To appear in Proc. International Database Engineering and Applications Symposium (IDEAS'99), Canada, Aug. 1999.

On a Buzzword "Extensibility"
— What We Have Learned from the ORIENT Project?

Nan Zhang, Theo Härder

Department of Computer Science
University of Kaiserslautern
67653 Kaiserslautern, Germany
e-mail: {zhang | haerder}@informatik.uni-kl.de

Abstract

To meet the ever challenging requirements of today's applications, both database research and industry are being engaged in adding new features to database management systems (DBMSs). As a major effort in this direction, object-relational DBMSs (ORDBMSs) gain more and more popularity due to their extensibility, i. e., allowing user-defined facilities to be used in the same way as built-in ones. Inspired by this idea, we started to develop a prototype ORIENT to enhance DBMSs with complex relationship semantics that are versatile in the real world, but still insufficiently supported by the new database technology. Our work reveals that data model changes to accommodate relationship semantics imply extensions at the processing level, which are, however, not feasible in "extensible" ORDBMSs. Proper integration of new processing capabilities calls for a more open approach to database extensibility than what the current ORDBMSs exhibit.

1. Introduction

The good old days when database systems were used for storing flat data with unsophisticated operations are over. Today's database systems must do much better. They must be extensible to meet the ever challenging demands from different application domains. Since the mid-eighties, a number of research projects have had extensibility in mind [5]. Some of the efforts have come into commercial systems with success stories, e. g., current ORDBMSs [8, 9, 13]. In contrast to the classical way of defining a supplementary layer on top of the system, i. e., through application programming interfaces, the object-relational approach pursues a closer integration of the new functionality. Typically object-relational extensions are based on user-defined types (UDTs) and user-defined routines (UDRs) in the simplest form, scalar functions. More expressiveness is brought about by table functions with more complicated behavior. Is it feasible to go this way further to integrate even powerful functionality, e. g., version management, access to heterogeneous data sources, or refined control of data relationships, to name a few?

To explore database extensibility, we focus our effort on the semantic enrichment of relationships which are of great importance in many applications, but only insufficiently supported: ORDBMSs try to assimilate object relationships by providing a simple pointer-based reference mechanism [2]. References reveal little semantics. The user is forced to implement the required support manually. Meanwhile, the value-based relational approach considers relationship semantics to some extent, but not thoroughly. The integrity enforcement mechanisms, such as referential integrity constraints, *CHECK* constraints, and triggers are neither powerful enough to cover all semantic aspects of relationships existing in the real world nor general enough to offer a concise and uniform solution [24].

Some attempts (e. g., [16]) try to improve the relationship support. But their consideration is more or less limited: Either a few special relationship types or general relationship types with only simple characteristics are addressed. While several prototypes have been built by these attempts, theoretical or formal approaches are rarely realized in any system. Hence, we believe that the support of relationship semantics deserves further and more detailed attention. And most importantly, this work should be put to practical use through the integration with promising and frequently used data models, including the entire spectrum of object-relational ones.

To this end, we are developing a prototype ORIENT (Object-based Relationship Integration ENvironmenT) [24]. Our main concerns are to support explicit specification and automatic maintenance of semantically rich relationships and, at the same time, to investigate the way how to seamlessly integrate them into a DBMS. The fundamental questions include: While extensions at the level of UDTs and UDRs can be achieved rather easily in ORDBMSs, does the fact also hold for more complex functionality? What is the desirable extensibility infrastructure?

The remainder of this paper is organized as follows. In Sect. 2, we will briefly discuss ORIENT's data model extensions referring to examples in a software repository scenario. Then the new features will be reflected at the language level. Sect. 3 highlights the impacts that the data model extensions will have on the DBMS processing and its underlying constituents. On this basis, we will investigate the difficulties in integrating concepts which go beyond the UDT scope. Afterwards, we will describe in Sect. 4 our attempts to implement ORIENT using a public-domain and a commercial ORDBMS, respectively. Both approaches, with their own merits and deficiencies, underline the fact that the extensibility demand posed by the integration of semantic relationships exceeds what the current ORDBMSs can supply. Therefore, in Sect. 5 we will present our reflections on what a desired extensibility infrastructure means. And finally, we will conclude the paper in Sect. 6.

2. ORIENT relationship facility

Our primary goal is to take a simple but general solution to accurately model relationships and their semantics, instead of resorting to indirect means such as complex *CHECK*s, triggers, or progam code.

To facilitate the discussions, we take a running example from a software repository scenario, where an application program consists of modules which, in turn, are composed of functions and procedures. All these constituents are represented as components in a repository. Components are further divided into packages according to the specifications they realize. Furthermore, programs and modules may be under continuous evolution, which may produce new versions. Fig. 1 shows the scenario with ER-like notations.

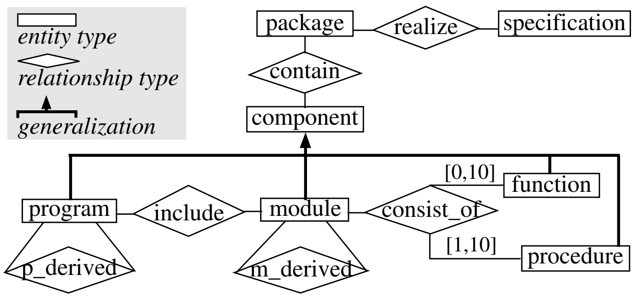

Fig. 1. Application scenario

2.1. Data model extensions

Tab. 1 outlines the main conceptual dimensions along which we extend object-relational data models (ORDMs) to accommodate more relationship semantics, in contrast to the expressiveness of relational primary/foreign keys (PK/FK) and object references, respectively. Our purpose is not to prove the comprehensiveness of ORIENT's relationship concepts (indeed, due to the space limitation, some aspects such as relationship attributes are omitted), but to show the adequacy of the semantic enrichments as well as to lay the baseline for the subsequent discussions.

