On Row Version Verifying (RVV) Data Access Discipline for avoiding Blind Overwriting of Data

"RVV Paper" version 2

Martti LAIHO [a], and Fritz LAUX [b]
[a] Haaga-Helia University of Applied Sciences, Finland
[b] Reutlingen University, Germany

Abstract

Transaction processing literature is usually focusing on SQL transactions (local in a database connection) and distributed transactions consisting of a set of local transactions building a logical atomic transaction, concurrency problems of these. This paper extends the discussion on transaction concurrency to user transactions built of a sequence of inter-related SQL transactions and to the programming discipline called "Optimistic Locking" The discipline was invented decades ago, before the era of RDBMS systems, but seems to have been forgotten in some modern complicated software architectures. We consider the name "Optimistic Locking" as misleading, since to be precise the question is on Row Version Verifying (RVV) Data Access Discipline in case of single row / object updates.

In Part I of this article present the concept of user transaction for a generic use case, and various data access programming patterns. We compare the classic Lost Update Problem of concurrent transactions with Blind Overwriting Problem of user transactions and discuss the technologies for avoiding it. We will focus on the client-side programming discipline with examples on using SQL for modern RDBMS systems, but the programming discipline is a valid requirement for accessing any DBMS systems including ODBMS systems.

Part II evaluates different approaches, services and programming technologies for row version control in the latest DB2, Oracle, and SQL Server versions on the server-side.

In the Appendices we apply the RVV discipline to the use case from Part I and apply it to various modern data access technologies, typically using persistent objects which turn out to be susceptible to the blind overwriting problem. We focus on the data access layer and show that instead of separate database transactions it is necessary to consider the varying isolation needs of sequence of inter-related transactions which together build the data access of a use case (ie. user transaction). Reliability comes before performance and we focus on reliability. The performance comparisons are not in the scope of this article, but will be an interesting subject area of further studies.

Keywords: database, transaction, concurrency, lost update, blind overwriting, optimistic locking, row version verification, persistent objects

CONTENTS

PART I: Lost Data Problems and Client-Side RVV Data Access Discipline 3
1.1 On the Lost Update and Blind Overwriting Problems 4
1.2 Scenarios of Data Access Patterns, a Use Case 7
1.2.1 Optimistic Lock Pattern 10
1.2.2 Pooled Connection Pattern 13
1.2.3 Retryer Pattern 14
1.3 Data Access in Modern Application Architectures 15
1.3.1 ADO.NET and Paradigm of Disconnected Data Processing 15
1.3.2 J2EE and EJB2 Entity Beans 17
1.3.3 TopLink and EJB3 19
1.3.4 Hibernate 23
1.3.5 LINQ to SQL 25
1.3.6 Web Services 26
1.3.7 PHP 27
1.3.8 Ruby 27
2.1 DB2 28
2.2 SQL Server 29
2.3 Oracle 31
Summary 33
References 35
Appendix 1 Testing the Myth of DBMS Timestamp Uniqueness 37
Appendix 2 Testing the Behaviour of the Mainstream DBMS Systems in a Simple SELECT-UPDATE Concurrency Case 38
Appendix 3 A Baseline Implementation of RVV in Java/JDBC 47
Appendix 4 Sample ADO.NET programs using RVV Discipline 53
Appendix 5 RVV Implementation using J2EE™ BMP 59
On RVV Implementation using J2EE™ CMP 68
Appendix 6 Programmed RVV for Hibernate Core 68
Appendix 7 Programmed RVV for Hibernate EntityManager (JPA) 74
Appendix 8 RVV and Microsoft LINQ to SQL 81
Appendix 9 RVV and Web Services 86
Appendix 10 RVV using PHP 94
Appendix 11 RVV using Ruby 103
RVV implementation using OCI8 104
RVV implementation using Ruby DBI 106
Index 109

PART I: Lost Data Problems and Client-Side RVV Data Access Discipline

-

Databases provide applications with reliable services for storing and retrieving data, but to preserve the reliability of the data, these services need to be used by well-formed transactions of applications. The ideal of these transactions is the ACID model, or actually ACiD as we have redefined the concept in our previous paper on "SQL Concurrency Technologies" (Concurrency Paper) in terms of the real concurrency control mechanisms of the mainstream DBMS systems used in business information systems today. The acronym comes from initials of the four transaction properties: Atomicity, Consistency, Isolation, and Durability, where the isolation is taken care by the concurrency control services of the DBMS. For trade-off between full ACID isolation and transaction throughput (affecting performance) some relaxed isolation levels have been defined in the SQL standard, in terms of the concurrency anomalies, and implemented in DBMS systems.

The classical "Lost Update Problem", as described by Chris Date and almost every database textbooks, is a basic anomaly in which data written by the transaction will be overwritten by some concurrent transaction while the transaction itself has not yet ended. This is possible in file based applications, but concurrency control (CC) services, such as multi-granular locking schemes (LSCC), multi-versioning (MVCC), or optimistic concurrency control [1](OCC) of the modern DBMS systems will eliminate this anomaly. In spite of this, data in database can be lost by some badly formed transactions, typically by insensitive updating transactions in a sequence of SQL transactions of a user transaction, also called business transaction. Usually the "C" property of an ACID transaction is understood to mean that the transaction cannot violate any SQL constraints defined in the database, but a more general interpretation requires that also the application logic in the transaction is well-formed, so that it does not violate any business rules, which may not be controllable by the DBMS itself. An important application rule is that application must not execute blind overwriting of data, and thus losing data from the database. In this paper we will focus on this requirement and present programming discipline for avoiding such bad transaction behaviour.

---

[1] Optimistic concurrency control was presented in the article "On Optimistic Methods for Concurrency Control" of Kung and Robinson in ACM TODS 6/1981, which is referenced in C. J. Date's textbook as follows "The so-called optimistic methods are based on assumption that conflict (in the sense of two transactions requesting simultaneous access to the same object) is likely to be quite rare in practice. The methods operate by allowing transactions to run to completion completely unhindered, and then checking at COMMIT time to see whether a conflict did in fact occur. If it did, the offending transaction is simply started again from the beginning. No updates are ever written to the database prior to successful completion of COMMIT processing, so such restarts do not require any updates to be undone. ... Optimistic methods [...] have already been implemented in a number of commercial products, including in particular the 'Fast Path' version of IMS." (Date 1986). The only RDBMS implementation of OCC that we know is the Pyrrho DBMS developed at University of the West of Scotland (UWS), see http://www.pyrrhodb.com. The database literature uses sometimes the term "Optimistic Concurrency Control"

1.1 On the Lost Update and Blind Overwriting Problems

Updates made during a transaction in progress will be protected against overwriting by other concurrent transactions using locking, multi-versioning, optimistic concurrency control by the modern RDMBS systems. So during the transaction we don't have the Lost Update Problem on our own updates like we can have in legacy file-based applications, but we need to consider also series of database transactions in their application context of user transactions and use cases.

Let us consider the following scenario of transactions of two concurrent processes A and B updating balance of the same account as follows:

Listing 1 A concurrency scenario of a SELECT - UPDATE transaction

step	Process A	Accounts balance	Process B
1	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;		
2		1 000 €	
3	SELECT balance INTO :balance FROM Accounts WHERE acctId = :id;		
4			
5	newBalance = balance - 100;		UPDATE Accounts SET balance = balance - 200 WHERE acctId = :id;
6		800 €	COMMIT;
7	UPDATE Accounts SET balance = :newBalance WHERE acctId = :id;		
8	COMMIT;	900 €	

Clearly the withdrawal of 200 € made by transaction B will be overwritten by A, in other words the update made by B in step 5 will be lost in step 7 when transaction of A will overwrite the updated value by value 900 € which is based on **stale data** i.e. outdated value of the balance from step 3. If the transactions A and B would serialize properly, the correct balance value after these transactions would be 700 €, but there is nothing that the DBMS could do to protect the update of step 5, since guilty to this

---

on other server-side concurrency control services, such as implementations of cursor-level "optimistic" concurrency control (of SQL Server and DB2 LUW V9.5 and later) and on multi-versioning concurrency control (MVCC) services provided by some DBMS systems on transaction level, but also as a synonym of "Optimistic Locking" on the client-side programming discipline

Blind Overwriting Problem is the programmer of process A, who has ordered from the DBMS a wrong isolation level. READ COMMITTED, which for performance reasons is the default transaction isolation level used by most RDBMS systems, does not protect any data read by the transaction of getting outdated right after reading the value. The proper isolation level in this case should be REPEATABLE READ or SERIALIZABLE, which would protect the values read in the transaction of getting updated by others during the transaction, for example by holding shared locks on these rows up to end of the transaction - if it is possible. The isolation service of the DBMS does guarantee that the transaction will either get the ordered isolation or in case of concurrency conflict the transaction will be rejected by the DBMS. Means of this service and transactional outcome for the very same application code can be different on use of different DBMS systems, which we will show in Appendix 2. Usually a transaction rejected due to concurrency conflict should be retried by the application, but this is not always the case as we will show later.

The erroneous scenario above would have been the same if *user transaction* of A had committed its SQL transaction of steps 1 and 3 (called transaction A1) in step 4 and continued with another SQL transaction A2 of steps 7-8. In this case no isolation level could have helped, but A2 would have also made a **blind write** (based on stale data, insensitive of the current value) over the updated balance value.

The blind write of the update transaction A2 of steps 7-8 (losing the data written by transaction B) could have been avoided by any of the following types of practices:

Type 0: if A2 in step 7 had been using the **cumulative form** of the update (sensitive to the current value, no risk for blind writing) like B uses in step 5 as follows

```
UPDATE Accounts
SET balance = balance - 100
WHERE acctId = :id;
```

Type 1: by transaction A1 first reading the **original row version data** in step 3 and transaction A2 in step 7 verifying in a **single (atomic) comparison expression** that the **current row version** in the database is still the same as it was when the process previously accessed the account row and using now the following update form

```
UPDATE Accounts
SET balance = :newBalance
WHERE acctId = :id AND
      (rowVersion = :old_rowVersion);
```

The comparison expression can be a single comparison predicate, RVV predicate, where `rowVersion` is for example a column (or a pseudo column provided by the DBMS) reflecting any changes made in the contents of the row and the `:old_rowVersion` is a host variable containing the value of that column when the process previously read the contents of the row, or the comparison expression

can be built of version comparisons of all columns used, based on the 3-value logic of SQL.

Type 2: (re-SELECT .. UPDATE) as an other variant of Type 1 by transaction A2 first using a strong enough isolation level, at least REPEATABLE READ, before reading (or explicit row level locking like Oracle's "SELECT .. FOR UPDATE" on reading) the current rowVersion from the database so that the current rowVersion cannot be changed by others, and then using the conditional update

```
if (current_rowVersion = old_rowVersion) then
   UPDATE Accounts
   SET balance = :newBalance
   WHERE acctId = :id ;
```

Even in this case the RVV predicate should be included in the update command to support all CC technologies.

A general solution for row version management is to include a **technical row version column** `rv` in the table as follows

```
CREATE TABLE Accounts (
acctId  INTEGER  NOT NULL  PRIMARY KEY,
balance DECIMAL(11,2)  NOT NULL,
rv      INTEGER  DEFAULT 0);  -- row version
```

and using a row-level trigger to increase the value of column `rv` on any row automatically every time the row is updated. We provide examples of these triggers and product dependant alternatives in Part II.

Before modern RDBMS systems the application programming methods on verifying the record version were based on comparing

- contents of the whole record, or
- a checksum value (“finger print”) calculated of the contents of the whole record

and the method was called “Weak Locking” or “Optimistic Locking” (Nock 2004), although the method does not deal with locking at all.

1.2 Scenarios of Data Access Patterns, a Use Case

We will focus on the discipline requirements set for programming models in terms of row version control to avoid the blind overwritings. SQL transaction is the basic reliable data access pattern when building reliable applications, although in the traditional object-oriented Design Patterns arranged by Erich Gamma et al (GoF, 1994) the authors only briefly cover the issue in their Command pattern by commenting ”A transaction encapsulates a set of changes to data”.

A typical multi-tier architecture today makes use of the Model-View-Controller (MVC) model, where the Model part (M) is responsible on accessing the database (Data Access). Various Data Access design patterns have been introduced to implement the data access part of the model, for example Data Access Object (DAO).

Reliable data access requires transactionality and various transaction patterns can be used depending on the phases of the user transactions implementing use cases.

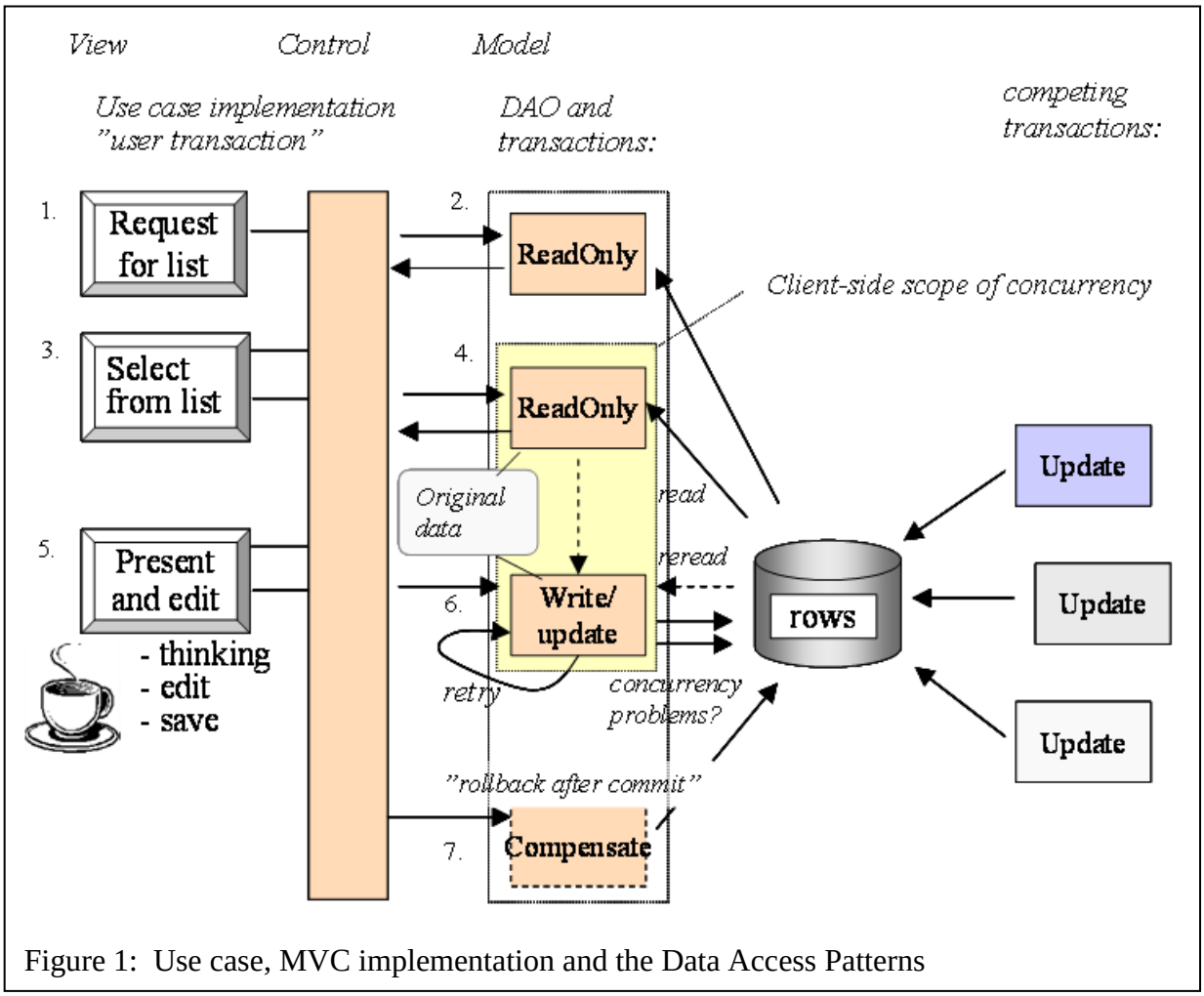


Figure 1: Use case, MVC implementation and the Data Access Patterns

Figure 1 presents by numbered steps the scenario of a typical user transaction of some data maintenance scenario (on View tier) where the user first picks the proper object from a search list (steps 1 and 3), the object data is then presented to the user on a form (step 5), after updating the data on the form the user presses some “save” button and the changed data will be updated in the database. We will now focus on the implementation of the scenario on the Model tier:

Step 2 is implemented as a READ ONLY transaction which reads some relevant attributes of objects using some selection criteria set by the user at step 1, and returns the result set to step 3 at View tier for selecting the proper object.

Step 4 is implemented as a READ ONLY transaction using “singleton SELECT” fetching all relevant attributes of the selected object to be presented to the user at step 5 on the View tier. Obviously this transaction is not allowed to read uncommitted data. For minimal blocking of concurrent transactions we don’t keep the object locked in the database, but step 4 starts the “client-side scope of concurrency” as seen by the application saving the original data (or what will be needed, for example just id and row version data) in a cache of the Model tier for the version verifying at step 6.

Step 5 in our scenario stands for the user interface steps on View tier, which may take unpredictable time to complete. This may also require additional READ ONLY lookup database transactions in the Model tier, but to keep the picture simple we have skipped presenting them.

Step 6 (later we will call this also as "Phase 6") is a typical case of an updating OLTP transaction. It will get updated data from the step 5 on View tier. Since some other concurrent transactions may have updated the data of the object in the database during user’s “thinking time” (and potential coffee break), and not to lose these updates by “blind overwriting”, the current data of the object has to be read from the database and to be compared to the original data in the cache. If the object version in the database has not changed, the update may proceed. However, if the object has changed, then our user application shall not be allowed to proceed updating the database, but the user shall be notified (on step 8 which is missing in the figure 1) of database containing more recent data and control shall return to step 3 (and in some case perhaps to step 1).

Whenever the object update transaction requires use of multiple SQL commands, which is the case in Type 2 update (SELECT .. UPDATE) or if the data of the object to be updated is actually stored in multiple tables, then it is possible that the transaction will fail due to concurrency conflict, for example deadlock, with some concurrent transaction.

Usually concurrency conflict can be solved by applying the **Retryer Transaction Pattern** (discussed later), but when we need to avoid blind overwriting of "meanwhile updates" of other transactions this may not be the proper solution.

At the end of successful step 6 we should **refresh the object version data** in the Model cache in case the user wants to continue updating the object data.

A committed transaction can not be rolled back, but database textbooks discuss of possible **compensating transactions**, which by reverse update statements will restore the object data into the original state of step 4 (for which we would need copy of the original data). This is presented as step 7 in Figure 1, which is not guaranteed to be a success, since concurrent transactions may have already affected the situation. Sometimes this may be possible apart from the database transactions and based on pure business rules: for example we may cancel the hotel / ticket reservation based on the reservation number of our own - a kind of business level locking.

Analysis of Concurrency Requirements and Options

At server-side the scope of concurrency is limited to a transaction at a time. We have covered the general theory of transaction concurrency, ISO SQL Standard, and implementations of concurrency control in the big three mainstream DBMS, DB2, Oracle and SQL Server in our preceding tutorial, "SQL Soncurrency Technologies". Here we assume that the reader is familiar with the isolation levels on tuning concurrency control of transactions and also on principles of database locks, but we shortly review these in the following.

Basically the concurrency control has been implemented either using some variant of the Locking Scheme of the early System R implementation (see Gray). We will call these systems as Multi-Granular Locking Scheme Concurrency Control (LSCC) systems (also called pessimistic concurrency control systems) in the following to make distinction with the Multi-Versioning Concurrency Control systems called as MVCC systems.

Examples of LSCC systems are DB2 and SQL Server, and in these systems the selected isolation level affects on shared locks (S-locks, read locks) blocking exclusive lock requests on various lock granularity levels. The isolation levels implemented in LSCC systems are principally in line with the ISO isolation levels except DB2 uses different names for these (see Appendix 2), whereas the MVCC systems have implemented only isolation levels which they call READ COMMITTED and SERIALIZABLE, but which have semantics based on database snapshots. In MVCC systems when a row is updated a new version of the row is written in the database and as long as the writing transaction has not committed all other transactions can only see some old version of the row, the latest in case of READ COMMITTED and in case of SERIALIZABLE the latest before the beginning of the reading transaction. A MVCC system never blocks readers, but as the price for that the readers may get stale data. Examples of MVCC systems are Oracle and SolidDB. Although Oracle also uses write locks (X-locks) and provides explicit row-locking by SELECT .. FOR UPDATE commands. Beside the LSCC system behaviour Microsoft has implemented also MVCC in SQL Server 2005 and the later versions when a database is set into ALLOW_SNAPSHOT_ISOLATION mode.

The results provided by MVCC and LSCC systems can be very different. In the following we will consider what this means for the transactions of our use case.

The transaction of step 2 is producing a list of keys and some characteristics of the target objects (rows) to support selection of the proper object by the user in step 3. The non-blocking READ COMMITTED isolation level of MVCC systems is ideal for this purpose. READ COMMITTED isolation level of LSCC may cause some blocking and since the user at this phase is interested only on existence of the objects the non-blocking READ UNCOMMITTED isolation level might be the best practice. If the characteristics of the selected object has been changed in step 5 the user can return to make a new selection.

The transaction of step 4 will fetch the object and this time the data need to be committed, so READ COMMITTED of both LSCC and MVCC systems is the best practice.

The transaction of step 6 is updating. If we can apply Type 1 updating using UPDATE statement with instant version verifying predicates, then the isolation level has no meaning and results of the services provided by LSCC and MVCC systems are the same. In this case we need to check if the UPDATE statement really affected the row.

For Type 2 (SELECT .. UPDATE) we need to set at least the isolation level REPEATABLE READ which in case of LSCC systems guarantees that the row version read by the SELECT command is protected by S-lock and unless we happen to become victim of accidental deadlock situation we will manage to do the UPDATE part. Here LSCC systems will provide the better service, since in case of MVCC systems we need to set the isolation level as SERIALIZABLE we will lose the competition to any concurrent updates.

The compensating transaction in step 7 is analogous with step 6 except that the row version to be updated shall be the version updated in step 6 and the new content of the object shall be the original content from the step 4.

1.2.1 Optimistic Lock Pattern

Regarding the popular Design Patterns as defined by GoF it is interesting to note that they hardly mention databases or the concept of database transaction, even if databases and transaction processing build the basis of all business critical applications. One of the most covering collection of design patterns for accessing databases has been presented by Clifton Nock in his book ”Data Access Patterns”. He calls the row version verifying design pattern as the Optimistic Lock pattern, which he defines as follows:

> ”Maintains version information to prevent missing database updates. Optimistic locking uses application semantics to validate a **working copy**’s version before updating the database.”

The version validation can be based on

a) timestamp of the update, which Nock considers unreliable
b) values of some subset of row columns.
c) incremental change identifiers of the row, the table or more general data set

Let's have a deeper look at these methods in the following:

a) Timestamp-based row version validation has been popular in database textbooks for some 20 years. It requires that every timestamp **measurement of time** by the DBMS has a unique value in the database instance so that no serial transactions can get the same value per same row to be updated. According to our test in Appendix 1 the SQL timestamp accuracy provided by the mainstream RDBMS systems on a typical 32bit Windows platform is just milliseconds while our simple uniqueness test of Oracle TIMESTAMP data type proves that even in a serial sequence of short transactions on single CPU system we may get tens of the very same timestamp value. Today processors are getting faster and faster, commercial servers may have some 256 processors, and a database instance can make use of 8 or more of those processors in parallel - so we can forget using SQL timestamps any more for row version validation.

b) Version validation based on values of row columns provides reliable method if done correctly. Version validation based on comparing value contents of whole data records in the old days was quite easy – just a single comparison. Comparison of SQL data is more complicated and we have more options. We may access only a subset of the columns. Comparison can also be focused to those columns values of which have changed. The 3-value logic of SQL due to possible NULL values gives interesting challenge to comparisons, which we will study below. Let us consider the following table as an example:

```
CREATE TABLE Table1 (
id      INT NOT NULL PRIMARY KEY,
s       VARCHAR(20),
r       REAL DEFAULT 0.0
) ;
INSERT INTO Table1 (id, s) VALUES (1,'Something');
INSERT INTO Table1 (id, r) VALUES (2, NULL);
INSERT INTO Table1 (id) VALUES (3);
```

To simplify our example let's assume that in step 4 (of our scenario in Figure 1) we have read the row 2 and the host language of Model Tier can manage NULL values (like for example PL/SQL) so that host variable '`oldS`' contains the NULL value of column s, as well as '`oldR`' contains the NULL value of column `r`. In step 6 we have got a new value for column `s` in the host variable '`newS`'. We can now update the database row 2 using a single UPDATE command as follows without first rereading the row

```
UPDATE Table1
SET s = :newS
WHERE id = 2
AND (s = :oldS OR s IS NULL AND :oldS IS NULL)
AND (r = :oldR OR r IS NULL AND :oldR IS NULL)
```

since a NULL value is not equal to any value.

If the non-NULL value of either s or r has been changed in the database after step 4, then the UPDATE command "succeeds", but will not actually find the row, in which case we need to sort this out, for example using an update count of a proper ODBC, JDBC, or ADO.NET API call.

c) The **incremental change identifier** of row level can be implemented as a technical column incremented either on every UPDATE command of the applications at client side(efficiently but unreliably), or using some row-level trigger at server side (reliably but costing some performance overhead) as we will show later in part II of this paper.

Let us consider the use of the following table where column `rv` (for row version) is the incremental change identifier of every row:

```
CREATE TABLE Table2 (
id      INT NOT NULL PRIMARY KEY,
s       VARCHAR(20),       -- representing data columns ..
r       REAL DEFAULT 0.0,
rv      BIGINT DEFAULT 0   -- incremented by UPDATE trigger
) ;
```

In case the column rv will be incremented automatically by a server side trigger on every UPDATE of any application, then the well-behaving UPDATE command in step 6 can be written as follows:

```
UPDATE Table2
SET  s = :newS
WHERE id = :keyValue AND rv = :oldRv
```

where the host variable :newS contains the new value for column s, :keyValue contains the value of the id column of the row to be updated, and :oldRv contains the original value of column rv in the working copy.

The following Java/JDBC example demonstrates how an application can detect the conflicting update made by some other transaction:

```
String sql = "UPDATE Table2 SET s=? WHERE id=? AND rv=?";
PreparedStatement pstmt = con.prepareSatement(sql);
pstmt.setString(1, newS);
pstmt.setInt(2, keyValue);
pstmt.setLong(3, oldRv);
int updateCount = pstmt.executeUpdate();
pstmt.close();
if (updateCount != 1)
   throw new Exception(
      "Sorry, your view on data is too old. " +
      "Please refresh from the database and try again.");
```

1.2.2 Pooled Connection Pattern

Opening a new database connection consumes quite a lot of resources. The Data Access Pattern called Resource Pool by Nock is more widely known as Pooled Connections and is in general use in modern JavaEE and .NET architectures. In these architectures the Container service of application components in the application server opens a set of database connections for every unique connection string value, and maintains these free connections in a Connection Pool which is managed by the Container. Whenever an application component instance opens a new database connection, the connection is actually picked up from the connection pool of the corresponding connection string, which operation consumes hardly any measurable amount of resources, - and when the application closes the connection it will actually be returned as open connection back to the connection pool for recycling. This has changed the way we do database accessing. Instead of trying to keep the database connection open as long as possible, we open a new database connection for every method run and we will close the connection at the end of the method. It is worth remembering that the DBMS is not aware of this recycling of connections and that the local transactions as seen by the DBMS depend on the connection. It also means that for every logical connection the physical connection activated from pool may be different, so it is not possible to keep locks from method to method.

This new data access pattern suits well for mobile applications accessing databases via application servers like JavaEE, etc. But it is leaving out the transaction manager of the DBMS, i.e. the application is not getting any help from the transaction manager to ensure serializability in general and avoid blind overwritings in particular. The

application itself has to re-implement any concurrency control. We have shown one possibility to avoid blind overwritings with the RVV.

1.2.3 Retryer Pattern

According to Clifton Nock the Retryer Pattern "Automatically retries operations whose failure is expected under certain defined conditions. This pattern enables fault-tolerance for data access operations."

Transaction is the basic data access pattern for reliable use of the reliable services of a DBMS. The atomicity property of ACID transactions emphasises the two outcomes of a single transaction run: succeeded or failed (in SQL terms: committed or rolled back). However, from the point of view of application architecture there are four outcomes:

1. succeeded,
2. rolled back by the application logic or user decision (i.e. using ROLLBACK so it is not worth retrying),
3. failed due to concurrency conflict, but may succeed on next try (worth retrying), and
4. failed due to broken connection, but may succeed using a new connection (for example to a stand-by DBMS in a cluster).

A typical reason for outcome 3 is a concurrency conflict with some concurrent transaction. Lock based concurrency control of the DBMS may have selected the transaction as the victim of deadlock and rolled it back automatically, or in case of Oracle rolled back the command which would have led to deadlock. On multi-versioning concurrency control the conflict is due to competing row version written by some concurrent transaction.

