Measuring the Contributions of (O)RDBMS to Object-Oriented Software Development

W. P. Zhang, N. Ritter

University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
e-mail: wpzhang/ritter@informatik.uni-kl.de

Abstract

In this paper, we describe a new benchmark approach which is needed for two reasons. First, today's computer systems almost exclusively use relational database management systems (DBMS), and, besides (purely) relational DB technology, only the new trend of object-relational technology is expected to obtain practical relevance. Second, today's (complex) software systems are developed in an object-oriented way. Thus, a benchmark approach is needed assessing the contributions of (object-)relational DBMS to object-oriented software development. Consequently, such a benchmark does not only have to 'measure' the efficiency of the DBMS itself, but does also have to take the overhead to be spent for mapping object-oriented data structures to the DBMS interface into account. We describe how our approach achieves this goal and report on first measurement results.

1. Introduction

Nowadays, complex software systems usually embody one or more DBMS in order to achieve persistency, consistency and protected concurrent access to the data to be maintained as well as being able to exploit powerful query facilities. Actually, only relational DBMS (RDBMS) have a practical relevance today and only the successor generation of RDBMS, object-relational RDBMS (ORDBMS, [21]), is expected to be similarly successful.

A second observation is that nowadays software is developed in an object-oriented way. The consequence is that the data to be managed by the underlying (relational or object-relational) DBMS are given as classes organized within class hierarchies (inheritance, polymorphism, etc.) as well as instances of these classes.

It is well known that RDBMS are not very well suited to fulfil this task, since the relational data model and the data model of the object-oriented programming languages (OOPL) used in modern system development, e.g., Java or C++, are rather different. An additional layer in between OOPL and RDBMS is needed which is causing additional overhead and, therefore, has to be taken into account, when assessing the efficiency of RDBMS regarding these object-oriented applications.

ORDBMS [21], although they are aiming at an integration of the object-oriented and the relational paradigm, require a mapping layer either, since neither the SQL derivatives of currently available commercial DBMS nor the new SQL:1999 standard [23, 9] allows to treat DB objects in a way comparable to the handling of objects in OOPL. For sample mismatches see [14].

Obviously, if the efficiency of an (O)RDBMS regarding a software system, which is to be developed newly in an object-oriented way, is to be captured quantitatively, the following two questions are to be answered. How efficient is the DBMS in answering representative queries? Which overhead is to be expected for mapping the object-oriented structures needed at the OOPL level to the DB interface? While the first question can be answered by using a traditional benchmark [8], the second one requires further examinations. Our approach aims at contributing to answer the second question, too, by providing a corresponding new benchmark approach. This is the central topic of this paper, which is organized as follows. Section 2 outlines our project and discusses new benchmark requirements. Afterwards, we report on an implemented benchmark system, which can be considered as a first step on the way towards the benchmark system we are aiming at. While Sections 3 and 4 describe the corresponding system, Section 5 gives some indicative measurement results. Section 6 concludes the paper and gives an outlook to the work to be done in the future in order to reach the goal set.

2. Our approach

The work, this paper reports on, is part of the *SENSOR* project which, in turn, is part of the SFB 501 *Development of Large Systems with Generic Methods* founded by the *German Science Foundation*. SENSOR [20] deals with the support of software development processes by object-relational database technology. SERUM [13], a subactivity of SENSOR, aims at providing ORDB applications (or, in general terms, software systems) with generic methods.

Starting from a UML specification given by some user the SERUM generator is supposed to generate major parts of the system to be developed (e.g., a certain persistent object management system). The latter may consist of

- DBS extensions, a commercially available ORDBMS can be extended by, e.g., user-defined data types (UDT), user-defined routines (UDR), user-defined index structures (UDI), etc.;

0-7695-0789-1/00 $10.00 © 2000 IEEE

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

- application server modules, i.e., application functions the DB server cannot be extended by, and, therefore, are to implemented on top of the DBS interface;
- and an API allowing the actual application (which may also be a superordinated software system) to access the DB application system.

