Relations with Inherited Attributes

Witold Litwin*, Mohammad Ketabchi**, Tore Risch
Software Technology Laboratory
HPL-92-45
April, 1992

relational databases,
inheritance,
object-oriented
database systems,
next generation
relational systems

Object Oriented Database Systems (OODBSs) have demonstrated the utility of inheritance of data and methods. We propose enhancing the relational model with inheritance. The key ideas are relations (tables) in which some attributes are base attributes, while others are attributes inherited through predicates as in relational views. Such tables bridge the gap between base tables and views and become the basic data definition construct of a relational schema. The user benefits from semantically richer and more natural conceptual schemes. Queries and updates usually are formulated without logical navigation and referential integrity is implicit. Some view update problems are solved and versioning becomes easier. The inheritance predicates allow also for rules and for recursive tables that are a non-procedural solution to the common transitive closure and traversal recursion problems. Some of the proposed ideas should prove useful for OODBSs as well.

*Dauphine University, Paris, France
**Santa Clara University, Santa Clara, California

1 Introduction

Some object-oriented (OO) features, in particular the inheritance, have come to be recognized as desirable properties of a database. In a OO model, property P, (attribute, method or function P) which an object type (class) O inherits from object type O' applies to O as if P was declared for O. If O' is **Persons** and O is **Employees** one need not redefine for **Employees**, the attributes declared for **Persons**, such as **SS#** or **NAME**. One can declare only attributes particular to an employee, e.g. **SAL**, (salary), while a query can still refer to **Employees.SS#**, instead of **Persons.SS#**. The referential integrity is enforced, as updates to **Employees** cascade to **Persons**. New attributes of O' become automatically reusable by queries to O. Thus, inheritance facilitates the application modeling and can make the use of a database more effective.

If **Persons** and **Employees** were relations, an SQL query referring to **Employees.SS#** would fail. To allow for such a query, one would need to create a view, with attributes of both **Persons** and **Employees**. The view would need a name different from those of the base tables, requiring that either queries about **Employees.SS#**, use this name instead of **Employees** or that the **Employees** relation is renamed. The latter choice would make impossible to add new attributes to **Employees**, since it would be a view. Finally, if this approach were systematic, one would typically end up with many views. Clearly, relational databases do not allow for inheritance at present [19] except for a primitive form of inheritance for views, and sometimes for referential integrity [5].

In what folllows, we show enhancements to the relational model that provide inheritance. The key concept is relations with inherited attributes (IA). A relation with IAs typically has both base attributes (BA) and *inherited attributes* that are views of other attributes through **SELECT** expressions. A relational database is a set of relations with IAs and traditional base relations and views are special cases of relations with IAs. Base relations have no IAs, while views have only IAs.

In Section 2, we discuss the concept of tables with IAs in more detail. We present the extensions to the **CREATE TABLE** statement of SQL and we discuss query evaluation principles. Finally, we present the semantics of updates and we illustrate the use of IAs through examples.

Inheritance discussed in Section 2 is a static inheritance [15]. Section 3 generalizes it to some forms of dynamic inheritance in the relational model. The user benefits by more non-procedural manipulations,

including recursive queries.

Section 4 compares properties of the relational inheritance we propose with related work. We show advantages of relational inheritance over that in current OODBSs, due to its definition through predicative expressions. We show that a similar approach should be beneficial to OODBs as well. We also show advantages of our approach over other proposals of extensions to the relational model with OO features. Finally, Section 5 points to further research problems.

2 Tables with Inherited Attributes

2.1 Rationale

Consider the Suppliers-Parts database, usually called S-P, Fig. 1:

(1)

```
S (S#*, SNAME, STATUS, CITY)
P (P#*, PNAME, COLOR, WEIGHT)
SP (S#*, P#*, QTY)
```

The '*' denotes the key attributes. The relations S and P model suppliers and parts [3]. The relation SP models supplies, associating suppliers and the parts they supply. Since a supply associates all properties of a supplier and of a supplied part, the SP schema should naturally be:

(2)

```
SP (S, P, QTY)
```

where S, P are relations. However, then it would not be in First Normal form (1NF), which is not advisable [3]. In flat form, SP should be:

(3)

```
SP1 (S#*, SNAME, STATUS, CITY, P#*, PNAME, COLOR, WEIGHT, QTY)
```

SP1 has all the attributes of S and P. However, this form of SP is still usually ineffective, because of redundancies and anomalies [3, 20, 5]. The use of foreign keys S# and P# which abstract S and P in SP in (1), avoids these problems. They act like objects identifiers (OIDs) in OO databases and can be semantically void. The keys define the inheritance path (IP) between S and SP through which a supply inherits the attributes of a supplier with the same key. A similar path exists between P and SP. However, these IPs are unknown to the SP schema (1). In a typical relational language like SQL, the user has to explicitly define inheritance paths in queries. This leads to unnatural logical navigation, e. g., the use of equijoins `S.S# = SP.S#` and `P.P# = SP.P#` on foreign keys in a query.

Another solution for a full form of SP would be the *full view*:

(4)

```
CREATE VIEW SP2 AS SELECT S.*, P.*, SP.QTY
                          FROM SP S P
                          WHERE S.S# = SP.S# AND SP.P# = P.P#
```

SP2 has the same attributes as SP1, but they are all inherited. Hence it does not have the disadvantages of either SP or of SP1. However, SP2 has other disadvantages. A view needs a different name, e.g., SP2, than any base relation [1]. Also, the SP relation does not seem necessary anymore in the S-P conceptual schema which now becomes unorthodox:

```
S-P = (S, P, SP2).
```

One consequence is that no new attribute can be (directly) added to SP2, which can become a problem for the user if the DBA (as often) is reluctant to alter the corresponding base table. Finally, this approach would lead in general to many full views in relational schemes, since many different inheritance paths can exist. These are most likely the reasons why no one recommends it for the practical use.

[1] instead of SP2 being the name of a view, the SP relation could be renamed SP1 and the view named SP.

Base tables SP and SP1 do not inherit anything, while view SP2 inherits everything. This dichotomy is the origin of the discussed problems. It is also less flexible than the OO paradigm. A natural idea steams out, of extending the relational model with relations that could have jointly both: the base and the inherited attributes. The definition of SP could be then as follows:

(5)

```
CREATE TABLE SP

  SELECT S.* FROM S SP WHERE S.S# = SP.S#          /* inheritance path from S
  SELECT P.* FROM P SP WHERE P.P# = SP.P#          /* inheritance path from P
  QTY  INTEGER                                              /* base attribute
```

Fig. 2 shows the conceptual schema of S-P and tuples. Each SP tuple now has instances of views of all the corresponding attributes. It also has a base attribute QTY. This makes the SP schema semantically more complete and natural than the SP schema in (1). Although, the attributes of SP are conceptually those of table SP1 and of view SP2, SP is neither one, as its schema incorporates both types of attributes.

Note that SP inherits from S and P only the values corresponding to asserted supplies, i.e., to tuples first inserted into SP with the corresponding key values SP.S# and SP.P#. Thus, SP does not inherit values related to supplier 'S5' for instance, who in S-P does not currently supply any parts, and does not inherit values related to part 'P6' that has no supplier at present.

An implementation of SP with IAs should allow for the definition of tuples gluing the corresponding instances of S and of P. The choice of (foreign) keys S# and of P# is natural, althought not the only possible. The traditional table SP appears to be the canonical implementation, as shown in Fig. 2. It retains for SP with IAs the advantages of the original SP, as defined by (1). As an implementation, it now belongs however only to the internal schema.

A query to SP would now be formulated as to SP1 or SP2, i.e., without logical navigation. For instance, the query "select names of suppliers who supply parts with weight of 10 and quantity of each part" would be simply:

(Q1)

```
SELECT SNAME, QTY
       FROM SP
       WHERE WEIGHT = '10'
```

Obviously, a query to SP should be evaluated for an inherited attribute, as a query to a view and as a traditional query otherwise. The evaluation algorithm should extend the query modification technique [17]. The FROM clause should be expanded with S and P. The inheritance predicates should be added to the WHERE clause, as terms ANDed to those already in the query. (Q1) would end up in a traditional, more cumbersome form:

(Q1')

```
SELECT SNAME QTY
       FROM S SP P
       WHERE S.S# = SP.S# AND SP.P# = P.P# AND P.WEIGHT = '10'
```

Note all that the discussed advantages come with almost no cost. If the canonical implementation is used, the same storage and processing requirements can characterize in practice the the traditional S-P and that with IPs.

2.2 Table creation

2.2.1 CREATE statement

We now present the general syntax and semantics of the CREATE TABLE statement for relations with IAs. This will allow for a deeper discussion of query evaluation. It will also allow for the exploration of further possibilities already popular in OO paradigm, e.g., graceful specialization, overriding and versioning. Examples, mostly in Section 2.5, will illustrate the discussion.