A surprisingly common misunderstanding is that DBMS would restart the failed transaction, which cannot be the case since DBMS is not aware of the transaction logic even it could remember the command history. Some application servers can provide restart service to methods of the data access components, but a fault-tolerant application should be capable to control the restarts also itself. Before every retry a short random pause of 0 to 1 seconds can help the conflicting transactions to get ready. Retryer logic shall also control the number of retries to avoid continuous loop of failures.

For a Phase 6 type of RVV transaction using non-cumulative updates and failed due to concurrency conflict (see Appendix 6 and 7), it is hardly useful to apply the Retryer Pattern when the winning transaction has already written the competing row version.

1.3 Data Access in Modern Application Architectures

Two major software architectures are nowadays evolving and already dominating in the ICT industry, namely .NET Architecture of Microsoft and J2EE (just renamed as JavaEE) of Sun Microsystems and the Java Camp of companies. Microsoft's .NET is considered as language independent but dependent on the Windows platform although the main programming language is C# (now an ISO standard and looking very much like Java) and there are also implementations on Linux platforms. J2EE is tied to Java language and therefore available on almost any Java compatible platforms. Data access of Java is typically based on JDBC or JDO, while data access in .NET is based on a new design called ADO.NET. In the following we will consider the applicability of the RVV Discipline in both of these data access technologies.

1.3.1 ADO.NET and Paradigm of Disconnected Data Processing

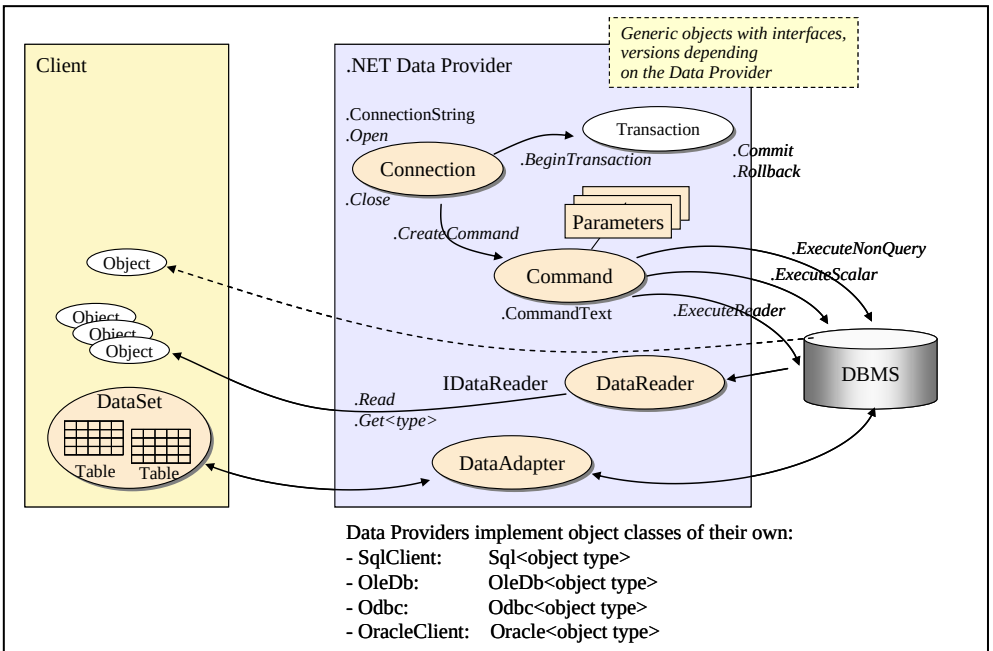


Figure 2a Object model of ADO.NET
(based on original drawing of David Chappell)

Figure 2a presents an operational, generic object model of ADO.NET in the .NET Framework 1. Compared to Microsoft ADO it is a new data access design close to JDBC, simplified and extended, but instead of a universal data access to all kinds of data sources it consists of a family of data models and data providers which can be generic like OleDb data provider or native providers of certain DBMSs like SqlClient of SQL Server. In Appendix 4.1 we demonstrate how the data access of our use case example can be implemented using RVV with the SqlClient provider of ADO.NET.

A major extension of ADO.NET is the provider independent client-side "in-memory database" `DataSet`, local cache of the client, which is connected to original data sources via DataAdapters and where data can be manipulated in "disconnected mode" i.e. without open connections to any databases. Data is typically first copied from a database (or databases) using a `DataAdapter` object and its `SelectCommand` object during a short database connection into a `DataSet`. The `DataSet` itself is disconnected from the database, and its contents can be changed locally while it also saves the original content copied from the database and its update history. Finally the updated data contents can be synchronized with the database contents in a new connection using `InsertCommand`, `UpdateCommand` and `DeleteCommand` objects of `DataAdapter` object row-by-row from the `DataSet` depending on the current status of each row. (For details see Johnson or Microsoft MSDN)

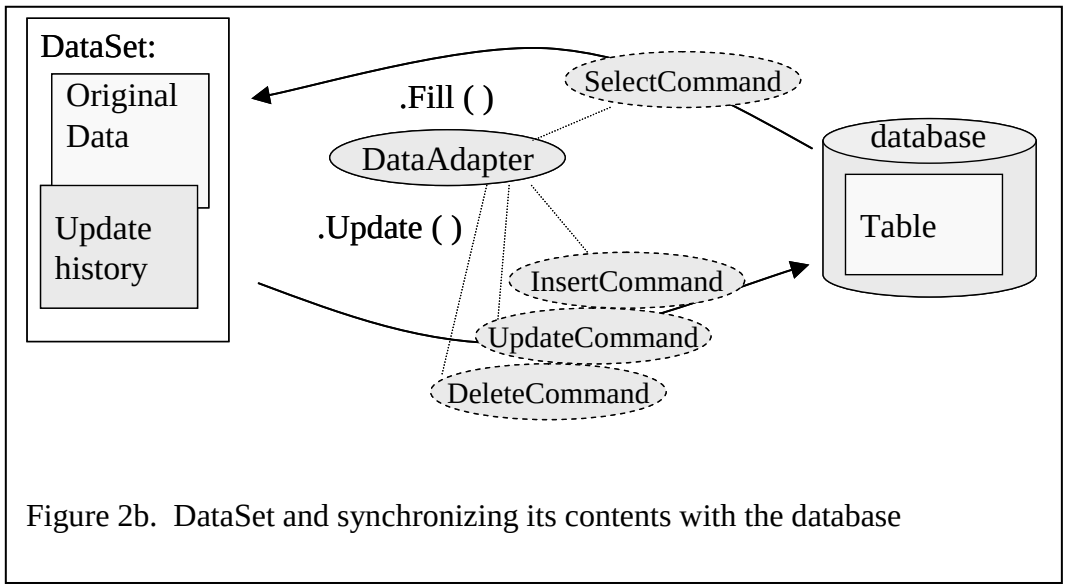


Figure 2b. DataSet and synchronizing its contents with the database

The disconnected mode suits fine for example for mobile applications. The data flow and ADO.NET objects used in the disconnected mode is presented in Figure 2b.

We demonstrate the disconnected programming paradigm in Appendix 4.2. To avoid losing the updates other clients may have made in the database during the disconnected phase, the synchronization operation needs to apply row version verification for each updated / deleted row in the `DataSet`. The original data contents in the `DataSet` can be used for row version validation and if the `Command` objects of the synchronization are generated automatically using `CommandBuilder` object, then the row version validation is based on comparing contents of all row columns. If the version validation of any row fails, then a `DBConcurrencyException` will be raised automatically by the `DataAdapter` object.

Following is a sample Profiler trace of `UpdateCommand` on synchronizing into the Table2 the update of row of `id` 1 which has been made earlier in the `DataSet` replacing the original value "`first`" by the value "`new value`" in column `s`

```
exec sp_executesql N'UPDATE [Table2] SET [s] = @p1 WHERE (([id] = @p2)
AND ((@p3 = 1 AND [s] IS NULL) OR ([s] = @p4)))', N'@p1 varchar(9),@p2
int,@p3 int,@p4 varchar(5)', @p1='new value',@p2=1,@p3=0,@p4='first'
```

Note that the system procedure `sp_executesql` above has following parameters: as first parameter the string containing the parameterized Transact-SQL statement, as second the string defining data types of the parameters, and finally value assignments for the parameters.

Even if the comparison is complicated as we've seen above, this paradigm provides **minimal "locking granularity"** and is not sensitive on updates to those row columns which are not included in the `DataSet`.

1.3.2 J2EE and EJB2 Entity Beans

The aim of Enterprise Java Bean (EJB) specification EJB2 of J2EE is to organize and simplify the development work of large multitiered distributed applications. The EJB components are designed to use all resources they may need only through interfaces under control of the EJB Container, the application server environment, which takes care of threading, distributed/local transactional services and security, so that the "bean developer can concentrate on solving business problems" (J2EE Tutorial 1.4).

Special persistent data objects called Entity Beans present the data in the database to the application following a very limited data access interface pattern. The persistence operations, i.e. refreshing the data contents from the database in the beginning of every transactional method and synchronizing the data with the database at the end of the method, can be programmed using JDBC by the developer, in which case the bean type is called Bean Managed Persistence (BMP) Bean, or the persistence operations can be taken automatically by the EJB container, in which case the bean type is called Container Managed Persistence (CMP) Bean. For CMP Beans it is advertised that the developer does not even need to know the DBMS system to be used for the database. The persistence manager of the EJB Container may also take care of the Object-Relational Mappings (ORM) while accessing the database. (J2EE Tutorial, and Roman)

While trying to make the application development simple following the Entity Bean patterns, the J2EE architecture is an example of over-sophisticated architecture where maybe no one of the designers really understood the whole big picture in all details.

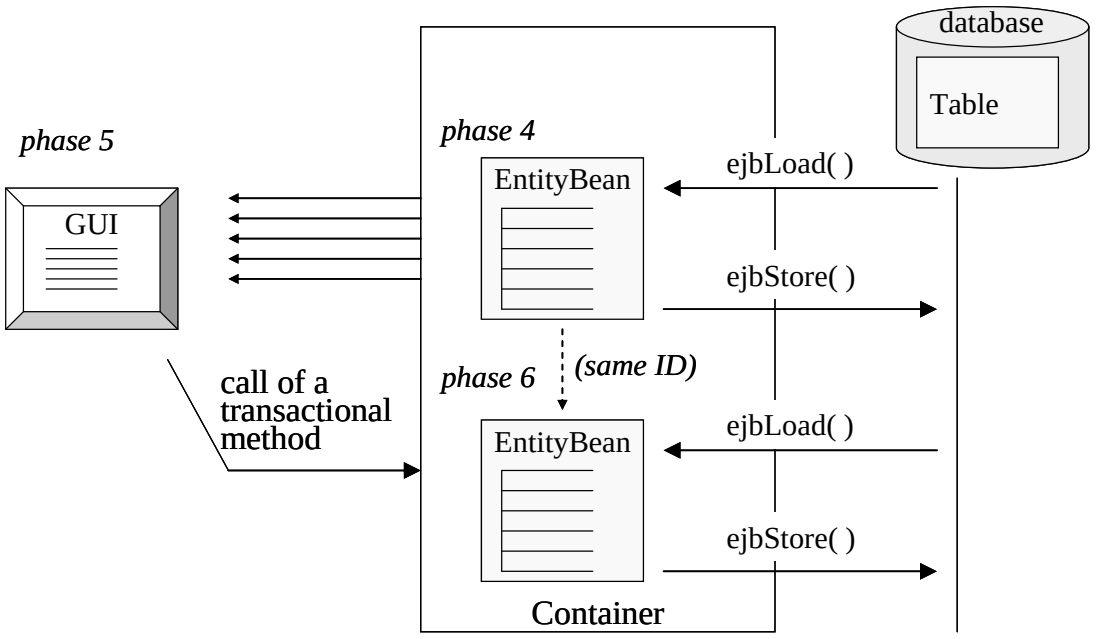


Figure 3. J2EE / EJB2 Synchronization of Entity Bean contents with the database

The big problem in the Entity Bean pattern is that according to the EJB specification the bean instance refreshes its content from the database at the beginning of every transactional method. So when we consider implementation of phase 6 in our use case (see Figure 3), the specification does not offer any solution to apply row version verification based on the original contents of phase 4 while storing the data back to the database (as pointed out by Marinescu). The "memory" of Entity Bean is based on the new contents fetched by ejbLoad in the beginning of every transaction only and the set methods use blind writing over these new values from the database!

The store operations may even write old contents read by ejbLoad over concurrent committed updates (blind overwriting problem even during the transaction) when using the Read Committed isolation level which is the default. Clearly the isolation level should be at least Repeatable Read!

In Appendix 5 we present a modified the BMP pattern which applies the RVV Discipline presenting extra rules providing means to store the original row version field value on the bean client-side. However, this is not possible to apply to CMP beans. For CMP beans the row version verification should be provided by the PersistentManager service of the Container.

Sun's J2EE Application Server has a specific solution of its own for consistency of CMP beans data as follows:
Check Modified at Commit - this will verify that row version in the database just before commit is still the same as it was in the beginning of the transaction (optimistic locking based on contents of all fetched columns)

Lock When Loaded - locks the row with a write lock when the row is first fetched (during the transaction?). This protects the row from concurrent updates during phase 6, - but not from concurrent updates during phase 5(?)
Version Consistency - this will store the row contents in a non-transactional cache when it is first fetched. On transaction commits and whenever the row is fetched as part of a query the row content of which has changed ("dirty instances") is compared with the row in the database based on the primary key and a technical version column maintained by the Container (i.e. using client-side stamping).

Sun has redesigned the architecture, now calling it as JavaEE and has replaced EJB2 by EJB3. The new EJB3 specification recommends the use of "Optimistic Locking", obviously meaning the services of the JPA middleware which we explain in the next chapter, but EJB2 is still supported without any words of warning (see DeMichiel & Keith). - So, who knows how many updates have been or will be lost due to these design errors. We have studied mainly the EJB2 implementation of Sun Application Server. The persistence agreement of EJB2 specification leaves room for different implementations, and some other products, for example IBM's WebSphere and BEA's WebLogic may have implemented some support of Optimistic Locking beyond the specification.

In the following we will look at 2 popular persistence manager products, TopLink and Hibernate, which can be used to fix the EJB2 Entity Bean problem on row version validation.

1.3.3 TopLink and EJB3

TopLink is a separate persistence manager product for J2EE application servers. It is a separate service layer (middleware) between the application components and the database. It provides data loading, data caching, Object-Relational Mapping (ORM), and data storing with row version validation services as presented in figure 4.

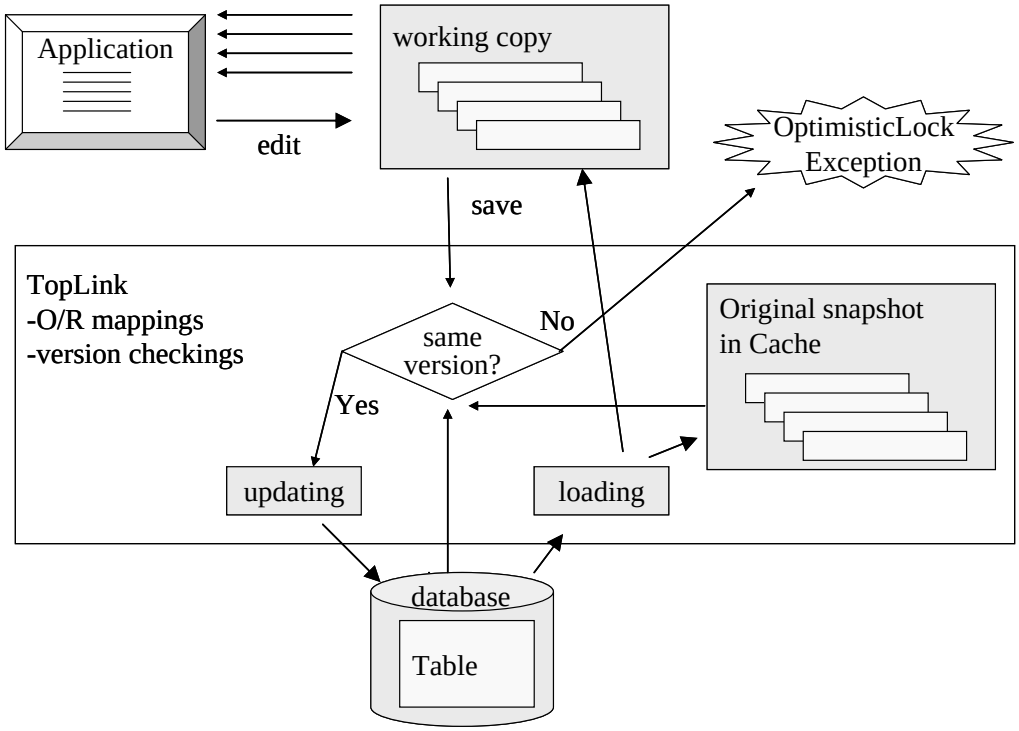


Figure 4. Version management using TopLink persistence services

According to TopLink's manual its *Optimistic Version Locking Policies* can be configured to the client-side version stamping of updated rows, using either a numeric version field or a timestamp field (an option they don't recommend), and always saving the original version fields in the cache for version validation. On *Optimistic Field Locking Policies* it stores the original column values in the cache and in case of

- *AllFieldsLockingPolicy* verifies the row version is based on all columns,
- *ChangedFieldsLockingPolicy* verifies the row version comparing only original values of those columns which have been changed by the application, and
- *SelectedFieldsLockingPolicy* verifies the row version based only on the selected columns (among of which could also be the version field of server-side stamping which we will consider in Part II).

Oracle has recently bought the product. There is a commercial version and now also a free version called TopLink Essentials, which is available on the otn.oracle.com Web pages.

JDeveloper

Oracle has integrated TopLink Essentials in the Oracle JDeveloper workbench as the persistence manager of the supported JavaEE technology, but also as so called TopLink POJO (Plain Old Java Objects) technology of its own. Beside these technologies Oracle has introduced the *Oracle Application Development Framework*

(ADF) based on extended MVC architecture and supported in JDeveloper IDE as shown in Figure 5 (adapted from Roy-Faderman and ADF Developers's Guide).

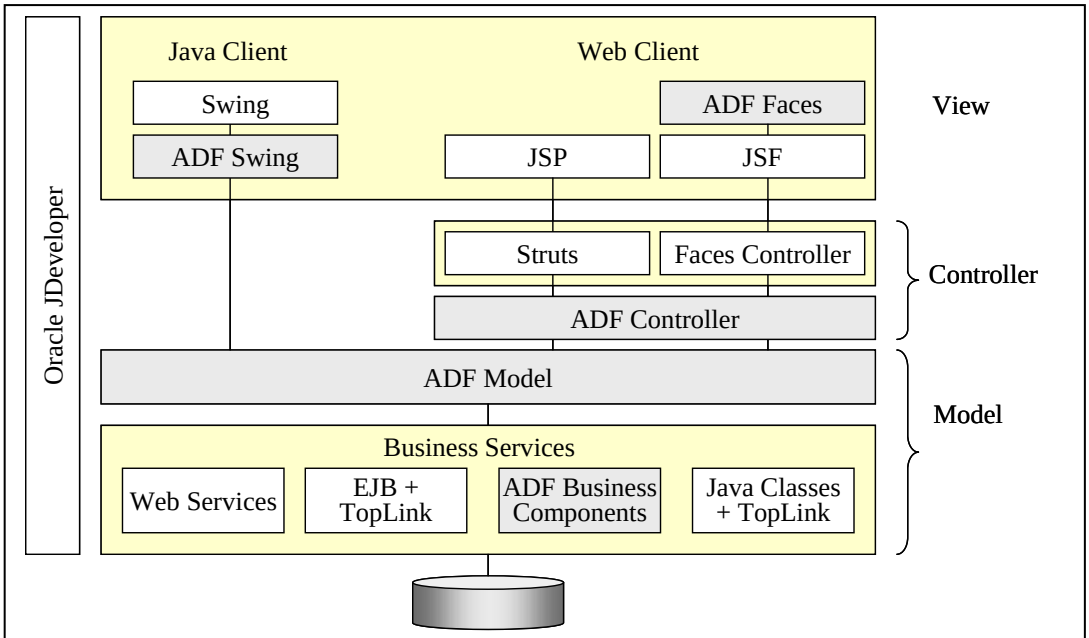
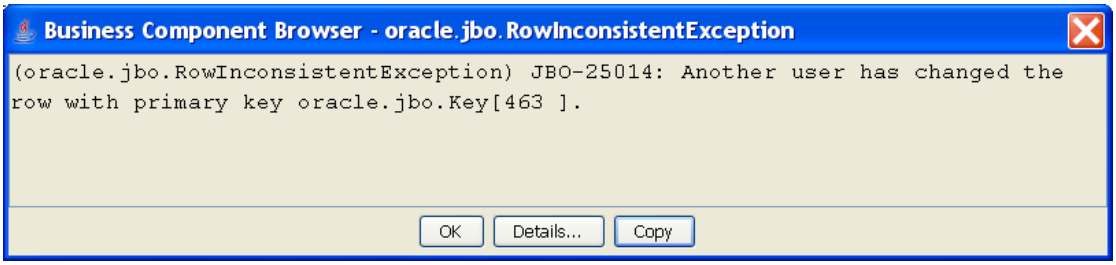


Figure 5 JDeveloper and ADF Architecture

The ADF Business Components technology implemented in JDeveloper automatically provides the Optimistic Locking services for the generated application depending on the selected locking mode. In ADF applications the optimistic lock exception appears as RowInconsistentException as seen in following message box example:



We have also tested JDeveloper's TopLink POJO, but it has turned out that the developers have decided not to implement ORA_ROWSCN support in the product in near future. Developers also seem to be strong believers in TopLink's own local cache (second-level cache above the DBMS bufferpool) as performance booster. However, bypassing this cache is a tricky issue which makes RVV discipline difficult to fulfil using TopLink, even if they recommend use of Optimistic Locking in the manual.

EJB3 JPA and imlementations

Oracle is the co-developer of the EJB3 specification with Sun and the TopLink architecture has affected the development of EJB3. EJB3 defines Java Persistence API (JPA) specification based on developments in following products: TopLink, Java Data Objects (JDO) of Sun, and Hibernate of JBoss. JPA simplifies the persistence management and Object/Relational mappings by configuring the behaviour of the objects by annotations embedded in the Java source code. An interesting annotation for our current topic is the @Version annotation of an entity which enables optimistic locking by defining the technical version field/column to be used by EntityManager (persistence manager) for version validation. For more information see http://otn.oracle.com/jpa.

With JavaEE SDK Sun is shipping TopLink Essentials as a reference implementation of EJB3 JPA called GlassFish JPA. Another TopLink Essentials variation is the new open source EclipseLink of Eclipse Group.

Other open source JPA implementations are Hibernate EntityManager and the new OpenJPA of Apache Group. We will focus on Hibernate in the next chapter below.

An example of JPA annotations can be found in Appendix 7 where we test the use of Hibernate EntityManager.

1.3.4 Hibernate

A competitor of TopLink is Hibernate, a free and popular ORM and persistence manager product for JavaEE applications as well as for standalone JavaSE Client/Server applications. Hibernate is an open source spin-off product of JBoss Application Server and it has been on the market over ten years. The developers have their own Web pages at http://www.hibernate.org. The original Hibernate engine is called as Hibernate Core. Hibernate has also affected the development of EJB3 Java Persistence API and provides now its EJB3 / JPA implementation (EntityManager and Annotations) as a wrapper of Hibernate Core. These two different APIs and available software stacks are presented roughly in the Figure 6.

Hibernate Core services can be accessed using the Hibernate API, which allows also direct JDBC access to the data sources. Just like TopLink, Hibernate tries to optimize data access performance using its own cache, which makes row version verification difficult. One must bypass the cache when fetching the current row version from the data source. The services of Hibernate Core can be configured programmatically and can be overwritten by XML-based configuration files. One configurable behaviour of the data access is the SQL Isolation Level, which unfortunately cannot be changed for a single transaction. We need this capability for Phase 6 in our example scenario, and we have solved this using JDBC API. The Hibernate programming paradigm is on higher level than JDBC, for example the database connections are managed automatically using a connection pool for every transaction.

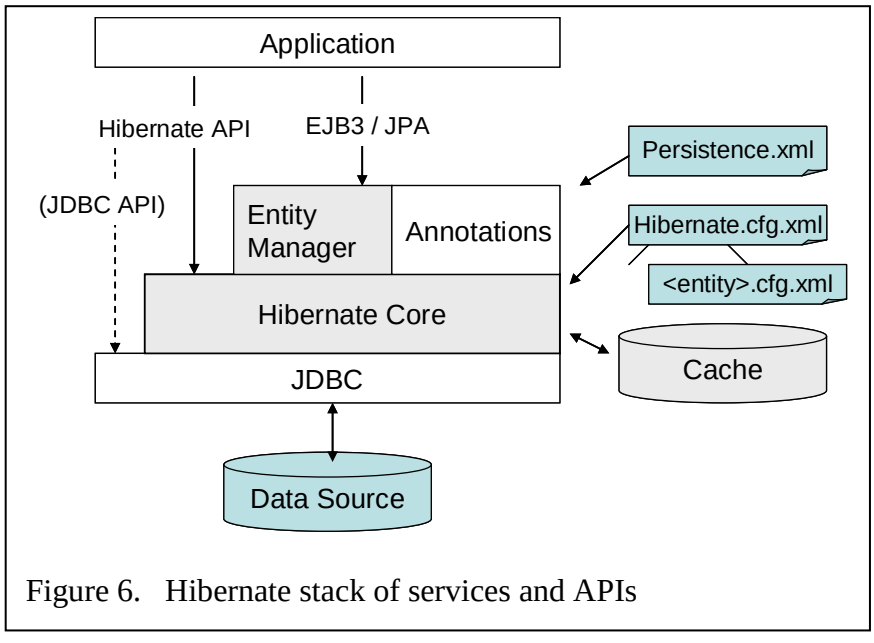


Figure 6. Hibernate stack of services and APIs

The programming paradigms for persistent classes using optimistic locking (described in the Hibernate 3.2.2 Reference Documentation) include the following

- version checking by application
- automatic version checking by entity manager

- automatic version checking of detached objects by entity manager

Automatic version checking takes place for every instance of the class at the commit phase of the transaction based on the technical version column in case of attribute setting

```
optimistic-lock="version"
```

As an alternate validation model in the Object/Relational Mapping for a class Hibernate provides validation based on a set of columns using attribute value

```
optimistic-lock="all"
```

which will compare the contents of all columns, or attribute value

```
optimistic-lock="dirty"
```

which will compare only the contents of columns which have been changed by the transaction.

The single technical column for version validation can be defined by the XML element `<version>` of the Hibernate Object/Relational Mapping declaration in an entity's cfg.xml file as follows:

```
<class name="foo" table="TABLE2" ..
     optimistic-lock="version">
      <id name="id" column="ID" />
      <version column="RV" generated="always" ../>
</class>
```

where the attribute value `generated="always"` means that the value of the technical column is generated by the DBMS whereas the attribute `generated="never"` means that Hibernate will generate the value while synchronizing the contents with the database. The drawback of the validation based on Hibernate generated technical column is that it is not reliable in case the data may get updated also by some other software.

In Appendix 6 we experiment with a small example of this to test how the RVV Discipline can be programmed using the Core + JDBC Services. Instead of automatic version checking we verify the row version by the application code. Solutions, results and findings are commented in the Appendix.

For Java Persistence API (JPA) Hibernate JPA provides implementation of EJB3 EntityManager and the EJB3 Annotations as a wrapper of the Hibernate Core. This provides a more limited and a bit different programming paradigm compared to Hibernate API. Also the configuration techniques are different. The entity behaviour is mainly configured using EJB3 annotations.

In Hibernate JPA the only portable optimistic locking seems to be based on automatic row version of single technical column generated by the persistence service.

Appendix 7 contains our RVV example implemented using Hibernate JPA compared with findings in Appendix 6.

Hibernate Core for Java has been ported also to the Microsoft .NET platform as NHibernate for .NET, but the evolution of persitence APIs continues and Microsoft is

introducing an ORM solution of its own as Language-Integrated Query LINQ to be discussed in the next chapter.

1.3.5 LINQ to SQL

The Language-Integrated Query (LINQ) includes ORM tools and a set of data providers and APIs accessing objects, XML, DataSets, or relational data directly using LINQ expressions as native part of the host language so that in the typical development workbench IDE, Visual Studio 2008, the developer using LINQ expressions is supported by the IntelliSence to avoid syntax errors and in case of working with relational data even checking names and types against the metadata in the database catalogs. The host language can be C#, Visual Basic .NET, etc .NET languages. LINQ to SQL is the API for accessing SQL Server databases, and in future also other databases.

The programming paradigm in LINQ to SQL reminds JPA, but may be easier. What it comes to query expression, LINQ to SQL queries look similar to SQL SELECT statements with the clauses in different order, and in fact the actions are mapped into SQL statements for run time. Compared with JPA, there is not way for direct use SQL statements, so all possibilities of Transact-SQL are not available for the programmer. Without going to the rich features and tools of LINQ we will focus only on the row version management. In Appendix 8 we have programmed our use case example in C# using LINQ to SQL data accessing so that it is easy to compare with the JPA programming paradigm.

