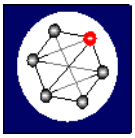

www.DBTechNet.org

# 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.

---

# Introduction

The mainstream database systems used by ICT industry today are DB2, Microsoft SQL Server, and Oracle (in alphabetic order). These together cover the major part of serious OLTP (Online Transaction Processing) market, so students entering on careers as application developers or database professionals in the ICT industry should achieve relevant knowledge on how to use the latest versions of these systems.

This tutorial, which we also call "Concurrency Paper", is the first in our trilogy on data access concurrency topics, and in this paper we will focus on the services and interfaces of *single connection flat SQL transactions* and *concurrency control,* which are perhaps the most important topics in data access technologies. Flat transactions are the type of single SQL transactions covered in SQL standard. The next writing, the "RVV Paper" in the trilogy focuses on sequences of SQL transaction from application point of view, and dependencies between transactions in the sequence on accessing the same data. Even if

the transactions access the same data, they do not need to be bound to the same database connection. The third writing, the "D-Concurrency Paper", focuses on concurrent transactions of same application process accessing the same or different database servers, thus building a distributed transaction, with additional complexity of concurrency.

In their book on Transaction Processing Gray and Reuter [8] present the classical serializability theory and isolation concept from the application programmer (intuitive view), mathematical (theorems and proofs), and operational (implementation) points of view. The tradition of database textbooks is mainly focusing on the results of the theoretical basic research without making the difference between the mere “paper ware” and the real implementations in the mainstream database products, which leaves those texts partially irrelevant from the point of view of ICT industry. In the following, we will focus on implementations and the application programmer's view of concurrency control as defined by the ISO SQL standard and implemented in the mainstream database systems used by the industry. We expect that the readers are already familiar with the basics of SQL and SQL transactions, but just to keep the readers "on our map of concepts" we repeat some of these basics in the following.

A main pedagogical theme of DBTechNet and our workshops is not only "learning by believing", but "learning by doing" and more importantly "learning by verifying". The modern DBMS products with their free Express-versions provide useful tools for experimenting with the real products of the ICT industry.

## ACID or ACiD Transactions

An SQL transaction is a sequence of actions (SQL commands) of an application through a database connection (SQL session) using services of the DBMS ending with either a COMMIT or ROLLBACK request. In addition to these, the SQL transaction may include application logic programmed in the application programming language such as C++, C#, Java, etc. and this transaction logic may depend on the input from the application and on the current contents in the database.

The ideal reliable SQL transaction follows the ACID principle of **A**tomicity, **C**onsistency, **I**solation, and **D**urability, the ACID properties [17], as we teach on our basic SQL courses:

Transaction **Atomicity**, the “all-or-nothing” principle, can be explained by the ROLLBACK request which will generate actions removing all the changes made by the transaction in the contents of the database. Some DBMS products, like DB2 and SQL Server, allow both DML and DDL commands inside a transaction, whereas Oracle allows only DML commands in a transaction and DDL commands (affecting the system catalog) will always be committed so that they cannot be rolled back.

A well-formed and correctly programmed transaction moves the database from a **consistent** state to another consistent state. The DBMS controls database consistency on command level based on the constraint definitions. The ISO SQL standard and some implementations allow the consistency checking against the constraints to be deferred to the end of transaction so that during the transaction the database can be in some inconsistent state, but the DBMS products on which we are focusing have not implemented this feature.

Haerder and Reuter [17] defined **Isolation** in 1983 as follows: "Events within a transaction must be hidden from other transactions running concurrently". This has now been criticized by Chris Date [6]. In the ideal case execution of concurrent transactions will be **serializable** so that the results of the transactions are equivalent to some **serial** execution of those transactions i.e. as though one transaction at a time had been in execution. Unfortunately, the mainstream DBMS products of today do not provide this kind of isolation service in general[1], but they can provide service which prevents the transaction from seeing updates made by concurrent transactions and protect updates made by the transaction against overwriting by others. This kind of **isolation** is a property of the transaction, and we could call the transaction as "**ACiD** transaction". Even the ACiD isolation has often been considered too restricting for concurrent transactions, and therefore some less demanding *isolation levels* have been defined both in the ISO SQL standard and implementations. We will cover these in detail in the following chapters. The isolation solutions affect response times of the application and performance of the DBMS server providing tradeoffs between performance and reliability. However, the programmer must always consider the **reliability** needs as the first priority.

