**An extended SQL and QBE with seamless incorporation of genitive relation and natural quantifier constructs**

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

***ABSTRACT*** An extended SQL, called E-SQL, is described, for use with an extended relational data model. The data model allows for restricted unnormalized relations; sets or lists of atomic attributes are allowed but not attributes that are relations, and there is support for both inheritance and composite entities.

E-SQL permits the constructs of conventional SQL, but also permits use of genitive relations to model containment relationships and facilitate application of natural quantifiers to containment sets. A genitive relation is a relation that is the equivalent to the genitive case grammatical construct in natural language. The genitive relation construct and the conventional constructs of SQL can be mixed in a seamless manner. For example, the condition within a conventional SQL IN-construct could contain a condition that is a quantified genitive relational condition, and vice versa. Use of conditions that involve comparison of a quantified genitive relation with a quantified base table is also permitted. E-SQL can also manipulate a set of composite entity instances modeled as a group of relations. E-SQL is essentially a seamless merger of conventional SQL and an experimental genitive relation language, called COOL, designed for efficient manipulation of containment sets. It is shown that an Extended Relational Algebra (ERA) can be used to reduce E-SQL expressions. E-SQL expressions are much more concise and much less errror-prone than conventional SQL expressions. It is also shown how the quantified genitive relation concept can be used in an extension of the graphical query language QBE that greatly increases both its retrieval power and simplicity of use.

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

**Presently at: Bell-Northern Research*
*Ottawa, Ontario, Canada*

## INTRODUCTION

This paper largely concerns the design and analysis of an experimental extended SQL (called E-SQL) for use with an extended relational data model. To create E-SQL, conventional SQL was extended by seamless incorporation of the main constructs of an experimental extended relational containment oriented language called COOL[Bra96]. COOL was designed mainly for efficient programming of both retrieval expressions involving containment relationships between entities and retrieval expressions involving composite entities. The core of COOL has been implemented, so that there can be confidence in the viability of its constructs. COOL, and E-SQL, were designed for use with the same extended relation data model. The final goal is a database system called ComposeR, that embodies an extended relational data model and database manipulation by both E-SQL and COOL. COOL was implemented merely to test out its constructs, and not with the idea that it could ever replace SQL, whose use is too well entrenchd for that to make sense. Rather the idea is to improve SQL in those areas where it is weak, and to do so by adding constructs that have proven their worth elsewhere.

E-SQL has all the usual features of conventional SQL, but also has non-SQL features for dealing with containment structures, composite entity instances and inheritance supertype-subtype hierarchies. E-SQL 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, supertype and subtype relations, and user-written functions.

Extended relational database systems [SRL+90, Kim92, Kim95] 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.

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). 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 [KL89b, BOS91], O2 [BBB+88, Deu91] and Orion [KL89a, 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. The development of E-SQL is consistent with achieving these goals.

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 being the Entity Relationship model [Che76] and enhanced versions. 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 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, Kim95]. Starburst supports a rich type system, enhanced performance features, encapsulation of behaviour with data, identifiers for stored entites, large structured complex entities, 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. 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 E-SQL.

## 1.0 Motivation

The fundamental motivation is the desire to eliminate from SQL, by means of the proposed extended SQL or E-SQL, 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.

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. 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* as 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 Planet P, Moon M, Lunarcrater 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 < 10000*
*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 casual 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 misinterprete 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 blue and within 100,000 miles.* This will be discussed in Section 4.3.

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 and precise meaning, which is lacking in the mathematical universal quantifier.

The solution to the problem is to modify predicate calculcus to allow for containment relations. Briefly the solution involves allowing for containment relations defined in terms of the containment entities. Although in ComposeR, and with E-SQL, we refer to genitive relations instead of containment relations, for predicate calculus purposes we might define a containment tuple set, or relation, 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. 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[Cat93].

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)}*

which would correctly retrieve only Mars. 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

*{ 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 type mental gymnastics, 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 this paper we discuss incorporation of genitive relational and natural quantifier constructs into SQL in a seamless manner, to produce a language called E-SQL derived from the modified predicate calculus above. These additional constructs have already been concentrated in a laboratory language called COOL. COOL is an SQL-like sublanguage, but without conventional SQL constructs, that is derived from the modified predicate calculus outlined above, instead of conventional predicate calculus from which SQL is

derived. COOL has already been implemented as an extended relational database declarative language. For details of COOL semantics and sysntax see [Bra96] and for implementation details see [Ra95].