Symmetry: Unlike directed references, PK/FK relationships can be accessed symmetrically. However, the referential integrity controls only the direction from FK to PK.[1] ORIENT relationships provide not only structural, but also integrity symmetry by defining semantics between associated participants in bidirectional relationships.

Composition: This property determines how the participants of a relationship cohere. In composite relationships (e. g., *aggregation* [11]), there is a special participant (composition owner), from which operations affecting the whole relationship are propagated to other participants.

Sharability: In case of composition, sharability determines whether an object can participate in more than one relationship with another object at a higher level. For example, if a function belongs to only one module, then it is non-sharable in *consist_of*, otherwise, it is sharable.

Degree: A reference type describes a binary relationship between a referencing and a referenced type, just as a PK/FK pair integrated into associated types. ORIENT takes a more general approach by representing an n-ary relationship directly as an atomic unit.

Cardinality: The cardinality of a relationship places restrictions on the number of related objects. Relationships expressed through PK/FKs are basically [0,1]:[0,n]. The exact number of rows involved in a relationship is not specified. Improvements are possible through *NOT NULL, PENDANT*, or *CHECK* constraint. In ORDMs, although collection types are useful for (1:n)- and (m:n)-relationships, finer cardinality restriction can only be simulated with complex *CHECK*s, triggers, or user code. ORIENT provides a general solution in the form of, e. g., [1,1]:[2,10].

Partial Relationships: Partial relationships are needed to represent unknown participants.While it is impossible to express unavailable object identities, the relational modeling of missing information employs dangerous *NULL* values [6]. ORIENT allows some participants to be absent.

Scope: The referential integrity indicates the parent table exactly, with however no exact "dereferencing" [1]. Since a single ORDM type can be reused to define several tables [2], it is sensible to restrict a relationship to certain table(s) for the sake of precise modeling as well as dereferencing.

Transitivity: Referential actions can control so-called existence dependencies, i. e., a certain insertion or deletion should be automatically propagated to related rows. It is also useful to have (possibly heterogeneous) transitivity for other database operations. For example, in an *aggregation* relationship, selections at the aggregate level should be propagated to the part level. And the relationship *m_derived* means that any update of a module should result in the insertion of a new version.

Relationship Hierarchy: A brand new feature in ORIENT is that relationships are organized into a hierarchy

1. Preliminary versions of the SQL standard proposed to achieve integrity symmetry by *PENDANT* option that requires the existence of at least a child row for the corresponding parent row. But this proposal is neither adopted by SQL99 nor implemented by any system.

Table 1. Data model extensions

	PK/FK	Reference	ORIENT Relationship
Symmetry	only structural symmetry	✗	both structural and integrity symmetry
Composition	✗ (simulated)	✗ (simulated)	explicit, concise specification
Sharability	simulated (*CHECK* or trigger)	simulated	explicit, concise specification
N-arity	basically binary; n-ary by separate table	binary	n-ary relationship as an atomic unit
Cardinality	[0,1]:[0,n]; improved by *NOT NULL*/*CHECK*	collection; *CHECK*	general solution for exact restriction
Partial Relationship	special purpose *NULL*s	✗	absent participant(s)
Scope	parent table in referential integrity; no precise "dereferencing"	scoped	*SCOPE* restricting a relationship to certain participant table(s)
Transitivity	existence dependency: homogeneous propagation of insertion/deletion	✗	homogeneous and heterogeneous propagation of all operations
Relationship Hierarchy	✗	✗	incremental modeling/implementation

to facilitate incremental modeling and implementation. This is motivated by the observation that on one hand, there are relationship types with generic, reusable characteristics, and on the other hand, relationship types may be tailored to specific applications. For example, *consist_of*, a special kind of *aggregation* [11], specifies that a module (aggregate) and all its functions and procedures (parts) exist as a whole and can be addressed collectively.

2.2. OrientSQL

Now we illustrate how the data model extensions are reflected at the language level. To conform to the current database standard, we provide SQL99-like OrientSQL for specification as well as manipulation purposes [19, 24].

2.2.1. Relationship definition

Fig. 2 shows the *CREATE RELATIONSHIP* statement (generated from graphic inputs by the design tool *OrientMap* [23]) specifying *consist_of*. Structurally, *module* plays a special role as composition owner with cardinality [1,1], whereas *procedure* and *function* are non-sharable participants with their own cardinality. Moreover, the scope of *consist_of* is limited to table *module_project1*. As to operational semantics, selecting a module should render all functions and procedures related to it, and deleting a module causes the deletion of related functions and procedures. A module cannot be constructed if any function or procedure is missing. In contrast, functions or procedures can be added even though the module does not exist yet. In this case, the partial relationship is established with a placeholder for the absent module.

2.2.2. Establishing relationship instances

The *INSERT INTO RELATIONSHIP* statement acts as a "multi-connect" construct. Here, we have to distinguish three cases, depending on the specific meaning of the relationship (cf. Fig. 3a) and the existence of participants.

In the first case when all participants already exist, a single statement establishes the relationship.

In the second case, the existence of some participants is optional (CONDITIONAL INSERTION ***CI***). *INSERT INTO RELATIONSHIP* will be always executed regardless if any participant is missing. The absent participants will be represented as placeholders ("stubs", cf. Fig. 3b).

