Proceedings of the 5th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases, Madrid, Spain, February 15-17, 2006 (pp360-365)

Extended Principle of Orthogonal Database Design

ERKI EESSAAR
Department of Informatics
Tallinn University of Technology
Raja 15,12618 Tallinn
ESTONIA
http://staff.ttu.ee/~eessaar

Abstract: - One of the main new features in the Object-Relational Database Management Systems (ORDBMS) is possibility to define new data types. It increases the amount of design options as well as possibilities to make bad design decisions. For example, entity type in a conceptual data model can be implemented as a relation variable (relvar for short) or in some cases as an attribute of a relvar that has a scalar-, tuple- or relation type. One guideline that helps to avoid bad design decisions is The Principle of Orthogonal Design [1]. It states that if a new tuple is added to a relational database, then there shouldn't be more than one base relvar which is suitable for recording it. Violation of this principle causes data redundancy. But original version of this principle doesn't take into account that relvars could have attributes with the generated relation- or tuple types or with the user-defined scalar types. The value of such an attribute that is part of the value of one relvar (relation) could contain the same data as value of some other relvar. Main contribution of this article is the proposal of an extended version of the orthogonal database design principle that takes into account possibility that relation-, tuple- or user-defined scalar types are used in a database. In addition we propose two heuristic rules that help to reduce data redundancy within one relvar that has an attribute with a relation- or tuple type.

Key-Words: - Relational data model, Object-relational databases, Database design, Data type, Relational algebra, SQL

1 Introduction

The set of the allowed data types in the relational data model has been and still is a source of discussion. Founder of the relational data model E. F Codd acknowledges possibility of nonsimple domains which permitted values are relations [2]. He argues for eliminating them by using a process of normalization. One reason is that they require more complicated data structures at the storage level than simple domains. Codd [2] suggests that relations that have attributes with the nonsimple domains and relations that are at the first or higher normal forms are mutually exclusive. Early SQL took the same approach and permitted usage of "simple" data types. It didn't allow to define new data types. For many years researchers have proposed to ease restrictions that are imposed to the relations by the first normal form. They have described "Non-First Normal Form" relations and relational algebra of such relations [3],[4]. These relations have relation-valued attributes or in other words they have relations that are nested inside of them. Researchers and developers have understood that limited support of the data types restricts usefulness of SQL in the complex applications. Therefore new type of DBMS has been proposed, namely Object-Relational DBMS (ORDBMS) which among other things permits creation of the new data types. These types can be used in a database. SQL has been extended in order to make it suitable to use in the ORDBMSs. SQL:1999 standard extends SQL language by allowing creation of user-defined types (UDTs) and type constructors for creating row-, array- and reference types [5]. SQL:2003 extends SQL further by specifying type constructor for constructing multiset collection types [6]. For example, this allows to create tables which attribute values are multisets of rows (nested tables).

More lately scientists have argued that if a relation has a relation valued attribute, then it doesn't mean that relation is not at the first normal form because by definition relation is at least at the first normal form (otherwise it is not a relation) [7]. We adopt the approach taken by Date and Darwen in the so called *Third Manifesto* [8, p. 21]: "The question as to what data types are supported is orthogonal to the question of support for the relational model."

Freedom of usage of the data types means that database designer has more design options but also more possibilities to come up with a bad design. For example, entity type in a conceptual data model could be implemented as a relation variable (relvar

for short) or in some cases as an attribute of a relvar that has a scalar type, a tuple type or a relation type. Soutou [9] presents simple conceptual data model with 6 entity types and 4 relationships. He offers 384 different solutions for implementing these relationships in an object-relational database by using collections and pointers. He doesn't take into account solutions that have multiple nesting levels.

