Proceedings of the 10th WSEAS International Conference on COMPUTERS, Vouliagmeni, Athens, Greece, July 13-15, 2006 (pp1263-1268)

Whole-Part Relationships in the Object-Relational Databases

ERKI EESSAAR
Department of Informatics
Tallinn University of Technology
Raja 15,12618 Tallinn
ESTONIA

Abstract: - Widely accepted view is that the relational data model is not powerful enough for preserving semantics of the aggregation and composition relationships (whole-part relationships in general) in a relational database. Data model that is specified in SQL:2003 standard and used by the Object-Relational Database Management Systems (ORDBMSs) is believed to have better qualities in this regard. The Third Manifesto provides thorough revision of the relational data model and can be considered to be the manifest about ORDBMSs. We present designs for implementing whole-part relationships in a database that is maintained by the system which takes *all* the principles of the Third Manifesto into account. We show that ORDBMSs that use SQL language don't have all necessary means for implementing these designs and therefore still need improvement. We also show that some of the problems are caused by the shortcomings of the SQL standard.

Key-Words: - Whole-part relationships, Data models, Database design, SQL, Data types, ORDBMS

1 Introduction

It is possible to specify aggregation and composition relationships between entity types in the conceptual data models. Their semantics is described for example by Barbier et al. [1]. General name of this kind of relationships is "whole-part relationship". How to preserve their semantics in a database?

Names of the database objects are not sufficient and suitable method in order to make semantics of the data understandable to a Database Management System (DBMS) [2]. But DBMS can enforce structural and operational properties of the relationships and objects which participate in these relationships [3]. Properties of its underlying data model (for example networked, relational etc.) determine how well DBMS can preserve semantics of the reality in a database.

There exists works (for example by Rahayu et al. [4]) that describe how to map whole-part relationships to the relational database tables and constraints. But in general, relational data model that is used in the current mainstream Relational DBMSs ($RDBMS_{SQL}$) is considered to be unsuitable in order to preserve semantics of the complex relationships (including whole-part) in a database. $RDBMS_{SQL}$ uses the database language that conforms to SQL:1992 or earlier standard. Object-Relational DBMS ($ORDBMS_{SQL}$) is considered to be better than $RDBMS_{SQL}$ in this regard. $ORDBMS_{SQL}$ uses the database language that conforms to SQL:1999 or later standard.

The Third Manifesto [5] contains thorough revision of the relational data model. According to Date [6], the relational data model allows to *extend* the sets of types and operators that a database designer could use. It consists of a collection of scalar types, relation type generator, facilities for defining relation variables (relvars) of such types, assignment operations for assigning relation values (relations) to relvars and a collection of generic relational operators for deriving relation values from other relation values.

The Third Manifesto can be seen as a manifest about the desired properties of the object-relational DBMSs because many of its proposed features have counterparts in SQL:1999 and SQL:2003 standard. We reference to the system that follows the rules of the Third Manifesto as $ORDBMS_{TTM}$. The *goals* of this article are:

1. To propose a set of designs for preserving semantics of the whole-part relationships in an $ORDBMS_{TTM}$ database. Our goal is not to give detailed overview when to use each case but to use these designs in the following evaluation.
2. To evaluate how well we can use these designs in an $ORDBMS_{SQL}$ database. Results help to find problems of $ORDBMS_{SQL}$s and SQL standard. Such study is important because it gives guidelines how to improve currently widely accepted language and systems.

The rest of the paper is organized as follows. Section 2 gives an overview of the existing work about implementing aggregation and composition relationships in an $ORDBMS_{SQL}$ database. Section 3 presents design alternatives for implementing whole-part relationships in an $ORDBMS_{TTM}$

database. We are not aware of any existing study about that. Section 4 describes problems that hamper usage of these designs in an $ORDBMS_{SQL}$ database. Section 5 summarizes this article.

2 Related Works

Some researchers try to extend an underlying data model of an $ORDBMS_{SQL}$ with the relationships as first class objects. For example, extension module ORIENT [3] extends Informix DBMS by providing CREATE RELATIONSHIP statement and means for recording and using relationship data.

