A Comparative Study of the Features and Performance of ORM Tools in a .NET Environment

Stevica Cvetković and Dragan Janković

Faculty of Electronic Engineering, Aleksandra Medvedeva 14,
18000 Niš, Serbia
{stevica.cvetkovic,dragan.jankovic}@elfak.ni.ac.rs

Abstract. Object Relational Mapping (ORM) tools are increasingly becoming important in the process of information systems development, but still their level of use is lower than expected, considering all the benefits they offer. In this paper, we have presented comparative analysis of the two most used ORM tools in .NET programming environment. The features, usage and performance of Microsoft Entity Framework and NHibernate were analyzed and compared from a software development point of view. Various query mechanisms were described and tested against conventional SQL query approach as a benchmark. The results of our experiments have shown that the widely accepted opinion that ORM introduces translation overhead to all persistence operations is not correct in the case of modern ORM tools in .NET environment. Therefore, at the end of this paper we have discussed some reasons for insufficiently widespread application of ORM technology.

Keywords: Object-relational mapping (ORM), persistence, performance evaluation.

1 Introduction

Modern information systems rely heavily on relational databases, mostly because of their reliability and standardized query language. Design of these kinds of systems is based on layered software architecture which implies three logical layers: presentation, business and data layer. Implementation of the data layer is essential step in the process of information system development and could take up to 40% of complete development time [5]. It is usually developed in the object oriented environment, where objects are used to represent data that needs to be persisted in a relational database. Traditional data layer development approach has shown numerous disadvantages that could be summarized in the following:

- The "object-relational impedance mismatch" problem occurs because the object-oriented and relational database paradigms are based on different principles [2]. The impedance mismatch is manifested in several specific differences: inheritance implementation, association implementation, data types, etc.

A. Dearle and R.V. Zicari (Eds.): ICOODB 2010, LNCS 6348, pp. 147–158, 2010.

- Development of advanced data layer mechanisms, such as concurrency control, caching or navigating relationships could be very complex for implementation.
- Frequent database structure changes could cause application errors due to mismatch of application domain objects with latest database structure. Those errors can't be captured at compile-time and therefore require development of complex test mechanisms.
- Developed information systems become significantly dependent on concrete database managing system (DBMS), and porting of such a system to another DBMS is complex and tedious task.

Object Relational Mapping (ORM) tools have been developed in an attempt to overcome the above listed problems. As described in the agile methodology for software and databases development [2], these tools automatically create data layer and provide a mapping between the application object model and the database relational model. They act as an intermediary between an object oriented code and a relational database. The most successful examples of ORM tools are Hibernate [3], Oracle TopLink [9] and recently introduced Microsoft Entity Framework (EF) [1], [4]. With these tools, application developer is encouraged to think in terms of data layer objects (persistent objects) and their relationships. The system takes over all the details of handling objects and relationships at runtime. It automatically tracks updates made to the objects and performs the necessary SQL insert, update, and delete statements at commit time. This way, business layer development can be done in the comfort of object-oriented languages which dramatically reduces the overall complexity and increases code understandability in applications.

However, application of ORM tools requires a very deep understanding of the concrete object model, the relational model and the DBMS used. All software design and implementation decisions should be done with this fact kept firmly in mind, since a large part of the transferring of persistent objects to the database is performed by the underlying DBMS. Different DBMS-s can implement common database functionalities (e.g. integrity control) in specific way, causing a part of the software errors to be produced by the underlying DBMS. This is the reason why the ORM generated data layers are not fully database independent.

The aim of this paper is to investigate the usage and performances of the most commonly used ORM tools in a .NET environment – NHibernate and EF. Beside a comparison of the tools from a software development point of view, we concentrate heavily on the performance analysis in order to check the common opinion that ORM tools introduce a significant slowdown compared to conventional data layer approach. In the literature there is already a comparison between different Java based ORM tools and object databases [6], [10], [11]. However, to the best of our knowledge, no scientific studies have been made comparing the different .NET ORM tools such as EF and NHibernate.

In the rest of the paper ORM specific concepts are analysed for both tools. After that, different query mechanisms are described, including query samples. Finally, performance test results are presented and discussed.

2 ORM Tools: Entity Framework vs. NHibernate

EF and NHibernate are compared from a software development point of view. Summary overview of compared features is given in Table 1. More detailed comparison is given below.

Table 1. Comparison of features between Entity Framework and NHibernate

Feature	Entity Framework	NHibernate
Programming languages	.NET (C#, VB)	.NET (C#, VB), Java
Generation of XML mapping file	Automatic	Only with 3rd party tools
Bi-directional relationships	Yes	Yes
Transactions handling	Automatic	Automatic
Locking mechanisms	Optimistic and pessimistic	Optimistic and pessimistic
Optimization	Lazy loading, caching	Lazy loading, caching
Query methods	LINQ, Entity SQL, SQL	HQL, Criteria API, SQL
Supported DBMS	MS SQL Server, MySQL, commercial for other DBMSs	MS SQL Server, Oracle, MySQL, PostgreSQl,...

2.1 Mapping Mechanism

In the context of ORM tools, Ambler [2] provides the following definition of mapping: "Mapping: The act of determining how objects and their relationships are persisted in permanent data storage, in this case a relational database". Mapping mechanism, that establishes a relationship between the persistent objects and the data stored in the database, is the backbone of an ORM tool (Figure 1). There are a number of technical challenges that have to be addressed by any mapping solution. It is relatively straightforward task to build an ORM that uses one-to-one mapping to expose each row in a database table as an object. However, when dealing with the relationships, inheritance mappings, multi-DBMS-vendor support and performance issues, mapping mechanism could become extremely complex.

Both EF and NHibernate provide a comprehensive mapping mechanism, and have GUI tools to increase developer productivity. While EF includes a collection of design-time tools for automatic mapping schema generation (integrated in Microsoft Visual Studio IDE), NHibernate mapping can be generated only by 3rd party tools. Commonly used approach is describing mapping details in XML files. The syntax and structure of mapping files are specified using a declarative language that has well-defined semantics and covers a wide range of mapping scenarios. Both tools support bidirectional relationships when the objects on both ends of the relationship contain references of each other.

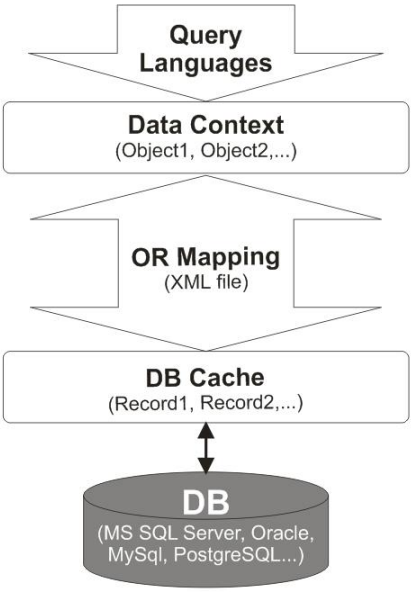

Fig. 1. Simplified general architecture of ORM Tools

Compared to NHibernate, EF introduces more general mapping mechanism represented with three separate XML sections incorporated in one file [1]. The bottom-level section of mapping file uses Store Schema Definition Language (SSDL) to describe the data source (tables, columns, constraints, etc.) that persist data for applications. Conceptual Schema Definition Language (CSDL) is used to declare and define the entities and associations of the object model being designed. The programmable data layer classes are built from this schema. The third section is written in Mapping Specification Language (MSL) which connects (maps) the declarations in the CSDL section to the data source described in the SSDL section of mapping file.

2.2 The Data Context

EF and NHibernate both provide a private cache of database persistent objects for the application execution in one thread. EF *ObjectContext* and the NHibernate *Session* are terms used for this common concept that we will call *Data Context*. The Data Context is a core which handles the loading and saving of persistent objects, where database updates are deferred until synchronization between the object cache and database is needed. It is also responsible for the management of database connections, transactions, concurrent access, etc.

In the typical scenario, when a new persistent object is created, it has no connection to the Data Context and must be explicitly introduced. In-memory changes to the object are then tracked until synchronization with database is explicitly requested. When the object is accessed again, the Data Context provides the needed data out of the cache, or if it is not found, the database is accessed. In both tools, persistent objects data obtained during the Data Context lifetime, is still there after the Data

Context is disposed. In the typical three-layer architecture, the persistent objects are filled out in the Business Layer and returned for display in the Presentation Layer, after the Data Context is disposed.

All EF persistent objects are inherited from one system super class, while in NHibernate there is no such a class. This EF approach could raise a question about code reusability because of .NET languages limitation that class can inherit only one base class. However, the problem is overcame with "partial classes" concept so that all the ORM tool provided code can be in one file and the application extended code in another.

2.3 Transactions and Concurrency

When the *Data Context* actually interacts with the database (insert or update operations), the database will open a transaction if none is currently open, and commit it after the statement. This process is known as "auto-commit". To extend a transaction lifetime to contain multiple database accesses, both ORM tools provide a transaction interface with methods for enclosing a database transaction. Actual propagation of any of the database actions is deferred until transaction commit is called. This triggers a synchronization of the *Data Context* with the database.

In order to maintain transaction isolation, databases rely on locking, which prevents concurrent access to particular data structures. Both tools provide optimistic and pessimistic concurrency models and always use the locking mechanism provided by DBMS (never lock persistent objects in memory). Optimistic locking strategy assumes that conflicting updates will cause an application exception that should be properly handled. Concurrency model could be defined for separate table columns, by setting XML mapping file, or directly in source code.

2.4 Query Approaches

Although both tools are designed to work with multiple query languages, including native SQL, this study will be focused on the two most widely used approaches – SQL Derivatives and Language Queries.

2.4.1 SQL Derivatives

SQL derivatives represent extensions of standard SQL that allows query definition on persistent objects instead of tables. They extend standard SQL in the following ways:

- Introduce native support for persistent objects within SQL queries (member accesses, relationship navigation, etc.)
- Add support for aggregation functions against objects (min, max, sum, average, etc.)
- Query results are strongly-typed .NET objects and not rows or columns.

EF EntitySQL and Hibernate Query Language (HQL) are derivatives of SQL, designed to support previously described mechanisms. Both tools use dot notation when referring child objects. While the query result of NHibernate query is a List object, in EF it is a special *ObjectQuery* object which contains methods to get List of objects.

2.4.2 Language Queries

One of the main disadvantage of SQL derivative queries, common to all string represented queries, is a software compiler inability to detect errors during compilation. Therefore, compiler cannot help the developer with compile-time checking of syntactic and semantic correctness, like it does for the rest of the program. In order to overcome the problems, both tools introduce approaches that we will identify as *Language Queries*. EF introduces LINQ [7], while NHibernate provides Criteria API for the purpose.

EF LINQ is an innovation in the programming languages that introduces query-related constructs to .NET programming languages. The query constructs are not processed by an external tool. They are rather expressions of the languages themselves. LINQ introduces nine new operators into .NET programming languages: from, join, join...into, let, where, orderby, group...by, select, and into. In addition, queries formulated using LINQ can run against various data sources such as in-memory data structures (Lists), XML documents and databases.

NHibernate offers Criteria API that uses actual .NET classes and methods to set restrictions for the query. Unlike LINQ, it doesn't introduce new operators into programming language; it only uses a new API for retrieving entities. Queries are constructed by composing API class method calls on corresponding persistent objects. This form of queries is usually called *method-based* queries. EF also provides this kind of query formulation, but it is not described in more details in this text, since LINQ has been proven as more robust method for query representation.

3 Performance Testing

In order to test EF and NHibernate performances and answer the question of whether ORM tools significantly harm overall performance, we will measure query execution speed against conventional SqlClient approach as a benchmark. Seven typical SQL test queries are defined, together with the corresponding HQL, EntitySQL and LINQ representations (Table 2). First five are SELECT queries chosen to cover various reading scenarios (joins, grouping, subqueries...), while the last two are INSERT and DELETE tests. For the first five queries, each query was executed 100 times, and the average execution time was calculated. For the last two, we calculated execution time for INSERT and DELETE of 100 records. All of the queries involve iterating through the returned results and sending back a number of objects found.

3.1 Testing Environment

Tests were executed on a PC with hardware consisted of an AMD Dual Core 2.51GHz processor, 2 GB of DDR2 RAM, 500GB hard disk (7300 rpm). Machine runs MS Windows XP Pro (SP2), with MS SQL Server 2005 installed with default settings and parameters. All tests were performed on a single machine in a computer laboratory. The software configuration that we used to run tests includes:

- MS Visual Studio 2010 (C#)
- .NET Framework 4.0 including Entity Framework
- NHibernate 2.0.1

- MS SQL Server 2005 – Standard Edition
- AdventureWorks database which is part of the SQL Server 2005 Sample Databases [8]. List of all database tables used for testing, including approximate number of records is as follows: Sales.Customer ≈ 20000, Sales.SalesOrderHeader ≈ 31500, Sales.SalesTerritory ≈ 10, Sales.SalesOrderDetail ≈ 121000.

3.2 Data Context Initialization

It is important to emphasize that in both cases, the initialization of Data Context is "expensive" operation due to one-time costs of establishing database connection and generating execution plan. Approaches for the initialization are different in EF and NHibernate. While NHibernate contains specific class methods for this purpose, EF implicitly initializes Data Context the first time an application executes a query. Explicit method call is a more flexible solution which allows a developer to implement Data Context initialization during the application initialization. It is much more acceptable solution from user's point of view, than to wait for certain period of time on first query execution when application is already started. EF can overcome lack of explicit initialization by calling some "unnecessary" query during application start just in order to initialize Data Context. The costs associated with the first-time initialization in both tools were around 3 seconds and were disregarded from the results presented in Table 2.

3.3 Test Results Analysis

A comparative evaluation of query performances between SqlClient, EF, and NHibernate approaches are given in Table 2. Additionally, graphical illustration is presented in Figure 2. Expectedly, SqlClient shows the best test results. However, query performances of the two described ORM frameworks were not significantly slower than SqlClient in the most of test cases.

The EF appears to perform better than NHibernate, except in case of Q4_Select_GroupBy. The slow performance of EF for this query type can be explained with the non-optimal way in which group-by statements are translated into native SQL form. In fact, they will not be translated into SQL group-by statement. The EF group-by query, either EntitySQL or LINQ, is a hierarchical query that returns a sequence of groups, where each group contains the key and all the elements (records) that made up the group. It means that Entity Framework group-by is translated into simple SQL select query that fetches all the records which are further processed in the memory to get group-by results.

Performances of SQL set-based updating statement were intentionally omitted in previous discussion because they need careful consideration. From ORM tool point of view, this kind of statement cannot be expressed in its original form. Instead, ORM update query must be simulated in three successive steps including querying for the objects to be updated followed by in-memory properties update and finally submit of changes to the database. It introduces obvious performance overhead which is not dramatic in case of updating a small number of records. However, when dealing with applications that require large sets of records to be updated at once ("bulk updates"), ORM performance is hundred times worse. This is probably the weakest and the most

Table 2. A comparative evaluation of query performances using SqlClient, HQL (NHibernate), EntitySQL (Entity Framework) and LINQ (Entity Framework)

Id	SQL query string	Execution Time (ms)			
		SqlClient	HQL	Entity SQL	LINQ
Q1_Select_Simple	SELECT AccountNumber FROM Sales.Customer AS c WHERE c.ModifiedDate > '2001-12-06' AND c.CustomerType = 'S' AND c.AccountNumber LIKE 'AW000002%'	102.3	109.6	108.1	149.8
Q2_Select_Join	SELECT soh.AccountNumber FROM Sales.SalesOrderHeader soh JOIN Sales.Customer c ON soh.CustomerID = c.CustomerID JOIN Sales.SalesTerritory t ON c.TeritoryID = t.TerritoryID WHERE t.Name = 'Australia' ORDER BY soh.AccountNumber	55.3	65.2	55.7	61.7
Q3_Select_Subquery	SELECT c. AccountNumber FROM Sales.Customer c WHERE c.TerritoryID = (SELECT c1.TerritoryID FROM Sales.Customer c1 WHERE c1.AccountNumber = 'AW00000021')	21.7	22.3	15.4	37.9
Q4_Select_GroupBy	SELECT t.Name FROM Sales.Customer c JOIN Sales.SalesTerritory t ON c.TerritoryID = t.TerritoryID GROUP BY t.Name ORDER BY t.Name	0.7	0.9	21.3	22.5
Q5_Select_Top	SELECT TOP 1000 sod.CarrierTrackingNumber FROM Sales.SalesOrderDetail sod WHERE sod.UnitPrice > 123 ORDER BY sod.SalesOrderID	2.1	5.5	5.9	6.1
Q6_Insert	INSERT INTO Sales.Customer(TerritoryID, CustomerType, rowguid, ModifiedDate) VALUES (1, 'S', NEWID(), GETDATE())	48.9	63.5	62.5	62.5
Q7_Delete	DELETE FROM Sales.Customer WHERE ModifiedDate > @x	19.9	30.2	31.2	31.2

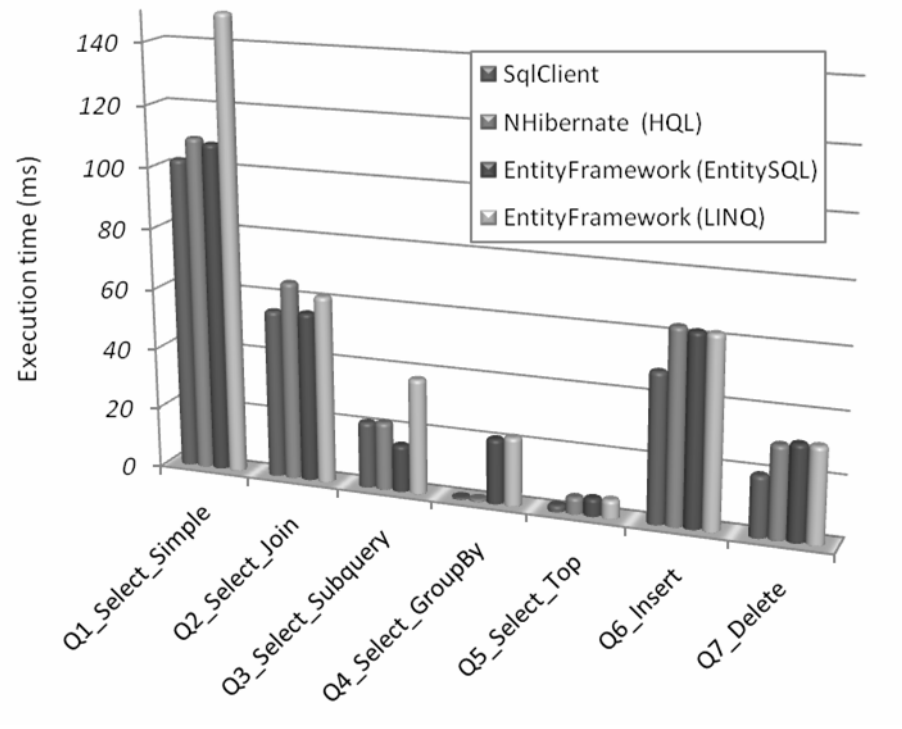

Fig. 2. Graphical representation of query performances comparison between SqlClient, HQL, EntitySQL and LINQ

criticized part of ORM concept. However, in practical applications bulk updates are not so frequent operations. In such cases there is always possibility for "hybrid" solution by explicit use of SQL updating statements via SqlClient approach.

EF offers few mechanisms for query performance improvement: use of compiled queries, defining the smart connection strings, disable change tracking, etc. According to documentation as well as some technical articles, highest LINQ performance improvement could be achieved using so-called compiled queries. It is due to obvious drawback that every time LINQ query is executed, it needs to be converted to SQL statement. The query is first checked for syntax errors by LINQ query engine and then translated into SQL statement. As a solution, compiled LINQ queries provide mechanism to cache the query plan in a static class so it can be executed much faster. Tests that were conducted on compiled and non-compiled LINQ queries have shown that the time saved with compiled queries is almost constant (in our case around 5ms) and doesn't depend on query complexity. Therefore, its relative impact on complex queries execution performance (those that last over 100ms) is insignificant. Note that all the results in Table 2 are measured for compiled LINQ queries. Also, when analyzing results, it should be taken into account that ORM tools assume additional costs of the first-time *Data Context* initialization, which were not included in presented results because they are performed only once per application.

The overall results have shown that widely held opinion that ORM is significantly slower than the conventional data layer approach for all persistent operations, is not correct in the case of modern ORM tools. Although there are some specific cases, like bulk updates or EF group-by statements, where ORM extremely degrades performances, for most of the typical data access scenarios, performances are comparable. Early ORM systems were actually slower than custom data layer solutions, because they were introducing overheads of reading metadata, reflecting on classes, generating queries, etc. In a custom data layer solution, these tasks would be completed at design time by the developer and would not affect system performance. However, modern-day ORM systems like EF and NHibernate are based on advanced architecture. They apply a variety of performance improvement mechanisms like caching, lazy loading or dirty checking that manage to compensate the overhead that ORM introduce by nature.

4 Barriers to the Adoption of ORM

Although our comparison showed that performances of modern ORM tools are close to the conventional data layer approach for the most of the typical data access scenarios, there is still skepticism toward massive adoption of ORM for information system development. In this section, we present a number of technical and cultural barriers that are responsible for insufficiently widespread application of ORM.

4.1 Learning Time Period and Costs

From the perspective of a software development company, period of time needed to learn general ORM concepts as well as specific solution details, could present serious obstacle to its adoption. Setting up and installing ORM software on all the development, testing and production machines can be a substantial undertaking. Also, new query definition approaches (like LINQ, HQL, etc.) require software developers to learn a new API that may be quite complex. Regardless of which ORM solution is chosen for adoption, it is likely that the company will have to invest in the training of software developers. However, the long-term benefits afforded by ORM can definitely outweigh these one-time personnel and technological costs.

4.2 Legacy Database Adoption

Very often, new systems need to use data from a legacy database. In this case, the software developers don't have any impact on the architecture of the database; they have to work with what is available. By the rule, legacy databases have a questionable structure reflected in poor normalization, incomplete definition of constraints, etc. With a complex OR mapping definition mechanism, it may be difficult or even impossible to represent the mapping to the legacy database. Redefinitions of OR mapping file require developers to think carefully about details that had previously been largely hidden from them. Additional problem could represent inadequate support for old versions of DBMS.

4.3 Distrust of ORM

In addition to the previously described objective barriers, there may also be bureaucratic obstacles to the adoption of new technologies, including ORM, from the ranks of project managers. Unfamiliarity with ORM concepts could represent a significant problem. Many project managers as well as software developers are not familiar with the scope of application and exact features of ORM tools. Usually, developers have outdated notions of the capabilities of ORM, believing it to be slow and incapable of supporting sophisticated database operations. Therefore, improving awareness of the features that modern ORM tools provide is an important step to their more widespread adoption.

5 Conclusion

This paper reports on the first findings of our investigation into the usage and performance of ORM technologies in .NET environment. It was found that overall performances of conventional SqlClient approach are comparable to ORM tools for the most of the typical data access scenarios. It is in contrast to the popular opinion that ORM tools add translation overhead to all persistence operations and hence are proportionally slower. ORM tools allow developers to use more powerful object oriented modeling techniques, to benefit from ease of development and provide unified programming model for different data sources. On the other side, their effective use will largely depend on the skill set possessed by project team members.

In the future, our plans are directed towards comparing ORM tools using modified versions of some standard SQL benchmarks. In addition, issues at more of an architectural level should also be investigated, for instance a comparison between distributed and a multi-user benchmark.

Acknowledgments. Work on this paper was supported in part by the Ministry of Science and Technological Development of the Republic of Serbia (Project number TR13015).

References

1. Adya, A., Blakeley, J., Melnik, S., Muralidhar, S.: Anatomy of the ADO.NET Entity Framework. In: ACM SIGMOD International Conference on Management of Data, Beijing, China, pp. 877–888 (2007)
2. Ambler, S.: Agile Database Techniques. Wiley, Chichester (2003)
3. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications (2006)
4. Castro, P., Melnik, S., Adya, A.: ADO.NET entity framework: raising the level of abstraction in data programming. In: ACM SIGMOD International Conference on Management of Data, Beijing, China, pp. 1070–1072 (2007)
5. Keene, C.: Data Services for Next-Generation SOAs. SOA World Magazine (2004), `http://soa.sys-con.com/node/47283`
6. Kopteff, M.: The Usage and Performance of Object Databases compared with ORM tools in a Java environment. In: 1st International Conference on Objects and Databases (ICOODB 2008), Berlin, Germany (2008), `http://soa.sys-con.com/node/47283`

7. Meijer, E., Beckman, B., Bierman, G.M.: LINQ: Reconciling Objects, Relations and XML in the.NET Framework. In: ACM SIGMOD International Conference on Management of Data, Chicago, IL, USA, pp. 706–706 (2006)
8. Microsoft Download Center SQL Server, Samples and Sample Databases (2005), http://www.microsoft.com/downloads/details.aspx?familyid=e719ecf7-9f46-4312-af89-6ad8702e4e6e
9. OracleTopLink, http://www.oracle.com/technology/products/ias/toplink
10. Van Zyl, P., Kourie, D.G., Boake, A.: Comparing the performance of object databases and ORM tools. In: Bishop, J., Kourier, D. (eds.) Annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (SAICSIT 2006), Somerset West, South Africa, pp. 1–11 (2006)
11. Zhang, W., Ritter, N.: The Real Benefits of Object-Relational DB-Technology for Object-Oriented Software Development. In: 18th British National Conference on Databases: Advances in Databases, Chilton, UK, pp. 89–104 (2001)