The **durability** property of the transaction against software or hardware failures is guaranteed by write ahead logging (WAL) service of the DBMS at a successful COMMIT request. This guarantees that after any failures, the content of a well managed database can be recovered up to the last committed transaction.

## Some Problematic Concurrency Scenarios

If we don't have proper concurrency control as a service of the DBMS, the concurrent SQL transactions in a multi-user database may generate unreliable content in the database. The following problematic behavior scenarios are widely used as examples of possible concurrency problems, for example in textbooks by Chris Date [5]

- The lost update problem
- The uncommitted dependency problem
- The inconsistent analysis problem.

[1] This implies that the ACID isolation depends on the DBMS, so it actually is a property of the DBMS, in which all transactions need to be ACID transactions. We know only one such a DBMS, namely Pyrrho DBMS.

The **lost update problem** can be described using the following kind of scenario of two concurrent processes accessing the same row (account id 12345 in table Accounts), which we present using embedded SQL and a pseudo programming language in Fig. 1.

The update made by process A in step 6 gets lost in step 8 when process B overwrites it without even checking for potential concurrent updates after step 3. This kind of update is called **Blind Write** or **Dirty Write**. In this scenario the last writer wins, which is not an acceptable business rule, for example, in bank transactions, or in reserving seats to a concert or to a flight. This scenario is possible, when the processes don't use transactions or if they use autocommit mode, in which every SQL command is committed automatically. However, if the processes use SQL transactions, then this lost update, in terms of overwriting an update, is not possible in real DBMS products as long as transaction A is active. However, after commit of the transaction, the data can be overwritten by other transactions. In our RVV Paper, we will cover the problem of lost committed updates and the needed programming discipline to avoid the blind writing.

| step | Process A | Accounts balance of id 12345 | Process B |
|---|---|---|---|
| 1 | id := 12345; | 1,000 € | id := 12345; |
| 2 | SELECT balance<br>INTO :balance<br>FROM Accounts<br>WHERE acctId = :id; | | |
| 3 | | | SELECT balance<br>INTO :balance<br>FROM Accounts<br>WHERE acctId = :id; |
| 4 | newBalance :=<br>balance - 100; | | |
| 5 | | | newBalance :=<br>balance - 200; |
| 6 | UPDATE Accounts<br>SET balance =<br>:newBalance<br>WHERE acctId = :id; | | |
| 7 | | 900 € | |
| 8 | | | UPDATE Accounts<br>SET balance =<br>:newBalance<br>WHERE acctId = :id; |
| 9 | | 800 € | |

**Figure 1. A Lost Update problem**

The **uncommitted dependency problem** arises, for example, when a transaction A accesses data updated by some concurrent transaction B before transaction B gets committed like in the following scenario in Fig. 2 where the transaction B actually ends by ROLLBACK command.

| step | Process A | Accounts balance of id 12345 | Process B |
|---|---|---|---|
| 1 | id = 12345; | 1 000 € | id = 12345; |
| 2 | | | UPDATE Accounts<br>SET balance =<br>balance + 1000<br>WHERE acctId = :id; |
| 3 | SELECT balance<br>INTO :balance<br>FROM Accounts<br>WHERE acctId =<br>:id; | 2 000 € | |
| 4 | | | ROLLBACK; |
| 5 | PRINT 'Great, now<br>I can spend 2000 €' | 1 000 € | |

**Figure 1. An Uncommitted Dependency problem**

The read operation by transaction A, without concurrency control protection by DBMS, is also called as **Dirty Read**. The application of transaction A has seen data which has never been officially true, and will act depending on it, perhaps writing erroneous data to the database.

The **Inconsistent Analysis Problem** of a transaction occurs when some concurrent transaction changes contents of some of the multiple rows which this transactions aims to read. This means that we cannot get results which are the consistent truth during the whole transaction. We can identify two different cases of this problem as follows:

a) If some rows which we have already read will get changed or deleted, and for some reason we need to reread the same set of rows, we will get different results. This anomaly is usually called **Non-Repeatable Read** or **Unrepeatable Read**.

b) If some concurrent transactions change contents of some rows which did not qualify the search conditions of our transaction into values which now would qualify, or if some new qualifying rows are inserted into the database, which our transaction would see if it repeats the same searches, these new rows in our readset are called **Phantoms**.

