in: Proc. 3rd Int. Symposium on Cooperative Database Systems for Advanced Applications (CODAS'01), Bejing, April 2001, pp. 53-62

Enriched Relationship Processing in Object-Relational Database Management Systems

Nan Zhang, Norbert Ritter, Theo Härder
Database & Information Systems Group, University of Kaiserslautern
P.O.Box 3049, D-67653 Kaiserslautern
e-mail: {zhang / ritter / haerder}@informatik.uni-kl.de

Abstract

In this paper, we bring together two important topics of current database research: enhancing the data model by refined relationship semantics and exploiting ORDBMS extensibility to equip the system with new functionality. Regarding the first topic, we introduce a framework to capture diverse semantic characteristics of application-specific relationships. Then, in order to integrate the conceptual extensions with the data model provided by SQL:1999, the second topic comes into play. Our efforts to realize semantically rich relationships by employing current ORDB technology clearly point up the benefits as well as the shortcomings of its extensibility facilities. Unfortunately, deficiencies still prevail in the OR-infrastructure, since the features specific to the extensions cannot sufficiently be taken into account by DBMS-internal processing such as query optimization, and there are very limited mechanisms of adequately supporting the required properties, e. g., by adjusted index and storage structures as well as suitable operational units of processing.

1. Motivation

To meet the increasingly challenging requirements of today's applications, both database research and industry are being engaged in adding new features to database management systems (DBMSs). A major effort in this direction is represented by object-relational DBMSs (ORDBMSs), which gain more and more popularity due to their extensibility, i. e., the ability to allow user-defined extensions to be added to the system and used in the same way as native DBMS facilities [18].

As a typical sort of extension, relationships which capture the meaning of the associations among DB objects are of particular interest and importance in database applications. Thus, it would be very beneficial to have relationships available as modeling constructs and implemented in the database. However, little progress has occurred in effectively incorporating their semantics into DBMSs, including the current ORDBMSs. In most cases, warranting specific relationship semantics remains to be burdened on the application developer [17].

Motivated by these observations, we address how to exploit the extensibility as well as the expressiveness of current OR-DBMSs to realize an enhanced support of relationship semantics. As the baseline of the whole work, Sect. 2. presents modeling concepts to capture semantic properties of relationships in the real world and discusses the implications such a data model enrichment has. Our main concern lies in adequately incorporating the proposed concepts into the system. For this purpose, a DataBlade-based approach which tries to balance the desired integration depth against the available OR extensibility is explored in Sect. 3.. There, user-defined relationships are implemented as constructs resided in the database server, namely, as user-defined routines (UDRs), thereby making the relationship support a more integral part of the system than just a conventional on-top supplement. Our effort will reveal how well the ORDB technology can meet the demands of handling semantically rich relationships. This leads us to a thorough deliberation on DBMS extensibility in Sect. 4., by inspecting the mechanisms that are employed to support DBMS extensibility (especially, those of the prospering ORDBMSs). Finally, we give our opinion upon what characteristics a proper extensibility infrastructure should present, what fundamental principles should be followed, as well as where the challenging problems or risks may remain. We believe, such a sound understanding of the problem-specific processing will demonstrate a progressive step towards attaining a realistic notion of DBMS extensibility, as to be concluded in Sect. 5..

2. Capturing Relationship Semantics

Relationships are "semantically rich" in that their interpretation does not solely lie in their names or their structural connections, but also in the constraints which restrict their behavior and must be satisfied. Semantically rich relationships exist in many application domains (e. g., CAx, document management, etc.). To obtain an impression of the diversity they exhibit, we take a simplified scenario from another project SENSOR ("Supporting Software Engineering Processes by Object-Relational Database Technology" [9], sub-project of the SFB 501 "Development of

Table 1. Relationship examples with their semantics

Semantic aspect	(generic) aggregation		(application-specific) consist_of		
	O	**P**	**O** (*module*)	**P** (*function*)	**P** (*procedure*)
Degree	-		3		
Composition	✓		→ (*aggregation*)		
Cardinality	1,1	-	→	0,8	1,10
Sharability	✗	-	→	✗	✗
Existence dependency	**RI/CI**, **MD/CD**	**RI/CI**	→ **RI**, **MD**	→ **CI**	→ **CI**
Transitive operation	**CS**	-	→ **CS**, **MU** (last_modified)	**CU** (length)	**CU** (length)

Notes: "**O**" owner; "**P**" participant; "✓" predefined; "**-**" to be further specified; "→" inherited; "✗" not existing

Large Systems with Generic Methods", funded by the German Research Association DFG) conducted in our group. In this scenario, software constituents arising throughout the whole development process are interrelated. At the implementation level, a program consists of several modules. Each module, in turn, is made up of different functions and procedures. All these components are managed in a repository. To achieve the "traceability" between different development phases, software components are further divided into packages according to the requirement specifications they realize. Moreover, programs and modules may be continuously under evolvement. Some changes in a program or module produce a new version of it. While the new version replaces the old one, the dated version might be archived for future reference. This scenario encompasses a number of relationships, including not only generic ones but also application-specific ones. Typical examples of generic relationships are among others *aggregation* relationships [14] (e. g., the relationship *consist_of* expressing that entity *module* is assembled with entity *function* and entity *procedure*). Thus, the data involved is in general not only structurally related to, but also semantically dependent on each other, which has great implications both on data modeling and on data processing.

In our approach, we consider a **relationship** modeling an association between two or more entities (**participants**). We use the term "relationship" as synonym of "relationship type" and "participant" as synonym of "participating type". Relationships are primary modeling constructs with their exact, application-specific properties modeled in an explicit manner. Generic, built-in relationships are supported as well. In this way, it is possible, on one hand, to capture subtle differences among relationships and, on the other hand, to make the relationship facility easy to be reused and extended.