We adopt the following syntax extending that of traditional SQL:

```
CREATE [TABLE] T
[I1] SELECT [i1] FROM R, R'...  WHERE ...  [WITH CONTENT], [WITH CHECK], [AS VERSION]
...
C1 type [NOT NULL]
...
[KEY (...)]
```

The terms in brackets are optional. The term **TABLE** is optional, since there is no need for a **CREATE VIEW** statement. The syntax of a **SELECT** expression is a superset of that of SQL, where some terms can be implicit. This the case of the SQL terms **X.*** and in general of equijoins defining IPs. In our example, the definition (5) of **SP** could be simply:

(6)

```
CREATE TABLE SP

SELECT FROM S                  /* attr. inherited from S
SELECT FROM P                  /* attr. inherited from P
QTY  INTEGER                   /* base attribute in SP
```

2.2.2 Role names

The i_k are (sets of) inherited attributes, optionally renamed to *role names* I_k. Role names provide aliasing and avoid name conflicts. A role name can be a new name mapped by position to an attribute name, as is classically done in a view definition. It can also be an atomic (role) prefix of form **X.** added to all i_k. Finally, it is by default in the form **X.Y** for any IA **Y**, where **X** is the source table of **Y**, provided there is only one **X**. This use of role names extends that of domain role names in [3], the "." replacing the "_", as more homogeneous notation. In a **SELECT**, a role name in the form **X** or **X.*** means that all attributes with this

role name should be selected.

A query can refer to a role name as it would refer to a unique proper name of an attribute in SQL. Multiple role names can be enclosed in parentheses. For example:

```
CREATE SP
SUP SELECT FROM S;
(P#, NM, COL, W) SELECT FROM P;
QTY  INT;
```

would lead to the schema:

```
SP (SUP.S#, SUP.SNAME, SUP.STATUS, SUP.CITY, P#, NM, COL, W);
```

The query selecting cities from SP would be:

```
SELECT SUP.CITY FROM SP;
```

The query selecting all attributes of a supplier in London would be:

```
SELECT SUP FROM SP WHERE SUP.CITY = 'London';
```

Range variables, if any, also define role names. Finally, an attribute can be inherited through different paths, in which case different role names in the target table are mandatory. The following could be the table for pairs of suppliers with the same ZIP code.

```
CREATE COMZIP
          SELECT FROM S
     S1   SELECT FROM S
          ZIP    INT
```

The attributes i_k can be named explicitly. Alternatively, all attributes of source tables are selected. One can also use the form R\a_j\. Then, all but the attributes named a_j are inherited from R.

2.2.3 Default joins in inheritance predicates

To form a **SELECT** expression defining an IA, the traditional view definition syntax of SQL is used, with extensions already discussed. Furthermore, each inheritance predicate can include a clause with equijoin terms linking target tuples with inherited attributes to corresponding source tuples. These joins did not exist in view definitions, but may be necessary for IP calculus. As in (5), they usually evaluate to the traditional equijoins on foreign keys. They can be provided explicitly, but exist by default anyhow. Since keys are not always inherited, the default is an equijoin for each IA. The general form of the default clause is as follows:

Let $\mathtt{A}_1, \ldots, \mathtt{A}_j$ be IAs in a **SELECT** clause, inherited respectively from $\mathtt{a}_1, \ldots, \mathtt{a}_j$, where each $\mathtt{a}_k$ is a source attribute, perhaps of a range variable or perhaps itself inherited, or a value expression, e.g., **PRICE*QTY**. The default clause is ANDed to all the others in the **WHERE** clause and has the following form:

...**AND** ($\mathtt{A}_1$ = $\mathtt{a}_1$ **AND**,..., $\mathtt{A}_j$ = $\mathtt{a}_j$)

The calculus of the default clause can in general be reduced to a more efficient one. The case when keys are inherited is discussed in the next subsection. No calculus is necessary when the table with IAs is a view. Other possibilities depend on implementation. Multiattribute joins can be transparently replaced with more efficient joins on hidden **IDE** type attributes, discussed in next subsection.

Through **WHERE** clauses with implicit joins, the **SELECT** expressions define IPs to the corresponding source relations. Fig. 3 shows the corresponding database graph for **S-P**. An IP from **V** to **T** in a **SELECT** expression is seen as an edge **V -> T** of the database graph **D**, whose nodes are all the tables and that contains all the corresponding edges. A **SELECT** expression defines an edge per the source table it refers to or per range variable. An edge from **V** to **T** is labeled with the predicate that is the subexpression of the **SELECT** expression involving only and all terms referring to **V**.

An IA whose source is a value expression is called a *computed attribute*. In traditional relational databases, computed attributes cannot exist in tables together with the base attributes from which they are derived. The new capability is very attractive, as it naturally offers spreadsheet-like possibilities for relations (viz. Ex. 2.5.j).

2.2.4 Base attributes

A base attribute (BA) can be of any traditional SQL type, e.g., **INT, or CHAR**. It can also be of two new types, denoted **IDE** and **EXE**.

The values of type **IDE** are possibly universally unique. An **IDE** attribute can be a system defined key. It can act as an OID or a surrogate, [4], but is defined explicitly, bears a name and can be referred to by name. It is useful when a primary key is semantically void or is multiattribute and thus cumbersome for queries with interrelational equijoins.

An attribute of **EXE** type corresponds to the Unix executable type. It contains methods (programs, procedures,...). When selected, it is transparently executed. **EXE** attributes are not discussed here, given space limits of this paper, see [12].

2.2.5 Key inheritance

The key clause designates primary key attributes. The key may contain both inherited and base attributes. Inherited keys enter the key implicitly, if no **KEY** clause exists.

The keys and base attributes of a table **T** with IAs, define *the canonical implementation* of **T** in the internal schema. As for **SP** in Fig. 2, it consists of the key and of all other base attributes. Other more efficient implementations are possible as well. In any case, the choice of implementation is considered system dependent.

If primary keys are inherited, the default equijoins boil down to the joins on keys only, e.g., to the familiar equijoins in our **S-P** example. That is why such equijoins do not need to be provided. The same is true, if an inherited attribute is known to be a candidate key. If the implementation is not the canonical one, the default clauses can be further transformed at the implementation level.

2.2.6 Content inheritance

A **CREATE T** statement defines only **T**'s schema, unless the **WITH CONTENT** option is used. A tuple **t** of **T** has to be created through the **INSERT T** statement. The statement should at least provide the values of

key attributes of t, since any key values are traditionally **NOT NULL** by default ([12] shows that it can be useful to relax this constraint, especially in the multidatabase context). Values of IAs of t do not need to be provided. They are enabled through inheritance, by default. Those that are provided basically cascade to sources, unless the update semantics prohibits the operation.

By contrast, the **WITH CONTENT** option allows for inheritance of the tuples produced by the **SELECT** expression without the explicit insertion into T. A tuple is then inherited iff it conforms to T's semantics. It should especially respect the **NOT NULL** constraints on T's base attributes.

If there are no base attributes in T, the **WITH CONTENT** option leads to a view. Fig. 4 shows both kinds of inheritance for S-P, using examples (b) and (c) in Section 2.5.

2.3 Queries

Queries have SQL syntax with extensions that we will introduce progressively. The operational semantics for IAs extend that of query modification for of traditional attributes in views [17]. In short, let Q be a query referring to an inherited attribute T.I. Any reference to T.I in Q is replaced with that to its source in the **SELECT** clause, let it be the attribute V.i. Then V enters the **FROM** clause of Q and the IP in the **WHERE** clause of T.I, including the implicit joins, is ANDed to the **WHERE** clause of Q. The substitution process is carried further if V.i happens to be itself an inherited attribute.

The novelty in the operational semantics for IAs is that the final query can refer to both table T and table V, instead of to V only. The reason is that base attributes may exist for T. The equijoin clauses in the T.I definition provide the necessary edges making Q meaningful. A meaningful query has the query graph connected, the nodes being the tables referred to in the query and the edges being the equijoin terms.

2.4 Updates

Updates to IAs in T basically cascade to the sources. Exceptions are deletions in T, unless **WITH CONTENT** option is used. The results can be an insertion of a new tuple into the source table, let it be T' or no effect on T' or a tuple modification in T' or a deletion in T', or, finally, an abort of the update. The details are

beyond the scope of this paper. If the T' key is not inherited by T, then the inheritance expression can act as an integrity constraint, prohibiting updates to T, if no corresponding values exist in T'. The **CHECK** option of SQL is considered as generalized to this new context.

As for views, some inheritance predicates, e.g., with aggregate functions, may make cascading impossible. However, insertions to a table with IAs may remain possible, as we show in the next section. Deletions in T' usually cascade to T with obvious effects.