Second approach tries to add support to the relationships by using existing facilities of the DBMSs and their underlying data model. SQL:2003 defines type constructors ROW, ARRAY, REF and MULTISET and permits creation of the user defined structured types (UDTs) [7]. There are already a lot of suggestions how to implement whole-part relationships in an $ORDBMS_{SQL}$ database by using constructed array type or table types (the latter is interpretation of a multiset type constructor in Oracle DBMS) [8], indexed clusters or table types (features in Oracle DBMS) [9], constructed multiset – or row type [10]. Data about the part instances can be recorded in the columns that have complex data types and therefore data about the whole instances and their part instances can be recorded in one table at the logical level. Usage of the clusters in Oracle means that data about the whole and part instances can be recorded together at the physical level, but at the logical level they remain in the separate tables.

Our comment about the array types is that array is a collection in which elements have a defined order and the same element can be in the collection more than once. Tuples in the body of a relation are unordered and relations can't contain duplicated tuples [5]. Therefore we can't use arrays in order to implement relationships if we want to treat their participants in a uniform way.

Proponents claim that the object-relational features help to implement relationships in more natural and semantics-preserving ways. But researches have also identified problems of using collections in the conceptual modeling [11] and in the database schema. "A collection is a composite value comprising zero or more elements, each a value of some data type DT." [7, p. 45] Halpin and Bloesch [11] note that collections make harder to express constraints (which typically occur on members, not collections) in a conceptual model. But if it is difficult to use declarative language like OCL in order to express constraints in a conceptual model, then it is also difficult to express declarative constraints and queries on collections in a database. Smith and Smith [12] propose usage of the complex types as domains for the attributes in relations in order to record semantically important information about an aggregation of objects in a relational database. They also identify possible problems that include restrictions to ways how user can access data and duplication of data. The latter causes waist of storage space as well as introduces problems of possible inconsistency [12]. One solution could be usage of pointers but "Pointers are objects which have no real-world analog and serve to dramatically increase the complexity of database interactions." [12] Soutou [13] has also identified this problem and writes: "Collections should model relationships when there are no strong integrity constraints and when there is a particular data access (via a separate relation)." Collections offer little performance gain according to experience of Halpin and Bloesch [11]. "Collections can provide better performance than a standard relational database, but require more complex queries for data retrieving." [12] Comment to the last observation is that performance is an implementation issue, not a model issue [6] and shouldn't be a criterion for evaluating different data models.

3 Whole-part Relationships in an $ORDBMS_{TTM}$ database

Let's assume that a conceptual data model contains two entity types *Whole* (represents the whole) and *Part* (represents its part) that are associated with a whole-part relationship. *Whole* has attributes *a* and *b*. *Part* has attributes *c* and *d*. These attributes have the type INT. Attributes *a* and *c* are unique identifiers of the corresponding entity types.

Following section contains statements that are written in Tutorial D relational language and have mostly been tested using the prototypical $ORDBMS_{TTM}$ *Rel* [14]. We haven't tested outer join operator that is not yet fully supported by *Rel*.

Next we present types of the *base* relvars that one could create based on the entity types *Whole* and *Part*. These types are accompanied with the possible *names of relvars*. Relvars in the designs 2 and 4 use attributes that have a generated tuple type and relation type, respectively. Attributes of these types are created based on the entity type *Part*. The attribute *part* in design 3 has a scalar type ST. Each attribute of the entity type *Part* has a corresponding component of a possible representation in this scalar type.

Design 1: *Whole*: RELATION {a INT, b INT}
Part : RELATION {c INT, d INT, a INT}

Design 2: *Whole*: RELATION {a INT, b INT, part TUPLE {c INT, d INT}}

Design 3: *Whole*: RELATION {a INT, b INT, part ST}

Design 4: *Whole*: RELATION {a INT, b INT, part RELATION {c INT, d INT}}

Design 5: *Whole*: RELATION {a INT, b INT}
Part: RELATION {c INT, d INT}
PartOfWhole: RELATION {a INT, c INT}

The relvar *Part* has one foreign key (attribute *a*) and relvar *PartOfWhole* has two foreign keys (attributes *a* and *c*) in case of the designs 1 and 5, respectively. For example, value of the foreign key must match value of some candidate key of the relvar *Whole* in case of the design 1.

All these designs require additional constraints depending on the properties of the relationship that they help to implement. These properties are thoroughly described by Barbier et al. [1].

For example, following constraint ensures in case of the design 4 that data about the same part instance is not recorded more than once, across all the values of the attribute *part*.

