Implementing Optimistic Concurrency Control for Persistence Middleware using Row Version Verification

Martti Laiho
Dpt. of Business Information Technology
Haaga-Helia University of Applied Sciences
FI-00520 Helsinki, Finland
Email: martti.laiho@haaga-helia.fi

Fritz Laux
Fakultät Informatik
Reutlingen University
D-72762 Reutlingen, Germany
Email: fritz.laux@reutlingen-university.de

***Abstract*—Modern web-based applications are often built as multi-tier architecture using persistence middleware. Middleware technology providers recommend the use of Optimistic Concurrency Control (OCC) mechanism to avoid the risk of blocked resources. However, most vendors of relational database management systems implement only locking schemes for concurrency control. As consequence a kind of OCC has to be implemented at client or middleware side.**

A simple Row Version Verification (RVV) mechanism has been proposed to implement an OCC at client side. For performance reasons the middleware uses buffers (cache) of its own to avoid network traffic and possible disk I/O. This caching however complicates the use of RVV because the data in the middleware cache may be stale (outdated). We investigate various data access technologies, including the new Java Persistence API (JPA) and Microsoft's LINQ technologies for their ability to use the RVV programming discipline.

The use of persistence middleware that tries to relieve the programmer from the low level transaction programming turns out to even complicate the situation in some cases. Programmed examples show how to use SQL data access patterns to solve the problem.

***Keywords*-persistence middleware; caching; data access pattern; row version verification.**

I. Introduction

With the advent of multi-tier web applications or more precisely with decentralized and loosely coupled transactional systems OCC has gained new attention. Providers of enterprise architecture frameworks (like Java Enterprise Edition (JEE)) and persistence middleware (like object relational mappers) propose to use optimistic concurrency control to avoid the risk of blocked resources and orphan locks. Developers face now the situation that they have to implement a kind of optimistic concurrency control over the existing concurrency control provided by the DBMS. In order to help application developers we propose to use RVV. Lets explain the mechanism shortly: RVV depends on a technical row version column. Its value is incremented whenever any data in the row is changed. By checking the row version, it can be found out if any concurrent transaction has modified the data meanwhile. If this happened, the validation fails and the transaction has to abort. If the row version is still the same, then the transaction may write the new data. If the row version is cached by the middleware this could lead to stale data. Therefore, it is necessary to circumvent the middleware cache for the row version in order to apply RVV.

Another motivation to use RVV results from the practice that web applications usually split a user transaction into several SQL transactions. In this case the database concurrency mechanism cannot ensure transactional properties for the user transaction, but RVV helps to avoid at least the lost update problem. Consider a typical concurrency scenario with two users that try to book the same flight online. First, a list of fights is displayed (step 2, in Figure 1), second, a certain flight is chosen (step 4), and third, the flight is booked (step 6). When the second user selects the same flight and reads the available seats before the first user has updated the number, a *lost update* will be produced. This could be avoided by re-reading the seats in step 6 and compare it with step 4 before storing the new number.

We consider the RVV discipline as a critical reliability criterion in web based transactional application development. The proper way for applications to meet the RVV discipline is to strictly follow the SQL data access patterns presented in Laux and Laiho [2]. This patterns essentially ensure that the row version is checked before overwriting a data value. The patterns describe how to deal with different concurrency schemes of the underlying DBMS and how to exploit triggers to support RVV from the database side.

In the present paper these data access patterns are applied to a generic use case and code samples show implementations for various technologies.

A. Structure of the Paper

After this Introduction, Section II starts with the presentation of a typical use case including SQL statements for its setup on a relational database. Each of the following Sections, III (JDBC, .NET), IV (Hibernate, JPA), and V (MS LINQ) present implementations of RVV using the data access patterns of [2]. Section VI concludes the paper with a comparison between these technologies.

B. Related Work

Optimistic concurrency control (OCC) mechanisms have been studied already 30 years ago ([3][4][5][6]), but hardly any commercial DBMS has implemented algorithms of this type (see Bernstein and Newcomer [7] or Gray and Reuter [8]) except for MVCC.

In case of MVCC the middleware has to make sure, that caching is not invalidating the multi-versioning system. This problem is discussed by Seifert and Scholl [9] who counteract with a hybrid of invalidation and propagation message. In Web applications the risk of not properly terminated transactions is extremely high (users simply *click away*). In such cases snapshot data or locks in the case of a locking protocol are held until (hopefully) some timeout elapses. This can severely degrade performance.