One consequence of this update semantics is that IAs usually enforce referential integrity. If needed, exceptions can be created using ideas for the **CASCADE** statement in [5].

Another consequence is the need for making tables T and T' mutually consistent, if T' becomes the source for T following the restructuring of T through the **ALTER** T statement. The statement should be able to convert a base attribute into an IA. It can happen that a tuple in T should then inherit a value from T' that was not there. For instance, one may alter a table **EMP** to inherit from a table **PERS**, while there is no tuple in **PERS** for an already existing one in **EMP**. For instance, **PERS** is just created as generalization of **EMP** and yet empty. One may need then to update **PERS** or **EMP** or both, as a part of the alteration process. The corresponding production rules are beyond the scope of this paper.

2.4.1 Overriding

A BA and an IA can share a name, in which case the BA *overrides* the IA. It is nevertheless always possible to refer to the IA value, using the explicit or default role name of the IA. We distinguish two possibilities for the override:

- Let X be a BA in table T, T'.X the default role name of an IA and Q a query referring to an attribute X. There is no attribute with role name T''.X in T. Then, when a query selects X:

1. Either only the values of X are considered
2. Or, whenever X does not exist, then the value of T'.X is considered.

We call form (2) the *intra-tuple value inheritance.* For this form, we distinguish between a value that does not exist and a value that exist, but is known to be unknown. The latter is our semantics for a null value

(see [5] for more on null values). We consider that the semantics of the **SET** clause of the **UPDATE** statement allows for both cases, although we will ll not work out here the corresponding syntax. The **NOT NULL** clause means that a value must exist. An existing value, null or not null, can be unset, i.e., made not existing, unless of course it should be **NOT NULL**.

An **INSERT T (..., X,..)** statement instantiates **X** and does not cascade to **T'**. One can also instantiate **T'.X**, through the **INSERT T (..., T'.X,..)** statement, in which case the new value can cascade. A deletion of a tuple in **T** in form (2), unsets every **X** in the tuple and can cascade only to sources of not overridden IAs.

Both forms (1) and (2) are justified through case studies. In particular, value inheritance allows for versioning and is a solution to some view update problems [5], as we'll show through examples later on. From now on we consider intra-tuple inheritance as the default, form (1) being an option whose syntax is not detailed here, in the **CREATE** statement.

2.4.2 Versioning

A version is a copy of a class in which updates to the source remain visible or can be overridden, i.e., **versioned**. Intra-tuple value inheritance allows for versioning of an attribute. The **AS VERSION** option simplifies the versioning of several attributes simultaneously. It versions all the attributes inherited through the corresponding **SELECT** and implies the **WITH CONTENT** option by default.

This approach preserves the traditional semantics of versioning and offers new possibilities. The traditional semantics would be here the versioning of the entire base tables. New possibilities are the versioning of any tables with IAs, including views, selection of only some attributes or of only some tuples. One can also create new attributes specifically for the version or one can combine several source tables (Ex. 2.5.j). As usually, the updates to the versioned attributes may stay local to the version, but one can make them cascading to the sources as well. Finally, **AS VERSION** option can be combined with **WITH CHECK** option, filtering illegal updates to the version. the **WITH CONTENT** option by default.

This approach both preserves the traditional semantics of versioning and offers new possibilities. The traditional semantics would imply the versioning of all the entire base table. New possibilities are the versioning

of any tables with IAs, including views, the selection of only some attributes or of only some tuples. One can also specifically create new attributes for a version or one can combine several source tables (viz. Ex. 2.5.j). As usual, the updates to the versioned attributes may remain local to the version, but may be made to cascade to the sources as well. Finally, the **AS VERSION** option can be combined with the **WITH CHECK** option, filtering illegal updates to the version.

2.4.3 Rules

A selection expression can also act on a part of the target table. Its effect can be that of a rule, preserving integrity or cascading values like a production rule. Our formalism is however less procedural. It describes only the database state, instead of perhaps numerous lower-level actions to be performed on insertions or deletions to keep that state. Example (k) below illustrates the new possibilities.

2.5 Examples

(a) - There are suppliers in **S** unknown to **SP** who do not supply any parts yet (potential suppliers). Select the names of all suppliers who supply parts (we do not care about duplicates). Then, select the names of all suppliers.

```
(1)  SELECT SNAME FROM SP;
(2)  SELECT SNAME FROM S;
```

In (1), **SNAME** is an inherited attribute, in (2), it is a base attribute. The query (1) will be evaluated to **Q'**:

```
SELECT SNAME FROM S SP WHERE S.S# = SP.S#;
```

Suppliers unknown to **SP** will be eliminated by the equijoin.

(b) - Create a table inheriting suppliers in London.

```
CREATE SL
     SELECT FROM S WHERE CITY = 'LONDON' WITH CONTENT;
```

This is a traditional view. The inherited tuples could be those in the shaded area of Fig. 4a.

(c) - Specialize SL with the area code (NW, NE,..):

```
ALTER SL
     AREA  CHAR (2);
```

The SL schema is now (S, AREA). It is no longer a view. The new tuples are extensions of only and all those in SL that were defined in (b). Insertions and deletions in S are inherited by SL. The value of AREA in an inherited tuple is initially null and has to be set by an UPDATE SL operation. For Fig. 4.1b, it would be:

```
UPDATE SL
SET AREA = 'NW' WHERE S# = 'S1';
```

Alternatively, all the values in a tuple can result from an INSERT SL statement. Insertions would be necessary if the definition of SL were without WITH CONTENT. The table would then be created empty. To create the tuple at Fig. 4.1a, one could state:

```
INSERT SL (S#, AREA) VALUES (S1, NW);
```

This would enable the inheritance of remaining values in the tuple.

The key of SL remains the same, i.e., S#. The canonical implementation is:

```
SLC (S#*, AREA)
```

A query such as "find the status and names of all suppliers in the area NW" would be:

```
SELECT SNAME STATUS FROM SL WHERE AREA = 'NW';
```

Its operational semantics is:

```
SELECT SNAME STATUS
FROM S SL
WHERE S.S# = SL.S# AND S.CITY = 'LONDON' AND SL.AREA = 'NW';
```

(d) - Create SL with the status hidden, since it is private to S.

```
CREATE SL
     SELECT S\STATUS\ FROM S WHERE S.CITY = 'LONDON'
     AREA  CHAR (2);
```

A query referring to SL.STATUS would fail. Alternatively one could use the GRANT statement to restrict access to STATUS.

(e) - Create a table, COS, for the colocated suppliers. Add the travel time by car between each pair.

```
CREATE COS
     SELECT FROM S S1, S S2
     WHERE  S1.CITY = S2.CITY AND S1.S# < S2.S# WITH CONTENT
     TTIME INT
```

The range variable names S1 and S2 act as role names. The key (S1.S#, S2.S#) is implicit as entirely inherited. An example query could be:

```
SELECT TTIME S1.SNAME S2.SNAME FROM COS WHERE TTIME < '10'
```

The traditional solution would be the table COS' (S1.S#, S2.S#, TTIME) which is now the canonical implementation. User would need to populate COS' and to keep it consistent with COS. The query would be more cumbersome:

```
SELECT TTIME S1_SNAME S2_SNAME FROM COS COS'
WHERE COS.S1_S# = COS'.S1_S# AND COS.S2_S# = COS'.S2_S# AND TTIME <
'10';
```

(f) - For the colocated suppliers in Paris or in London, consider the travel by metro.

```
CREATE COS-PL
     SELECT FROM COS WHERE CITY = 'PARIS' OR CITY ='LONDON' WITH CONTENT
     TTIME INT
```

COS-PL specializes COS. TTIME overrides COS.TTIME. The query:

```
SELECT TTIME S1.SNAME S2.SNAME FROM COS-PL WHERE TTIME < 10
```

refers now to the time of travel by metro. The inherited travel time by car is still available for manipulations of COS-PL under its role name COS.TTIME.

(g) - Insert the following tuple to SP:

```
INSERT INTO SP (S#, SNAME, CITY, P#, PNAME, QTY)
      VALUES (12, 'PH', 'PA', 34, 30);
```

All the attributes except QTY will cascade to S and P, unless the corresponding tuples exist there already. In the latter case, it suffices to insert key and base values.

(h) - Consider the table:

```
CREATE STC
SELECT STATUS, CITY FROM S WHERE STATUS > '100' WITH CONTENT;
```

Insertions into STC are basically impossible, since S.S# is not selected[2]. If, however, STC is the table:

```
CREATE STC
     SELECT STATUS, CITY FROM S WHERE STATUS > '100;
```

[2]One aspect of the classical view update problem [5]).

then, insertions are allowed whenever there is the corresponding tuple in S. The condition on STATUS should prohibit updates resulting in STATUS =< 100, or should lead to the deletion of the corresponding tuples from STC.