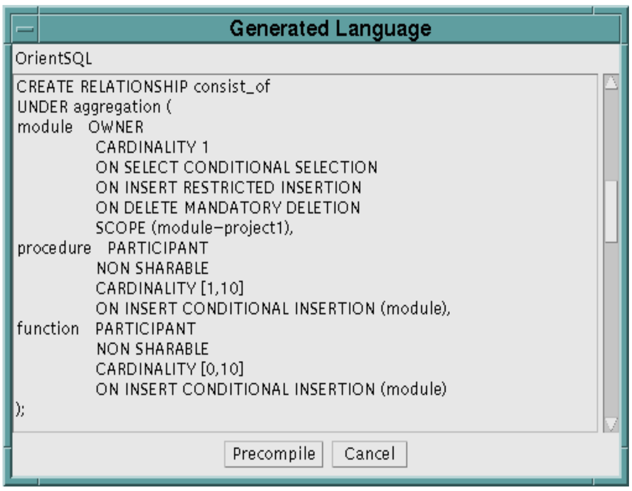

Fig. 2 Relationship definition

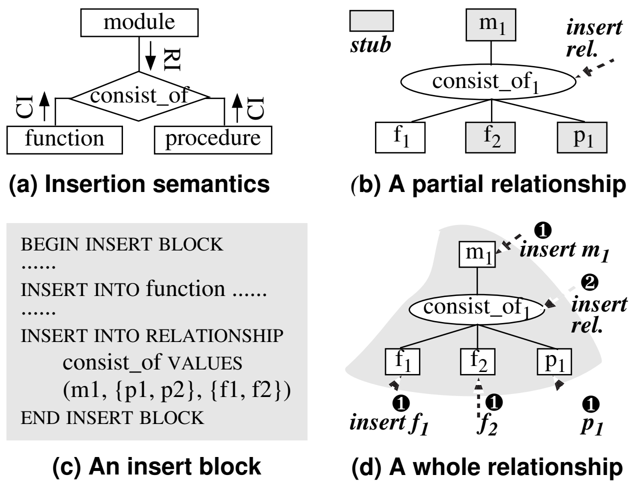

Fig. 3 Relationship establishment

In the third case, the existence of all participants is mandatory (RESTRICTED INSERTION ***RI***). Before the relationship multi-connect (❷ in Fig. 3d), native SQL statements are needed to insert the participants (❶ in Fig. 3d). To facilitate the consistent execution of multiple insertions, an explicit *INSERT BLOCK* mechanism is introduced to delay the integrity checking of single operations until all relevant operations have been executed (cf. Fig. 3c).

2.2.3. Querying

We illustrate only some special cases occurring in the new context. In Fig. 4, the first example expands the SQL99 referencing/dereferencing mechanism to allow traversal across the ternary relationship *consist_of* (cf. Fig. 1). The explicit indication of *function* helps to correctly access the attribute *language* in (multiple) functions (rather than procedures) involved in a *consist_of* instance.

The second query renders qualified complex objects (*module*) and their associated parts (*function* and *procedure*) together. To avoid the ambiguity caused by the fact that each participant may be involved in more than one relationship and various relationships may possess different selection semantics, the query is restricted in a certain relationship (i. e., *consist_of* in the *FROM* clause). Here, we can observe that the composition property makes the query syntax simple, but at the same time, more complex processing logic is required.

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'
```

Fig. 4 Querying examples

2.2.4. Modification

Due to the space limitation, we will give no example here. Note that ORIENT simplifies the user's work by providing usual modification mechanisms such as *DELETE module*, which will cause the automatic deletion of related functions and procedures. To accomplish transitive propagation, certain processing steps need to be adapted.

3. Required concepts and mechanisms

In this section, we look at the required concepts and implementation techniques to integrate the desired functionality outlined so far into DBMSs. Our discussions will be restricted to OrientSQL processing phases, with other facilities briefly mentioned when necessary.

3.1. DDL processing

The main task of DDL processing is to transform declarative specifications into metadata that aids the system to handle data reasonably. OrientSQL provides relationships as separate modeling constructs, which implies extensions of the DDL syntax for creating, dropping, as well as altering relationship types and/or semantics. As a consequence, the system catalogs must be expanded to accommodate relationship-specific metadata at two levels: New type-level catalogs contain information about relationship types (*relationship*), semantics specified between two participants (*reference*), as well as semantics valid for single participants (*participant*). For each relationship type, a separate instance-level catalog describes connections and semantics of its instances. Fig. 5a depicts the new type-level catalogs, whereas Fig. 5b sketches the instance-level catalog *rel_consist_of* for the relationship *consist_of*. We use informal phrases to describe the entries and arrows to illustrate some inter-relationships. Readers are referred to [19] for more details of the metadata structures in ORIENT.

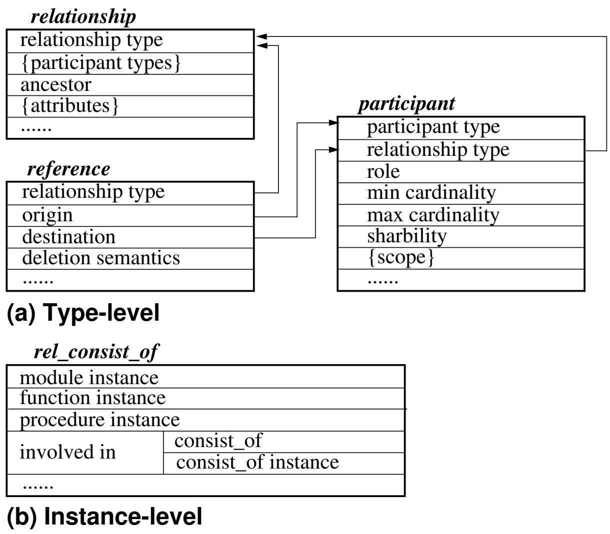

Fig. 5 New system catalogs

3.2. DML processing

On one hand, ORIENT's extensions facilitate more natural modeling and help to avoid complex operations (cf. Sect. 2.2), on the other hand, they also add complexity to the overall DML processing. The most important issues are how to handle data according to the specified semantics and how to use special relationship semantics to deliver satisfactory efficiency. OrientSQL processing goes through the same phases as usual SQL processing [7, 10]. Depending on how deep the integration is aspired, a growing number of processing phases is affected, calling for different levels of "extensibility". In the following, we will briefly discuss the issues to be considered in these processing phases to accommodate ORIENT's extensions.

3.2.1. Parsing and validation

At first, an OrientSQL query is parsed, validated, and then standardized into an internal query representation (IQR). Fig. 6[1] gives the IQRs for the two queries in Fig. 4.

Since these steps can find strong correspondences in existing DBMSs or require only intuitive extensions, we will skip them and concentrate on the follow-up steps with more ORIENT-specific problems.

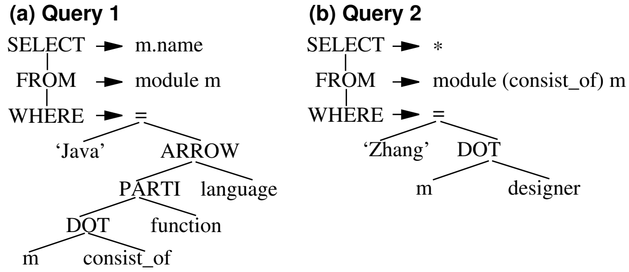

Fig. 6 After parsing

3.2.2. Query rewriting

Next, the query is rewritten for optimization purposes. In the second example (cf. Fig. 4), the semantics of *consist_of* implies that the query against *module* should also retrieve *function* and *procedure*. Retrieval hidden in the *FROM* clause should be made explicit. We call this processing step *relationship expanding*.

Relationship expanding can be conducted syntactically, in that OrientSQL queries are simply replaced with native SQL ones. The second query can be mapped to Fig. 7, with the help of internal relationship metadata (i. e., *rel_consist_of,* cf. Sect. 3.1). Such a direct translation is easy to realize. The subsequent processing steps need not be changed, thereby allowing reuse of other original DBMS facilities. However, the translation reveals no possibility to address relationship-specific handling and optimization. Thus, deeper integration of ORIENT's functionality requires more extensions of the DBMS processing logic!

Relationship expanding means that before actually performing the operation, the target objects must be selected. For a non-transitive relationship, any operation on participating types can be confined to these types. However, if the operation is of transitive nature, the selection must comprise all potentially affected participating types.