For further explanations of the SERUM approach see [13]. Generally, we found it being an interesting question, how w.r.t. the situation mentioned in Section 1 the behaviour of a DBMS in such a logical application server architecture can be assessed empirically. Thus, given a certain DBMS to be assessed, the benchmark system to be designed has to be capable of:

- providing a typical application scenario as every benchmark approach has to;
- w.r.t. the DBMS to be considered finding the most efficient way of implementing the application functionality either as DBS extensions (as far as the DBMS provides a corresponding infrastructure) or as an additional layer on top of the DBMS interface.

Obviously the second point is crucial and we currently try to find rules saying how to implement a given benchmark scenario (representative application) on a given DBMS exploiting the capabilities of this DBMS in the best way possible.

As a first step on the way towards achieving the mentioned objectives (new benchmark system design), we have already implemented the benchmark system, the remainder of the paper is reporting on.

In order to take requirements of object-oriented software development into account, we have chosen a so-called *class system* as the application scenario underlying the benchmark. This class system provides basic functionality for the management of class hierarchies and corresponding instances (persistent objects). The object model of the class system and its mapping to (O)RDBMS are discussed in the following section.

3. The Class System

3.1. Object Model

First, we introduce an object model which is comparable to those commonly used [2, 7, 25]. Each *class* defined in the class system can be derived from one or more other classes (*Multi-Inheritance*). In fact, a class implements at least one *object type* that is associated with a name and a set of *attributes*. Each *attribute* conforms to a particular *data type* and is either *single-valued* or *set-valued* (*collection type*). A *data type* can be scalar, i.e., integer, string or boolean. It can also be a complex data type defined by users (*user-defined data type*, UDT) or an object-valued data type (*object reference*). The latter used in a diametric way constitutes an object relationship. Each persistent *object* conforms to an *object type* and is described by its *state* which consists of a set of *attribute values*. Additionally, each object has a unique object identifier (OID).

Typically, navigational as well as set-oriented, descriptive [15, 10] access is needed to the objects managed within the class system. Descriptive access operations usually do not only return simple, flat relations as query results, but deliver complete objects or even complex structures consisting of several objects and relationships in between [15, 10].

Referring to our objectives, as presented in Section 2, we now consider the class system, as outlined in this section, as the application scenario to be implemented on the DBMS to be 'benchmarked'. We have implemented the class system on one of the leading, currently available commercial DBMS, once by exclusively exploiting relational features and a second time by strictly exploiting the provided object-relational extensibility infrastructure. Thus, the corresponding measurements, which will be presented in Section 5, allow to assess the object-relational features in comparison to a pure relational approach. Both implementations required an additional layer on top of the DBMS interface, i.e., the object-relational infrastructure did not allow to completely realize the class system functionality by DBMS extensions.

Generally, a method invocation at the programming interface offered by the class system must be transformed into one or several SQL queries which, in turn, are to be delegated to the backend DBMS. By a proper mapping not only the overhead of client/server communication (between the DBMS and the class system), which could considerably influence the overall system performance, can be reduced, but also the DBMS can be given the opportunity to optimize queries, so that the costs of evaluating the queries and loading the data of qualified objects can be kept low.

How a method invocation is to be transformed into an equivalent set of SQL queries is decisively depending on how the data structures have been mapped. Section 3.2 deals with the mapping of structures, while Section 3.3 details the query transformation. In order to distinguish the relational from the object-relational implementation, we use the terms *System R* and *System OR* in the following.