As an example of Unrepeatable Read, let's consider the following case presented in Fig 3.

Clearly transaction A gets here a wrong analysis of the sum of balances of owner 123, and if it repeats the read operations, it will get another sum, maybe even the right sum of 3040 €.

Accounts of owner 123

| acctId | balance |
|---|---|
| 1001 | 2 000 € |
| 2002 | 1 000 € |
| 3003 | 40 € |
| total: | 3 040 € |

| step | Transaction A | acctId | balance | sum | Transaction B |
|---|---|---|---|---|---|
| 1 | owner = 123; | | | 0 € | |
| 2 | DECLARE crs CURSOR FOR SELECT acctId, balance FROM Accounts WHERE owner = :owner; | | | | |
| 3 | OPEN crs; | | | | |
| 4 | FETCH crs ..; | 1001 | 2 000 € | 0 € | |
| 5 | sum := sum + balance; | | | 2 000 € | |
| 6 | | | | | UPDATE Accounts SET balance = balance - 2000 WHERE acctId = 1001; |
| 7 | | | | | UPDATE Accounts SET balance = balance + 2000 WHERE acctId = 3003; |
| 8 | | | | | COMMIT; |
| 9 | FETCH crs ..; | 2002 | 1 000 € | 2 000 € | |
| 10 | sum := sum + balance; | | | 3 000 € | |
| 11 | FETCH crs ..; | 3003 | 2 040 € | 3 000 € | |
| 12 | sum := sum + balance; | | | 5 040 € | |

**Figure 2. A Unreapeatable Read**

Now let's consider the case that the owner 123 opens a new account in the bank in between the transaction A as presented in Fig. 4.

Accounts of owner 123

| acctId | balance |
|---|---|
| 1001 | 2 000 € |
| 2002 | 1 000 € |
| 3003 | 40 € |
| total: | 3 040 € |

| step | Transaction A | acctId | balance | sum | Transaction B |
|---|---|---|---|---|---|
| 1 | owner = 123; | | | 0 € | |
| 2 | DECLARE crs CURSOR FOR SELECT acctId, balance FROM Accounts WHERE owner = :owner; | | | | |
| 3 | OPEN crs; | | | | |
| 4 | FETCH crs ..; | 1001 | 2 000 € | 0 € | |
| 5 | sum := sum + balance; | | | 2 000 € | |
| 6 | FETCH crs ..; | 2002 | 1 000 € | 2 000 € | |
| 7 | sum := sum + balance; | | | 3 000 € | |
| 8 | | | | | INSERT INTO Accounts (acctId,owner,balance) VALUES (1009, 123, 5000); |
| 9 | | | | | COMMIT; |
| 10 | FETCH crs ..; | 3003 | 40 € | 3 000 € | |
| 11 | sum := sum + balance; | | | 3 040 € | |

**Figure 3. Example of a Phantom row**

Transaction A does not see the new account 1009 which we now call as a phantom row.

## Isolation Levels

Without considering the implementation techniques of transaction isolation, the ISO SQL standard defines the following 4 isolation levels in terms of anomalies (badly behaving transaction types such as Dirty Read, Unrepeatable Read, and Phantoms) against which the selected isolation level gives protection for the transaction. We refer to the basic SQL course on explanations of these and we just present the summary of effects of these transaction isolation levels in table 1. The isolation levels of ISO SQL standard don't cover all isolation level implementations in real DBMS products, not all products implement even all these isolation levels, and for example Oracle calls its snapshot isolation (SI) as serializable, which is misleading. These differences with the ISO SQL standard have been discussed by Berenson et al [2].

| Anomaly:<br><br>Isolation level: | Lost Update | Dirty Read | Unrepeatable Read | Phantoms |
|---|---|---|---|---|
| Read Uncommitted | not possible | possible | possible | possible |
| Read Committed | not possible | not possible | possible | possible |
| Repeatable Read | not possible | not possible | not possible | possible |
| Serializable | not possible | not possible | not possible | not possible |