For concurrency control LINQ uses the row version verification based on values of all columns fetched from the database row. Therefore in connected mode the ROWVERSION column is of no use, it is just harmful since it will introduce the U-LOCK problem in SELECT – UPDATE scenario, even if it is not included in the fetched columns. Since the database can be used by some other software too, we however keep the ROWVERSION in the base table, and to test how to cope with it in our program we even fetch it to the program. For disconnected mode like Web Services the ROWVERSION would be most useful. Since we have not found a way to introduce query options in LINQ queries, we enter the (UPDLOCK) option in the SQL view variant RvTestU of the original RvTest as follows

```
CREATE VIEW rvv.RvTestU AS
SELECT id, s, CAST(rv AS BIGINT) AS rv
FROM   rvv.VersionTest WITH (UPDLOCK)
```

This will request us the U-LOCK in SELECT-UDPATE, and often extra U-LOCKs requests. When we want avoid those extra U-LOCK requests, we can always access the database also using an alternate view which does not have the (UPDLOCK) option.

1.3.6 Web Services

Web Services is the basis of the new Service Oriented Architecture (SOA) in which applications are built of existing and published, stateless services to be accessed using XML based SOAP message dialogues (see SOAP 1.2 http://www.w3.org/TR/soap/).

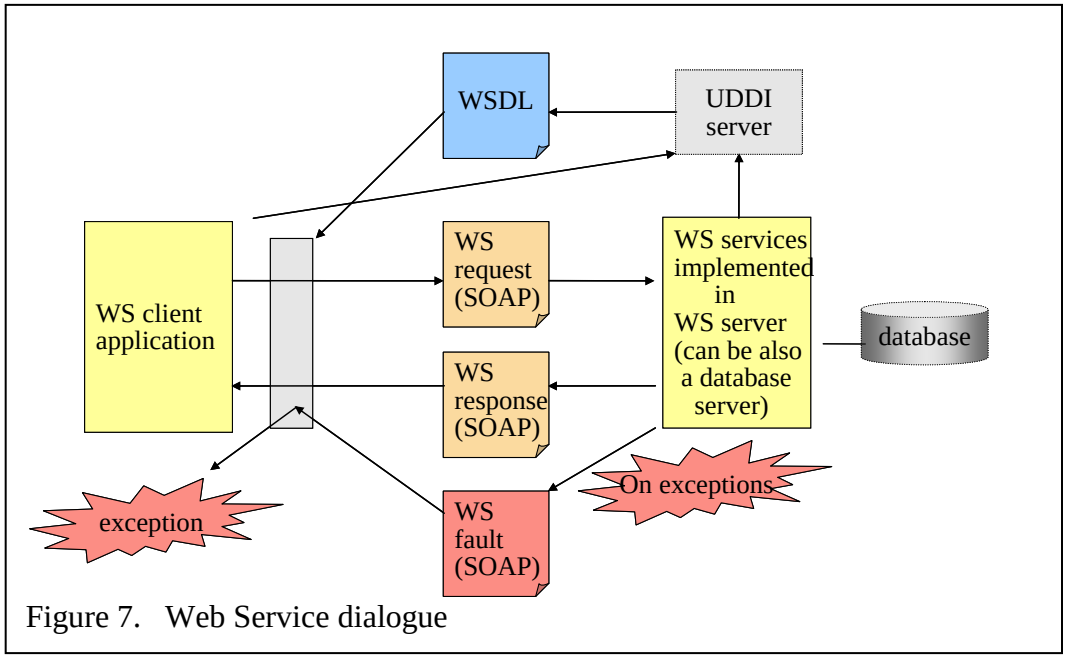


Figure 7. Web Service dialogue

Figure 7 presents a WS dialogue initiated by WS request

The WS client starts the dialogue sending WS request message to the WS server which processes the request and generates corresponding WS response message sending it to the client. However, if some exception is raised while processing the request then the server shall generate a WS fault message instead, and the WS proxy on the client side propagates the fault message raising corresponding exception to the client.

Web Services is the technology that can connect Java and .NET architectures. Web Services can be considered also as a data access technology since the latest versions of the mainstream DBMS systems (Oracle, DB2, and SQL Server) can provide Web Services. However, in Appendix 9 we show using just a simple ASP.NET example how our use case can be implemented using web service dialogues passing the row version data as extra parameter.

Local transactions of the Web Services are not a problem, but participating of a Web Service in global (distributed) transaction is a problem area which needs specifications of its own (see Web Services Transactions specifications), but this is out of scope of our study. We will focus on simple Web Services and which access databases using just reliable local transactions of their own, and proper exception handling at server-side and client-side of the SOAP message exchanges. Also the dialogue sequence with the same service provider can be made reliable using the RVV Discipline, as we demonstrate in Appendix 9.

On details of Web Services we refer to the vast literature of the subject, for example the J2EE™ Tutorial of Sun. Since we just want to demonstrate how to transfer the row version from a dialogue step to the next step for RVV, we have implemented both Web Services and the client using .NET since Visual Studio makes this a really easy task.

1.3.7 PHP

One of the first server-side scripting languages used for generating dynamic HTML pages is Perl. It was originally developed already in 1987 for system administration tasks in Unix platforms, but soon after invention of HTML it has been applied also to generate dynamic HTML pages using CGI technology. For accessing databases Perl needs DBMS dependent DBD (Database Driver) modules, and a de facto standard general DBI (Database Interface) wrapper has been written to provide a unified data access interface over these DBD modules.

CGI technology does not scale well and for performance reasons new technologies of server-side scripting engines integrated with the Web server have been developed. One of the most popular server-side scripting language is PHP which can easily be used to create dynamic Web pages which can also access databases. Instead of generic data access APIs PHP programmers need to use DBMS dependent libraries for accessing databases. In Appendix 10 we show how PHP code can access Oracle database using OCI8 API of Oracle or access SQL Server database using SQL Server Driver for PHP applying reliable SQL transactions and the RVV discipline. DB2 databases can also be accessed using special DB2 driver, but we leave it as exercise of interested readers.

1.3.8 Ruby

Ruby is a fairly new open source, object-oriented programming language which like Java is platform independent. As a dynamic, interpreted language it can also be considered as a sophisticated scripting language. It can be used as a standalone language, but recently it has become best known as part of the Ruby on Rails Framework. In Appendix 11 we will however focus on database access using the plain Ruby. Like PHP it does not have a generic native data access API of its own, but DBMS vendors have written proprietary libraries of their own like the OCI8 API of Oracle. However there is DBI API, a generic wrapper over various proprietary APIs like the OCI8 API. We will show samples codes of both of these in Appendix 11.

PART II: Server-Side Implementations of Incremental Change Identifier

In this part we will focus on the server-side SQL implementations of the incremental change identifier by DB2, Oracle and SQL Server products. All these products can increase some numeric row version column automatically by some row-level database trigger, but there may be alternate solutions. We call all these technologies as server-side stamping.

For portability reason we will apply in our trigger based solutions a 64-bit integer column called `rv` (for row version) starting from value 0 and incremented every time the row is updated, and after reaching the maximum value of a 64-bit integer we will reset the value to the lowest 64-bit integer value. Maybe we could use a 32-bit integer or 16-bit integer, but since it has hardly any effect on the performance we will use the maximal range of the values.

2.1 DB2

Let us create a test table as follows

```
CREATE TABLE VersionTest (
id     INT NOT NULL,
s      VARCHAR(20),
rv     BIGINT DEFAULT 0,
CONSTRAINT PK_VersionTest PRIMARY KEY (id)
) ;
```

for which we can create a row-level UPDATE trigger using DB2 SPL language as follows

```
CREATE TRIGGER TRG_VersionTest
NO CASCADE BEFORE UPDATE ON VersionTest
REFERENCING NEW AS new_row OLD AS old_row
FOR EACH ROW
MODE DB2SQL
IF (old_row.rv = 9223372036854775807) THEN
   SET new_row.rv = -9223372036854775808;
ELSE
   SET new_row.rv = old_row.rv + 1;
END IF;
```

The use of this trigger will cost us some 2 percent more in execution time of UPDATE commands, but no software can bypass it, so this is an ideal row version identifier for DB2.

In Appendix 1 we have tested suitability of timestamp data as an alternative for row version identifier and it turns out that it is not accurate enough at least on the current Windows platforms. However, DB2 V9.5 introduces the new **`ROW CHANGE TIMESTAMP`**, a semi-timestamp implementation to be used both as the row version

indicator and indicator of the time when the row has previously changed in any means. Compared with actual TIMESTAMP data the **ROW CHANGE TIMESTAMP** values for a row are made unique for all updates of the row and can be defined as row version indicator as follows:

```
CREATE TABLE VersionTest (
id      INT NOT NULL,
s       VARCHAR(20),
rv      TIMESTAMP NOT NULL
        GENERATED BY DEFAULT
        FOR EACH ROW ON UPDATE AS
        ROW CHANGE TIMESTAMP,
CONSTRAINT PK_VersionTest PRIMARY KEY (id)
) ;
```

For example on 32bit Windows platform where the accuracy of TIMESTAMP is milliseconds, we have seen that a sequence up to 21 updates can get the same timestamp value, but for **ROW CHANGE TIMESTAMP** DB2 will renumber them using the following sequence of microsecond offset values 0.000001, 0.000002, 0.000003, ... - and suprisingly this will not generate any performance penalty.
The `rv` column can be mapped into BIGINT typed row change token using following view definition:

```
CREATE VIEW RvTest
AS
SELECT id, s,
 (ROW CHANGE TOKEN FOR VersionTest) AS rv
FROM VersionTest;
```

and the `rv` column can be accessed just like in the trigger based solution, so this will be attractive row version indicator. In theory it is not fully reliable if the system clock is changed manually or in case system is returning from the sunlight saving time, whereas the trigger based solution is always 100 % reliable.

2.2 SQL Server

Transact-SQL does not provide actual row-level triggers like SQL standard, Oracle and DB2 do. However, we can do BIGINT row version stamping using the following command-level trigger, if we simply skip the overflow test of `rv` values:

```
CREATE TRIGGER TRG_VersionTest ON VersionTest
FOR UPDATE
AS
UPDATE VersionTest
SET rv = rv + 1
WHERE id IN (SELECT id FROM INSERTED);
```

Row-level processing can be implemented in Transact-SQL command-level trigger by cursor processing of the temporary tables INSERTED (row images with new values) or DELETED (row images with old values) as follows

```
CREATE TRIGGER TRG_VersionTest ON VersionTest
FOR UPDATE
AS BEGIN
    DECLARE @id INTEGER,
            @rv BIGINT ;
    DECLARE updc CURSOR FOR
            SELECT id, rv FROM INSERTED ;
    OPEN updc ;
    FETCH NEXT FROM updc INTO @id, @rv ;
    WHILE @@FETCH_STATUS = 0
    BEGIN
        IF (@rv = 9223372036854775807)
           SET @rv = -9223372036854775808
        ELSE
           SET @rv = @rv + 1 ;
        UPDATE VersionTest
        SET rv = @rv
        WHERE id = @id ;
        FETCH NEXT FROM updc INTO @id, @rv ;
    END ;
    CLOSE updc ;
    DEALLOCATE updc ;
END ;
```

As one might guess, this is slow. We have measured 2.6 times slower performance that is 160 % load increase using this solution.

Instead of presenting the trigger solution we recommend use of ROWVERSION data type for the column as follows.

```
CREATE TABLE VersionTest (
id      INT,      -- primary key
s       VARCHAR(20),    -- representing data columns ..
rv      ROWVERSION,
CONSTRAINT PK_VersionTest PRIMARY KEY (id)
) ;
```

ROWVERSION is a synonym name for Transact-SQL native data type TIMESTAMP. The name is misleading and the values are “measuring” time only on ordinal scale (Nilsson) as follows: In every SQL Server database there is an internal sequence generator that is used automatically to assign a new unique value for each row UPDATE for the column of TIMESTAMP data type. We prefer to use the name ROWVERSION.

Handling of ROWVERSION data in applications is difficult. In C# we can use a special class for it, but more generally it is better to cast the data type by Transact-SQL function

```
CAST (rv AS BIGINT)
```

using for example the following view instead the original table:

```
CREATE VIEW vVersionTest (id, s, rv) AS
```

```
SELECT id, s, CAST(rv AS BIGINT)
FROM VersionTest;
```

The use of the ROWVERSION column will cost us some 1-2 percent in execution time of UPDATE commands, but no software can bypass it.

2.3 Oracle

The table of the trigger-based example we presented above of DB2 can be modified to Oracle PL/SQL as follows:

```
CREATE TABLE VersionTest (
id     INT,            -- primary key
s      VARCHAR2(20),   -- representing data
rv     NUMBER DEFAULT 0,
  -- rv reserved for triggered RowVersioning
CONSTRAINT PK_VersionTest PRIMARY KEY (id)
) ;
```

and the trigger as follows:

```
CREATE OR REPLACE TRIGGER TRG_VersionTest
BEFORE UPDATE ON VersionTest
FOR EACH ROW
BEGIN
IF (:OLD.rv = 9223372036854775807) THEN
    :NEW.rv := -9223372036854775808;
ELSE
   :NEW.rv := :OLD.rv + 1;
END IF;
```

The use of this PL/SQL trigger will cost in execution time of UPDATE commands, but no software can bypass it. The license terms of Oracle do not allow publishing the performance results, but this can be easily tested by the interested readers themselves.

We have found a much more efficient solution for Oracle 10g R2 (see the SQL Reference manual) which introduces the new ROWDEPENDENCIES[2] table clause and pseudo column ORA_ROWSCN. When a table is created using the new ROWDEPENDENCIES clause as follows:

[2] Unfortunately the ROWDEPENDENCIES clause is not supported in the free Oracle XE version.

```
CREATE TABLE VersionTest (
id      INT,            -- primary key
s       VARCHAR2(20),   -- representing data
rv      NUMBER DEFAULT 0,
  -- rv just left here from the previous test
CONSTRAINT PK_VersionTest PRIMARY KEY (id)
) ROWDEPENDENCIES ;
```

this will increase the size of every row storage with SCN field of 6 bytes. The field will contain the System Change Number SCN of the latest committed transaction which updated the row and the value is available in the new pseudo column ORA_ROWSCN, except if the transaction itself has modified the row, in which case the ORA_ROWSCN contains NULL, as can be seen in the following test

```
SQL> INSERT INTO VersionTest (id,s) VALUES (1,'foo');

1 row created.

SQL> COMMIT;

Commit complete.

SQL> SELECT id,s,rv,ORA_ROWSCN FROM VersionTest;

        ID S                              RV ORA_ROWSCN
---------- -------------------- ---------- ----------
         1 foo                           0    4958855

SQL> UPDATE VersionTest SET s='newval' WHERE id = 1;

1 row updated.

SQL> SELECT id,s,rv,ORA_ROWSCN FROM VersionTest;

        ID S                              RV ORA_ROWSCN
---------- -------------------- ---------- ----------
         1 newval                        0 <NULL>

SQL> COMMIT;

Commit complete.

SQL> SELECT id,s,rv,ORA_ROWSCN FROM VersionTest;

        ID S                              RV ORA_ROWSCN
---------- -------------------- ---------- ----------
         1 newval                        0    4958857

SQL>.
```

If we now access the table through the following Base View

```
CREATE OR REPLACE VIEW RvTest (id, s, rv) AS
SELECT id, s, COALESCE(ORA_ROWSCN, 0)
FROM VersionTest;
```

then the `rv` column of the view can be used for the row version validation. It will cost the extra 6 bytes space for every row, but with minimal performance overhead

compared with the trigger-based solution. The problem is that all tables created in older Oracle versions need to be rebuilt using the new ROWDEPENDENCIES clause.

Summary

In Part I we have been reviewing the Lost Update Problem, Blind Overwriting and the concepts of Optimistic Concurrency for avoiding the problem. We are not interested in the so called "optimistic concurrency control" (typically based on multi-versioning) services provided by some DBMS systems inside a single transaction, but in programming methods called "optimistic locking" to avoid the "blind writer anti-pattern" i.e. writing over updates of transactions which are running concurrently with series of transactions of our use case. We consider this as a pure programming discipline issue on the client-side, independent of the DBMS to be used, but some middleware services may help in this discipline or complicate things like J2EE EJB. We see the programming discipline as a row version verification (RVV) issue, which can be based on verifying versions of rows based on values of columns which our use case is processing, or based on the stamped value of some technical row version column. Some software tends to maintain these technical row version columns by client-side stamping for performance benefits, but this is error prone if it needs programming and totally unreliable if the data in the database is maintained also by some external software which does not use the very same discipline. Reliable and unavoidable maintenance of the row version column we get only as a **server-side stamping service**, although at the price of some performance overhead.

In Part II we have been investigating what kind of services we can get from the server-side for automatic maintenance of the row version column from the current mainstream RDBMS systems, DB2, Oracle and SQL Server. A row-level UPDATE trigger is a general solution available in almost every mainstream RDBMS, in DB2 even at very acceptable performance, but SQL Server and Oracle can provide a much more efficient alternate solution of their own compared to their trigger solutions. These server-side row version stamping services suit well in the principle of SQL Base Views in which the tables are never accessed directly but for every table a base view is created instead to fulfil the principle of data independence.

In the Appendixes we finally test and verify some key data access technologies in terms of RVV. Even if the current trend is rapid program development using quite abstract level programming paradigms, the less we know of the underlying functionalities the less reliable are our applications. For example the middleware caching services may ruin the reliability of RVV. SQL Server's Profiler tracing of the DBMS interface has been one of the most valuable tools to verify what is really going on under the software layers.

If we omit the RVV Discipline in software development, our software may "usually work without side effects" - but for a high quality software this is not enough. It is necessary that software architects, application developers, and data access programmers understand the Blind Overwriting Problem and select a consistent RVV discipline for avoiding it.

A comparison of the programming paradigms

If we consider the row update types applicable for Phase 6 which we presented in chapter 1.1 and look at what kind of the types can be applied in the programming paradigms discussed in chapter 1.3 and the corresponding Appendices, we will notice the following scenario:

	Type 0 (sensitive, no risk)	Type 1 (update with rvv predicates)	Type 2 (select-if-then-update)
JDBC client	yes	yes	yes
ADO.NET client	yes	yes	yes
ADO.NET DataSet		yes (set of rows!)	
PHP data access APIs	yes	yes	yes
Ruby	yes	yes	yes
J2EE BMP		yes	
J2EE CMP		?	
TopLink		?	yes
Hibernate Core		?	yes
JPA		?	yes
LINQ to SQL		automatic	
Ruby on Rails			?
Web Services	yes	yes	yes

On implementation of Web Services the programmer is free to select the local data access technology at server-side, so we can say that “anything goes”. For persistent objects in object-oriented data access technologies the Type 0 (sensitive) update form can not be used and therefore these are susceptible to the Blind Overwriting problem. Type 1 update is mainly available as a configurable service.

In a transaction using Type 2 (select-if-then-update) we read the current row version from the database before the update. In this case we need to avoid the cache services to make sure that the row version really is the current version in the database. Still, we are allowed to UPDATE the row only if no one else has managed to update the row after the SELECT. If our DBMS is using multi-versioning for concurrency control (MVCC), for example Oracle SERIALIZABLE and SQL Server Snapshot isolation, we may lose the competition to concurrent updates since “the first writer wins” in MVCC. If our DBMS is using locks as concurrency control we just need to set strong enough isolation level like Repeatable Read, with exception of SQL Server using TIMESTAMP column for which we found as solution the UPDLOCK view. For Oracle we can achieve the row-level locking by using the form SELECT .. FOR UPDATE, in which case we cannot use ORA_ROWSCN for verifying the row version, but we should verify the row version based on comparison of all columns in the applied select view.

Acknowledgements

DBTechNet is an initiative and network of European teachers on database technologies. The cooperation started early in 1996 when we got worried of the popularity among young programmers especially in SME companies of the easy-to-use desktop-based DAO Data Access Object model in Visual Basic family of languages. This often led to chaos-like transaction programming, which "usually seemed to work". We realized the need for proper transaction programming models (Data Access Patterns) in Data Access Technologies. The cooperation has then been expanded to cover methodologies and technologies also in the fields of Database Design and Database Administration.

This paper was inspired by discussions with the members of the DBTech network and the ideas presented during the DBTech Pro workshops - learning by doing and verifying things. The DBTech Pro project was supported by the European Leonardo da Vinci programme during the years 2002 - 2005. You can find more information on DBTechNet and DBTech Pro on our Web pages at www.DBTechNet.Org.

References

Christian Bauer, Gavin King: Java Persistence with Hibernate, Manning, 2007

Philip A. Bernstein, Eric Newcomer: Principles of Transaction Processing, Morgan Kaufmann, 1997

David Chappell: Understanding .NET, A Tutorial and Analysis, Addison-Wesley 2002

C. J. Date: An Introduction to Database Systems Vol I, 1986 (or later)

Linda DeMichiel, Michael Keith: Enterprise JavaBeans™,Version 3.0, EJB Core Contracts and Requirements (JSR 220), Sun, Proposed Final Draft December 21, 2005

Erich Gamma et al: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

Hibernate Reference Documentation, Version 3.2.2 (Hibernate core)

Hibernate EntityManager User guide, Version 3.3.0.GA

Hibernate Annotations Reference Guide, Version 3.3.0.GA

Glen Johnson: Programming ADO.NET 2.0 Applications, Microsoft Press, 2005

IBM: DB2 Version 9 for Linux, UNIX, and Windows, System Monitor Guide and Reference SC10-4251-00

Floyd Marinescu: EJB Design Patterns, Wiley, 2002

Jimmy Nilsson: .NET Enterprise Design with Visual Basic .NET and SQL Server 2000, SAMS, 2002

Clifton Nock: Data Access Patterns, Addison-Wesley, 2004

Avrom Roy-Faderman et al: Oracle JDeveloper 10g Handbook, Oracle Press, 2004

Oracle: SQL Reference 10g Release 2 (10.2) B14200-01, June 2005

Oracle: TopLink Developer's Guide 10g (10.1.3.1.0) B28218-01, September 2006

Oracle: Application Development Framework Developer's Guide 10g Release 3 (10.1.3.0) B28967-01, June 2006

Ed Roman: Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition, Wiley, 1999

Sun: The J2EETM 1.4 Tutorial

Web links:

Hibernate
http://www.hibernate.org/

Thiru Thangarathinam: Exception Handling in Web Services
http://www.developer.com/net/csharp/article.php/10918_3088231_1

Brian Swan: Accessing SQL Server Databases with PHP
http://msdn.microsft.com/en-us/library/cc793139.aspx

W3C: SOAP Version 1.2
http://www.w3.org/TR/soap/

Web Services Transactions specifications
http://www.ibm.com/developerworks/library/specification/ws-tx/

Appendix 1 Testing the Myth of DBMS Timestamp Uniqueness

Database textbooks used to mention timestamp as a possible row version identifier, and as we have seen above today even some software tools are still offering it as an option. The reliability of this method **depends on timestamp accuracy offered by the DBMS in SQL,** but this can be very different from the timestamps of the operating system. In the following we test how the mainstream RDBMS systems behave today on a typical 32bit Windows platform:

- Accuracy of SQL Server 2005 DATETIME, which stands also for ISO timestamp in Transact-SQL, is internally 3.33 milliseconds and gives following result

```
SELECT GETDATE() AS timestamp ;
timestamp
-----------------------
2007-10-30 12:25:50.780
```

- The nominal accuracy and accuracy on Linux platform of DB2 TIMESTAMP is microseconds, but on DB2 LUW V9 on 32bit Windows platform the actual accuracy of TIMESTAMP is milliseconds while 3 lowest decimals are always zeroes, for example applied to a single row in test table T

```
SELECT CURRENT_TIMESTAMP AS timestamp FROM T WHERE id=1
TIMESTAMP
2007-10-30-11.10.40.109000
```

  However, in version V9.5 the developers of DB2 have introduced a modified TIMESTAMP implementation called ROW CHANGE TIMESTAMP which provides unique timestamp values for any row version updates. We have covered this briefly in chapter 2.1.

- For Oracle we can define the TIMESTAMP accuracy up to 9 decimals, but again on 32bit Windows platform the real accuracy is a 3 decimal fraction of seconds, as we can see of the following example

```
SQL> SELECT CURRENT_TIMESTAMP(9) FROM DUAL ;
CURRENT_TIMESTAMP(9)
---------------------------------------
30.10.2007 11:34:08,307000000 +02:00
```

But in a 0.001 seconds time a DBMS can update the very same row in many transactions. We prove this by the following simple test on Oracle on 32bit Windows platform as follows:

```
SQL> CREATE SEQUENCE id_gen;

Sequence created.

SQL> CREATE TABLE Timestamps (
  2  id      INTEGER   NOT NULL  PRIMARY KEY,
  3  sessio  INTEGER,
  4  tstamp  TIMESTAMP  DEFAULT CURRENT_TIMESTAMP(9)
```

```
  5  );

Table created.

SQL> CREATE OR REPLACE PROCEDURE InsertTimestamps (sess INT)
  2  AS
  3  BEGIN
  4  FOR i IN 1..1000 LOOP
  5    INSERT INTO Timestamps (id,sessio)
              VALUES (id_gen.NEXTVAL, sess);
  6    COMMIT;
  7  END LOOP;
  8  END;
  9  /

Procedure created.

SQL> EXECUTE InsertTimestamps (1);

PL/SQL procedure successfully completed.

SQL> -- How unique timestamps we've got ?
SQL> SELECT tstamp,COUNT(*)
  2  FROM Timestamps
  3  GROUP BY tstamp
  4  HAVING COUNT(*) > 10;

TSTAMP                         COUNT(*)
-----------------------------------
31.10.2007 17:58:55,930000         88
31.10.2007 17:58:55,945000        112
31.10.2007 17:58:56,008000        110
31.10.2007 17:58:56,117000         20
31.10.2007 17:58:55,836000         23
31.10.2007 17:58:55,992000         94
31.10.2007 17:58:56,054000         97
31.10.2007 17:58:55,867000        115
31.10.2007 17:58:55,883000        113
31.10.2007 17:58:56,070000        112
31.10.2007 17:58:55,852000        116
11 rows selected.
```

The conclusion of the simple tests is that we can forget using SQL timestamps as the row version identifier, even if on some other platforms such as Unix and Linux the accuracy of Oracle TIMESTAMP can be 6 decimals of a second.

Beside the accuracy problem all timestamps are problematic at least during one hour a year when the system clocks are moved backwards after returning to normal time from the daylight saving time, and in addition on different hours depending on the local time zones round the world – while the daylight saving time itself is just a stupid relic from history, if we may say so.

Appendix 2 Testing the Behaviour of the Mainstream DBMS Systems in a Simple SELECT-UPDATE Concurrency Case

We will experiment with concurrency management of the mainstream DBMS systems DB2, SQL Server, and Oracle using a minimalistic SELECT-UPDATE test case in plain SQL based on the simple concurrency scenario of listing 1 in chapter 1. This

will also serve as reference for implementations of Phase 6 of our use case scenario in Figure 1, in Chapter 1.2. Even if the concurrency scenario is simple, the technical details are not trivial.

We have created the following table and content in the test databases:

```
CREATE TABLE Accounts (
acctId  INTEGER  NOT NULL  PRIMARY KEY,
balance DECIMAL(11,2)  NOT NULL
);
INSERT INTO Accounts (acctId, balance) VALUES (100, 10000);
COMMIT;
```

DB2 Express-C V 9.1

The isolation levels of DB2 (except CS) match the isolation levels of SQL standard but using different names (and command syntax) as follows:

DB2:	ISO SQL standard:
UR (uncommitted read)	Read Uncommitted
CS (cursor stability, S-lock on current row)	Read Committed
RS (read stability)	Repeatable Read
RR (repeatable read)	Serializable

The Command Editor and CLP Window seem to work in autocommit mode, so we use separate DB2 Command Windows for processes A and B (presented in textboxes) as follows and "+c" option in front of every command to turn off autocommit:

Command Window of process A:

```
C:\IBM\SQLLIB\BIN>DB2 +c SET CURRENT ISOLATION RS
DB20000I  The SQL command completed successfully.

C:\IBM\SQLLIB\BIN>DB2 +c SELECT balance FROM Accounts WHERE acctId=100

BALANCE
-------------
      1000.00

  1 record(s) selected.
```

Command Window of process B:

```
C:\IBM\SQLLIB\BIN>DB2 +c UPDATE Accounts SET balance=balance+200 WHERE
acctId=100
```

```
C:\IBM\SQLLIB\BIN>rem Process A step 7
C:\IBM\SQLLIB\BIN>DB2 +c get snapshot for locks on TEST global
...
```

```
C:\IBM\SQLLIB\BIN>DB2 +c UPDATE Accounts SET balance=1100 WHERE acctId=100
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0911N  The current transaction has been rolled back because of a deadlock
or timeout.  Reason code "2".  SQLSTATE=40001
C:\IBM\SQLLIB\BIN>
```

Command Window of process B:

```
DB20000I  The SQL command completed successfully.
C:\IBM\SQLLIB\BIN>DB2 +c COMMIT
DB20000I  The SQL command completed successfully.
```

Note: If we look at the snapshot of locks (the get snapshot listings would have been long so we have skipped here, see DB2 System Monitor Guide and Reference), UPDATE command of transaction B waits for X-lock on the row, but gets U-lock as the first transaction requesting X-lock, so when UPDATE command transaction A requests for X-lock this will conflict with the U-lock of transaction B and DB2 selects A as victim of the deadlock..