Tab. 1 characterizes two relationship examples. The first example, *aggregation*, deals with how a complex object (a composite object or an aggregate) is assembled with its integral parts (or components). The aggregate (denoted as "**O**") and all its parts (denoted as "**P**") should be addressed collectively as a whole and can be defined to exist together. As a generic relationship, *aggregation* has some properties predefined (denoted with "✓") while others varying in concrete application domains (denoted with "**-**"). The second example, *consist_of*, reflects that a module is assembled with functions and procedures. It is a special kind of *aggregation* with add-on semantics (denoted with "→") as to be discussed later on in this section. In the table, symbol "✗" means that a semantic aspect is not necessary for the relationship.

Below we give a brief introduction of the structural and operational properties that are relevant to *consist_of*. Some of the semantic properties have also been (partly) analyzed by related work such as [6, 13, 16]. However, a thorough consideration of them can be found nowhere else but in [22].

Structural properties. In respect of structural connections, we identify the following fundamental characteristics.

- **Degree** defines the number of participants associated in the relationship. For *consist_of*, it is three.
- **Cardinality** places restrictions on the number of instances of the participant that can be associated with a single instance of the relationship. In our example, at least one and at most ten procedures have to be included in a module.
- **Composition** determines how the participants of a relationship cohere. A relationship with the composition property is called a "composite relationship" (e. g., *consist_of* as mentioned above); otherwise, it is a "non-composite relationship". An instance of a composite relationship is a "composition". While in a non-composite relationship all participants play the uniform role, the composition property assigns different participants of a relationship different roles: **Owner** (e. g., *module*) plays the superordinated role and **participant** (e. g., *function*; from now on, when speaking about participants in a composite relationship, we always mean participants other than the owner) are subordinated to the owner. Operations are propagated from the owner to other participants.
- **Sharability** (or **exclusiveness**) denotes, in case of a composite relationship, whether a participant instance can be associated in more than one relationship instance. In *consist_of*, a function or a procedure can not be shared among different modules.

Operational properties. Operational properties can be interpreted as consecutive actions that an operation on a participant may cause, i. e., operation propagation. To govern existence dependency, certain insertion or deletion operations should automatically trigger actions on related objects. Besides, there are also other cases when DB operations must be executed transitively. For instance, selection operations at the aggregate level can be propagated to the part levels, or vice versa.

• **Mandatory Deletion (MD)**: Upon deletion of an object, its associated objects are also deleted, even though they may be involved in other relationship instances. In addition, the corresponding relationship instances are also removed. In *consist_of*, the deletion of a module will cause the deletion of all its parts.
• **Conditional Selection (CS)**: The selection of an object returns this object and (only) those associated objects involved in the given relationship instance. In *consist_of*, selecting a module also renders its parts.
• **Conditional Insertion (CI)**: Upon insertion of an object, the relationship instance is established between this object and those associated objects that already exist, while associated objects that do not exist yet are represented with placeholders (stubs). In *consist_of*, a function or procedure can always be inserted even if there exists no module for it. In this case, a "stub" module is used to establish a "partial" relationship instance.
• **Restricted Insertion (RI)**: Upon insertion of an object, the relationship instance must be established between this object and all other associated objects. If any associated object is not available in the database, the insertion is denied. In *consist_of*, the insertion of a module cannot take place if there exists no function or procedure yet. In this case, an "entire" relationship instance is required.

For semantics controlling purpose, the modeling constructs should be reflected at the database language level. Both, DDL extensions for the definition of semantically rich relationships as well as DML extensions for the retrieval and manipulation of the data corresponding to the specified schemas are needed. This results in OrientSQL, with SQL-like syntax conforming to the current database standard [21]. While a thorough description of OrientSQL can be found in [22], some examples are given here.

Relationship specification: The statements in Fig. 1 define a new relationship on the basis of a generic one.

```
CREATE RELATIONSHIP aggregation (
   aggregate OWNER
   ON DELETE MADATORY DELETION
   ON SELECT CONDITIONAL SELECTION,
   part PARTICIPANT);

CREATE RELATIONSHIP consist_of UNDER aggregation (
module OWNER (aggregate)
   ON INSERT RESTRICTED INSERTION
   ON UPDATE (last_modified) MANDATORY UPDATE,
function PARTICIPANT (part)
   NON SHARABLE
   CARDINALITY [0,8]
   ON INSERT CONDITIONAL INSERTION
   ON UPDATE (length) CONDITIONAL UPDATE,
procedure PARTICIPANT (part)
   NON SHARABLE
   CARDINALITY [1,10]
   ON INSERT CONDITIONAL INSERTION
   ON UPDATE (length) CONDITIONAL UPDATE);
```

Figure 1: Relationship definition

Querying: Fig. 2 shows two simplified OrientSQL queries.
• OrientSQL's path extensions enable to access data objects across user-defined relationships. For this purpose, the name of the relationship to be addressed is explicitly indicated following the referencing construct ".", like *m.consist_of* in the first query.
• The contents to be accessed is specified with the dereferencing construct "->". To avoid obscurity when several participants of a relationship have the same attribute (e. g., *language*), a certain participant (e. g., *function*) can be designated with the participant-resolving construct "*()*" following the relationship name.
• OrientSQL's query facility considers a complex object and all its parts collectively, in accordance with the composition property and the selection semantics. For this purpose, the owner (e. g., *module*) and the relationship name (e. g., *consist_of*) are explicitly used in the *FROM* clause to specify the scope of the query. The second query then returns not only the qualified modules but also their functions and procedures.

Query 1: Find the names of all modules which consist of functions written in Java.

```
OrientSQL: SELECT m.name
           FROM module m
           WHERE m.consist_of (function) -> language = 'Java'
```

Query 2: Find all modules designed by Zhang together with their constituents.

```
OrientSQL: SELECT *
           FROM module (consist_of) m
           WHERE m.designer = 'Zhang'
```

Figure 2: Querying examples

Relationship insertion and insert block: To build a consistent relationship instance, a special facility is provided for two purposes: First, for the user to intentionally group together several operations so that participating objects are inserted before the insertion of a relationship instance; second, for the system to ensure the "success unit" of all necessary operations to build a consistent relationship instance. This facility is called **insert block**.

