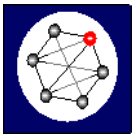

www.DBTechNet.org

On SQL Concurrency Technologies
- for Application Developers

With the support of the EC LLP Transversal programme of the European Union

Martti Laiho[1], Fritz Laux[2]

[1] Dpt. of Business Information Technology, Haaga-Helia University of Applied Sciences, Ratapihantie 13, 00520 Helsinki, Finland

[2] Dpt. of Informatics, Reutlingen University, Alteburgstraße 150, 72762 Reutlingen, Germany,

Disclaimers

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Trademarks of products mentioned are trademarks of the product vendors.

Advanced Single-Connection Transaction Models

The concept of simple flat transactions is not that simple after all, but already the early System R prototype RDBMS implemented more advanced transaction features such as *savepoints*, and even more advanced transaction models have been studied by researchers. Savepoints are now a proven working technology implemented in all mainstream DBMS systems of today, and some implementations of *nested transactions* exist. The idea of *saga* model has influenced SQL standard, but sagas and so called *long transaction*s are not really supported in the mainstream DBMS systems today, and trying to apply these in business critical OLTP applications is just bad SQL transaction design, of which we present sagas just as a warning example. The concept of long transaction doesn't even have an exact definition and the concept of *workflow* model is out of scope of the SQL transaction programming.

Transaction Savepoints

As a general rule the transactions should be designed to be as short as possible. However, some transactions may need many database actions and application logic. This transaction logic can include also multiple tentative parts, starting with named savepoint definitions and ending optionally with the rollback of all database modifications made after the named savepoint. According to the ISO SQL [11] a savepoint is defined by command of following form

```
SAVEPOINT <savepoint name>
```

and the partial rollback of database modifications made in the transaction after a named savepoint (including "nested savepoints", meaning the savepoint definitions after that) is done using the command

```
ROLLBACK TO SAVEPOINT <savepoint name>
```

The scope in which the defined savepoints are available is the active SQL transaction, but according to the standard it is also possible to destroy an existing savepoint and its nested savepoints in the transaction using a command

```
RELEASE SAVEPOINT <savepoint name>
```

which means that these savepoints are no more available for partial rollbacks.

All mainstream DBMS systems have a savepoint implementation, but all use syntax of their own. An interesting question is does a partial rollback release locks achieved for the actions which have been rolled back. According to our tests, rollbacks to savepoints don't release locks.

Nested Transactions

A popular and inspiring topic in research papers is the concept of nested transactions, and there exists different nested transaction models. Gray and Reuter [8] mention *sequential nested transactions* model as a synonym of savepoints, and *nested transactions with parallelism among atomic subtransactions* as another model of which there exists variants and even some implementations. Nested SQL transaction consists of root transaction and tree hierarchy of subtransactions in the scope of the root transaction using the same a single database connection. Leaf level transactions in the tree are flat transactions. Typically referenced nested transaction model in literature is Moss's model according to which only the leaf level transaction can access the database, but in the following we quote the model presented by Gray and Reuter [8] which allows actions on all levels of the hierarchy, and defines following behavior rules for the nested transactions:

Commit rule:	commit of a subtransactions makes its results available only to its parent transaction, but finally the results will be made durable in the database only when the root transaction commits.
Rollback rule:	if the root or a subtransaction is rolled back, then also results of its all subtransactions will be rolled back.
Visibility rule:	all objects available to a parent transaction are also available to its subtransations, but modifications made by a subtransaction will be visible to its parent transaction

ISO SQL standard does not define nested transactions, and the concept is not supported in the mainstream DBMS systems.

Only implementation of nested transactions is available in SQL Server has a simplified nested transaction model in which subtransactions cannot be rolled back. A rollback on any level will roll back the whole transaction. Commit of a subtransaction does not release any locks granted to the subtransaction. In fact, benefits of using SQL Server's nested transaction are not clear at all.

Instead of nested transactions described above Oracle has implemented the concept of **autonomous** subtransactions in stored procedures, functions and triggers. An autonomous transaction is fully independent of the parent transaction. Modifications made by committed autonomous transaction are durable even if the parent transaction will be rolled back. For details of autonomous transactions we refer to Oracle PL/SQL Language Reference manual [24].

Sagas of Chained Transactions with Compensating Transactions

As an alternative of using sequence of savepoints, a multi-step transaction can be split into a chain of transactions so that at commit or rollback phase a new transaction starts immediately with same transaction properties as the committed transaction without freeing the resources reserved by the previous transaction. For these purposes ISO SQL standard defines the special syntax for committing transactional steps in a chained transaction by

```
COMMIT [WORK] AND CHAIN
```

In case some transactional step in the transaction chain fails, there should be reverse chain of compensating transactions which should undo the committed transactions in the chain, and a ROLLBACK statement of failed transaction starts the compensating transaction chain by

```
ROLLBACK [WORK] AND CHAIN.
```

Sequence of chained transactions with corresponding chain of compensating transactions presented in Figure 12 is called saga.

Figure 12. Saga of chained transactions with compensating transactions

The idea of a compensating transaction, for example C1, is to programmatically "rollback" the committed work of the corresponding transaction T1. In a multi-user environment this cannot be guaranteed to succeed, since some competing transaction may have intervened and changed the results in the database. In general, chained transactions with compensating transactions are a model of **bad transaction programming design**, due to uncontrolled concurrency.

The current mainstream RDBMS systems don't support these syntaxes, but the concept of chained transactions is available using external TP monitors.

Best Practices for Transaction Programming

- Accessing OLTP databases should always be based on SQL transactions. You cannot rollback actions in autocommit mode!
- SQL transactions should be as short as possible and contain only interactions with the database. Especially user interaction has to occur outside SQL transactions..
- Define transaction isolation level at the beginning of every transaction. Don't trust on the default isolation. Consider carefully which is the reliable, but not too blocking isolation level for your transaction. The ISO SQL isolation level SERIALIZABLE is not always the best solution. Avoid writing on isolation levels less than ISO SQL REPEATABLE READ.
- Know the behavior of your DBMS. Verify the concurrency control service of your transaction against potential competing transactions. In case of read-write scenario, consider exclusive row-locking in the read operations using SELECT..FOR UPDATE, if it is supported.
- Performance is important, but **reliability is even more important**.
- Avoid blind overwriting of existing data (explained in our RVV Paper).
- To avoid deadlocks, it has been recommended that accessing resources in the database should follow the same order of rows trying to avoid circular waits of locks. The effect of accessing order can be tested using our bank transfer example in Appendix 5. However, this rule cannot always be applied.
- Transactions should be *idempotent* [8]. It means that transaction should not have any side effects in case of re-runs. Especially a transaction should not include any interactions with the user of the application.
- Even if RBMS systems allow dynamic changes in the database structures, DDL commands concurrently with transactions may lead to blocking since DDL commands update the system tables and the RDBMS need to protect these updates. SQL Server uses special schema locks to manage concurrency with DDL commands. Oracle does not allow DDL commands inside transactions.
- On writing stored procedures and triggers, consider that the isolation level of the invoking/firing transaction is the default isolation in your routine, and you don't know it generally.

Summary

In figure 13 we summarize, in free format, the major findings in this Concurrency Paper, relating serializability theory with ISO SQL standard and the sample of RDBMS products. ISO SQL standard defines isolation levels as abstract terms for concurrency control, based on the concurrency problems, which are supposed to be solved by the isolation levels, without regarding the actual concurrency control mechanisms.

Serializability theory is built on simplified read, write, lock and unlock operations resulting principles like two-phase locking (2PL) for achieving serializable executions of transactions.

As real concurrency control mechanisms in the mainstream database systems we have seen two basic implementations, namely single-version locking schemes for which we use the categorical name *locking scheme concurrency control* (LSCC) instead of name 2PL used in database textbooks, and *multi-version concurrency control* (MVCC). The third concurrency control mechanism is the server-side *optimistic concurrency control* mechanisms, implemented for example in Pyrrho DBMS.

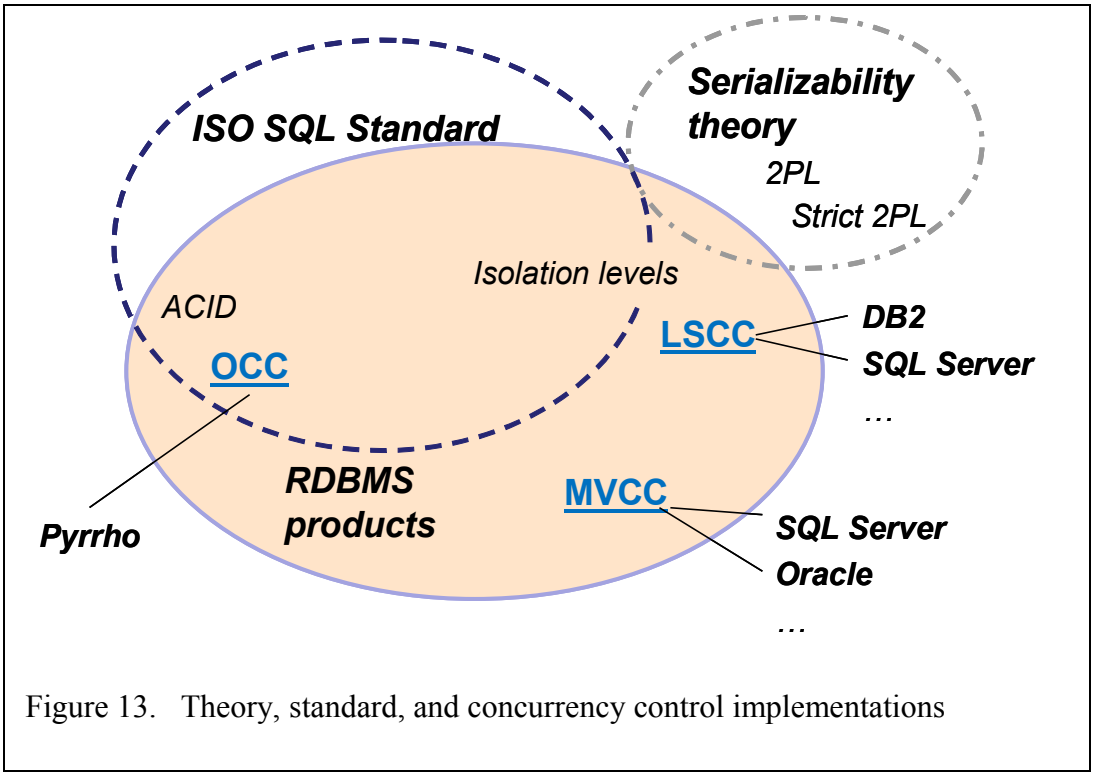

Figure 13. Theory, standard, and concurrency control implementations

The actual concurrency control mechanisms of the mainstream DBMS systems covered above allow also non-2PL behavior, for example updating after select while READ

COMMITTED isolation has been ordered from the DBMS – which, of course, may also lead to problems.

Concurrency:			Locking Scheme Concurrency Control (**LSCC**):			Multi-Versioning Concurrency Control (**MVCC**):		
Scope:	**Isolation:**	ISO SQL	DB2 V9.7	SQL Server 2008	Oracle 11g	SQL Server 2008 read committed snapshot isolation	SQL Server 2008 allow snapshot isolation	Oracle 11g
Cursor	read uncommitted		select .. with RU					
	Read Only (committed)			yes				
	Optimistic with values			yes				
	Optimistic with timestamp		(row change timestamp)	yes				
	Cursor Stability		CS	"Scroll Locks"				
	(X-locking of rows)		select .. with RS,RR		select.. for update			
	(WITH HOLD)	yes	yes		yes			
Transaction	Read Uncommitted	yes	RU	yes	N/A	yes 1)	yes 1)	N/A
	(Snapshot) Read Committed	N/A	CC		N/A	yes	N/A	"Read Committed"
	Read Committed	yes	CS	yes	N/A	N/A	yes 1)	N/A
	Snapshot	N/A	N/A	N/A	N/A	N/A	yes	"Serializable"
	Repeatable Read	yes	RS, RR 2)	yes	N/A	yes 1)	yes 1)	N/A
	Serializable	yes	N/A	yes	N/A	yes 1)	yes 1)	N/A

Table 8: Summary of Isolation Levels and Concurrency Control Implementations
1) based on locking scheme, so the database option is only semi-MVCC
2) RS holds locks for selected rows, whereas RR holds locks on all read rows preventing "update phantoms"

Table 8 summarises roughly the SQL isolation levels and concurrency control implementations in modern mainstream DBMS systems.

In their paper, Berenson et al [2] criticize the isolation levels of ANSI/ISO SQL in that the standard does not recognize the Cursor Stability isolation provided by DB2 and some other implementations, nor the Snapshot Isolation. Regarding the Cursor Stability (CS), the authors forget that, on *scope of transaction,* CS affects otherwise like Read Committed, but it will affect the behavior of cursors, and the standard does not state anything on isolation in *scope of single commands and cursor processing*, for which the

isolation on transaction level provides the default behavior. The Snapshot Isolation, implemented using MVCC, is not in line with the idea of database as collection of current / single-version facts, which is the ideology of DB2 SQL and so original ideology of the ISO SQL standard. MVCC has its advantages as a non-blocking protocol, but also problems of its own as shown by Berenson et al [2]

Even if locking should usually be left as the responsibility of the DBMS, occasionally it just is necessary to use explicit locking using row-level locking (Oracle), explicit table locking, table hints, or cursor level isolations.

What next?

In the next paper of this concurrency series, the RVV Paper [12] we will consider SQL transactions from application point of view, series of dependent SQL transactions building the data access for user transactions. The optimal isolation level may differ in SQL transactions of the same user transaction, and occasionally we may need explicit row-level locking. It is also possible that SQL transactions of the same user transaction don't share the same SQL session in terms of same physical database connection.

We have covered only reliability issues, but when we can manage with the reliability topics, the next question is performance. We will cover this in our DBTech **CCLab2**.

In this paper we have covered only transactions in a single database in a single connection (SQL session), which are also called *local transactions* apart from *distributed transactions* (which are also called as *global transactions)* accessing multiple databases and controlled by external transaction managers of application servers. Distributed transactions are the theme of our third paper in this concurrency series, introducing interesting topics, such as

- Two-Phase Commit (2PC)
- Global deadlock in heterogeneous distributed database
- More challenging performance questions.