We repeat the test using isolation level RR (=ISO Serializable):

Command Window of process A:

```
C:\IBM\SQLLIB\BIN>DB2 +c SET CURRENT ISOLATION RR
DB20000I  The SQL command completed successfully.

C:\IBM\SQLLIB\BIN>DB2 +c SELECT balance FROM Accounts WHERE acctId=100

BALANCE
-------------
      1000.00

  1 record(s) selected.
```

Command Window of process B:

```
C:\IBM\SQLLIB\BIN>DB2 +c UPDATE Accounts SET balance=balance+200 WHERE
acctId=100
```

```
C:\IBM\SQLLIB\BIN>rem Process A step 7
C:\IBM\SQLLIB\BIN>DB2 +c get snapshot for locks on TEST global
...
C:\IBM\SQLLIB\BIN>DB2 +c UPDATE Accounts SET balance=1100 WHERE acctId=100
DB20000I  The SQL command completed successfully.
C:\IBM\SQLLIB\BIN>DB2 +c get snapshot for locks on TEST global
...
C:\IBM\SQLLIB\BIN>DB2 +c COMMIT
DB20000I  The SQL command completed successfully.
```

Command Window of process B:

```
DB20000I  The SQL command completed successfully.

C:\IBM\SQLLIB\BIN>DB2 +c COMMIT
DB20000I  The SQL command completed successfully.

C:\IBM\SQLLIB\BIN>DB2 +c SELECT balance FROM Accounts WHERE acctId=100

BALANCE
-------------
      1300.00

  1 record(s) selected.
```

Verifying from lock snapshots using isolation level RR for transaction A it gets S-lock on the table for SELECT and requests for IX. The UPDATE command of B requests first IX on the table blocking on S of A, so A gets X lock for its UPDATE command. Transactions serialize fine and the balance gets updated correctly without need for retries.

SQL Server 2005

We open 2 query windows of SQL Server Management Studio, one for Process A, one for Process B. The window of Process A will also be used for monitoring the locks:

```
-- Process A (spid 52) steps 1-3
BEGIN TRANSACTION
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SELECT balance FROM Accounts WHERE acctId=100
balance
---------------------------------------
10000.00

(1 row(s) affected)
```

```
-- Process B (spid 53) step 5
UPDATE Accounts SET balance=balance+200 WHERE acctId=100
```

(Executing query ..)

```
-- Process A (spid 52) step 7
sp_lock
spid   dbid   ObjId       IndId  Type Resource                       Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     7      0           0      DB                                    S        GRANT
52     7      5575058     1      PAG  1:159                            IS       GRANT
52     1      1115151018  0      TAB                                   IS       GRANT
52     7      5575058     0      TAB                                   IS       GRANT
52     7      5575058     1      KEY  (6400b740ff6a)                   S        GRANT
53     7      5575058     1      KEY  (6400b740ff6a)                   X        WAIT
53     7      5575058     0      TAB                                   IX       GRANT
53     7      5575058     1      PAG  1:159                            IX       GRANT
```

```
53     7      0           0      DB                                  S        GRANT

UPDATE Accounts SET balance=1100 WHERE acctId=100
(1 row(s) affected)
sp_lock
spid   dbid   ObjId       IndId  Type Resource                       Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     7      0           0      DB                                  S        GRANT
52     7      5575058     1      PAG  1:159                          IX       GRANT
52     1      1115151018  0      TAB                                 IS       GRANT
52     7      5575058     0      TAB                                 IX       GRANT
52     7      5575058     1      KEY  (6400b740ff6a)                  X        GRANT
53     7      5575058     1      KEY  (6400b740ff6a)                  X        WAIT
53     7      5575058     0      TAB                                 IX       GRANT
53     7      5575058     1      PAG  1:159                          IX       GRANT
53     7      0           0      DB                                  S        GRANT
54     7      0           0      DB                                  S        GRANT

COMMIT
Command(s) completed successfully.
```

```
-- ... Process B (spid 53) step 5
 (1 row(s) affected)
SELECT * FROM Accounts
acctId      balance
----------- ---------------------------------------
100         1300.00

(1 row(s) affected)
```

Execution of both transactions serializes successfully and the total result is correct.

Using isolation level SERIALIZABLE for Transaction A yields exactly the same locking scenario and produces the same correctly updated value of balance.

In Chapter 2.2 we have studied the use of ROWVERSION column for row version verification, so let's add the new technical column in our table as follows

```
ALTER TABLE Accounts ADD rv ROWVERSION ;
```

and repeat our experiment using also a 3rd window (spid 54) for monitoring the locks

```
-- Process A steps 1-3
BEGIN TRANSACTION
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SELECT balance FROM Accounts WHERE acctId=100
balance
---------------------------------------
1000.00

(1 row(s) affected)
```

```
-- Process B step 5
BEGIN TRANSACTION
UPDATE Accounts SET balance=balance+200 WHERE acctId=100
(Executing query ..)
```

```
-- Process A step 7
sp_lock
spid   dbid   ObjId       IndId  Type Resource                       Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     7      0           0      DB                                  S        GRANT
52     7      5575058     1      PAG  1:159                          IS       GRANT
```

```
52      1       1115151018  0       TAB                                     IS        GRANT
52      7       5575058     0       TAB                                     IS        GRANT
52      7       5575058     1       KEY   (6400b740ff6a)                    S         GRANT
53      7       5575058     1       KEY   (6400b740ff6a)                    X         CNVT
53      7       5575058     1       KEY   (6400b740ff6a)                    U         GRANT
53      7       5575058     0       TAB                                     IX        GRANT
53      7       5575058     1       PAG   1:159                             IX        GRANT
53      7       0           0       DB                                      S         GRANT
54      7       0           0       DB                                      S         GRANT
```

```
UPDATE Accounts SET balance=1100 WHERE acctId=100
```

```
sp_lock -- 54
spid   dbid   ObjId       IndId  Type Resource                         Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     7      0           0      DB                                    S        GRANT
52     7      5575058     1      PAG  1:159                            IU       GRANT
52     7      5575058     0      TAB                                   IX       GRANT
52     7      5575058     1      KEY  (6400b740ff6a)                   S        GRANT
52     7      5575058     1      KEY  (6400b740ff6a)                   U        CNVT
53     7      5575058     0      TAB                                   IX       GRANT
53     7      5575058     1      KEY  (6400b740ff6a)                   X        CNVT
53     7      5575058     1      KEY  (6400b740ff6a)                   U        GRANT
53     7      5575058     1      PAG  1:159                            IX       GRANT
53     7      0           0      DB                                    S        GRANT
54     7      0           0      DB                                    S        GRANT
54     1      1115151018  0      TAB                                   IS       GRANT
```

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 52) was deadlocked on lock resources with another process and
has been chosen as the deadlock victim. Rerun the transaction.
```

Let's try this using SERIALIZABLE isolation level for transaction A:

```
-- Process A (spid 52) steps 1-3
BEGIN TRANSACTION
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
SELECT balance FROM Accounts WHERE acctId=100
balance
---------------------------------------
1000.00

(1 row(s) affected)
```

```
-- Process B (spid 53) step 5
BEGIN TRANSACTION
UPDATE Accounts SET balance=balance+200 WHERE acctId=100
(Executing query ..)
```

```
-- Process A (spid 52) step 7
sp_lock
spid   dbid   ObjId       IndId  Type Resource                         Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     7      0           0      DB                                    S        GRANT
52     7      5575058     1      PAG  1:159                            IS       GRANT
52     1      1115151018  0      TAB                                   IS       GRANT
52     7      5575058     0      TAB                                   IS       GRANT
52     7      5575058     1      KEY  (6400b740ff6a)                   S        GRANT
53     7      5575058     1      KEY  (6400b740ff6a)                   X        CNVT
53     7      5575058     1      KEY  (6400b740ff6a)                   U        GRANT
53     7      5575058     0      TAB                                   IX       GRANT
53     7      5575058     1      PAG  1:159                            IX       GRANT
53     7      0           0      DB                                    S        GRANT
54     7      0           0      DB                                    S        GRANT
```

```
UPDATE Accounts SET balance=1100 WHERE acctId=100
       ( immediate deadlock detection so no time to check the locks by spid 54)

Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 52) was deadlocked on lock resources with another process and
has been chosen as the deadlock victim. Rerun the transaction.
```

So introducing the ROWVERSION column forces the optimizer to use U-locks resulting to deadlock. After studying SQL Server 2005 Books Online / Transact-SQL pages on Table Hints, we decided to apply the table hint (UPDLOCK) on the SELECT command of transaction A. This will allow concurrent reading but blocks new U-lock requests.

```
-- Process A (spid 52) steps 1-3
BEGIN TRANSACTION
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SELECT balance FROM Accounts (UPDLOCK) WHERE acctId=100
balance
---------------------------------------
1000.00

(1 row(s) affected)
```

```
       -- Process B (spid 53) step 5
       BEGIN TRANSACTION
       UPDATE Accounts SET balance=balance+200 WHERE acctId=100
       (Executing query ..)
```

```
-- Process A (spid 52) step 7
sp_lock
spid   dbid   ObjId       IndId  Type Resource                       Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     7      0           0      DB                                    S        GRANT
52     7      5575058     1      PAG  1:159                            IU       GRANT
52     1      1115151018  0      TAB                                   IS       GRANT
52     7      5575058     0      TAB                                   IX       GRANT
52     7      5575058     1      KEY  (6400b740ff6a)                   U        GRANT
53     7      5575058     1      KEY  (6400b740ff6a)                   U        WAIT
53     7      5575058     0      TAB                                   IX       GRANT
53     7      5575058     1      PAG  1:159                            IU       GRANT
53     7      0           0      DB                                    S        GRANT
54     7      0           0      DB                                    S        GRANT

UPDATE Accounts SET balance=1100 WHERE acctId=100
(1 row(s) affected)
sp_lock
spid   dbid   ObjId       IndId  Type Resource                       Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     7      0           0      DB                                    S        GRANT
52     7      5575058     1      PAG  1:159                            IX       GRANT
52     1      1115151018  0      TAB                                   IS       GRANT
52     7      5575058     0      TAB                                   IX       GRANT
52     7      5575058     1      KEY  (6400b740ff6a)                   X        GRANT
53     7      5575058     1      KEY  (6400b740ff6a)                   U        WAIT
53     7      5575058     0      TAB                                   IX       GRANT
53     7      5575058     1      PAG  1:159                            IU       GRANT
53     7      0           0      DB                                    S        GRANT
54     7      0           0      DB                                    S        GRANT

COMMIT
Command(s) completed successfully.
```

```
-- .. Process B step 5
(1 row(s) affected)
SELECT balance FROM Accounts
balance
---------------------------------------
1300.00

(1 row(s) affected)
```

So the table hint UPDLOCK solves the deadlock problem on using the SQL Server ROWVERSION column.

In addition to the ISO SQL isolation levels SQL Server 2005 introduces a new isolation level called SNAPSHOT. This applies optimistic concurrency control for databases configured to provide this service. In all write operations old versions of updated rows are copied and chained in the TempDB database of the SQL Server instance. Read operations will access the committed versions. On isolation level Snapshot the read operations do not acquire S-locks, but will see only the latest row versions committed before the start time of the transaction. For update/delete operations of rows for which there already exists a new version the DBMS will raise serialization conflict. We applied this to our test case as follows:

```
CREATE DATABASE SI_TESTS  -- for Snapshot Isolation tests
GO
ALTER DATABASE SI_TESTS SET ALLOW_SNAPSHOT_ISOLATION ON
USE SI_TESTS
CREATE TABLE Accounts (
acctId  INTEGER  NOT NULL  PRIMARY KEY,
balance DECIMAL(11,2)  NOT NULL
)
GO
ALTER TABLE Accounts ADD rv ROWVERSION ;
GO
INSERT INTO Accounts (acctId, balance) VALUES (100, 10000);
UPDATE Accounts SET balance=1000 WHERE acctId=100
GO
-- Process A steps 1-3
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRANSACTION
SELECT balance FROM Accounts
WHERE acctId=100
balance
---------------------------------------
1000.00

(1 row(s) affected)

        -- Process B step 5
        BEGIN TRANSACTION
        SET TRANSACTION ISOLATION LEVEL READ COMMITTED
        UPDATE Accounts SET balance=balance+200 WHERE acctId=100
        (1 row(s) affected)

-- monitor window of spid 54
sp_lock
spid   dbid   ObjId       IndId  Type Resource                       Mode     Status
------ ------ ----------- ------ ---- -------------------------------- -------- ------
52     10     0           0      DB                                    S        GRANT
53     10     2073058421  1      PAG  1:143                            IX       GRANT
53     10     2073058421  0      TAB                                   IX       GRANT
53     10     0           0      DB                                    S        GRANT
53     10     2073058421  1      KEY  (6400b740ff6a)                  X        GRANT
54     10     0           0      DB                                    S        GRANT
54     1      1115151018  0      TAB                                   IS       GRANT
-- Process A step 7
```

```
UPDATE Accounts SET balance=1100 WHERE acctId=100
Executing ...

-- monitor window of spid 54
sp_lock
spid   dbid   ObjId       IndId  Type Resource                        Mode     Status
------ ------ ----------- ------ ---- ---------------------------------- -------- ------
52     10     0           0      DB                                    S        GRANT
52     10     2073058421  1      PAG  1:143                            IX       GRANT
52     10     2073058421  0      TAB                                   IX       GRANT
52     10     2073058421  1      KEY  (6400b740ff6a)                   X        WAIT
53     10     2073058421  1      KEY  (6400b740ff6a)                   X        GRANT
53     10     2073058421  0      TAB                                   IX       GRANT
53     10     2073058421  1      PAG  1:143                            IX       GRANT
53     10     0           0      DB                                    S        GRANT
54     10     0           0      DB                                    S        GRANT
54     1      1115151018  0      TAB                                   IS       GRANT

       -- Process B step 6
       COMMIT
       Command(s) completed successfully.

--  .. Process A step 7
Msg 3960, Level 16, State 5, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot
isolation to access table 'dbo.Accounts' directly or indirectly in database 'SI_TESTS'
to update, delete, or insert the row that has been modified or deleted by another
transaction. Retry the transaction or change the isolation level for the update/delete
statement.
COMMIT
Msg 3902, Level 16, State 1, Line 1
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

This proves that transaction A applying Snapshot isolation will be selected as victim of the serialization conflict and SQL Server automatically rolls it back. We have tested this both with and without the ROWVERSION column and this has no effect on the concurrency scenario.

Oracle 11g 1

Oracle supports only isolation levels READ COMMITTED (the default) and SERIALIZABLE (which is actually snapshot serializable applying optimistic concurrency control). Using the SERIALIZABLE isolation level our test case runs as follows:

```
-- Process A steps 1-3
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT balance FROM Accounts WHERE acctId=100;
BALANCE
----------------------
10000

1 rows selected

-- Process B step 5
UPDATE Accounts SET balance=balance+200 WHERE acctId=100
1 rows updated
-- Process A step 7
UPDATE Accounts SET balance=1100 WHERE acctId=100
(executing ..)
-- Process B step 6
COMMIT;
COMMIT succeeded.
```

```
-- .. Process A step 7
Error starting at line 2 in command:
UPDATE Accounts SET balance=1100 WHERE acctId=100
Error report:
SQL Error: ORA-08177: can't serialize access for this transaction
08177. 00000 -  "can't serialize access for this transaction"
*Cause:    Encountered data changed by an operation that occurred after
           the start of this serializable transaction.
*Action:   In read/write transactions, retry the intended operation or
           transaction.
```

Oracle does not use read locks on rows and for write locked rows Oracle provides other readers old version of the row. UPDATE made by transaction B first blocks the UPDATE of transaction A and after B commits its work Oracle rejects the transaction A since it cannot serialize it with transaction B.

Instead of using isolation level SERIALIZABLE, Oracle users usually prefer to use **SELECT..FOR UPDATE** -locking of rows as follows.

```
-- Process A steps 1-3
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION succeeded.
SELECT balance FROM Accounts WHERE acctId=100 FOR UPDATE;
BALANCE
----------------------
1000

1 rows selected
-- Process B step 5
UPDATE Accounts SET balance=balance+200 WHERE acctId=100;
(executing ..)
-- Process A step 7
UPDATE Accounts SET balance=1100 WHERE acctId=100;
1 rows updated
COMMIT;
COMMIT succeeded.
-- .. Process B step 5
1 rows updated
COMMIT;
COMMIT succeeded.
SELECT * FROM Accounts;
ACCTID                BALANCE
--------------------- ----------------------
100                   1300
```

1 rows selected

Appendix 3 A Baseline Implementation of RVV in Java/JDBC

SQL Server 2005 has a new interesting authorization scheme which we use in the following to create our test environment:

```
CREATE DATABASE TEST;
GO
USE TEST
GO
```

```
CREATE SCHEMA rvv AUTHORIZATION dbo;
CREATE LOGIN rvv WITH PASSWORD = 'test', DEFAULT_DATABASE=TEST;
CREATE USER rvv FOR LOGIN rvv WITH DEFAULT_SCHEMA = rvv;
GO
CREATE TABLE rvv.VersionTest(
id      INT NOT NULL,
s       VARCHAR(20),
rv      ROWVERSION,
CONSTRAINT PK_RvTest1 PRIMARY KEY (id)
) ;
GO
CREATE VIEW rvv.RvTest (id,s,rv) AS
SELECT id,s,CAST(rv AS BIGINT) rv
FROM rvv.VersionTest WITH (UPDLOCK);
GO
GRANT SELECT,UPDATE,INSERT ON rvv.RvTest TO rvv;
GO
SETUSER 'RVV';
USE TEST;
INSERT INTO RvTest (id,s) VALUES (1,'some text');
INSERT INTO RvTest (id,s) VALUES (2,'some text');
```

For DB2 and Oracle the implementation is more simple.

Writing code that can be used with any DBMS system turned out to be complicated. In step/phase 2 of the use case scenario we would like to use READ UNCOMMITTED isolation, but MVCC systems don't support it and all JDBC drivers don't propagate it to READ COMMITTED. The propagation is just recommendation in the JDBC specification, so we need to use adaptive programming and set the lowest isolation level depending on the values supported by the DBMS. For RVV we use the tables thrue views like RvTest above, but for step 2 decided to create a separate view as follows

```
CREATE VIEW RVV.RvTestList
AS
SELECT id, s
FROM   RVV.VersionTest ;
```

since we don't need yet the RV columns, and since for SQL Server 2005 we need to include (UPDLOCK) option in the RvTest view (see Appendix 2) and this would prohibit the use of READ UNCOMMITTED isolation. Especially for SQL Server we fine tune the RvTestList to include NOLOCK option as following

```
CREATE VIEW RVV.RvTestList
AS
SELECT id, s
FROM   RVV.VersionTest WITH (NOLOCK);
```

The following simple Java client demonstrates applying the RVV discipline in our use case using JDBC data access. Instead of n-Tier implementation we use a console application to keep the presentation of data access and database transactions as simple as possible. This will build us a baseline implementation to which we can compare the implementations of other programming paradigms in the Appendixes.

```
/*************************************************************
DBTechNet / Martti Laiho

A sample java program demonstrating data access client
using a single database connection and applying the RVV Discipline
in Type 1 update.

usage examples:

rem Oracle 10gR2 local ORCL
set CLASSPATH=.;C:\Oracle\product\10.2.0\client_1\jdbc\lib\classes12.jar
java RvvCase oracle.jdbc.driver.OracleDriver
"jdbc:oracle:thin:@localhost:1521:ORCL" user psw

rem DB2 V9 Express-C
set CLASSPATH=.;C:\IBM\SQLLIB\java\db2jcc.jar;
               C:\IBM\SQLLIB\java\db2jcc_license_cu.jar
java RvvCase com.ibm.db2.jcc.DB2Driver "jdbc:db2://localhost:50000/sample"
user psw

rem SQL Server 2005
set CLASSPATH=.;C:\Program Files\Microsoft SQL Server 2005 JDBC
Driver\sqljdbc_1.1\enu\sqljdbc.jar
java RvvCase com.microsoft.sqlserver.jdbc.SQLServerDriver
"jdbc:sqlserver://localhost;databaseName=TEST" user psw
*************************************************************/
```

```
import java.io.*;
import java.util.*;
import java.sql.*;

class RvvCase {
  static BufferedReader stdin =
    new BufferedReader(new InputStreamReader(System.in));

  public static void main(String[] args) throws SQLException {
    Connection myConnection = null;
    PreparedStatement myPreparedStatement;
    Statement myStatement;
    ResultSet myResultSet;
    String sqlCommand;
    String oldS, newS;
    long oldRv, newRv;
    long id = 0;

    loadDriver(args[0]);
    String user = args[2];
    String psw  = args[3];

    try {
      myConnection = DriverManager.getConnection(args[1],user,psw);

      System.out.println("\nRVV/JDBC Test <2.0>\nListing of the rows:");

      // *** Phase 2 - DATA ACCESS. ***
      myConnection.setAutoCommit(false);
      if (myConnection.getMetaData().supportsTransactionIsolationLevel(
                                  Connection.TRANSACTION_READ_UNCOMMITTED))
          myConnection.setTransactionIsolation(
                      Connection.TRANSACTION_READ_UNCOMMITTED);
      else
          myConnection.setTransactionIsolation(
                      Connection.TRANSACTION_READ_COMMITTED);

      myStatement = myConnection.createStatement ();
      sqlCommand = "SELECT id, s FROM rvv.RvTestList";
      myResultSet = myStatement.executeQuery (sqlCommand);

       // *** Phase 2/3 - Data Access / USER INTERFACE ***
      System.out.println("ID: \t S:");
      while (myResultSet.next()){
        System.out.println (myResultSet.getInt(1) + " \t" +
                            myResultSet.getString(2));
      }
      myResultSet.close();
      myStatement.close();
      myConnection.commit();
      id = readLong("Select a row by id : ");

      // *** Phase 4 - DATA ACCESS ***
      myConnection.setAutoCommit(false);
      myConnection.setTransactionIsolation(
                     Connection.TRANSACTION_READ_COMMITTED);
      sqlCommand = "SELECT s, rv FROM rvv.RvTest WHERE id = ?";
      myPreparedStatement = myConnection.prepareStatement (sqlCommand);
      myPreparedStatement.setLong(1, id);
      myResultSet = myPreparedStatement.executeQuery();
      if (myResultSet.next() == false) {
        throw new Exception("Unknown ID!");
      }
      oldS = myResultSet.getString(1);
      oldRv = myResultSet.getLong(2);
```

```
    myResultSet.close();
    myPreparedStatement.close();
    myConnection.commit();

    // *** Phase 5 - USER INTERFACE ***
    System.out.println("Found the row ");
    System.out.println("ID=" + id + " S=" + oldS + " RV=" + oldRv);
    System.out.print("Enter new value for column S: ");
    newS = stdin.readLine();

    // *** Phase 6 - UPDATE (Transaction) ***
    myConnection.setAutoCommit(false);
    // Type 1 update - so no need to set isolation level?
    sqlCommand =
        "UPDATE rvv.RvTest " +
           "SET s = ? " +
         "WHERE id = ? AND rv = ? ";
    myPreparedStatement = myConnection.prepareStatement(sqlCommand);
    myPreparedStatement.setString(1, newS);
    myPreparedStatement.setLong(2, id);
    myPreparedStatement.setLong(3, oldRv);

    int updated = myPreparedStatement.executeUpdate();
    if (updated != 1) {
      throw new Exception("Conflicting row version in the database! " );
    }
    myPreparedStatement.close();

    // Update succeeded -> The application needs to know the new RV
    sqlCommand = "SELECT rv FROM rvv.RvTest WHERE id = ?";
    myPreparedStatement = myConnection.prepareStatement(sqlCommand);
    myPreparedStatement.setLong(1, id);

    myResultSet = myPreparedStatement.executeQuery();
    if (myResultSet.next() == false) {
      throw new Exception("Read failure after update! - Impossible?");
    }

    newRv = myResultSet.getLong(1);

    myResultSet.close();
    myPreparedStatement.close();
    myConnection.commit();

    System.out.println("New RV is " + newRv);
  }
  catch (SQLException se) {
    System.out.println ("Sql exception: " + se);
  }
  catch (Exception e) {
    System.out.println("Exception: " + e);
  }
  finally {
    myConnection.close();
  }
}

// loadDriver
private static void loadDriver(String driverName) {
  try {
    Class.forName(driverName);
  }
  catch(java.lang.ClassNotFoundException e) {
    System.err.print("ClassNotFoundException: ");
    System.err.println(e.getMessage());
```

```
      System.exit(-1);  // exit due to driver problem
    }
  }

  // readLong
  private static long readLong(String prompt) {
    long value = 0;
    System.out.print(prompt);
    try {
      value = Long.parseLong(stdin.readLine());
    }
    catch (Exception ex) {
      System.out.println("Exception: " + ex);
    }
    return value;
  }
}
// End
```

In the following test runs we have used SQL Server 2005 server. First run is without competition:

```
java RvvCase com.microsoft.sqlserver.jdbc.SQLServerDriver
"jdbc:sqlserver://localhost;databaseName=TEST" user1 sql

RVV/JDBC Test <2.0>
Listing of the rows:
ID:       S:
1        some
2        some
Select a row by id : 1
Found the row
ID=1 S=some RV=228002
Enter new value for column S: new value
New RV is 228003
```

Then we test with concurrent transactions

```
BEGIN TRANSACTION
UPDATE RVV.RvTest
SET s='something'
WHERE id = 1

(1 row(s) affected)
```

```
java RvvCase com.microsoft.sqlserver.jdbc.SQLServerDriver
"jdbc:sqlserver://localhost;databaseName=TEST" user1 sql

RVV/JDBC Test <2.0>
Listing of the rows:
ID:       S:
1        something
2        some
Select a row by id : 1
```

```
COMMIT

Command(s) completed successfully.
```

```
Found the row
ID=1 S=something RV=228004
Enter new value for column S: test
```

```
BEGIN TRANSACTION
UPDATE RVV.RvTest
SET s='changed'
WHERE id = 1
COMMIT

(1 row(s) affected)
```

```
<ENTER>
Exception: java.lang.Exception: Conflicting row version in the
database!
```

Appendix 4 Sample ADO.NET programs using RVV Discipline

A4.1 Connected Data Access

The following simple C# client demonstrates applying the RVV discipline using ADO.NET data access. Instead of n-Tier implementation we use a console application to keep the presentation of data access and database transactions as simple as possible. Instead of a universal data access to services of different DBMS systems like ODBC and JDBC ADO.NET comes with varying data providers of which we demonstrate use of the native SQL Server data provider. We also demonstrate the use of ADO.NET transaction paradigm of .NET Framework 1.1 in which a local transaction is controlled by a separate transaction object and all statement objects accessing the database during the transaction need to be bound to the transaction object. This paradigm has changed in Framework 2 with introduction of TransactionScope object.