An insert block comprises a sequence of insertion operations that will be executed with the examination of insertion semantics delayed at the block end. For **CI** property, it will be checked up to the block end, which associated objects of the relationship instance are missing and whether stubs have to be created. For **RI** property, it will be decided up to the block end, whether all the associated objects have already been inserted so that the relationship instance can be constructed, or whether all the insertions within the block have to be rejected.

3. Implementation Approach

Our work has two primary goals in respect of relationship support. The first is to exceed the modeling limitations imposed by the DBMS and its data model. Such a goal has been achieved through two levels of extensions [22]: the conceptual constructs close to the user's perception of relationship semantics appearing in the real world, and the language mechanisms allowing relationships semantics to be defined in the schema and considered in data retrieval and manipulation. The second goal, i. e., to make the DBMS responsible for enforcing the specified relationship semantics, will be addressed in this section.

3.1. General consideration

Generally, for any extension of DBMS facilities, there exist different implementation approaches that entail varying acceptability of the resulting system. These alternatives can be distinguished according to the depth of integration between the underlying DBMS and the extensions.

One extreme along this dimension is a “naive” mapping approach in which declarative OrientSQL constructs are realized as “syntactic sugar” of built-in constructs of the logical data model, without resorting to any procedural code. The functionality achieved, however, is restricted by native features of the underlying system. In our case, even with the expressive power of object-relational data model (ORDM) constructs, e. g., references, PK/FK pairs, collection types, declarative constraints, and triggers, some precise relationship semantics such as composition still gets obscured during translation or is difficult to translate, as evident in [22].

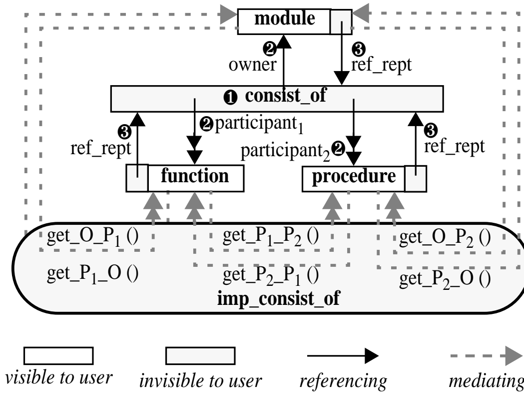

Figure 3: Relationship definition

The other extreme is deep integration that seamlessly couples newly-added modeling constructs with the rest of the DBMS. To achieve this promising goal, reasonable extensions at the processing level are indispensable. Unfortunately, this demand cannot be met, even by current ORDBMSs that are labeled “extensible”. Some functions, such as the support for a kind of block atomicity necessary for the construction of composite relationship instances, may not be realizable. This is because none of the available ORDBMSs (and of course other commercial systems) provide access to transaction manager information, nor do they allow us to gain control as needed and to define the commit or abort methods. While the OR extensibility makes the integration of user-defined types (UDTs) and type-specific behavior relatively easy, its application to the relationship support which goes beyond the scope of data types is still not feasible [22].

Between these two extremes is a spectrum of layered architectures where extensions are built on top of the existing DBMS. In comparison with “direct mapping”, this alternative aims at dedicated implementation rather than only naive use of the built-in data model features. Moreover, it also differs from “seamless integration” in that it incorporates new functionality in the system through existing interfaces instead of touching the internal processing. Our relationship support follows this way.

It makes particular sense to implement relationships as constructs handled in the system, thereby reducing the client/server communication overhead and enclosing application semantics in querying. The OR extensibility appears to be very instrumental to this task: In contrast to the classical way of defining a supplementary layer on top of the system, ORDBMSs offer several means for the user to compile and reside programming code within the server as “predefined database routines”. These means include stored procedures, triggers, and UDRs. Among them, UDRs are most attractive to us because of their portability, the expressiveness they exhibit, as well as the ability to define type-specific behavior and to hierarchically organize “relationship” UDTs. Accordingly, the intended functionality can be achieved in such a manner that:

- User-defined relationships are realized as UDTs with specially designed data structures and operations.
- UDTs are organized into a hierarchy to support refinement and extension during implementation.
- The whole implementation is encapsulated in a pluggable package, which extends the system with desired relationship support and can live inside the DBMS when needed.

This way, a reasonable compromise is made between desired integration depth and available OR extensibility. Below, we will discuss the implementation philosophy in more detail.

3.2. Basic constructs

ORDMs allow the definition of UDTs to build up an organizational framework for the new functionality.

Representation constructs. First of all, to represent relationships and to accommodate their instances, special data structures are defined in a DB schema. This is accomplished through “schema expansion” in that a user-defined relationship is expanded into a structured type and a corresponding table. As shown in Fig. 3, the schema expansion process has several steps:

❶ Type definition: A separate “relationship representation type” (or “representation type”) *consist_of* is defined to represent the relationship *consist_of* between the participants, which, in turn, are represented by structured types (i. e., “participant representation types”) *function*, *procedure* and *module*.

❷ Multi-connection: The relationship representation type *consist_of* is connected to its associated participant representation types via reference attributes $participant_1$, $participant_2$, and *owner*, respectively.

❸ Reference mediation: In the opposite direction, a hidden reference attribute *ref_rept* is added to each of the participant representation types to refer to the relationship representation type *consist_of*. Such a reference attribute actually mediates the connection between one participant to the other participants through the user-invisible relationship representation type in the expanded schema.

Each instance of a representation type, i. e., "representation object", describes an occurrence of the user-defined relationship. Interfaces to manipulate representation objects, such as the creation of new ones and the connection of them with participating objects, are provided by special UDRs (see below).