We need guidelines that help to avoid bad design decisions. One relevant guideline is The Principle of Orthogonal Design (POD for short) [1]. POD states that if a new tuple is added to a database which design follows the POD, then there shouldn't be more than one base relvar which is suitable for recording this tuple. "The overall objective of orthogonal design is to reduce redundancy and thereby avoid update anomalies" [7, p. 398] If this principle is violated, then data about the same object could be recorded more that once using different relvars. But original version of this principle doesn't take into account that relvars could have attributes that have a relation type, a tuple type or a scalar type that has complex possible representations. Main goal of this article is to extend POD in order to take into account usage of the complex types in a database. Another goal is to present rules that guide correct usage of relation- or tuple valued attributes in the relvars.

In this article we use terminology of the relational data model that is used by Date and Darwen [8] and Date [7]. Firstly, the same terminology is used in the original version of POD [7] that we extend in this article. Secondly, relational model as described by Date and Darwen [8] can be seen as a kind of object-relational data model. Therefore this discussion is also relevant to the database designers who design object-relational databases for the systems that use SQL.

The rest of the paper is organized as follows. Section 2 contains an overview of the original version of POD. Section 3 presents examples of the problems that database designer could encounter by designing object-relational database. Section 4 proposes the extended version of POD and two additional heuristic rules that help to reduce data redundancy in a database. These are main contributions of our work. Discussion of the principle and rules together with the examples is in the section 5. Section 6 summarizes this article.

2 The Principle of Orthogonal Design

In this section we describe original version of the Principle of Orthogonal Design (POD).

Firstly, let's see an example. All employees receive salary. Some (but not all) employees receive bonuses. There are two base relvars *Emp* and *Emp_bonus* in the database:

Emp(empno, ename, sal, deptno) Key (empno);
Emp_bonus(empno, ename, bonus) Key (empno);

Attributes *empno* and *ename* in the relvar *Emp* have the same data types as attributes with the same name in the relvar *Emp_bonus*. Relvar *Emp* is used in order to record all employees. Relvar *Emp_bonus* is used in order to record bonuses of employees. Names of the employees who receive bonuses are recorded in two different relvars. It causes data duplication and doesn't seem reasonable. What is general guideline for avoiding such design?

POD presents guideline of the database design in a formal way: *"Let A and B be distinct base relvars. Then there must not exist nonloss decompositions of A and B into A1, A2, ..., Am and B1, B2, ..., Bn (respectively) such that some projection Ai in the set A1, A2, ..., Am and some projection Bj in the set B1, B2, ..., Bn have overlapping meanings."* [7, p. 397]

Nonloss decomposition means that the original relvar can be recreated by joining *all* the projections. Another assumption is that *all* considered projections are *needed* in order to recreate original relvar. For example, projection *Emp'* is not needed in order to restore relvar *Emp*:

Emp'(empno) Key (empno)
Emp''(empno, ename) Key (empno)
Emp'''(empno, sal, deptno) Key (empno)

What is an overlapping meaning of the relvars? The relvar predicate for relvar R "is the logical AND or conjunction of the constraints that apply to - in other words, mention relvar R." [7, p. 259]. Let's assume that *R1* and *R2* are two relvars, with associated relvar predicates *R1A* and *R1B*, respectively. The meanings of *R1* and *R2* are said to overlap if and only if it is possible to construct some tuple *t* so that *R1A(t)* and *R1B(t)* are both true [1]. In other words, if the relvars *R1* and *R2* have overlapping meanings, then tuple *t* could be part of the value of both these relvars.

One possible projection of relvar *Emp* is relvar *Emp''(empno, ename) Key (empno)*. One possible projection of relvar *Emp_bonus* is relvar *Emp_bonus'(empno, ename) Key (empno)*. Both these relvars have following predicate:

e.empno EMPNO_TYPE AND
e.ename ENAME_TYPE AND
(IF e.empno=f.empno THEN e.ename=f.ename)

Therefore these relvars have overlapping meanings and relvars *Emp* and *Emp_bonus* don't follow POD guideline.

3 Additional Examples of Possible Design Problems

In this section we give examples of the design problems that are possible if database designer can define new data types and use them in a database.

For example, there is a conceptual data model with the entity types *Emp* and *Contract* (see Fig. 1).