3.2. Mapping of Structures

First of all, we consider *System R*. Commonly, classes are mapped to relational tables, called *class tables* in the following. Objects are represented by table rows. Since RDBMS 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. Thus, several tables, may be required to map a given class. The class system manages metadata about a user-defined model (class hierarchy) within a corresponding metadata management system. These metadata are used to map classes to relations by observing the following rules:

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

- For each class a class table is created; its columns represent single-valued, scalar class attributes, e.g., of a numeric, string, or boolean data type.
- OIDs correspond to primary keys.
- Each set-valued attribute is mapped to an additional table referring to the class table via foreign key.
- User-defined data types (UDTs) are also mapped to one or, in case of a nesting of complex data types, to several additional table(s), which, again, are connected by PK/FK (primary key / foreign key) pairs.
- Relationships are represented by PK/FK pairs. In the case of (m:n)-relationship types, an additional table has to be used.

Principally, there are three different ways of representing a class hierarchy in the relational model, i.e., *horizontal partitioning*, *vertical partitioning*, and the *hierarchy table*, comprehensively discussed in [5]. After studying pros and cons, we decided to adapt the horizontal partitioning approach to our needs, since it provides good performance in most cases, and is also used in most commercial products [2, 12]. Descriptions of some other commercial products that map objects to relations can be found in [16, 18, 19].

Since the data models of ORDBMS and OOPL are coming closer, the mapping became easier in *System OR* (compared to *System R*). Besides the extensibility features already mentioned in Section 2, the object-relational data model [9, 22] directly supports object-oriented concepts, e.g., complex objects, set-valued attributes, and object references (REF). Consequently, besides class tables no additional tables are needed. Furthermore, the object-relational type system allows users to define type hierarchies and corresponding table hierarchies. Regretfully, commercially available ORDBMS currently only support single inheritance. Thus, in order to provide the multiple inheritance offered by the class system (see Section 2), the mapping layer has to explicitly control the semantics of inheritance relationships. Current ORDBMS also do not support object relationships by means of diametric references. Consequently, maintaining (diametric) references (incl. referential integrity control) is also task of the mapping layer.

3.3. Query Transformation and Execution

Besides commonly used mechanisms, such as object caching and context-based object prefetching which serve to reduce the impedance mismatch between the navigational OOPL and the set-oriented query language of (O)RDBMS and to reduce communication overhead, the performance of the entire system primarily depends on the supported query execution strategy.

Queries issued at the class system's interface (CS queries) are assumed to have the following skeleton:

SELECT ALL [DIRECT | TRANSITIVE]
FROM <*class*> WHERE <*qualification*>;

The semantics is that all (either direct or transitive) instances of the specified class fulfilling the specified condition are retrieved. Due to space restrictions, we cannot detail the query language. We just want to mention that the qualification may contain conditions on (direct or inherited) attributes of the specified class or of other classes that can be reached from the specified class via path traversals.

A CS query has to be transformed by the mapping layer into an equivalent set of SQL queries, which are to be issued to the underlying DBMS. The results of the SQL queries are to be synthesized by the mapping layer in order to create the CS query's result set.

In general, the following steps are to be performed, in order to execute a CS query:

- *Normalization*: The CS query has to be transformed into an internal representation being the prerequisite of deriving corresponding SQL queries.
- *Transformation*: A set of SQL queries which is semantically equivalent to the given CS query is derived. This is a very critical step, since (SQL) queries must be expressed in a way allowing the underlying DBMS to accurately exploit its optimization capabilities.
- *Execution*. The SQL queries are executed by the underlying DBMS. This step is also critical, because the order of delegating the SQL queries may influence the system performance.
- *Merge*: The data delivered by the DBMS as results of the SQL queries must be merged in order to establish the result set of the CS query.

Query Execution in System R

Regarding that in *System R* a class is usually represented by several tables (see Section 3.2) different strategies might be applied. The first one is to map a CS query to exactly one SQL query, which can be expected to contain many joins. Furthermore, due to the feature of set-valued attributes (see Section 3), the result set of this SQL query may contain highly replicated data, i.e., several tuples may contain data of the same object just differing in the values of the set-valued attribute(s). The mapping layer would have to unfold these tuples again in order to reconstruct objects. Obviously, the overhead for this strategy is too high.