Before presenting E-SQL, first we look at a reasonable extended relational data model.

## 2.0 Summary of the ComposeR data model

E-SQL can be used to manipulate both data representations of simple entities and composite entities. In the extended relational data model of the prototype system ComposeR, in which both E-SQL and COOL are declarative languages, 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 may be 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 genitive relations, 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 E-SQL expressions (for example the function *overlap*() might be used to test for overlap of one crater with another in a retrieval condition).

### 3.0 Overview of E-SQL and genitive relations

As an example of a ComposeR database, consider a database for simple entities Solsystem, Planet, Moon, and Lunarcrater where a solar system can have many planets, a planet can have many moons, and a moon many lunar cratera (and an imagined time when we know about other solar systems). 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).

*Solsystem (s#, dist, planetlist)*
*Planet (pname, s#, 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())*

The attributes s#, *pname, r#, mname* and *cname* are 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.

To understand how natural quantification, along the lines of COOL, can be seamlessly integrated into SQL, it is necessary to analyse first how SQL derives from predicate calculus. Consider the retrieval:

*Get name and diameter of each planet over 6,000 miles in diameter that has a moon less than 100,000 miles distant.*

The predicate calculus expression is:

*[P.pname, P.diam: P.diam > 6000 and*
*exists M(M.pname = P.pname and M.dist > 100000)]* *(1)*

M ranges over all of Moon so that the entire set of Moon tuples is quantified. An alternative expression is:

*[P.pname, P.diam: P.diam > 6000 and*
*P.pname in [M.pname: M.dist > 100000]]* *(2)*

This formulation employs set membership, denoted by in, of a nested set of Moon tuples. The first expression (1) can be rearranged to employ a nested set of Moon tuples:

*[P.pname, P.diam: P.diam > 6000 and*
*exists [M : M.pname = P.pname and M.dist > 100000]]* *(3)*

But notice that the nested sets of Moon tuples in (2) and (3) are different. In (2) the nested set of Moon tuples represents that set of moons, regardless of parent planet, each of which is > 100,000 miles from its planet. In (3) it represents the set of moons of a specific planet where each moon is more than 100,000 miles from its planet.
If we agree to denote specification of a set of tuples derived from a relation, nested or not, by the syntax:

*(Select <attributes> from <relation tuples> where <condition>)*

we can translate the above two predicate calculus formulations to:

```
Select P.pname, P.diam from Planet.P
where P.diam > 6000 and P.pname in (select M.pname Moon M
                    where M.dist > 100000)                    (4)
```

and

```
   Select P.pname, P.diam from Planet.P
where P.diam > 6000 and exists (select M.* from  Moon M
                    where M.pname = P.pname and
                    M.dist > 100000)                          (5)
```

which gives us the well-known IN-construct and EXISTS-construct in SQL for dealing with existential quantification. And, as in in the case of corresponding predicate calculus expressions (2) and (3), the nested sets of Moon tuples in SQL expressions (4) and (5) are different, a peculiarity of SQL that can cause confusion among novice SQL users. Now consider the same retrieval except that it involves the universal quantifier:

*Get name and diameter of each planet less than 10,0000 miles in diameter where all the planet's moons are less than 1000 miles distant from the planet's surface*

The conventional predicate calculus expression, as we saw in the Motivation section earlier, requires that all M tuples be quantified and that this involve a Horn clause.

Since any attempt to write a faithfully corresponding SQL construct would necessarily involve a Horn clause, as for example in:

```
Select P.pname, P.diam from Planet.P
where P.diam  < 10000 and
   for all Moon M (M.pname != P.pname or M.dist  < 1000)      (6)
```

an SQL implementation that would allow this was never done. The original developers of SQL presumably reasoned that this kind of construct was too error-prone or just to difficult for most users. In this kind of reasoning they were probably correct. But this meant that the only way of specifying the universal quantifier in SQL had to be by means of the negated existential negated condition construct, which is also error-prone, as we have already seen. [The later addition to SQL of the term *all*, which is not the universal quantifier, for use in restricted circumstances, and which we cannot recommend because it is error-prone, and which space does allow us to discuss, has not solved this problem.]