```
SELECT  m.*, p.*, f.*
FROM    module m, procedure p, function f, rel_consist_of r
WHERE   m.designer = 'Zhang' AND r.module = m AND
        r.procedure = p AND r.function = f
```

Fig. 7 Direct translation (Query 2)

1. The notations are in the style of those used by [14]. Here, *DOT*, *ARROW*, and *PARTI* are corresponding to the referencing construct ".", the dereferencing construct "*->*", and the participant-resolving construct "*( )*" in the OrientSQL queries (cf. Fig. 4), respectively.

Moreover, they might have to be treated differently from the initial triggering operation (in case of heterogeneous propagation). We introduce a generic operator *GET* $(R, T_i, O_i, \{T_p\}, \{O_p\})$ to capture the following details:

- Initial target T_i: What are the initial target objects?
- Initial operation O_i: Which operation must be performed on T_i?
- Is the relationship *R* transitive?
- Propagated targets $\{T_p\}$: For a transitive relationship, according to which criteria can connected participants be discerned?
- Propagated operation $\{O_p\}$: For a transitive relationship, on each participant in $\{T_p\}$, which operation must be performed?

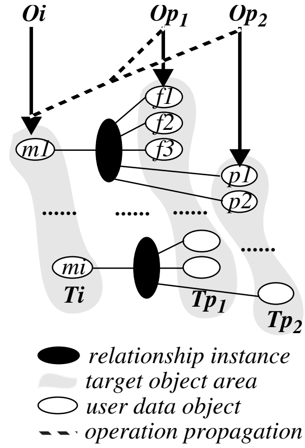

Fig. 8 GET ()

As an example, Fig. 8 graphically illustrates that querying the complex object *module* may render not only qualified modules themselves (e. g., m_1), but also all their associated functions (f_1, f_2, f_3) and procedures (p_1, p_2). Consequently, transformations for rewriting IQRs (and also for optimizing the resulting IQRs) can take advantage of this operator. The second query can be mapped to Fig. 9b.

Now look at the first query, which uses an OrientSQL path expression *m.consist_of (function) -> language* (cf. Fig. 4). Three operators are involved: The referencing construct "." represents the retrieval from *module* via *consist_of*; the participant-resolving construct "*( )*" indicates *function* to be queried; and the dereferencing construct "*->*" specifies a lookup of the attribute *language* within *function*. For query optimization, only the first two operators allow variations of the evaluation order to obtain performance improvement. To facilitate the optimizer's decision, a logical operator *REF* is introduced to explicitly

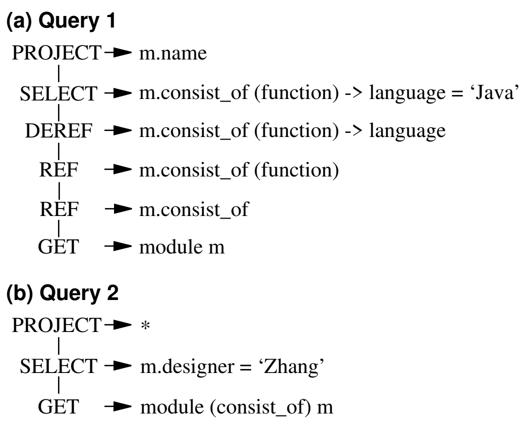

Fig. 9 After rewriting

represent the computation of each connection. Both *m.consist_of* and *consist_of (function)* are mapped to *REF*, bringing user-transparent metadata (e. g., *rel_consist_of*, cf. Sect. 3.1) into play. In this way, a sequence of *REF* operators not only permits to traverse from one participant to another via a relationship, but also enables the optimizer to analyze all feasible permutations for computing a relationship. Another logical operator *DEREF* is used to rewrite OrientSQL dereferencing constructs (cf. Fig. 9a).

3.2.3. Optimization

Now the rewritten IQR is optimized, i. e., transformed into a (logical) query execution plan (QEP) such as in Fig. 10 (corresponding to Fig. 9a). Recall that the *REF* operators explicitly represent each of the connections to be traversed. They can be transformed to *JOINs* (and be planed in the next step as, e. g., hybrid-hash join (*ALG-HH-JOIN*)). Particularly, while naively following a relationship chain is not always optimal, the explicit sequence of *REF* operators (and the resulting sequence of *JOINs*) offers the optimizer essential knowledge to speed up OrientSQL path computations by employing transformation rules similar as in [14].

Besides structural information, ORIENT relationships also possess semantic meanings that may have considerable influence on query processing. For example, in the context of selectivity and access path selection, cardinality restrictions can be used to estimate the number of instances of a specific type involved in a query. Moreover, the fact that *function* is non-sharable in *consist_of* indicates that an instance of *function* can be associated with only one relationship instance. In this case, pointer-based join algorithms will outperform their value-based counterparts for *ALG-HH-JOIN (f.RID = c.function)*, since only a very small fraction of *consist_of* instances actually participates in the *JOIN* and can be selected effectively using the pointer in *function*. The query optimizer should be able to take advantage of the relationship-specific knowledge, and consequently, it must be extensible!

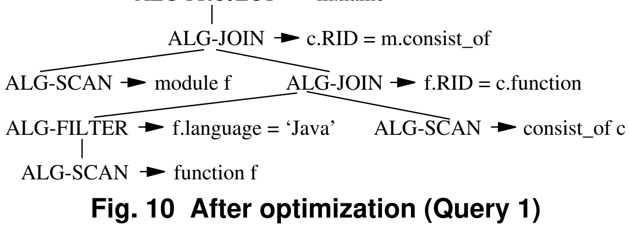

Fig. 10 After optimization (Query 1)

3.2.4. Plan generation

Subsequently, the logical QEP must be translated into a representation (physical QEP) ready for evaluation by the query execution engine. Since some new logical operators require low level algorithms genuinely different from those provided by the system, new physical primitives must be defined to supply building blocks for the physical QEP.

Especially, set processing and matching algorithms will be implemented in such a manner that they can operate on heterogeneous sets across several types, resulting from transitivity. For instance, when an aggregate is accessed, the corresponding parts must also be scanned. Such a *propagated scan* realizes the generic operator *GET ( )* (cf. Fig. 8). In this context, the query optimizer should be able to construct subplans for scanning involved tables and link them with a *propagation* control node, which instructs the execution engine to skip from subplan to subplan.

