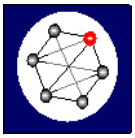

www.DBTechNet.org

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.

Concurrency Control Techniques

Traditionally database literature presents the following *concurrency techniques*

- *two-phase locking (2PL)*
- *timestamp ordering*
- *multi-version timestamp ordering and*
- *optimistic concurrency control*

based on various research papers without paying much attention to implementations in real DBMS products.

Instead of the purpose of research papers, our purpose in the following is to explain and guide you to verify the real techniques which are really implemented in the mainstream DBMS products used by the industry. For example, in this tutorial we consider 2PL and its variants as general principles which can be applied using LSCC, but not as a sole implemented concurrency control mechanism of some DBMS.

The isolation levels of ISO SQL standard are defined with such database systems in mind which store only the 'current facts' of the target world, in other words *single versions* of rows, and even if the standard does not say anything on the concurrency control technique to be used, all the isolation levels can be implemented using a *multi-granular locking scheme concurrency control* (which we call shortly LSCC), which in literature is often called *pessimistic* concurrency control (PCC).

Multi-Granular Locking Scheme Concurrency Control (LSCC)

Locking is the most common technique to solve concurrency problems in databases. Before the current era of relational databases, the concurrency control was based on locking and unlocking of single records by the applications, and the skill of writing proper locking plans was critical in database programming, since the early DBMS products were not even capable of detecting and solving the deadlock situations. A secure technique to avoid deadlock[1] was to conditionally reserve locks on all records, data sets and predicates needed before starting the actual transaction, and continue the transaction only if the locks were granted. At that time, the early theories of *transactions* and *serializability* were born and principles like *two-phase locking* (2PL) protocol were developed. In terms of locking, a transaction is called *well-formed* if it reads only data items for which it has acquired shared (read) locks and writes only data items for which it has acquired exclusive (write) locks. Accordingly, in the basic locking scheme for read operation a *shared lock* (S-lock) is requested to the target and for writing an *exclusive lock* (X-lock) is requested. Some locking schemes have implemented also an *update queuing lock* (U-lock), which depending on the command types can be granted to the first X-lock requester when X-lock cannot be granted immediately. The purpose of U-lock is to prevent deadlocks, but in the next paper in this series of concurrency papers, in the RVV Paper, we have an example of an SQL Server U-lock actually leading to a deadlock!

The requested lock is granted depending on existing locks on the target as described in the lock compatibility matrix of Table 4.

Lock requested:	Lock already granted to some other process			
	none	S	U	X
S	grant	grant	grant[2]	wait
U	grant	grant	wait	wait
X	grant	wait	wait	wait

Table 4. Basic lock compatibility matrix

[1] We will explain the deadlock concept later in this paper.
[2] In compatibility matrix presented by Gray and Reuter [7] p 408 U-locking is asymmetric: S-lock allows U-lock, but U-lock blocks S-lock requests. This would accelerate the updater to get X-lock. However, table 4 presents the way U-lock has been implemented in current DB2 and SQL Server editions, favoring reading transactions.

Lock for a S-lock request is not granted if someone else already has an X-lock on the target, otherwise an unlimited number of concurrent S-locks can be granted to the same target. X-lock can be granted only if there exist no other locks on the target.

S, U and X are the basic *locking modes*. Modern database systems have implemented also some additional locking modes, and we will return to these later. Other topics concerning the locks include *lock granularity* i.e. locking unit, *duration* of lock, and *lock timeout* (i.e. maximal waiting time for locks).

Typical locking units in relational databases include table, page, and row, but also other units can be used as presented in Figure 7. A *page* is block of 4 to 32 kB (typically 8 kB) of continuous area for table rows and some control data. A *tablespace* is a physical space consisting of one or more disc files or raw disc space for tables and indexes. An *extent* is a continuous set of table pages. The larger the used locking unit is the less concurrency can be supported. On the other hand, the smaller the used locking unit is the more working space and processing overhead is required for lock management work of the DBMS. In early days, a typical locking unit was one page (*page level locking*). Nowadays a typical locking unit is one row (*row level locking*).

A solution to control the locking overhead is *lock escalation*. When the number of page or row-level locks granted to a transaction grows over a configured limit, the DBMS tries to get the corresponding lock mode on the whole table, and after getting this lock, it releases those lower level locks of the transaction. Unfortunately this will often force the transaction to wait, and may sometimes lead to a deadlock situation.

Quite often, after reading a row, the transaction wants to update the row and for that purpose needs to *promote* / *upgrade* the S-lock first into an X-lock. This may also sometimes lead to a deadlock situation.

Figure 7. Locking granularity and locking on different levels

The locking units presented in Fig. 7 form a tree hierarchy and this provides a possibility to build multi-level granularity locking, also called hierarchical locking scheme, where in addition to the locking modes S and X on the basic locking level, corresponding *intent locks* (IS and IX) are first applied on higher level units in the hierarchy. When an S-lock is needed, then an IS-lock is requested first on the highest level containing the target data item, then to its descendant level units, and finally the S-lock on the actual data item, and correspondingly for X-lock request. The purpose of intent locks is to reflect lower level locking requests to granular levels above for checking compatibility of locks on those levels. The compatibility of locks on these upper hierarchical units is shown by Table 5.