Implementation constructs. A user-defined relationship is implemented as an object type in its own right. Such a "relationship implementation type" or "implementation type", is composed of **participant-indicating attributes, semantics-describing attributes, semantics-ensuring routines** ("enforcers" governing semantics specified for a user-defined relationship and "monitors" scheduling the enforcers attached to an implementation type; they supply the basis for operation rewriting which preprocesses SQL DML statements to let them pay attention to the specified semantics), **traversal-mediating routines** ("mediators" leading the traversal through the participating objects in the presence of the relationship (representation) constructs; on this basis, the referencing/dereferencing mechanism is overloaded so that it can continue to work in OrientSQL just as in SQL:1999), **relationship-manipulating routines** ("manipulators" processing those OrientSQL statements that support the manipulation of relationships such as *INSERT RELATIONSHIP*; they provide the instance-level operations on representation constructs in the following forms: instantiation of representation types, retrieval/modification/deletion of representation objects, connection/disconnection of relationship representation objects and participating objects).

Implementation types are arranged within a hierarchy. At the top level of this hierarchy is the most generic type *OrientRelationship* which delivers common behavior embedded in all relationships. Specialization down the hierarchy enables subtypes to inherit or augment the structure as well as to inherit or even overwrite the behavior of the supertype, thus facilitating incremental implementation. In the hierarchy, the bottom level reflects application-specific implementations; and those at the higher levels are generic with their semantics common in several applications.

The following two subsections will address the procedural part of implementation constructs, regarding how to ensure semantics and how to traverse relationships, respectively.

3.3. Ensuring semantics

Automatically warranting relationship semantics demands the system to react on given DB operations, which can be naturally reflected using a set of descriptions based on the ECA notion. These descriptions, called "enforcement rules" [22], supply a suitable basis for constructing semantics-ensuring measures.

Enforcers and monitors. An abstract enforcement rule is "materialized" by replacing its action part with a procedural enforcer. Basically, the reactions to semantics violation are of two kinds: "rejection" and "propagation". For instance, the existence of an aggregate object causes the rejection of the removal of a part, and the removal of an aggregate causes the propagation of the removal to all its parts. When a user-initiated operation is executed, the monitoring implementation types activate enforcers. In case of rejection, the effect of a user-initiated operation must be undone. For this purpose, the transaction mechanism can be employed. The propagation processing, on the other hand, is more complicated. Propagation means manipulating multiple instances of this and other related participants as well as multiple instances of the involved relationships. Hence, the "enforcement target" such as "instances-to-be-deleted" has to be determined before the propagation. For this reason, each enforcer *eRoutine (p)* is composed of two subroutines: one (such as *eTargetDelete (p, I)*) determining the enforcement target, the other (such as *eDelete (I)*) performing the operation propagation to enforce the semantics. Since an implementation type may be equipped with a number of enforcers, they must be correctly arranged and scheduled by monitors to deliver expected semantics control behavior. For each participant, a monitor is defined. All the enforcers "related to" the participant, i. e., needed to maintain certain semantic properties that may be violated by operations issued on that participant, are included in that monitor. These enforcers are identified by analyzing the entries in the catalog *eRoutine* where all the enforcers defined for the relationship concerned are registered. A monitor includes a "checker" and a "dispatcher". The checker detects the operations on the participant that may violate semantics under the current condition. Based on the checker's output, the dispatcher then activates the enforcers encapsulated in the monitor.

Operation rewriting. Note, aforementioned routines themselves do not exhibit inherent active behavior. Explicitly invoking them from the application code would entail all the negatives of user-managed semantics. Hence, it is more reasonable to execute them implicitly. In this way, the impact of semantics-ensuring measures on regular SQL DML statements will also be minimized. For this purpose, the SQL data manipulation requests are rewritten (by OrientPre, cf. Sect. 3.5) to those that invoke appropriate monitors encapsulated in appropriate implementation types. In this way, DML operations are endowed with new meaning. A deletion operation on the module *m*, for example, is at first captured by the rewriting process and then augmented with a call to the monitor defined for *module*. After proper checking and dispatching, this monitor will invoke the subroutine *eDelete (I)*, where *I* is obtained by calculating the enforcement target using the subroutine *eTargetDelete (m, I)*.

3.4. Traversing relationships

To support OrientSQL's paths, SQL:1999's referencing/dereferencing mechanism which facilitates traversal across references should be overloaded to allow traversal via user-defined relationships. Mediators build an essential basis for this by providing the participating objects with a local view on the relationship. As mentioned previously, for each user-defined

relationship, there is a group of mediators. A mediator retrieves participating objects of one type from a participating object of another type via the representation object. Assume, from an instance (*m*) of the referencing type (*module*) to an instance (*f*) of the referenced type (*function*), there is an instance (*c*) of the representation type (*consist_of*). The expression *m.consist_of (function)* is resolved using the mediator encapsulated in the corresponding relationship implementation construct.

3.5. ORIENT prototype

Now, we build the prototype ORIENT (Object-based Relationship Integration ENvironmenT) with system-controlled use of relationships as the focus and other auxiliary specification and maintenance support around. Its architecture is outlined in Fig. 4.