We distinguish two further strategies both splitting the CS query up into several SQL queries. In the first strategy, several queries (one for each table needed to map the queried class) are used to retrieve the data needed to construct the CS query's result set, each containing the complex join operation mentioned above for qualification purposes. The second strategy first executes an SQL query on the class table in order to determine the OIDs of the CS query's result set. Afterwards this OID list is used within a number of subsequent SQL queries retrieving the remaining object data.

Although, this second approach decreases query execution costs, it increases the client/server communication, if

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

the number of qualified objects is high. We have tested both strategies and observed that the second scheme exhibits a slim performance benefit at small result sets, whereas the first method behaves much better at result sets of more than 100 objects. Since none of these two strategies can be considered to be the most efficient in all possibly occurring situations (w.r.t. cardinality of result set, query complexity, network bandwidth, etc.) we decided to implement both strategies. All measurement results given in Section 5 refer to the best strategy in each special case, respectively.

Query Execution in System OR

Since *System OR* mapps a class to exactly one table, a CS query can easily be mapped to a single SQL query (see SQL:1999 standard, [23, 9]). This, on one hand, allows the DBMS to accurately exploit its query optimization capabilities, and, on the other hand, to reduce client/server communication overhead. The result set of the SQL query contains exactly one entry for each object belonging to the result of the CS query, so that the merge overhead is minimal, too.

4. The Benchmark

The performance of RDBMS or ORDBMS have 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* [24] as well as the new *Bucky benchmark* [4]. These benchmarks are very suitable for comparing different DBMS with each other [8]. However, none of these benchmarks helps to assess the contributions of a DBMS to object-oriented software development. Consequently, these other approaches do not take into account the typical application server architecture and the fact that the DBMS capabilities determine the overhead of the application/mapping layer. This is also true for the OO7-Benchmark [6], although it deals with object-oriented database languages.

Although the design of the mapping layer depends on the DBMS used, we expect that many of the modules we have implemented can be easily *adapted* to be used with DBMS different to those we have tested so far. In order to provide this reusability, we designed the benchmark modules in a strictly object-oriented way. Furthermore, in order to reflect the different requirements of the broad range of applications this benchmark may be applied to and to allow user to easily refine and improve the prototype, our benchmark system has been designed to be easily *configurable* and *parameterizable*. Thus, the properties of the benchmark database can be controlled by several parameters and the benchmark queries are not fix, but can be instantiated from an extendable set of query templates. The latter allows to flexibly create a set of queries reflecting typical application characteristics.

4.1. Architecture of the Benchmark System

Figure 1 shows the major components of the benchmark system. This system supports a parameterizable database population, that means, users may specify parameters for the number of classes, the number and distribution of instances, the number and distribution of super-/subclasses of a class, etc., which are taken into account by the database population performed by the *data generator* in the following steps:

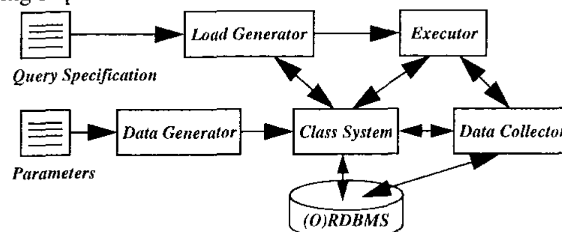

Figure 1: Architecture of the Benchmark System

- *Establishing class hierarchies*: according to the number of root classes and the total number of classes, as specified by the user, empty (the attributes are associated in the subsequent step) classes are created and arranged within corresponding class hierarchies. These are spanned by inheritance relationships, which are created by observing certain statistical distributions [1].
- *Generating and assigning attributes*: In order to obtain a proper workload, scalar attributes as well as attributes of complex (UDT) and reference types have to be generated and assigned to classes. Attributes can be optionally single-valued or set-valued.
- *Generating instances*: Instances are generated and associated with existing classes, again by observing certain statistical distributions [1].

