in: Proc. 18th British National Conference on Databases (BNCOD 2001), Oxford, July 2001, Advances in Databases, Read, B. (Ed.), LNCS 2097, Springer, pp. 89-104

The Real Benefits of Object-Relational DB-Technology for Object-Oriented Software Development

Weiping Zhang, Norbert Ritter

University of Kaiserslautern, Germany
P.O. Box 3049, 67653 Kaiserslautern
{wpzhang, ritter}@informatik.uni-kl.de

Abstract: Object-oriented programming languages (OOPLs like C++, Java, etc.) have established themselves in the development of complex software systems for more than a decade. With the integration of object-oriented concepts, object-relational database management systems (ORDBMSs) aim at supporting new generation software systems better and more efficiently. Facing the situation that nowadays more and more software development teams use OOPLs 'on top of' (O)RDBMSs, i. e., access (object-)relational databases from applications developed in OOPLs, this paper reports on our investigations on assessing the contribution of object-relational database technology to object-oriented software development. First, a conceptual examination shows that there is still a considerable gap between the object-relational paradigm (as represented by the SQL:1999 standard) and the object-oriented paradigm. Second, empirical studies (performed by using our new benchmark approach) point at mechanisms, which are not part of SQL:1999 but would allow to reduce the mentioned gap. Thus, we encourage the integration of such mechanisms, e. g., support for navigation and complex objects (structured query results), into ORDBMSs in order to be really beneficial for new generation software systems.

1 Motivation

Object-oriented programming languages (OOPLs), such as C++, Java, SmallTalk, etc., have established themselves in the development of complex software system for more than a decade. Both, the structure of these systems as well as the structure of objects managed by these systems have become very complex. Object-oriented concepts offered by OOPLs are well suited for managing complex structured objects. However, there are additional requirements, such as persistence and transaction-protected manipulation, which can only be fulfilled efficiently by integrating a database management system (DBMS). Consequently, database technology becomes one of the core technologies of modern software systems. In times of the 'breakthrough' of object-oriented system development, two kinds of DBMSs were of practical relevance: relational DBMSs (RDBMSs) and object-oriented DBMSs (OODBMSs). Using OODBMSs has proved inefficient/inflexible for reasons we cannot go into in this paper. Consequently, OODBMS did not gain wide acceptance [5], and, therefore, will not be further considered in this paper.

Using an RDBMS, on one hand, requires to overcome the well known impedance mismatch [5, 13], i. e., performing the non-trivial task of mapping complex object structures and navigational data processing (at the OOPL layer) to the set-oriented, descriptive query language (SQL92), which supports just a simple, flat data model. Despite this considerable mapping overhead, mature RDBMS technology (index structures, optimization, integrity control, etc.), on the other hand, contributes to keep the overall system performance acceptable. Several commercial systems [2, 12, 14, 15, 17, 19] mapping object-oriented structures onto the relational data model are currently available. Such systems are often referred to as *Persistent Object System built on Relation* (shortly: POS).

The object-relational wave [22] in database technology has decisively reduced the gap between RDBMSs and OOPLs. Although object-relational DBMSs (ORDBMSs) are able to (internally) manage object-oriented structures (see data definition part in [20, 21]), the required seamless coupling of OOPLs and ORDBMSs is not yet possible, because (as in SQL92) results of SQL:1999 queries (see data manipulation part in [20, 21]) are rather (sets of data) tuples than (desired sets of) objects. In summary, the gap between OOPLs and ORDBMSs can be traced back to a whole bunch of modelling and operational aspects, as we will detail in the following sections. Furthermore, the SQL:1999 standard and the commercially available ORDBMSs differ very much in their object-oriented features. Thus, it is by no means clear, how a given object-oriented design can be mapped to a given ORDBMS (most) efficiently, or which features should be offered by ORDBMSs in general in order to enable an efficient mapping of object-oriented structures, respectively.

Our long-term objective is to influence the further development of ORDBMSs towards a better support of object-oriented software development (minimal mapping overhead). Thus, we have proposed a new benchmark approach in [26] allowing to assess a given ORDBMS by taking into account both, its own performance as well as the required mapping overhead. Furthermore, [26] presents basic comparisons of purely relational and object-relational mappings. This paper goes beyond [26] in that the focus is to point up new directions of ORDBMS development, which, as is proved by corresponding empirical examinations, object-oriented software development can leverage from and, therefore, should be further pursued. Thus, this paper is structured as follows. A conceptual examination (section 2) outlines how the object-relational data model (standardized by SQL:1999) corresponds to OOPLs. Section 3 discusses corresponding mapping rules. Afterwards, section 4 gives a brief introduction into our benchmark approach needed to interpret the measurement results detailed in section 5. These results show that object-oriented software development can leverage from object-relational technology (in comparison to purely relational technology), but that further improvements can be reached by a better support of navigational access (retrieving objects by object identifiers) and appropriate mechanisms for retrieving complex structured sets of objects. We propose to admit corresponding mechanisms for navigation and complex object support to future versions of the SQL:1999 standard as concluded in section 6.

2 Conceptual Consideration

There is a multiplicity of object data models, for example ODMG [9], UML [24], COM [2], C++ and Java. All these models support the basic concepts of the OO paradigm, however, there are certain differences. Independently from the modelling language used in the OO software development (e. g., UML), SQL:1999 must be coupled with a concrete OOPL. In accordance to their overall relevance and conceptual vicinity, we concentrate on the object model of C++ and ODMG and compare it with the SQL:1999 standard [11, 20, 21].

