






An SQL^o Front-End for Non-monotonic Inheritance and De-referencing

Joel Oduro-Afriyie  and Hasan M. Jamil  

Department of Computer Science, University of Idaho, Moscow, ID, USA
odur8117@vandals.uidaho.edu, jamil@uidaho.edu

Abstract. We revisit the issues of non-monotonic inheritance and structure traversal in object-relational databases with new insights to propose OO extensions of SQL and demonstrate that they are sufficient and powerful enough for modeling classes, non-monotonic inheritance and de-referencing. In particular, we show that simple tweaking of SQL with *tuple ID* helps capture these OO features cleanly and empowers application developers with a powerful knowledge modeling tool.

Keywords: Object-oriented modeling · Abstract relations · Object-relational query language · Translational semantics · Inheritance and overriding

1 Introduction

Numerous applications can benefit from the simple software engineering idea of inheritance and overriding. Despite significant interests in modeling these convenient features in database query languages, a fully functional object-oriented (OO) [2] or object-relational (OR) database [4] did not materialize mainly because it was extremely difficult to combine the simplicity and declarativity of SQL-like languages with the power of full object-orientation in a single platform. Serious efforts to craft an OO SQL date back to early to late 90s [5, 7], and no similar efforts can be seen since then. Even in those early efforts, researchers were mainly focused on supporting abstract data types (ADTs). The community then was eager to find a query language that looks and feels like C++. Not surprisingly, the CQL++ [3], or SQL/XNF [7] type database languages basically attest to the reality, although a limited number of research focused on features such as inheritance without much success [5]. The OQL [1] or O₂ [2] languages are complex to say the least, and this explains why they did not become popular.

In this paper, we propose a novel approach to class hierarchy and inheritance modeling with overriding, and object de-referencing, in classical relational database systems without the need for a new algebra based on the conviction that minimally extending SQL to support the urgently needed OO features is prudent. In the remainder of the paper, we mainly use an illustrative example to expose the modeling and query mapping technique we propose without much

details for the sake of brevity and for expository purposes. A complete technical discussion on the model, language and query transformation is deferred to a full article we plan to publish elsewhere.

2 The OR Model

The *object relational model*, or the OR data model, we propose, has two types of tables – *traditional* tables (called simply tables) and *abstract* tables. Tables are defined in standard ways using `create table` statements. For example, the instance *Professors* in Fig. 1(a) is declared by the statement

```

c1: create table Professors (
    PiD tupleID(3) primary key,
    Name string(10),
    Rank string(10)
    Dept string(10));
q1: select *
    from Professors;
    
```

Professors is a first normal form traditional table declaration, and thus all standard SQL statements can be used on it and query q_1 above returns the entire table in Fig. 1(a). In this statement, `tupleID` is a special string data type discussed in the context of objects and classes below in more detail.

PiD	Name	Rank	Dept
p-1	Sharon	Assoc	CS
p-2	Pierre	Full	⊥
p-3	Tanaka	⊥	Econ
p-4	Alfredo	Full	CS

(a) Table *Professors*

PiD	Dept	Salary
p-1	e-7	110K
p-4	e-7	⊥
p-3	e-8	105K

(b) Table *Works*

TiD	Name	State
t-a	⊥	DC
t-b	Pria	⊥
t-c	Aphrodite	TX

(c) Abstract table *People*

TiD	Name	State
t-a	⊥	⊥
t-b	Pria	DC
t-c	Aphrodite	TX

(d) View of table *People*

TiD	Name	State	SiD	Par
n-a	⊥	ID	s-1	t-4
n-b	Clint	⊥	s-2	t-5
n-c	Moira	TX	s-3	t-3
n-d	Alex	PA	s-4	⊥

(e) Abstract table *Students*

TiD	Name	State	SiD	Par
n-a	⊥	⊥	⊥	⊥
n-b	Clint	ID	s-2	t-5
n-c	Moira	TX	s-3	t-3
n-d	Alex	PA	s-4	⊥