(i) - Consider the table:

```
CREATE PQ
    (P#, TOTQTY) SELECT P#, SUM(QTY) FROM SP
    GROUP BY P# WITH CONTENT;
    TOTQTY INT;
```

Without the base attribute TOTQTY, PQ is not an updatable view [5]. This can be a problem for the user who knows that a TOTQTY value is in error. In contrast, the existence of the base attribute TOTQTY allows for PQ.TOTQTY updates. The correct value is stored in the base attribute and acts as an exception through the principle of intra-tuple value inheritance. It is transparently selected, instead of the inherited value, iff the base attribute is not null. Thus tables with inherited attributes are a solution to the corresponding aspect of the view update problem.

(j) - Create a version P1 of P to experiment with characteristics of any part with WEIGHT > 10. For experimenting with WEIGHT, provide P1 with the computed attribute giving the total weight of supplied parts with the same P#.

```
CREATE P1
SELECT FROM P AS VERSION WITH CHECK WHERE WEIGHT > 10;
TWEIGHT SELECT SUM(QTY)*WEIGHT FROM P SP GROUP BY SP.P#;
```

Experiments with P1 will not affect P. If a versioned attribute has no new value, the query will get the inherited one. One can further create a version P2 of P1, or a whole hierarchy of versions. Through intra-tuple value inheritance, an attribute in a version inherits the value overridden by the nearest version on the path toward the root, or the value in P.

(k) - The following inheritance expressions within a CREATE S statement would implement the rules: (1) the status of suppliers in London is 100, (2) the status of supplier 'S1' is that of 'S3', and (3) any status is under

150.

```
(1) STATUS SELECT 100 FROM S WHERE CITY = 'London'
(2) STATUS SELECT X.STATUS FROM S, S X WHERE X.S# = 'S3' AND S.S# = 'S1'
(3) STATUS SELECT X.STATUS FROM S X WHERE X.STATUS < 150
```

Table S is both target and source for the expressions, eventually logically replicated as the source through the use of range variable X. The first expression takes S as the source, finds tuples matching the predicate, and assigns 100 to the STATUS field of each such tuple. The next expression finds the tuple in S x X that matches the predicate and assigns to the S.STATUS field the value of the X.STATUS field of the tuple. As a production rule, it will thus propagate to 'S1' a change to the status of 'S3'. However, in this case one would need to write three production rules to propagate the insertion, an update, and the deletion. The last expression does a similar job for any STATUS, provided that (source) STATUS is under 150. It therefore de-assigns any other attempted value or signals an integrity violation. The semantics of a WITH CHECK option could allow the signalling of a rule violation to override the default silence.

3 General Inheritance Calculus

3.1 Rationale

The inheritance through SELECT clauses goes downwards in the database graph, as in Fig. 3. It is modelled after the static downward inheritance in OODBs [15]. It allows queries invoking only attributes *in schemas* which are attributes of the form T.A where A is a base attribute in T or is defined by a SELECT clause in T. The OO approach also uses the concept of dynamic inheritance [15], making possible all kinds of inheritance, defined by a query and resolved by exploring the database schema and its content. The OO concept of dynamic inheritance appears useful for relational databases as well. Dynamic inheritance in relational databases results in more useful queries without logical navigation. Three kinds of dynamic inheritance exist in a relational database:

Let G be a database graph. An IP in G is a *directed path* if all edges follow the same direction, otherwise it is *undirected*. Let T and T' be two tables. Table T' has an attribute named A in its schema, but T does not. T can nevertheless inherit A in the following ways:

- Indirect downward inheritance

  A can be indirectly downward inherited by T if there is a directed IP from T' to T through which A gets a different role name A' $\neq$ A. The IP should be the shortest path between T and any T', if there are several tables T'.

For instance, **SNAME** is indirectly downward inherited (**IDI**) from S by COS (Fig. 5). The ID inheritance can be multiple. Invoking a single name in a query then suffices to access many roles, e.g. **COS.SNAME**. ID inheritance is as yet unknown to the OO approach which implicitly assumes that inheritance preserves method names.

- Upward inheritance

  A can be upward inherited by T if T' is on a directed IP from T and this is the shortest possible path.

For instance, **SP.QTY** can be upward inherited by S (Fig. 5).

- Undirected inheritance

  A can be indirectly inherited by T if there is an undirected IP between T and T'. The IP should be the shortest such path.

Undirected inheritance can provide meaning to a query referring, for instance, to **S.WEIGHT**. This type of inheritance is also unknown to the OO approach.

These kinds of inheritance allow for queries that would otherwise be meaningless. A query can invoke IDI, or upward inherited, or indirectly inherited attributes. The query can also be incomplete, which means it has a disconnected query graph, even if all invoked attributes are in the schemas of the qualifying tables. Finally, the query can invoke unqualified attribute names, i.e., where the schema cannot find the table containing the attribute from the query itself. Such queries either have no **FROM** clause or designate an unqualified attribute X as *.X. Fig. 5 shows some such queries, and examples in Section 3.4 show more.

The general inheritance calculus consists of a determination of the base attributes to enter a modified query through IPs. There can be multiple choices for attributes and IPs, as well as for inheritance conflicts.

Several tables sharing the name of an unqualified attribute can lead to multiple qualified names. There can be multiple IPs between an IDI attribute and its target table, or there can be multiple role names, e.g., between COS and SNAME for the invocation of COS.SNAME. There also can be multiple source attributes sharing a name for each kind of inheritance. Finally, there can be inheritance conflicts, e.g., an attribute name can correspond both to an IDI attribute and to an upward inherited attribute.

3.2 Calculus rules

We now discuss the general inheritance calculus we propose. The corresponding algorithm is in Appendix A. Step 1 qualifies a multiple identifier A with all tables T with a base attribute A, if there are such Ts. Otherwise, one looks for all tables T with an IA whose role name is A, not inherited downward from another attribute A.

Step 2 looks for the table sources of IAs other than downward inherited (DI) attributes. In case of conflict, downward inheritance has the priority over upward inheritance, which preempts undirected inheritance.

The result of Step 1 and of Step 2 is a query or subqueries with DI attributes only. Step 3 modifies each subquery as described in Section 2, to a query with base attributes only.

Unlike in Section 2, however, we assume now that a resulting query can be incomplete. In Step 4, IPs are then used to provide the missing semantics. The choice of the shortest-path semantics in this step, as for the downward and upward inheritance calculus, matches the intuition as we show in the next section. One could, however, argue for other paths as well, e.g., for the longest path especially in the case of upward inheritance when the most specific value is desired, or for any other paths. However, such an approach seems cumbersome in practice. Use of unqualified attribute names also allows for the possibility of using IPs to provide missing semantics, as examples will show.

It can also happen that no complete query is found, although one could consider then the usually meaningless completion through the cartesian product that always exists.

Step 5 combines the subqueries into the final result R. It can, of course, happen that R is empty if one of the steps failed, e.g., no source table was found for an attribute. Union semantics for subqueries sharing a target list seems the rationale choice for combining subqueries into a target list. In contrast, multirelational semantics is preferable for subqueries with different target-lists. It carries more semantics, and might be the only possible solution for union-incompatible **SELECT** lists. However, multirelations cannot be used as sources in **SELECT** expressions in schemas.

The calculus rules are extended later for recursion. As above, they match but generalize the inheritance principles in OODBs. Examples will illustrate these rules in more detail.

3.3 Examples

We now consider a more traditional OO application, inspired by [7]. We call the database PH. If it were an OO database it would contain the types **Persons, Employees, Secretaries, Researchers** and **Taxpayers**. A taxpayer can be a person or a subsidiary of PH. An employee has a basic salary, but secretaries have an additional salary. Furthermore, researchers and secretaries use workstations. The corresponding class hierarchy and (multiple) inheritances are obvious. The corresponding relational schemas could be as follows (Fig. 6 shows the database graph):

```
PERS (PID#*, SS#, NAME, ADDR, TEL)
EMP (PERS, TAXP, E#, DIV, TEL, SAL)          /* subtype of Pers & Taxp
SEC (EMP, WS, SAL)                           /* subtype of Emp
RES (EMP, DEP, WS)                           /* subtype of Emp
TAXP (TNAME*, ADDR*, TAX)
```

The attribute **PID#** is an identification attribute of type **IDE**, since persons of the same family can share any other attribute. The **PERS.TEL** is private, while **EMP.TEL** is the office number that overloads the personal one. The **SEC.SAL** overloads that of an employee, but is understood as a supplement to the basic employee salary.

(a) - Select names of employees.

```
SELECT NAME FROM EMP;
```

NAME means here **EMP.NAME**. It is statically and downward inherited from the base attribute **PERS.NAME**, and there are no other attributes **NAME**. As in Section 2, and also according to Step 3 of the inheritance algorithm, the query should be modified to refer to that attribute through the corresponding inheritance path. The same function (method) would be chosen if **PH** were an OODB. The resulting query is complete so Step 4 does not apply. The final result is:

```
SELECT NAME FROM PERS WHERE PERS.PID# = EMP.PID#;
```

(b) - Select names and telephones of employees with salary '123'.

```
SELECT NAME TEL FROM EMP WHERE SAL = '123';
```

EMP table has the base attribute **TEL**. So, **EMP.TEL** designates this attribute. The final query is:

```
SELECT NAME EMP.TEL FROM PERS EMP
     WHERE SAL = '123' AND PERS.PID# = EMP.PID#;
```

The attribute **EMP.TEL** overloads **PERS.TEL**. If the personal number were needed, one would use the role name:

```
SELECT NAME PERS.TEL FROM EMP PERS WHERE SAL = '123';
```

If any or both telephone numbers are wished, one should say:

```
SELECT NAME *.TEL FROM EMP WHERE SAL = '123'
```

The attribute **TEL** is a multiple identifier of any table with attribute **TEL** in the database. Step 1 of the inheritance calculus would produce two subqueries:

```
SELECT NAME PERS.TEL FROM PERS EMP WHERE SAL = '123'
SELECT NAME EMP.TEL FROM PERS EMP WHERE SAL = '123'
```

The result would be a multirelation. These relations could be joined by union provided **EMP.TEL** is union-compatible with **PERS.TEL**. A loss of semantics would result, however. One would no longer know whether a phone number is the personal one or the office number.

The multiple application of such methods does not seem to have been considered in OO models.

(c) - Select department, salary and taxes of a researcher whose address is 'Chester Str'.

```
SELECT DEP SAL TAX FROM RES WHERE ADDR = 'Chester Str'
```

Here, **ADDR** is not a DI attribute. Step 2 finds attributes that **RES.ADDR** inherits, and their role names in **RES**. Two subqueries result:

```
SELECT DEP SAL TAX FROM RES
     WHERE (EMP.(PERS.ADDR)) = 'Chester Str';
SELECT DEP SAL TAX FROM RES
     WHERE (EMP.(TAXP.ADDR)) = 'Chester Str';
```

These queries are modified in Step 3 and both are complete. Their target lists are the same, so they are finally joined by union. Note that they share IPs and common subexpressions, so there is room for optimization.

(d) - Select name and any salary of a researcher in department 'DTD'.

```
SELECT NAME *.SAL FROM RES WHERE DEP = 'DTD';
```

Since SAL is unqualified, Step 1 applies and produces two subqueries:

```
(1) SELECT NAME EMP.SAL FROM RES EMP WHERE DEP = 'DTD';
(2) SELECT NAME SEC.SAL FROM RES SEC WHERE DEP = 'DTD';
```

Both queries are incomplete. Steps 3 and 4 complete (1) with the shortest directed IPs between **PERS** and **EMP, RES**, to the query:

```
SELECT PERS.NAME EMP.SAL FROM EMP RES PERS WHERE RES.DEP = 'DTD'
        AND EMP.PID# = RES.PID# AND PERS.PID# = EMP.PID#;
```

This query expresses conceptually the DI of **EMP.SAL** by **RES**, i.e., the query:

```
SELECT NAME SAL FROM RES WHERE DEP = 'DTD';
```

Query (2) requires an IP between **RES** and **SEC**. However there is no directed IP between these tables. Step 4 will choose the shortest undirected IP. When the IP labels are added, the query is finally:

```
SELECT PERS.NAME SEC.SAL FROM EMP SEC RES PERS
     WHERE RES.DEP = 'DTD'
     AND RES.PID# = EMP.PID# AND EMP.PID# = SEC.PID#;
```

The query expresses the dynamic undirected inheritance of **SEC.SAL** by **RES**, i.e., the query:

```
SELECT NAME SEC.SAL FROM RES WHERE RES.DEP = 'DTD'
```

Alternatively, **RES.DEP** can be seen as dynamically inherited by **SEC**.

The query produces tuples, iff there is a researcher who is also a secretary and so has two salaries. The modified form can be optimized. The multirelation generated by the whole query corresponds to a semi-outerjoin, where any value of **EMP.SAL** is preserved. A similar kind of inheritance, between subtypes of the same type, is not considered in OO models.

(e) - Select name, department and taxes of persons whose personal phone is '234'.

```
SELECT NAME DEP TAX FROM PERS WHERE TEL = '234'
```

The query requires upward inheritance for **DEP** and undirected inheritance for **TAX**.

(f) Select the names of employees using an HP123 workstation.

```
SELECT NAME FROM EMP WHERE WS = 'HP123';
```

Here **WS** is upward inherited by **EMP** from **SEC** and **RES**. Since there are two shortest paths, there will be two subqueries. They share the target list and so will be joined by union.

3.4 Recursive tables

Inherited attributes make recursive definitions of tables possible and non-procedural, unlike the traditional relational model. We simply allow an IP to point to the same relation (Fig. 7) through some role name. Queries implying a transitive closure or, more generally, a traversal recursion [16], also become non-procedural.

For instance, consider the classical example of a hierarchy of employees. One traditional way would be to create a table such as **EMP-H' (PID#, MAN_PID#)**. IAs allow for a more semantically complete definition as follows:

```
CREATE EMP-H
   SELECT FROM EMP
   MAN SELECT FROM EMP-H
   KEY (PID#);
```

The semantics of the statement is that **EMP-H** inherits all the attributes from **EMP**, and through the role name **MAN**, it also inherits them from itself. The values are inherited through the default equijoin **MAN.PID#** = **PID#**, where **EMP-H'** is the canonical implementation. Therefore, let **t** be a tuple in **EMP-H'** with, e.g., **PID#** = 1 and **MAN.PID#** = 3. The corresponding **t** in **EMP-H**, has all the attributes of **EMP-H'** and inherits these of the tuple **t'** having **PID#** = 3, defining the manager as an employee in **EMP-H'** and **EMP-H**. As **t'** possibly also inherits attributes of a manager by the same token, **t** recursively inherits the attributes and values of all the managers of the employee. For each **t**, the evaluation stops when no new tuple can be selected.

An attribute inherited through **MAN** gets **MAN** as the role name, e.g., **MAN.SAL**, or **MAN.(PERS.TEL)**, or **MAN.(MAN.SAL)** for the salary of a second level manager. Since employees can have more than one manager, **EMP-H** can be seen as having as many attributes as needed for the longest management path, padded with nulls at the right side for shorter paths. The number of tuples is the number of employees. A query referring to **MAN** is evaluated through the IP.

This semantics leads to the natural notation for the recursion, illustrated through the following example. The queries are: (1) select salaries of direct managers of 'litold', and, recursively, (2) those of any managers of 'litold'. The notation '..' stands for the transitive closure, e.g., **MAN.SAL** and **MAN.(MAN.SAL)**.

```
(Q1) SELECT MAN.SAL FROM EMP WHERE NAME = 'litold'
(Q2) SELECT MAN..SAL FROM EMP WHERE NAME = 'litold'
```

The whole notation, for the structure definition and queries is rather simple compared with the notation for logic database languages [20], as well as with the extensions of SQL allowing for recursive queries to non-recursive tables, e.g., SQL2. Note also that there is no need to define the rule for a manager of any level. Also, if the schema of **EMP** is altered, so is that of **EMP-H**, unlike in other known languages. For a more convenient designation of a specific element on the transitive closure path, the '..' notation can optionally be '.[i].' where i is a natural number denoting the path length, e.g., **MAN.[0].SAL** is equivalent to **SAL**. The path length i can be referred to in the query as if it were an attribute. This notation allows for the natural expression of popular transitive closure queries such as: "all managers above that of litold".

Recursive tables are an elegant solution to some important classical problems of traversal recursion [16]. The following examples illustrate this claim.

a. Each **EMP** employee is provided with a travel budget. A manager has a budget for her/himself, and is considered to be provided with the total budget of all the subordinates. Non-procedural manipulation of budget data is the dream of any managers. This dream can now become reality through the following alteration of **EMP-H**:

```
ALTER EMP-H ADD
```

```
BUDGET INT,
TBUDGET SELECT BUDGET + SUM(X.TBUDGET) FROM EMP-H, EMP-H X
  WHERE X.MAN.E# = E#;
```

Here BUDGET is a base attribute, and TBUDGET a recursively inherited one. The SELECT clause defines these attributes for each tuple in EMP-H, given its E# value. For simplicity, the values of TBUDGET are left undefined for employees who are not managers, assuming that SUM then yields zero.

b. Consider the part explosion problem as defined in [5], p. 216, with the traditional table PART_STR', added to the S-P database:

```
PART_STR' (P#*, MINOR_P#*, QTY);
```

Here QTY is the number of occurrences of the minor part within its parent. The problem of traversal recursion for this case is to produce the transitive closure of PART_STR', let it be PART_STR, with the right number of occurrences of the subpart in the part in each tuple. For instance, for ('P1', 'P2', 5) and ('P2', 'P4', 7) in PART_STR', PART_STR should also contain the tuple ('P1', 'P4', 35). IAs allow for a non-procedural solution to this problem through the following recursive table:

```
CREATE PART_STR

  SELECT FROM PART
  QTY INT
  MINOR_P SELECT *\QTY\ FROM PART_STR
  MINOR_P.[i+1].QTY SELECT X.QTY * MINOR_P.[i].QTY
    FROM PART_STR, PART_STR X
    WHERE X.P# = MINOR_P.[i+1].P#
  KEY (P#, MINOR_P#);
```

The recursive join on MINOR_P.P# = P# is again implicit. First, recursive selection does not select the QTY attribute. Second, recursive selection defines the QTY attribute for the parts in the tuples created

by the recursion and assigns its cumulative value. To find the quantity of a subpart somewhere deep in a part is now rather simple, e.g.:

```
SELECT MINOR_P..QTY FROM PART_STR
     WHERE P# = 'P1' AND MINOR_P..P# = 'P5';
```

4 Related work

4.1 Object-Oriented Methodology

Inheritance through **SELECT** clauses in relational databases, as in Fig. 3, is modeled after the static downward inheritance in OO databases [15]. As the examples show, it augments relational databases with the well-known OODB features, e.g., of subtyping, private and public methods, and overriding. Attribute inheritance in the relational schema is a kind of OO part inheritance [15]. IA values can also be subject to scope inheritance that are properties of the target table. The scope properties that an IA receives are the target table name as the qualifier, perhaps a different (role) name, or a new value type resulting from a value expression within the **SELECT** clause, e.g. **WEIGHT/1000** to convert from grams to kilograms. The scope inheritance can also include integrity constraints on the scope, authorization specifications, and other features. Finally, both upward and multiple inheritances are possible.

Relational inheritance offers possibilities not known to an OODBS. Relational inheritance is not limited to supertype-subtype relationships, e.g., **SP** that inherits from **S** and **P**, for it is a subtype of neither **S** nor **P**. The non-procedural formulation of (Q1) also is not possible in known OO query languages. Also, the OO-like relationship that exists between **COS** and **S** in example 2.5.e does not seem to be classified in OO modeling. Finally, the relational inheritance expression can be an arbitrary predicate. That is why **SL** in example 2.5.b is a derived type, not possible to declare in any OODBSs about which we know.

The relational value inheritance that we defined is modelled after the value inheritance for OODBSs [13]. However, value inheritance in [13] is limited to the component relationship. The multiple inheritance resolved through multirelations does not seem to have an OO equivalent either. Finally, we are not aware of OODBSs providing an undirected inheritance, a self- inheritance or a multidatabase inheritance. As examples show,

all these new possibilities in relational databases appear interesting, and should characterize an OODBS as well.

4.2 Relational Systems

Many efforts have been made to avoid the logical navigation problems of relational databases, or to enhance them with OO features. With respect to the former aim, our work follows ideas that have been developed regarding both the universal relation interface and the implicit joins method [20], [11], [1]. In addition to the concept of a table with IAs itself, the main novelty of our approach is the enrichment of the schema with inheritance paths. The additional semantics allows both for the evaluation of queries and for updates, areas in which the other approaches fail in practice.

With respect to adapting OO features to relational databases, the closest attempts are probably RM/T, the ANSI SQL3, Postgres, and Intelligent SQL (ISQL) [4], [14], [17]-[19], [2]. These proposals expand the language syntax with OO constructs for explicit creation of class hierarchies (except for Postgres), complex objects, and abstract data types. Such constructs are beyond this work, although the implicit result can be the same. Also, we address the problem of rules differently, the production rules, e.g., in POSTGRES, being eventually an implementation of our corresponding inheritance expressions. Next, the proposals mentioned above work out details of methods' management beyond our aims (see however [12] for more on methods' as **EXE** attributes). With respect to our scope that is only the inheritance, the following comparison appears.

All these proposals lead to nested relations, through explicit use of ADTs such as **TUPLE** in ISQL, or **SET** in Postgres. We stick to flat relations for simplicity. Also, the POSTGRES and ISQL follow the traditional distinction between real tables and views, through **DEFINE VIEW** and rules in Postgres, or the **WHERE** clause in ISQL. No other proposal allows for the naming of OID attributes, while we believe it is highly useful to do so. Furthermore, all the proposals use only the static downward inheritance for queries. They have limitations on what cases of multiple inheritance are acceptable, which we avoid through role names (space limits prohibit a detailed discussion).

We have fewer concepts in the data model, sharing with Postgres the aim of as few concepts as possible. Our single **SELECT** statement suffices for all types of queries, unifying the Postgres **EXECUTE** statement for

methods, and **RETRIEVE***, or, e.g., **RETRIEVE EMP*** variants [17]-[19]. Also, there are no terms in our model equivalent to the (confusing) **LIKE** of SQL3, or **DEFINE VIEW** of Postgres, or **SPECIALIZE, GENERALIZE, DISJOINT**, or **COVERING** of ISQL. Our **CREATE** statement implicitly covers all the corresponding possibilities. None of these proposals allows for general overriding, recursive inheritance, multidatabase inheritance [12], especially of methods, and the '..' notation or equivalent. Finally, none has the extensive possibilities our model offers for versioning.

5 Conclusion

Inherited attributes bring many advantages to relational databases. Relational schemas become more natural, and manipulations less procedural and more powerful, e.g., through simpler use of recursion. Intra-tuple value inheritance solves a class of view update problems and provides for the benefits of overriding and versioning. OO conceptual modeling with its well-appreciated advantages becomes available for relational databases, e.g., through the successive refinement of concepts.

Our OO concept has deep implications for the fundamentals of relational design. The concepts of a base table and a view are no longer fundamental. The basic possibilities that IAs bring can be implemented within any existing RDBS, at almost no additional storage and CPU cost, reusing the existing relations as canonical implementations. The relational design principles should be revisited, especially when they concern the ideas of a base table, or of a view, of logical navigation, of referential integrity, of normalization. Likewise, other approaches to extending the relational model, like nested relations, production rules, Datalog, or higher-order languages [10] are worthy of a revisit.

The capabilities of RDBSs with respect to inheritance and non-procedurality of manipulations appear more extensive than those of current OODBSs, and those capabilities should be further investigated. The main direction for further investigation should be a more predicative definition of static and of dynamic inheritance, e.g., further defining some kind of undirected inheritance.

Future work should also address implementation issues. They appear relatively straightforward for the basic cases, and clearly interesting for the general inheritance calculus, e.g., for recursive tables, and when the result is a union of relations or a multirelation, such as on the basis of [8]. Finally, visibility of the inheritance

paths in our relational schemas should be useful for query compilers in general, e.g., to better prepare for joins and recursive queries.

ACKNOWLEDGEMENTS:

We thank the members of Pegasus project team, and of Database Technology Department of HP Labs for fruitful discussions.

References

[1] Byung, S. L., Litwin, W., Wiederhold, G. Implicit Joins in the Structural Data Model. IEEE COMPSAC, (Sep. 1991).

[2] Chan, A., et al. Intelligent OO Features in Distributed Databases. IEEE COMPCON, (Feb. 1991), 504-509.

[3] Codd, E. F. A Relational Model of Data for Large Shared Data Banks. CACM, 13; 6, (June 1970). Also in Readings in Databases, (M. Stonebraker, ed.), Morgan-Kaufmann, 1988.

[4] Codd, E. F. Extending the Database Relational Model to Capture More Meaning. ACM TODS, 4, (Dec. 1979).

[5] Date, C. J. An Introduction to Relational Database Systems. Addison-Wesley, 1990.

[6] El-Masri, R. On the design, use and integration of data models. Ph. Dissertation. Comp. Sc. Dep. Stanford Univ, (May 1980).

[7] Fishman, D., et al. IRIS: An Object-Oriented Database System. Readings in Object-Oriented Database Systems (Zdonik and Maier, ed.), Morgan-Kaufmann, 1990.

[8] Grant, J., Litwin, W., Roussopoulos, N., and Sellis, T. An Algebra and Calculus for Relational Multidatabase Systems. 1st Int. Workshop on Interoperability in Multidatabase Systems. IEEE-Press 1991, 118-124. Extended version to appear in VLDB Journal.

[9] Korth, H., Silberschatz, A. Database System Concepts. McGraw Hill. 1991.

[10] Krishnamurthy, R., Litwin, W., Kent, W. Language Features for Interoperability of Databases with Schematic Discrepancies. ACM-SIGMOD 1991, 40-49.

[11] Litwin, W. Implicit Joins in the Multidatabase System MRDSM. IEEE Compsac, 1985, 495-504.