As pointed out earlier, if we introduce a modified predicate calculus that allows for containment sets, then using the containment set [P::MOON], the expression for the above retrieval can be written:

```
[P.pname, P.diam: P.diam < 10000 and
      for all [P::Moon] M (M.dist  1000)]                    (7)
```

where M ranges over the containment set [P::Moon]. If we include the expression for [P::Moon], expression (7) can be rewritten:

```
[P.pname, P.diam: P.diam < 10000 and
      for all [m in Moon: P.pname = m.pname] M (M.dist  1000)]    (8)
```

As pointed out earlier, this expression does not have any Horn clauses and the final condition states that for all of the members of the containment set of moons, that is, moons belonging to a specific planet, the distance is less than 1,000 miles. The author believes this to be a more reasonable way of specifying a universal quantifier condition for average users of a database. It reflects how humans think and how they use language when dealing with containment.

If we now derive an SQL-like expression that faithfully corresponds to this modified predicate calculus expression (8) we get literally:

```
Select P.pname, P.diam from Planet.P
where P.diam  <10000 and
      for all (select m.* from Moon m
       where m.pname = P.pname) M (M.dist < 1000)             (9)
```

The block

```
(select m.* from Moon m
 where m.pname = P.pname)
```

defines a containment set of Moon tuples, or a genitive relation, or more precisely, a genitive Moon relation, namely the Moon tuples representing a specific planet's moons. M is now a range varible ranging over the members of this containment set of Moon tuples, and not over the entire Moon set.

If we predefine the genitive relations in the database definition for the commonly used relationships of the database, and give them useful names or aliases, as for example:

*Range Planet P*
*Create genitive relation as*
*select m.* from Moon m*
*where m.pname = P.pname*
*alias Planet's Moons, P's Moons, planetary_moons*

we are free to use more concise constructs in an SQL expression, such as:

Select P.pname, P.diam from Planet.P
where P.diam < 10000 and
for all P's Moons M (M.dist < 1000)

The above is an extended SQL (E-SQL) expression that permits cross-reference constructs of the kind

*for all P's Moons M (M.dist > 100,000)*

with formal syntax:

*<quantifier><genitive relation>[<range variable>](<condition>)*

in addition to such traditional SQL cross reference constructs as:

*P.Pname in (select M.pmname from Moon M*
*where M.dist > 100000)*

and

*exists (select M.* from Moon M*
*where M.pname = P.pname and*
*M.dist > 100000)*

which are logical expressions with value true or false.

A drawback for this proposal, although merely one of syntactic tradition, is that a cross-reference clause that is a genitive relational clause is not a Select-from-where clause as it is in the case of IN and EXISTS constructs, although a select-from-where block is implicit in the definition of the genitive relation. Thus the inclusion of the genitive relational clauses breaks with the now well-entrenched tradition of using only Select-from-where block syntax for cross references in SQL. But there can be no doubt that allowing for genitive relational clauses in SQL unleashes a great expressive power on the part of the user. This follows because a genitive relational clause can be used not only with the universal quantifier, so avoiding the traditional NOT EXISTS or NOT IN constructs, but with any of the quantifiers of natural language (provided the query processor is designed to handle them).

For example consider the set retrievals:

*Get name and diameter of each planet over 10,0000 miles inn diameter where a majority of (all) the planet's moons less than 1,000 miles distant.*

*Get name and diameter of each planet over 10,0000 miles in diameter where one and all of the planet's moons are less than 1,000 miles distant.*

*Get name and diameter of each planet over 10,0000 miles in diameter where all but 2 of (all) the planet's moons are less than 1,000 miles distant.*

These can all be expressed simply by changing the quantifier for all in the above~ E-SQL expression to *for majority,* or *for one and all* or *for all but 2,* as required, for example:

```
Select P.pname, P.diam from Planet.P
where P.diam > 10000 and
   for majority of P's Moons M (M.dist > 1000)
```

### 3.1 Seamless integration of genitive relational cross references

An obvious requirement must be that the genitive relational facilities be incorporated seamlessly into SQL. It is easy to show that this hinges on permission to use a genitive relational conditional clause of the kind:

*<quantifier> <genitive relation> [<range variable>]*
*(<E-SQL condition>)*

as an E-SQL condition. Such permission will ensure such seamless integration.

We can demonstrate this with some examples in which SQL IN and EXISTS contructs are seamlessly mixed with genitive relational constructs.