As DBMS we use now SQL Server 2005 which provides an interesting solution for reading the current rowversion value in Phase 6 by OUTPUT clause of Transact-SQL UPDATE command.

```
/* DBTechNet / Martti Laiho
 * A sample C# ADO.NET program demonstrating data access client
 * using a single database connection and applying the RVV Discipline
 * in Type 1 update.
 ************************************************************/
using System;
using System.Data;
using System.Data.SqlClient;

class SqlRvvCase {
```

```
/// <summary>
/// Simple ADO.NET Client/Server paradigm of RVV written in C#
/// using a single connection
/// </summary>
static void Main(string[] args) {
    string strConn = @"Data Source=(local)\MSSQLSERVER;" +
      "Initial Catalog=TEST; Trusted_Connection=Yes;";
    SqlConnection cn = new SqlConnection(strConn);

    try {
        Console.WriteLine(
               "RVV.NET Test <2.0>\nListing of the rows:");
        // ~ Phase 2  - data access
        cn.Open();
        SqlCommand cmd = cn.CreateCommand();
        SqlTransaction txn = cn.BeginTransaction(
                             IsolationLevel.ReadUncommitted );
        cmd.Transaction = txn;
        cmd.CommandText = "SELECT id, s FROM rvv.RvTest";
        SqlDataReader rdr = cmd.ExecuteReader();
        // Phase 2/3 - user interface
        Console.WriteLine("ID:\t S:");
        while (rdr.Read()) {
            Console.WriteLine("{0} \t{1}", rdr.GetInt32(0),
                                           rdr.GetString(1));
        }
        rdr.Close();
        txn.Commit ();
        Console.Write("Select a row by id : ");
        int id = Int32.Parse(Console.ReadLine());
        // Phase 4 - data acess
        txn = cn.BeginTransaction();
        cmd.Transaction = txn;
        cmd.CommandText =
            "SELECT s, rv FROM rvv.RvTest WHERE id = @id";
        cmd.Parameters.Add("@id", SqlDbType.Int, 5).Value = id;
        rdr = cmd.ExecuteReader();
        if (!rdr.Read()) {
            throw new Exception("Unknown ID!");
        }
        String oldS = rdr.GetString(0);
        long oldRv = rdr.GetInt64(1);
        rdr.Close();
        txn.Commit();
        // Phase 5 - user interface
        Console.WriteLine("Found the row ");
        Console.WriteLine("ID={0}, S={1}, RV={2}",
                                  id, oldS, oldRv);
        Console.Write("Enter new value for column S: ");
        string newS = Console.ReadLine();
        // Phase 6 - update transaction
        txn = cn.BeginTransaction();
        cmd.Transaction = txn;
        // Type 1 update:
        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;
        //int intRecordsAffected = cmd.ExecuteNonQuery();
        long newRv = 0L;
```

```
            try {
                newRv = (long) cmd.ExecuteScalar();
                txn.Commit();
            }
            catch (Exception e) {
                throw new Exception("Conflicting row version in database "
                                    +e.Message);
            }
            cmd.Dispose();
            Console.WriteLine("New RV is " + newRv);
        }
        catch (SqlException e) {
            Console.WriteLine("Sql error: " + e.Message);
        }
        catch (Exception e) {
            Console.WriteLine("Exception: " + e.Message);
        }
        finally {
            if (cn.State == ConnectionState.Open)
                cn.Close();
        }
        Console.Write("\nPress ENTER to exit ...");
        Console.ReadLine();
    }
}
```

In the following test runs we have used SQL Server 2005 server. First run is without competition:

```
J:\DBTechNet\Concurrency\SQLServer>SqlRvvCase
RVV.NET Test <2.0>
Listing of the rows:
ID:      S:
1       Something
2       new text
Select a row by id : 1
Found the row
ID=1, S=Something           , RV=223005
Enter new value for column S: new value
New RV is 223006

Press ENTER to exit ...
```

Then we apply a concurrent update as follows:

```
J:\DBTechNet\Concurrency\SQLServer>SqlRvvCase
RVV.NET Test <2.0>
Listing of the rows:
ID:      S:
1       new value
2       new text
Select a row by id : 1
Found the row
ID=1, S=new value           , RV=223006
Enter new value for column S: test
```

Concurrent session:

```
UPDATE RVV.RvTest
SET s='changed'
WHERE id = 1;

(1 row(s) affected)
```

```
<ENTER>
Exception: Conflicting row version in database Object reference not set to
an in
stance of an object.

Press ENTER to exit ...

J:\DBTechNet\Concurrency\SQLServer>
```

A4.2 Disconnected Data Access

To demonstrate is data access First we will build the table Table2 in a SQL Server database Test as follows:

```
CREATE TABLE Table2 (
id      INT NOT NULL PRIMARY KEY,
s       VARCHAR(20),       -- representing data columns ..
r       REAL DEFAULT 0.0,
rv      ROWVERSION
) ;
INSERT INTO Table2 VALUES (1,'first',1.0,1);
INSERT INTO Table2 VALUES (2,'second',1.0,1);
```

a) We will use only columns id and s in the following version of our sample C# program using SQL .NET Data Provider, which is using the `CommandBuilder` for generating the helper commands for synchronizing the `DataSet` data with the database. In case some row version validation fails a `DBConcurrencyException` is raised. For the database transactions we use the new transaction model of .NET Framework 2, which hides the complicated use of `Transaction` object of ADO.NET while using the new `TransactionScope` object.

```
/* DBTechNet / Martti Laiho
 * A Sample ADO.NET program demonstrating use of Disconnected
 * data processing in a DataSet and applying the RVV Discipline
 * in synchronizing the data into the database.
 */
using System;
using System.Data;
using System.Data.SqlClient;
using System.Transactions;  // need to be referenced manually to
// C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Transactions.dll
class DataSetSynchExample1 {
    public static void Main() {
        SqlConnection cn = new SqlConnection(
            @"Data Source=(Local)\MSSQLSERVER;" +
            "Integrated Security=SSPI; Initial Catalog=TEST");
        try {
         // Phase 4 transaction, default isolation level Read Committed
            SqlCommand cmd = cn.CreateCommand();
            cmd.CommandText = "SELECT id, s FROM Table2";
            SqlDataAdapter da = new SqlDataAdapter();
            da.SelectCommand = cmd;
            DataSet ds = new DataSet();
            cn.Open();
            using (TransactionScope ts = new TransactionScope()) {
                da.Fill(ds, "Table2");
            }
            cn.Close();
```

```
            // Phase 5 with simulated user thinking time ..
               Console.WriteLine("\nPress ENTER to continue ...");
               Console.ReadLine();
               // and some simulated data updates
               foreach (DataRow r in ds.Tables["Table2"].Rows) {
                   if (r["id"].Equals(1)) {
                       r.BeginEdit();
                       r["s"] = "new value";
                       r.EndEdit();
                   }
                   if (r["id"].Equals(3)) {
                       r.Delete();
                   }
               }
            // end of Phase 5
            // Phase 6, writing transaction synchronizing data with database
            // We use CommanBuilder to generate the Insert, Update and Delete
            // commands to be used by the DataAdapter da if it needs them:
               SqlCommandBuilder cb = new SqlCommandBuilder(da);
               da.InsertCommand = cb.GetInsertCommand();
               da.UpdateCommand = cb.GetUpdateCommand();
               da.DeleteCommand = cb.GetDeleteCommand();
               cn.Open();
            // and execute the transaction as follows
               using (TransactionScope ts = new TransactionScope()) {
                   da.Update(ds, "Table2");
               }
           }
           catch (SqlException ex){
               Console.WriteLine("\nError: {0}", ex.Message);
               // ...
           }
           catch (DBConcurrencyException ce) {
               // The exception that is thrown by the DataAdapter during
               // an insert, update, or delete operation if the number
               // of rows affected equals zero
               Console.WriteLine("\nError: {0}", ce.Message);
               // ...
           }
           finally {
               if (cn.State == ConnectionState.Open)
                   cn.Close();
           }
           Console.Write("\nPress ENTER to exit ...");
           Console.ReadLine();
       }
}
```

By running the program and tracing the execution run by the SQL Server Profiler we can see that the UPDATE command generated by `CommandBuilder` is the following:

```
exec sp_executesql N'UPDATE [Table2] SET [s] = @p1 WHERE (([id] = @p2)
AND ((@p3 = 1 AND [s] IS NULL) OR ([s] = @p4)))',N'@p1 varchar(9),@p2
int,@p3 int,@p4 varchar(5)',@p1='new value',@p2=1,@p3=0,@p4='first'
```

b) Even if we include the ROWVERSION type column `rv` in our DataSet, the `CommandBuilder` will not make use of its properties. If we want to make use of the `rv` column in row version validations we need to build the helper commands of the `DataAdapter` ourself. For that purpose we replace the program codes of phases 4 and 6 with the following program codes. The code of Phase 4 is replaced by the following code:

```
// Phase 4 transaction, default isolation level Read Committed
   SqlCommand cmd = cn.CreateCommand();
   cmd.CommandText =
       "SELECT id, s, CAST(rv AS BIGINT) rv FROM Table2";
   SqlDataAdapter da = new SqlDataAdapter();
   da.SelectCommand = cmd;
   DataSet ds = new DataSet();
   cn.Open();
   using (TransactionScope ts = new TransactionScope()) {
       da.Fill(ds, "Table2");
   }
```

CASTing the `rv` value as BIGINT data type will make it easier to handle the original row version value.

The code of Phase 6 is replaced by the following code in which we build the UpdateCommand manually as parameterized SQL UPDATE binding the parameter values from the table columns in the DataSet:

```
// Phase 6, writing transaction
   // building the parameterized UpdateCommand:
   SqlCommand ucmd = cn.CreateCommand();
   ucmd.CommandText = @"UPDATE Table2 SET s = @s " +
       "WHERE id = @id AND CAST(rv AS BIGINT) = @rv ";
   // binding the parameter values from the DataSet
   ucmd.Parameters.Add("@s", SqlDbType.NVarChar, 20, "s");
   ucmd.Parameters.Add("@id", SqlDbType.Int, 10, "id");
   ucmd.Parameters.Add("@rv", SqlDbType.Int, 10, "rv");
   da.UpdateCommand = ucmd;
   // and after building of the InsertCommand and DeleteCommand
   // accordingly we will synchronize data with the database
   cn.Open();
   using (TransactionScope ts = new TransactionScope()) {
       da.Update(ds, "Table2");
   }
```

Please note that we can use the default isolation level since there are no independent reading operations which would require read locks.

Appendix 5 RVV Implementation using J2EE™ BMP

J2EE™ architecture of Sun and the Java camp has evolved from various architectures for building component-based multitiered distributed applications merging various technologies as J2EE specifications and implementations of more or less compatible application servers called J2EE Containers. The business logic is composed of components called session beans, entity beans, and message-driven beans. The entity beans are speciality of J2EE as presenting the persisted data from database as object instances for the session beans and other application clients. The synchronization of the data with the database can be programmed in which case the bean class is called Bean Managed Persistence (BMP) bean, or the synchronization can be generated by tools of the Container in which case the bean class is called Container Managed Persistence (CMP) bean. We will first focus on implementing RVV discipline using BMP approach. For more details of the J2EE architecture we refer to java.sun.com/j2ee pages and J2EE literature, and we assume that the readers interested in our RVV implementation already have background information of the entity beans.

According to the J2EE EJB2 specification BMP bean has strict rules to follow to get its infrastructure services from the Container. This way the bean programmer does not need to worry about use of threads, object pooling, transaction programming etc, but "the programmer can focus on business issues". The transactionality of methods is defined outside the actual application code in the deployment descriptor XML file. Beside the business methods the programmer need to implement some ejb callback routines, which the Container calls in appropriate phases of the bean instance's life-cycle taken care by the Container.

Our `Rvtest` bean operates on `RvTest` view which presents the `id`,`s`,and `rv` columns to the bean where the `rv` is the technical column of row version. Since the bean's state of the persistent fields is saved transactionally in the database and instance is not capable to cache the `rv` values from transaction to transaction, we need to pass the `rv` values to the bean client. This introduces following additional restrictions to rules of the BMP bean

- `getRv` should be transactional and invoked as the first getter method
- all other getter methods need to be non-transactional
- instead of setter methods a single set type transactional business method should be used passing all changed field values and the original `rv` value to the bean.

Please note that cumulative i.e. Type 0 updates as defined in Chapter 1 cannot be applied in entity beans.

We have modified our `Rvtest` bean from the SavingsAccount BMP example of J2EE 1.4 Tutorial of Sun Microsystems. The remote interface `Rvtest.java` is of the form:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
```

```
public interface Rvtest extends EJBObject {
    public void updateS(String s, long Rv)
                throws RemoteException;
    public int getId() throws RemoteException;
    public String getS() throws RemoteException;
    public long getRv() throws RemoteException;
}
```

and the remote home interface `RvtestHome.java` is as follows

```
import javax.ejb.*;
import java.util.Collection;
import java.rmi.RemoteException;

public interface RvtestHome extends EJBHome {
    public Rvtest create(int id, String s, long rv)
        throws RemoteException, CreateException;
    public Rvtest findByPrimaryKey(RvtestPK key)
        throws FinderException, RemoteException;
    public Collection findAll()
        throws FinderException, RemoteException;
}
```

For the primary key management we include the class

```
import java.io.Serializable;
public class RvtestPK implements java.io.Serializable {
  public int rvtestID;
  public RvtestPK (int id) {
    this.rvtestID = id;
  }
  public RvtestPK () {
  }
  public String toString() {
     return ""+ rvtestID;
  }
}
```

The code of `RvtestBean` looks very much like the SavingsAccountBean code entity bean except the update method `updateS` and the database routines invoked by the ejb callback methods

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;

public class RvtestBean implements EntityBean {

    private EntityContext context;
    // JNDI lookup string for the database
    private static final String dbName =
            "java:comp/env/jdbc/ejbRvvDB";
    // Database connection object handle
    private Connection con;
```

```
// state fields of bean instances
private int rvtestID;  // PK
private String s;
private long rv;
// Bean constructor
public RvtestBean() {
}
// Business logic methods
public void updateS(String sNew, long rvOld) throws Exception {
   if (this.rv == rvOld) {  // applying RVV Type 2
       this.s = sNew;
   }
   else
       throw new Exception("Row version too old");
}
// getter [and setter] methods
public int getId() {
        return rvtestID;
  }
  public long getRv() {    // tansactional? First to be invoked !
   return rv;
}

public String getS() {   // not transactional!
   return s;
}

// EJB finder methods
public RvtestPK ejbFindByPrimaryKey(RvtestPK primaryKey)
  throws FinderException, RemoteException {
  boolean result;
  try {
     result = selectByPrimaryKey(primaryKey.rvtestID);
   } catch (Exception ex) {
       throw new EJBException("ejbFindByPrimaryKey: " +
          ex.getMessage());
   }
  if (result) {
     return primaryKey;
  }
  else {
     throw new ObjectNotFoundException
        ("Row for id " + primaryKey + " not found.");
  }
}

public Collection ejbFindAll()
  throws FinderException, RemoteException {
  Collection result;
  try {
       result = selectAll();
   } catch (Exception ex) {
       throw new EJBException("ejbFindAll: " + ex.getMessage());
   }
  return result;
}

// EJB callback methods accessed by the container
```

```
public RvtestPK ejbCreate(int id, String s)
   throws CreateException, RemoteException {
   long rv =0L;
   try {
       rv = insertRow(id, s);  // we get rowversion from the DBMS
       this.s = s;
       this.rv = rv;
   } catch (Exception ex) {
       throw new EJBException("ejbCreate: " + ex.getMessage());
   }
   this.rvtestID = id;
   return new RvtestPK (id);
}

public void ejbRemove()  throws RemoteException {
   try {
       deleteRow(this.rvtestID);
   } catch (Exception ex) {
       throw new EJBException("ejbRemove: " + ex.getMessage());
   }
}

public void ejbActivate()  throws RemoteException {
}

public void ejbPassivate() throws RemoteException {
}

public void ejbLoad()  throws RemoteException {
  try {
     RvtestPK pk = (RvtestPK) context.getPrimaryKey();
     int id = pk.rvtestID;
     loadRow(id);
     this.rvtestID = id;
   } catch (Exception ex) {
       throw new EJBException("ejbLoad: " + ex.getMessage());
   }
}

public void ejbStore()  throws RemoteException {
   try {
       storeRow();
   } catch (Exception ex) {
       throw new EJBException("ejbStore: " + ex.getMessage());
   }
}

public void ejbPostCreate(int id, String s)
            throws RemoteException {
}

public void setEntityContext(EntityContext context)
            throws RemoteException {
  this.context = context;
  try {
      makeConnection();
  } catch (Exception ex) {
      throw new EJBException("Unable to connect to database. " +
         ex.getMessage());
```

```
      }
    }

    public void unsetEntityContext() throws RemoteException {
      try {
          con.close();
      } catch (SQLException ex) {
          throw new EJBException("unsetEntityContext: " +
ex.getMessage());
      }
    }
/********************** Database Routines ***********************/
private void makeConnection() {
      try {
          InitialContext ic = new InitialContext();
          DataSource ds = (DataSource) ic.lookup(dbName);
          con = ds.getConnection();
      } catch (Exception ex) {
          throw new EJBException("Unable to connect to database. " +
                                 ex.getMessage());
      }
    }

    private void releaseConnection() {
      try {
          con.close();
      } catch (SQLException ex) {
          throw new EJBException("releaseConnection: " +
                                 ex.getMessage());
      }
    }

    private long insertRow (int id, String s) throws SQLException {
       long rv = 0L;
       try {
          makeConnection();
          String insertStatement =
                "INSERT INTO RvTest VALUES ( ? , ? )";
          PreparedStatement prepStmt =
                con.prepareStatement(insertStatement);
          prepStmt.setInt(1, id);
          prepStmt.setString(2, s);
          prepStmt.executeUpdate();
          prepStmt.close();
          // picking the rv of the inserted row
          String selectStatement =
                "SELECT rv FROM RvTest WHERE id = ? ";
          prepStmt =
                con.prepareStatement(insertStatement);
          prepStmt.setInt(1, id);
          ResultSet rs = prepStmt.executeQuery();
          if (rs.next()) {
            rv = rs.getLong(1);
            prepStmt.close();
          }
          else
            throw new EJBException("Select rv of the INSERTed " +
                                   id + " failed.");
        }
```

```
     finally {
       releaseConnection();
     }
     return rv;
  }

  private void deleteRow(int id) throws SQLException {
     try {
        makeConnection();
        String deleteStatement =   // Type 1
              "DELETE FROM RvTest WHERE id = ? AND rv = ?";
        PreparedStatement prepStmt =
              con.prepareStatement(deleteStatement);
        prepStmt.setInt(1, id);
        prepStmt.setLong(2, rv);
        int rowCount = prepStmt.executeUpdate();
        prepStmt.close();
        releaseConnection();
        if (rowCount == 0) {
           throw new EJBException("Deleting row for id " +
                                  id + " failed.");
        }
     }
     finally {
         releaseConnection();
     }
  }

  private boolean selectByPrimaryKey(int primaryKey)
                 throws SQLException {
    makeConnection();
    String selectStatement =
          "SELECT id " +
          "FROM RvTest WHERE id = ? ";
    PreparedStatement prepStmt =
          con.prepareStatement(selectStatement);
    prepStmt.setInt(1, primaryKey);
    ResultSet rs = prepStmt.executeQuery();
    boolean result = rs.next();
    prepStmt.close();
    releaseConnection();
    return result;
  }

  private void loadRow(int id) throws SQLException {
     try {
        makeConnection();
        String selectStatement =
              "SELECT s, rv " +
              "FROM RvTest WHERE id = ? ";
        PreparedStatement prepStmt =
              con.prepareStatement(selectStatement);
        prepStmt.setInt(1, id);
        ResultSet rs = prepStmt.executeQuery();
        if (rs.next()) {
           this.rvtestID = id;
           this.s = rs.getString(1);
           this.rv = rs.getLong(2);
           prepStmt.close();
```

```
        }
        else {
           prepStmt.close();
           throw new NoSuchEntityException("Row for id " + id +
              " not found in database.");
        }
     }
     finally {
         releaseConnection();
     }
  }

  private Collection selectAll() throws SQLException {
    makeConnection();
    String selectStatement =
          "SELECT id FROM RvTest ";
    Statement stmt =
          con.createStatement();
    ResultSet rs = stmt.executeQuery(selectStatement);
    ArrayList a = new ArrayList();
    while (rs.next()) {
        int id = rs.getInt(1);
        a.add(new RvtestPK(id));
    }
    stmt.close();
    releaseConnection();
    return a;
  }

  private void storeRow() throws SQLException {
    try {
        makeConnection();
        // RVV Type 1 - to guarantee the Type 2 used above
        String updateStatement =
              "UPDATE RvTest SET s =  ? " +
              "WHERE id = ? AND rv = ?";
        PreparedStatement prepStmt =
              con.prepareStatement(updateStatement);

        prepStmt.setString(1, s);
        prepStmt.setInt(2, this.rvtestID);
        prepStmt.setLong(3, rv);
        int rowCount = prepStmt.executeUpdate();
        prepStmt.close();
        if (rowCount == 0) {
           throw new EJBException("Storing row for id " +
                                  this.rvtestID + " failed.");
        }
     }
     finally {
         releaseConnection();
     }
  }
```

The following RvtestClient.java code simulates the presentation layer of our use case as follows

```
import java.io.*;
```

```
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class RvtestClient {
   public static void main(String[] args) {
       BufferedReader stdin = new BufferedReader(
                              new InputStreamReader(System.in));
       String s = "";
       int id = 0;
       long rv = 0L;
       try {
        // Phase 1 - User interface
           System.out.println("J2EE/RVV Test <1.0>");
           Context initial = new InitialContext();
           Object objref =
               initial.lookup("java:comp/env/ejb/RvTest");
           RvtestHome home = (RvtestHome)
               PortableRemoteObject.narrow(objref,RvtestHome.class);
        // Phase 2 - request for the entity list
           Collection col = home.findAll();
        // Phase 3 - User interface
           System.out.println("Listing of the rows \nID:\tS:" );
           Iterator i=col.iterator();
           while (i.hasNext()) {
              Rvtest re = (Rvtest)i.next();
              id = re.getId();
              s  = re.getS();
              System.out.println(id + ":\t" + s);
           }
           System.out.println("Select row by ID: ");
           try {
                id = Integer.parseInt(stdin.readLine());
                System.out.println("value: " + id);
           }
           catch (Exception ex) {
                System.out.println("Exception: " + ex);
           }
        // Phase 4 - request for the entity
           RvtestPK pk = new RvtestPK();
           pk.rvtestID = id;
           Rvtest re1 = home.findByPrimaryKey(pk);
        // Phase 5 - User interface
           rv = re1.getRv();
           System.out.println( "ID= " + id +
                               "\nS = " + re1.getS() +
                               "\nRV= " + rv );
           System.out.println("Enter new value for S: ");
           try {
              s = stdin.readLine();
              System.out.println("value: " + s);
           }
           catch (Exception ex) {
              System.out.println("Exception: " + ex);
           }
        // Phase 6 - update and save the entity
           re1.updateS (s, rv);
           System.exit(0);
```

```
        } catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
    }
}
```

Running the client without competition

```
appclient -client RvvTest_BMPClient.jar

J2EE/RVV Test <1.0>
Listing of the rows
ID:     S:
10:     some
20:     thing
30:     else
Select row by ID:
10
value: 10
ID= 10
S = some
RV= 0
Enter new value for S:
test1
value: test1
```

Running the client with competition as follows

```
appclient -client RvvTest_BMPClient.jar
J2EE/RVV Test <1.0>
Listing of the rows
ID:     S:
10:     test1
20:     thing
30:     else
Select row by ID:
10
value: 10
ID= 10
S = test1
RV= 0
Enter new value for S:
```

```
and in concurrent SQL session:
UPDATE rvtest SET s='new' WHERE id =10;
SELECT * FROM RVTEST;
COMMIT;
```

```
test2
value: test2
Caught an exception.
java.rmi.ServerException: RemoteException occurred in server thread;
nested exception is:
        java.rmi.RemoteException: ; nested exception is:
        java.lang.Exception: Row version too old
        ...
```

which proves that we can use RVV discipline in J2EE BMP bean, but considering length of the code lines raises the question if this is the way to go.

On RVV Implementation using J2EE™ CMP

The bean content synchronization with the database of a CMP bean is generated by the tools of the Container. Rules concerning the form of a CMP bean are even more strict than the rules of BMP. According to EJB2 specification the bean class is abstract and the getter and setter methods shall be defined as abstract, which all means that we cannot implement the RVV discipline in application code of CMP like we can in BMP. For avoiding Blind Overwritings we can only apply the options provided by the persistence manager of the Container.

Appendix 6 Programmed RVV for Hibernate Core

Documentation of both TopLink and Hibernate advertise the use of optimistic locking, even if the main targets of these middleware solutions seem to be ORM and caching services. However, so called "2nd level caching" beyond the bufferbool caching of the used DBMS is in contradiction with the services needed by RVV. As default the old row version is fetched from the local cache, so it itself is stale data and of no use for optimistic locking. Bypassing the cache services turns out to be a tricky task.

In our use case scenario the phase 6 needs isolation level REPEATABLE READ as minimum. This is available in DB2 and SQL Server, but not in Oracle, which for our purposes can only provide the snapshot isolation, calling it SERIALIZABLE. Taking this as a challenge and wanting to prove that ORA_ROWSCN can be used as row version field managed at server-side, we started experimenting with the Oracle table VERSIONTEST and view RVTEST of our article above, and we finally managed to sort out the following programmed Hibernate solution, without the Hibernate optimistic locking services:

We defined access via the RVTEST view using the following Hibernate mapping file:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
       <class name="rvvtest.RvvEntity" table="RVTEST">
         <id name="id" column="ID"/>
         <property name="s" column="S" update="true"/>
         <property name="rv" column="RV" update="false"/>
       </class>
```

```
</hibernate-mapping>
```

Since column RV is actually the ORA_ROWSCN pseudo column, we don't allow it be updated by Hibernate. Our entity conforms of the following POJO (annotations would not have any influence here):

```
/*  DBTechNet / M Laiho 2007-12-30
*
*   Entity for the RVV example using Hibernate Core
*
*******************************************************/
package rvvtest;

public class RvvEntity {
    private Long id;
    private String s;
    private long rv;
    public RvvEntity() {
        super();
    }
    public Long getId() {
        return (Long)this.id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getS() {
        return this.s;
    }
    public void setS(String s) {
        this.s = s;
    }
    public Long getRv() {
        return this.rv;
    }
    // since RV is server-side pseudo column we don't need this
    public void setRv(Long rv) {  // but to keep Hibernate happy
        this.rv = rv;
    }
}
```

In our experiment we are focusing only on the data access issues and therefore we have implemented the presentation level simulation, application logic and transactions as the following simplified Java client:

```
/*  DBTechNet / M Laiho 2007-12-30
*
*   Java client of the simplified RVV example
*   using programmatic Hibernate optimistic locking
*
*******************************************************/
package rvvtest;

import java.io.*;
import java.util.*;
import java.sql.*;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;

public class RvvTest {

    public static void main(String[] args) {
```

```
BufferedReader stdin = new BufferedReader(
                       new InputStreamReader(System.in));
String s = "";
Long id = 0L;

// Configuration and opening session
SessionFactory sessionFactory;
sessionFactory = new
       Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();

// Phase 1 - User interface
System.out.print("RVV Test <2.0>\n Listing of the rows: - " );

// Phase 2 - "model"
Transaction tx = session.beginTransaction();
//tx.begin();
List entities =
    session.createQuery(
           "SELECT e FROM RvvEntity e ORDER BY e.id ASC")
           .list();
tx.commit();  // to keep the transaction short

// Phase 3 - User interface
System.out.println( entities.size() +
                  " entities found.\nID:\tS:\n" );
for (Object o : entities) {
    RvvEntity re = (RvvEntity) o;
    System.out.println(re.getId() + "\t" + re.getS());
}
System.out.println("Select row by ID: ");
try {
    id = Long.parseLong(stdin.readLine());
    System.out.println("value: " + id);
}
catch (Exception ex) {
    System.out.println("Exception: " + ex);
}

// Phase 4 - "model"
Long oldRv = 0L;
tx = session.beginTransaction();
RvvEntity re1 = (RvvEntity)
                session.load( RvvEntity.class, id);
tx.commit();
oldRv = (Long) re1.getRv();

// Phase 5 - User interface
System.out.println( "ID= " + re1.getId() +
                  "\nS = " + re1.getS() +
                  "\nRV= " + re1.getRv() );
System.out.println("Enter new value for column S: ");
try {
    s = stdin.readLine();
    System.out.println("value: " + s);
}
catch (Exception ex) {
    System.out.println("Exception: " + ex);
}

// Phase 6 - "model"
try {
    tx = session.beginTransaction();
    Connection conn = session.connection(); // 1) => JDBC
    conn.setTransactionIsolation(
```

```
                    conn.TRANSACTION_SERIALIZABLE); // 2)
            RvvEntity re2 = (RvvEntity)
                session.load( RvvEntity.class, id);
            session.refresh(re2);
            Long newRv = (Long)re2.getRv();
            if (oldRv.equals(newRv)) { // see Type 2
                re2.setS(s);
                session.save(re2);
                {/* 3) Programmed breakpoint for concurrency testing: */
                  System.out.println("Time for concurrency test. "+
                                     "Press ENTER to continue .. ");
                  try { s = stdin.readLine();
                  } catch (Exception ex) {
                        System.out.println("Exception: " + ex);
                  } /****************************************/
                }
                tx.commit();
            } else
                throw new Exception("StaleObjectState \n" +
                                "oldRv=" + oldRv + " newRv=" + newRv);
            System.out.println("persisted S = " + re2.getS() +
                               "\n oldRv=" + oldRv + " newRv=" + newRv);
        }
        catch (Exception ex) {
            System.out.println("Exception: " + ex);
        }
        // Shutting down the application
        finally {
            session.close();
        }
    }
}
```

Transaction of Phase 6 needs special tuning and testing for RVV. In the following we explain the marked places of its code:
1) The default isolation level READ COMMITTED suits in other phases of our use case, but it would lead to Blind Overwriting of concurrent transactions during Phase 6. Hibernate does not offer possibilities to change isolation level dynamically, so we need to switch first to the level of JDBC services.
2) REPEATABLE READ would be proper isolation level for Phase 6, but Oracle requires SERIALIZABLE, and Hibernate's Oracle dialect adapter does not transform REPEATABLE READ into SERIALIZABLE, so to keep the code portable we stick to SERIALIZABLE.
3) We have programmed this breakpoint to allow time for testing concurrent updates just before our transaction commits.

In the following test runs (using asant version of ant) we first test a simple update and then repeat the test twice running a competing SQL-session at two different steps of Phase 6 just before hitting ENTER key as marked below. The test runs prove that use of ORA_ROWSCN as row version field and programming logic provide reliable RVV implementation:

```
C:\hibernate-3.2\RvvHibernateCore>asant run
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file to C:\hibernate-3.2\RvvHibernateCore\build

copymetafiles:
```

```
run:
     [java] HBN/RVV Test <2.0>
     [java]  Listing of the rows: - 2 entities found.
     [java] ID: S:
     [java] 1   some text
     [java] 2   some text
     [java] Select row by ID:
1
     [java] value: 1
     [java] ID= 1
     [java] S = some text
     [java] RV= 2300127
     [java] Enter new value for column S:
new text
     [java] value: new text
     [java] Time for concurrency test. Press ENTER to continue ..