In order to avoid the above problems the application has to implement an OCC on top of the database's locking mechanism. Nock ([10], pages 395-404) describes an "optimistic lock" pattern based on version information. He points out that this access pattern does not use locking (despite of its misleading name) and therefore avoids orphan locks. His pattern does not address caching.

Adya et al [11] recommend to use the system clock as blocking-free protocol for global serialization in distributed database systems. However this approach has to fail if the resolution is not fine enough to produce unique timestamps as we proofed for Oracle in [1].

During decades of development in relational database technologies the programming paradigms in data access technologies have constantly changed. The two mainstream schools in object oriented programming today are the Java camp [12] and the camp of Microsoft's .NET framework [13], both providing call-level APIs and Object-Relational Mappings (ORM) of their own for object persistence. The problems of using RVV with the older version of Enterprise Java Beans 2.0 are discussed in [15].

We follow in this paper the object persistence abstractions of Hoffer, Prescott, and Topi ([14], Chapter 16) and implement the access patterns at application or middleware layer. At SQL level we apply patterns B and C from [2] to different technologies. *Pattern B* uses a conditional UPDATE .. WHERE *oldRV = RV* statement to verify that the old row version *oldRV* is still the same as the actual one (*RV*). *Pattern C* consists of a SELECT *RV* .. statement to re-read the actual row version, then a programmed validation (if *oldRV = RV* then ..) follows, and if the validation was successful, the UPDATE takes place. The code samples in the present paper use SQL Server 2008 as DBMS for our basic use case, since SQL Server provides some interesting tuning possibilities and can be used by all technologies we want to present. Examples using DB2 and Oracle can be found in our full RVV paper [1].

II. A Basic Use Case Sample

Figure 1 presents a typical user transaction in 6 steps (phases) containing three SQL transactions like the flight booking mentioned before plus an optional compensation step. The ideal isolation levels listed for each SQL transaction depends on the concurrency control provided by the DBMS. The default concurrency control mechanism on SQL Server is locking (LSCC), but it can alternatively be configured to use "snapshot" isolation (MVCC).

For the SQL Server implementation of our use case the following Transact-SQL commands will be needed:

```
CREATE TABLE rvv.VersionTest(
id INT NOT NULL PRIMARY KEY (id),
s VARCHAR(20), -- a sample data column
rv ROWVERSION -- incr by SQLServer at row-update
) ;
GO
CREATE VIEW rvv.RvTestList (id,s)) -- for phase 2
AS SELECT id,s FROM rvv.VersionTest ;
GO
CREATE VIEW rvv.RvTest (id,s,rv) --for phases 4 and 6
AS SELECT id,s,CAST(rv AS BIGINT) AS rv
FROM rvv.VersionTest WITH (UPDLOCK) ;
GO
INSERT INTO rvv.RvTest (id,s) VALUES (1,'some text');
INSERT INTO rvv.RvTest (id,s) VALUES (2,'some text');
```

For technical details of the above script the reader is referred to the SQL Server online documentation [16].

III. Baseline RVV Implementations using Call-Level API

The first open database call-level interface and de facto standard for accessing almost any DBMS is the ODBC API specification which has strongly influenced the data access technologies since 1992. The current specification is almost the same as the SQL/CLI standard of ISO SQL. Many class libraries have been implemented as wrappers of ODBC and many data access tools can be configured to access databases using ODBC. Based very much on the same ideas Sun has introduced the JDBC interface for Java applications accessing databases. This has become industry standard in the Java world. Using the previously defined SQL views for accessing table VersionTest and applying the RVV discipline, the following sample Java code for Phase 6 (the update phase) of Figure 1 reveals the necessary technical details:

```
 // *** Phase 6 - UPDATE (Transaction) ***
con.setAutoCommit(false);
// Pattern B update - no need to set isolation level
string sqlCommand = "UPDATE rvv.RvTest " +
"SET s = ? " +
"WHERE id = ? AND rv = ? ";
```