Until now, we have considered read-only queries. Usually, update requests may contain search predicates to determine which database objects are to be modified. Therefore, most of the discussions apply also to update requests. In addition, the specified ORIENT semantics have to be enforced during update processing. That means, the system must be able to take adequate reactions by, e. g., rejecting an operation that violates integrity, propagating an operation across a relationship, etc. Therefore, the update processing primitives must be redefined or overloaded [19].

3.2.5. Execution

At run time, the execution engine collaborates with various DBMS components to evaluate QEPs. These components deliver vital DBMS services such as concurrency control, buffer management, and recovery [7].

An important topic in this context is the transactional boundary. The associativity of operators influences where those borders can be put. Especially, to guarantee the correct establishment of a whole relationship, a kind of block atomicity is required (cf. Sect. 2.2.2). Since SQL guarantees only statement atomicity, an effective way is the use of nested transactions to support fine tuning of the control/rollback scope [12]. An example is illustrated in Fig. 11 where the transaction *TA* embodies the subtransaction SA_i bracketing the insert block to establish a *consist_of* instance in its entirety (cf. Fig. 3). This implies that existing rules of drawing transaction borders might have to be modified upon introducing new operators for flexible transaction control.

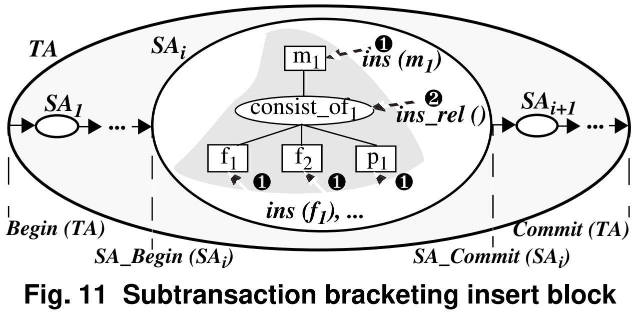

Fig. 11 Subtransaction bracketing insert block

3.3. Physical database design

The physical database design deserves even more attention [7]. ORIENT offers plentiful relationship

Table 2. Extensibility implications

DBMS Constituent	ORIENT's Need	ORDBMS' Support
DB Interface	Defining/administrating relationship extensions	✓ DataBlade interface
DDL Processor	Dealing with specifications; generating metadata	✓
Parser	Recognizing OrientSQL involving extensions	✓
Optimizer	Relationship-specific rewriting; choosing evaluation direction/sequence; exploiting optimization potential	**?** Poor optimization of (expensive) UDRs
Executor	New/overloaded low-level operators considering semantics; relationship connect/disconnect; inheritance resolution	Table-driven executor **?** Poor low-level algorithm/processing extensions
System Catalog	Providing special relationship metadata	**?** Impossible to define new metadata structure
Storage/Access Manager	Special storage/access methods considering relationship semantics and hierarchies	Indexing UDT/UDR **?** Index across types; physical design extensions
DBMS Utilities	Corresponding extensions to maintain relationship semantics, e. g., transaction logic, concurrency issue	**?** Little public interface for DBMS utilities such as concurrency control, recovery, etc.

semantics to model complex data structures. Different data objects can be related to each other in various ways and may reveal distinct access patterns. Therefore, it is non-trivial to find good storage and access structures that support a variety of algorithms on these data objects reasonably well. Thus, ideally, a wide spectrum of physical database design options should be provided, such as hierarchical clustering methods, hierarchical access structures, special relationship indexes, as well as corresponding scan algorithms and cost estimation methods. Hence, instead of providing hard-coded storage and access methods, extensibility is also desirable in this context to support efficient retrieval needed for, especially, transitive relationships.

From all these arguments it becomes evident that, to support ORIENT's extensions at the data model level seamlessly and efficiently, extensions at the processing level are indispensable.

4. ORIENT integration scenario

ORDBMSs have been particularly attractive to us, because their extension philosophy exactly conforms to the idea of ORIENT — to add features to the user interface of a DBMS by incorporating new functionality in the DBMS engine appropriately.

4.1. Discrepancy between desire and reality

To analyze how easy or difficult it is to integrate concepts that exceed the scope of UDTs and UDRs, we will sum up the main DBMS constituents that would be affected by ORIENT's extensions and contrast them to the possibilities offered by ORDBMSs.

ORDBMSs support the definition and registration of UDTs, UDRs, and user-defined access methods. However, these extensions have more impact on DDL processing than on DML processing. DBMS engines are made aware of the new features by catalog-driven and table-driven mechanisms in, e. g., parsers and executors. But how to process the new features efficiently and how to blend them with existing DBMS facilities are still open questions [4]. An example here is the incorporation of UDR optimization with the traditional table-oriented optimization [20]. Elegant solutions to these questions ask to change the existing system code base reasonably, which is infeasible in current ORDBMSs. Moreover, ORDBMS extensions are targeted to the type system. Some advanced functionality such as generic B-trees, R-trees, or grid files are all associated with new (complex) data types. The ORIENT proposal at the data model level goes beyond this scope and puts more challenges on the processing as well as the integration:

- At the data model level, user-defined relationships with complex semantics mean extensions in a broader sense than just new domain types and operations on such types.
- To accommodate all the data model extensions, language-level and interface-level changes are needed (e. g., OrientSQL and the graphical user interface *OrientDraw* [23]).
- These changes affect query processing, which must understand new syntax, conduct selective transformation and optimization, produce appropriate output, and ensure integrity in the whole course.
- At a lower level, the system should support the implementation of new algorithms considering ORIENT semantics.
- Similarly, the system should enrich its repertoire of basic mechanisms, e. g., adding a dedicated transaction paradigm to facilitate block atomicity, where subtransactions are needed for multiple insertions to be made atomic.
- Moreover, new storage and access methods are necessary to store and retrieve related data efficiently. Accordingly, processing constituents such as query optimizer should employ new physical design options.

In a word, extensions to one component often require support from the others. In spite of the success story of ORDBMSs, we find the term "extensibility" misinterpreted, since it is hardly ever the entire DBMS which is extensible. The current object-relational approach relies on assumptions about the design and implementation of the DBMS into which extensions are plugged. A more satisfactory support of semantic relationships calls for a more open approach to extensibility than what ORDBMSs exhibit. To expound this further, we will briefly present our efforts to integrate ORIENT, together with some comments.

4.2. Deep integration approach

We have prototyped the deep integration approach with PostgreSQL [18]. The rationale is to use the existing system architecture and to exploit knowledge on relationship semantics wherever possible throughout query processing.

4.2.1. Prototype

In correspondence with the desired support addressed in Sect. 3, Fig. 12 adjusts the architecture explained in [18] and shows which system components are extended (marked with shaded areas) in order to meet ORIENT's needs.

- The ORIENT *Parser* deals with OrientSQL both syntactically and semantically. And the ORIENT *Path Manager* is responsible for statements involving relationships.
- The ORIENT *Function Manager* dispatches newly-added statements.
- The ORIENT *Query Rewriter* generates IQRs (OrientAGs) for further processing.
- The ORIENT *Query Optimizer* exploits relationship-specific optimization potential.
- New processing primitives or overloaded DBMS primitives take relationship semantics into account.
- Support from other DBMS services is indispensable, including *System Catalog* and *Transaction Manager*.
- Physical database design issues are not addressed yet.

4.2.2. Comments

Metadata is used in many processing phases. It is therefore crucial to forward all the "hints" of relationships from one processing phase to the subsequent ones. For example, associativity of relationships, which might be no longer obvious when expressed as symmetrical *JOINs,* may be later on necessary to achieve optimal *JOIN* orders. We keep these "hints" as metadata to guide the whole course of processing. As described in Sect. 3.1, relationship-specific metadata acts as a "scratchpad" that is looked up in different processing phases. Relationship properties are enumerated in this "scratchpad". Each time a property can be exploited, it is ticked off in the list (marked as "considered").

Furthermore, a trade-off between gains and efforts should be determined. The deeper the integration is, the better is the performance promise, and at the same time the more effort is required to modify the respective system components. Modifications must not, however, alter the default behavior (SQL mechanisms) of the system. Finding a good balance between radically changing existing facilities and implementing new ones "on top" of old ones (which will for sure render suboptimal performance) is a major design issue. However, it can be decided separately for all components. That is why we put implementation emphasis on the parser, optimizer and physical primitives, but touch on other components only lightly. Moreover, the OO concept "function overloading" is a good means to smoothly integrate existing functionality with some variations thereof. But it necessitates a clear structure of system design and implementation code, which is, unfortunately not the case in PostgreSQL (programmed in C).

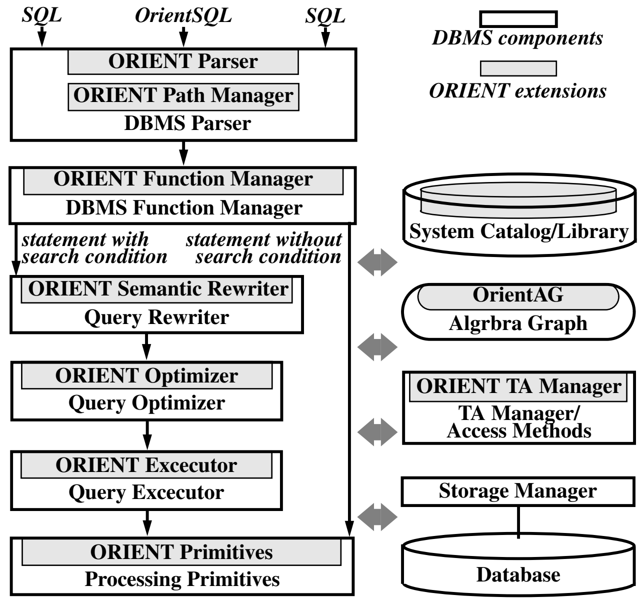

Fig. 12 Deep integration of ORIENT

Finally, this approach has a strict limitation: the candidate platform. Commercial products (including those claiming to be extensible) with closed internal interfaces are absolutely out of question since no modification of the DBMS core is possible. Public-domain DBMSs, on the other hand, are mostly neither mature nor stable, which causes a lot of problems during implementation.

4.3. DataBlade-like approach