     [java] persisted S = new text
     [java]  oldRv=2300127 newRv=2300127

BUILD SUCCESSFUL
Total time: 25 seconds
```

This test run without competing transactions managed to update column s on the selected row.

```
C:\hibernate-3.2\RvvHibernateCore>asant run
Buildfile: build.xml

compile:

copymetafiles:

run:
     [java] HBN/RVV Test <2.0>
     [java]  Listing of the rows: - 2 entities found.
     [java] ID: S:
     [java] 1   new text
     [java] 2   some text
     [java] Select row by ID:
1
     [java] value: 1
     [java] ID= 1
     [java] S = new text
     [java] RV= 2300182
     [java] Enter new value for column S:
test
```

```
-- concurrent test for row 1 using SQL session
UPDATE RvTest SET S='new value' WHERE ID=1;
COMMIT;
SELECT * FROM RvTest;
ID                   S                    RV
-------------------- -------------------- ----------------------
1                    new value            2300210
2                    some text            2300127
```

```
<hitting of RETURN key>
     [java] value: test
     [java] Exception: java.lang.Exception: StaleObjectState
     [java]  oldRv=2300182 newRv=2300210

BUILD SUCCESSFUL
Total time: 36 seconds
```

This test run detected the new row version of the concurrent transaction committed before Phase 6 which failed raising exception with text "StaleObjectState".

```
C:\hibernate-3.2\RvvHibernateCore>asant run
Buildfile: build.xml
```

```
compile:

copymetafiles:

run:
     [java] HBN/RVV Test <2.0>
     [java]  Listing of the rows: - 2 entities found.
     [java] ID: S:
     [java] 1   new value
     [java] 2   some text
     [java] Select row by ID:
1
     [java] value: 1
     [java] ID= 1
     [java] S = new value
     [java] RV= 2300210
     [java] Enter new value for column S:
test value
     [java] value: test value
     [java] Time for concurrency test. Press ENTER to continue ..
```

```
-- concurrent test for row 1 using SQL session
UPDATE RvTest SET S='new value' WHERE ID=1;
COMMIT;
SELECT * FROM RvTest;
ID                   S                    RV
-------------------- -------------------- ----------------------
1                    new value            2300210
2                    some text            2300127
```

```
<hitting of RETURN key>
     [java] 22:18:53,484  WARN JDBCExceptionReporter:71 - SQL Error: 8177, SQLSt
ate: 72000
     [java] 22:18:53,484 ERROR JDBCExceptionReporter:72 - ORA-08177: can't seria
lize access for this transaction
     [java] 22:18:53,500 ERROR AbstractFlushingEventListener:301 - Could not syn
chronize database state with session
     [java] org.hibernate.exception.GenericJDBCException: Could not execute JDBC
 batch update
...
     [java]     at org.hibernate.jdbc.AbstractBatcher.executeBatch(AbstractBatch
er.java:242)
     [java]     ... 8 more
     [java] Exception: org.hibernate.exception.GenericJDBCException: Could not e
xecute JDBC batch update

BUILD SUCCESSFUL
Total time: 38 seconds
C:\hibernate-3.2\RvvHibernateCore>
```

This test run failed in Phase 6 due to the selected isolation level SERIALIZABLE on serialization error detected by Oracle DBMS because the concurrent transaction committed just before commit of Phase 6 transaction. So this works fine preventing Phase 6 on writing over the update of the competing transaction. The program code should be improved to skip the unnecessary Java stack trace and the JPA exception message should catch the actual ORA-8177 (SQLState 72000) error instead of the generic JDBC exception.

We have run the same test with DB2 V9.1 and SQL Server 2005. SQL Server using the locking protocol with U-lock fails in deadlock at the same code place where Oracle fails above (compare with tests in plain SQL in Appendix 2). For persistence operations Hibernate maintains a queue of commands, optimizing the order of commands and finally applies the commands to synchronize the data with database

just before committing the transaction (according to Bauer & King this is called transactional write-behind). So both Oracle and SQL Server combined with the Hibernate build a persistence engine providing for Phase 6 a kind of optimistic concurrency service protecting application against writing over the update of the competing transaction (the winner). For SQL Server Hibernate should apply the (UPDLOCK) table hint, which we tested in Appendix 2. In program code this would make the code DBMS dependent, but the table hint can also be supplied in the Transact-SQL view. Hibernate and DB2 correctly blocks the competing transaction and manages to update the selected row.
- It is really interesting to see how differently these three mainstream DBMS systems serve the same application code.

Experienced Hibernate and Oracle users might have considered using "SELECT .. FOR UPDATE" row locking using Hibernate's LockMode UPGRADE locking in phase 6 instead of SERIALIZABLE isolation level, but we noticed that this would have cleared the ORA_ROWSCN values for the transaction, so our solution may provide the best total performance for Oracle. Disadvantages of the use of ORA_ROWSCN are that

- Phase 6 transaction will lose competition to concurrent updates
- Update clears the value of ORA_ROWSCN, so it is not possible to read the value back to application in the same transaction.

These are a subject for further studies.

Appendix 7 Programmed RVV for Hibernate EntityManager (JPA)

We have applied our example also to Hibernate implementation of the EJB3 Java Persistence API (JPA), which is built on Hibernate Core. The programming paradigm is very different as you can see from the example below.

JPA specification requires following kind of persistent.xml file configuring the database connection etc to be stored in META-INF folder:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">
   <persistence-unit name="rvvtest" transaction-type="RESOURCE_LOCAL">
      <properties>
         <!-- Scan for annotated classes and Hibernate mapping XML files -->
          <property name="hibernate.archive.autodetection"
                    value="class, hbm"/>
          <!-- SQL stdout logging -->
          <property name="hibernate.show_sql" value="true"/>
          <property name="hibernate.format_sql" value="true"/>
          <property name="use_sql_comments" value="true"/>
         <!-- Oracle -->
          <property name="hibernate.connection.driver_class"
                    value="oracle.jdbc.driver.OracleDriver"/>
          <property name="hibernate.connection.url"
                    value="jdbc:oracle:thin:@oracle11g:1521:ORCL"/>
          <property name="hibernate.connection.username"
                    value="RVV"/>
```

```
            <property name="hibernate.connection.password"
                      value="test"/>
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.OracleDialect"/>
           <!-- DB2 Express-C V9.x
            <property name="hibernate.connection.driver_class"
                      value="com.ibm.db2.jcc.DB2Driver"/>
            <property name="hibernate.connection.url"
                      value="jdbc:db2://DB2xecV9:50000/TEST"/>
            <property name="hibernate.connection.username"
                      value="RVV"/>
            <property name="hibernate.connection.password"
                      value="test"/>
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.DB2Dialect"/>
            -->
           <!-- SQL Server 2005
            <property name="hibernate.connection.driver_class"
                      value="com.microsoft.sqlserver.jdbc.SQLServerDriver"/>
            <property name="hibernate.connection.url"
                      value="jdbc:sqlserver://sql2005;databaseName=TEST"/>
            <property name="hibernate.connection.username"
                      value="RVV"/>
            <property name="hibernate.connection.password"
                      value="test"/>
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.SQLServerDialect"/>
            -->
            <!-- JDBC connection pool (use the built-in) -->
            <property name="connection.pool_size" value="2"/>
        </properties>
    </persistence-unit>
</persistence>
```

Instead of XML configuration the POJO entity of our previous example has now annotations configuring the use of our database table (actually the view using the ORA_ROWSCN as the version field) as follows:

```
/*  DBTechNet / M Laiho 2007-12-30
*
*   Entity for the RVV example using Hibernate JPA
*
*   modified from the HelloWorld example in the book
*   "JAVA PERSISTENCE with HIBERNATE" by Bauer and King
*
*****************************************************/
package rvvtest;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.Column;
// import javax.persistence.Version;
@Entity
@Table (name="RVTEST")  // Table: VERSIONTEST, View: RVTEST
public class RvvEntity {
    @Id
    @Column(name="ID")
    private Long id;
    @Column(name="S")
    private String s;
  //@Version
  // cannot use this for column maintained at server-side!
    @Column(name="RV", updatable=false)
    private long rv;
```

```
        public RvvEntity() {
            super();
        }
        public Long getId() {
            return (Long)this.id;
        }
        public void setId(Long id) {
            this.id = id;
        }
        public String getS() {
            return this.s;
        }
        public void setS(String s) {
            this.s = s;
        }
        public Long getRv() {
            return this.rv;
        }
    }
```

Our test program looks now a bit different from the Hibernate Core example, as follows:

```
/*  DBTechNet / M Laiho 2007-12-30
*
*   Simplified Java client for the RVV example using Hibernate JPA
*
*******************************************************/
package rvvtest;

import java.io.*;
import java.util.*;
import java.sql.*;
import javax.persistence.*;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class RvvTest {

    public static void main(String[] args) {
        BufferedReader stdin = new BufferedReader(
                               new InputStreamReader(System.in));
        String s = "";
        Long id = 0L;

        // Start EntityManagerFactory
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("rvvtest");

        // Phase 1 - User interface
        System.out.print("JPA/RVV Test <2.0> \nListing of the rows: - " );

        // Phase 2 - "model"
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        List entities =
            em.createQuery("SELECT e FROM RvvEntity e ORDER BY e.id ASC")
              .getResultList();
        tx.commit();

        // Phase 3 - User interface
        System.out.println( entities.size() + " entities found.\nID:\tS:" );
        for (Object o : entities) {
```

```
    RvvEntity re = (RvvEntity) o;
    System.out.println(re.getId() + "\t" + re.getS());
}
System.out.println("Select row by ID: ");
try {
    id = Long.parseLong(stdin.readLine());
    System.out.println("value: " + id);
}
catch (Exception ex) {
    System.out.println("Exception: " + ex);
}

// Phase 4 - "model"
tx.begin();
RvvEntity re1 = em.find( RvvEntity.class, id);
tx.commit();
Long oldRv = (Long) re1.getRv();

// Phase 5 - User interface
System.out.println( "ID= " + re1.getId() +
                  "\nS = " + re1.getS() +
                  "\nRV= " + re1.getRv() );
System.out.println("Enter new value for S: ");
try {
    s = stdin.readLine();
    System.out.println("value: " + s);
}
catch (Exception ex) {
    System.out.println("Exception: " + ex);
}

// Phase 6 - "model"
em.clear();  // 1) clear cache
try {
    /*********************************************
    *  2) To get transaction of Phase 6 reliable we need
    *  REPEATABLE READ or SERIALIZABLE isolation level !
    *********************************************/
    Session session = (Session)em.getDelegate(); // 3) JPA  => Core
    Connection conn = session.connection();
    Transaction tx6 = session.beginTransaction();
    conn.setTransactionIsolation(
        conn.TRANSACTION_SERIALIZABLE); // 4)

    RvvEntity re2 = em.find(RvvEntity.class, id);
    Long newRv = (Long)re2.getRv();
    if (oldRv.equals(newRv)) {  // see Type 2
        re2.setS(s);
        em.persist(re2);
        { /*** 5) Programmed breakpoint for concurrency testing: */
          System.out.println("Time for concurrency test. "+
                             "Press ENTER to continue .. ");
          try { s = stdin.readLine();
          } catch (Exception ex) {
                System.out.println("Exception: " + ex);
          } /*****************************************/
        }
        tx6.commit();
    } else
        throw new Exception("StaleObjectState \n oldRv=" + oldRv +
                            " newRv=" + newRv);
    // just for testing:
    System.out.println("persisted S = " + re2.getS() +
                       " oldRv=" + oldRv + " newRv=" + newRv);
}
```

```
        catch (Exception ex) {
            System.out.println("Phase 6, catched exception: " + ex);
        }
        // Shutting down the application
        finally {
            em.close();
            emf.close();
        }
    }
}
```

The code above is almost portable JPA code, but current JPA specification is not enough for our RVV implementation, and we need some Hibernate Core functionalities commented on Phase 6 as follows:
1) We need to read the current row version directly from the database. For the time being we have not yet found effective JPA solution for this, so we just temporarily clear the cache.
2) The default isolation level READ COMMITTED suits in other phases of our use case, but it would lead to Blind Overwriting of concurrent transactions during Phase 6.
3) JPA does not offer possibilities to change isolation level dynamically, so we need to switch to the Hibernate Core services using the getDelegate() method of EntityManager and then get down to JDBC services.
4) Just like in the Hibernate Core example above to keep the code portable in terms of the used DBMS we tune the isolation level into SERIALIZABLE.
5) We have programmed this breakpoint to allow time for testing concurrent updates just before our transaction commits.

The following test runs we first test a simple update and then repeat the test twice running a competing SQL-session at two different steps of Phase 6 just like in the Hibernate Core test runs above:

```
C:\hibernate-3.2\RvvHibernateJPA>asant run
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file to C:\hibernate-3.2\RvvHibernateJPA\build

copymetafiles:

run:
     [java] JPA/RVV Test <2.0>
     [java] Listing of the rows: - 2 entities found.
     [java] ID: S:
     [java] 1   some text
     [java] 2   some text
     [java] Select row by ID:
1
     [java] value: 1
     [java] ID= 1
     [java] S = some text
     [java] RV= 2220384
     [java] Enter new value for S:
new text
     [java] value: new text
     [java] Time for concurrency test. Press ENTER to continue ..

     [java] persisted S = new text oldRv=2220384 newRv=2220384

BUILD SUCCESSFUL
```

```
Total time: 27 seconds
C:\hibernate-3.2\RvvHibernateJPA>

C:\hibernate-3.2\RvvHibernateJPA>asant run
Buildfile: build.xml

compile:

copymetafiles:

run:
     [java] JPA/RVV Test <2.0>
     [java] Listing of the rows: - 2 entities found.
     [java] ID: S:
     [java] 1   new text
     [java] 2   some text
     [java] Select row by ID:
1
     [java] value: 1
     [java] ID= 1
     [java] S = new text
     [java] RV= 2220521
     [java] Enter new value for S:
second text
```

```
-- concurrent test for row 1 using SQL session
UPDATE RvTest SET S='changed' WHERE ID=1;
COMMIT;
SELECT * FROM RvTest;
ID                   S                    RV
-------------------- -------------------- ----------------------
1                    changed              2220745
2                    some text            2220384
```

```
<hitting of RETURN key>
     [java] value: second text
     [java] Phase 6, catched exception: java.lang.Exception: StaleObjectState
     [java]  oldRv=2220521 newRv=2220745

BUILD SUCCESSFUL
Total time: 42 seconds
C:\hibernate-3.2\RvvHibernateJPA>
C:\hibernate-3.2\RvvHibernateJPA>asant run
Buildfile: build.xml

compile:

copymetafiles:

run:
     [java] JPA/RVV Test <2.0>
     [java] Listing of the rows: - 2 entities found.
     [java] ID: S:
     [java] 1   new value
     [java] 2   some text
     [java] Select row by ID:
1
     [java] value: 1
     [java] ID= 1
     [java] S = new value
     [java] RV= 2220745
     [java] Enter new value for S:
third text
     [java] value: third text
     [java] Time for concurrency test. Press ENTER to continue ..
```

```
-- concurrent test for row 1 using SQL session
UPDATE RvTest SET S='new value' WHERE ID=1;
COMMIT;
SELECT * FROM RvTest;
ID                   S                    RV
-------------------- -------------------- ----------------------
1                    new value            2221443
2                    some text            2220384
```

```
<hitting of RETURN key>

     [java] 10:36:38,750  WARN JDBCExceptionReporter:71 - SQL Error: 8177, SQLSt
ate: 72000
     [java] 10:36:38,750 ERROR JDBCExceptionReporter:72 - ORA-08177: can't seria
lize access for this transaction

     [java] 10:36:38,765 ERROR AbstractFlushingEventListener:301 - Could not syn
chronize database state with session
     [java] org.hibernate.exception.GenericJDBCException: could not update: [rvv
test.RvvEntity#1]
     [java]     at org.hibernate.exception.SQLStateConverter.handledNonSpecificE
xception(SQLStateConverter.java:103)
     [java]     at org.hibernate.exception.SQLStateConverter.convert(SQLStateCon
verter.java:91)
...
     [java]     at oracle.jdbc.driver.OraclePreparedStatement.executeUpdate(Orac
lePreparedStatement.java:3367)
     [java]     at org.hibernate.persister.entity.AbstractEntityPersister.update
(AbstractEntityPersister.java:2342)
     [java]     ... 12 more
     [java] Phase 6, catched exception: org.hibernate.exception.GenericJDBCExcep
tion: could not update: [rvvtest.RvvEntity#1]

BUILD SUCCESSFUL
Total time: 27 seconds
C:\hibernate-3.2\RvvHibernateJPA>
```

The results are analogical with the results of the Hibernate Core test runs with the differences we noticed on using the three different DBMS systems.

These Hibernate examples prove that Oracle's ORA_ROWSCN can be used as the effective row version field in RVV programming discipline. The test runs prove that for reliable RVV discipline we need to bypass the Hibernate cache. For Oracle we need isolation level SERIALIZABLE. The cost of this technique compared with the SELECT .. FOR UPDATE is that without the row lock in Phase 6 we loose the competition to concurrent updates. RVV discipline correctly prevents us from writing over the updates of the winners. If some concurrent update has won the competition, there is no reason to retry the transaction of Phase 6.

The transaction code of Phase 6 suits best for DB2 which based on the locks of the SERIALIZABLE isolation (mapped to DB2's corresponding isolation level RR) can protect the row against concurrent updates and save the work done by the user of the whole use case.

For SQL Server Hibernate should apply the (UPDLOCK) table hint as tested in Appendix 2. This can be done using the table hint as part of the view definition like we use in Appendix 8. SQL Server 2005 specific code could also be used to force the Phase 6 code to win the concurrency conflict by setting `DEADLOCK_PRIORITY` to `10`,

but winning the deadlock would mean that the competitor would be selected as the victim of the deadlock – and rolled back by DBMS – so losing the update work.

Appendix 8 RVV and Microsoft LINQ to SQL

The evolution / divergence of ORMs and persistence APIs of data access technologies continues. After Java Persistence API JPA, Microsoft has introduced as part of .NET Framework 3.5 Language-Integrated Query LINQ which has been integrated with various .NET languages such as C#, Visual Basic .NET, etc and provides service layer above ADO.NET for accessing various data sources, LINQ to SQL for accessing relational data, LINQ to XML for accessing XML data, LINQ to Objects for accessing data in object layers, etc. We will focus on LINQ to SQL integrated in C# and accessing SQL Server 2005 databases in the following. Comparing with Embedded SQL and SQL/CLI the data access will be programmed using SQL looking "native" data access expressions of C# with help of IntelliSence of the Visual Studio 2008 IDE. However for run time these data access expressions will be translated into SQL.

For our example use case we have implemented the following C# program including the client and the data access parts in the same code as follows:

```
/* RVV test implemented using C# and LINQ to SQL
*
*******************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Transactions;
using System.Text;

namespace Linq_RvTest {
    // strong typing of the SQL view RvTestU as class RvTest
    [Table(Name = "rvv.RvTestU")]
    public class RvTest {
        private int _ID;
        [Column(IsPrimaryKey = true, Storage = "_ID")]
        public int ID {
            get {
                return this._ID;
            }
            set {
                this._ID = value;
            }
        }
        private string _S;
        [Column(Name = "S")]
        public string S {
            get {
                return this._S;
            }
            set {
                this._S = value;
            }
        }
        private long _Rv;
```

```
    [Column(Name = "RV", IsDbGenerated = true)]
    public long Rv {
        get {
            return this._Rv;
        }
        set {
            this._Rv = value;
        }
    }
}

class ClientLINQ {
    static void Main(string[] args) {
        try {
            // DataContext connection string.
            DataContext db = new DataContext
                (@"Data Source=(local)\SQLEXPRESS;" +
                  "Initial Catalog=TEST;" +
                  "Trusted_Connection=Yes");
            // Attach the DataContext log to show generated SQL
            db.Log = Console.Out;
            // Phase 1
            Console.WriteLine("RVV/LINQ Test <2.0>\n" +
                              "Listing of the rows:");
            // ~ Phase 2  - data acess
            // Get a typed table to run queries
            Table<RvTest> myTable = db.GetTable<RvTest>();
            // Query for rows
            IQueryable<RvTest> rvQuery =
                from r in myTable select r;
            // Phase 3 - user interface
            Console.WriteLine("ID:\t S:");
            foreach (RvTest r in rvQuery) {
                Console.WriteLine("{0}\t{1},", r.ID, r.S);
            }
            Console.Write("Select a row by id : ");
            int id = Int32.Parse(Console.ReadLine());
            // Phase 4 - data acess
            var myRow = (from r in myTable
                         where r.ID == id
                         select r).First();
            // Phase 5 - User interface
            Console.WriteLine("Found the row ");
            Console.WriteLine("ID={0}, S={1}, RV={2}",
                         myRow.ID, myRow.S, myRow.Rv);
            long oldRv = myRow.Rv;
            Console.Write("Enter new value for column S: ");
            string newS = Console.ReadLine();
            // Phase 6
            TransactionOptions txOpt = new TransactionOptions();
            txOpt.IsolationLevel =
                System.Transactions.IsolationLevel.RepeatableRead;
            using (TransactionScope txs = new TransactionScope
                  (TransactionScopeOption.Required, txOpt)) {
                try {
                    //----------- rvv lineset 1 ---------
                    //var curRow = (from r in myTable
                    //              where r.ID == id
                    //              select r).First();
                    //if (curRow.Rv == oldRv) {
                    //    curRow.S = newS;
                    //-----------------------------------
                        myRow.S = newS;
                        db.SubmitChanges();
```

```
                    //----------- rvv lineset 2 ----------
                    //}
                    //else
                    //   throw new Exception(
                    //             "New version in database!");
                    //-----------------------------------
                    txs.Complete();
                }
                catch (Exception e) {
                    Console.WriteLine("SubmitChanges error: " +
                            e.Message + "\nSource: " + e.Source +
                            "\nInnerException: " + e.InnerException);
                }
            }
        }
        catch (Exception ex) {
            Console.WriteLine("Error: " + ex.Message +
                "\nSource: " + ex.Source +
                "\nInnerException: " + ex.InnerException
               );
        }
        finally {
            // Prevent console window from closing.
            Console.Write("Press ENTER to exit ..");
            Console.ReadLine();
        }
    }
  }
}
```

The database view RvTestU is created as follows

```
CREATE VIEW rvv.RvTestU
AS
SELECT id, s, CAST(rv AS BIGINT) AS rv
FROM   rvv.VersionTest WITH (UPDLOCK)
```

The clause "`WITH (UPDLOCK)`" will force U-lock request also for SELECT commands to avoid deadlocking.

The LINQ queries are translated into Transact-SQL commands, but their affect to application logic can be suprise programmer. We tried to apply our RVV logic for Phase 6 code (see linesets 1 and 2). By debugging the C# code in Visual Studio 2008 and tracing the communication with SQL Server using Profiler we found out that LINQ query of lineset 1 generated corresponding Transact-SQL SELECT query to database, but after that the contents of object `curRow` reflected the old row values, so there was no use for the RVV logic of Type 2. Instead of the manual RVV logic we found that `db.SubmitChanges()` automatically takes care of the row version verification according to Type 1, so we could comment out the linesets 1 and 2.

In the following we run the program with concurrent sessions (see textboxes below) and log the DataContext to the console window:

```
RVV/LINQ Test <2.0>
Listing of the rows:
ID:      S:
SELECT [t0].[ID], [t0].[S], [t0].[RV] AS [Rv]
FROM [rvv].[RvTestU] AS [t0]
```

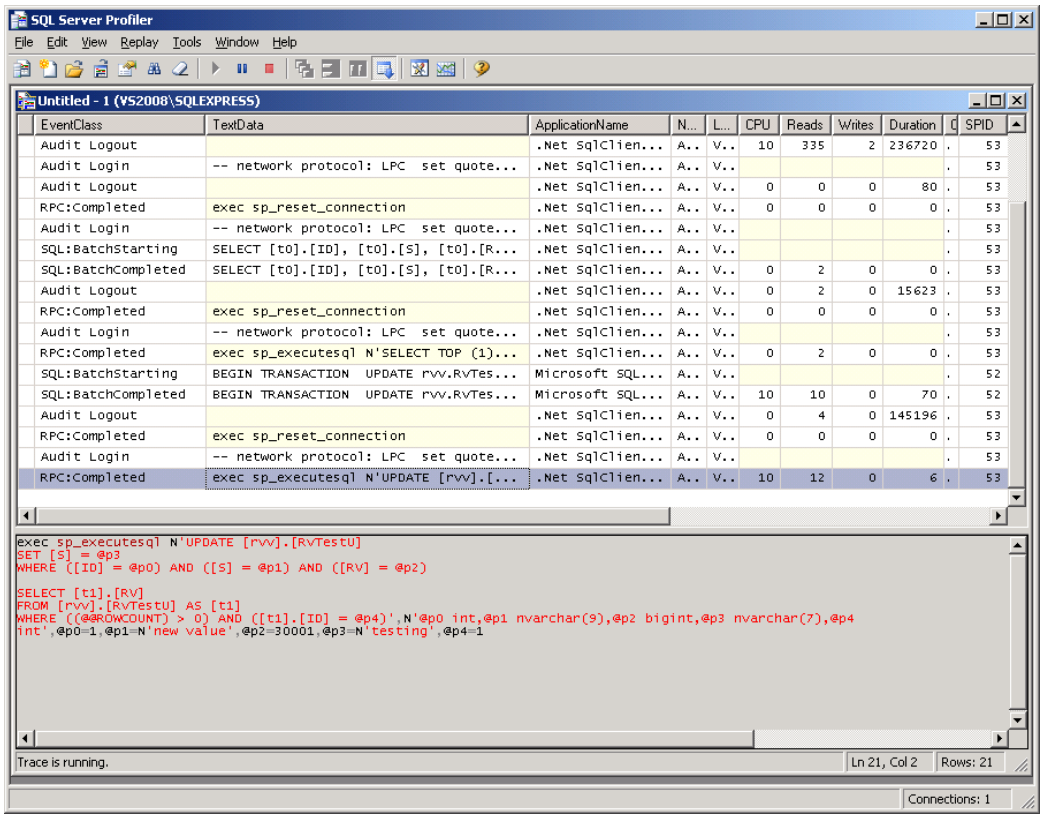
```
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.21022.8

1       some text,
2       some text,
Select a row by id : 1
SELECT TOP (1) [t0].[ID], [t0].[S], [t0].[RV] AS [Rv]
FROM [rvv].[RvTestU] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.21022.8

Found the row
ID=1, S=some text, RV=29001
Enter new value for column S: new value

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) [some text]
-- @p2: Input BigInt (Size = 0; Prec = 0; Scale = 0) [29001]
-- @p3: Input NVarChar (Size = 9; Prec = 0; Scale = 0) [new value]
-- @p4: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.21022.8

Press ENTER to exit ..

RVV/LINQ Test <2.0>
Listing of the rows:
ID:      S:
SELECT [t0].[ID], [t0].[S], [t0].[RV] AS [Rv]
FROM [rvv].[RvTestU] AS [t0]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.21022.8

1       new value,
2       some text,
Select a row by id : 1
SELECT TOP (1) [t0].[ID], [t0].[S], [t0].[RV] AS [Rv]
FROM [rvv].[RvTestU] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.21022.8

Found the row
ID=1, S=new value, RV=30001
Enter new value for column S: testing
```

Concurrent session S52:

```
BEGIN TRANSACTION
UPDATE rvv.RvTestU
SET s = 'changed'
WHERE id = 1;
SELECT * FROM rvv.RvTestU;
COMMIT;
```

```
<ENTER key>
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) [new value]
-- @p2: Input BigInt (Size = 0; Prec = 0; Scale = 0) [30001]
-- @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

SubmitChanges error: Row not found or changed.
Source: System.Data.Linq
InnerException:
Press ENTER to exit ..
```

In final step traced by SQL Server Profiler we see that LINQ compares the row version comparing all columns in the view:



Appendix 9 RVV and Web Services

For comparison of different data access technologies we have implemented our use case also as synchronous Web Services, although we don't consider Web Service as a proper technology for this kind of use cases. Web Services are meant for accessing external services and has been advertised as a platform-independent technology for integrating loosely-coupled applications. For service request and response messages Web Services use SOAP envelope messages standardised by the W3C organisation.

Since demonstrating the platform independence of the Web Services is out scope of our presentation, we try to keep this simple using just .NET Web Services. We use Visual Studio 2008 Web Service project template and access the local SQL Server 2005 Express database instance which comes with Visual Studio. For data access we use SqlClient .NET Data Provider and C# source language. With small changes in the code we could use for example OleDb .NET Data Provider and access any DBMS. For explanations on use of ADO.NET and transactions we refer to .NET Framework documentation of Microsoft.