*1. Get full details of each solar system within 100 light years in which all the planets are larger than 8,000 miles in diameter and have at least one moon closer than 10,000 miles to its planet.*

```
Select S.* from Solsystem S
where S.dist < 100 and
   for all Solsystem's Planets P (P.diam > 8000 and
         P.pname in (select M.pname from Moon M
               where M.dist < 100000)
```

*3. Get full details of each solar system within 100 light years in which at least one of the planets is larger than 8,000 miles in diameter and has all its moons closer than 100,000 miles to its planet.*

```
Select S.* from Solsystem S
where S.dist < 100 and
```

```
S.s# in (Select P.s# from Planet P (P.diam > 8000 and
        for one and all Planet's Moons M (M.dist < 100000)
```

*3. Get full details of each solar system within 100 light years in which at least one of the planets is larger than 8,000 miles in diameter and has a majority of its moons closer than 100,000 miles to its planet.*

```
Select S.* from Solsystem S
where S.dist < 100 and
  S.s# in (Select P.s# from Planet P (P.diam > 8000 and
      for majority of Planet's Moons M (M.dist < 100000)
```

In each case we haved seamlessly mixed two distint type of condition:
(a) the conventional SQL in-construct condition
*x in (select R.x from <relation R> where <condition>*
(b) a quantifed genitive relation condition
*q A's Bs (<condition>)*

The <condition> in both cases can be a compound condition. The basic rules for expansion of SQL to E-SQL are therefore:

(1) An *exists (select .. where-condition)* construct or *x in (select R.x ... where-condition)* construct can be used in E-SQL exactly as in conventional SQL.
(2) Where the *exists (select ... where-condition)* or *x in (select R.x ... where-condition)* construct can be used in conventional SQL, in E-SQL a quantified genitive relation condition construct can be used instead.
(3) Where *(select ... where-condition)* is used in E-SQL, a quantified genitive relation condition construct can be (or be part of) the where-condition.
(4) Where the quantified genitive relation condition construct is used in E-SQL, *an exists (select ... where-condition)* or *x in (select R.x ... where-condition)* or *n = (select aggregation-fn () ... where-condition)* construct can be (or be part of) the condition within the quantified genitive relation condition construct.
(5) The whole range of natural quantifers is available for use with the quantified genitive relation construct in E-SQL expressions. All of the natural quantifiers of the English language can in theory be used, if implemented in the query processor. A facility for conveniently defining new natural quantifiers for the query processor to use might be a good idea. The prototype implementation of COOL described in [Ra95] permits 14 quantifiers including the universal and existential quantifiers of conventional predicate calculus.

Where the quantifier involved in a quantified cross-reference condition is *for at least 1*, then for the user there is not much to choose between the quantified genitive relation construct and the conventional SQL x *in (select R.x ... where-condition)* or *exists (select ...)* construct. However where the quantifier is a universal type quantifier, such as *for all, for one and all,* or *for majority of [all]*, then the use of the genitive relation

construct is clearly superior in terms of time consumption to write the expression and probability of freedom from error. Where the quantifier is a non-universal type natural quantifier, for example, *for more than 3* or *for exactly 1,* etc., it is probably easier to use the genitive relation construct, although it is not that difficult to use the alternative SQL

*n = (Select count() from ... where ...)*

construct, which is mathematically equivalent.

### 3.2 Support for inheritance

The ComposeR data model and E-SQL support inheritance, as is illustrated by the retrieval:

*Get the name of each planet of diameter less than 10,000 miles whose moons are all less than 100 miles in diameter, and with at least 1 crater less than 10 miles in diameter.*

The E-SQL expression is:

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

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

### 4.0 E-SQL 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 E-SQL. In an 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, E-SQL 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 E-SQL 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 E-SQL the genitive relation is formally 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. But other aliases would be possible and may be preferred, such as *orbiting Moons*, or *attached Moons,* or even *related Moons.*

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*
*alias attached Moons*

If no formal genitive relation name can be specified because no reference list attribute like moonlist 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 E-SQL, in the same manner as relation names serve as range variables in SQL. The names of relations can also serve as range varibles E-SQL. .

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 E-SQL, 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. However there is probably little advantage to using this construct here as compared with using the conventional SQL IN-construct.

*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.