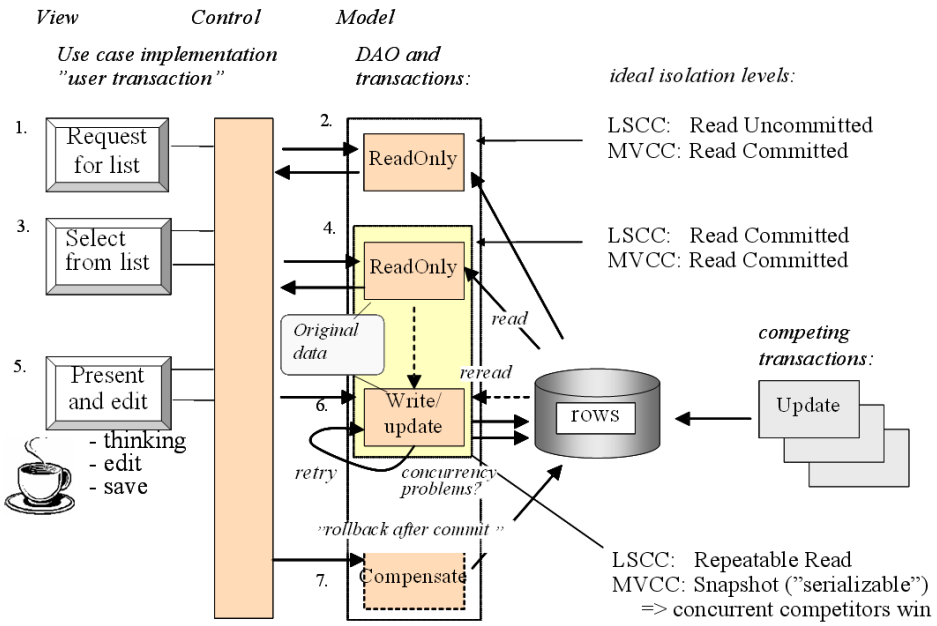

Figure 1. Transactions and isolation levels of the sample use case

```
pstmt = con.prepareStatement(sqlCommand);
pstmt.setString(1, newS);
pstmt.setLong(2, id);
pstmt.setLong(3, oldRv);
int updated = pstmt.executeUpdate();
if (updated != 1) {
throw new Exception("Conflicting row version in the
database! " );
}
pstmt.close();
// Update succeeded -> The application needs to know
the new value of RV
sqlCommand = "SELECT rv FROM rvv.RvTest WHERE id =
?";
pstmt = con.prepareStatement(sqlCommand);
pstmt.setLong(1, id);
ResultSet rs = pstmt.executeQuery();
newRv = rs.getLong(1);
rs.close();
pstmt.close();
con.commit();
```

In the above, as in all following examples, it is assumed that the version attribute *rv* will be maintained by the database itself, e.g., by a row level trigger. If the database has no such capability the application itself has to take care of incrementing the version on every update.

Every 4-5 years Microsoft has introduced a new data access technology after ODBC, and in the beginning of this millennium ADO.NET built on various data providers. Without going into details of this rich technology we just show below the phase 6 from our baseline implementation of RVV using C# language and the native .NET Data Provider (SqlClient) [17] to access SQL Server 2008:

```
// Phase 6 - update transaction
txn = cn.BeginTransaction();
cmd.Transaction = txn;
// Pattern B update including reread of rv using
OUTPUT clause of T-SQL:
cmd.CommandText = "UPDATE rvv.RvTest " +
"SET s = @s OUTPUT INSERTED.rv " +
"WHERE id = @id AND rv = @oldRv ";
cmd.Parameters.Clear();
cmd.Parameters.Add("@s", SqlDbType.Char, 20).Value =
newS;
cmd.Parameters.Add("@id", SqlDbType.Int, 5).Value =
id;
cmd.Parameters.Add("@oldRv", SqlDbType.BigInt,
12).Value = oldRv;
long newRv = 0L;
try { newRv = (long) cmd.ExecuteScalar();
txn.Commit();
}
catch (Exception e) {
throw new Exception("Conflicting row version in
database "+ e.Message);
}
```

All the latest versions of the mainstream DBMS systems can act as servers of Web Services providing light and direct call-level API to DBMS services. One RVV example of