The classical way of extensibility defines a supplementary layer on top of the DBMS, which requires extra data transfer between the DBMS and ORIENT and makes it difficult to employ new data types seamlessly. In contrast, the object-relational approach offers a closer integration of the new functionality. For this reason, we use the ORDBMS Informix [9] and implement ORIENT as an extension module comparable to a DataBlade.

4.3.1. Prototype

Following the DataBlade philosophy, the required supports are "encapsulated" in a package with relationship implementation constructs organized into a hierarchy (cf. Fig. 13). The most general construct *OrientRela-*

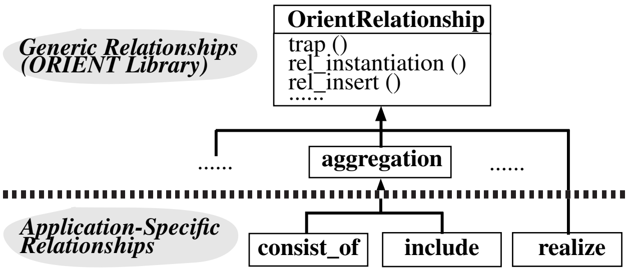

Fig. 13 Relationship hierarchy

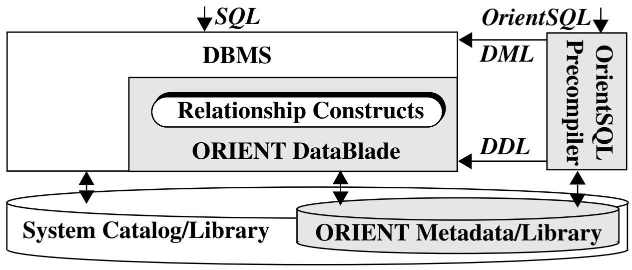

Fig. 14 DataBlade-like ORIENT

tionship provides default methods for other ORIENT-generated constructs to inherit and adjust. These methods deliver common behavior needed for all relationships, e. g., to instantiate relationships (*rel_instantiation ()*) and to insert (*rel_insert ()*) an instance into a relationship type.

Relationship-specific methods take appropriate reactions to maintain semantics. The relationship properties, the related objects, and the affected operations determine what and how enforcement measures are derived from default ones. Whenever the need of new processing logic is recognized, a corresponding method will be called. Here, a precompiler is essential for two purposes. First, it transforms, with the help of the ORIENT Metadata/Library, declarative DDL specifications into user-transparent relationship constructs. Second, it recognizes new DML features and replaces them with calls to encapsulated methods of relationship constructs. The system architecture is presented in Fig. 14.

4.3.2. Comments

In this approach, the basic problem remains the same as before: How to convey the knowledge about relationships and their semantics to the system? Current ORDBMSs cannot deliver a satisfactory answer, since they treat UDRs in a "black box" way. While ORDBMSs optimize relational operations like *JOINs*, there is hardly any optimization for UDRs, since the system understands little semantics about each UDR. A UDR is merely a signature, with minimal information such as user-speculated cost [20].

Due to the rather closed system architecture, it is a delicate task to preprocess OrientSQL in such a fashion that the resulting query is not "misunderstood" by the underlying query optimizer, that means, query processing should neither compromise the optimization goals nor violate the specified relationship semantics. In order to prevent this, it is indispensable to have sound knowledge of the underlying DBMS, particularly of its query optimizer. But even then the DBMS may be too "dumb" for passing its valuable optimization information.

Hence, this approach (probably not only for ORIENT) renders a new "application layer" in between the DBMS and the real application. In our setting, although the ORIENT package can be registered in the system, or in other words, the system is made aware of its existence, the solution is still a "on top" one, in the sense that the package has no opportunity to customize the internal DBMS implementation. Some changes to the underlying DBMS can only be simulated by methods, which leads to a deviation from the expected functionality (e. g., proper transaction logic). Moreover, unlike the seamless usage of UDRs in queries, ORIENT's methods can only be invoked "indirectly" through the precompiler during relationship manipulation and semantics control.

All these observations lead to a question: What is the necessary infrastructure to achieve real extensibility?

5. Reflection on extensibility infrastructure

In our opinion, three principles are of paramount importance for an extensibility infrastructure: a modular system architecture, proper abstraction levels, and common internal interfaces. Proven software engineering methodologies have demonstrated the feasibility of these principles for constructing extensible systems [22]. Meanwhile, some DBMS prototypes have also applied similar principles carefully, going towards making the system more or less extensible [4, 5]. For instance, Open OODB [3] defines an architectural framework by dividing a DBMS into a series of separate modules supporting specific facilities. The clean separation between different submodules (such as internal operators) allows exploration with alternative methods for e. g., query processing.

Now let us comprehend the implications of these principles to DBMS extensibility.

The first point is revealing internal processing logic in controllable ways. There is a distinction between exporting DBMS facilities for interoperation and exposing them for extensions. The former can be well covered through functional signature interfaces. While such a "black box" abstraction might be suitable for developing some applications where code can be reused without relying on anything else but the specification, reasonable extensions to a DBMS core demand a controlled part of its processing logic to be exposed, thereby leading to a "gray box" abstraction.