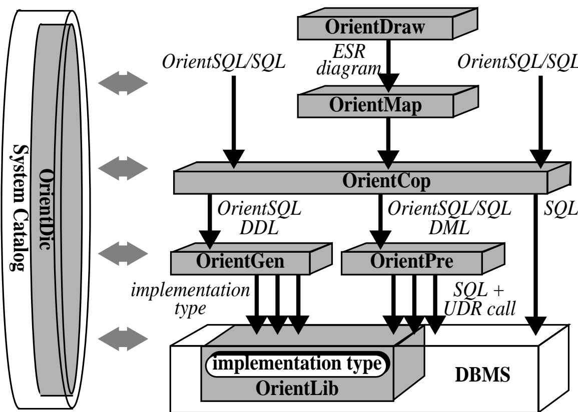

Figure 4: Architecture of ORIENT

OrientDraw: The schema editor provides an easy-to-use design interface by supporting ESR (Entity/Semantic-Relationship) diagrams [22].
OrientMap: The schema translator is responsible maps graphical elements into corresponding OrientSQL specifications.
OrientLib: The extension package contains relationship implementation types. This component is considered to be a pluggable module of the database server.
OrientDic: The metadata manager contains all metadata useful for ORIENT. It is utilized and also continually enriched by the other components.
OrientCop: The statement dispatcher is a "filter" of user-initiated requests or statements. It distinguishes statements that need ORIENT-specific processing from statements that do not. For each OrientSQL statement of the former kind, OrientCop makes a call to OrientGen or OrientPre, depending on whether that statement is for definition or for manipulation purpose.
OrientGen: The package generator converts declarative OrientSQL (DDL) specifications into operational ORDM constructs. It is composed of two submodules: ReptBuilder responsible for the creation of relationship representation types and ImptBuilder responsible for the generation of relationship implementation types. In comparison with ReptBuilder which conducts schema expansion as discussed in Sect. 3.2, the task of ImptBuilder is more complicated, especially regarding how incremental, extensible implementation is facilitated. For this purpose, OrientDic and OrientLib deliver the necessary support. OrientLib packages all the existing implementation types (organized into a hierarchy as mentioned in Sect. 3.2), from which new ones can be derived. As a complement, OrientDic provides the metadata needed for the generation process. It supplies catalogs that contain information about existing implementations: *eRule* indicating enforcement rules already considered, *eRoutine* indicating enforcers already generated, as well as *imptType* indicating implementation types and monitors already constructed.

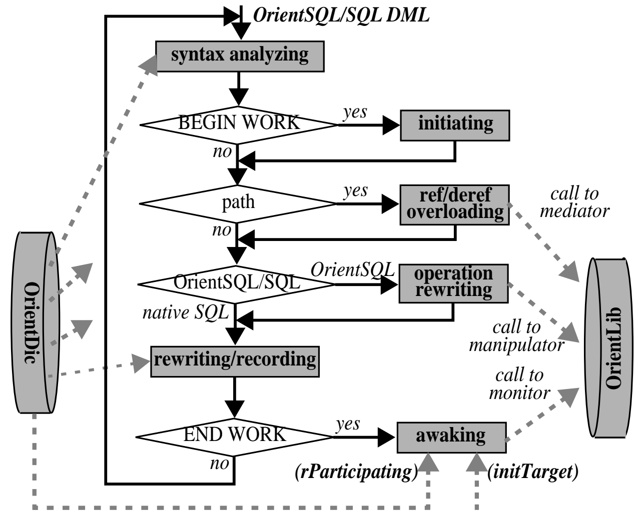

Figure 5: Processing steps of OrientPre

OrientPre: The precompiler rewrites a user-initiated request to meet the need for relationship-specific processing. It augments the user-initiated request with invocation of appropriate routines (i. e., mediators or monitors) using three kinds of transformation: The first overloads SQL:1999's referencing/dereferencing mechanism to support OrientSQL paths; the second translates OrientSQL DML operations into appropriate manipulators; and the third rewrites native SQL DML operations by activating proper semantics enforcement code. While the first two tasks can be realized almost purely through syntactic analysis, some explanation of the third is necessary. Ideally, to support the non-local nature of a relationship, an implementation type should be able to supervise the operations applied to its participants. However, there is no direct communication path for a UDR encapsulated in an implementation type to be aware of any operation performed on any participant. To solve this problem, OrientPre adds a "recording" mechanism through which information about operations issued on DB objects is collected. The result is stored in the table *initTarget*, with entries describing user-initiated operation (*OP*), target participant (*P*), and target participant instances (*p*). Then, OrientPre uses an "awaking" mechanism to choose and to call proper monitors for ensuring the relationship semantics which may be violated. The monitors are determined with the aid of a certain *rParticipating* catalog in OrientDic. This catalog

lists, for each entity in the schema concerned, all the relationships it participates in, together with the monitors defined for the given participant in the corresponding implementation types. Thus, with the help of OrientPre, relationship semantics is controlled in such a way that an implementation type is equipped with the "simulated" ability to supervise the participants. This preprocessing alone, however, places only extensions to individual DB operations. To make relationship maintenance an integral part of normal transaction processing, it is certainly not enough to take semantics-ensuring measures only as "side effects" of single DB operations. Rather, a user-initiated transaction as an entirety should be considered as the rewriting granularity and consequently the enforcement granularity. Generally, a DB transaction is composed of a sequence of user-initiated operations which must be applied to the database as an atomic unit. Having this in mind, the rewriting is carried out in a "bulk" fashion: Until the last command within a user-initiated transaction is encountered, the enforcement code is not constructed, but rather "accumulated" by OrientPre, which then works as in Fig. 5.

3.6. Remarks

Taking the commercial ORDBMS IDS/UD [11] as implementation platform, the UDRs together with UDTs that are encapsulated in OrientLib constitute a DataBlade which can be installed into or removed from IDS/UD.

Concerning extensibility, obviously, using an extension package like DataBlade offers a way to enhance the DBMS functionality. However, registering OrientLib as a DataBlade in the system only means that the system is made aware of its existence. No customization of the internal DBMS implementation is possible. Deviations from the expected functionality are inevitable.

In our setting, some tasks desired from the underlying DBMS cannot be realized or have to be simulated. For example, overloading the referencing/dereferencing mechanism only resembles a kind of view substitution, rather than deeply-integrated processing of relationships. Moreover, although registered UDRs decrease the communication overhead across the DBMS interface, this is only the case for seamless usage of extensions. The implementation of ORIENT resorts to the precompiler OrientPre to invoke appropriate UDRs, which implies a layer of indirection between ORIENT and the DBMS.

Furthermore, the "bulk" rewriting approach taken by OrientPre seems to comply with the insert block concept which requires to delay the enforcement of insertion semantics. However, the way OrientPre processes an insert block, the only possible action when the insert block must be rejected is to let the whole transaction embodying that block be rolled back. True block atomicity cannot be obtained. To provide the desired failure handling semantics and to avoid the complete transaction rollback, nested transactions or user-defined checkpoints are needed, but, unfortunately, are not supported by IDS/UD.