(f) View of table *Students*

TiD	DiD	Name	Chair
e-6	d-0	CS	p-1
e-7	d-1	⊥	p-4
e-8	d-2	Math	p-1
e-9	d-3	Econ	p-3

(g) Abstract table *Departments*

TiD	Name	State	SSN	Income
t-2	⊥	⊥	000	45K
t-3	Joe	WA	001	⊥
t-4	⊥	OH	014	90k
t-5	Maria	⊥	207	⊥

(h) Abstract table *Parents*

TiD	Name	State	SSN	Income
t-2	⊥	⊥	⊥	⊥
t-3	Joe	WA	001	45K
t-4	⊥	OH	014	90k
t-5	Maria	DC	207	45K

(i) View of table *Parents*

TiD	DiD	Name	Chair
e-6	⊥	⊥	⊥
e-7	d-1	CS	p-4
e-8	d-2	Math	p-1
e-9	d-3	Econ	p-3

(j) View of table *Departments*

TiD	Name	State	SiD	Par	Major
u-e	⊥	⊥	s-5	⊥	⊥
u-f	Ovro	MI	s-6	⊥	e-7
u-g	Abebi	ID	s-7	t-5	e-9
u-h	Odelia	⊥	s-8	t-4	e-8

(k) Abstract table *UnderGrads*

TiD	Name	State	SiD	Par	Major
u-e	⊥	⊥	⊥	⊥	⊥
u-f	Ovro	MI	s-6	t-4	e-7
u-g	Abebi	ID	s-7	t-5	e-9
u-h	Odelia	ID	s-8	t-4	e-8

(l) View of table *UnderGrads*

Name	CName
Sharon	Alfredo

(m) Aggregation query

Fig. 1. OR model tables: traditional and abstract relations in class hierarchy.

Objects, Classes and Instances. In contrast, the abstract table *People* models a class object of type *People* and a set of instance objects of the same type through the create abstract table declaration below.

```

c2: create abstract table People (
    TiD tupleID(3) auto,
    Name string(10),
    State string(2),
    default values ((⊥, ⊥, "DC"));
                                q2: select *
                                    from People;

```

This create abstract table statement specifies an extended first normal form table under the scheme *People* (*TiD*, *Name*, *State*) with several unique properties. First, the scheme includes a distinguished attribute named *TiD*. This attribute represents a domain of unique object IDs mandated by the concepts of OO database models. In our model, all objects have an immutable ID, called the *OID*, and these IDs in OR model are synonymous to the concept of tuple IDs (denoted *TiDs*) first introduced by Sieg and Sciore [8]. These tuple IDs can be created in several ways. The keywords `tupleID(3) auto` states that *TiD* is an automatically generated string type object ID of length three. In OR model, `tupleID` has a string domain that can have system generated values. Thus, it requires a type declarations and optionally a method for generating it (e.g., `auto`). In contrast, the declaration

TiD tupleID compose(string(2)+"-" +integer(2))

says the tuple ID is a five character long string supplied by the user which has the format first two characters, followed by a hyphen and then finally has a two digit integer, resulting in a five character unique tuple ID. In this case too, database wise uniqueness is preserved. Furthermore, since these IDs in *TiD* columns are unique database wide in all abstract and traditional tables, they are candidate keys by default. We call them *object keys*. However, *tupleID*, *auto* and *compose* features can be used to type any attribute. But the uniqueness is enforced only for the distinguished attribute *TiD* in ways consistent with the TiD algebra [8].