2.1 Modelling Aspects

Object Orientation in OOPL. The concept *object* represents the foundation of the OO paradigm. An object is the encapsulation of data representing a semantic unit w. r. t. its structures/values and its behaviour. It conforms to a particular class [16]. In fact, a class implements an object type (*classification*) which is characterized by a name as well as a set of attributes and methods. Each attribute conforms to a certain data type and is either single-valued or set-valued (*collection types*). Furthermore, a data type can be scalar (e. g., integer, boolean, string, etc.) or complex. In the latter case corresponding values can be references (*association*) or objects of other classes (*aggregation*) so that complex structures can be modelled. A class may implement methods (*behaviour*) which can be invoked in order to change the object's state. Classes may be arranged within class hierarchies. A class inherits structures and behaviour from its superclasses (*inheritance*), but may refine these definitions (*specialization*). Due to space restrictions we do not give a more detailed description of the OO paradigm, but discuss how the OR data model conforms to OO concepts.

Object Orientation in SQL:1999. While the relational data model (SQL2) did not support semantic modelling concepts sufficiently, in SQL:1999 the fundamental extension supporting object-orientation is the structured *user-defined data type* (UDT, [11]). UDTs, which can be considered as object types, can be treated in the same way as predefined data types (*built-in data types*). Consequently, similarly to the type system of OOPLs the type system of SQL:1999 is extensible. UDTs may be complex structured and, therefore, may not only contain predefined data types but also set-valued attributes (*collection types*) and even other UDTs (*aggregation*) or references (*associations*). Obviously, UDTs are comparable to the classes of the OO paradigm. However, according to the SQL:1999 standard a UDT must be associated with a table. The notion of *typed table*, also referred to as *object table*, allows to persistently manage instances of a certain UDT within a table. Each tuple of such a table represents an instance (*object*) of a particular UDT and is identified by a unique object identifier (*OID*) which can be system-generated or user-defined. Besides instantiable UDTs, SQL:1999 also supports non-instantiable UDTs, which conforms to the notion of abstract classes in OOPLs. In addition, UDTs may have methods (*behaviour*) which are either system-generated or

implemented by users. They may participate in type hierarchies, in which more specialized types (*subtypes*) inherit structure and behaviour from more general types (*supertypes*), but may specialize corresponding definitions. Thus, SQL:1999 supports polymorphism and substitutability, however, multiple-inheritance is not supported. Due to the association of UDTs with tables (see above) SQL:1999 does not support encapsulation and, consequently, there is nothing like the degree of encapsulation known from OOPL (*public, protected, private*).

2.2 Operational Aspects

Beside the fundamental modelling aspects discussed so far, we also have to examine operational aspects in order to figure out the *conceptual distance* adequately. The following aspects are most relevant to our consideration:

Descriptive Queries vs. Navigational Processing. While OOPL processing is inherently navigational, SQL supports a set-oriented, descriptive query language. Both navigational and set-oriented query processing are important to modern software systems. Therefore, ORDBMSs should also directly facilitate navigational processing to fulfil this requirement of OO applications. *Direct support of navigational access* by the DBMS would mean that a database object referred to by its OID can be provided as instance of an OOPL class. However, to the best of our knowledge none of the currently available ORDBMSs directly supports this notion of navigation.

A naive coupling of OOPLs with descriptive SQL requires to issue one or several corresponding SQL queries (see Fig. 1) to the database for processing a dereferencing operation, e. g. *GetObject(Ref)*, and retrieving the requested object from the database server. Such a processing scheme will surely lead to a bad runtime behaviour of the entire system, since the costs of transforming a navigational operation to SQL queries, of evaluating these queries in the DBS, and of the client/server communication can be very high. Obviously, the lack of DB APIs in directly supporting navigational access impairs the system efficiency badly. Thus, either direct support for navigation[1] must be provided or efficient prefetching mechanisms exploit-

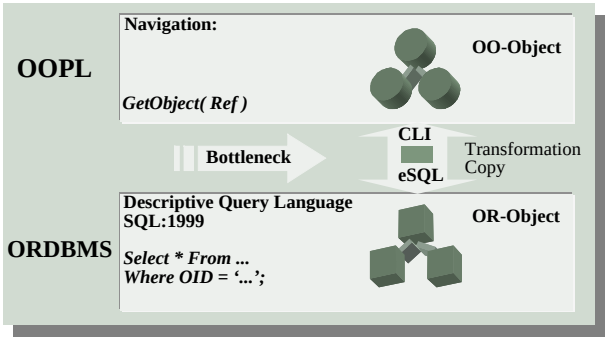

Fig. 1: Bottleneck between OOPL and ORDBMS

1. In section 5 we will see that one of the commercially available ORDBMS provides some basic means for a direct access to objects by OIDs. Measurement results show that this is at least a step into the right direction.

ing set-oriented database access and, thereby, reducing the number of database round-trips must be applied in order to effectively couple OOPLs with ORDBMSs.

