunarcraters),
        Rg.* from R's Rings Rg.
```

The keyword *composite* following *select* alerts the database system that this is not a normal query that retrieves relations but one that will retrieve hierarchically structured composites made up of segments, possibly repeating, that are stored as the tuples of relations, and that the originating relation name and the number of tuples retrieved is to be placed before each occurrence of such segments. The above expression thus would retrieve a set of composite structures of the kind:

*Ringplanet R3*
*Moon 1 M2*
*Lunarcrater 2 L2 L4*
*Moon 1 M4*
*Lunarcrater 3 L6 L7 L9*
*Moon 2 M7 M9*
*Ring 3 Rg1 Rg4 Rg6*

In addition this retrieved set of composites would be stored virtually as a view V1 (if required) and transferred to program structure variable or class variable S (if required).

Once such a view as V1 exists it can be used to (virtually) store new composite entities of that type in the data base. For example, if such a composite entity instance is constructed in a program structure variable T (complete with relation names and segment counts in the same manner in which such structures are retrieved), the structure can be stored simply by the command

*Insert structure T into view V1.*

Because the relation names and quantities of the tuples making up the structure must be specified within the structure, the database system has the information needed to decompose the structure correctly and place the component tuples in the correct relations. Note that the structure submitted in T will be rejected by the database system if it does not obey the conditions required for selecting the structures in V1, as given in the Create expression above. If accepted, the newly stored structure will be recorded (virtually stored) as a member of V1 and so can be retrieved easily from V1 into any further program structure W. For example, if the structure inserted from T into V1 is for the planet Saturn, it can be retrieved again into another program variable W by the simple command.

Select into W from view V1
where Planet. pname = "Saturn"

In this way COOL and retrieve and store composite entities that can decompose into relations.

Note that this composite entity is facility in ComposeR is somewhat similar to those of early hierarchical data base systems, for example IBM's IMS [EN95], except that COOL (and SQL) are much more powerful retrieval languages for specifying which roots are required. However, details of the programming language/ComposeR interface are not the focus of the present paper; the above discussion is merely to demonstate that ComposeR is designed with comprehensive programming language interface facilities.

**Graphical Retrieval Language (GRL)**

A graphical version of COOL, called Graphical Query Language has been designed for ComposeR, although the extensive detail are beyond the scope of this paper. We merely point out here that COOL lends itself well to graphical retrieval structures, along the lines of Query-by-Example [EN95], the major enhancement being the addition of a quantifier box to specify the quantity of related tuples that must satisfy a condition. The quantifier box enables selection of some 50 different types of cross-referenced retrievals at the click of a mouse, something that appears to be impossible with graphical versions of SQL.

## 2. Other Facilities

The other basic commands are Create for database definition and Insert, Delete and Update commands. These commands are fairly conventional. In addition recursive relationships are allowed for.

### 2. 1 Recursive 1:n relationships

Recursive 1:n and n:m relationships can also be defined by the Create command and manipulated by COOL. In the case of a recursive 1:n relationship involving the entity Employee, for instance, in the Create command we can define a primary key *ekey*, and a foreign key *chief*, an employee's boss employee number. This is sufficient to define the relationship, and have it supported by system generation of a list of related child Employee identifiers, for example:

*Employee (ekey, ename, esalary, eyear, chief, subordinatelist)*

Here *chief* is a superkey or foreign key giving the primary key of the parent and *subordinatelist* is a reference list that lists the identifiers of children, that is, immediate subordinates, of an Employee instance.

We need to take care with genitive relation names when dealing with recursive relationships, particularly child-to-parent genitive relationships. In the case of the Employee entity, Employee is in a 1:n relationship with itself. The question then arises: Is the genitive relation *Employee's Employee* a parent to child or a child to parent genitive relation? We have an ambiguity that is resolved by using a full genitive relation name, as allowed for in COOL's genitive relation syntax, or by an unambiguous alias. If the Employee relation is defined as shown above, then the full genitive relation names are:

***A. Parent to child***

| *Informal Alias* | *Formal* |
|---|---|
| Employee's subordinate Employees | Employee.subordinatelist*employee |

***B. Child-to-parent***

| | |
|---|---|
| Employee's chief Employee | Employee.chief*Employee |
| Employee's Chief | |

Expressions with genitive relations of this nature can be handled by the implemented version of COOL. Recursive many-to-many are handled similarly in the implemented version of COOL but space does not permit a discussion[see Ra95].