*Solsystem (s#, dist, planetlist)*
*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, and the 1:n relationship between planet and visit and between probe and visit:

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

*Planetary Visits* can be defined as an alias for the formal genitive relation name *Planet.visitlist*Visit,* and *Visiting Probe* as an alias for the name *Visit.p#*Probe.*

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. 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
```

The use of genitive relations in E_SQL can be looked upon as a way of reusing SQL code. Once the code for a specific genitive relations is defined, users who need to write SQL

expressions that involve the corresponding relationship can merely quote the requisite genitive relation name, and thus save having to write the underlying definition SQL code.

### 4.1 E-SQL and comparisons involving genitive relations and other relations

In a further extension of SQL, E-SQL can permit expressions that 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 supplier's 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 proposed E-SQL rule is that we can use a quantification clause that is a condition only where we have a condition of the kind

*<quantifier><genitive relation> (<condition>).*

However arithmetic quantification of any relation, where only a possible subset defined by 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 a quantity of the tuples in Probe. The expression *majority of Probe* is not a logical condition; it merely specifies a set of Probe tuples of a quantity that is equal to a majority of the complete set of Probe tuples. Readers who are curious about the expressive power of these new E-SQL constructs should try to write the above expression in conventional SQL.

### 4.2 Composite genitive relations

Just as we can have composite genitive cases in natural language we can have composite genitive relations in E-SQL. 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*, or composite containment set, for the

composite 1:n relationship between Solsystem and Moon. Similarly we can construct *composite genitive relations*. Consider the retrieval and E-SQL equivalent:

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

```
Select * 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 an obvious alias such as *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 E-SQL 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 solar system. 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 Ssystem'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 conventional SQL is used. It gives rise to complex and error-prone SQL retrieval expressions (the reader might try the above retrieval in SQL). There is no simple SQL solution to the problem. With E-SQL the solution is to use the composte genitive relation.

### *4.3 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. E-SQL 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)

The requisite syntax rule is that where a condition within parenthesis occurs within a genitive relation alias, as subgrouped genitive relation is specified.

## 5.0 Composite entities

E-SQL 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 E-SQL 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 an E-SQL expression for retrieval of a composite entity. They are exemplified by the structure of the E-SQL 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 E-SQL and retrieve and store composite entities that can decompose into relations. This E-SQL facility was also designed for COOL.

## 6.0 Use of genitive relations and natural quantifiers in graphical database query languages.

The best known graphical database query language is Query-by-Example (QBE), developed at IBM about the same time as SQL was developed [Zlo77, SKS97]. However, although QBE is easily used with simple queries involving one relation, when relationships between relations are involved in the query, then QBE queries can become difficult, arcane or impossible to construct. This is especially true when natural quantification of sets of related tuples is needed. The appeal of QBE is that it is easy to represent an elementary SQL expression of the form

Select <attributes>
from <relation>
where <condition>

as a graphical query, provided the condition does not involve a cross-reference to other relations. Unfortunately, where a cross reference to other relations is involved, there appears to be no simple way of graphically representing the equivalent of SQL complex in-construct and exists-construct conditions, although simple ones can be managed. As a result, despite its obvious potential, QBE is limited.

However, as should be clear from the seamless integration of conventional SQL in-constructs and exists-constructs with naturally quantified genitive relational constructs, as described above, the use of naturally quantified genitive relational constructs does permit simple unambiguous expression of conditions involving cross references to other relations. But what is important from the viewpoint of graphical query language development is that is that graphical representation of naturally quantifed genitive relations is quite easy. Furthermore, since for a specific genitive relation type, the expression of a wide variety of kinds of cross reference conditions is possible simply by use of any of a wide variety of natural quantifiers, incorporation of such a facility into a language like QBE would greatly expand its expression power, and allow expression of a large number of kinds of queries that are simply not currently possible. A simple example should illustrate how this can be accomplished.

Suppose we have the following database about oil companies:

*Company (cname, hq, nemp)* */*company name, headqarters city, no. of employees */*
*Well (cname, w#, depth, res)* */* company name, well no., depth, result */*

Suppose also a non techical executive who wishes to perform the retrieval:

*Get the name of each Denver company the majority of whose wells are dry.*

The conventional SQL expression is likely to be beyond the skill of the executive. The person might be able to manage the E-SQL expression, however:

Select cname from Company
where for a majority of Company's Wells (res = "dry")

If E-SQL cannot be used, the person will not be anble to do the query with expert help. QBE is out of the question, since it could not manage the complex condition following where. But were QBE to be extended as illustrated below, it would be a trivally simple matter to express the query:

| **Company** | ***cname*** | ***hq*** | ***nemp*** |
|---|---|---|---|
| | P. _x | Denver | |

| ***and/or*** | ***quantifier box*** |
|---|---|
| and | for majority of |

| **Well** | ***cname*** | ***w#*** | ***depth*** | ***res*** |
|---|---|---|---|---|
| | _x | | | dry |