Regarding typical application scenarios, a user may select from a given set of query templates and indicate how many times each template is to be instantiated. Accordingly, the *load generator* creates a set of CS 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 costs, i.e., amounts of time spent, they want to be measured, e.g., costs for CS query transformation, SQL query evaluation and data loading, and result set construction. Corresponding values are collected by the data collector during execution of the query set and afterwards written to the DBS in order to save them for further evaluations.

4.2. Queries

Naturally, different kinds of queries are predefined in order to get a complete performance evaluation. All queries conform to the skeleton given in Section 3.3. They differ in their qualification predicates and, therefore, in their selectivity. The benchmark comes with a predefined set of 48 query templates [1]. The templates can be parameterized with a class name, in order to instantiate a query. The user may directly specify the class name or may instruct the load generator to randomly assign class names.

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

Retrieval queries are organized in the following categories:

a) ***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.

b) ***Predicates on set-valued attributes***: This group contains queries with simple predicates (a single IN operation) on nested sets.

c) ***Predicates on UDTs***: This group contains queries with simple predicates (a single comparison operation) on an attribute with a structured, non-atomic data type.

d) ***Path predicates***: This group contains queries evaluating simple predicates after path traversals.

e) ***Complex predicates***: Queries of this group contain complex predicates challenging both query transformation as well as query optimization.

f) ***Queries on the class hierarchy***: While all other queries exclusively deliver direct instances of the queried class, these queries deliver direct and transitive instances. Predicates conform to those of the first category.

In addition to retrieval operations, insert operations are considered. The system performance w.r.t. insertion of large numbers of objects is measured when populating the database (batch insert).

5. Measurements and Results

5.1. Measurements

We performed our measurements on a database with 250000 instances. 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. Of course, we defined index structures appropriate to support the evaluation of the benchmark queries in both systems (*System R* and *System OR*).

To study and compare the performance of System R and System OR, we ran all measurements on a stand-alone SUN Enterprise 450 Server with four 300 MHz SPARC processors, 1 GB main memory and 36 GB hard disk, and the operating system Solaris 2.7. As DBMS we used one of the leading commercially available object-relational systems. In System R exclusively the relational features were exploited, whereas in System OR the object-relational features were extensively used. Between two consecutive measurements, the server was rebooted and restarted to ensure that no related data is cached in the database buffer or in the Unix buffer pool. We ran both the class system and the database server on the same machine. Within each measurement, it was guaranteed that each query refers to a different class in order to prevent the related data from being cached in the database buffer. The metadata was loaded by the class system before starting the measurements and cached in its buffer during the entire test. This means that each query was executed "*warm*", but never "*hot*".

We measured the database time (DB time) and the total system time (TS time). The DB time of an SQL query is the time in between delegating the query to the DBMS and receiving back the results. 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 set. This has to be taken into account, when analysing the measurement results. A CS query's DB time is the sum of the DB times of all corresponding SQL queries. The TS time is defined as the total elapsed time from issuing a CS query until having received the complete result set. It contains the time spent within the class system (mapping layer) as well as the DB time.

5.2. Results and Evaluations

The database population, i.e., generating and inserting 250000 instances (batch insert), required 6 hours and 12 minutes in *System R* and 5 hours and 43 minutes in *System OR*. Note that *System OR* only needs a single SQL statement to insert an instance, whereas *System R* needs several SQL statements (one statement for each table) increasing the DB time, the client/server communication, and, therefore, the total costs as well.

Table 1 lists the results of testing the retrieval queries (TS time and DB time in seconds, respectively). Regarding queries with simple predicates on scalar attributes (category a), both DB time and TS time in *System OR* are generally better than in *System R*. However, the difference in the case of a small result set (query 1, 8, 9, 10, 11, 14, 19) is tiny.