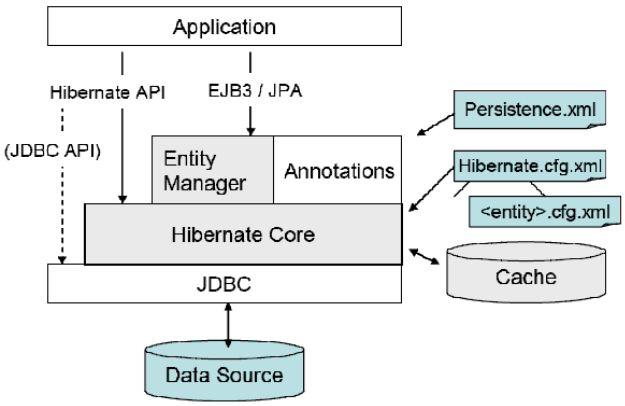

Figure 2. Hibernate architecture

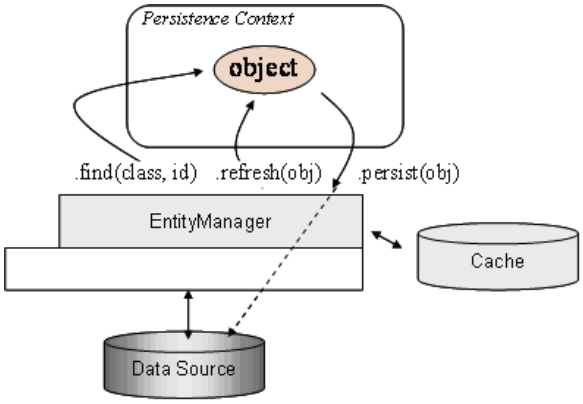

Figure 3. JPA persistence management

.NET using Web Services by Soap 1.2 envelopes is given in [1]. While it will be interesting to study the Web Services of DB2 and Oracle, the news of the future version of SQL Server tell that the native XML Web Services of SQL Server will be replaced by Windows Communication Foundation (WCF) [18].

IV. RVV Implementations using ORM Middleware

Data access patterns solving the impedance mismatch between relational databases and object-oriented programming are called Object-Relational Mappers (ORM) [10]. One widely known ORM technology is the Container Managed Persistence (CMP) pattern of the Java Persistence API (JPA) as part of the Java Enterprise Beans 3.0 (EJB3). The JPA specification assumes the use of ”optimistic locking” [19].

The JPA raised the market for sophisticated persistence managers providing object-relational mappings, such as TopLink [20] and Hibernate [21]. Figure 2 shows our interpretation of the alternatives of the current Hibernate framework which implements the JPA but also allows full control over Hibernate's Core down to JDBC code level, which we actually need for our RVV implementation when using Hibernate.

In terms of RVV we are mainly interested in the object persistence services of ORM frameworks. As an example of these services Figure 3 presents methods of JPA for managing persistence of an entity object.

Object properties can be changed only for loaded objects. This means that only *Pattern C* (re-SELECT..UPDATE) updates are possible in ORM frameworks. The caching service of ORM middleware improves performance by buffering objects, but RVV requires the current row version from the database and therefore needs to bypass the cache. ORM frameworks provide automatic ”optimistic locking” services based on timestamp or version column, but according to the JPA specification these are maintained by the ORM middleware itself (persistence provider) at client-side, so any other software can bypass the version maintenance. Therefore, a server side versioning should be preferred.

The following Java code sample from our RVV Paper [1] shows how to avoid stale data from Hibernate's cache. To set the isolation level via JDBC we first need to switch to Hibernate's core level. Then, the object is reloaded and the actual *newRv* is read. The used *Pattern C* requires *repeatable read* or stronger isolation level to ensure that the row version will not change during validation and execution of the update. If the validation succeeds the object is updated:

```
// Phase 6 - "model"
em.clear(); //1) clear EntityManager's cache for RVV
try {
   Session session = (Session)em.getDelegate();
   // JPA => Hibernate Core
   Connection conn = session.connection(); // =>
   JDBC
   Transaction tx6 = session.beginTransaction();
   conn.setTransactionIsolation(
   conn.TRANSACTION_SERIALIZABLE); // Pattern C
   RvvEntity re2 = em.find(RvvEntity.class, id);
   // reload the object
   Long newRv = (Long)re2.getRv(); // read current
   row version
   if (oldRv.equals(newRv)) { // verifying the
   version
       re2.setS(s); // update of properties
       em.persist(re2); // Pattern C RVV update
       tx6.commit();
       } else
       throw new Exception("StaleObjectState:
       oldRv=" + oldRv + " newRv=" + newRv);
   }
   catch (Exception ex) {
   System.out.println("P 6, catched exception: " +
   ex);

}
```