To build reliable applications while using loosely-coupled Web Services in synchronous way, we need to react on exceptions at the server-side and pass the information of the exceptions also to the client. In SOAP 1.2 message envelopes the exception information is passed in Fault node as the only element inside the Body element. In .NET 3.5 platform the SoapException provides this processing as a service, so that we need to care only need to build the Detail element of the Fault node at server-side and to extract the exception information from it at the client-side.

```
/* DBTechNet / Martti Laiho 2008-03-21
 * .NET Web Service implementation of the RVV Test
 *
 * ************************************************************/
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Transactions;
using System.Data;
using System.Xml;
using System.Data.SqlClient;

[WebService(Namespace = "http://dbtechnet.org/ws")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
// To allow this Web Service to be called from script, using ASP.NET AJAX,
// uncomment the following line.
// [System.Web.Script.Services.ScriptService]
    public class Service : System.Web.Services.WebService {

    static string strConn = (@"Data Source=(local)\SQLEXPRESS;"+
                  "Initial Catalog=TEST; Trusted_Connection=Yes");

    public Service () {
        //Uncomment the following line if using designed components
        //InitializeComponent();
    }

    [WebMethod]
    public string GetRvList() {
```

```
        DataSet ds = new DataSet();
        TransactionOptions txOpt = new TransactionOptions();
        txOpt.IsolationLevel =
            System.Transactions.IsolationLevel.ReadCommitted;
        try {
            using (TransactionScope txs = new
            TransactionScope(TransactionScopeOption.Required, txOpt)) {
                using (SqlConnection cn = new SqlConnection(strConn)) {
                    string strSql =
                        "SELECT id,s FROM rvv.RvTest ORDER BY id";
                    cn.Open();
                    SqlDataAdapter da = new SqlDataAdapter(strSql, cn);
                    da.Fill(ds, "RvTest");
                }
                txs.Complete();
                return ds.GetXml().ToString();
            }
        }
        catch (SqlException ex) {
            throw RaiseException("GetRvList", "rvv",
                                 "http://dbtechnet.org/ws", ex.Message,
                                 ex.Number.ToString(), ex.Source,
                                 SoapException.ServerFaultCode);
        }
        catch (Exception ex) {
            throw RaiseException("GetRvList", "rvv",
                                 "http://dbtechnet.org/ws", ex.Message,
                                 "1001", ex.Source,
                                 SoapException.ServerFaultCode);
        }
    }
    [WebMethod]
    public string GetRvTest(int id) {
        DataSet ds = new DataSet();
        TransactionOptions txOpt = new TransactionOptions();
        txOpt.IsolationLevel =
            System.Transactions.IsolationLevel.ReadCommitted;
        try {
            using (TransactionScope txs = new
                TransactionScope(TransactionScopeOption.Required, txOpt)) {
                using (SqlConnection cn = new SqlConnection(strConn)) {
                    string strSql =
                        "SELECT id,s,rv FROM rvv.RvTest WHERE id = " +
                        id.ToString();
                    cn.Open();
                    SqlDataAdapter da = new SqlDataAdapter(strSql, cn);
                    da.Fill(ds, "RvTest");
                }
                txs.Complete();
            }
            return ds.GetXml().ToString();
        }
        catch (SqlException ex) {
            throw RaiseException("GetRvTest", "rvv",
                                 "http://dbtechnet.org/ws", ex.Message,
                                 ex.Number.ToString(), ex.Source,
                                 SoapException.ServerFaultCode);
        }
        catch (Exception ex) {
            throw RaiseException("GetRvTest", "rvv",
                                 "http://dbtechnet.org/ws", ex.Message,
                                 "1002", ex.Source,
                                 SoapException.ServerFaultCode);
        }
    }
```

```
[WebMethod]
public string RvUpdate(int id, string s, long rv) {
    int rowsUpdated = 0;
    long curRv = -1;
    TransactionOptions txOpt = new TransactionOptions();
    txOpt.IsolationLevel =
        System.Transactions.IsolationLevel.ReadCommitted;
    try {
        using (TransactionScope txs = new
            TransactionScope(TransactionScopeOption.Required, txOpt)) {
            using (SqlConnection cn = new SqlConnection(strConn)) {
                cn.Open();
                SqlCommand cmd = cn.CreateCommand();
                cmd.CommandText =
                    "UPDATE rvv.RvTest SET s = @s " +
                    "WHERE id = @id AND rv = @rv ";
                cmd.Parameters.Add("@s",SqlDbType.VarChar,
                                                s.Length).Value = s;
                cmd.Parameters.Add("@id",SqlDbType.Int, 5).Value = id;
                cmd.Parameters.Add("@rv",SqlDbType.BigInt,9).Value = rv;
                rowsUpdated = cmd.ExecuteNonQuery();
                if (rowsUpdated < 1)
                    throw RaiseException("RvUpdate", "rvv",
                             "http://dbtechnet.org/ws",
                             "Row Version conflict", "1004",
                             this.ToString(),
                             SoapException.ServerFaultCode);
                cmd.CommandText =
                    "SELECT rv FROM rvv.RvTest WHERE id = @id ";
                curRv = (long)cmd.ExecuteScalar();
            }
            txs.Complete();
        }
        return "<NewDataSet>" +
               "<rowsUpdated>" + rowsUpdated + "</rowsUpdated> " +
               "<currentRv>" + curRv + "</currentRv> " +
               "</NewDataSet> " ;
    }
    catch (SqlException ex) {
        throw RaiseException("RvUpdate", "rvv",
                             "http://dbtechnet.org/ws", ex.Message,
                             ex.Number.ToString(), ex.Source,
                             SoapException.ServerFaultCode);
    }
    catch (Exception ex) {
        throw RaiseException("RvUpdate", "rvv",
                             "http://dbtechnet.org/ws", ex.Message,
                             "1003", ex.Source,
                             SoapException.ServerFaultCode);
    }
}

public SoapException RaiseException(string uri,string ns,
       string webServiceNamespace,string errorMessage,
       string errorNumber, string errorSource,
       XmlQualifiedName faultCodeLocation) {
// modified from article "Exception Handling in WebServices"
// written by Thiru Thangarathinam
// http://www.developer.com/net/csharp/article.php/10918_3088231_1
    XmlDocument xmlDoc = new XmlDocument();
    //Create the Detail node
    XmlNode rootNode = xmlDoc.CreateNode(XmlNodeType.Element,
            SoapException.DetailElementName.Name,
            SoapException.DetailElementName.Namespace);
    //Build specific details for the SoapException
```

```
        //Add first child of detail XML element.
        XmlNode errorNode = xmlDoc.CreateNode(XmlNodeType.Element,
                ns + ":Error",webServiceNamespace);
        //Create and set the value for the ErrorNumber node
        XmlNode errorNumberNode = xmlDoc.CreateNode(XmlNodeType.Element,
                ns + ":Number", webServiceNamespace);
        errorNumberNode.InnerText = errorNumber;
        //Create and set the value for the ErrorMessage node
        XmlNode errorMessageNode = xmlDoc.CreateNode(XmlNodeType.Element,
                ns + ":Message", webServiceNamespace);
        errorMessageNode.InnerText = errorMessage;
        //Create and set the value for the ErrorSource node
        XmlNode errorSourceNode = xmlDoc.CreateNode(XmlNodeType.Element,
                ns + ":Source", webServiceNamespace);
        errorSourceNode.InnerText = errorSource;
        //Append the Error child element nodes to the root detail node.
        errorNode.AppendChild(errorNumberNode);
        errorNode.AppendChild(errorMessageNode);
        errorNode.AppendChild(errorSourceNode);
        //Append the Detail node to the root node
        rootNode.AppendChild(errorNode);
        //Construct the exception
        SoapException soapEx = new SoapException(errorMessage,
                faultCodeLocation, uri, rootNode);
       //Return the exception instance back to the caller
        return soapEx;
    }
}
```

Following is the C# client code implementing our example use case accessing the Web Services above. We have first used standalone client to test reading of the XML messages by XmlReader of .NET 2.0 using conditional compiling directives (#define ..., #if ..., #endif) of C#, and we have left this test coding to help some readers in use of this technique.

```
/*  DBTechNet / Martti Laiho 2008-03-23

RvTestClient.cs
Test client for the RvTest implemented as .NET Web Service

rem  Scripts for .NET command prompt:
rem  Open Visual Studio Command Prompt and CD <your working directory>
rem  Generating RvTestProxy.cs for the web service (use single line!)
wsdl /language:CS /out:RvTestProxy.cs
http://localhost:1104/rvtest/Service.asmx?wsdl

rem  Compiling the RvTestClient assemly (use single line!)
csc /r:System.Web.dll,System.Web.Services.dll,System.XML.dll,System.dll
RvTestClient.cs RvTestProxy.cs

rem  Running the client
RvTestClient

*/
#define PROD
//#undef PROD
using System;
using System.IO;
using System.Xml;
using System.Data;
using System.Web.Services.Protocols;
namespace WsRvTest {
  class RvTestClient {
```

```
    public static void Main() {
#if PROD
       Service ws = new Service();
#endif
       string nsUri = "http://dbtechnet.org/ws";
       string myProlog =
                "<?xml version='1.0' encoding='utf-8' ?> " +
                "<string xmlns='" + nsUri + "'>";
       try {
            // Phase 1
            Console.WriteLine("RVV/WebService Test <1.0> \n"+
                              "Listing of the rows:");
            string sdoc = myProlog +
#if PROD
              ws.GetRvList() +
#else
              "<NewDataSet> <RvTest> <id>1</id> <s>test</s> "+
              "</RvTest> <RvTest> <id>2</id> <s>some text</s> "+
              "</RvTest> </NewDataSet>" +
#endif
            "</string> ";
            // Phase 3 - user interface
            Console.WriteLine("ID:\t S:");
            using (XmlReader rdr =
                   XmlReader.Create(new StringReader(sdoc))) {
                rdr.ReadToFollowing("NewDataSet", nsUri);
                while (rdr.ReadToFollowing("RvTest", nsUri)) {
                    rdr.ReadToFollowing("id");
                    int id1 = rdr.ReadElementContentAsInt("id", nsUri);
                    Console.Write(id1 + "\t");
                    rdr.ReadToFollowing("s");
                    string s2 = rdr.ReadElementContentAsString("s", nsUri);
                    Console.WriteLine(s2);
                }
            }

            Console.Write("Select a row by id : ");
            int id = Int32.Parse(Console.ReadLine());
            // Phase 5 - User interface
            Console.WriteLine("Found the row ");
            sdoc = myProlog +
#if PROD
                ws.GetRvTest(id) +
#else
                "<NewDataSet> <RvTest> <id>1</id> <s>test</s> "+
                "<rv>13006</rv> </RvTest> </NewDataSet>" +
#endif
                "</string> ";
            long oldRv = 0L;
            using (XmlReader rdr =
                   XmlReader.Create(new StringReader(sdoc))) {
                rdr.ReadToFollowing("NewDataSet", nsUri);
                rdr.ReadToFollowing("id");
                int id2 = rdr.ReadElementContentAsInt("id", nsUri);
                Console.WriteLine("ID:\t" + id2);
                rdr.ReadToFollowing("s");
                string s2 = rdr.ReadElementContentAsString("s", nsUri);
                Console.WriteLine("S: \t" + s2);
                rdr.ReadToFollowing("rv");
                oldRv = rdr.ReadElementContentAsLong("rv", nsUri);
                Console.WriteLine("RV: \t" + oldRv);
            }
            Console.Write("Enter new value for column S: ");
            string newS = Console.ReadLine();
            // Phase 6
```

```
            Console.WriteLine("updating row 1:");
            sdoc = myProlog +
#if PROD
                ws.RvUpdate(id, newS, oldRv) +
#else
                "<NewDataSet> <rowsUpdated>1</rowsUpdated> "+
                "<currentRv>13007</currentRv> </NewDataSet> " +
#endif
                " </string> ";
            using (XmlReader rdr =
                   XmlReader.Create(new StringReader(sdoc))) {
                rdr.ReadToFollowing("NewDataSet", nsUri);
                rdr.ReadToFollowing("rowsUpdated");
                int rows = rdr.ReadElementContentAsInt("rowsUpdated",
                                                       nsUri);
                Console.WriteLine("rowsUpdated:\t" + rows);
                rdr.ReadToFollowing("currentRv");
                long newRv = rdr.ReadElementContentAsLong("currentRv",
                                                          nsUri);
                Console.WriteLine("currentRv: \t" + newRv);
            }
        }
        catch (System.Web.Services.Protocols.SoapException soapE)
        {
            // SOAP Fault info
            Console.WriteLine("SOAP fault...");
            Console.WriteLine("Code: " + soapE.Code.ToString());
            Console.WriteLine("Actor: " + soapE.Actor);
            XmlDocument doc = new XmlDocument();
            doc.LoadXml(soapE.Detail.OuterXml);
            XmlNamespaceManager nsExp = new
                    XmlNamespaceManager(doc.NameTable);
            nsExp.AddNamespace("rvv","http://dbtechnet.org/ws");
            XmlNode categoryNode =
            doc.DocumentElement.SelectSingleNode("rvv:Error",nsExp);
            string errNumber =
                categoryNode.SelectSingleNode("rvv:Number",nsExp).InnerText;
            Console.WriteLine("Number: " + errNumber);
            string errMessage =
                categoryNode.SelectSingleNode("rvv:Message",nsExp).InnerText;
            Console.WriteLine("Message: " + errMessage);
            string errSource =
                categoryNode.SelectSingleNode("rvv:Source",nsExp).InnerText;
            Console.WriteLine("Source: " + errSource);
        }
        Console.WriteLine("\nPress ENTER to exit ..");
        Console.ReadLine();
    }
  }
}
```

Following is a test run by this client, similar to our previous examples:

```
C:\TEMP>RvTestClient
RVV/WebService Test <1.0>
Listing of the rows:
ID:      S:
1       initial
2       some text
Select a row by id : 1
Found the row
ID:     1
S:      initial
RV:     26002
```

```
Enter new value for column S: new value
update row 1:
rowsUpdated:    1
currentRv:      26003

Press ENTER to exit ..


C:\TEMP>RvTestClient
RVV/WebService Test <1.0>
Listing of the rows:
ID:      S:
1       new value
2       some text
Select a row by id : 1
Found the row
ID:     1
S:      new value
RV:     26003
Enter new value for column S: test
```

Concurrent database update:

```
UPDATE rvv.RvTest
SET s = 'changed'
WHERE id = 1 ;
```

```
<ENTER>
update row 1:
SOAP fault...
Code: http://schemas.xmlsoap.org/soap/envelope/:Server
Actor: RvUpdate
Number: 1003
Message: Row Version conflict
Source: App_Code.s4cd9tp3

Press ENTER to exit ..
```

Test after changing the name of column `rv` in the database:

```
C:\TEMP>RvTestClient
RVV/WebService Test <1.0>
Listing of the rows:
SOAP fault...
Code: http://schemas.xmlsoap.org/soap/envelope/:Server
Actor: GetRvList
Number: 207
Message: Invalid column name 'rv'.
Could not use view or function 'rvv.RvTest' because of binding errors.
Source: .Net SqlClient Data Provider

Press ENTER to exit ..
```

Test after shutting down the database server:

```
C:\TEMP>RvTestClient
RVV/WebService Test <1.0>
Listing of the rows:
SOAP 1.1 fault...
Code: http://schemas.xmlsoap.org/soap/envelope/:Server
Actor: GetRvList
Number: 233
```

```
Message: A transport-level error has occurred when sending the request
to the se
rver. (provider: Shared Memory Provider, error: 0 - No process is on
the other e
nd of the pipe.)
Source: .Net SqlClient Data Provider

Press ENTER to exit ..
```

Appendix 10 RVV using PHP

Perl is an old scripting language for system administration tasks, but it has been applied also to generate dynamic HTML pages using CGI technology. For accessing databases Perl needs DBMS dependent DBD (Database Driver) modules, and a de facto standard general DBI (Database Interface) wrapper has been written to provide a unified data access interface over these DBD modules.

PHP is a scripting language dedicated for dynamic HTML pages. There is not yet a single universal data access API for PHP like DBI for Perl scripting language, although some competing solutions are available in Web, but the DBMS vendors are providing there proprietary database drivers, the programming paradigms of which are tailored to the behaviour of the DBMS in question. For comparison of these different data access technologies we have implemented our use case also in PHP language embedded on HTML pages and using PHP APIs of Oracle and SQL Server 2005. The following examples show that these DBMS dependent data access APIs have totally different programming models and provide only limited services. We have avoided use of real presentation layer this far, but PHP is mainly used as the script language for creating dynamic HTML pages, so now we present "full scale solutions":

Oracle

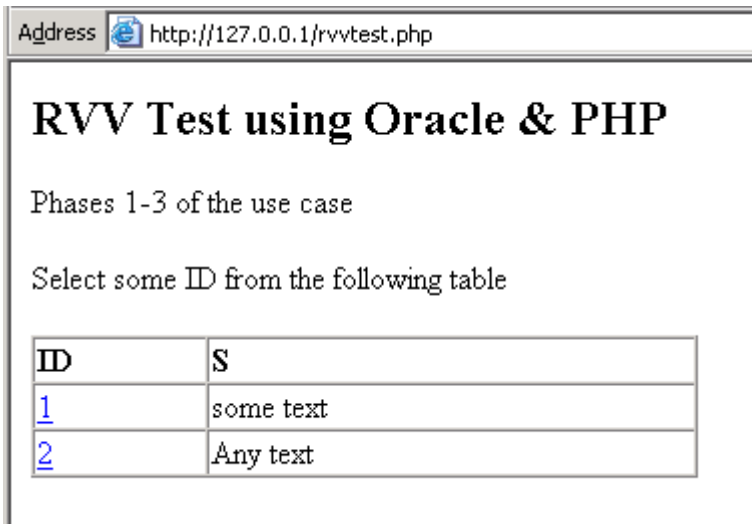
Currently available PHP data access driver of Oracle is called OCI8. For this test we have used Oracle Express 10g, and since it does not support the rowdependencies table option, we use the trigger solution for server-side stamping of column RV. However, the following solution is independent of the server-side stamping and applies to Oracle 11g1 as well.

Our solution starts from the following HTML page:

```
<!-- rvvtest.php
Start page of the RVV test sample
accessing local Oracle XE using PHP and OCI8 API.
2008-09-10 Martti Laiho
-->
<html><head><title>Oracle RVV Test</title></head>
<body>
<h2>RVV Test using Oracle & PHP </h2>
Phases 1-3 of the use case
<p> Select some ID from the following table</p>
<table border=1 cellspacing='0' width='50%'>
<tr><td><b>ID</b></td><td><b>S</b></td></tr>
<?php
    $db_conn = oci_connect("rvv", "rvv", "//127.0.0.1/XE");
    $cmdstr = "SELECT id, s FROM rvv.RvTest";
    $parsed = oci_parse($db_conn, $cmdstr);
    oci_execute($parsed);
    $nrows = oci_fetch_all($parsed, $results);
    for ($i = 0; $i < $nrows; $i++ ) {
      $ID = $results["ID"][$i] ;
      echo "<tr>\n";
      // link to next phase passing the ID of the selected row as parameter
      echo "<td> <a href=\"rvvphase4.php?ID=".$ID."\">" . $ID . "</a></td>";
      echo "<td> " . $results["S"][$i] . "</td>";
      echo "</tr>\n";
```

```
    }
    oci_close($db_conn);
?>
</body> </html>
```

and it presents the following form to the user for phase 3 of our use case

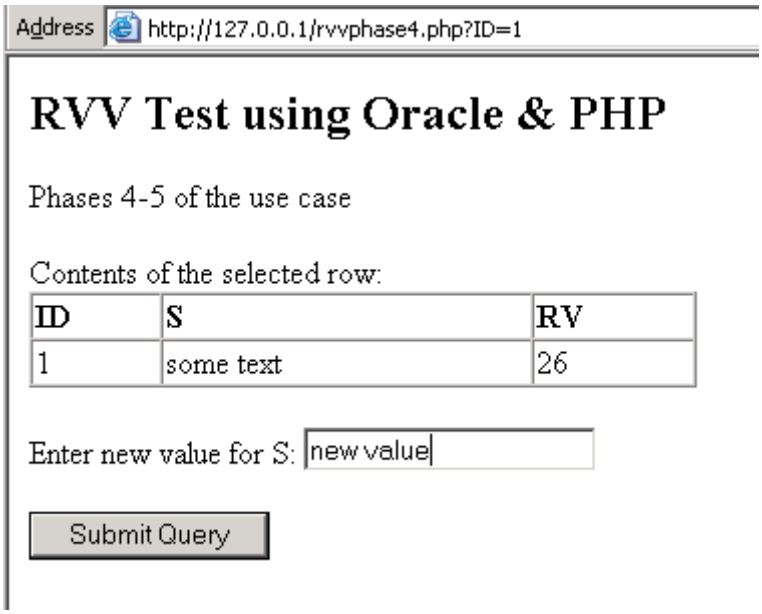


User selects the row to be updated by clicking the ID link of the row and this link leads the dialogue to the following PHP page rvvphase4.php

```
<!-- rvvphase4.php
Show contents of the selected row and get the new value for S
2008-09-10 Martti Laiho
-->
<html><head><title>Oracle RVV Test</title></head><body>
<h2>RVV Test using Oracle & PHP </h2>
Phases 4-5 of the use case<br/><br/>
Contents of the selected row:
<table border=1 cellspacing='0' width='50%'>
<tr><td><b>ID</b></td><td><b>S</b></td><td><b>RV</b></td></tr>
<?php
    $ID = $_GET['ID'];
    $db_conn = oci_connect("rvv", "rvv", "//127.0.0.1/XE");
    // Only committed data available in Oracle,
    // but it may already be stale data !
    $cmdstr = "SELECT id, s, rv FROM rvv.RvTest WHERE id =" . $ID ;
    $parsed = oci_parse($db_conn, $cmdstr);
    oci_execute($parsed);  // auto-commit
    $nrows = oci_fetch_all($parsed, $results);
    for ($i = 0; $i < $nrows; $i++ ) {
      $RV = $results["RV"][$i];
      echo "<tr>\n";
      echo "<td>" . $results["ID"][$i] . "</td>";
      echo "<td> " . $results["S"][$i] . "</td>";
      echo "<td> " . (int)$results["RV"][$i] . "</td>";
      echo "</tr>\n";
    }
    echo "</table>";
    oci_close($db_conn);
    echo "<form action='rvvphase6.php' method='post'>
        <input type='hidden' name='ID' value='" . $ID ."'/>
        <input type='hidden' name='RV' value='" . $RV . "'/>
        <p>Enter new value for S: <input type='text' name='S' /></p>
```

```
        <p><input type='submit' /></p> </form> ";
?>
</body></html>
```

This will present the following form of phase 5 to the user

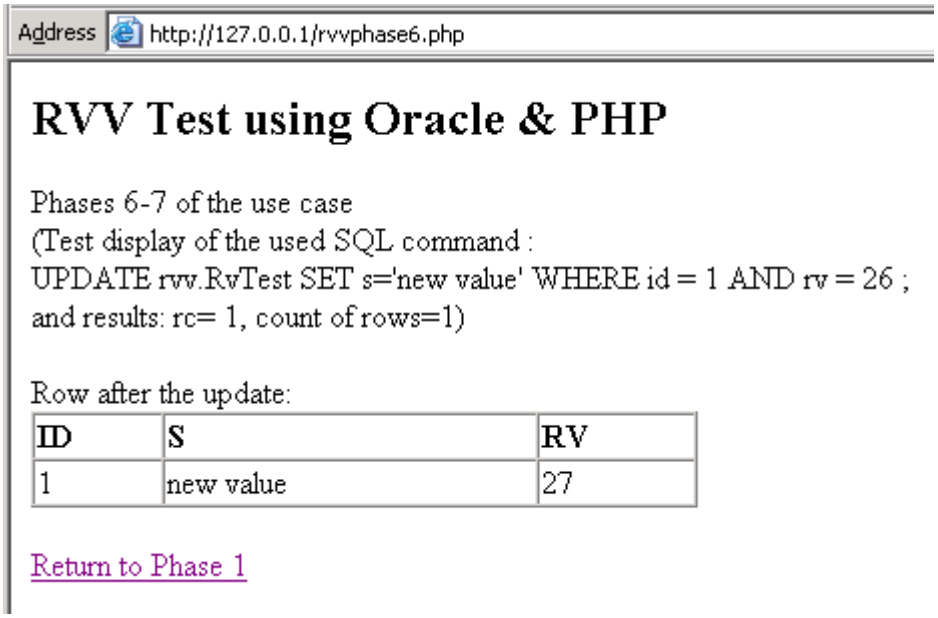


User enters a new value for column S in the textbox and presses the "Submit Query" button, which leads the dialogue to following php file of phase 6

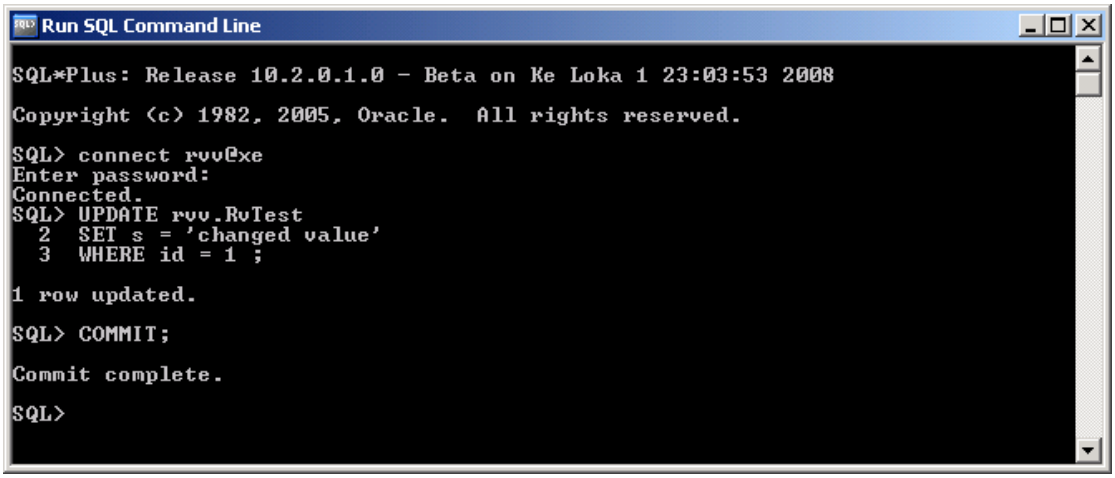
```
<!-- rvvphase6.php
Type 1 RVV update transaction and showing the results after that
Transaction starts with UPDATE - so no need for setting isolation level !
2008-09-10 Martti Laiho
-->
<html><head><title>Oracle RVV Test</title></head><body>
<h2>RVV Test using Oracle & PHP</h2>
Phases 6-7 of the use case<br/>
<?php
    // load the parameters
    $ID = $_POST['ID'];
    $S = $_POST['S'];
    $RV = $_POST['RV'];
    $db_conn = oci_connect("rvv", "rvv", "//127.0.0.1/XE");
    $cmdstr = "UPDATE rvv.RvTest SET s='" . $S .
              "' WHERE id = " . $ID . " AND rv = " . $RV ;
    echo "(Test display of the used SQL command : <br> " . $cmdstr .
         " ;<br>";
    $parsed = oci_parse($db_conn, $cmdstr);
    $rc = oci_execute($parsed, OCI_DEFAULT); // starts a transaction
    $count = OCIRowCount($parsed);
    echo "and results: rc= ".$rc.", count of rows=".$count.")<br/> ";
    if ($rc<>1) {
        $er = oci_error($parsed);
        var_dump ($er);
        echo "<br>Error : " . htmlspecialchars($er['message']);
             oci_rollback($db_conn);  // rollback the transaction
    }
    else
    if ($count<>1) {
        echo "** Update failed due to stale data!<br>";
             oci_rollback($db_conn);  // rollback the transaction
```

```
    }
    else {
        // read back the current contents
        $cmdstr = "SELECT id, s, rv FROM rvv.RvTest WHERE id = " . $ID ;
        $parsed = oci_parse($db_conn, $cmdstr);
        oci_execute($parsed, OCI_DEFAULT);  // transaction continues
        $nrows = oci_fetch_all($parsed, $results);
        echo "<br/> Row after the update:  ";
        echo "<table border=1 cellspacing='0' width='50%'>\n<tr>\n";
        echo "<td><b>ID</b></td>\n<td><b>S</b></td>\n";
        echo "<td><b>RV</b></td>\n</tr>\n";
        for ($i = 0; $i < $nrows; $i++ ) {
              echo "<tr>\n";
              echo "<td> " . $results["ID"][$i] . "</td>";
              echo "<td> " . $results["S"][$i] . "</td>";
              echo "<td> " . $results["RV"][$i] . "</td>";
              echo "</tr>\n";
        }
        echo "</table>";
        oci_commit($db_conn);  // commit the transaction
    }
    oci_close($db_conn);
    echo "<p> <a href='rvvtest.php'>Return to Phase 1</a></p>";
?>
</body></html>
```

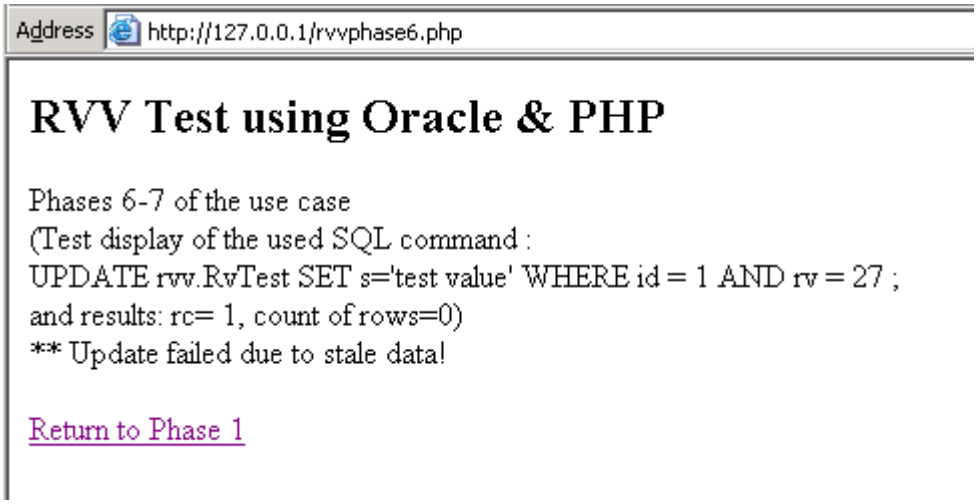
and the results of phase 6 will be presented to the user on the following HTML page



To test our RVV technique we return to beginning PHP page and repeat the previous phases up to the form of phases 4-5. Before pressing the “Submit Query” button we update the row of ID 1 using SQL*Plus session as follows



After this we continue our test by pressing the "Submit Query" button, and we will see that the RVV predicate in our Type 1 update prevents us from writing over the update of the SQL*Plus transaction.