The quantifier box is used to specify the quantity of those Well tuples (in this case a *majority of)* whose Cname value matches the cname value in a Company tuple, as specified by the conventional QBE example variable _x, that is, the quantity of those Well tuple in a genitive relation specified using _x. Thus a convenient genitive relation defining facility already exists in QBE. QBE could be extended as shown above so that a quantifier box can appear on clicking an icon, or even automatically when two relations are used in a query. Things could easily be arranged, in current GUI fashion, so that on clicking the quantifier box it would be possible to scroll through the quantifiers available, with a final click to select the one desired.

It should be obvious that this arrangement is both far more simple and far more powerful than anything currently available. And although its simplicity might not appeal to the database systems expert, it should certainly appeal to busy database users whose expertise necessarily lies in other fields.

## 7.0 Extended Relational Algebra

It has been shown both in theory and in implementation[Ra95] that COOL expressions, regardless of complexity, can be reduced to relational algebra routines, whence further reduction is relatively straight forward. However, the relational algebra required contains not only the conventional operations of select, join, project, intersect and union, but three new operations, for handling natural quantifiers and genitive relations in an efficient

manner. These new operations have been described elswhere [Bra88, Bra96] but will be reviewed here in an E-SQL context. They are the *group-select, possibility join* and the *subgroup-select* operations, and together with conventional relational algebra operations form the *Extended Relational Algebra* (ERA) needed for reduction of E-SQL expressions.

### 7.1 Group-select operation

Consider the simple retrieval request and E-SQLexpression

*Retrieve full data about each planet with a diameter <10,000 miles whose moons all exceed 100 miles in diameter.*

The E-SQL expression is:

Select from Planet where diam < 10000
and for all Planet's Moons M (M. diam < 100)

The optimized ERA routine corresponding to this is:

R0 = group_select (Moon(for all pname(diam < 100)))
R1 = select (Planet (diam < 100000))
R2 = R1 (pname) join R0 (pname)

Operations R1 and R2 are conventional relational algebra *select* (or *restrict)* and natural join operations. However operation R0 is new: R0 selects out each group of Moon tuples with the same *pname* (i.e. foreign key ) value provided all of the Moon tuples of the group obey the condition *(diam< 100)*, and from the resulting Moon tuples so selected projects out the *pname* values. R1 then selects Planet tuples with the right diameter and the join in R2 of these tuples to the set of pname values in R0 gives the final answer.

Provided the group-select operation is implemented optimally, the above routine is optimal with selection coming before join. The best conventional relational algebra equivalent is:

*R0 = select (Moon(diam !<100))*
*R1 = project (R0(pname))*
*R2 = select (Planet (diam <100000))*
*R3 = R2(pname) join R1(pname)*
*R4 = R2 -R3*

The general case of the group-select operation is

*Rn = group_select (R(q F (condition)))*

This extracts from relation R each set of R-tuples with the same foreign key F value for which the quantity q of tuples satisfies the condition, and then projects out the foreign key (parent key) F values of the R tuples selected. Since q is any quantifier, the operation can be used with any of the natural quantifiers.

### 7.2 Possibility join

The group-select operation solves the problem of a single level of nesting of quantified cross-reference to a related object type. In the retrieval above the quantified

cross-reference (syntax variable *quantified-xreference)* is *for all Planet's Moons (diam< 100).* In general a quantified_xreference has one of the structure:

*<quantifier> <genitive-relation>(<condition>)*

If the condition is a simple condition it is the structure used above. If the condition is complex the structure can expand to:

*<quantifier> <genitive-relation>(<condition> and/or <quantified-xreferfence> and/or <quantified-xreference> ...*

which allows for unlimited expansion and levels of nesting.

Consider again the retrieval earlier:

*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.*

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

Here the *quantified-xreference* is

*and for one and all Planet's Moons*
*(diam < 100 and for majority of Moon's Lunarcraters (diam < 10))*

and the nested quantified-xreference is

*for majority of Moon's Lunarcraters(diam < 10)*

To handle multiple levels of nesting of quantified-xreferences an additional relational algebra operation called the *possibility join* was developed.