Figure 1(c) shows an instance of the abstract table *People*. In our model, all abstract tables have the column *TiD* (but unlike TiD algebra, not all tables have *TiD* columns), and thus all tuples in every abstract table have a unique object ID. Observe also that the instance has two partitions. In the top partition, we have the tuple $\langle t-a, \perp, DC \rangle$, and in the bottom partition we have tuples $\{\langle t-b, Pria, \perp \rangle, \langle t-c, Aphrodite, TX \rangle\}$. The lone tuple with *TiD* *t-a* in the top partition is the default value of the class object *People* as stated in the `default values` clause in the create abstract table statement. In this tuple, the first \perp corresponding to the *TiD* column is replaced by the system generated object ID *t-a*. This tuple contains the class default values for each column, e.g., *State* has default class value *DC*, but *Name* does not. Finally, the bottom partition contains the instance objects, each of which also has an object ID, e.g., *t-b* and *t-c*.

Inheritance and Overriding. The consequence of having a class default value is interesting and far reaching. For example, the query q_2 above now returns the abstract table “view” in Fig. 1(d). We make several important observations. First, this table does not have a class default value tuple, i.e., all the values are null (\perp) because we have closed the inheritance and the default values are no longer useful. Also note that tuple $t-b$ inherited the default *State* value *DC* and replaced the null value. However, since the tuple $t-c$ already has a local value *TX*, it overrode the value *DC* and not inherited. This is in the spirit of dynamic inheritance with overriding in OO systems, called *non-monotonic inheritance*.

Relationships and Aggregation. Being a superset of the relational data model and SQL, the OR model and its query language SQL^O supports relationships by respecting foreign keys. In the `create table` declaration below, the `references` clause declares a foreign key in *Works* that references the primary key of *Departments*, indicating *Dept* can accept null values. In contrast, the `aggregates` clause (in the sense of SDM [6]), though similar to `references`, cannot accept null values. Here too, the *PiD* column references a column in another table, but not necessarily a primary key. Instead, it is an OID or tuple ID column. Note that *PiD* is not a distinguished column name though it has the tuple ID domain. Thus uniqueness is not maintained by default, but declaring it the primary key enforces uniqueness in traditional sense, not in OO sense.

```
c3: create table Works (
    PiD tupleID(3) primary key references Professors(PiD),
    Dept tupleID(3) aggregates Departments(TiD),
    Salary integer(7));
```

The instance table in Fig. 1(b) over the scheme *Works*(*PiD*, *Dept*, *Salary*) is essentially a relationship between *Professors* and *Department* in ER sense. The fundamental difference between `aggregates` and `references` is that the objects in the former referenced tables need not be explicitly joined to access their columns as is the case for latter reference types. The query below clarifies this distinction.

```
q3: select P1 →Name, Chair→Name as CName
    from Works W1, Professors P1, Professors P2
    where W1.PiD = P1.PiD and Salary > 109K and W1.Dept→Chair =
    P2.PiD and W1.Dept→Name = P2.Dept;
```

This query returns names of all professors and their chair’s who earn more than \$109K with their chair also from the same department. This query will return the table in Fig. 1(m). Had the *Professors* table been declared as an abstract table, we could have written this query in a much simpler way using OO de-referencing features. Also note that the *Department* table is not referenced in the `where` clause yet became accessible via de-referencing.

Class Hierarchies. Similar to classes in OO systems, abstract tables can be organized in table hierarchies. While classes or abstract tables¹ can have multiple subclasses, they can only have unique superclasses. Subclasses in OR model inherit properties and their default values, and all key and other integrity constraints, from their superclasses. While integrity constraints and the scheme of a class are inherited monotonically, their class default values are inherited non-monotonically in an overriding fashion based on specificity preference principle.

For example, consider an instance object $s-1$ in *Students* class in Fig. 1(e), where *Students* is a subclass of *People* in Fig. 1(c). The following create abstract table statement defines the subclass relationship between these two tables.

```
c4: create abstract table Students inherits People (
  SiD string(3) primary key,
  Par tupleID(3) aggregates Parents,
  default values (( $\perp$ ,  $\perp$ , "ID", "s-0",  $\perp$ ));
```

Being a subclass of *People*, not only does *Students* inherit the scheme of *People* and the object key, it also introduces two new attributes $\{SiD, Parent\}$, a new primary key *SiD*, and a new default value *ID* for the inherited attribute *State*. In this case, all instances of *Students* (as well as all its subclasses) will inherit, when appropriate, the default value *ID* for *State*, and not *DC* since the local or specific value *ID* at *Students* overrides the inherited value for *State* in *People*.