IS_EMPTY((SUMMARIZE (Whole UNGROUP part) PER Whole UNGROUP part {c} ADD COUNT AS cnt) WHERE cnt>1);

IS_EMPTY (<relation exp>) is the scalar operator that evaluates to true if the body of the relation denoted by <relation exp> contains no tuples [5]. The idea of this constraint is to "unnest" the attribute *part* and count how many times each value of the attribute *c* participates in the result. The set of *c* values that exist in the result more than once must be empty. Attribute *c* is the unique identifier of the entity type *Part*. We need similar constraint with the UNWRAP operator in case of the design 2 that uses a generated tuple type. In case of the design 3 we must declare that the attribute *part* is a candidate key in order to ensure that data about the parts is not recorded repeatedly within the value of one relvar.

Relationship ends have participation and cardinality constraints which determine the possible amount of instances that can be associated with an instance in the context of a relationship. These constraints can be presented as endpoints of a range.

The following constraint is usable in case of the design 1 and specifies the amount of part instances that can be associated with a whole instance. Left join operator is needed in order to tackle the case when whole instance has no associated part instances. The set of tuples that represent unsuitable cardinality range must be empty.

IS_EMPTY((SUMMARIZE Whole LEFT JOIN Part PER Whole {a} ADD Count AS card) WHERE NOT (*constr*));

Examples of the constraint *constr* are: card=1, card>2; card>=1 AND card <=8.

3.1 Advantages and Disadvantages of the Designs that Use Complex Types

Designs 1-5 can be divided into two groups. Each entity type has a corresponding base relvar in a database in case of the designs 1 and 5. Data about the whole instance as well as its part instances is in one tuple that is part of the value of one base revar in case of the designs 2-4. Designs 2-4 use complex data types in order to record data about the parts.

Possible advantage of the designs 2-4 is that data about the instance and its associated instances can be accessed by only accessing one relation. But in case of the designs 1 and 5 we could use virtual relvars for the same purpose. They provide even more flexibility because user has more choices about which relvar to use and how complex tuple to retrieve. For example, in case of the design 5 we can create virtual relvar with the following general expression:

Whole LEFT JOIN PartOfWhole LEFT JOIN Part GROUP {c, d} AS part;

This relvar has the following relation type: RELATION {a INT, b INT, part RELATION {c INT, d INT}}

Designs 2-4 don't necessarily reduce the amount of relvas in a database because we may create virtual relvars for accessing directly data about the parts. For example, in case of design 4 we can create relvars with the following relational expressions in order to make access to the data of part and whole instances more comfortable:
OnlyWhole: Whole {ALL BUT part};
Relation type is: RELATION {a INT, b INT}
OnlyPart: Whole UNGROUP part {a, c, d};
Relation type is: RELATION {a INT, c INT, d INT}

If we use such virtual relvars, when why to create base relvars with the complex types in the first place?

Designs 2-4 make more difficult to discover data redundancy across different relvars. For example one could create relvars with the following types: *Emp: RELATION{empno INT, ename CHAR, sal INT}* and *Contract: RELATION{contractno INT, creation_time, supervisor TUPLE {empno INT, ename CHAR, sal INT}}*

Values of both relvars can contain data about the same employees. Orthogonal database principle helps to discover data redundancy across different relvars. Version of the principle that takes into account usage of the complex data types (designs 2-4) [15] is much more complicated than the original principle [2] that doesn't take them into account.

Designs 2-4 make *naive* implementation of versioning easier. If we change the tuple, then the versioning system must preserve old version of it. Problem is that even the smallest change causes recording of the old version of the entire tuple and therefore data about the whole instance as well as its associated part instances. It causes data redundancy and increases needs for the storage space.

Designs 2-4 can make easier to implement concurrency control by database vendors, because it is possible to use existing functionality of the DBMSs. For example, if DBMS records data about the parts and wholes together at the physical level and uses locking, then only one tuple of a base relvar has to be locked in order to lock data about the entire object. This tuple may be recorded to one data block that is part of a data file. But locking is method of concurrency control that belongs to the implementation level of the system.

If we want to keep logical distinction of model and implementation, then "easy implementation" shouldn't be argument that forms and reshapes the model. If there are two separate relvars at the model level, then at the physical level their data might by recorded together. For example, Oracle [16] permits creation of indexed- or hash clusters in order to achieve just that. Date [6, p. 301] describes The *Principle of Interchangeability* according to which there must be no arbitrary and unnecessary distinctions between the base and virtual relvars. Therefore $ORDBMS_{TTM}$ should allow update virtual relvars the same way as base relvars. Such update propagates to the underlying base relvars and causes locking of the relevant tuples that are part of their values.

Conclusion of our analysis is that usage of the complex data types doesn't have such clear advantages that one could conclude after reading existing research papers [3], [10].