The possibility join operation applied to relations A and B with common join attribute m is written:

*R = A(m) pjoin(p) B(m)*

As a result of this operation, R is assigned every tuple in A concatenated to an additional attribute p; this attribute p is called the posibility join attribute. The operation can be used to reduce the nested quantifier expression above as follows:

*R1 = group_select (Lunarcrater(for majority of mname (diam < 10)))*
*R2 = Moon (mname) pjoin(p) R1 (mname)*
*R3 = select (Planet (diam < 10000))*
*R4 = group_select (R2 (for one and all pname (diam < 100 and p)))*
*R5 = R3(pname) join R4(pname)*
*R6 = project (R5(pname))*

### 7.3 Subgroup-select operation

The subgroup-select operation is used with subgrouped genitive relations. Consider the retrieval:

*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.

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

The genitive relation is now specified by the alias: *Planet's (dist > 100000) Moons.* This genitive relation involves subsets of the moons belonging to a planet, namely those moons over 100,000 miles distant from the planet. This is an example of a subgrouped genitive case or genitive "subcase", as described earlier.

The corresponding ERA routine requires the subgroup-select operation:

*R1 = select (Planet (diam > 10000))*
*R2 = subgroup_select (Moon (for all (pname(dist >100000)) (diam > 100)))*
*R3 = R1(pname) join R2(pname)*
*R4 = project(R3(pname))*

Formally the subgroup-select operation has the syntax:

*R0 = subgroup-select (R (q(F (C1)) (C2)))*

with the semantics that for each set of R tuples with the same (foreign-key) attribute F value and obeying C1, we select the F value of that set if and only if a quantity q of the set satisfies C2.

### 7.5 Reduction of an E-SQL expression with mixed conventional SQL and genitive relational constructs

The above techniques can be used to reduce E-SQL expressions of mixed origin, for example one with an in-construct and a genitive relation construct nested in the in-construct condition, for example:

*Get full details of each solar system within 100 light years in which at least one of the planets is larger than 8,000 miles in diameter and has a majority of its moons closer than 100,000 miles to its planet.*

```
Select S.* from Solsystem S
where S.dist < 100 and
  S.s# in (Select P.s# from Planet P (P.diam > 8000 and
      for majority of Planet's Moons M (M.dist < 100000)
```

There are many ways to approach the reduction of this mixed expression, but an obvious way is to convert the in-construct to its equivalent genitive relation construct and thus make use of the quite tight and concise reductions possible with extended relational algebr operations. Thus we use the fact that the clause:

... S.s# in (Select P.s# from Planet P (P.diam > 8000 and ...)

is equivalent to:

for at least one S's Planets P (P.diam > 8000 and ...)

The relational algebra is therefore:

*R1 = group_select (Moon(for majority of pname (dist > 10000)))*
*R2 = Planet (pname) pjoin(p) R1 (pname)*
*R3 = select (Solsystem(dist < 100))*
*R4 = group_select (R2 (for at least one s# (diam > 8000 and p)))*
*R5 = R3(s#) join R4(s#)*

A reduction anything like as concise as this would not be possible with conventional SQL and relational algebra.

### 7.6 Query processor and optimizer considerations

In general a query processor first generates a standard form relational algebra routine in reducing an SQL expression. Then an enhancer component tranforms and simplifies the standard form routine by removing redundant operations, giving an enhanced or simplified form of the original relational algebra routine. Finally an access routine generator transforms the relational algebra routine to a final optimized low level-routine. For each operation in the simplified relational algebra routine, or for certain groups of operations that go well together, such as select followed by project, the access routine generator selects one of an an array of possible low level strategies for access to the underlying physical files. The low-level routine chosen depends on an analysis of the projected cost of the operation, which in turn depends on the size of the underlying files and the nature of support indexes and pointers available.

It is the enhancer and access routine generator that carry out most of the optimizing. This standard form enhancer component tranforms and simplifies the standard form relational algebra routines from the first stage of reduction by removing redundant operations, thus giving an enhanced form of the original relational algebra routine. The access routine generator component then selects one of an an array of possible low level strategies for access to the underlying physical files.

With the proposed E-SQL query processor, an enhancer is almost unnecessary, since the discipline of converting SQL in-and exists-constructs to equivalent genitive relation constructs in the reduction process, drastically reduces the number of ways in which a given query can be reduced to ERA. As a result it is currently appears very difficult, in most cases, to significantly further enhance the output from the E-SQL standard form generator.