## 5. Future Work

The most important avenue of research opened by this work involves the development of a second version of ComposeR in which the query language is not COOL and SQL separately SQL but an extended SQL in which the features of COOL have been seamlessly merged. In such a language conventional SQL and COOL constructs can be mixed as convenience warrents. The project has already begun and will be reported on elsewhere[]. A major part of such an endeavour will be the development of a fully optimized query processor. A prototype implementation of COOL has been carried out and is desvcribed in [Ra95].

## SUMMARY

In this paper we have described an experimental database system called ComposeR with an extended relational data model and two declarative languages, SQL and a new language called COOL aimed at queries involving containment relationships. The primary motivation for the design of COOL was avoidance of the universal quantifier difficulties of conventional SQL with containment relationships. ComposeR system supports inheritance and composite entity retrieval.

Fundamentally COOL has a predicate calculus expression structure, and allows the use of genitive relations that model containment relationships as well as natural quantifiers A genitive relation is also a relation equivalent of the genitive case in natural language. It is used to specify containment related entity instances, most commonly in 1:n relationships, as in Planet's Moons, for example, where Planet and Moon are relations that repesent a set of planet instances and moon instances respectively. As with the genitive case we can have genitive relations coresponding to parent-to-child and child-to-parent relationships, both composite and non composite. We can also have genitive relations coresponding to recursive relationships. The natural quantifiers are the quantifiers of natural language. SQL allows only the existential quantifier *exists*, although IBM's Starburst allows for the quantifier *for majority* in its extended SQL [LLPS91].

To translate COOL query expressions an Extended Relational Algebra (ERA) was employed, with a COOL expression being translated to an ERA routine. Extended relational algebra is the same as conventional relational algebra except for three additional operations, namely the group-select, the subgroup-select and the possibility join operations.

COOL is not designed or intended as a replacement for SQL. The ultimate goal is incorporation of the features of COOL into SQL in a seamless manner in order to improve user productivity and reduce errors.

## REFERENCES


1. ABD+89 M. Atkinson, F. Bancilhon, D. DeWitt, K.Dittrich, D. Maier and S. Zdonik. The object-oriented database system manifesto. In Proc. of the 1st Int'l Conf. on Deductive and Object-oriented Databases, pp 40-57, 1989.
2. Atw93. T. Atwood. OMDG-93: The object DBMS standard. Object Magasine 3(3), pp 37-44, 1993.
3. Atw94. T. Atwood OMDG-93: The object DBMS standard, part 2. Object Magasine 3(5), p32., 1994.
4. Ban93. F. Bancilhon. Object database systems: Functional architecture. In Object Technologies for Advanced Software. First JSSST Int'l Symp. Proc., pp 163-75, 1993.
5. BBB+88 F. Bancilhon et al. The design and implementation of O2, an object-oriented database system. In Advances in Object-oriented Database Systems. In 2nd Int'l Workshop on Object-oriented Database Systems. Proc. LNCS 334, pp 1-22, Springer-Verlag, 1988.
6. Bra88 J. Bradley. A group-select operation for relational algebra and implications for database machines. IEEE Trans. on Software Systems, 14(1), pp 126-29, 1988.
7. Bra92 J. Bradley. A genitive relational tuple calculus for an object-oriented relational data model, Tech. Report, Dept. of Comp. Sci., Univ. of Calgary, No. 92/488/26, 1992.
8. Bra96. J. Bradley. Extended relational algebra for reduction of natural quantifier COOL expressions, J. of Systems and Software, 33(1), pp 87-100, 1996.
9. BOS91 P. Butterworth, A. Otis, and J. Stein. The Gemstone database management system. Comm. of the ACM, 34(10), pp 64-67, Oct. 1991.