Another problem is the interaction of declarative queries with UDRs that embody relationship processing, whose optimization raises a question as how to convey knowledge about relationships and their semantics to the system. Current ORDBMSs cannot deliver a satisfactory answer, since they treat UDRs as "black boxes". While ORDBMSs optimize relational operations like *JOIN*, there is hardly any optimization for UDRs, since the system understands little semantics about each UDR. A UDR is merely a name or a signature, with minimal information such as side effects or user-speculated costs. Due to the closed system architecture, it is a delicate task to preprocess OrientSQL in such a fashion that the resulting query is not "misunderstood" by the query optimizer. To this end, sound knowledge of the underlying system, particularly of its query optimizer, is imperative. But even then the DBMS may be too "dumb" for passing its valuable optimization information.

Therefore, the presented approach is still an "on top" one, in that it is outside the DBMS engine and, consequently, outside all its underlying constituents. This solution (and probably not only for ORIENT) renders a new "application layer" in between the DBMS and the real application. In the long run, we hope, the improvement of the OR extensibility as well as the evolvement of our prototype will lead to a more satisfactory result. Particularly, when more knowledge of relationships is conveyed to the system, their entailed potential for a more dedicated implementation could be well utilized. But at present, the fundamental question is still: What is the necessary infrastructure to achieve real extensibility?

4. Deficiencies of (OR)DBMS Extensibility

The previous discussion reveals that integrating our relationship concepts would be less painful and more effective, if some adequate facilities could be provided by the underlying system. DBMS extensibility addresses the need to get a DBMS offering enough possibilities to adapt or expand its functionality, so that special processing could be better incorporated with the existing mechanisms. We have also observed that the current ORDBMSs are successfully presenting some degree of extensibility. However, the actual exploitation thereof often runs up against its limit. Therefore, this topic still deserves our further investigation. Let us at first review the efforts (especially, those of the ORDB technology) that intend to make DBMSs more or less extensible.

Since the mid-eighties, several research projects have been dealing with extensibility [4]. Some of them select a specific data model (mostly relational) and implement interfaces through which extensions can be added. Usually, they open up the type system to incorporate more complex data types [20]. Abstract data types (ADTs), their functions, and possibly their access methods can be defined by the user. Once registered with the database system, an ADT (ADT is termed differently in different places, e. g., UDT or structured type in SQL:1999 [21] or opaque

type in IDS/UD [11]) is used just like a built-in type. This approach is pioneered by the ADT-Ingres project, carried on by the Postgres project [20], and now followed by the ORDB technology [11]. Others such as Exodus [7], Genesis [1], and Open OODB [3] strive for more generality by supplying a set of kernel facilities plus "toolkits" for constructing domain-specific DBMSs, but toolkits did not gain general acceptance [5].

Driving the relational database technology in the direction of object orientation, current (ORDBMS) products provide support for two kinds of "objects", ADTs and row types. Row types offer a direct enhancement of the type system for relation tuples. In comparison, the role of ADTs, à la Postgres [20], is to enable the set of built-in types to be expanded with the UDTs. In this sense, the ADT concept embodies more extensibility and reflects a big step forward from the "BLOB" approach used by relational systems to support complex data. Currently, vendors or third-party developers are marketing ready-made, ADT-based extension packages (e. g., DataBlades [11], DataCartridges [15], and Extenders [10]) for managing complex data such as text and image.

Current ORDBMSs follow a "table-driven" approach [20] in regard to the ADT concept, i. e., they are able to "recognize" the newly-added extensions and, this way, allow users to integrate them into "normal" processing. However, with the table-driven mechanism, additional features are treated by the system just like the existing ones. Extensibility demands calling for more changes of processing logic, unfortunately, cannot be satisfied, as to be exemplified below.

Extensible query optimization. Considering, e. g., the first OrientSQL query given in Fig. 2, the transformation performed by OrientPre (cf. Sect. 3.5) results in several extensions such as the UDT defining the relationship implementation type and UDRs representing traversal-mediating routines (Sect. 3.2). Traversal mediation introduces path elements (i. e., relationships and participants) of an OrientSQL path into processing scope, thus permitting to traverse from one participant to another via existing relationships. This, however, facilitates only naive following of a relationship structure, which is not always optimal when computing an OrientSQL path. As indicated in [22], query processing would be more efficient if the optimizer could recognize the optimization potential embodied in the path, analyze feasible permutations of the path computation sequence, and rewrite the path into joins if necessary. For this purpose, it is essential to represent an OrientSQL path as a sequence of new operators which make all inter-object relationships explicit and to extend the optimizer by additional operator-shuffling transformations. Besides, ORIENT relationships are not merely descriptions of structural connections, they also possess a lot of semantic meaning which may have considerable influence on query optimization. Particularly, special composition properties not only determine the construction of a complex object and sharing of part objects, but also characterize the transitive propagation of operations in a (possibly multi-level) hierarchy. Hence, they can play an important role in query optimization if well exploited. Taking the second query in Fig. 2 as example, the owner (e. g., *module*) and other participants (e. g., *function* and *procedure*) are expected to be used together in one query to build up the complex objects. In [22], it is specially treated by means of a new operator which brings all the target objects of a propagated operation into the processing scope. Regrettably, it is difficult for current ORDBMSs to correctly optimize (expensive) UDRs. Although the developer can provide simple information about user-defined extensions to influence query optimization and the optimization rules can access certain system tables to get this information, more semantic knowledge (such as that about explicitly specified relationships and their semantics) cannot be exploited. Moreover, although UDRs offer theoretically unlimited possibilities to extend the operations available for data, they cannot provide extensibility at the level of operators or algorithms. The reason is that, in the table-driven way, the query optimizer is unable to get additional strategies for the processing of new features.

Extensible access method. Another shortcoming of current extensibility mechanisms occurs, when special relationship indexes or path indexes containing condensed information about inter-object references are expected to accelerate the construction of complex objects or the traversal in a relationship structure [22]. While generalized B-trees in some ORDBMSs make B-trees extensible with respect to the data types that can be indexed, they do not allow to define all kinds of index structures such as those across more than one type or table.