Null Closure. In a select query, the relation list in the from clause can be both traditional and abstract tables. Since abstract tables can be subclass of another class table, a long chain of inheritance becomes complicated. Each abstract table has the potential to have inherited values from superclasses at arbitrary height. Since updates in all tables are allowed, a static inheritance of all default values to lower subclasses and instances is not a prudent choice though the approach could make query processing substantially cheaper. But updates in class default values have the potential to invalidate statically inherited values before the update and leave the recovery from the state of erroneous inheritance at jeopardy. We use a process called *null closure* to dynamically inherit the class default values down to all subclasses and instances in an overriding manner.

3 Mapping SQL^O to SQL

Implementation of the SQL^O language is based on a translational semantics of SQL^O programs to SQL, so that we can understand the semantics in terms of the well known meaning of SQL, and obviate the need for a native SQL^O implementation, saving effort and cost. The correctness of SQL^O is then established based on the soundness and completeness properties of SQL relative to the OR data model and its intended semantics. We argue that SQL^O is sound and complete

¹ In this article, we use the terms sub and superclasses interchangeably with sub and supertables for convenience.

too by showing that the translation outlined preserves the intended semantics of SQL^O. In the following sections, we only discuss translation of the SQL^O specific statements not available in SQL by way of examples.

3.1 Creating Class Tables

The *People* class table declaration in Sect. 2 is translated as follows. We create two separate tables in SQL for each `create abstract table` statement to implement class and instance objects in two partitions. The class tables are annotated with subscript *c* and instance tables with *i* as follows.

```

c5: create table Peoplec (
    TiD varchar(3) auto unique,
    Name varchar(10),
    State varchar(2) );

c6: create table Peoplei (
    TiD varchar(3) auto unique,
    Name varchar(10),
    State varchar(2) );

u1: insert into Peoplec(TiD, Name, State)
    values ($AutoKey, NULL, 'DC');
```

In the above statements `auto` is a directive to create a random key that will never be assigned to another *TiD* column of any tuple. Major database systems like Oracle support similar unique primary key generation. In the insert statement we use the `$AutoKey` keyword to call a function to generate the OID or the tuple ID, and insert this tuple into *People*_c as the class default value. The `unique` declaration makes *TiD* a candidate key, but not the primary key of the table. The uniqueness of *TiD* is ensured by checking a unary system table called *UniqueKeys* we maintain which logs all *TiD* values ever assigned and in use in our databases. Note that the statement *u*₁ above, implements the semantics of the default values declaration in statement *c*₂ in Sect. 2.

The subclass table *Students* in Sect. 2 is accomplished by creating the SQL statements below. Note that for aggregation, we required that the *Parent* cannot have null values, and the referenced *Parents* object cannot be deleted without deleting the *Students* object.

```

c7: create table Studentsc (
    TiD varchar(3) unique,
    Name varchar(10),
    State varchar(2),
    SiD varchar(3) primary key,
    Par varchar(3) not null
        foreign key references Parentsi (TiD)
        on update cascade
        on delete restrict);

c8: create table Studentsi (
    TiD varchar(3) unique,
    Name varchar(10),
    State varchar(2),
    SiD varchar(3) primary key,
    Par varchar(3) not null foreign key
        references Parentsi (TiD)
        on update cascade
        on delete restrict );

u2: insert into Studentsc(TiD, Name, State,
    SiD, Par)
    values ($AutoKey, NULL, 'ID', 's-0', NULL);

u3: insert into ClassHierarchy(SubClass,
    SuperClass)
    values ('Students', 'People');
```

We do not separately discuss the statements such as `insert`, `delete` and `update`, which can be handled trivially. Finally, we enter the subtable relationship specified in the `inherits` keyword into the system table *ClassHierarchy* as a pair

$\langle \text{'Students'}, \text{'People'} \rangle$ to be able to create the class hierarchy for null closure discussed next. The `inherits` keyword also prompts the inclusion of the attributes in the superclass *People* into the current table *Students*.

3.2 Computing Null Closure and Table View

Prior to processing queries, we first process null closure discussed in Sect. 2 for all directly or implicitly referred abstract tables to ground the tables with inherited values in real time. On analysis of the query in terms of the tables included in the `from` clauses, and the cross referencing of the de-reference operators with the schemes, a list of abstract tables is created that potentially warrant null closures. A precedence graph of subclass-superclass relationship for each of these tables is constructed using the *ClassHierarchy* system table and for every table, a maximal scheme is created to list the attributes that all clauses will need. We then proceed to create two sets of views – one for the class tables and one for the instance tables, and we then use only the views corresponding to each instance table in the rewritten queries as follows.

Let us explain the process of using the query that asks *list the names of all undergraduate non computer science majors resident in Idaho and their parents' income such that their parents earn more than \$75K and their department chairs are computer science professors*. This query can be posed in SQL^O as the following expression.

```
q4: select Name, Par→Income
      from UnderGrads, Professors
      where State = 'ID' and Par→Income > 75K and Major→Name ≠ 'CS'
            and Major→Chair = PiD and Dept = 'CS';
```

This query assumes that the following DDL statement has already been defined.

```
c10: create abstract table UnderGrads inherits Students (
      Major tupleID(3) aggregates Departments(TiD));
```

In this query three abstract tables *UnderGrads*, *Departments* and *Parents*, and a traditional table *Professors* are involved. This information is derived from the database schema definitions, i.e., *Major* in *UnderGrads* aggregates *Departments* where student majors are found. Similarly, *Parent* aggregates *Parents* where their *Income* is listed. The de-reference operators in the query actually give away this information. Finally, *Chair* in *Departments* aggregates *Professors* where we find their department. While *UnderGrads* and *Persons* participate in a class hierarchy and require null closure as shown below, *Departments* does not.

```
c11: create view Peoplecv(TiD, Name, State) as
      select TiD, Name, State
      from Peoplec;
```

```
c12: create view Studentscv(TiD, Name, State, SiD, Par) as
      select V.TiD,
```

```

    case when V.Name=NULL then U.Name else V.Name,
    case when V.State=NULL then U.State else V.State,
    case when V.Par=NULL then U.Par else V.Par
from Peoplecv as U, Studentsc as V;

```

```

c13: create view UnderGradscv (TiD, Name, State, Par, Major) as
select V.TiD,
    case when V.Name=NULL then U.Name else V.Name,
    case when V.State=NULL then U.State else V.State,
    case when V.Par=NULL then U.Par else V.Par,
    case when V.Major=NULL then U.Major else V.Major
from Studentscv as U, UnderGradsc as V;

```

```

c14: create view Parentscv (TiD, Income) as
select V.TiD,
    case when V.Income=NULL then U.Income else V.Income,
from Peoplecv as U, Parentsc as V;

```

The above statements only close the nulls in class tables. To truly inherit the default values, we now close the inheritance in all three instance tables as follows.

```

c15: create view UnderGradsiv (TiD, Name, State, Par, Major) as
select V.TiD,
    case when V.Name=NULL then U.Name else V.Name,
    case when V.State=NULL then U.State else V.State,
    case when V.Par=NULL then U.Par else V.Par,
    case when V.Major=NULL then U.Major else V.Major
from UnderGradscv as U, UnderGradsi as V;