Only one lock mode at a time can be granted for a single resource unit access. Since S-lock and IX-lock have different restrictive profiles, so if both would be granted, the existing S or IX lock will be upgraded to a new shared-intention-exclusive mode SIX, which indicates that the current unit is shared locked, and exclusive locks will be requested to the descendant level units. So a SIX-lock is simply the result of S + IX locks, and a SIX-lock on a table guarantees that only the owner of the SIX-lock can get X-locks on pages or rows of the table, while others might get IS-locks on the table and thus might get S-locks on the other pages or rows of the table.

Lock requested:	Lock already granted to some other process				
	IS	IX	S	SIX	X
IS	grant	grant	grant	grant	wait
IX	grant	grant	wait	wait	wait
S	grant	wait	grant	wait	wait
SIX	grant	wait	wait	wait	wait
X	wait	wait	wait	wait	wait

Table 5. Compatibility matrix of granularity locks

For more details see Gray and Reuter [8], Barghouti et al [1], or Elmasri and Navathe [7]

Lock timeout and releasing locks

As default, the mainstream database systems don't apply *lock timeout*, although a timeout value can usually be configured so that a lock request is automatically terminated with an error after waiting time exceeds the limit defined by the timeout parameter. Setting of a short timeout value provides the programming technique for conditional locking. This can be important also in managing concurrency problems of distributed transactions.

In earlier database systems, *lock* requests and releasing of locks using some *unlock* statement were programmed in the application code, but in relational database systems locks are requested implicitly by the DBMS based on the operation and its execution plan as prepared by the DBMS optimizer. Programmers may explicitly lock only tables, but cannot release any locks. The duration of locking is tied to durations of transactions. X-locks are released automatically only at the end of a transaction, whereas releasing of S-locks depends on the current isolation level as we will see later.

According to the 2PL protocol, a well-formed transaction consists of two phases so that locks are acquired only in the first phase, the 'growing phase', and locks are released in the second phase, the 'shrinking phase', as presented in Figure 8. This guarantees serializability of the transaction although it may lead into a deadlock situation. If all the locks are kept up to the end of the transaction, the protocol is called '*strict 2PL'*, whereas the transaction from the programmer's point of view consists only of a single growing phase. The shrinking phase is generated implicitly by the COMMIT or ROLLBACK request of the transaction [8] execution of which actions like for all SQL commands are *atomic operations* of DBMS.

Figure 8. Phases of 2PL protocol

Based on the concepts of record-at-a-time-processing, lock, read, write, unlock, and transaction commit or abort (i.e. rollback) theoreticians have developed abstract concepts like classifications of schedules, histories and schedulers, which are favorite 'brain teaser' topics in many textbooks. Quite often the textbooks forget to mention that these are just simplifications. For example, there is no such mechanism as a 2PL scheduler, which is just an abstraction. As critique on these simplifications, we should remember that SQL is based on atomic set processing operations. For example, an UPDATE command requests first implicit locking operations, then reading and writing of data occurs in complicated table and index *data structures* on database pages. Furthermore, the concurrency of memory structure operations among the internal processes or threads of the DBMS are protected by *latches*, kind of "short term locks". Also Weikum and Vossen [16] admit that "executing transactions is a highly dynamic situation in reality". The real sequence of operations is dependent on SQL commands, the execution plans generated by the DBMS optimizer, the mixture of transactions using different isolation levels, and perhaps differently isolated cursors, programming logic in transactions and thus even on the current contents in the database. Physical reading and writing to disc takes place only on page units.

LSCC and Deadlock Problem

Let's now consider how LSCC systems cope with the concurrency anomalies which we have discussed above.

If we apply Read Uncommitted or Read Committed isolation level for the transactions of the lost update example in Figure 1, no S-locks will protect selected data from being updated by concurrent transactions, but the first transaction issuing an UPDATE command will get the X-lock and succeed updating, while the other needs to wait until the first one will COMMIT, and proceed then with the UPDATE command. In this case, the UPDATE of the first one will not get lost during the transaction and, from the point of view of that transaction, the lost update problem has been solved, but right after the transaction is committed, the update will be lost due to overwrite by the competing transaction.

If we apply Repeatable Read or Serializable isolation level in these transactions, the lost update problem will really be solved, but we will have a new concurrency problem presented in Figure 8b. Both concurrent transactions will first get S-locks for the SELECT commands, and when proceeding to UPDATE commands, both transactions need to promote their S-locks to X-locks, but end up waiting for each other since X-lock is not compatible with the S-lock of the competing transaction. This circular waiting situation is called **deadlock**, but it is not any more a catastrophe for modern DBMS products, which will detect the situation and solve the problem by selecting one of the participating transaction as victim canceling it by automatic ROLLBACK and raising error exception. For example, SQL Server in an interactive mode will give the following kind of error messages:

```
Msg 1205, Level 13, State 51, Line 15
Transaction (Process ID 54) was deadlocked on lock resources with another process
and has been chosen as the deadlock victim. Rerun the transaction.
```

and DB2 LUW will give the following kind of error messages:

```
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0911N  The current transaction has been rolled back because of a deadlock
or timeout.  Reason code "2".  SQLSTATE=40001
```

The SQLSTATE value 40001 shown in the DB2 diagnostics is the concurrency conflict error code as defined by the SQL standard. In Appendix 5, we will show how this should be captured by an application written in Java.

step	Process A	Accounts balance of id 12345	Process B
1	id := 12345; set transaction isolation level Repeatable Read;	1,000 €	id := 12345; set transaction isolation level Repeatable Read;
2	SELECT balance INTO :balance FROM Accounts WHERE acctId = :id; (gets S-lock on the row, ..)		
3			SELECT balance INTO :balance FROM Accounts WHERE acctId = :id; (gets S-lock on the row, ..)
4	newBalance := balance - 100;		
5			newBalance := balance - 200;
6	UPDATE Accounts SET balance = :newBalance WHERE acctId = :id; (waits X-lock on the row, ..)		
7	(waits X-lock on the row, ..)		
8	(waits X-lock on the row, ..)	1,000 €	UPDATE Accounts SET balance = :newBalance WHERE acctId = :id; (waits X-lock on the row, ..)
9			Deadlock detected! => automatic ROLLBACK

Figure 8b. The Lost Update problem solved by locking, but leading to Deadlock

The S-lock of the victim will be released on ROLLBACK, and the other transaction may proceed. On rerun of the victim transaction, a common misunderstanding is that DBMS would restart the transaction, but of course, this is not the case since the transaction logic may depend on the database content, and the content may be different on the next try of the transaction. So it is the responsibility of the application to *retry the victim transaction*. In case the transaction consists of a method of a component in application

server, the rerun of the method can be a service of the application server. In Appendix 5, we will present an example of the *transaction retry programming model*.

A deadlock is not an error nor a catastrophe which should be avoided by all means, even if some people believe so. In fact deadlock management is an important service provided by the DBMS and guarantees that processing can continue even in case that some transactions cannot be run concurrently. What is lost in case of a deadlock is the work of the victim transaction and the waiting time of the application. The waiting time is affected by the time interval of the *deadlock detection mechanism* of the DBMS, which typically is something like 1 to 4 seconds.

A **Dirty Read** anomaly will be solved without other problems by applying any isolation level stronger than Read Uncommitted.

Using isolation level Repeatable Read or stronger to the scenario we presented in Figure 3 on Unrepeatable Read, may lead to a deadlock problem or not, depending on the order of commands. We urge the reader to consider the case if steps 6 and 7 of Figure 3 are executed in a different order.

Timestamp Ordering

Basic timestamp ordering does not use locks, but controls all read and write accesses to data items based on unique transaction timestamps of the last read and last write action. Access is allowed only to data items which have write timestamp of some earlier transaction. The advantage of this technique is that it eliminates waiting times, so instead of deadlock problems, the concurrency conflicts appear as losing the competition when the first writer wins and all competing transactions need to rollback and later restart themselves. For more details, see Weikum and Vossen [16], Coulouris et al [4], Connolly and Begg [3], Elmasri and Navathe [7] etc.

From our point of view, this technique is not relevant since it is not used in the mainstream commercial databases of today.

Multi-Version Timestamp Ordering (MVCC)

Variants of timestamp ordering that have real DBMS implementations are called multi-version timestamp ordering or Multi-Version Concurrency Control (MVCC) techniques. Compared with the basic timestamp ordering which, like LSCC, tries to keep record of current facts only (single versions of data items), the multi-version timestamp ordering preserves a history of data item versions (row versions).

A transaction trying to read, will always get the data item version with the highest write timestamp that is less or equal to the starting timestamp of the transaction, and read timestamp will be set to max(read timestamp, timestamp of the transaction).

A write operation by a transaction to a data item succeeds and read and write timestamps of the new version will be set to the timestamp of the transaction if the start timestamp of the transaction is higher than the read and write timestamps of the latest version of the data item, otherwise the write operation fails and the transaction needs to roll back.

Challenges of this concurrency control technique are the extra storage required on rows for the timestamps and the uniqueness of these timestamps. We will return to some implementations later.

Optimistic Concurrency Control (OCC)

Lock management as well as timestamp comparisons generate some overhead and some techniques have been proposed which optimistically defer the checking of concurrency conflicts to the commit phase of the transaction assuming that concurrency conflicts will occur only seldom. Only read operations are done to the database during the transaction and all writing is done to local copies which, after the concurrency validation phase, will be synchronized to the database, and in case of detected concurrency conflict the transaction is aborted. [7]

OCC is a favorite protocol in database textbooks, but usually vendors calling the concurrency control mechanism of their DBMS as optimistic have in fact implemented a MVCC mechanism. The only real OCC implementation we have verified is concurrency control mechanism of Pyrrho DBMS, developed at University of the West of Scotland (UWS, [25], see www.pyrrhodb.com). In Appendix 1 we present 2 baseline concurrency scenarios tested using Pyrrho DBMS professional version 3.3.