The second point is giving things that will change a handle that does not change. System catalogs are one example. In Postgres [21] and its ORDBMS successors [9, 18], descriptions about types, routines, access methods etc. are all stored in system catalogs rather than being hardwired in the DBMS. Besides, the query processor is table-driven with the help of the metadata. As a result, new features can be added easily to the catalogs without modifying the DBMS engine. Internal rule sets are another example. They provide a mechanism for responding to changes flexibly. Some optimizers (e. g., in Starburst [17]) guarantee extensibility w.r.t. new strategies and new operators by means of rule-based techniques.

The third point is the reuse strategy. The prototype TIGUKAT follows the building-block philosophy [15]. The search space, the search strategy and the cost function are all modeled as objects, thus providing for the query optimizer the extensibility inherent in the OO paradigm.

Most of the previous efforts, however, pursue extensibility in one dimension (e. g., query processing and transaction control), but neglect the other dimensions. As a result, all mechanisms at different levels (as mentioned in Sect. 4.1) are not extensible in a coordinated way. Even in their aimed dimension, the complexity problem is not well addressed [4], leaving a lot of issues open.

To us, extensibility is a matter of degree. Even "ancient" products were extensible by the system implementers. Even today, DBMS extensibility is not available to "ordinary" users or skilled application programmers. Some doubts about extensibility w.r.t. security, stability and performance should inspire us to go further instead of giving up, with the object-relational approach as a good reference point.

6. Conclusion

We tried to employ ORDBMS' extensibility to realize ORIENT's concepts, i. e., providing uniform and concise modeling as well as adequate integration of semantically rich relationships. However, although the term extensibility is used quite generously, its application is restricted to (more or less) advanced data types. To support ORIENT's extensions at the data model level seamlessly, extensions at the processing level are necessary, which is not feasible in current ORDBMSs. With commercial systems we encounter obstacles due to the closed internal interfaces, whereas the lack of maturity and stability in public-domain systems causes a huge amount of implementation effort. All these lessons learned let us ponder on a fundamental question: What is the necessary extensibility infrastructure?

To offer a satisfactory answer to this question, there remain a lot of challenges. Because the development of an extensibility infrastructure in a general sense may be unrealistic, we focus on our special target — semantic relationship support. We take a sound understanding of the problem-specific processing and its required extensibility techniques as a further step towards achieving a comprehensive notion of DBMS extensibility.

References

[1] ANSI-X3H2-95-456 R2: *Introducing Reference Types and Cleaning up SQL3's Object Model,* Dec. 1995.

[2] ANSI-X3H2-99-079/WG3:YGJ-011: *Database Language SQL — Part 2: Foundation (Working Draft)*, March 1999.

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

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

[5] M. J. Carey, L. Haas: *Extensible Database Management Systems*, SIGMOD Record 19:4., 1990, 54-60.

[6] C. J. Date: *Relational Databases: Selected Writings 1985-1989*, Addison-Wesley, 1990.

[7] G. Graefe: *Query evaluation techniques for large databases*, ACM Computing Surveys 25:2, 1993, 73-170.

[8] *IBM DB2 Universal Database (V 5.0)*, IBM Corp., 1997.

[9] *Informix Universal Server (V 9.12)*, Informix Software, 1997.

[10] M. Jarke, J. Koch: *Query Optimization in Database Systems*, ACM Computing Surveys 16:2, 1984, 111-152.

[11] N. M. Mattos: *An Approach to Knowledge Base Management*, LNCS 513, Springer-Verlag, 1991.

[12] J. E. B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing,* M.I.T. Press, 1985.

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

[14] M. T. Özsu, J. A. Blakeley: *Query Optimization and Processing in Object-Oriented Database Systems*, in: Modern Database Management — The Object Model, Interoperability, and Beyond, W. Kim (ed.), Addison-Wesley, 1994.

[15] M. T. Özsu, R. J. Peters, D. Szafron, B. Irani, A. Lipka, A. Muñoz: *TIGUKAT: A Uniform Behavioral Objectbase Management System*, The VLDB Journal 4:3, 1995, 445-492.

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

[17] H. Pirahesh, J. Hellerstein, W. Hasan: *Extensible/Rule Based Query Rewrite Optimization in Starburst*, Proc. 1992 ACM SIGMOD Conf., San Diego, June 1992, 39-48.

[18] *PostgreSQL v6.4.2*, http://www.postgresql.org.

[19] T. Rieffel: *Semantic Relationships in Object-Relational Database Systems — Concepts, Language Design, and Implementation Approach* (in German), Diploma Thesis, Computer Science, Univ. of Kaiserslautern, April 1998.

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

[21] M. Stonebraker, G. Kemnitz: *The POSTGRES Next Generation DBMS*, Comm. ACM 34:10, 1991, 78-92.

[22] C. Szyperski: *Component Software — Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[23] A. Vorberg: *Supporting Semantic Data Relationships — User Interface and Precompiler* (in German), Diploma Thesis, Computer Science, Univ. of Kaiserslautern, April 1998.

[24] N. Zhang, T. Härder: *Enhanced Support of Relationship Semantics in Object-Relational Databases*, Proc. XII Brazilian Symp. on Advances in Database Systems, Maringa, Brazil, Oct. 1998, 7-21.