Structured Query Results. As already mentioned several times before, OOPLs support complex structured object types, especially by the possibility of nesting complex data types as well as using collection types and references. We have also mentioned previously that these facilities of modelling complex structured objects have been integrated into SQL by the SQL:1999 standard. Unfortunately, because of the traditional basic concepts of SQL, complex structures (actually supported both in OOPL and ORDBMS) get lost at the DBMS interface, since only (sets of) simply structured, flat data tuples can be retrieved. Therefore, if we want to couple an OOPL with an (O)RDBMS, it is necessary to separately retrieve simple fragments of complex objects by issuing several SQL queries, and then rebuild complex object structures at the programming language level (see Fig. 1). The mentioned problem even gets worse, if not individual complex objects, but complex structures (object graphs) containing numerous related objects interconnected by object references are to be selected as units. Obviously, the lack of direct support for complex structured objects at the DB API reveals a *bottleneck* between the two paradigms (see Fig. 1), and prevents new generation software systems from exploiting the potential power of ORDBMSs most effectively.

Object Behaviour. Of course, the operational aspects also encompass the object behaviour implemented in the database. Because of special implementational aspects these methods (UDFs) can almost exclusively be executed at the server side, or, if these UDFs or special client-invokable pendants are executed at the client side, it cannot be guaranteed that these pendants perform the original semantics. For example, there may be complex dependencies between UDFs and integrity constraints, e. g., referential integrity constraints and triggers, which are implemented by using SQL and are automatically ensured by the DBMS. Thus, it is almost impossible to support calling UDFs at the OOPL level in the same (‘natural’) way as usually object methods can be called. Therefore, we do not consider a mapping of object methods in this paper and restrict our considerations to navigational and set-oriented access.[2]

3 Mapping Rules

In the previous section, we outlined the conceptual distance between the OO paradigm and SQL:1999. Considering an individual ORDBMS, its OO features determine the overhead which has at least to be spent in order to bridge this distance. Nevertheless, in theory there is an entire spectrum of possibilities to design the required mapping layer. On one hand, it depends on how ’natural’ coding in the OOPL has to remain, and, on

2. At this point, we want to mention that there are some more aspects of ORDBMSs, which OO applications may benefit from, but which cannot be captured in this paper, e. g., facilities for integrating external data sources into database processing.

the other hand, on how far the OO features are to be exploited. Regarding the first point ('natural' coding), we demand that the programmer must not be burdened by having to take data management aspects into account. Thus, programming must be independent of the database as well as the mapping layer design. Regarding the second point (degree of exploiting OO features), we want to outline the two extremes of the mentioned spectrum, i. e., pure relational mapping and full exploitation of the OO features offered by the considered ORDBMS[3].

Pure Relational Mapping. As mentioned before, there are several commercial POSs mapping OO structures to relational tables. Objects are represented by table rows. Since RDBMSs do not support set-valued attributes, user-defined data types, and object references, additional tables are required to store corresponding data and to connect them with the corresponding class tables via foreign keys [1]. Thus, several tables, may be required to map a given class. Principally, there are several ways of representing a class hierarchy in the relational model, comprehensively discussed in [8]. After studying pros and cons, we decided to use the horizontal partitioning approach (see [8] for details), since it provides good performance in most cases, and is also used in most commercial POSs [2, 12].

Object-Relational Mapping. Exploiting the OO features of ORDBMSs is commonly argued to be more promising [3], but the real benefits in comparison to the pure relational mapping are not very well studied yet. This paper will give some performance evaluations later on.

Before outlining general mapping rules exploiting OO features of ORDBMSs, we have to re-emphasize the following point. Our benchmark approach, which will be outlined in the subsequent section, assesses a certain ORDBMS by taking the required mapping overhead into account. In order to be fair, the mapping layer used throughout the measurements must be designed in an optimal way w. r. t. the capabilities of the ORDBMS considered. Therefore, the design of the mapping layer may differ with the ORDBMSs to be assessed. In the following, we just outline general mapping rules, which are based on the SQL:1999 concepts, in order to provide some basic understanding on how a mapping layer can be designed.

A C++ class maps to a UDT in SQL:1999. Non-instantiable UDTs correspond to abstract C++ classes. A UDT is associated with exactly one table (*typed table*) to initialize its instances. Each tuple in this table represents a persistent instance (*object*) of a particular class and is associated with a system-generated OID. Embedded objects (*aggregation*) entirely belong to their top-level object and, therefore, do not own an OID. Extents are mapped to the list constructor of C++ STL. Keys are managed at the mapping layer by applying the map constructor of C++ STL. A C++ class hierarchy maps to a hierarchy of structured UDTs. However, SQL:1999 only supports single-inherit-

3. Note, SQL:1999 and the commercially available ORDBMSs differ very much in their OO features, as we will see in the subsequent sections of this paper.

ance so that multi-inheritance has to be simulated at the mapping layer. SQL references are mapped to C++ pointers. Since SQL:1999 does not support diametric references, a relationship type is broken down into two separate primary-key/foreign-key connections and the mapping layer maintains the referential integrity. Except for mutator and observer methods, which are generated by the mapping layer w. r. t. the constraints defined in the user database schema, object behaviour is not yet considered in our performance investigation. Navigation is supported by offering the function *GetObject(Ref)* which, in the case that the DB API does not directly support OID-based object fetching, is implicitly transformed into an SQL query.

After having discussed (modelling and operational) discrepancies between ORDBMSs and OOPLs as well as the mapping rules needed to bridge the gap, we proceed with our performance evaluations.

4 Performance Evaluation

Our discussion in section 2 shows that there is only a small difference between the OO and OR paradigms w. r. t. modelling aspects, but a considerable distance w. r. t. the operational aspects and the application semantics. In order to further evaluate this distance as well as to quantify the overhead required for bridging this gap, we propose a configurable benchmark approach [18, 26].