As reported in [4, 26], RDBMS perform much better than ORDBMS as far as simple predicates on scalar attributes or set-valued attributes are concerned, while ORDBMS, on the other hand, perform better in executing queries involving function invocations on abstract data types and in processing dereferencing operations. Regarding our results, we see that this statement has to be revised. Especially in handling predicates on set-valued attributes (see Table 1, block b) ORDBMS have obviously become gradually more mature, since there is nearly no difference between the two systems.

UDTs are modelled as *named row types* in *System OR* and as separate tables in *System R*. The results of corresponding queries (listed in Table 1, block c) show less difference between the two systems (just slight advantageous for *System OR*). The reason might be that the ORDBMS-internal UDT management is similar to the out-of-line storage within relational systems (separate tables for UDTs).

Regretfully, the ORDBMS used in our measurements currently does not support references. Most, currently available ORDBMS lack in this feature. Consequently, we could not test our queries of category d. Nevertheless we want to

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

mention that object references, which allow to directly model (n:m)-relationships between object types, can be very helpful to simplify the implementation of the class system, since they highly improve the performance of navigational data access [4, 26]. So, w.r.t. queries of category d, future ORDBMS versions offering references can be expected to show a much better performance than an RDBMS.

			System R		System OR	
Cat.	Query	Result Set Cardinality	TS Time	DB Time	TS Time	DB Time
a	01	1	0.64	0.48	0.63	0.48
	02	1251	60.10	8.23	50.800	7.91
	03	620	28.05	3.43	23.28	2.91
	04	253	13.31	1.87	9.09	1.51
	05	121	5.61	1.46	5.44	1.40
	06	25	2.04	1.09	1.73	1.05
	07	2	1.13	0.78	1.13	0.77
	08	1	0.66	0.52	0.69	0.52
	09	0	0.37	0.33	0.32	0.31
	10	0	0.26	0.22	0.26	0.25
	11	1	0.79	0.48	0.79	0.49
	12	250	12.74	1.78	10.08	1.81
	13	24	1.93	1.16	1.88	1.13
	14	3	1.16	0.74	1.18	0.72
	15	250	13.32	1.84	9.82	1.71
	16	249	12.67	1.81	9.53	1.72
	17	246	12.62	1.79	9.05	1.72
	18	25	2.32	1.52	2.21	1.53
	19	3	1.18	0.82	1.17	0.81
b	20	1	0.70	0.56	0.71	0.55
	21	1	0.74	0.59	0.73	0.60
c	22	1	0.94	0.73	0.78	0.50
	23	1259	62.09	9.12	51.27	9.00
	24	236	14.19	3.45	11.93	3.24
	25	26	2.34	1.57	2.29	1.51
	26	2	1.26	0.82	1.59	0.81
	27	0	0.32	0.30	0.36	0.28
	28	246	14.01	3.38	11.59	1.80
	29	27	2.66	1.53	2.57	1.50
	30	3	1.22	0.84	1.41	0.81
e	40	315	18.84	4.27	14.24	3.92
	41	40	3.96	2.45	3.29	2.28
f	42	1	1.35	1.21	0.87	0.77
	43	178	10.12	3.45	6.10	2.16
	44	21	3.46	2.35	2.51	1.12
	45	14	3.72	2.45	2.27	1.12
	46	3	1.75	1.59	1.38	1.09
	47	61	4.43	3.11	2.97	1.32
	48	17	3.69	2.41	2.27	1.14

Table 1: Measurement Results

Queries with complex predicates (category e) are handled very well in both systems. Since the DB time values do not differ very much, the conclusion can be drawn that ORDBMS optimizer have become as mature as their pure relational predecessors. Most TS time values point up again that the mapping layer of *System R* is more costly than the one of *System OR*.