[12] Litwin, W., Ketabchi, M., Risch, T. Relations with Inherited Attributes. Techn. Rep. HPL-DTD-91-25, Hewlett-Packard Labs, Palo Alto, (Nov. 1991), 31.

[13] Loomis, M., Shah, A., Rumbaugh, J. An Object Modeling Technique for Conceptual Design, European Conf. on OO Programming, AFCET, (June 1987).

[14] Melton, J. ISO/ANSI Database Language SQL3 (working draft). ANSI X3H2-91-254, (Sept. 1991).

[15] Nierstrasz, O. A Survey of Object-Oriented Concepts, in Object-Oriented Concepts, Databases, and Applications (Kim, W., ed.), ACM Press, 1989, 3-22.

[16] Rosenthal, A., et al. Traversal Recursion: A Practical Approach to Supporting Recursive Applications. ACM-SIGMOD, 1986, 166-176.

[17] Stonebraker, M. Implementation of Integrity Constraints and Views by Query Modification. ACM-SIGMOD, (May 1975)

[18] Stonebraker, M., Rowe, A. L. The Design of POSTGRES. ACM-SIGMOD 1986, 340-355.

[19] Stonebraker, M. The Interaction between SQL and Object-oriented Databases. Inv. presentation. Database World, (July 1990).

[20] Ullman, J. Principles of Databases and Knowledge-Base Systems. Computer Sc. Press, 1988.

A Inheritance Calculus Algorithm

The description that follows is rather informal, for easier comprehension. The algorithm follows conceptually simple, although embedded, steps. Q denotes the query and D the database graph.

Step 1. Qualification of multiple identifiers, if any. For each unqualified attribute named A in the query, choose from the database schema only and all attributes A that are not downward inherited from an attribute also named A. Let Q1 be the resulting query. If multiple qualifications result, produce a Q1 for each choice.

Step 2. Calculus of inheritance for attributes not in schemas.

a. For each Q1, and each T.A that is not in T's schema, search for any IP from any T'.A that is IDI by T. For any such IP, produce a subquery, where T.A is replaced with T.A' inheriting from T'.A through the IP.

b. for each subquery, for any remaining T.A search for any node T'.A that is upward inherited by T. Replace T.A with T'.A and add the IP to the subquery. Produce such a subquery for any T'.A and any IP found.

c. for each subquery, for any remaining T.A search for any node T'.A that is indirectly inherited by T. Replace T.A with T'.A and add the IP to the subquery. Produce such a subquery for any T'.A and any IP found.

Step 3. Removal of schema inherited attributes. Let Q2 denote a subquery resulting from Step 2. For each Q2, produce a query Q3 with only base attributes, including the calculus of the override. Retain each complete Q3.

Step 4. Completion of a (sub)query, if needed. If no complete Q3 resulted from (3), then for each Q3, find paths in D that when added to Q3 connects it without new cycles. The priority order for choosing a path between T1 and T2 in Q3, is (i) any directed shortest IP between T1 and T2, (ii) any shortest IP between T1 and T2. If multiple choices result, produce a subquery Q4 for each different choice, adding the IP to a **WHERE** clause. Retain only and all complete Q4s.

Step 5. Final query. If only one Q4 results, the final result R is this query. If there are several, then R is the set of these subqueries, except that all subqueries with the same target lists are joined by union.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Screw	Blue	17
P4	Screw	Red	14
P5	Cam	Blue	12
P6	Cog	Red	19