Remind, we do not consider OODBMSs, but ORDBMSs, because we more and more have to face the situation that people are using OOPLs for software development and (O)RDBMSs for data management purposes so that there is a need for a more detailed examination of the efficiency of possible coupling mechanisms. Consequently, the OO7-Benchmark [5] representing an important standard for benchmarking OO systems, is not appropriate for our purposes. The performance of RDBMSs or ORDBMSs has traditionally been evaluated in isolation by applying a standard benchmark directly at the DBMS interface. Sample benchmarks [8] are the Wisconsin benchmark [3], the TPC benchmark [23] as well as the Bucky benchmark [6]. These benchmarks are very suitable for comparing different DBMSs with each other [8]. However, none of these benchmarks helps to assess the contributions of a DBMS to OO software development. Consequently, these other approaches do not take the typical application server architecture and the fact that the DBMS capabilities determine the overhead of the required application/mapping layer into account. Furthermore, data types as well as operations of the applications we consider may differ significantly (double-edged sword [6, 22]), so that a standard benchmark can not cover the entire spectrum. Therefore, we propose an open, configurable benchmark system allowing to examine the entire system (incl. mapping layer) w. r. t. to its typical applications. Such a system will also help us to get results transcending those reported on in this paper (see succeeding sections), e. g., more detailed examinations of navigational support. In the following, we outline our first prototype. Further details can be found in [1, 26].

4.1 Benchmark System

An open, configurable benchmark system is not necessarily difficult to be applied, as our approach proves. Indeed, our current prototype offers 3 predefined configurations, which w. r. t. database size are *small*, *medium* and *large* in order to be sufficiently scalable. Both, structures (data type and type hierarchies) as well as complexity of data in the 3 standard configurations are determined in cooperation with one of the leading software vendors for business standard software and, thus, represent a wide spectrum of typical application domains. In addition, our benchmark system can be simply configured according to the particular properties of a concrete application. Fig. 2 gives an overview of the architecture of our prototype. Among other possibilities, users can directly control the generation process of the benchmark database, e. g., specify the database size, the complexity of the class hierarchy and the complexity of the individual objects, in order to take care that the special requirements of the application in mind are taken into account. After having generated the benchmark database the user may select from a given set of query templates (see section 4.2 for further details) and indicate how many times each template is to be instantiated. New query templates can be easily added, if the existing templates do not reflect application characteristics sufficiently. Based on these user selections/ specifications the load generator creates a set of queries which is passed to the query executor, which, in turn, serves as a kind of driver for measurements. Users can also specify which kinds of measurement data are to be collected by the system, i. e., amount of time spent at the DB or the mapping layer for query transformation, or the time spent for SQL query evaluation, data loading, and/or result set construction. Corresponding values are collected by the data collector during execution of the query set and afterwards stored in the DBS for further evaluations.

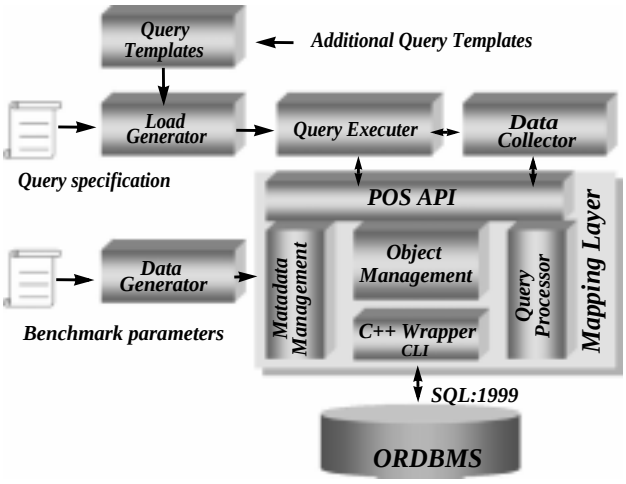

Fig. 2: Architecture of the benchmark system

As explained in more detail in [1, 26], the special challenges of this benchmark approach are, on one hand, to properly take into account the requirements of OO system development, and, on the other hand, to guarantee an optimal mapping w. r. t. the particular capabilities of an individual ORDBMS.

4.2 Benchmark and Measurements

In order to get a complete performance evaluation, we concentrate on answering the following questions:

1. Which performance gains offer ORDBMSs in comparison to RDBMSs regarding their usage in OO software development?
2. Which additional overhead has to be spent at the mapping layer in order to bridge the gap between the OO and OR paradigms and how does it behave facing different query types?
3. To which extent is the system performance influenced by the capabilities of the (O)RDBMS API?

In order to be able to answer the first two questions, we have selected a set of typical benchmarking queries according to a long-term study of a leading software company. These queries represent a wide spectrum of typical operations in the target applications of ORDBMSs. We have compared a purely relational mapping with an object-relational one (by means of exploiting its OO modelling power) by using a currently available commercial ORDBMS. This way we 'measured' how OO software development can leverage from the OO extensions offered by ORDBMSs (e. g., structured UDTs, references, etc.). The operations considered for that purpose are implemented as query templates and grouped in following categories:

Navigation operations: Navigation operations, such as *GetObject(OID)*, are not directly supported by almost all currently available ORDBMSs. Considering such operations helps us to assess the performance of ORDBMSs in supporting navigational processing. We hope that corresponding results 'help' ORDBMS vendors to make ORDBMSs as efficient as OODBMSs are in this concern.