The values Table 1 contains in block f, clearly indicate that there is a considerable difference between *System R* and *System OR* w.r.t. queries against the class hierarchy. The reason is obvious. In *System R*, due to the relational mapping, several SQL queries, at least one for each class in the considered hierarchy, have to be issued. In contrast, the object-relational mapping requires only one query (except for multi-inheritance) to load all the data of qualified objects. This affirms our conjecture that the more complex the object structure and the class hierarchy, the higher the performance difference between the two systems.

Generally, the measurement results illustrate that *System OR* performs better than *System R* by an average factor of about 20 %. The reasons are twofold. First, the object-oriented features provided by the ORDBMS contribute to keep the complexity of the mapping layer simple (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 gains when using these object-oriented features. Starting from these general observations, we intend to further develop our benchmark approach and test more ORDBMS. This way we want to provide a benchmark, which is well suited by means of

- assessing the modelling capabilities w.r.t. a usage of the DBMS within object-oriented system development;
- assessing the overhead to be spend within a mapping layer adequately supporting object-oriented system development;
- assessing the accuracy of the DBMS-internal realization of object-oriented features.

6. Conclusions and Outlook

In this paper, we reported on our first step towards a new benchmark approach, which, as we think, is needed to assess the efficiency of (O)RDBMS w.r.t. their usage in modern software systems which are developed in an object-oriented manner. Thus, the accuracy of (O)RDBMS for being accessed from OOPL is a new crucial issue and, therefore, must be considered to be a major concern of new benchmarking. Obviously current benchmark approaches, e.g., the Bucky benchmark [4], do not fulfil this task, since they do not sufficiently take the inherent characteristics of the mentioned application class into account. For example, if an application layer is needed to fulfil the application's data management needs the corresponding overhead, as we think, has to be taken into account as well.

Although object-oriented DBMS propose a more seamless coupling of database language and OOPL than (O)RDBMS, they did not achieve practical relevance. Therefore, (O)RDBMS must be considered to be the kind of DBMS, which have to be assessed by the new benchmark approach, we are aiming at. Many recent publications, e.g., [2, 15, 10, 12], affirm that there is a growing demand for of-

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

fering persistent object interfaces on top of (O)RDBMS.

We described the prototypical implementation of a benchmark system serving as a starting point for future work. We used this system to compare the potentials of the (pure) relational facilities and the object-relational facilities of a leading, commercially available ORDBMS. Our measurements have shown that the system exploiting the object-relational features performs better, especially regarding complex queries with large result sets.

Furthermore we argued that a benchmark system must flexibly be adaptable in order to support the huge class of applications in mind. The system we have implemented, already provides this feature, by allowing

- to parameterize the database population algorithm,
- to parameterize and select from query templates,
- to add new query templates, and
- to set measurement points, i.e., to decide which measurement data is to be collected.

In the near future, the current approach has to be refined by answering the following questions:

- Is the application scenario (class system and corresponding query templates) the best possible by means of being really representative?
- Are there any rules saying what the best way is to implement the application scenario on top of a given ORDBMS?

Answering the second question is crucial. It encompasses guidelines to effectively exploit the modelling and extensibility features of the ORDBMS and to make the application server (mapping layer) as efficient as possible.

After having answered these questions, the resulting benchmark system is supposed to compare the performance of ORDBMS w.r.t. modern application scenarios. These performance evaluations, in turn, are supposed to contribute to improving ORDBMS, so that eventually no additional mapping layer will be needed, at least as far as the implementation of basic data management functionality is concerned. Of course, application servers will further on be needed in order to provide scalability and other non-functional properties.

Acknowledgements

The authors would like to thank H.-P. Steiert, M. Flehmig and T. Härder for their support.

References

[1] R. Bernhard, M. Flehmig, A. Mahdoui, N. Ritter, H.-P. Steiert, W. P. Zhang: *"Building a Persistent Class System on top of an (O)RDBMS - Concepts and Evaluatons"*, Internal Report, University of Kaiserslautern, 1999.