In the case of E-SQL query processing the bulk of the optimizing must rest with the access routine generator (as is the case with the COOL query processing [Ra95]). The critical ERA operations are *join*, which has been widely researched, and the *group-select* operation, which is even more critical as far as E-SQL reduction is concerned, and which

has not yet been much researched. Recall that a group-select operation can involve any of the many natural quantifiers, so that it is far more versatile than any other relational algebra operation. Clearly, for each quantifier there will be an array of possible access routines that can be used, depending on the conditions within the data base. Future work is therefore expected to be concerned with optimizing access routine generation

## SUMMARY

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

Like SQL, E-SQL has a predicate calculus expression structure, and allows the use of the conventional SQL in-constructs and exists-constructs, as well as genitive relations that model containment relationships. E-SQL also permits natural quantifiers with genitive relations. 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 represent 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. SQL allows only the existential quantifier *exists*, although IBM's Starburst allows for the quantifier *for majority* in its extended SQL [LLPS91].

It was also shown how the the quantified genitive relation construct can be used to advantage in a graphical query language. It was shown how it could be used to extend QBE and thus greatly enhance its retrieval power in a simple manner.

How an Extended Relational Algebra (ERA) can be employed for E-SQL translation is also described. 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.

## 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. Ban93. F. Bancilhon. Object database systems: Functional architecture. In Object Technologies for Advanced Software. First JSSST Int'l Symp. Proc., pp 163-75, 1993.

3. 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.

4. 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.

5. Bra96. J. Bradley. Extended relational algebra for reduction of natural quantifier COOL expressions, J. of Systems and Software, 33(1), pp 87-100, 1996.
6. BOS91 P. Butterworth, A. Otis, and J. Stein. The Gemstone database management system. Comm. of the ACM, 34(10), pp 64-67, Oct. 1991.
7. Bro91 A.W. Brown. Object-oriented Databases and their Application in Software Engineering. McGraw-Hill, 1991.
8. Cat91 R.G.G. Catell. Object Data Management: Object-oriented and Extended Relational Database Systems. Addison-Wesley, 1991.
9. Cat93 R.G.G. Catell, editor. The Object Database Standard: ODMG:93.
Morgan Kaufman, 1993.
10. 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.
11. Cod81 E.F. Codd. Data models in data base management. ACM SIGMOD record 11(2), 1981.
12. Dat95 C.J. Date. An Introduction to Database Systems. Addison Wesley, 1995.
13. Deu91 O. Deux et al. The O2 system. Com. of the ACM, 34(10), pp 35-48, 1991.
14. HR87 R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. ACM Computing Surveys, 19(3) pp 140-73, 1987
15. Kim90 W. Kim. Introduction to Object-oriented Database Systems, MIT Press, 1990.
16. 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.
17. Kim94 W. Kim Observations on the ODMG-93 proposal for an object-oriented database language. SIGMOD Record, 23(1), pp 4-9, 1994.
18. Kim95. W. Kim (Editor) "Modern Database Systems". ACM Press/Addison Wesley, 1995.
19. KL89a 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.
20. KL89b 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.
21. 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.
22. LLPS91 G. M. Lohman et al. Extensions to Starburst: objects, types, functions and rules. Com. of the ACM, 34(10), pp 95-109, 1991
23. Mai83 D Maier "The theory of relational databases", Computer Science Press, 1983
24. Mel94 J. Melton ed. Database language SQL3, ISO/ANSI Working Draft, 1994.
25. 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..
26. SK91 M. Stonebraker and G. Kemnitz. The Postgres next-generation database management system, Com. of the ACM, 34(10), pp 79-92, 1991.
27. SKS97 A. Silberschatz, H. K. Forth, S. Sudershan. "Database System Concepts", McGraw-Hill, New York, 1997.

28. SR86 M. Stonebraker and L. Rowe. The design of Postgres. In Proc. of the ACM SIGMOD Conf., pp 340-55, 1986.
29. SRL+90 M. Stonebraker et al. Third generation database system manifesto. ACM SIGMOD Record. 19(3) pp 31-44, 1990.
30. 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.
31. 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.
32. Zlo77. L Zloof, "Query-by-Example, a Database Language, IBM Systems J., 16(4), pp 324-343.
33. ZM89 S.B. Zdonik and D. Maier. Readings in Object-oriented Database Systems, Morgan-Kaufman, 1989.
□