Queries with simple predicates on scalar attributes: Queries of this category have simple predicates just containing a single comparison operation on a scalar attribute. This group mainly serves to provide a performance baseline that can be helpful when interpreting results of more complex queries.

Queries with predicates on UDTs: This group contains queries with simple predicates (a single comparison operation) on attributes of structured, non-atomic data types. Thus, it mainly serves for assessing the efficiency of mapping UDTs to (O)RDBMSs.

Queries with predicates on set-valued attributes: This group contains queries with simple predicates (a single IN operation) on nested sets. ORDBMSs directly support set-valued data types. In the relational mapping, several tables (according to the degree of nesting), which are connected by primary/foreign-keys, are necessary.

Queries with path predicates: This group contains queries evaluating simple predicates after path traversals. These queries allow to evaluate the efficiency of processing dereferencing operations (path traversals) in ORDBMSs.

Queries with complex predicates: Queries of this group contain complex predicates challenging both query transformation as well as query optimization.

Queries on the class hierarchy: While all other queries exclusively deliver direct instances of a single queried class, queries of this group deliver transitive instances as well. Predicates conform to those of the second category. This group of queries allows to evaluate the efficiency of the ORDBMS in handling class hierarchies (inheritance). The comparison with the relational mapping has been expected to demonstrate the advantages of ORDBMSs.

The third question posed at the beginning of this section deals with the capabilities of the DB interface especially w. r. t. support for complex structured objects and navigational access. In order to examine these aspects, we performed measurements on two different (commercially successful) ORDBMSs. One of these systems offers the more traditional interface, whereas the second one provides some basic means of supporting complex structures objects.

We performed our measurements[4] on a benchmarking database with 100 classes and 250000 instances (configuration *medium*). In order to use a representatively structured class hierarchy, we studied typical application scenarios of a renowned vendor of business standard software and parameterized our population algorithm accordingly. We measured the database time (DB time) and the total system time (TS time). The DB time of SQL queries is the time elapsed between delegating the queries to the DBMS and receiving back the results (open cursors, traverse iterators). It includes the time for client/server communication, the time for evaluating the queries within the DBS and the time for loading the complete result sets. This has to be taken into account, when analysing the measurement results. The TS time is defined as the total elapsed time from issuing a query operation at the OOPL level until having received the complete result set. It contains the time spent within the mapping layer as well as the DB time.

We think that these 3 questions have to be answered before we can think about, how OR technology can be improved in order to support OOPLs better and more efficiently. In the following section, we report on our measurement results.

5 Measurement Results and Observations

5.1 ORDBMS vs. RDBMS

In the first test series, we have compared a purely relational mapping with an OR mapping by using one of the leading currently available commercial ORDBMSs. This in-

4. All experiments in this paper use commodity software. The hardware and the software configurations are left unspecified, to avoid the usual legal and competitive problems with publishing performance numbers for commercial products. All performance measurements are averages of multiple trials, with more trials for higher variance measurements. For each DBMS tested, we put much effort in optimization (e.g., indexes) and mapping layer design in order to achieve the best performance possible.

vestigation aims at quantifying the benefits of OO extensions offered by ORDBMSs in more detail.

Fig. 3 illustrates the measurement results. Due to space restrictions, it is not possible to analyse all results in detail. It can be observed that the OR mapping outperforms the purely relational mapping in all query categories. Although the OR mapping shows only tiny advantages in retrieving small result sets, it provides performance gains of up to 40% in retrieving large result sets, or processing queries on class hierarchies. The reasons are twofold. First, the OO features provided by the ORDBMS contribute to keep the complexity of the mapping layer low (better query evaluation strategy, less overhead for synthesizing the result set) and to reduce client/server communication (less queries). Second, the implementation of the ORDBMS (we used) enables performance improvements, (even) if using OO extensions.