SQL Server 2005

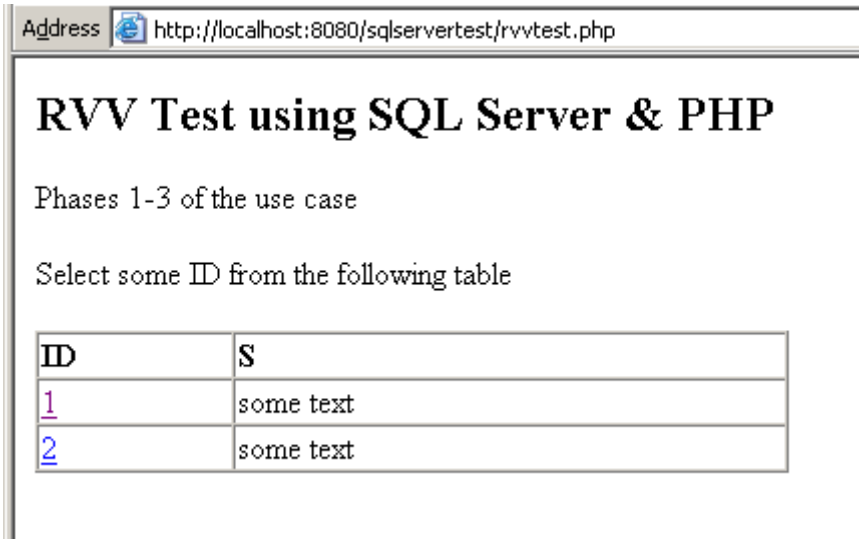
SQL Server 2005 Driver for PHP is available at the Microsoft Download Center and the API of driver is documented on MSDN pages at http://msdn.microsoft.com/en-us/library/cc296221.aspx

Due to proprietary DBMS and API of the driver we don't need to try to keep the data access model general, so we have modified the solution previous solution for Oracle now just for accessing SQL Server. The table and views in SQL Server database are the same which we used in Appendix x using the rovwersion column RV for server-side stamping.

Our solution starts from the following HTML page:

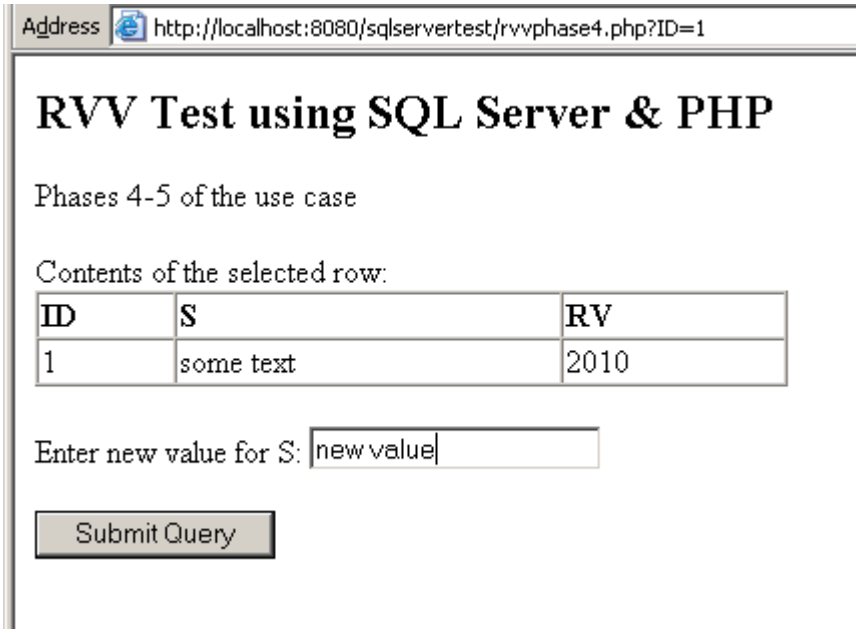
```
<!-- rvvtest.php
Start page of the RVV test sample accessing local SQL Server using PHP API.
```

```
2008-09-30 Martti Laiho
-->
<html><head><title>SQL Server RVV Test</title></head>
<body>
<h2>RVV Test using SQL Server & PHP </h2>
Phases 1-3 of the use case
<p> Select some ID from the following table</p>
<table border=1 cellspacing='0' width='50%'>
<tr><td><b>ID</b></td><td><b>S</b></td></tr>
<?php
    $serverName = "(local)";
    $connectionOptions = array("Database"=>"TEST");
    /* Connect using Windows Authentication. */
    $conn = sqlsrv_connect( $serverName, $connectionOptions);
    if( $conn === false ) {
       die( FormatErrors( sqlsrv_errors() ) );
    }
    $tsql = "SELECT id, s FROM rvv.RvTestList";
    $rset = sqlsrv_query( $conn, $tsql);
    if ( $rset === false ) {
       echo "Error in statement execution.\n";
       die( print_r( sqlsrv_errors(), true));
    }
    while( $row = sqlsrv_fetch_array( $rset, SQLSRV_FETCH_ASSOC)) {
       $ID = $row['id'] ; /* Note: identifiers are case-sensitive */
       echo "<tr>\n";
       // link to next phase passing the ID of the selected row as parameter
       echo "<td> <a href=\"rvvphase4.php?ID=".$ID."\">" . $ID . "</a></td>";
       echo "<td> " . $row['s'] . "</td>";
       echo "</tr>\n";
    }
    sqlsrv_close($conn);

    function FormatErrors( $errors ) {  /* Display errors. */
           echo "Error information: <br/>";
           foreach ( $errors as $error ) {
                 echo "SQLSTATE: ".$error['SQLSTATE']."<br/>";
                 echo "Code: ".$error['code']."<br/>";
                 echo "Message: ".$error['message']."<br/>";
           }
       }
?>
</body> </html>
```



We select the first ID 1 and proceed to the next form:

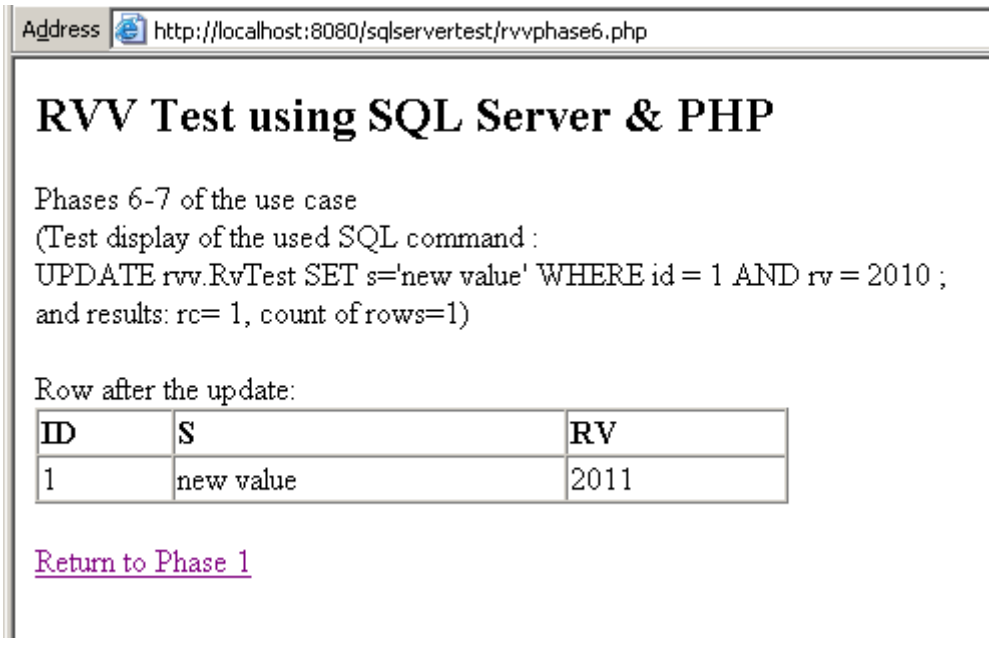
```
<!-- rvvphase4.php
Show contents of the selected row and get the new value for S
2008-09-30 Martti Laiho
-->
<html><head><title>SQL Server RVV Test</title></head><body>
<h2>RVV Test using SQL Server & PHP </h2>
Phases 4-5 of the use case<br/><br/>
Contents of the selected row:
<table border=1 cellspacing='0' width='50%'>
<tr><td><b>ID</b></td><td><b>S</b></td><td><b>RV</b></td></tr>
<?php
    $ID = $_GET['ID'];
    $serverName = "(local)";
    $connectionOptions = array("Database"=>"TEST");
    /* Connect using Windows Authentication. */
    $conn = sqlsrv_connect( $serverName, $connectionOptions);
    if( $conn === false ) {
        die( FormatErrors( sqlsrv_errors() ) );
    }
    /* setting Isolation level - to be sure */
    $tsql = "SET TRANSACTION ISOLATION LEVEL READ COMMITTED";
    $stmt = sqlsrv_prepare($conn, $tsql);
    sqlsrv_execute($stmt);
    /* and still in autocommit mode */
    $tsql = "SELECT id, s, rv FROM rvv.RvTest WHERE id = ? ";
    $params = array ( $ID );
    $rset = sqlsrv_query( $conn, $tsql, $params );
    if ( $rset === false ) {
       echo "Error in statement execution: ";
       die( print_r( sqlsrv_errors(), true));
    }
    while( $row = sqlsrv_fetch_array( $rset, SQLSRV_FETCH_ASSOC)) {
       $RV = $row["rv"];
       echo "<tr>\n";
       echo "<td> " . $row["id"] . "</td>";
       echo "<td> " . $row["s"] . "</td>";
       echo "<td> " . (int)$row["rv"] . "</td>";
       echo "</tr>\n";
    }
    echo "</table>";
    sqlsrv_close($conn);

    echo "<form action='rvvphase6.php' method='post'>
         <input type='hidden' name='ID' value='" . $ID ."'/>
         <input type='hidden' name='RV' value='" . $RV . "'/>
         <p>Enter new value for S: <input type='text' name='S' /></p>
         <p><input type='submit' /></p> </form> ";

    function FormatErrors( $errors ) {     /* Display errors. */
           echo "Error information: <br/>";
           foreach ( $errors as $error ) {
                 echo "SQLSTATE: ".$error['SQLSTATE']."<br/>";
                 echo "Code: ".$error['code']."<br/>";
                 echo "Message: ".$error['message']."<br/>";
           }
    }
?>
</body> </html>
```



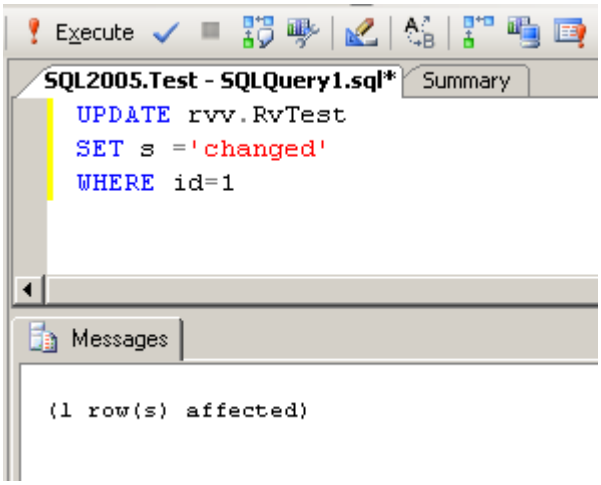
After entering a new value for S we proceed to the form of phase 6

```
<!-- rvvphase6.php
Type 1 RVV update transaction and showing the results after that
Transaction starts with UPDATE - so no need for setting isolation level !
2008-09-30 Martti Laiho
-->
<html><head><title>SQL Server RVV Test</title></head><body>
<h2>RVV Test using SQL Server & PHP</h2>
Phases 6-7 of the use case<br/>
<?php
    // load the parameters
    $ID = $_POST['ID'];
    $S = $_POST['S'];
    $RV = $_POST['RV'];
    $serverName = "(local)";
    $connectionOptions = array("Database"=>"TEST");
    /* Connect using Windows Authentication. */
    $conn = sqlsrv_connect( $serverName, $connectionOptions);
    if( $conn === false ) {
        die( FormatErrors( sqlsrv_errors() ) );
    }
    sqlsrv_begin_transaction ($conn );
    $tsql = "UPDATE rvv.RvTest SET s= ? " .
            "WHERE id = ? AND rv = ? ";
    $params = array ($S, $ID, $RV );
    echo "(Test display of the used SQL command : <br> " . $tsql . " ;<br>";
    $stmt = sqlsrv_prepare($conn, $tsql, $params);
    $rc = sqlsrv_execute($stmt);
    $count = sqlsrv_rows_affected($stmt);
    echo "and results: rc= ".$rc.", count of rows=".$count.")<br/> ";
    if ($rc<>1) {
        echo "** Update failed due to stale data?<br>";
        FormatErrors( sqlsrv_errors() ) ;
        sqlsrv_rollback($conn);  // rollback transaction
    }
    else
    if ($count<>1) {
        echo "** Update failed due to stale data!<br>";
        sqlsrv_rollback($conn);  // rollback transaction
```

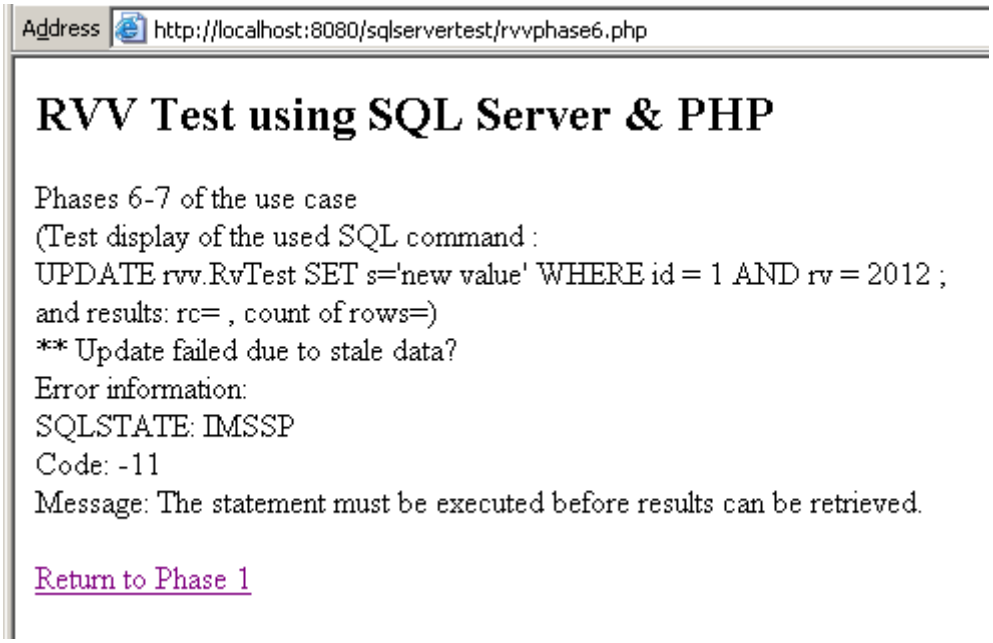
```
    }
    else {
        // read back the current contents in auto-commit mode
        $tsql = "SELECT id, s, rv FROM rvv.RvTest WHERE id = ?";
        $params = array ( $ID );
        $rset = sqlsrv_query( $conn, $tsql, $params );
        echo "<br/> Row after the update:  ";
        echo "<table border=1 cellspacing='0' width='50%'>\n<tr>\n";
        echo "<td><b>ID</b></td>\n<td><b>S</b></td>\n<td>";
        echo "<b>RV</b></td>\n</tr>\n";
        while( $row = sqlsrv_fetch_array( $rset, SQLSRV_FETCH_ASSOC)) {
                $RV = $row["rv"];
                echo "<tr>\n";
                echo "<td> " . $row["id"] . "</td>";
                echo "<td> " . $row["s"] . "</td>";
                echo "<td> " . (int)$row["rv"] . "</td>";
                echo "</tr>\n";
        }
        echo "</table>";
        sqlsrv_commit ($conn);
    }
    sqlsrv_close($conn);
    echo "<p> <a href='rvvtest.php'>Return to Phase 1</a></p>";

    function FormatErrors( $errors ) {  /* Display errors. */
              echo "Error information: <br/>";
              foreach ( $errors as $error ) {
                       echo "SQLSTATE: ".$error['SQLSTATE']."<br/>";
                       echo "Code: ".$error['code']."<br/>";
                       echo "Message: ".$error['message']."<br/>";
              }
    }

?>
</body></html>
```



When we continue the test returning to phase 1 and just before pressing query on entering again new text value for S on form of phase 4 we accidently update the row of ID 1 in SQL Server 2005 Management Studio as follows



So, on perssing the Submit Query we get the folloing messages on our screen



Unfortunately the error diagnostic of the PHP driver is not accurate enough, so we just know from the context of our experiment that the reason is row version failure.

Appendix 11 RVV using Ruby

Ruby is a new object-oriented programming language which can be used as a stand-alone language or integrated in the Rails framework. The language itself has adapted features from many earlier programming languages and the result is quite intuitive. For database access the DBMS vendors provide drivers of their own and a Perl DBI like interface DBI wrapper is provided as the vendor independent universal database interface. As RVV implementations written in Ruby language we present a simple program accessing Oracle XE 10g using the OCI8 API and a modified program which uses the DBI API of Ruby.

RVV implementation using OCI8

```
# rvvOCI8.rb
# Martti Laiho 2008-09-10
#
# This is a simple RVV test accessing Oracle XE database
# using OCI8 API from Ruby script.
# For methods of OCI8 see
# http://ruby-oci8.rubyforge.org/en/api_OCI8.html
#
# sample run:
# cd C:\RubyOnRails\ruby\work
# ruby rvvoci8.rb
#
require 'oci8'
def main()
#
# Phase 1
puts "Ruby/OCI8 RVV Test <1.0>"
#
# Phase 2 - data access
conn = OCI8.new("rvv","rvv","//localhost/XE")
query = conn.exec('SELECT id, s FROM rvv.rvtest')
conn.commit
#
# Phase 3 - user interface
puts "Listing of the rows:"
puts "ID:     S:"
while row = query.fetch do
  puts row.inspect
end
print "Enter ID of the selected row: "
id = gets
#
# Phase 4 - data access
query = conn.exec("SELECT id, s, rv FROM rvv.RvTest WHERE id=" +
                  id.to_s + " FOR UPDATE")
row=query.fetch
conn.commit
#
# Phase 5 - user interface
puts "ID: " + row[0].to_s
puts "S:  " + row[1]
rv = row[2].to_i
puts "RV: " + rv.to_s
print "Enter new value for S: "
s = gets.to_s.delete("\n")
#
# Phase 6 - data access
# type 1 Update - no need for setting isolation level
sql="UPDATE rvv.rvtest SET s='"+s+"' WHERE id=" +
     id.to_s + " AND rv = " + rv.to_s
```

```
puts sql
num_rows = conn.exec(sql)
#puts num_rows.to_s + " rows updated"
unless num_rows == 1
  raise "trying to update old version of the row!"
end
#row = conn.exec("SELECT id, s, rv FROM rvv.RvTest WHERE id=" + id).fetch
#row=query.fetch
#puts row.inspect
conn.commit
conn.logoff
end
#
main()
# end of program
```

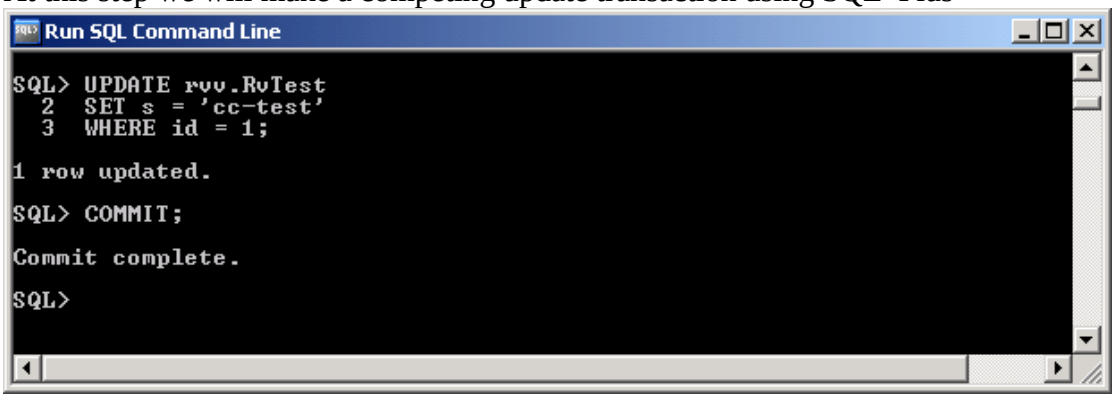
To test this Ruby program we first run it without competition

```
C:\RubyOnRails\ruby\work>ruby rvvoci8.rb
Ruby/OCI8 RVV Test <1.0>
Listing of the rows:
ID:     S:
[1, "some text"]
[2, "some text"]
Enter ID of the selected row: 1
ID: 1
S:  some text
RV: 10
Enter new value for S: new value
UPDATE rvv.rvtest SET s='new value' WHERE id=1
 AND rv = 10
```

and then we repeat the run as follows

```
C:\RubyOnRails\ruby\work>ruby rvvoci8.rb
Ruby/OCI8 RVV Test <1.0>
Listing of the rows:
ID:     S:
[1, "new value"]
[2, "some text"]
Enter ID of the selected row: 1
ID: 1
S:  new value
RV: 11
Enter new value for S:
```

At this step we will make a competing update transaction using SQL*Plus



.. and we continue our Ruby test as follows:

```
Enter new value for S: upd test
UPDATE rvv.rvtest SET s='upd test' WHERE id=1
 AND rv = 11
rvvoci8.rb:55:in `main': trying to update old version of the row!
(RuntimeError)

        from rvvoci8.rb:64

C:\RubyOnRails\ruby\work>
```

So the RVV logic prevents us from writing over the competing update.

RVV implementation using Ruby DBI

The next Ruby example shows how to implement RVV logic using the DBI interface

```
# rvvDBI.rb
# Martti Laiho 2008-09-10
#
# This is a simple RVV test accessing Oracle XE database
# using the generic DBI API (as wrapper of OCI8 API) from Ruby script.
# For methods of DBI see
# http://ruby-dbi.rubyforge.org/rdoc/index.html
#
# sample run:
# cd C:\RubyOnRails\ruby\work
# ruby rvvDBI.rb
#
require 'dbi'
def main()
#
# Phase 1
puts "Ruby/DBI RVV Test <1.0>"
#
# Phase 2 - data access
conn = DBI.connect("DBI:OCI8://localhost/XE","rvv","rvv")
query = conn.prepare('SELECT id, s FROM rvv.rvtest')
query.execute
conn.commit
#
# Phase 3 - user interface
puts "Listing of the rows:"
puts "ID:     S:"
while row = query.fetch do
  puts row.inspect
end
print "Enter ID of the selected row: "
id = gets
#
# Phase 4 - data access
query = conn.prepare("SELECT id, s, rv FROM rvv.RvTest WHERE id=" +
                   id.to_s + " FOR UPDATE")
query.execute
row=query.fetch
conn.commit
#
# Phase 5 - user interface
puts "ID: " + row[0].to_s
puts "S:  " + row[1]
rv = row[2].to_i
puts "RV: " + rv.to_s
```

```
print "Enter new value for S: "
s = gets.to_s.delete("\n")
#
# Phase 6 - data access
# type 1 Update - no need for setting isolation level
sql = "UPDATE rvv.rvtest SET s='"+s+"' WHERE id=" +
      id.to_s + " AND rv = " + rv.to_s
puts sql
num_rows = conn.do(sql)
unless num_rows == 1
  raise "trying to update old version of the row!"
end
conn.commit
conn.disconnect
end
#
main()
# end of program
```

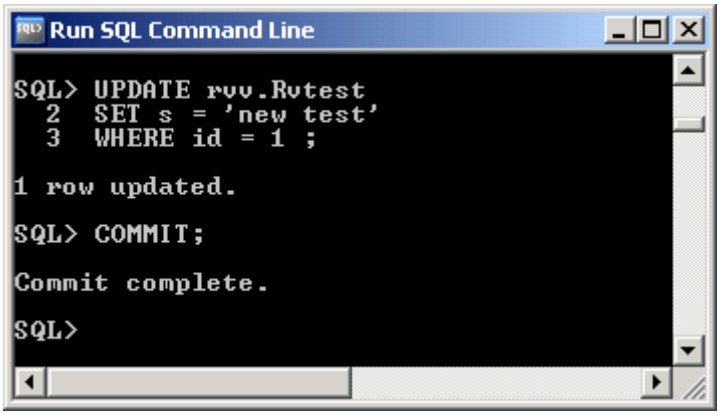
Like in our previous test we run the program without concurrent sessions as follows

```
C:\RubyOnRails\ruby\work>ruby rvvDBI.rb
Ruby/DBI RVV Test <1.0>
Listing of the rows:
ID:     S:
[1, "cc-test"]
[2, "some text"]
Enter ID of the selected row: 1
ID: 1
S:  cc-test
RV: 12
Enter new value for S: some text

UPDATE rvv.rvtest SET s='some text' WHERE id=1
 AND rv = 12
```

Then we repeat the test as follows

```
C:\RubyOnRails\ruby\work>ruby rvvDBI.rb
Ruby/DBI RVV Test <1.0>
Listing of the rows:
ID:     S:
[1, "some text"]
[2, "some text"]
Enter ID of the selected row: 1
ID: 1
S:  cc-test
RV: 13
Enter new value for S:
```

At this step we will make again a competing update transaction using SQL*Plus



.. and we continue our Ruby test as follows:

```
Enter new value for S: my test

UPDATE rvv.rvtest SET s='my test' WHERE id=1
 AND rv = 13
rvvDBI.rb:58:in `main': trying to update old version of the row!
(RuntimeError)
        from rvvDBI.rb:64

C:\RubyOnRails\ruby\work>
```

These tests prove that we can apply the RVV logic in Ruby on accessing Oracle either using directly the OCI8 API or the DBI wrapper.

Index

ADF, 20
ADO.NET object model, 15
blind overwriting problem, 5
BMP, 17
client-side scope of concurrency, 8
CMP, 17
DB2 SPL, 28
disconnected mode, 16
EJB2, 17
EJB3, 19
Hibernate Core, 23
Hibernate EntityManager, 24
Hibernate JPA, 24
Hibernate's LockMode UPGRADE, 74
incremental change identifier, 12
isolation levels of DB2, 39
isolation levels of SQL standard, 39
JDO, 22
JPA, 22
LINQ, 25
LINQ to SQL, 25
LSCC, 9
MVC, 7
MVCC, 9
  isolation levels, 49
OpenJPA, 22
optimistic concurrency control, 45
Optimistic Lock pattern, 10
Optimistic Locking, 6, 19
optimistic methods, 3
ORA_ROWSCN, 31
  disadvantages, 74
ORM, 17, 19
PL/SQL, 31
Pooled Connection, 13
principle of SQL Base Views, 33
Retryer Pattern, 14
ROW CHANGE TIMESTAMP, 29
row change token, 29
row version column, 6
ROWDEPENDENCIES clause, 31
row-level trigger, 28
ROWVERSION, 30
SCN, 32
SELECT..FOR UPDATE -locking, 47
server-side scope of concurrency, 9
SNAPSHOT
  isolation level, 45
snapshot of locks, 40
SOA, 26
SOAP, 26
stale data, 4
stamping
  client-side, 19, 33
  server-side, 28, 33
TIMESTAMP, 30
TopLink Essentials, 20
TopLink POJO, 20
TransactionScope, 56
Transact-SQL, 29
trigger, 28
Type 0 update, 5
Type 1 update, 5
Type 2 update, 6
U-LOCK problem, 25
UPDLOCK, 25
  table hint, 44
UPDLOCK view, 34
Web Services, 26