Extensible transaction model. Yet another shortcoming is evident, when trying to support the insert block concept introduced in Sect. 2.. Here, an extensible or at least flexible, hierarchical processing concept is needed to support atomicity at block level, to facilitate error reporting related to the block granule, to provide block-internal recovery, as well as to conduct multi-statement optimization which considers the set of statements contained in the same insert block as a unit. An extremely important topic in this context is about the transactional boundaries. The associativity of operators influences where those boundaries can be put. Especially, the use of an insert block inside a normal transaction requires dedicated "scope control" so that semantics checking is delayed only to the block end when all relevant insertion operations are executed. Since SQL (and also most of the commercial DBMSs) guarantees only statement atomicity, an effective solution is the use of nested transactions to support the fine-tuning of the control/rollback scope [22, 23]. When the deferred checks do not reveal a violation of the specified semantics, the block concept behaves just like SQL's deferred constraint checking mode. In case of an integrity violation, the subtransaction bracketing the insert block is rolled back, thereby providing the desired failure handling logic. All this implies that existing rules of drawing transaction borders might have to be modified upon introducing new operators for flexible transaction control. According to Sect. 3.3, data update requests are rewritten to take semantics-en-

suring operators into account. This processing, however, only retains SQL's statement atomicity in such a way that a user-initiated DB operation will still be logically atomic in spite of several physical operations involved for semantics-control purposes. In case the block atomicity is needed, transaction semantics should be respected during internal rewriting. What is needed are not only additional operators for maintaining relationship semantics, but also additional operators for providing adequate transaction support. However, in available ORDBMSs, it is almost impossible to provide extensible transaction models.

In a word, when pursuing a deep integration of our relationship concept that goes far beyond the scope of data types, changes to the internal processing are indispensable. Unfortunately, many extensions that are required for processing relationship-specific characteristics cannot be facilitated by the current OR extensibility.

5. Conclusions and Outlook

Obviously many database applications demand for an enriched relationship support encompassing adequate modeling constructs as well as semantics enforcement mechanisms. In this paper, we presented a model fulfilling this task and discussed the corresponding ORIENT prototype which exploits the extensibility features of current generation ORDBMSs. Regretfully, we have to realize that current extensibility features do not allow to effectively integrate our enriched relationship processing facilities into the DBMS server, because there are too less possibilities of tailoring internal processing, e. g., query optimization and access methods, to the specific properties of the extensions needed, and of associating appropriate processing units, e. g., 'sub-transactions'. Thus, this paper demands for a better extensibility infrastructure, ORDBMS have to be equipped by. Certainly, we have to admit that we are not quite sure about how much extensibility is good. The success stories of ORDBMS technology already demonstrates the importance of making a compromise between two extremes of extending DBMS: the "all-by-yourself" way versus the "encyclopedic" way. The former is impractical, since the user has to develop too complex code and needs too much knowledge of the DBMS internals. The latter, on the other hand, is far less flexible and requires the DBMS vendor to continually modify its product to accommodate new types. The current OR-DBMSs, combining benefits of both approaches, allow third party "experts" to provide special type libraries that might be needed. Pre-defined or third-party extensions such as DataBlades are offered as building blocks for application development. Such a practical strategy has largely accelerated the acceptance of the ORDB technology. Nevertheless, regarding the deficiencies observed, we are convinced that extensibility features have to be improved. In the following, we outline our vision of extensibility by discussing some, as we think, crucial issues.

Modular system architecture. Nowadays, modular architectures are prevailing. In the database area, some approaches try to open the architecture of a DBMS by dividing it into a collection of modules that carry out different functions. These modules can be added or deleted with well-defined, localized effects on other modules. This allows to extend database functionality in a flexible way. ADTs or DataBlades are built modularly [20]. That means, an ADT and its routines can be added to or removed from the DBMS without affecting the rest of the system. However, as stated previously, such an OR approach relies on assumptions about the architecture and design of the DBMS into which ADTs or the like are plugged. The system architecture is not affected as a whole. Recently, more extensibility than that of ADTs is pursued by an enhanced notion of ADTs, that is, E-ADTs [19]. Each E-ADT is able to define its own declarative language, query optimizer, catalog management, etc. Built on top of a layer of common database utilities such as persistent storage and concurrency control, E-ADTs are "loosely-coupled" modules in that there is little interaction between them. Hence, it is possible to plug in a new E-ADT or take out an existing one without adversely influencing other data types in the system. A more general way is followed by some earlier prototypes such as Open OODB [3]. Extensibility is facilitated, e. g., in its query optimizer, through the separation between different submodules (such as between algebraic operators and execution algorithms) which allows exploration with alternative methods for implementing certain facilities (such as algebraic operators). Nevertheless, it seems still unclear how and by which mechanisms an open architecture can be conveniently tailored to a concrete and efficient DBMS. Therefore, up to now, no adequate architectural framework for DBMSs is known. Designing such a framework is worth further study.

Common internal interface. The key to extensibility in an open architecture lies in how interfaces are designed between modular components. Common internal interfaces provide the foundation upon which specific extensions can be defined and implemented. Therefore, extensible systems call for a clean and comprehensible internal structure. In extended relational prototypes [20] and ORDBMSs [11], each ADT implements a common interface through which the system can access and manipulate ADT instances. The interface includes functions for the storage and indexed retrieval of ADT instances as well as methods for manipulating or querying ADT instances. Therefore, a package containing the definition and implementation of one or more ADTs can be plugged into or removed from the system. However, the OR extensibility is targeted to the type system only. Actually, besides complex data types, semantically rich relationship concepts are also a typical sort of extension which can be added, like version facilities and active features. Their integration, going beyond the scope of data types, is much more challenging. Reasonable access and changes to the existing system constituents

are essential. Hence, a more open solution to common interfaces than what ORDBMSs offer still needs to be pursued.