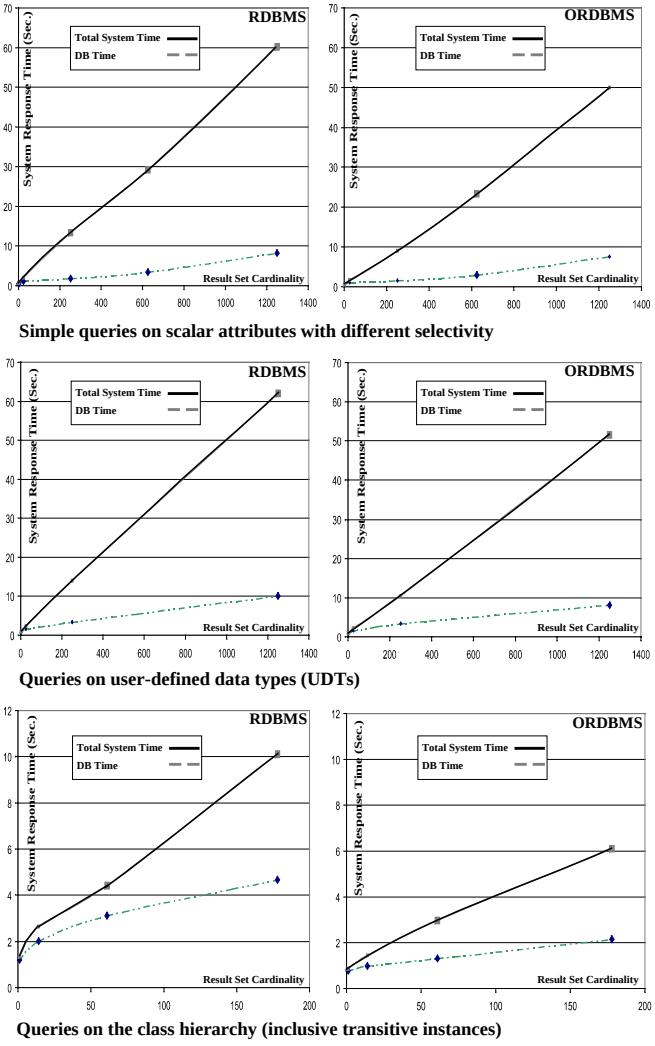

Fig. 3: Measurement Results I

In the purely relational case, it is not possible to map a (complex) class to exactly one table. Mapping set-valued attributes, aggregations and (m:n)-relationships properly demands several tables interconnected by primary/foreign keys. This, in turn, requires to pose several SQL queries in order to perform one query at the OOPL level implying a higher DB time and higher communication costs. Compared to a semantically equiva-

lent OR mapping, the more expensive query transformation and the necessary reconstruction of object structures in the pure relational mapping reduce the system efficiency additionally. This effect can even be reinforced if further OR features such as function-based indexing or index structures over class hierarchies, which can be tailored to OO applications, are used in order to further improve the OR mapping.

Since in our measurements both client and server ran on the same machine, the costs of communication between the database server and the mapping layer are relatively small. It is to be expected that a distributed client/server architecture will considerably enlarge the difference between these measurement results. Furthermore, the more complex the data structures, the more additional overhead is to be expected in the relational mapping. As reported in former measurements [6, 25], some OR systems performed in some cases even worse than semantically equivalent relational systems. Due to our examinations, we think that this statement has to be revised and ORDBMSs, in the meantime, have obviously become more and more mature.

5.2 Performance Characteristics of the Mapping Layer

In order to characterize the performance of the mapping layer adequately, we have investigated simple queries with different selectivity. The results are presented in Fig. 4. As already mentioned in section 2, the DB APIs of almost all currently available ORDBMSs do not support navigational access directly. Hence, a navigational operation, such as *GetObject(**Ref**)*, must be transformed to a database query (SQL:1999), such as "*Select * From... Where OID = **Ref***", by the mapping layer. Such a query strategy, especially when intensively dealing with navigational operations as usually required by most OO applications, obviously leads to high processing overhead spent in the database system as well as very high communication costs (over 70% of the entire system time, Fig. 4, left-hand side). The (probably not very astonishing) observation is that the traditional query strategy is not adequate for supporting navigational access. As we can see at the right-hand side of Fig. 4, the DB time of set-oriented queries shows only a slight ascent with increasing result sets, while the additional mapping overhead increases rapidly. When retrieving 1250 objects, the time spent at the mapping layer even exceeds 86% of the total system time (see Fig. 4, right-hand side). This observation can be explained as follows. In the

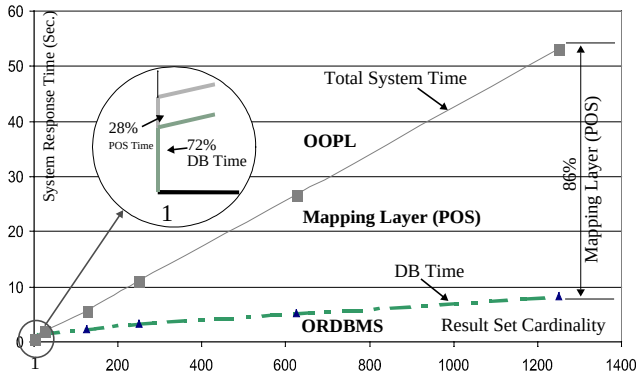

Fig. 4: Measurement Results II

early days of ORDBMSs, these systems comparable to RDBMSs were not very successful in supporting navigational access, but excellent in processing set-oriented access (as they are still today). Unfortunately, OO applications can hardly benefit from this advantage, because the ORDBMS API is 'inherited' from traditional RDBMSs and, therefore, still only supports simple, flat data. In lack of an *extensible* DB API which may generically support complex data types defined by the user, complex objects in an ORDBMS have to be first 'disassembled' into scalar values, and afterwards reconstructed (at the mapping layer) to objects of a certain class in the particular OOPL. This kind of overhead gets dramatic with increasing result set cardinality and impairs the entire system efficiency significantly.

Regarding these measurement results, we can draw the following conclusions. In order to be able to support navigational access better, the ORDB API should directly support navigational operations like *GetObject(Ref)*, so that the costs of transforming navigational operations to SQL queries and for evaluating these queries can be avoided. Furthermore, it should also support the notion of complex objects directly and offer the possibility of retrieving complex objects as units. According to our examinations, such improvements can increase the entire system efficiency by up to 400%.