[2] P. A. Bernstein, B. Harry, P.J. Sanders, D. Shutt, J. Zander, *"The Microsoft Repository"*, Proc. 23th. VLDB Conf., 1997, pp. 3-12.

[3] D. Bitton, D. DeWitt, and C. Turbyfill: *"Benchmarking database systems: A systematic approach"*, Proc. VLDB, 1983.

[4] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gehrke, D. N. Shah: *"The Bucky Object-Relational Benchmark"*, Proc. 1996 VLDB, pp. 135-146.

[5] M. J. Carey, D. Doole, N. Mattos: *"O-O, What Have They Done to DB2?"*, Proc. 1999 25th. VLDB Conf., pp. 542-553.

[6] M. J. Carey, D. J. DeWitt, C. Kant, J. F. Naughton: *A status report on the OO7 OODBMS benchmarking effort*, Proc. ACM OOPSLA, Portland, OR, USA, 1994, pp. 414-426.

[7] R. G. G. Cattell, D. Barry, D. Bartels, et al: "The *Object Database Standard:ODMG 2.0"*, Morgan-Kaufman Publishers, San Mateo, 1997.

[8] J. Gray: *"The Benchmark Handbook for Database and Transaction Processing Systems"*, Morgan Kaufmann Publishers, San Mateo, CA, USA, 2nd Ed., 1993.

[9] P. Gulutzan, T. Pelzer: *"SQL-99 Complete, Really"*, R&D Publications, 1999.

[10] P. Hamid, B. Mitschang, N. Südkamp, B. G. Lindsay, "*Composite-object views in relational DBMS: an implementation perspective*", Inf. Sys. 19(1), 1994, pp. 69-88.

[11] Informix Corp., http://www.informix.com

[12] A. Keller, R. Jensen, S. Agrawal: *"Persistence Software: Bridging Object-Orieted Programming and Relational Database"*, Proc. 1993 ACM SIGMOD, pp. 523-528.

[13] W. Mahnke, N. Ritter, H.-P. Steiert: *"Towards Generating Object-Relational Software Engineering Repositories"*, Proc. BTW'99, Freiburg, Germany, 1999.

[14] W. Mahnke, H.-P. Steiert: The Application Potential of ORDBMS in Design Environments, Proc. CAD 2000, Berlin Germany, 2000 (in German).

[15] B. Mitschang, P. Hamid, P. Pistor, B. G. Lindsay, N. Südkamp, *"SQL/XNF - Processing Composite Objects as Abstractions over Relational Data"*, Proc. 1993 Int'l Conf. On Data Eng., pp. 272-282.

[16] Ontos Business Data Server, http://www.ontos.com.

[17] Oracle Corp.,http://www.oracle.com.

[18] Poet Object Server,POET Software, POET SQL Object Factory, http://poet.com/.

[19] RogueWave Software, DBTools.h++, http://www.roguewav3e.com/products/dbtools/.

[20] N. Ritter, H.-P. Steiert, W. Mahnke, R. Feldmann: *"An Object-Relational SE-Repository with Generated Services"*, Proc. IRMA'99, IDEA Group Publ., May 1999, pp. 986-990.

[21] M. Stonebraker, P. Brown, D. Moore: *"Object-relational DBMSs - The Next Wave"*, Second Edition, Morgan Kaufmann 1998.

[22] SQL99: *ANSI/ISO/IEC 9075-1-1999 Database Launguages SQL Part 1 Framework.*

[23] SQL99: *ANSI/ISO/IEC 9075-2-1999 Database Launguages SQL Part 2 Foundation.*

[24] TPC: Transaction Processing Performance Council, Standard Specification 1.0, May 1995, http://www.tpc.org.

[25] UML, Rational Software Corp. *Unified Modeling Language*, http://www.rational.com/.

[26] W. P. Zhang: *"Evaluation of the First Generation ORDBMSs by Using Bucky Benchmark"*, Internal Report, University of Kaiserslautern, 1998.

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE