Implementing Row Version Verification for Persistence Middleware using SQL Access Patterns

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

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

Tim Lessner
School of Computing
University of the West of Scotland
Paisley PA1 2BE, UK
tim.lessner@uws.ac.uk

***Abstract*—Modern web-based applications are often built as multi-tier architecture using persistence middleware. Middleware technology providers recommend the use of Optimistic Concurrency Control (OCC) mechanism to avoid the risk of blocked resources. However, most vendors of relational database management systems implement only locking schemes for concurrency control. As a consequence a kind of OCC has to be implemented at client or middleware side. The aim of this paper is to recommend Row Version Verification (RVV) as a mean to realize an OCC at the middleware level. To help the developers with the implementation of RVV we propose to use SQL access patterns. For performance reasons the middleware uses buffers (cache) of its own to avoid network traffic and to reduce disk I/O. This caching, however, complicates the use of RVV because the data in the middleware cache may be stale (outdated). We investigate various data access technologies, including the Java Persistence API and Microsoft's LINQ technologies in combination with commercial database systems for their ability to use the RVV programming discipline. The use of persistence middleware that tries to relieve the programmer from the low level transaction programming turns out to even complicate the situation in some cases.**

The contribution of this paper are patterns and guidelines for an implementation of OCC at the middleware layer using RVV. Our approach prevents from inconsistencies, reduces locking to a minimum, considers a couple of mainstream technologies, and copes with the effects of concurrency protocols, data access technologies, and caching mechanisms.

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

I. Introduction

Databases provide reliable data storage services, but especially for business critical applications the use of these services requires that applications will obey the concurrency control protocol of the underlying Database Management System (DBMS).

In this paper we look at the application development from the Online Transaction Processing (OLTP) point of view, and especially on the modern mainstream commercial DBMS used by industry , namely DB2, Oracle, and SQL Server, with ISO SQL standard as the common denominator. The ideas described here are extensions of the work first presented in [1]. A cornerstone of data management is the proper transaction processing, and a generally accepted requirement for reliable flat SQL transactions is the ACID transaction model defined by Haerder and Reuter [2]. The acronym ACID stems from the initials of the four well known transaction properties: Atomicity, Consistency, Isolation, and Durability. However, the original definition for Isolation "Events within a transaction must be hidden from other transactions running concurrently" cannot be fulfilled by the mainstream commercial DBMS systems. They only provide a concurrency control (CC) mechanism that, according to the isolation level settings, prevents a transaction itself from seeing changes made by concurrent transactions, and protects the transaction's updates against overwrites from other transactions during its execution time. The implemented isolation levels follow roughly the definitions in the ISO SQL standard, but with semantics of the used CC mechanism.

A typical CC mechanism used to ensure the isolation of concurrent SQL transactions in mainstream DBMS, for example DB2 and SQL Server, is some variant of multi-granular Locking Scheme Concurrency Control (which we call shortly LSCC, whereas in literature the more general term pessimistic concurrency control is often used). LSCC systems use exclusive locks to protect writes until the end of a transaction, and shared locks protect read operations against concurrent modifications of the data. The term multi-granular refers to a mechanism of applying locks, for example, at row level or table level, and compatibility issues of these locking levels are solved using special intent locks. The isolation level declared for the transaction fine-tunes the duration of the shared locks. Locking may block concurrent transactions, and may lead to deadlocks as a kind of concurrency conflict, in which the victim transaction is chosen by the DBMS according to internal rules among the competing transactions. Another typical CC mechanism is some variant of Multi-Versioning Concurrency Control (MVCC), which is used for example by Oracle and a specially configured SQL Server database. In addition, Oracle also uses locking at table level and provides the possibility of programmatic row level locking by the SELECT .. FOR UPDATE variant

of the SQL select statement.

The MVCC mechanism always allows reading of committed data items without blocking. In case of concurrency competition between transactions, the first writer transaction wins and conflicting updates or write operations are prevented by raising serialization exceptions.

The third available concurrency control mechanism is server-side optimistic concurrency control (OCC), as presented by Kung and Robinson [3], in which transactions only read contents from the database while all changes are first written in the private workspace of the transaction, and finally at transaction commit phase - after successful validation - the changes will be synchronized into the database as an atomic action. In case of concurrency competition between transactions the first transaction to COMMIT is the winner and others will get a serialization exception, however, after requesting the COMMIT only. Currently server-side OCC has not yet been implemented in any commercial mainstream DBMS systems. The only implementation of which we know is the Pyrrho DBMS [4] and VoltDB [5], [6]. Both products focus on transaction processing in the Cloud and apply OCC to overcome the drawbacks of locking that significantly reduces the concurrency, hence, the response time and scalability crucial for Cloud storage services. Please note, however, that scalability is a general requirement and not limited to Cloud storage services. One reason, why OCC might be more appealing for the Cloud, is the basic assumption of OCC that conflicts are rare and especially applications that are built on top of the cloud are not primarily OLTP applications with a high concurrency on the same record.

With the advent of multi-tier web applications, or more precisely, decentralized and loosely coupled transactional systems, client-side OCC has gained new attention. Providers of enterprise architecture frameworks (like Java Enterprise Edition (JEE)) and persistence middleware (like object relational mappers, e.g., Hibernate) propose to use optimistic concurrency control to avoid the risk of blocked resources and orphan locks. Developers face now the situation that they have to implement a kind of optimistic concurrency control over the existing concurrency control provided by the DBMS. But shifting the burden to the middleware or application is a tricky task [1]. First, there is a need to distinguish between business/user transaction and SQL transaction. Second, we have to deal with transactional legacy applications that bypass the middleware.

Definition: (SQL transaction) A *SQL transaction* is defined as finite sequence of SQL statements that obey the ACID properties.

This definition is equivalent to the general transaction definition from Härder and Reuter [2], but restricted to SQL databases. SQL transactions are supported by relational DBMS.

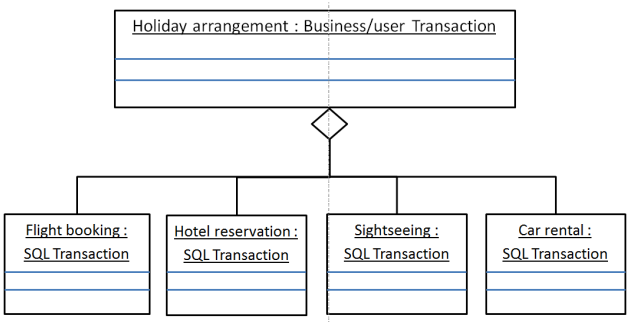

Figure 1. Example UML diagram for a business/user transaction

Definition: (Business/user transaction) A *business/user transaction* is an unit of work that holds the ACID properties. The unit of work is defined by the business (application) as an aggregate of applications, each one executing at least one SQL transaction.

Ideally business/user transactions should be supported by a transaction coordinator, that ensures the overall ACID outcome. But the result is not necessarily based on Herbrand semantic. Depending on the business rules there might be more than one successful outcome (e.g. an alternative result) or compensation transactions could be involved. As example, think of a complex business transaction like booking a holiday arrangement that involves multiple, non-integrated applications, e.g. flight booking, hotel reservation, car rental contract, or siteseeing tour (see Figure 1). If flight and hotel bookings succeed and car rental fails, but the siteseeing tour is possible, this could be considered as an alternative successful business transaction. However, a car rental without flight or hotel booking should be forbidden. Each of the applications that is part of a business/user transaction comes with its own SQL transactions that might be already committed before another application happens to fail, hence, a compensation for already committed SQL transactions is needed. In this paper the only aggregate used for RVV is a sequence of SQL transactions.

In general, the design of a multi-tier architecture requires to split a business/user transaction into a sequence of SQL transactions for several reasons:

- The business transaction needs to access more than one DBMS
- Access to a database is executed under different sessions because of session pooling
- The business transaction uses legacy systems not designed to support distributed transactions.

This leads to a situation where the DBMS is unaware of the complete business/user transaction because different sessions and different transactions form the actual transaction. On the one hand, the splitting avoids automatic locking for a possibly unpredictable time, on the other hand, it requires additional mechanisms to ensure the global consistency of

the business/user transaction.

But, if concurrent business transactions are splitted into multiple SQL transactions these may interfere without the possibility for any help by the DBMS. For instance a lost update could arise due to interleaved execution. Therefore, the applications, the middleware, and the DBMS need to co-operate somehow.

The implementation of a concurrency control mechanism in every application is inefficient and depends on the skills of the programmer. Therefore it is preferable to implement the concurrency control protocol in the middleware for the benefit of all applications using this middleware. The implementation of a concurrency control protocol providing full serializability is often to restrictive and derogates performance. A good compromise is the Row Version Verification (RVV) discipline that avoids the lost update problem from the business transaction's view.

The contribution of this paper is how to apply the data access patterns of [11] to different middleware technologies. The novel parts are techniques how to reliably implement RVV discipline for complex business transactions as defined above.

A. Structure of the Paper

In the next Section we motivate the RVV and describe its mechanism. After the related work, Section IV introduces the blind overwriting in the business context and compares it to the generally known lost update problem, which is typically covered in database literature. Section V presents three SQL patterns, which serve as guideline for the implementations of RVV. Section VI starts with the presentation of a typical use case including SQL statements for its setup on a relational database. Each of the following Sections, VII (JDBC, .NET), VIII (Hibernate, JPA), IX (MS LINQ), and X (JDO) present implementations of RVV using the data access patterns presented in Section V. Section XI concludes the paper with a comparison between these technologies.

II. The Row Version Verification (RVV) Discipline

The *lost update* is a typical phenomenon in multi-user file-based systems without proper concurrency control, i.e., a record x updated by some process A will be overwritten by some other concurrent process B like in the following problematic canonical schedule [7, pp. 62 - 63]: $r_A(x), r_B(x), w_A(x), w_B(x)$, where $r_T(x)$ and $w_T(x)$ denote read and write operations of transaction T on data item x .

With a sufficient isolation level a DBMS would not allow for a lost update and if a LSCC is used x would be locked for the transaction's duration and other transactions are prevented from accessing x. But, if we decide to follow the recommendation of the middleware vendors or the writers of JEE tutorials to use OCC, the DBMS should be configured not to use transactions to avoid locking or to use auto-commit such that every data access statement results in a transaction with locks held as short as possible. With this configuration the DBMS would be unable to prevent the lost update problem within the business transaction's view as it appears only as a blind write to the DBMS. A *blind write* is defined as overwriting an existing data item with a new value that is independent from the old value. RVV, however, as a concurrency control mechanism at the Middleware layer, on top of the DBMS, prevents from the lost update phenomenon.

RVV depends on a technical row version column that needs to be added to every SQL table. Its value is incremented whenever any data in the row is changed. A verification of the row version enables to detect if any concurrent transaction has modified the data meanwhile. If this happened, the validation fails and the transaction has to abort. If the row version is still the same, then the transaction may write the new data. In pseudo-code the mechanism looks like this:

```
(t1)    Read(x, versionX)
        // Input data
        // Process x
        old_versionX := versionX
(t2)    if (old_versionX = Read(versionX))
(t2)    then
(t2)        Write(x, versionX+1)
(t2)    else
(t2)        // Report serialization conflict
(t2)    end if
```

In the time period between t_1 and t_2 other concurrent transactions may access and modify data element x. The if-then-else block is a critical section that must be executed in an uninterruptable manner.

If the row version is cached by the middleware this could lead to stale data. Therefore, it is necessary to circumvent the middleware cache for the row version in order to apply RVV.

RVV is better known under the misleading and ambiguous name *optimistic locking* (see [7, pp. 365 - 367], [8], [9], [10]) even if there is no explicit locking involved.

The motivation to use RVV results from the practice that for example web applications usually split a business transaction into several SQL transactions as previously explained for multi-tier transactional applications. In this case the database concurrency mechanism cannot ensure transactional properties for the business transaction, but RVV helps to avoid at least the blind overwriting. Consider a typical concurrency scenario with two users that try to book the same flight online. First, a list of flights is displayed (Phase 2, in Figure 2), second, a certain flight is chosen (Phase 4), and third, the flight is booked (Phase 6). When a second user books the same seat and commits before the first user proceeds to Phase 6 then the first user would overwrite

the reservation of the second user. This could be avoided by re-reading the seats in Phase 6 and compare it with Phase 4 before storing the new number.

We consider the RVV discipline as a critical reliability criterion in web based transactional application development. The proper way for applications to meet the RVV discipline is to strictly follow the SQL data access patterns presented in Laux and Laiho [11], which will be recapped in Section V. The patterns describe how to implement the RVV protocol for different database programming interfaces. These patterns essentially ensure that the row version is checked before overwriting a data value. The patterns describe how to deal with different concurrency schemes of the underlying DBMS and how to exploit triggers to support RVV from the database side.

In the present paper these data access patterns are applied to the already mentioned generic use case (Figure 2) and code samples show implementations for various technologies.

III. Related Work

Concurrency control is a cornerstone of transaction processing, it has been extensively studied for decades. Namely Gray and Reuter [9] studied locking schemes, whereas Badal [12], Kung and Robinson [3] developed optimistic methods for concurrency control (OCC). OCC distinguishes three phases (see Figure 3) within a transaction:

- read phase
- validation phase
- write phase

The first phase includes user input and thinking time. It may last for an unpredictable period. The following phases are without any user interaction. Validation and write phases are therefore very short in the range of milliseconds. The last two phases are critical in the sense that exclusive access is required. Failing to do so could result in inconsistent data, e.g lost update.

Unland [13] presents OCC algorithms without critical section. He specifically focuses on a OCC solution that solves the starvation problem, increasing the chance for long living and read only transactions to survive. Using these algorithms would allow relaxed locking but involve checking the read set against all concurrent transactions. Even if a full OCC of this type would be implemented at the middleware layer, it could only control applications that use this middleware service. Therefore this algorithms can be ruled out for our approach. Although, OCC mechanisms have been studied already 30 years ago, hardly any commercial DBMS implements algorithms of this type (see Bernstein and Newcomer [14] or Gray and Reuter [9]) except for Multi-Version Concurrency Control (MVCC).

A higher concurrency for query intensive transactions provides MVCC as described by Stearns and Rosenkrantz [15] and Bernstein and Goldman [16]. Comparing locking with MVCC it can be said that a database system with locking holds a single truth of every data item, and if it is locked by others, one needs to wait until the lock is released, whereas a MVCC systems holds a history of the truth. On read committed isolation level it is always possible to read the latest committed truth without waiting, and on serializable isolation level (also called snapshot in some systems), the data that will be read is the latest committed truth at the beginning of the transaction.

In case of MVCC, the middleware has to make sure that caching is not invalidating the multi-versioning system. This problem is discussed by Seifert and Scholl [17] who counteract with a hybrid of invalidation and propagation message. In Web applications the risk of improperly terminating transactions is extremely high (users simply ”click away”). In such cases snapshot data or locks in the case of a locking protocol are held until (hopefully) some timeout elapses. This can severely degrade performance.

With the dissemination of middleware, OCC has been recommended by IT-vendors ([18], [19], [20]) for transactional e-business and m-commerce applications but only little effort has been spent on the realisation using commercial SQL databases. Adya et al [21] recommend to use the system clock as blocking-free protocol for global serialization in distributed database systems. However this approach has to fail if the resolution is not fine enough to produce unique timestamps as we have shown for Oracle [22].

Nock ([8, pp. 395-404] describes an *optimistic lock* pattern based on version information. He points out that this access pattern does not use locking (despite of its misleading name) and therefore avoids orphan locks. His pattern does not address caching. Akbar-Husain [19] believes that demarking the method that checks the version with the required transaction attribute will be sufficient to avoid lost updates. He does not consider that only a strong enough isolation level like REPEATABLE READ or SERIALIZABLE will achieve the desired results.

During decades of development in relational database technologies the programming paradigms in data access technologies have constantly changed. The two mainstream schools in object oriented programming today are the Java [23] and the Microsoft .NET framework [24] camps, both provide call-level APIs and Object-Relational Mappings (ORM) of their own. The problems of using RVV with the older version of Enterprise Java Beans 2.0 are discussed in [25].

We follow in this paper the object persistence abstractions of Hoffer, Prescott, and Topi [26, Chapter 16] and implement the RVV discipline at the middleware level applying the SQL access patterns described in Section V.

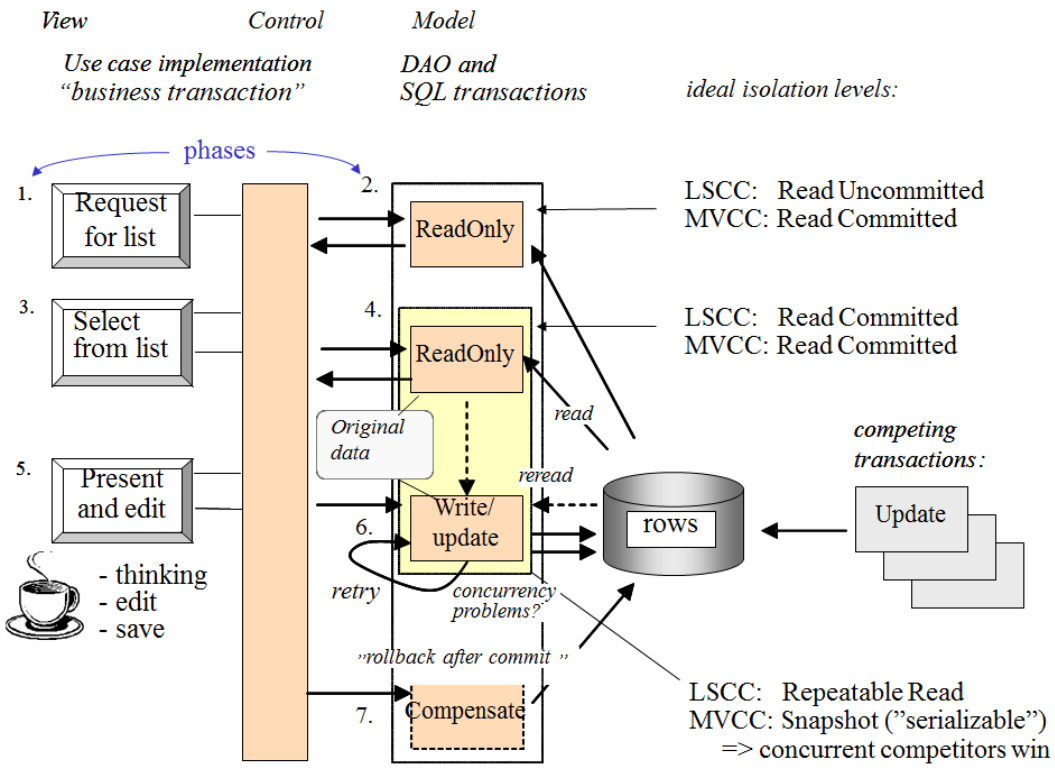

Figure 2. Business transaction with SQL transactions (phases 2, 4, 6, 7) and isolation levels of a sample use case

IV. Blind Overwriting Problem in the Application Context

Let us first consider the following problematic scenario of two concurrent processes A and B, each running a SQL transaction that is updating the balance of the same account, as shown in Table I. In this scenario step 4 of process A is empty.

The withdrawal of 200 € made by the transaction of process B will be overwritten by process A; in other words the update made by B in step 5 will be lost in step 7 when the transaction of A overwrites the updated value by value 900 € which is based on stale data, i.e., an outdated value of the balance, from step 3. If the transactions of A and B are serialized 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, assumed an isolation level of READ COMMITTED which is the default isolation level for most relational DBMS because of performance reasons. In READ COMMITTED isolation level the shared locks provided for the SELECT statement are released immediately after the completion of the statement and therefore it is possible for process B to obtain a lock and update the balance. So, READ COMMITTED does not protect any data read by a transaction of getting outdated right after reading the value. Locking Scheme Concurrency Control (LSCC) could prevent conflicting access to data, but not at READ COMMITTED isolation level. The proper isolation level on LSCC systems to prevent a lost update of process B should be REPEATABLE READ or SERIALIZABLE, which would protect the balance value read in the transaction of process A from getting outdated

Table I
A BLIND OVERWRITING SCENARIO USING SELECT-UPDATE IN TRANSACTION A

step	process A [T_1, T_2]	balance	process B
1	SET TRANSACTION ISOLATION LEVEL READ COMMITTED		
2		1000€	
3	SELECT BALANCE INTO :BALANCE FROM ACCOUNTS WHERE ACCTID = :ID;		
4	[COMMIT WORK;]		
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€	

during the transaction by holding shared locks on these rows up to the end of the transaction. As result the shared locks of Process A would block process B from a too early update. The isolation service of the DBMS guarantees that the transaction will either get the requested isolation level, or, in case of a serialization conflict, the transaction will be rejected by the DBMS. The means used for this service and the transactional outcome for the very same application code can be different when using different DBMS systems, and even in using different table structures. The LSCC may wait with granting a lock request until the possible conflict disappears. Usually a transaction rejected due to a serialization conflict should be retried by the application, but we discuss this later in Section VI.

In the second scenario of Table I, process A splits its transaction into two transactions. The first transaction, let it call T_1, consists of steps 1 to 4, including the COMMIT WORK at step 4. After some user interaction and the calculation in step 5, another transaction T_2 continues with steps 7 and 8. In this case, no isolation level can help, but transaction T_2 will make a blind write based on stale data from step 3. But, meanwhile the balance value was updated by transaction T_B of process B in step 5.

From the database perspective, the 3 transactions have been serialized correctly in the order: (T_1, T_B, and T_2). However, the problem is that there is no transaction boundary around T_1 and T_2; they are treated separately by the transaction manager. Hence, it is absolutely correct for T_B to interleave with T_1 and T_2. From a business or user point of view, especially the user that runs $_1$ *and* T_2, this is a semantically wrong behavior. Hence, as soon as a transaction is split into several sessions (e.g., due to connection pooling) or different SQL transactions, a transaction manager at the middleware layer is required that prevents from blind overwrites and provides one transactional context for T_1 and T_2. RVV provides such a context in a transparent way and the row verification ensures only consistent writes.

V. SQL Access Patterns for Avoiding Blind Overwriting

The blind write of the update transaction T_2 of steps 7 - 8 of Table I, which results in the loss of transaction T_B, could have been avoided by any of the following practices. The proposed access patterns assure that either before or during the write phase (step 7), a validation takes place and data will only be updated after a successful validation (see Figure 3). We present the patterns in the canonical form given by Coplien [27] that appears to be more compact than the one used by Gamma et al [28]:

A. Access Pattern: Sensitive UPDATE

Problem: How to prevent a blind overwrite in case of concurrent updates?

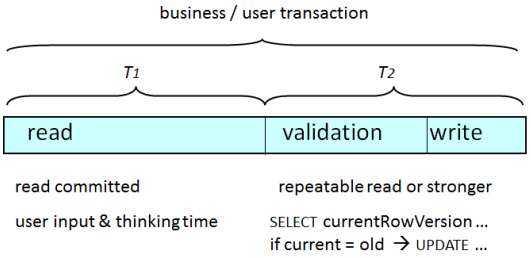

Figure 3. Context of the OCC Access Patterns

Context: Concurrent transaction processing in distributed systems has to deal with temporary disconnected situations, or sequences of SQL transactions belonging to one business transaction and nevertheless ensure correct results.

Forces:

- Using locks (LSCC) to prevent other transactions from changing values can block data items for unpredictable time in case of communication failure or in case of long user thinking time.
- Multiversion concurrency control (MVCC) or OCC do not block data access, but lead to abort conflicting transactions, except for the first one that updates the data.
- OCC is not supported by the mainstream commercial SQL databases, hence we cannot directly rely on the CC mechanism provided by the DBMS.

Solution: There is no risk of blind overwriting if T_2 in step 7 uses the form of the update which is sensitive to the current value, like B uses in step 5 as follows:

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

Consequences: Please note, the update of the balance is based on a value unseen by the application. Therefore, the user will not be aware of the changed balance and this access pattern does not provide repeatable read isolation. If the user needs to know about the changed situation the access pattern "RE-SELECT .. UPDATE" could be used (see Subsection V-C) or a SELECT command could be used after the UPDATE.

B. Access Pattern: Conditional UPDATE

Problem: How to prevent a blind overwriting and provide repeatable-read for a business transaction in case of concurrent updates without using locking?

Context: The "Sensitive UPDATE" pattern in concurrent read situations may result in non-repeatable phenomenon.

Forces: Same as for "Sensitive UPDATE" plus:

- The data value on which the update is based, as read and displayed to the user, may no longer be the same (non-repeatable read phenomenon).

Solution: After transaction T_1 first has read the original row version data in step 3, transaction T_2 verifies in step 7, using an additional comparison expression in the WHERE clause of the UPDATE command, that the current row version in the database is still the same as it was when the process previously accessed the account row. For example,

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

The comparison expression can be a single comparison predicate like in the example above where rowVersion is a column (or a pseudo-column provided by the DBMS) reflecting any changes made in the contents of the row and :old_rowVersion is a host variable containing the value of the column when the process previously read the contents of the row. In the case that more than one column is involved in the comparison, the expression can be built of version comparisons for all columns used. The comparison will be based on the 3-value logic of SQL.

Consequences: Since this access pattern does not explicitly read data, there is no need to set the isolation level. The result of the concurrency control services is the same for locking scheme concurrency control (LSCC) and multiversion concurrency control (MVCC) based database systems. The result of the update depends on the row version verifying predicate, and the application code needs to evaluate the return code to find out the number of updated rows to verify the result.

C. Access Pattern: Re-SELECT .. UPDATE

Problem: How to provide a repeatable-read isolation for a business transaction in case of concurrent updates? How to inform the user when the read set has changed and take an alternative decision?

Context: In case the result of the "Conditional UPDATE" pattern can not be directly communicated to the user. For example, when an application is using the database via middleware services (see Section VIII).

Forces: Same as for "Conditional UPDATE" plus:

- In the time between the re-SELECT and the UPDATE statement, the data read may be updated again by concurrent transactions. This serialisation conflict would force the transaction to rollback.

Solution: This is a variant of the "conditional UPDATE" pattern in which transaction T_2 first reads the current row version data from the database into some host variable current_rowVersion which allows the application to inform the user of the changed situation and take an alternative decision:

```
SELECT rowVersion
INTO :current_rowVersion
FROM Accounts
WHERE acctId = :id;
if (current_rowVersion = old_rowVersion)
then
   UPDATE Accounts
   SET balance = :newBalance
   WHERE acctId = :id;
else
   // inform the user if desired
   // and/or take an alternative decision
end if
```

To avoid any concurrency problems in this phase, it is necessary to make sure that no other transaction can change the row between the SELECT and the UPDATE. For this purpose, we need to apply a strong enough isolation level (REPEATABLE READ, SNAPSHOT, or SERIALIZABLE) or explicit row-level locking, such as Oracle's FOR UPDATE clause in the SELECT command.

Consequences: Since isolation level implementations of LSCC and MVCC based DBMS are different, the result of concurrency services can be different: in LSCC based systems the first writer of a row or the first reader using REPEATABLE READ or SERIALIZABLE isolation level will usually win, whereas in MVCC based systems the first writer wins the concurrency competition. On server side OCC the first one to commit wins, and in the event of LSCC deadlocks the victim (the transaction that is aborted) is determined by internal rules of the DBMS.

VI. A Basic Use Case Example

In the following scenario we will distinguish *SQL transactions* from *business transactions* (also called user transaction) as defined in Section I. An SQL transaction is known to the DBMS. It starts explicitly with a BEGIN TRANSACTION statement or implicitly with the first SQL statement after the last transaction. The SQL transaction terminates either with COMMIT or ROLLBACK. A business transaction in our case consists of a finite sequence of SQL transaction that are treated as a logical unit of work. The involved database system is unaware of this logical unit. Therefore, the databases can not support the atomicity of a business transaction. If the business transaction maps one-to-one to an SQL transaction as in legacy applications the DBMS can fully support the transactional properties.

In modern, web-based transactional applications a business transaction consists of multiple SQL transactions. This is not only because multiple database systems are involved, there is a technical reason too. Application servers use connection pooling, so even if only one database system is used, different SQL statements may belong to different connections and consequently to different transactions.

We see that the concurrency scope of an application needs to be extended to cover sequences of SQL transactions (or more generally server-side transactions), implementing some business transaction. Figure 2 presents a typical business transaction in 6 phases containing three SQL transactions (list, select, and edit/book) like the flight booking mentioned before plus an optional compensation phase. The ideal isolation levels listed for each SQL transaction depends on the concurrency control provided by the DBMS. For example, the default concurrency control mechanism on SQL Server is locking (LSCC), but it can alternatively be configured to use ”snapshot” isolation (MVCC). With RVV we refer to the sequence of inter-related SQL transactions (phases 4 and 6 in the Figure 2), which may belong to the same SQL connection, but typically in a Web application could belong to different connections as explained above. In this case the locks from Phase 4 cannot be held until Phase 6.

To make sure that no concurrent transaction has changed the contents of the row fetched in Phase 4, we need to verify that the content in the database is still the same when trying to update the row/object in Phase 6. Otherwise, the update will cause a blind write that overwrites the result from other competing transactions, thus loosing data from the database.

If the DBMS reports a concurrency conflict in Phase 6 (the write/update phase), the application may retry the statement because some conflicts are of transient nature. Temporary conflicts that disappear after the conflicting transactions have terminated could result either from an active deadlock prevention, from transactions terminated with ROLLBACK, or from released locks. A RVV validation that fails is unrepeatable as the version value is never decremented.

A committed transaction cannot be rolled back, but some systems provide a compensation transaction that reverses the effects of a previously successful transaction. This is like cancelling an order or contract that in fact results in a new order or contract that reverses the previous one.

For setting up the scenario on the SQL Server, the following Transact-SQL commands could be used:

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

For technical details of the above script the reader is referred to the SQL Server online documentation [29].

VII. Baseline RVV Implementations using Call-Level API

The first open database call-level interface and de facto standard for accessing almost any DBMS is the ODBC API specification which has strongly influenced the data access technologies since 1992. The current specification is almost the same as the SQL/CLI standard of ISO SQL. Many class libraries have been implemented as wrappers of ODBC and many data access tools can be configured to access databases using ODBC. Based on pretty much the same idea, Sun has introduced the JDBC interface for Java applications accessing databases which has become an industry standard in the Java world. Using the previously defined SQL views for accessing table VersionTest and applying the RVV discipline, the following sample Java code for Phase 6 (the update phase) of Figure 2 reveals the necessary technical details:

```
 // *** Phase 6 - UPDATE (Transaction) ***
con.setAutoCommit(false);
// Pattern B update - no need to set isolation level
string sqlCommand = "UPDATE rvv.RvTest " +
"SET s = ? " +
"WHERE id = ? AND rv = ? ";
pstmt = con.prepareStatement(sqlCommand);
pstmt.setString(1, newS);
pstmt.setLong(2, id);
pstmt.setLong(3, oldRv);
int updated = pstmt.executeUpdate();
if (updated != 1) {
throw new Exception("Conflicting row version in the
database! " );
}
pstmt.close();
// Update succeeded -> The application needs to know
the new value of RV
sqlCommand = "SELECT rv FROM rvv.RvTest WHERE id =
?";
pstmt = con.prepareStatement(sqlCommand);
pstmt.setLong(1, id);
ResultSet rs = pstmt.executeQuery();
newRv = rs.getLong(1);
rs.close();
pstmt.close();
```

```
con.commit();
```

In the above, as in all following examples, it is assumed that the version attribute `rv` will be maintained by the database itself, e.g., by a row level trigger or some pseudo-column mechanism as described in the following Subsection VII-A. If the database has no such capability, every application itself has to take care of incrementing the version on every update. If legacy applications do not follow this convention of incrementing the version, they are subject to lose their transactions.

Every 4-5 years Microsoft has introduced a new data access technology after ODBC, and in the beginning of this millennium ADO.NET built on various already existing data providers. Compared to Microsoft's ADO it is a new data access design close to JDBC, but simplified and extended. Instead of providing a universal interface to all kind of data sources it consists of a family of data models which can be generic like OleDb or native providers like SqlClient for SQLServer or OracleClient for Oracle. Each of these implement their own object classes. Without providing details about this rich technology we just show below the Phase 6 of Figure **??** from our baseline implementation of RVV using C# language and the native .NET Data Provider (SqlClient) [30] to access SQL Server 2008:

```
SqlCommand cmd = cn.CreateCommand(); //connection
creates new command object
// Phase 6 - update transaction
txn = cn.BeginTransaction();
cmd.Transaction = txn; // bind cmd to transaction
// Pattern B update including reread of rv using
OUTPUT clause of T-SQL:
cmd.CommandText = "UPDATE rvv.RvTest " +
"SET s = @s OUTPUT INSERTED.rv " +
"WHERE id = @id AND rv = @oldRv ";
cmd.Parameters.Clear();
cmd.Parameters.Add("@s", SqlDbType.Char, 20).Value =
newS;
cmd.Parameters.Add("@id", SqlDbType.Int, 5).Value =
id;
cmd.Parameters.Add("@oldRv", SqlDbType.BigInt,
12).Value = oldRv; // retrieved in step 4
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();
```

The above code-snippet shows how to bind the SQL command `cmd` to the controlling transaction object `txn`. The SQL Server provides an elegant solution for reading the current row version `rv` at the end of the SQL UPDATE command. Using the OUTPUT clause the Transact-SQL UPDATE command retrieves the new value when the SQL UPDATE is executed with the call to `ExecuteScalar()`.

A. Server-side version management

There are multiple options for verifying the row version. These include the comparison of the original contents of all or some relevant subset of column values of the row, a check-sum of these, some technical pseudo-column maintained by the DBMS, or an additional technical SQL column. In the latter case it is the question how the values of this column are reliably maintained.

A general solution for row version management is to include a technical row version column `rv` in each table as defined in the following example:

```
 CREATE TABLE Accounts (
acctId INTEGER NOT NULL PRIMARY KEY,
balance DECIMAL(11,2) NOT NULL,
rv BIGINT 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. The row-level UPDATE trigger is defined in SQL language as follows:

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

We call the use of a trigger or a DBMS maintained technical pseudo-column as *server-side stamping* which no application can bypass, as opposed to *client-side stamping* using the SET clause within the UPDATE command - a discipline that all applications should follow in this case. Row-level triggers are affordable, although they lead to lower performance and hence to approximately 2% higher execution time on Oracle and DB2 [22, p. 28], whereas SQL Server does not even support row-level triggers.

Timestamps are typically mentioned in database literature as a means of differentiating any updates of a row. However, our tests [22] show that, for example, on a 32bit Windows workstation using a single processor Oracle 11g can generate up to 115 updates having the very same timestamp. Almost the same problem applies to DATETIME of SQL Server 2008 and TIMESTAMP of DB2 LUW 9, with exception of the new ROW CHANGE TIMESTAMP option in DB2 9.5

which generates unique timestamp values for every update of the same row using the technical TIMESTAMP column.

The native TIMESTAMP data type of SQL Server is not a timestamp but a technical column which can be used to monitor the order of all row updates inside a database. We prefer to use its synonym name ROWVERSION. This provides the most effective server-side stamping method in SQL Server, although, as a side-effect it generates an extra U-lock which will result in a deadlock in the example at step 6 of Figure 2. The remedy for this deadlock is to use the table hint UPDLOCK which will block new U-lock requests and prevents the transactions from running into a deadlock.

In version 10 and later versions, Oracle provides a new pseudo-column, the Oracle Row System Change Number (ORA_ROWSCN) for rows in every table created with the ROWDEPENDENCIES option [31]. This will show the transaction's System Change Number (SCN) of the last committed transaction which has updated the row. This provides the most effective server-side stamping method for RVV in Oracle databases, although as a harmful side-effect, the row-locking turns its value to NULL for the duration of the writing transaction.

VIII. RVV Implementations using ORM Middleware

Data access patterns solving the impedance mismatch between relational databases and object-oriented programming are called Object-Relational Mappers (ORM) [8]. One widely known ORM technology is the Container Managed Persistence (CMP) pattern of the Java Persistence API (JPA) as part of the Java Enterprise Beans 3.0 (EJB3). The JPA specification assumes the use of "optimistic locking" [20].

The JPA stimulated the market for sophisticated persistence managers providing object-relational mappings, such as TopLink [32] and Hibernate [33]. Figure 4 shows our interpretation of the alternatives of the current Hibernate framework which implements the JPA but also allows full control over Hibernate's Core down to the JDBC code level, which we actually need for our RVV implementation when using Hibernate. In terms of RVV we are mainly interested in the object persistence services of ORM frameworks. As examples for these services Hibernate Core and Hibernate JPA providers are tested for their ability to implement RVV.

A. RVV implementation using Hibernate Core

Hibernate provides an optimistic concurrency mechanism called "optimistic locking" (described in the Hibernate Reference Documentation [34]) based on version control. This service can be configured programmatically and may be overwritten by XML-based configuration files.

For instance, the programming paradigm for persistent classes can chose any of the following options

- version checking by the application, e.g., RVV validation

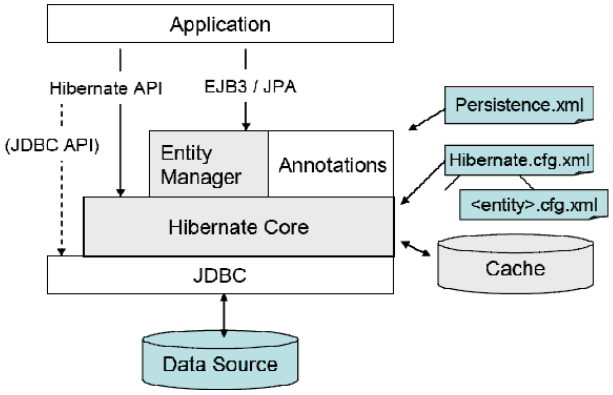

Figure 4. Hibernate architecture

- automatic version checking by an entity manager
- automatic version checking of detached objects by an entity manager

Automatic version checking takes place for every instance of the class during the transaction's COMMIT phase based on a technical version column when the attribute setting is `optimistic-lock="version"`. As alternative, Hibernate provides a validation based on a set of columns when setting the attribute to `optimistic-lock="all"` which will compare the contents of all columns, or by `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="rvvtest" table="RVTest" ...
        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 on insert and update, whereas the attribute *generated="never"* means that Hibernate will generate the value during synchronizing the contents with the database. The drawback of the validation based on Hibernate's generated technical column is that it is not reliable in case the data gets updated by some other software.

Another configurable behaviour of the data access is the SQL Isolation Level, which unfortunately cannot be changed for a single transaction. But exactly this capability is needed for Phase 6 of our example scenario. The original Hibernate engine, called Hibernate Core, provides a native interface providing the needed low level capability. Hibernate Core

services allow direct JDBC access to the data sources. Switching to the JDBC level involves the creation of a new connection on JDBC level and a subsequent method call to set the isolation level. See comments 1) and 2) in the example Java code of Phase 6.

Hibernate, like Toplink, tries to optimize data access performance using its own cache, that makes row version verification difficult. One must bypass the cache when fetching the current row version from the data source. Switching to the JDBC level allows to reload the RVV entity including the row version which bypasses Hibernate's cache mechanism. This is done in the example Java code using the `refresh(re2)` method on session level.

The use case scenario (see Figure 2) needs at least isolation level REPEATABLE READ in Phase 6. 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. We see this as a challenge and want to prove that ORA_ROWSCN can be used as row version field managed at server-side without Hibernate's optimistic locking services. It should be pointed out that the maintenance of the ORA_ROWSCN is done by the DBMS and cannot be bypassed by any application (including those not using Hibernate).

Hibernate requires a class definition with member variables matching the table columns of our view RVTEST. Objects of this class act as a wrapper for rows retrieved from or written to the view. The following XML-file defines the mapping between the RvvEntity class and the RVTest table for our scenario:

```
...
<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 the column *RV* is actually the ORA_ROWSCN pseudo column, we don't allow Hibernate to update it. The code portion of the critical Phase 6 shows that special tuning is needed to make Hibernate Core work correctly with Oracle (numbers refer to the comments in the code):

1) The default isolation level of 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 the possibility to change the isolation level dynamically, so we need to switch first to the level of JDBC services.
2) REPEATABLE READ would be the 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) The `save()` method used to store the modified entity back to the database allows no conditional update, i.e., only SQL Pattern C is applicable.

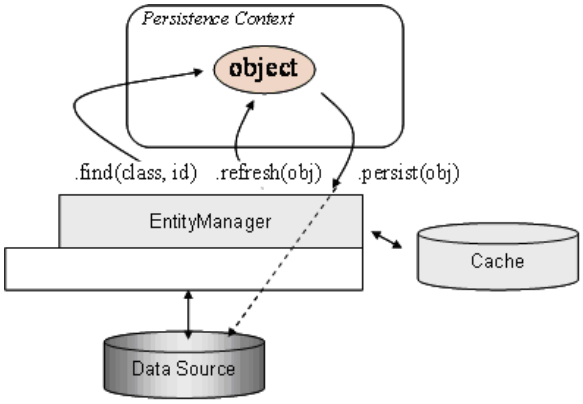

Figure 5. JPA persistence management

```
// Phase 6 - "model"
try {
tx = session.beginTransaction();
Connection conn = session.connection(); // 1) switch
to 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)) { // 3) Pattern C
   re2.setS(s);
   session.save(re2); // 3)
   /* Programmed breakpoint for concurrency testing:
*/
   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);
}
```

B. RVV implementation using Hibernate JPA

Hibernate provides now its Java Persistence API (JPA) implementation (EntityManager and Annotations) as a wrapper of Hibernate Core. Figure 5 presents methods of JPA for managing the persistence of an object.

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

The following Java code sample from our RVV Paper [22] shows how to avoid stale data from Hibernate's cache. To set the isolation level via JDBC we first need to switch to Hibernate's core level as in the previous example. This is done from the underlying session object at the JDBC level. The session object creates a new connection, the connection `conn` begins a new transaction and sets the isolation level to SERIALIZABLE. Then, the object is reloaded and the actual *newRv* is read. The used *Pattern C* requires REPEATABLE READ to ensure that the row version will not change during validation and execution of the update. But, for portability reasons we choose the stronger isolation level SERIALIZABLE.

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

Apart from different method names, Hibernate API and JPA provide approximately the same abstraction level and in both cases it is necessary to use JDBC level access to ensure the appropriate control over the SQL isolation level. To circumvent Hibernate's cache we need to either refresh the object with the session `refresh()` method (Hibernate API) or clear the entity manager's cache with the `clear()` method (Hibernate JPA).

IX. RVV Implementation using LINQ to SQL

Microsoft's answer to the ORM frameworks is Language Integrated Query (LINQ) for the .NET Framework 3.5. The class libraries of LINQ can be integrated as part of any .NET language providing developer "IntelliSense" support during coding time and error checking already at compile-time [35]. So called "standard query operators" of LINQ can be applied to different mappings using LINQ providers, such as LINQ to XML, LINQ to Datasets, LINQ to Objects, and LINQ to SQL. In the following C# code sample the object *myRow* is loaded from the database in Phase 4 and string *newS* contains a new value entered in Phase 5. In Phase 6 our use case enters the update phase, first the string *newS* is assigned to myRow's member variable *S* and then the changes are submitted to the DataContext *dc*. The DataContext object holds the open connection to the database in order to finally synchronize the object *myRow*'s data. The shaded part of the code is just a programmed break allowing concurrent processing for concurrency tests:

```
// Phase 4 - data access
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 { myRow.S = newS;
   // To allow time for concurrent update tests ...
   Console.Write("Press ENTER to continue ..");
   Console.ReadLine();
    dc.SubmitChanges(ConflictMode.FailOnFirstConflict);
    txs.Complete();
  }
  catch (ChangeConflictException e) {
    Console.WriteLine("ChangeConflictException: " +
      e.Message);
  }
```

```
  catch (Exception e) {
   Console.WriteLine("SubmitChanges error: " +
   e.Message + ", Source: " + e.Source +
   ", InnerException: " + e.InnerException);
  }
 }
```

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

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

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

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

The SQL UPDATE statement validates the row version with the `([RV] = @p2)` predicate. This is exactly our *Pattern B*. The following SELECT statements reads the new row version and ensures that the row count is greater than 0 `((@@ROWCOUNT) > 0)`. In case of a version conflict no row is updated and a `ChangeConflictException` is returned. There is no need to implement any RVV pattern on the application level as LINQ to SQL applies this pattern automatically.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="model">
   <class name="RvvEntity">
    <extension vendor-name="FastObjects" key="index"
        value="IdIndex">
            <extension vendor-name="FastObjects"
key="member"
        value="id"/>
    </extension>
   </class>
  </package>
</jdo>
```

Figure 6. Example XML file defines persistence capable class RvvEntity

X. RVV Implementation using JDO Object Database

Java Data Objects (JDO) is a vendor neutral programming interface that enables the persistent storage of Java objects developed in 2001. Meanwhile the JDO specification and reference implementation is maintained by the JDO Apache project under the Java community process and released its version 3 in April 2010. JDO has strongly influenced the definition of the Java Persistence API (JPA).

The programming model provides a transparent, easy to use object persistence for standard Java objects including transactional support. For this reason, vendors of object oriented databases quickly adopted JDO as programming interface. However, it is possible to implement the JDO API for any persistent data storage (called *datastore* by the JDO specification), in particular for relational database systems. In contrast to ORM software there is no need to define any transformation to tables. The programmer only needs to specify in an XML file the classes that he wants to make persistence capable and optionally - as an vendor extension - the attributes that should have indexed access (as in example Figure 6). This definition will be later used by the class enhancer to make ordinary Java classes persistence capable and the actual mapping - if applicable - is hidden from the programmer.

While in JDO version 1 the mapping was undefined, version 2 allowed different mappings to relational databases. If one starts with the class definition the relational schema can be generated from the optional XML-mapping file.

The JDO Specification 2.2 [36, pp 44-46] distinguishes three types of identity:

- Application identity - based on attribute values, managed by the application, and enforced by the database,
- Datastore identity - based on a system generated identity and managed by the database,
- Nondurable identity - the Java object identity, managed

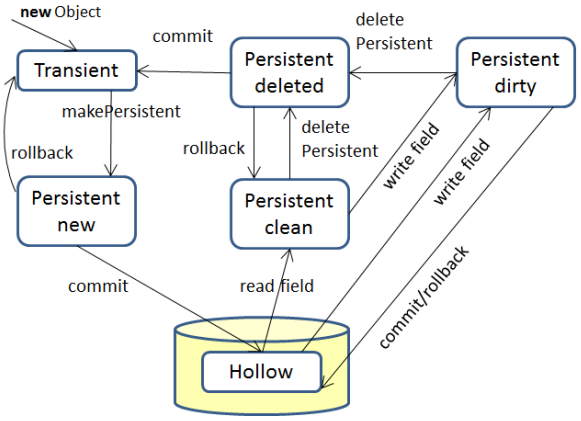

Figure 7. JDO persistence capable object states

by the Java virtual machine.

Only the datastore identity is a persistent identity in the object oriented sense. The representation of the identity is via the JDO object identity that is returned as a copy by method getObjectId(Object po). JDO defines two types of transaction management, the Datastore Transaction Management (DTM) and optimistic transaction management (OTM). DTM ensures transactional properties similar to SQL transactions even if connections to the datastore are dynamically handled by the ConnectionManager. In the case of OTM, persistent objects modified within or outside a transaction are not transactional until during commit. At commit time a short transactional datastore context is established in which all modifications are flushed to the datastore if validation is successful. If an `JDOOptimisticVerificationException` is raised, the transaction fails and in memory modifications are rolled back (the original state is restored). The tested implementation FastObjects J1 does not support the OTM and raised an `JDOUnsupportedOptionException`.

The programming model of JDO version 1 distinguishes between transient and persistent objects. The persistent objects may be in one of five states:

- persistent new - new object prepared to be committed to the datastore
- hollow - object in the datastore
- persistent clean - (partial) object loaded into application
- persistent dirty - (partial) object loaded and modified in application
- persistent deleted - persistent object deleted

The possible states and transitions for transactional objects are shown in Figure 7. Typically an object is created in the application by the Java constructor **new**. The resulting object is *Transient*. If the object's class is persistence capable, calling `makePersitent()` changes its state to *Persistent new*. If the object is committed to the datastore its state is *Hollow*. After reading some fields of an object from the datastore it reaches the state *Persistent clean*. Modifying any attribute value makes the object *Persistent dirty* until it is committed back to the datastore or the values are discarded by the `rollback()` method. In both cases the resulting state is *Hollow* again. A persistent object may be deleted at any time by calling `deletePersistent()`. The deletion is made durable by calling `commit()` or cancelled by `rollback()`.

In JDO version 2, *nontransactional* and *detached* operations were added that are not orthogonal to the above states. This leads to an explosion of states and different life cycles for JDO objects (see JDO 2.2 Specification [36, pp. 50 - 68].

JDO recognizes four isolation levels with identical names as the SQL isolation levels plus the snapshot isolation level. It is unclear if these isolation levels have the same semantics as the isolation levels for relational databases. The tested FastObjects J1 does not support this JDO 2.0 options, but uses strict two-phase-locking that ensures REPEATABLE READ, the isolation level needed for the RVV testing.

The class definition of `RvvEntity` should be stored in the object database as well in order to provide the class definition to all applications. The version attribute `rv` will only be incremented by the setter methods of the other attributes. It is important that all attributes are private and that the `rv` attribute has no setter method. So the only way to access an attribute is via its getter and setter methods. The setter methods ensure that any modification to the object's state, i.e., changing any attribute value, will result in a new row version. The setter method for each attribute should be modified in the following way:

```
package model;
public class RvvEntity {
  private int id;
  private String s;
  private long rv;
  public RvvEntity() {// default constructor
  } //needed for persistence capable classes
  public RvvEntity(int id, String s) {
     this.id = id;
     this.s=s;
     this.rv = 0;
  }
  public void setS(String s) {
     this.s = s;
     this.rv++;
  }
}
```

As in the previous Hibernate examples, only loaded objects can be modified. Therefore, only Pattern C is applicable as the following listing of Phase 4 and 6 shows.

```
PersistenceManager pm;
...
```

```
public void doRVV() throws ExampleException {
  RvvEntity p, p2RVV;
  Object p2oid;
  Extent e;
  Iterator<?> i;

  // Phase 4 of use case
  pm.currentTransaction().begin();
  e = pm.getExtent(RvvEntity.class, true);
  i = com.poet.jdo.Extents.iterator(e, "IdIndex");
  // find object with id = 1
  boolean found = com.poet.jdo.Extents.findKey(i,
1);
  // search for the first key match
  if (found == true && i.hasNext())
    p = (RvvEntity) i.next();
  else {
    System.out.println("RvvEntity #1 not found");
    return;
  }
  p2oid = pm.getObjectId(p); //1)
  long oldRvv = p.getRv(); //2)
  pm.currentTransaction().commit();

  // Phase 6 of use case
  pm.currentTransaction().begin();
  pm.currentTransaction().setIsolationLevel(
  TransactionIsolationLevel.repeatable-read);
  System.out.println("pm: Phase 6 started");
  p2RVV =
  (RvvEntity) pm.getObjectById(p2oid, true); //3)
  if (p2RVV.getRv() == oldRvv){ //4) Pattern C
    p.setS(" pm(Phase 6): NEW TEXT");
    try {
      pm.currentTransaction().commit();
      System.out.println("pm: Phase 6 committed!");
    }
    catch (javax.jdo.JDOException x) {
      pm.currentTransaction().rollback();
      System.out.println("Could not commit! Reason:
"
      + x.getMessage());
    }
  } else {
    System.out.println("pm: RVV serialisation
conflict!");
    pm.currentTransaction().rollback();
  }
}
```

In Phase 4 the RvvEntity #1 is retrieved from the database by its id number. From the programming model the index search could possibly retrieve more than one object with id = 1. It is the responsibility of the application to ensure uniqueness. Later, when the object is loaded, the method `getObjectId()` retrieves its persistent datastore object id (line with comment 1). This object id is later used in Phase 6 to reload the object by its object id (see line with comment 3). The row version from Phase 4 is saved in variable `oldRvv` (comment 2) and used in Phase 6 for RVV (comment 4).

Phase 6 relies on isolation level REPEATABLE READ, but the tested version of FastObjects does not support isolation level setting. From the exception messages we concluded that any read access was protected by a shared lock until the end of the transaction and conflicting implicit lock requests (e.g., by a setter method) resulted in an exception with message:

> write lock for object with id (0:0-1030#0, 1001) on behalf of transaction <unnamed> on database FastObjects://LOCAL/MyRvv_base.j1 not granted

The apparent use of strict two-phase-locking provided reliable isolation for a successful application of the RVV using Pattern C. During Phase 4, no competing transaction could modify the loaded object. Between phases 4 and 6 a concurrent transaction may successfully update objects loaded during Phase 4 but in Phase 6 the changes are discovered and the transaction under test had to abort. During Phase 6 the transaction under test was protected by read locks against any changes from the re-read (line with comment 3) until the end of the transaction.

XI. Conclusion

Table II presents a comparison of the Call-level APIs and ORM Frameworks with RVV practice in mind. It lists the access patterns that can be used in conjunction with different technologies and it depicts the level of control and its limitations. Major findings are the differences when applying the access patterns of Laux and Laiho [11] for different middleware technologies with regard to isolation levels, transaction control, caching, and performance overhead. While we are writing this paper LINQ to SQL is still in its beta phase and it was rather slow in our tests. However, we are impressed about the built-in ”optimistic concurrency control” as Microsoft calls it. Microsoft has the advantage of experiences from the competing technologies. Attributes of LINQ are more orthogonal than the numerous JPA annotations and its object caching did not produce side-effects in concurrency control making LINQ easier to use and manage. It also utilizes server-side version stamping. With the advanced features of the framework - as it proves to do things right - this is a most promising software development extension in the .NET Framework. The native DBMS for LINQ is currently SQL Server, but since IBM and Oracle have already shipped its own ADO.NET data providers, their support for this technology can be expected.

Table II
COMPARISON OF CLI APIS AND ORM FRAMEWORKS

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

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

As an advantage of ORM Frameworks Hoffer et al [26] lists "Developers do not need detailed understanding of the DBMS or the database schema". We don't share this view. Even if we use these higher abstraction frameworks we need to verify that we understand the low level data access operations so that our applications are transaction safe. For example it was necessary to circumvent middleware caches where possible or when using disconnected data sets we had to explicitly reread the row version from the database in repeatable read isolation (access *Pattern C*). The version stamping of the "optimistic locking" should not be handled at client or middleware side, but on the server side instead to avoid that applications can bypass the RVV mechanism.

The JDO programming interface shielded much of the mapping complexity and the implementation tested used straight forward strict two-phase-locking. So the behaviour was similar to SQL databases with locking scheme. Future tests with products supporting optimistic transaction control and disconnected (called detached by JDO) operations will show if these models can improve performance or facilitate programming.

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

ACKNOWLEDGEMENTS

This paper is the result of collaborative work undertaken along the lines of the DBTechNet Consortium. The authors participate in DBTech EXT, a project partially funded by the EU LLP Transversal Programme (Project Number: 143371-LLP-1-2008-1-FI-KA3-KA3MP)

REFERENCES

[1] M. Laiho and F. Laux; "Implementing Optimistic Concurrency Control for Persistence Middleware using Row Version Verification," in *Second International Conference on Advances in Databases Knowledge and Data Applications (DBKDA 2010)*, April 11-16, 2010, Les Menuires, France, pp. 45 - 50

[2] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, pp. 287 - 317, Vol. 15, No. 4, December 1983

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

[4] M. Crowe; (2011, Jan.), The Pyrrho Database Management System, University of the West of Scotland, [Online], Available: http://www.pyrrhodb.com

[5] VoltDB, Inc. (2011, Jan.), VoltDB Home Page, [Online], Available: http://voltdb.com/

[6] VoltDB, LLC (ed.) (2011, Jan.), VoltDB Technical Overview, [Online], Available: http://www.voltdb.com/_pdf/VoltDBTechnicalOverviewWhitePaper.pdf

[7] G. Weikum and G. Vossen, *Transactional Information Systems*, Morgan Kaufmann Publishers, 2002

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

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

[10] U. Halici and A. Dogac; "An Optimistic Locking Technique for Concurrency Control in Distributed Databases," *IEEE Transactions on Software Engineering*, Vol 17, pp. 712 - 724, 1991

[11] F. Laux and M. Laiho; "SQL Access Patterns for Optimistic Concurrency Control," in *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns (COMPUTATIONWORLD '09)*, November 15-20, 2009 - Athens/Glyfada, Greece, pp. 254 - 258

[12] D. Z. Badal; "Correctness of Concurrency Control and Implications in Distributed Databases," in *Proc. COMPSAC79*, Chicago, 1979, pp. 588 - 593

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

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

[15] R. E. Stearns and D. J. Rosenkrantz; Distributed Database Concurrency Controls Using Before-Values," in *Proceedings ACM SIGMOD International Conference on Management of Data*, 1981, pp. 74 - 83,

[16] P. A. Bernstein and N. Goodman; "Multiversion Concurrency Control – Theory and Algorithms," *ACM Transactions on Database Systems* **8**, pp. 465 - 483, 1983.

[17] A. Seifert and M. H. Scholl; "A Multi-version Cache Replacement and Prefetching Policy for Hybrid Data Delivery Environments," in *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002, pp. 850 - 861

[18] M. Heß; (2010, Oct), Lange Gespräche mit Hibernate, [Online], Available: http://www.ordix.de/ORDIXNews/2_2007/Java_J2EE_JEE/hibernate_long_conversation.html

[19] Y. Akbar-Husain, (2010, Oct.), Optimistic Locking pattern for EJBs, [Online], Available: http://www.javaworld.com/jw-07-2001/jw-0713-optimism.html

[20] L. DeMichiel and M. Keith, *JSR 220: Enterprise JavaBeans™, Version 3.0, Java Persistence API*, Final Release, 8 May, 2006, Sun Microsystems, Inc., Santa Clara, California, USA,

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

[22] M. Laiho and F. Laux; (2011, Jan.), On Row Version Verifying (RVV) Data Access Discipline for avoiding Blind Overwriting of Data, [Online], Available: http://www.DBTechNet.org/papers/RVV_Paper.pdf

[23] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase; (2010, Oct.), The Java EE 5 Tutorial, [Online], Available: http://download.oracle.com/javaee/5/tutorial/doc/

[24] N.N.; (2010, Oct.), .NET Framework 3.5, msdn .NET Framework Developer Center, [Online], Available: http://msdn.microsoft.com/ en-us/library/w0x726c2.aspx

[25] M. Laiho and F. Laux; "Data Access using RVV Discipline and Persistence Middleware," in *The Conference for International Synergy in Energy, Environment, Tourism and Contribution of Information Technology in Science, Economy, Society and Education (eRA-3)*, 2008, Aegina/Greece, pp. 189 - 198

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

[27] J. O. Coplien, "A Generative Development-Process Pattern Language," in J.O. Coplien and D.C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995

[28] E Gamma, R Helm, R Johnson, and J Vlissides; *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[29] N.N.; (2010, Oct.), SQL Server Books Online, msdn SQL Server Developer Center, [Online], Available: http://msdn.microsoft.com/en-gb/library/ms130214.aspx

[30] N.N.; (2010, Oct.), *SQL Server and ADO.NET*, msdn Visual Studio Developer Center, [Online], Available: http://msdn.microsoft.com/en-us/library/kb9s9ks0.aspx

[31] Diana Lorentz and Mary Beth Roeser; (2011, Jan.), *Oracle SQL Language Reference Manual*, 11g Release 2 (11.1), October 2009, [Online], Available: http://download.oracle.com/docs/cds/E11882_01.zip

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

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

[34] Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, and Steve Ebersole; (2010, Oct.), Hibernate Reverence Documentation, Version 3.5.1-Final, [Online], Available: http://docs.jboss.org/hibernate/stable/core/reference/en/pdf/hibernate_ reference.pdf

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

[36] Craig Russell; (2010, Oct.), Java Data Objects 2.2, JSR 243, 10 October 2008, Sun Microsystems, Inc., [Online], Available: http://db.apache.org/jdo/releases/release-2.2.cgi