5.3 Support for Complex Objects

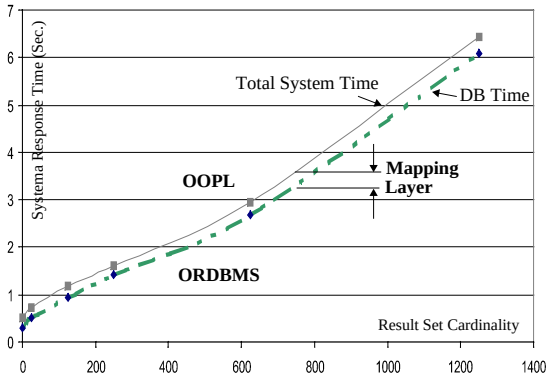

Fig. 5: Measurement Results III

As already mentioned before, the lack of direct support for complex objects and navigational access at the DB API level extremely impairs the overall system efficiency. Fortunately, a leading ORDBMS vendor already offers an extended call level interface, which, as we can see later in this section, directly supports navigational access as well as retrieval of complex objects as units, and, in addition, even retrieval of complex object graphs as units. Navigation is enabled by the possibility of autonomously retrieving complex structured objects (by OID) as instances of C structures. This simplifies the mapping to OOPLs, such as C++, considerably and, therefore, is undoubtedly the first step into the right direction, although this mechanism does not yet support the actually wanted seamless coupling (transparent transformation from a database object to an instance of an OOPL class). The mentioned support for complex objects at the level of the DB API allows to directly retrieve a complex object's data from the database into the main memory by specifying its OID or a predicate. Therefore, the expensive query processing strategy described in section 5.2 can be

avoided. Remind that the measurements described in section 5.2 have been performed on an ORDBMS that does not possess a DB API as the one described in this section. To show the importance of and the corresponding demand on a suitable support for complex objects at the DB API level, we repeated the measurements described in section 5.2 on the ORDBMS referred to in this section and providing the mentioned complex object support at its API. Fig. 5 illustrates the measurement results. Obviously, the additional overhead spent at the mapping layer is now independent from the cardinality of the query result sets. Thus, the direct support of complex objects at the DB API results in a clear performance gain (up to 400%).

The direct support for navigational access at the DB API level mentioned in this section, allowing to directly access objects by calling a function like *GetObject(Ref)*, avoids expensive processes (query transformation, data types conversion and object reconstruction). This obviously contributes to improve performance significantly. Fig. 6a shows a comparison between a query strategy (transforming a navigational operation to an SQL query) and a navigational strategy (directly calling a *GetObject(OID)* function at the DB API). The advantages of the navigational strategy are obvious. With a direct support of navigational access the entire system efficiency increases by approximately 200%.

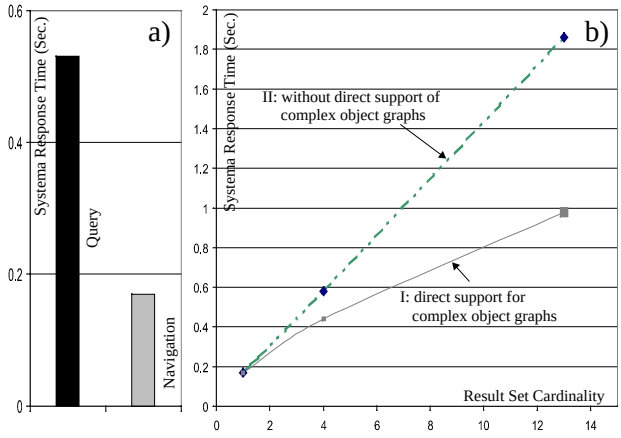

Fig. 6: Measurement Results IV

OO applications often want a set of objects interconnected by object references (object graph) to be retrieved completely within just a single database interaction. Fig. 6b shows a comparison of two strategies for retrieving complex object graphs. In this measurement, we used the ORDBMS directly supporting navigation as well as retrieval of complex object graphs. It can be seen clearly that strategy I exploiting the ability of retrieving object graphs exhibits a performance gain of about 100% already at a result set cardinality of 13 objects.

The design of a new DB API, which directly supports complex structured objects, is by no means an easy job and requires generic design methods, because user-defined data types can be arbitrarily structured, e. g., contain other complex data types, such as UDTs, references and set-valued attributes. Furthermore, a DB API has always to be multi-lingual requiring to support all common programming languages simultaneously and, therefore, making it very difficult to offer the best of both worlds (DBMSs and OOPLs) without any compromises.

6 Conclusions and Outlook

In this paper, we have emphasized the importance of assessing ORDBMSs w. r. t. their capabilities of supporting OOPLs. We have first qualitatively considered ORDBMSs and OOPLs regarding modelling and operational aspects relevant for object-oriented software development. As the object-relational (SQL:1999) and the object-oriented data model are essentially coming together, the operational distance between these two paradigms is still considerable so that an additional mapping layer is necessary to overcome this gap. Regrettably, such an additional layer impairs the performance of the overall system considerably. Additionally, we performed quantitative examinations (measurements) in order to assess ORDBMSs in their capabilities of supporting OOPLs. Indirectly, these measurements are supposed to contribute to promoting the optimal utilization of currently available ORDBMSs in object-oriented system development and to guide the future development of ORDBMSs in a way that the support for OOPLs is improved.