10. Bro91 A.W. Brown. Object-oriented Databases and their Application in Software Engineering. McGraw-Hill, 1991.
11. Cat91 R.G.G. Catell. Object Data Management: Object-oriented and Extended Relational Database Systems. Addison-Wesley, 1991.
12. Cat93 R.G.G. Catell, editor. The Object Database Standard: ODMG:93.
Morgan Kaufman, 1993.
13. Che76 P.P. Chen. The entity-relational model - towards a unified view of data. ACM Trans. on Database Systems 1(1) pp 9-36, 1976.
14. Cod81 E.F. Codd. Data models in data base management. ACM SIGMOD record 11(2), 1981.
15. Dat95 C.J. Date. An Introduction to Database Systems. Addison Wesley, 1995.
16. Deu91 O. Deux et al. The O2 system. Com. of the ACM, 34(10), pp 35-48, 1991.
17. EN95 R. Elmasri and S. B. Navathe. Fundamentals of Database Systems, 2nd Ed., Benjamin/Cummings, Menlo Park, Cal., 1995
18. FBC+87. D.H. Fishman et al. Iris: and object-oriented data base management system. ACM TOIS 5(1) pp48-69, 1987.
19. HM81 M. Hammer and McLeod. Database description with SDM: A semantic database model. ACM Trans on Database Systems, 6(3) 351-86, 1981.
20. HR87 R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. ACM Computing Surveys, 19(3) pp 140-73, 1987
21. Kim90 W. Kim. Introduction to Object-oriented Database Systems, MIT Press, 1990.
22. Kim92 W. Kim. On unifying relational and object-oriented database systems. In ECOOP '92. European Conf. on Object-oriented Programming. Proc., pp 1-18, 1992.

23. Kim94 W. Kim Observations on the ODMG-93 proposal for an object-oriented database language. SIGMOD Record, 23(1), pp 4-9, 1994.
24. KL89b W. Kim and F.H. Lochovsky. Features of the Orion object-oriented database system. In Object-oriented Concepts, Databases and Applications, Chapter 11, pp 251-282, Addison-Wesley, 1989.
25. KL89c W. Kim and F.H. Lochovsky (eds). The Gemstone Data Management System. In Object-oriented Concepts, Databases and Applications, Chapter 12, pp 283-208, Addison-Wesley, 1989.
26. LH90 B. Linsay and L. Haas. Extensibility in the Starburst experimental database System. In "Database Systems of the 90s." Int'l Symp Proc. pp217-248, 1990.
27. LLPS91 G. M. Lohman et al. Extensions to Starburst: objects, types, functions and rules. Com. of the ACM, 34(10), pp 95-109, 1991
28. Mai83 D Maier "The theory of relational databases", Computer Science Press, 1983
29. Mel94 J. Melton ed. Database language SQL3, ISO/ANSI Working Draft, 1994.
30. Ra95 C. D. Rata. A prototype front-end for a declarative object-relational database language employing natural quantifiers and genitive relations, Thesis, Department of Computer Science, University of Calgary, 1995.
31. SHI81 D. Shipman. The functional data model and the data language Daplex. ACM Trans. on Database Systems, 6(1) 140-73, 1981.
32. SK91 M. Stonebraker and G. Kemnitz. The Postgres next-generation database management system, Com. of the ACM, 34(10), pp 79-92, 1991.
33. SR86 M. Stonebraker and L. Rowe. The design of Postgres. In Proc. of the ACM SIGMOD Conf., pp 340-55, 1986.
34. SRL+90 M. Stonebraker et al. Third generation database system manifesto. ACM SIGMOD Record. 19(3) pp 31-44, 1990.
35. Sto87 M. Stonebraker. The design of the Postgres storage system. In Proc. of 13th Int'l Conf. on Very Large Database Systems, pp 289-300, 1987.
36. Ull88 J. D. Ullman. Principles of Database and Knowledge-Base Systems, Vol 1, Computer Science Press, 1988.
37. US90 R. Unland and G. Schlageter. Object-oriented database systems, Concepts and perspectives. In Database systems of the 90s. Int'l Symp. Proc. pp 154-97, 1990.
38. ZM89 S.B. Zdonik and D. Maier. Readings in Object-oriented Database Systems, Morgan-Kaufman, 1989.
□