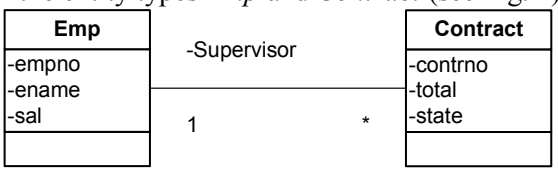

Fig. 1 Example of conceptual data model

Database designer has many options how to design database based on this model. For example:

1. Create distinct base relvars based on these entity types. Each atribute in the entity type has a corresponding attribute with a scalar type in a corresponding relvar. Relvar, that is created based on the entity type *Contract,* contains a set of foreign key attributes in order to allow recording associations between employees and contracts.
2. Create the relvar *Contract_RV* based on the entity type *Contract*. This relvar has the attribute *supervisor* with the tuple type *TUPLE{H}*. Each attribute in the entity type *Emp* has a corresponding attribute in the heading *H* of the tuple type.
3. Create the relvar *Contract_RV* based on the entity type *Contract*. This relvar has the attribute *supervisor* with the scalar type *EMP_TYPE*. Each attribute in the entity type *Emp* has a corresponding component of the possible representation in *EMP_TYPE*.
4. Create the relvar *Emp_RV* based on the entity type *Emp*. This relvar has the attribute *contracts* with the relation type *RELATION{H}*. Each attribute in the entity type *Contract* has a corresponding attribute in the heading *H* of the relation type.
5. Create the scalar type *CONTRACT_TYPE* based on the entity type *Contract*. Create the relvar *Emp_RV* based on the entity type *Emp*. This relvar has attribute *contracts* with the relation type *RELATION{H}*. Heading *H* of the relation type contains one attribute *contract* with the scalar type *CONTRACT_TYPE*.

Data about an employee could become duplicated within the value of one relvar in case of designs 2 and 3 if an employee is supervisor of more than one contract. In case of designs 4 and 5 we need constraints that enforce the rule that each contract has exactly one supervisor. Without that constraint we could register data about the same contract within the tuple of each employee and therefore state that contract has many supervisors.

Now let's assume that requirements to the database evolve. Database must be able to record orders, parties and their associations with other objects. Order is associated with exactly one employee who is supervisor of the order and must personally assure that it is fulfilled in time. Contract is associated with the party who is the client with whom the contract has been made (see Fig. 2).

One possible solution for recording orders in case of designs 2 and 3 is creation of the relvar *Order_RV* based on the entity type *Order*. This relvar has the attribute *supervisor* that has either scalar type or tuple type. In this case, data about an employee would be duplicated in the different relvars if employee is supervisor of some order as well as some contract.

One possible solution for recording parties in case of designs 4 and 5 is creation of the relvar *Party_RV* based on the entity type *Party*. This relvar has the attribute *contracts* that has a relation type. In this case, data about the contract would be duplicated in the values of different relvars.

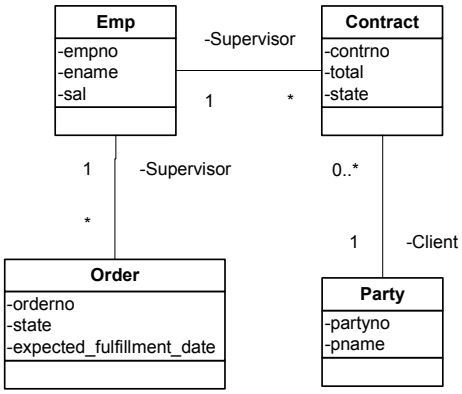

Fig. 2 Example of conceptual data model

4 Extended principle of Orthogonal Database Design

In this section we present the extended version of POD and two rules that help to prevent data redundancy within one relvar. All of them take into account the fact that database designer can use user-defined scalar types and generated tuple- and relation types. We describe assumptions that are used in the definition of principle and rules before presenting actual definitions.

Definition of principle, rules and following discussion contain statements that are written using

Tutorial D relational language. These statements have mostly been tested using the prototypical relational database management system *Rel* [10]. These statements have been tested with the built-in types and not with the user-defined types because *Rel* doesn't yet support user-defined types completely. Usage of THE_ operators has also not been tested because they are not yet implemented in *Rel*. *Tutorial D* language has been proposed in the *Third Manifesto* [8] and dialect used by *Rel* is based on that proposal.

4.1 Assumptions

Let's assume that we have the entity type *ET* in the conceptual data model. *ET* has attributes *a1, ..., ax*. Let *A, B, C, D, F, G, H, I, J* and *K* be distinct base relvars that help to record data about the entities that have type *ET*.

Each attribute of *ET* has a corresponding attribute in the relvars *A* and *B*.

Relvars *C* and *D* have tuple valued attributes *t1* and *t2* (respectively) that have tuple types with the headings *TH1* and *TH2* (respectively). Each attribute of *ET* has a corresponding attribute in the headings *TH1* and *TH2*.

Relvars *F* and *G* have relation valued attributes *r1* and *r2* (respectively) that have relation types with the headings *RH1* and *RH2* (respectively). Each attribute of *ET* has a corresponding attribute in the headings *RH1* and *RH2*.

Relvars *H* and *I* have attributes *h1* and *i2* (respectively). Both these attributes have the scalar type *ST*. Each attribute of *ET* has a corresponding component of possible representation in *ST*.

Relvars *J* and *K* have relation valued attributes *r3* and *r4* (respectively). Attributes *r3* and *r4* have relation types that have headings *RH3={j1 ST}* and *RH4={k1 ST}* (respectively).

We also assume that tuple type has at least two attributes in its heading and that possible representation of the scalar type has at least two components. We also don't consider multiple levels of nesting. For example, relation type could have an attribute that has a relation type.

Let's define virtual relvars C', D', F', G', H', I', J' and K' using following relational expressions:

- C'=C UNWRAP t1
- D'=D UNWRAP t2
- F'=F UNGROUP r1
- G'=G UNGROUP r2
- H'=(EXTEND H ADD (THE_a1 (h1) AS a1, THE_a2 (h1) AS a2, ..., THE_ax (h1) AS ax)) {ALL BUT h1}
- I'=(EXTEND I ADD (THE_a1 (i2) AS a1, THE_a2 (i2) AS a2, ..., THE_ax (i2) AS ax)) {ALL BUT i2}
- J'=(EXTEND J UNGROUP r3 ADD (THE_a1(j1) AS a1, THE_a2(j1) AS a2, ..., THE_ax(j1) AS ax)) {ALL BUT j1}
- K'=(EXTEND K UNGROUP r4 ADD (THE_a1(k2) AS a1, THE_a2(k2) AS a2, ..., THE_ax(k2) AS ax)) {ALL BUT k2}

It is possible to deduce predicate of a virtual relvar by knowing relational expression that is specified then this relvar is created and predicates of the relvars that are referenced by the expression [7].

UNWRAP is the relational operator that unwraps an attribute that has a tuple type. It forms a relation which heading contains attributes that correspond to the attributes in the heading H of the tuple type, *instead of* one attribute with the type *TUPLE{H}* [7].

UNGROUP is the relational operator that "unnests" an attribute that has a relation type. It forms a relation which heading contains attributes that correspond to the attributes in the heading H of the relation type, *instead of* one attribute with the type *RELATION{H}* [7].

EXTEND is the relational operator that adds additional attribute to the relation. Values of this new attribute are found using some computational expression.

THE_ operators expose components of the possible representations of scalar types [7]. For example, we could have the scalar type *EMP_TYPE*. Operator *THE_empno,* which should be generated automatically by the DBMS, allows to access the corresponding possible representation component *empno* of values with the type *EMP_TYPE*.

{ALL BUT x} states that we want to see in the result all attributes except *x*.

4.2 The Extended Principle

The extended version of POD:

- There must not exist nonloss decomposition of the relvars A, B, C', D', G', H', I', J' and K' into set of projections A1, A2, ... Aj and B1, B2, ..., Bk and C'1, C'2, ..., C'm and D'1, D'2, ... D'n and F'1, F'2, ... F'o and G'1, G'2, ... G'p and H'1, H'2, ... H'q and I'1, I'2, ..., I'r and J'1, J'2, ..., J's and K'1, K'2, ..., K't (respectively) such that two projections from different sets have overlapping meanings.

4.3 The Rules for Reducing Data Redundancy

We also present additional heuristic rules that help to reduce data redundancy within *one* relvar.

1. Let's have the relvar *R* where are attributes *a1,..., an* and the relation valued attribute *r* with the relation type *RT* that has the heading *H*. Let's assume that if we could have the relvar *R'* with the type *RT* then the set of attributes *ai, aj, ..., ak* that are subset of the attributes *a1,..., an* in *H* would be a candidate key of the relvar *R'*. Then value of the relvar *R* must satisfy following relvar constraint:

Count((SUMMARIZE R UNGROUP r PER R UNGROUP r {ai, aj, ...,ak} ADD COUNT AS card) WHERE card>1)=0

2. Let's have the relvar *R* where are attributes *a1,..., an* and the tuple valued attribute *t* with the tuple type *TT* that has the heading *H*. Let's assume that if we could have the relvar *R'* with the type which heading is *H* then the set of attributes *ai, aj, ..., ak* that are subset of the attributes *a1,..., an* in *H* would be a candidate key of the relvar *R'*. Then value of the relvar *R* must satisfy following relvar constraint:

Count((SUMMARIZE R UNWRAP t PER R UNWRAP t {ai, aj, ..., ak} ADD COUNT AS card) WHERE card>1)=0

5 Discussion and Examples

Original version of POD covers the case with two distinct base relvars *A* and *B*. The extended version, that is one of the main contributions of this work, takes more cases into account.

According to the extended version of POD, database design must assure that if we want to record a set of attribute values about an entity in a database, then only one set of attributes in one base relvar must be suitable for that.

For example, database could contain the base relvar *Emp_RV* that is created based on the design 1 and the base relvar *Contract_RV* that is created based on the design 2 (see section 3). Following statement creates the relvar *Contract_RV*. We use user-defined types like *EMPNO_TYPE* but we could use built-in types as well.

VAR Contract_RV BASE RELATION {supervisor *TUPLE{empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE}*, contrno CONTRNO_TYPE, total SUM_TYPE, state STATE_TYPE} KEY {contrno};

Possible values of the relvars *Emp_RV* and *Contract_RV* can be seen in Fig. 3. Data about employees is duplicated in the values of the different relvars. Value of the relvar is called "relation".

Emp_RV

empno	ename	sal
1	JOHN	1000
2	BOB	1500
3	ANN	2000

Contract_RV

supervisor					
empno	ename	sal	contrno	total	state
1	JOHN	1000	1	50000	10
2	BOB	1500	2	40000	15
1	JOHN	1000	3	25000	10
3	ANN	2000	4	100000	20

Fig 3 Examples of values of the relvars

One possible projection of a relation is a relation itself. If we define virtual relvars by using projection operation, then following virtual relvars have no overlapping meanings because they have different amount of attributes with the different types.

Emp_VRV{empno, ename, sal} Key {empno};
Contract_VRV{supervisor} Key {supervisor};

These relvars are not isomorphic and according to Date and McGoveran [1]: "Two tables cannot possibly have overlapping meanings if they are not isomorphic." Therefore original version of POD is insufficient in this case. But following virtual relvars that are found based on the extended version of POD have overlapping meanings:

Emp_VRV{empno, ename, sal} Key {empno};
Contract_VRV UNWRAP supervisor {empno, ename, sal} Key {empno};

There are no duplicated tuples in the result of the relational expression because duplicates are automatically removed from the result. Both of these relvars have the following predicate:

e.empno EMPNO_TYPE AND
e.ename ENAME_TYPE AND
e.sal SAL_TYPE AND
(IF e.empno=f.empno THEN e.ename=f.ename AND e.sal=f.sal)

Therefore design of the relvars *Emp_RV* and *Contract_RV* is not correct in terms of the extended version of POD. In addition, value of the relvar *Contract_RV* violates the rule 2 from the section 4.3.

Relation type is a collection type. Soutou [9] writes: "Collections should model relationships when there are no strong integrity constraints and when there is a particular data access (via a separate relation)." The extended version of POD and rule 1 extend the theory about constraints on the collections.

We are aware that rules 1 and 2 from the section 4.3 don't prevent every possible inefficient design

but nevertheless we see them as means for improving database design. They can be enforced as relvar constraints. For example, each employee has exactly one salary number and name. One could create relvar *Employee_RV* that among others has the attribute with the tuple- or relation type for recording name and salary. Such design would be acceptable in terms of these rules but it is inefficient for other reasons. It would make it more difficult to change/retrieve data and enforce constraints.

Attributes with the tuple- or relation types are suitable to use in case of composition relationship in order to record data about the objects that are part of the whole. The part can't be part of many wholes and can't exist without the whole.

Next we will present an example about the usage of the rule 2 (see section 4.3). We use previously defined relvar *Contract_RV* that is created based on the design 2 (see section 3). Data about one employee is duplicated (see Fig. 3).

SUMMARIZE Contract_RV UNWRAP supervisor PER Contract_RV UNWRAP supervisor {empno} ADD COUNT AS card

empno	card
1	2
2	1
3	1

Fig. 4 Result of the SUMMARIZE operation

Result of the relational operation (see Fig. 4) shows that data about the employee, who has empno 1, is duplicated in the value of the relvar *Contract_RV*. Result of the Count operation is 1 and not 0 as required by the rule.

We could enforce constraint that prevents such data in the database by specifying that attribute *supervisor* that has the tuple type is a candidate key in the relvar *Contract_RV*. But combination of the attributes *empno*, *ename* and *sal* is the *superkey* of the relvar that is used in order to record data about employees. This key has not irreducibility property. Therefore specification of the constraint 2 (see section 4.3) gives to the DBMS more information about the data and helps to determine predicates of the virtual relvars C' and D' (see section 4.1).

6 Conclusions

Possibility to define new data types in a database extends considerably the set of possible database designs. Formal guidelines help to improve overall quality of the system. Correspondence to the formal guidelines could also be checked by the software. One such database design guideline is The Principle of Orthogonal Design which describes how to reduce data redundancy *across* different relvars. This article presents the extended version of the principle that takes into account that database could use complex user-defined data types. It also presents two additional heuristic rules for reducing data redundancy within the value of one relvar that has an attribute with a tuple type or a relation type.

Future work will include research of constraints of the tuple-valued and relation-valued attributes because they must be taken into account if relvar predicate is determined.

References:

[1] Date CJ, McGoveran D, The Principle of Orthogonal Design, *Database Programming & Design* 7, No. 6 (June 1994).

[2] Codd EF, A relational model of large shared data banks, *Comm. ACM*, Vol. 13, No. 6, 1970, pp 377-387.

[3] Jaeschke G, Schek HJ, Remarks on the algebra of non first normal form relations, In Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM Press, 1982, pp. 124-138.

[4] Roth MA, Korth HF, Silberschatz A, Extended algebra and calculus for nested relational databases, *ACM Trans. Database Syst.*, Vol. 13, No. 4 (Oct. 1988), pp. 389-417.

[5] Gulutzan P, Pelzer T, *SQL-99 Complete, Really*, Miller Freeman, 1999

[6] Melton J, ISO/IEC 9075-2:2003 (E) Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation), August, 2003. Retrieved December 26, 2004, from http://www.wiscorp.com/SQLStandards.html

[7] Date CJ, *An Introduction to Database Systems*, Pearson/Addison Wesley, 2003

[8] Date CJ, Darwen H, *Foundation for Future Database Systems: The Third Manifesto*, Addison-Wesley, 2000

[9] Soutou C, Modeling relationships in object-relational databases, *Data and Knowledge Engineering*, Vol. 36, Issue 1, 2001, pp. 79-107.

[10] Voorish D, An Implementation of Date and Darwen's "Tutorial D". Retrieved December 17, 2005, from http://dbappbuilder.sourceforge.net/Rel.html