Regarding our performance examinations, we have motivated the necessity of an open, configurable benchmark approach, because not only the performance of ORDBMSs themselves but also the additional overhead, which is necessary for bridging the conceptual and operational distance between ORDBMSs and OOPLs, have to be taken into account and, therefore, properly characterized. It has been clearly illustrated by our performance measurements that object-relational database technology gets more and more mature, not only conceptually (data model, query processing), but also w. r. t. performance. The model facilities contribute to keep the mapping layer ‘thin’ in contrast to RDBMSs. This, on one hand, reduces the implementation efforts, and, on the other hand, increases the entire system efficiency. Despite the available object-oriented extensions, which entail an unambiguous gain in comparison to RDBMSs, the potential benefit of object-relational database technology in our opinion is not yet exhausted, since the traditional DB API is so far not capable of successfully supporting object-oriented principles. The DB API of almost all ORDBMSs still can not support navigational operations and complex object structures directly so that new generation software systems can not take advantage of object-relational database technology in an optimal way. It can be called the ‘bottleneck’ between the object-relational and object-oriented paradigms. Our examinations have shown clearly that a new DB API directly supporting navigational operations and complex objects is necessary. Obviously, it is not easy to equip ORDBMSs with such a new interface. For example, such an interface must be multi-lingual. For answering the question, how user-defined data types can be effectively represented at the OOPL level, further research efforts are required. Altogether, the problem of a seamless and effective mapping of SQL:1999 to OOPLs, such as C++ and Java, has to be worked on further. Our future work will mainly be looking for possible solutions. Furthermore, we plan intensive studies of ORDBMS capabilities for supporting navigational access, because ORDBMSs are still behind OODBMSs in this concern. Generally, after having characterized the performance aspects in more details, our

long-term objective is to develop applicable concepts contributing to increase the performance of ORDBMSs.

References

1. Bernhard, R., Flehmig, M., Mahdoui, A., Ritter, N., Steiert, H.-P., Zhang, W.P.: "*Building a Persistent Class System on top of (O)RDBMS - Concepts and Evaluations*", Internal Report, University of Kaiserslautern, 1999
2. Bernstein, P.A., Harry, B., Sanders, P.J., Shutt, D., Zander, J.: "*The Microsoft Repository*", Proc. VLDB Conf., 1997, pp. 3-12
3. Bernstein, P.A., Pal, S., Shutt, D.: "*Context-Based Prefetch for Implementing Objects on Relations*", Proc. VLDB Conf., 1999, pp. 327-338
4. Bitton, D., DeWitt, D.J., Turbyfill, C.: "*Benchmarking Database Systems: A Systematic Approach*", Proc. VLDB Conf., 1983, pp. 8-19
5. Carey, M.J., DeWitt, D.J.: "*Of Objects and Databases: A Decade of Turmoil*", Proc. VLDB Conf., 1996, pp. 3-14
6. Carey, M.J., DeWitt, D.J., Naughton, J.F., Asgarian, M., Brown, P., Gehrke, J.E., Shah, D.N.: "*The Bucky Object-Relational Benchmark*", Proc. VLDB Conf., 1996, pp. 135-146
7. Carey, M.J., DeWitt, D.J., Kant, C., Naughton, J.F.: "*A Status Report on the OO7 OODBMS Benchmarking Effort*", Proc. ACM OOPSLA, 1994, pp. 414-426
8. Carey, M.J., Doole, D., Mattos, N.M.: "*O-O, What Have They Done to DB2?*", Proc. 1999 25th. VLDB Conf., pp. 542-553
9. Cattell, R.G.G., Barry, D., Bartels, D., et al: "*The Object Database Standard:ODMG 2.0*", Morgan-Kaufman Publishers, San Mateo, 1997
10. Gray, J.: "*The Benchmark Handbook for Database and Transaction Processing Systems*", Morgen Kaufmann Publishers, San Mateo, CA, USA, 2nd Ed., 1993
11. Gulutzan, P., Pelzer, T.: "*SQL-99 Complete, Really*", R&D Publications, 1999
12. Keller, A., Jensen, R., Agrawal, S.: "*Persistence Software: Bridging Object-Oriented Programming and Relational Database*", Proc. ACM SIGMOD Conf., 1993, pp. 523-528
13. Mahnke, W., Steiert, H.-P.: "*The Application Protential of ORDBMS in the Design Environments*", Proc. CAD 2000, Berlin, pp. 219-239 (in German)
14. Ontos Business Data Server, *http://www.ontos.com*
15. Poet Object Server, POET Software, POET SQL Object Factory, *http://poet.com/*
16. Rao, B.R.: "*Object-oriented Databases: Technology, Applications, and Products*", McGraw-Hill, New York, 1994
17. RogueWave Software, DBTools.h++, *http://www.roguewav3e.com/products/dbtools/*

18. Schreiber, H.: "*JUSTITIA: A Generic Benchmark for the OODBMS Selection*", Int. Conference on Data and Knowledge Systems in Manufacturing and Engineering, Tokyio, 1994
19. Scheller, T.: "*Functionality of the Class System in System R/3*", Function Description, Version 0.9, SAP, Dec. 1997
20. SQL99: ANSI/ISO/IEC 9075-1-1999 Database Launguages SQL Part 1 Framework
21. SQL99: ANSI/ISO/IEC 9075-2-1999 Database Languages SQL Part 2 Foundation
22. Stonebraker, M., Brown, P., Moor, D.: "*Object-relational DBMSs - The Next Wave*", Morgan Kaufmann, 2nd Ed., 1998
23. TPC: Transaction Processing Performance Council, Standard Specification 1.0, May 1995, *http://www.tpc.org*
24. UML, Rational Software Corp. Unified Modeling Language, *http://www.rational.com/*
25. Zhang, W.P.: "*Evaluation of the First Generation ORDBMSs by Using Bucky Benchmark*" Internal Report, University of Kaiserslautern, 1998
26. Zhang, W.P., Ritter, N.: "*Measuring the Contribution of (O)RDBMS to Object-Oriented Software Development*", Proc. IDEAS 2000, pp. 243-249