```

```

c16: create view Parentsiv (TiD, Income) as
select V.TiD,
    case when V.Income=NULL then U.Income else V.Income,
from Parentscv as U, Parentsi as V;

```

```

c17: create view Departmentsiv (TiD, Name, Chair) as
select V.TiD,
    case when V.Name=NULL then U.Name else V.Name,
    case when V.Chair=NULL then U.Chair else V.Chair
from Departmentsc as U, Departmentsi as V;

```

The script above completes the steps for computing the null closures and generates three view tables for our query – i.e., *UnderGrads_{iv}*, *Parents_{iv}* and *Departments_{iv}*.

3.3 Inheritance and Object Traversal in SQL Using Query Rewriting

As a final step, we rewrite the SQL^o query in Sect. 3.2 as a large join query to accommodate object traversals anticipated by the de-reference operators over the three null closed instance views we have generated and the traditional table:


```

q6: select U.Name, V.Income
from UnderGradsiv as U, Parentsiv as V, Departmentsiv as W,
Professors as X
where U.State = 'ID' and U.Par=V.TiD and V.Income > 75K
and U.Major=W.TiD and W.Name ≠ 'CS' and W.Chair=X.PiD
and X.Dept = 'CS';

```

In our example database, there are two potential Idaho resident undergraduate students, *Abebi* and *Odelia*. However, *Abebi*'s parent *Maria*'s income is less than \$75K, and her department chair *Tanaka* is not a computer science professor, and thus does not qualify to be in our response. However, *Odelia* is a Math major, and her department chair *Sharon* is a computer science professor and her parent also has income higher than \$75K although the parent name is missing. So, SQL^O appropriately returns the tuple $\langle Odelia, 90K \rangle$ as a response.

4 Conclusion

Our goal in this paper was to show that complex objects, class hierarchies, inheritance, overriding and structure traversal can be modeled as a simple extension of SQL. While we did not discuss a complete translation algorithm for brevity, we have presented the overall idea behind the translation of an SQL^O database and queries to a semantically equivalent SQL database. We have shown that the two most coveted OO features, namely inheritance with overriding and object traversal, can be captured within relational model based on a translational semantics without the need for an entirely new language or a formal foundation.

References

1. Alashqur, A.M., Su, S.Y.W., Lam, H.: OQL: A query language for manipulating object-oriented databases. In: VLDB, pp. 433–442 (1989)
2. Bancilhon, F., Delobel, C., Kanellakis, P.C. (eds.): Building an Object-Oriented Database System, The Story of O2. Morgan Kaufmann, Burlington (1992)
3. Dar, S., Gehani, N.H., Jagadish, H.V.: CQL++: a SQL for the Ode object-oriented DBMS. In: Pirotte, A., Delobel, C., Gottlob, G. (eds.) EDBT 1992. LNCS, vol. 580, pp. 201–216. Springer, Heidelberg (1992). <https://doi.org/10.1007/BFb0032432>
4. Feuerlicht, G., Pokorný, J., Richta, K.: Object-relational database design: can your application benefit from SQL: 2003? In: Barry, C., Lang, M., Wojtkowski, W., Conboy, K., Wojtkowski, G. (eds.) ISD, Challenges in Practice, Theory, and Education, vol. 2, pp. 975–987. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-78578-3_30
5. Fuh, Y., et al.: Implementation of SQL3 structured types with inheritance and value substitutability. In: VLDB, pp. 565–574 (1999)
6. Hammer, M., McLeod, D.: Database description with SDM: a semantic database model. ACM Trans. Database Syst. **6**(3), 351–386 (1981)
7. Mitschang, B., Pirahesh, H., Pistor, P., Lindsay, B.G., Südkamp, N.: SQL/XNF - processing composite objects as abstractions over relational data. In: ICDE, pp. 272–282 (1993)
8. Sieg Jr., J., Sciore, E.: Extended relations. In: ICDE, pp. 488–494 (1990)