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Fig 1 The S-P database

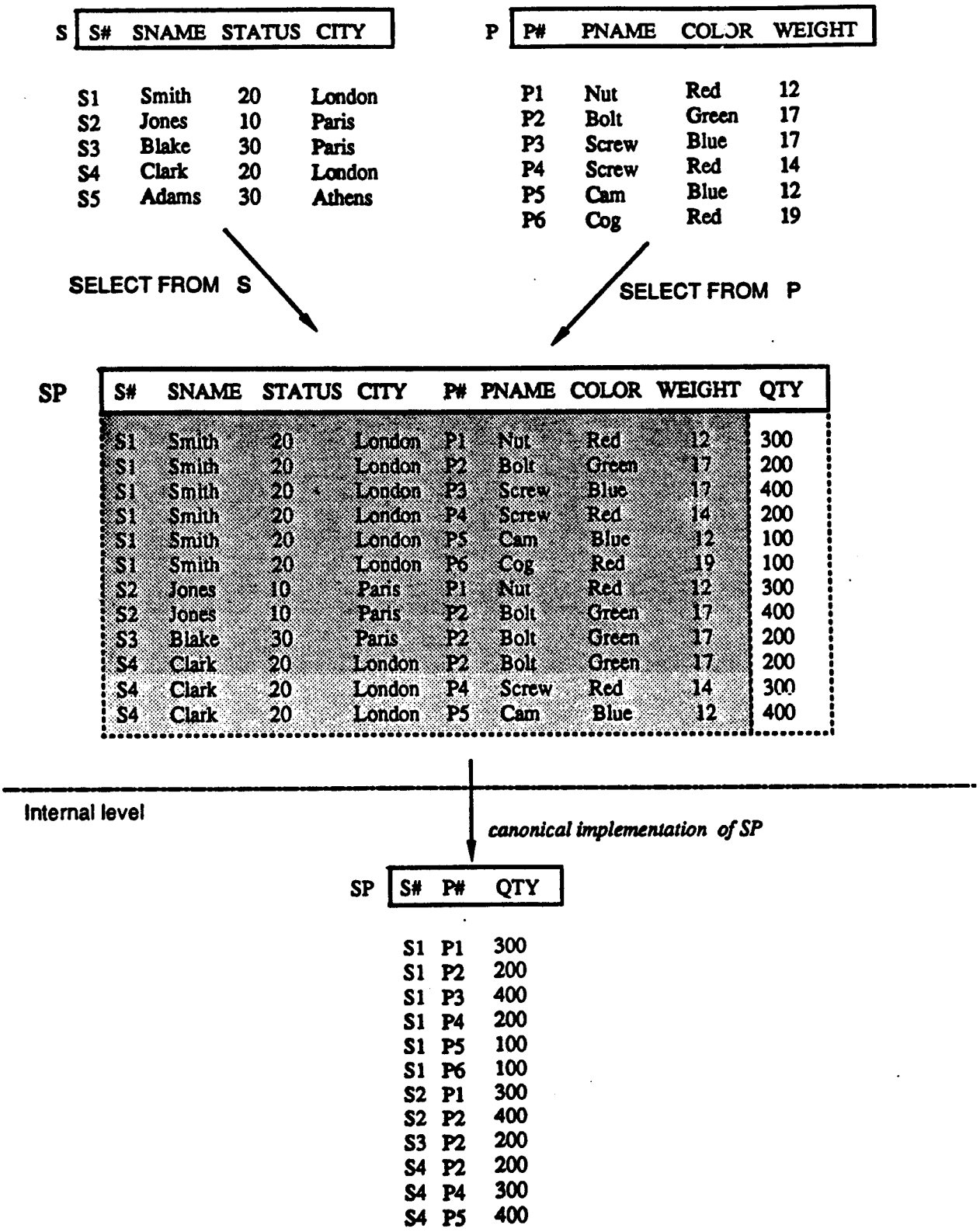


Fig 2 The S-P database with inheritance

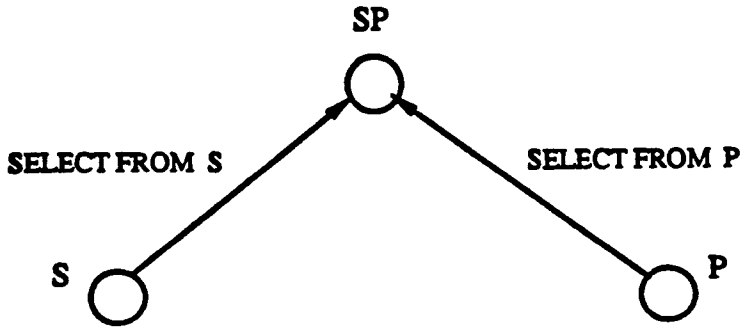


Fig 3 S-P database graph

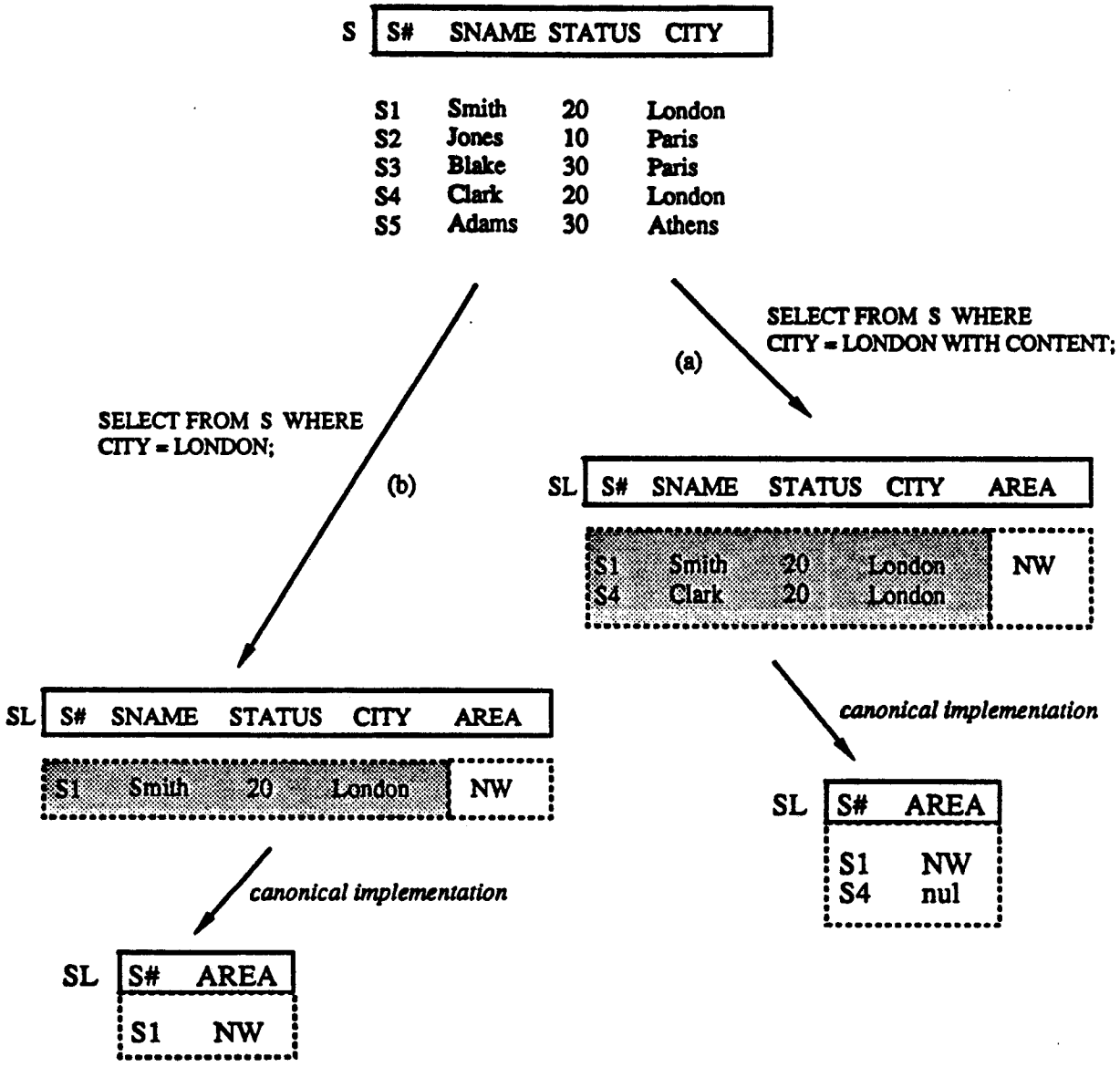
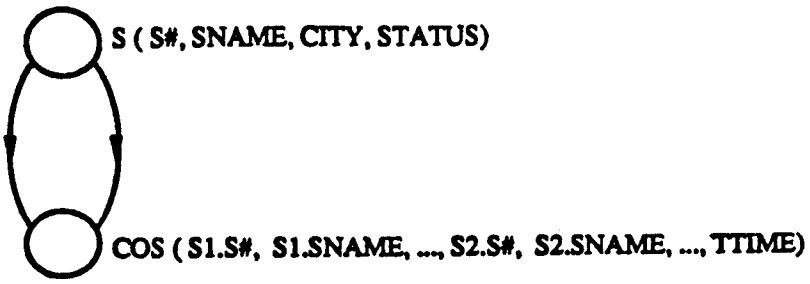
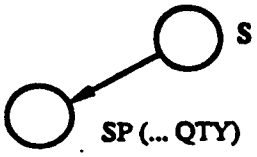


Fig 4 Inheritance from S:

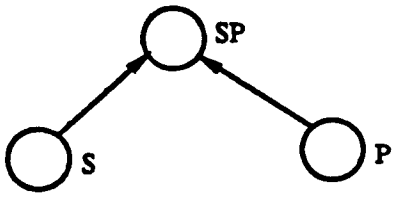
(a) with WITH CONTENT
(b) without this option



Example query: SELECT SNAME FROM COS WHERE TTIME < 10;



Example query: SELECT S.QTY FROM S WHERE SNAME = 'PH';



Example queries:

SELECT COLOR QTY FROM S WHERE SNAME = 'PH';
SELECT COLOR QTY WHERE SNAME = 'PH';

Fig 5 Types of inheritance in a relational database
(a) indirect downward inheritance
(b) upward inheritance
(c) undirected inheritance

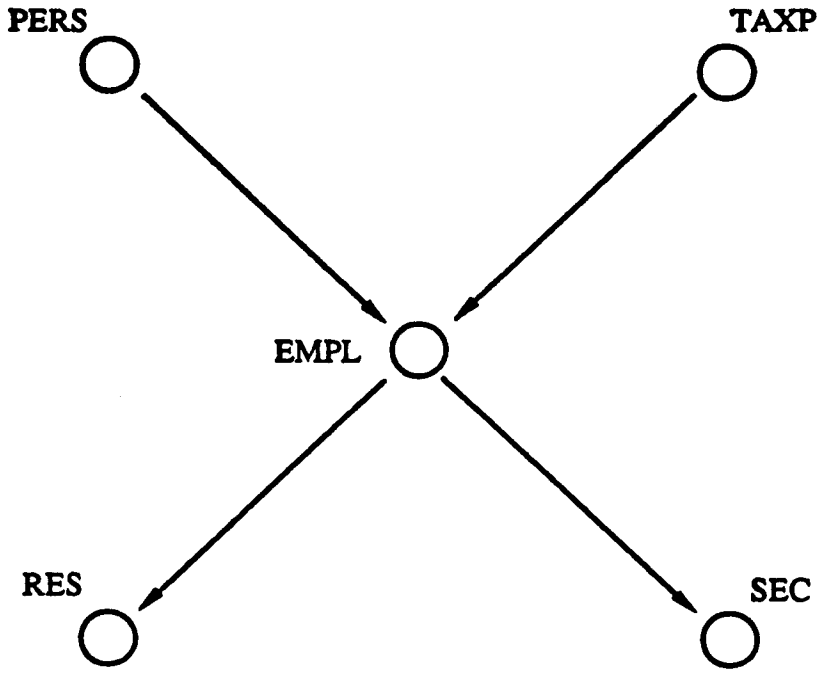
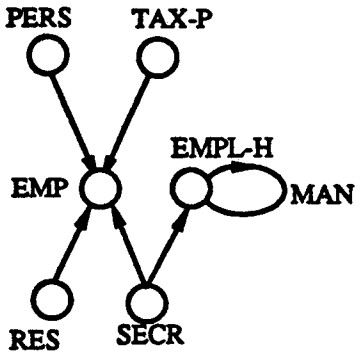


Fig 6 PH database graph



PID#	.....	BUD	TBUD
1	lito	2	2
3	shin	7	12
2	rore	3	3
4	kavi	5	5
5	haka	5	5
7	nari	7	17
9	fan	15	44

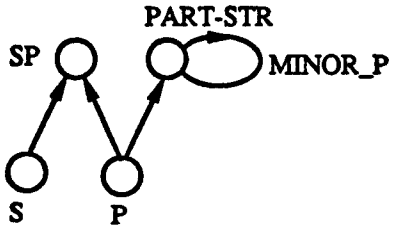
MAN

PID#	.....	BUD	TBUD
3	shin	7	12
9	fan	15	44
3	shin	7	12
7	nari	7	17
7	nari	7	17
9	fan	15	44

MAN.MAN

PID#	.....	BUD	TBUD
9	fan	15	44
9	fan	15	44
9	fan	15	44

EMP-H table with recursively computed budgets, for the hierarchy of employees problem



P#	....	QTY
P1	....	10
P3	.....	20
P5	.....	0
P2	.....	10
P6	.....	5
P7	.....	3
P8	.....	0

MINOR_P.

P#	....	QTY
P3	....	200
P5	....	0
P6	....	50
P7	....	15
P8	....	0

P#	....	QTY
P5	....	0
P7	....	150
P8	....	0

P#	....	QTY
P8	....	0

PART-STR table with recursively computed quantities, for the parts explosion problem

attributes inherited from other tables and base attributes

recursively inherited attributes

X .. Y columns of canonical implementation

Fig 7 Tables with recursive inheritance