Proper abstraction. Generally, while a "black box" abstraction (through, e. g., functional signature interfaces) might be suitable for developing applications where code can be reused without relying on anything else but the specification, reasonable extensions to a DBMS demand a controlled part of its processing logic to be revealed, thereby leading to a "grey box" abstraction. We consider two issues of great importance, metadata and internal rule set. Both possess the same capability to give things that will change a handle that does not change. Postgres [20] and its OR successors [11] adopt registration and cataloging mechanisms to add UDTs, UDRs, as well as access methods and to supply metadata on type-specific operations and access methods. The system constituents can then "react" to the extensions in a table-driven fashion. With user-supplied code also serving as a procedural specification of the evaluation strategy, each ADT is merely a "black box" [19] (some existing products such as IDS/UD [11] also allow the specification of simple semantics such as side effect or cost estimation, so that optimized access plans for queries involving the new type can be generated. However, such a facility is far from general enough to deal with arbitrary routines having arbitrary behavior). Therefore, it is still a crucial issue in the OR context how to expose internal processing logic adequately. Furthermore, to incorporate the relationship support fully in the system, the registration mechanism and integrated storage of metadata should accommodate more semantic implications of extension-describing information. And the whole DBMS processing (such as query transformation, transaction control, and semantics enforcement) should be accompanied by that information. Internal rules provide another flexible mechanism for expressing the behavior of system facilities and for responding to changes at different levels. But until now, most of the work that reveals internal processing logic by means of rule-based techniques concentrates on the addition of new query transformations and new physical operators [8]. For most of the systems and prototypes, rule usage at the application level does not guarantee that rules can be employed at the system level as well. Therefore, it is important to have the rule support available in the DBMS and usable to modify the internals of the system itself. For ORIENT, especially, rules can be employed to supply the guidelines for transforming queries, setting transaction boundaries, dispatching semantics enforcement measures, and so on.

In summary, extensibility is a matter of degree and up to now subjected to a certain expertise. The doubts about extensibility with respect to security, stability and performance should inspire us to do better instead of giving up. We believe, the object-relational approach presents a good reference point to go further.

References

[1] Batory, D., Barnett, J. R., Garza, J. F., Smith, K. P., et al.: GENESIS: An Extensible Database Management System, IEEE TSE 14:11, 1988, 1711-1730.

[2] Batini, C., Ceri, S., Navathe, S. B.: Conceptual Database Design: An Entity-Relationship Approach, Benjamin Cummings, 1992.

[3] Blakeley, J. A., Mckenna, W. J., Graefe, G.: Experiences Building the Open OODB Query Optimizer, Proc. 1993 ACM SIGMOD Conf., Washington, D. C., May 1993, 287-296.

[4] Carey, M. J. (ed.): IEEE Data Engineering Bulletin 10:2, Special Issue on Extensible Database Systems, 1987.

[5] Carey, M. J., DeWitt, D. J.: Of Objects and Databases: A Decade of Turmoil, Proc. 22nd VLDB Conf., Mumbai, India, Sept. 1996, 3-14.

[6] Díaz, O.: The Operational Semantics of User-Defined Relationships in Object Oriented Database Systems, Data & Knowledge Engineering 16 (1995), 223-240.

[7] Graefe, G, DeWitt, D.: The EXODUS Optimizer Generator, Proc. 1987 ACM SIGMOD Conf., San Francisco, California, May 1987, 160-172.

[8] Haas, L., Chang, W., Lohman, G., McPherson, J., et al.: Starburst Mid-Flight: As the Dust Clears, IEEE TKDE 2:1, 1990, 143-160.

[9] Härder, T., Mahnke, W., Ritter, N., Steiert, H.-P.: Generating Versioning Facilities for a Design-Data Repository Supporting Cooperative Applications, Int. Journal of Intelligent & Cooperative Information Syst. 9:1-2, 2000, 117-146.

[10] IBM DB2 Universal Database (Version 6.1), IBM Corp., 1999.

[11] Informix Dynamic Server Documentation, Informix Software, Inc., 1998.

[12] Jaedicke, M., Mitschang, B.: User-Defined Table Operators: Enhancing Extensibility for ORDBMS, Proc. 25nd VLDB Conf., Edinburgh, Scotland, Sept. 1999, 494-505.

[13] Kim, W., Bertino, E., Garza, J. F.: Composite Objects Revisited, Proc. 1989 ACM SIGMOD Conf., Portland, Oregon, June 1989, 337-347.

[14] Mattos, N. M.: Abstraction Concepts: the Basis for Data and Knowledge Modeling, Proc. 7th ER Conf., Rom, Italy, Nov. 1988, 331-350.

[15] Oracle8 Resource Page, Oracle Corp., http://www.oracle.com/st/products/uds/oracle8/.

[16] Peckham, J., Maryanski, F.: Semantic Data Models, ACM Computing Surveys 20:3, 1988, 153-189.

[17] Rumbaugh, J.: Relations as Semantic Constructs in an Object-Oriented Language, Proc. 2nd OOPSLA Conf., Orlando, Florida, Oct. 1987, 466-481.

[18] Stonebraker, M., Brown, P.: Object-Relational DBMSs — Tracking the Next Great Wave, 2nd ed., Morgan Kaufmann, 1999.

[19] Seshadri, P.: Enhanced Abstract Data Types in Object-Relational Databases, The VLDB Journal 7:3, 1998, 130-140.

[20] Stonebraker, M., Kemnitz, G.: The POSTGRES Next Generation DBMS, CACM 34:10, 1991, 78-92.

[21] ANSI/ISO/IEC 9075-2-1999: Information Technology — Database Languages — SQL — Part 2: Foundation (SQL Foundation), Sept. 1999.

[22] Zhang, N.: Supporting Semantically Rich Relationships in Extensible Object-Relational Database Management Systems, Doctoral Thesis, Dept. of Computer Science, Univ. of Kaiserslautern, Sept. 2000.

[23] Zhang, N., Härder, T.: On a Buzzword "Extensibility" — What we have learned from the ORIENT Project, Proc. Int. Database Engineering and Applications Symposium (IDEAS'99), Montreal, Canada, Aug. 1999, 360-369.