4 Problems of $ORDBMS_{SQL}$

Similarities and differences of the Third Manifesto constructs and SQL constructs are for example discussed by Date [6]. Researches like Pardede et al. [10] have concentrated their attention to the advantages of the new features of SQL that help to implement whole-part relationships. This section addresses some of the problems of SQL and $ORDBMS_{SQL}$s in this regard in order to show areas that need improvement. Table 1 contains comparison of the standards and some existing systems (Oracle 10g and PostgreSQL 8.0) in terms of some of the features that help to implement the designs 1-5. List of the features is not complete due to the space restrictions. Column "Designs" contains references to the designs where such feature is needed. We see that DBMSs implement only subsets of the standard and use proprietary extensions. Last row of Table 1 summarizes support to the features by SQL standard and two $ORDBMS_{SQL}$.

By default it is not possible to add key constraint to the column that has a row type in PostgreSQL database (row 4 in Table 1). One has first to create an operator class and a b-tree support function that compares two values that have a row type.

Examples of the constraints to the values of the generated types (row 7 in Table 1): (a) an attribute of the generated type must be mandatory; (b) a row that is part of the value with the generated type must satisfy some predicate; (c) a value with the multiset type can't contain some value repeatedly; (d) values with the generated type in one column of a table can't contain some value repeatedly; (e) an attribute of the generated type (row or multiset type) is also foreign key attribute; (f) a value of the multiset has a participation and cardinality constraint.

Constraint (c) will be automatically enforced in case of the type constructor SET. It is not present in SQL:2003. But we can create a view (virtual table) where duplicate elements that are part of the multiset value are removed by using the function SET. In theory we could create declarative constraints to the values of the multiset types by using table- or database constraints that use for example UNNEST operator or MEMBER, SUBMULTISET or SET predicates, introduced in SQL:2003 [7]. In practice we can't create these constraints because of the limited support to the declarative constraints in the $ORDBMS_{SQL}$s (row 6 in Table 1).

Table 1. Some features that help to implement designs 1-5 in the DBMSSs

ID	Third Manifesto [5]	Designs	SQL:2003 [7]	Oracle 10g [16]	PostgreSQL8.0
1	tuple type generator	2	row type constructor	NO	yes
2	relation type generator (relation is a *set*)	4	*multiset* type constructor	yes (table type – supports the feature but not the syntax)	NO
3	user defined scalar type	3	user defined structured type (UDST)	yes	NO
4	attribute with the complex type can be a key	2, 3, 4	yes (no reference that it can't be)	NO	default NO (needs programming)
5	attribute with the complex type can be mandatory	2, 3, 4	yes (no reference that it can't be)	yes in case of UDSTs. NO in case of the table types	yes
6	complex *declarative* relvar and database constraints	1, 2, 3, 4, 5	CHECK constraint with a subquery	NO - CHECK constraint can't contain a subquery	
			assertion object	NO - not possible to create assertions	
7	*declarative* constraints to the values of the complex types	2, 3, 4	attribute constraints and possible to declare others by using for example UNNEST function.	attribute constraints and relvar constraint that attribute of the UDST is mandatory	attribute constraints and relvar constraints to the values of row types
8	it is possible to change value of *multiple* relvars through a virtual relvar (view) which expression contains a join	1, 5	yes in case of one-to-one join. NO in case of one-to-many join - only "many side" is updatable [6, p. 322]	default NO (yes if not-standardized instead-of triggers (Oracle) or rules (PostgreSQL) are programmed)	
9	automatically generated = and ≠ *operators* for comparing values with a complex type [5]	2, 3, 4	yes	yes	default NO (there is CREATE OPERATOR statement and possible to program it)
10	*builtin* GROUP operator	1, 5	COLLECT + SET functions	CAST + SET functions	NO
11	*builtin* UNGROUP operator	4	UNNEST function	TABLE function	NO
12	*builtin* WRAP operator	1, 5	NO	NO	NO
13	*builtin* UNWRAP operator	2	NO	NO	NO
14	IS_EMPTY *builtin* scalar operator	1, 2, 3, 4, 5	NO	NO	NO
15	possibility to define new scalar *operators*	1, 2, 3, 4, 5	NO	yes	yes
∑	**supports fully / supports partially / doesn't support**		**8 / 3 / 4**	**3 / 5 / 7**	**3 / 4 / 8**

The same problems are with the database constraints that reference more than one table (designs 1 and 5). It is possible to use triggers for that. Problems of using triggers: (a) usage of a proprietary imperative language; (b) creation of a trigger doesn't cause automatic evaluation of the existing data; (c) we need many triggers in order react to all the events that can cause invalidation of the constraint; (d) SQL standard doesn't permit to defer execution of the trigger to the end of a transaction.