V. RVV Implementation using LINQ to SQL

Microsoft's answer to the ORM frameworks is Language Integrated Query (LINQ) for the .NET Framework 3.5. The class libraries of LINQ can be integrated as part of any .NET language providing developer ”IntelliSense” support during coding time and error checking already at compile-time [22]. So called ”standard query operators” of LINQ can be applied to different mappings using LINQ providers, such as LINQ

to XML, LINQ to Datasets, LINQ to Objects, and LINQ to SQL. In the following C# code sample of our use case Phase 6 the object *myRow* was loaded from the database in Phase 4 and string *newS* contains a new value entered in Phase 5, and *dc* is the DataContext object which holds the open connection to the database. The shaded part of the code is just a programmed break allowing concurrent processing for concurrency tests:

```
// Phase 6
TransactionOptions txOpt = new TransactionOptions();
txOpt.IsolationLevel =
System.Transactions.IsolationLevel.RepeatableRead;
using (TransactionScope txs = new TransactionScope
(TransactionScopeOption.Required, txOpt)) {
try { myRow.S = newS;
 // To allow time for concurrent update tests ...
 Console.Write("Press ENTER to continue ..");
 Console.ReadLine();
dc.SubmitChanges(ConflictMode.FailOnFirstConflict );
txs.Complete();
}
catch (ChangeConflictException e) {
   Console.WriteLine("ChangeConflictException: " +
   e.Message);
   }
catch (Exception e) {
   Console.WriteLine("SubmitChanges error: " +
   e.Message + ", Source: " + e.Source +
   ", InnerException: " + e.InnerException);
   }

}
```

At run-time the data access statements of LINQ to SQL are translated into native SQL which can be traced. The following sample test run trace proves that row version verification is automatic based on *Pattern B* (Conditional UPDATE) and LINQ automatically tries to refresh the updated row version content:

```
Press ENTER to continue ..
Before pressing the ENTER key the contents of column S in row 1 is updated
in a concurrent Transact-SQL session.
UPDATE [rvv].[RvTestU]
SET [S] = @p3
WHERE ([ID] = @p0) AND ([S] = @p1) AND ([RV] = @p2)

SELECT [t1].[RV]
FROM [rvv].[RvTestU] AS [t1]
WHERE ((@@ROWCOUNT) > 0) AND ([t1].[ID] = @p4)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p1: Input NVarChar (Size = 9; Prec = 0; Scale =
0) [TestValue]
-- @p2: Input BigInt (Size = 0; Prec = 0; Scale = 0)
[32001]
-- @p3: Input NVarChar (Size = 7; Prec = 0; Scale =
0) [testing]
-- @p4: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model:
AttributedMetaModel Build: 3.5.21022.8
ChangeConflictException: Row not found or changed.
```

The code shows the use of TransactionScope, the new transaction programming paradigm of .NET Framework which does not depend on LINQ. Setting Isolation level is actually not necessary for the transaction since it uses *Pattern B* (Conditional UPDATE), but we want to show that it can be set programmatically. The test also shows that no stale data was used in spite of LINQ caching.

VI. Conclusion

Table I presents a comparison of the Call-level APIs and ORM Frameworks with RVV practice in mind. Major findings are the differences when applying the access patterns of [2] for different middleware technologies with regard to isolation levels, transaction control, caching, and performance overhead. While we are writing this paper LINQ to SQL is still at beta phase and currently it turned out slow in our tests. However, we are impressed about the built-in ”optimistic concurrency control” as Microsoft calls it. Microsoft has the advantage of experiences from the competing technologies. Attributes of LINQ are more orthogonal than the numerous JPA annotations and its object caching did not produce side-effects in concurrency control making LINQ easier to use and manage. It also utilizes server-side version stamping. With the advanced features of the framework - as it proves to do things right - this is a most promising software development extension in the .NET Framework. The native DBMS for LINQ is currently SQL Server, but since IBM and Oracle have already shipped its own ADO.NET data providers, their support for this technology can be expected.

As an advantage of ORM Frameworks Hoffer et al [14] lists ”Developers do not need detailed understanding of the DBMS or the database schema”. We don’t share this view. Even if we use these higher abstraction frameworks we need to verify that we understand the low level data access operations so that our applications are transaction safe. For example it was necessary to circumvent middleware caches where possible or when using disconnected data sets we had to explicitly reread the row version from the database in repeatable read isolation (access *Pattern C*). The version stamping of the ”optimistic locking” should not be handled at client or middleware side, but on server side to avoid ignorant applications.

Some comparisons in Table I are still speculative instead of hard facts. In this respect Table I can be considered as suggestions for further studies.

References

[1] M. Laiho and F. Laux; *On Row Version Verifying (RVV) Data Access Discipline for avoiding Lost Updates*, URL: http://www.DBTechNet.org/papers/RVV_Paper_20090605.pdf (2009, Nov.)

Table I
COMPARISON OF CLI APIS AND ORM FRAMEWORKS

	CLI APIs		**ORM Frameworks**	
	ODBC JDBC ADO.NET	Web Service APIs	Java Hibernate JDO TopLink JPA	.NET LINQ
Access Pattern A	yes	yes	no	no
Access Pattern B	yes	yes	no	yes
Access Pattern C	yes	yes	yes	no
Performance overhead	low	DBMS http: low appl.serv: high	high	high (beta)
OOP Programming	labor-intensive		yes	yes
Persistence	SQL	SQL	middleware service	middleware service
Use of native SQL	detailed	detailed	limited	limited
– isolation	full control	full control	default	full control
– local transaction	full control	full control	TM 1)	full control
– global transaction	(ADO.NET)	difficult	TM 1)	implicit 2)
2^{nd} level caching			yes	
Optimistic Locking	RVV	RVV	configurable	built-in
Version stamping (default)			client-side	server-side

1) using Transaction Manager (TM), 2) using TransactionScope

[2] F. Laux and M. Laiho; *SQL Access Patterns for Optimistic Concurrency Control*, The First International Conferences on Pervasive Patterns and Applications (PATTERNS 2009), November 15-20, 2009 - Athens/Glyfada, Greece

[3] H. T. Kung and J. T. Robinson; *On Optimistic Methods for Concurrency Control*, In ACM Transactions on Database Systems (TODS) 6(2), 1981, pp. 213-226

[4] D. Z. Badal; *Correctness of Concurrency Control and Implications in Distributed Databases*. In Proc. COMPSAC79, Chicago, 1979

[5] G. Schlageter; *Optimistic Methods for Concurrency Control in Distributed Database Systems*. In Proceedings of the 7^{th} VLDB, Cannes, 1981, pp.125-130

[6] R. Unland, U. Prädel, and G. Schlageter; *Ideas on Optimistic Concurrency Control I: On the Integration of Timestamps into Optimistic Concurrency Control Methods and A New Solution for the Starvation Problem in Optimistic Concurrency Control*. In: Informatik Fachbericht; FernUniversität Hagen, Nr. 26. 1982

[7] Ph. Bernstein and E. Newcomer; *Principles of Transaction Processing*, Morgan Kaufmann, 1997

[8] J. Gray and A. Reuter; *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993

[9] A. Seifert and M. H. Scholl; *A Multi-version Cache Replacement and Prefetching Policy for Hybrid Data Delivery Environments*, Proceedings of the 28^{th} VLDB Conference, Hong Kong, China, 2002

[10] C. Nock; *Data Access Patterns*, Addison-Wesley, 2004

[11] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari; *Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks*, ACM SIGMOD Record, Volume 24 , Issue 2 (May 1995), pp 23 - 34, ISSN: 0163-5808

[12] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase; *The Java EE 5 Tutorial*, URL: http://java.sun.com/javaee/5/docs/tutorial/doc/sjsaseej2eet.html (2008, Oct.)

[13] N.N.; *.NET Framework 3.5*, msdn .NET Framework Developer Center, URL: http://msdn.microsoft.com/en-us/library/w0x726c2.aspx (2009. Nov.)

[14] J. A. Hoffer, M. B. Prescott, and H. Topi; *Modern Database Management*, 9th ed., Pearson Prentice-Hall, 2009

[15] M. Laiho and F. Laux; *Data Access using RVV Discipline and Persistence Middleware*, eRA-3, 2008, Aegina/Greece

[16] N.N.; *SQL Server Books Online*, msdn SQL Server Developer Center, URL: http://msdn. microsoft.com/en-gb/library/ms130214.aspx (2009, Nov.)

[17] N.N.; *SQL Server and ADO.NET*, msdn Visual Studio Developer Center, URL: http://msdn.microsoft.com/en-us/library/kb9s9ks0.aspx (2009, Oct.)

[18] N.N.; *Native XML Web Services: Deprecated in SQL Server 2008*, msdn SQL Server Developer Center, URL: http://msdn.microsoft.com/en-us/library/cc280436.aspx (2009, Nov.)

[19] L. DeMichiel and M. Keith; *JSR 220: Enterprise JavaBeans™, Version 3.0*, Java Persistence API , Final Release, 8 May 2006, URL: http://jcp.org/aboutJava/communityprocess/final/jsr220/ (2009, Nov.)

[20] Oracle; *TopLink Developers Guide 10g (10.1.3.1.0)*, B28218-01, September 2006

[21] C. Bauer and G. King; *Java Persistence with Hibernate*, Manning, 2007

[22] S. Klein; *Professional LINQ*, Wiley Publishing, 2008