Let's assume that we want to enforce the structural constraint that the whole instance must all the time be associated with between two and six part instances. We need triggers that react to the following events, in case of the design 1:
(a) Insertion of a new row to the table *Whole.*
(b) Insertion of a new row to the table *Part.*
(c) Modification of a part identifier in the table *Part.*
(d) Deletion of a row from the table *Part*.

For example, if a new row is added to the table *Whole,* then we have to associate it with the data about the part instance in the table *Part.* These operations must be part of one transaction and a DBMS must check the data at the end of the transaction. If any of the checks fails, then transaction should be rolled back. It can be implemented in PostgreSQL by using not-

standardized constraint triggers. They allow to defer execution of the trigger procedure to the end of transaction.

In PostgreSQL all views need further programming in order to be updatable (row 8 in Table 1). In Oracle DML statement must affect only one underlying table of the updatable join view.

Users of the current $ORDBMS_{SQL}$s must often manually create the additional database objects if some object is created in a database. We believe that system should create these objects automatically. Or at least the system should allow to use the schema triggers (like in Oracle) in order to allow to create the generation program that is executed, when database schema changes. SQL standard [7] doesn't currently permit such triggers.

5 Conclusions

The Third Manifesto provides thorough revision of the relational data model. We have presented possible database designs for implementing whole-part relationships in a database which is maintained by a manifesto-compliant DBMS. These designs help to preserve structural and operational properties of the relationships in a database. Some of these designs use complex data types. We conclude that the data model that is described by the manifest allows to implement all these designs. But designs with the complex data types don't have a big advantage compared to others because we still need constraints and virtual relvars. We have also investigated current SQL standard and two DBMSs in order to find out how well they support the proposed designs. We have found that SQL standard and current systems have shortcomings and need to be improved in order to allow to use all the designs.

References:

[1] Barbier F, Henderson-Sellers B, Le Parc-Lacayrelle A, Bruel J, Formalization of the Whole-Part Relationship in the Unified Modeling Language, *IEEE Trans. Softw. Eng.* Vol. 29, Part 5, 2003, pp. 459-470.

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

[3] Zhang N, Ritter N, Härder T, Enriched Relationship Processing in Object-Relational Database Management Systems, *In Proceedings of the CODAS'01*, 2001 pp. 53-62.

[4] Rahayu W, Chang E, Dillon TS, Implementation of Object-Oriented Association Relationships in Relational Databases, *In Proceedings of the IDEAS'1998*, IEEE Computer Society, 1998, pp. 254-263.

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

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

[7] 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

[8] Marcos E, Vela B, Cavero JM, Caceres P, Aggregation and composition in object-relational database design, *In Proceedings of the ADBIS'2001*, Springer, 2001, pp. 195-209.

[9] Rahayu JW, Taniar D, Preserving Aggregation in an Object-Relational DBMS. *In Proceeding of ADVIS 2002,* Lecture Notes in Computer Science, Springer-Verlag GmbH, Volume 2457 / 2002, pp. 1-10.

[10] Pardede E, Rahayu JW, Taniar D, Composition in Object-Relational Database, *Encyclopedia of Information Science and Technology*, IDEA Publishing, 2005, pp. 488-494.

[11] Halpin T, Bloesch A, Modeling Collection in UML and ORM. *In Proc. 5'h IFlP WG8.1 Int. Workshop on Evaluation of Modeling Method in System Analysis and Design*, 2000.

[12] Smith J, Smith D, Database abstractions: aggregation, *Communications of the ACM*, Vol. 20, No. 6, 1977, pp. 405-413.

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

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

[15] Eessaar E, Guidelines about Usage of the Complex Data Types in a Database, *WSEAS Transacions on Information Science and Applications*, Issue 4, Vol. 3, 2006, pp. 712-719.

[16] Oracle® Database SQL Reference 10g Release 1 (10.1) Part Number B10759-01. Oracle Corp., Retrieved October 4, 2005, from http://download-west.oracle.com/docs/cd/B14117_01/server.101/b10759/toc.htm

[17] PostgreSQL 8.0.3 Documentation. Retrieved October 4, 2005, from http://www.postgresql.org/docs/8.0/interactive/index.html