Table 1. Concurrency anomalies solved by the ISO SQL transaction isolation levels

*Lost Update* in a transaction scope means that some concurrent transaction writes over an update written by the transaction before the transaction commits. According to the ISO SQL standard, a Read Uncommitted transaction cannot write, which would imply that Lost Update is not possible. However, the mainstream database systems, which we are studying, do allow writing, but as we will see later, the two types of *concurrency control mechanisms* also prevent the lost update anomaly in scope of the current transaction:

- The *locking scheme* systems keep the write locks (X-locks) up to the end of transaction, thus preventing the lost update.
- In *Multi-versioning* systems the first one of concurrent transactions to update or delete a row wins (an update action writes always a new row version), whereas competing actions by concurrent transactions will fail, so no update will be lost. This is something you should test yourself.

The default isolation level for transactions according to the ISO SQL standard is SERIALIZABLE, which usually conforms to the ACiD isolation (and to ACID if SERILIZABLE is the only allowed isolation level). The default isolation level of the mainstream database systems is READ COMMITTED or a compatible isolation.

According to the ISO SQL standard the isolation level is one of the three characteristics (modes) of a transaction, the modes of which can be defined by the following command syntax:

```
SET TRANSACTION <mode> [{,<mode>}..]
```

where

```
<mode> :=
      <isolation level>
      | <transaction access mode>
      | <diagnostics size>
```

The <isolation level> is declared by :

```
ISOLATION LEVEL <level>
```

where

```
<level> :=
      READ UNCOMMITTED
      | READ COMMITTED
      | REPEATABLE READ
      | SERIALIZABLE
```

Another transaction characteristic is the access mode, which is by default READ WRITE, but can also be declared as READ ONLY. according to the standard Read Uncommitted implies READ ONLY access mode for the transaction.

`The SET TRANSACTION` command shall be executed before start of the transaction, and according to Melton ([11], p 513) only for the next transaction. According to ISO SQL standard SQL-92, a transaction starts *implicitly* always when the first SQL command which can be included in a transaction is entered after SQL session starts or after COMMIT or ROLLBACK of a previous transaction. The later standard SQL:1999 defines also an *explicit* transaction start command

```
START TRANSACTION <mode> [{,<mode>}..]
```

in which can include declarations of the access mode or isolation level for the transaction.

In the following we will look at the real implementations in current versions of Oracle, SQL Server, and DB2.

**Oracle** has implemented part of the SET TRANSACTION command as shown by the syntax diagram in Fig. 5.

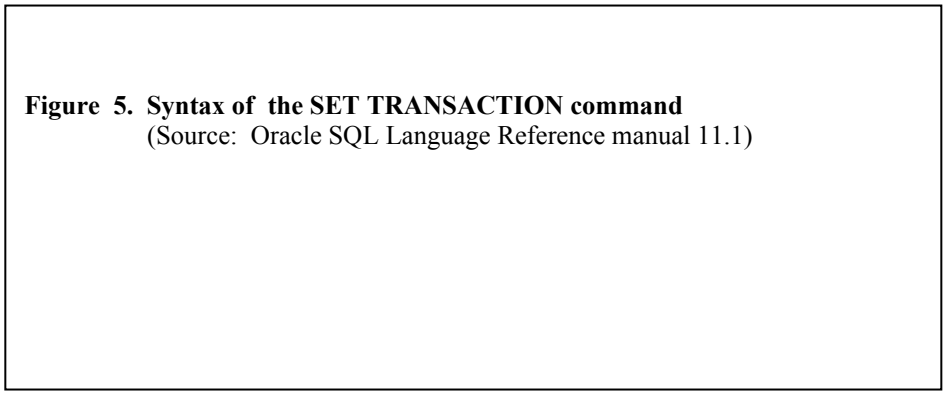

**Figure 5. Syntax of the SET TRANSACTION command**
(Source: Oracle SQL Language Reference manual 11.1)

If used, the command must be the first command of the transaction, and it applies only to the current transaction. The only supported isolation levels are READ COMMITTED and SERIALIZABLE, but the semantics are a bit different from the isolation semantics of the SQL standard. We will return to these later. Complying with the SQL-92 standard, Oracle has no explicit start command for a transaction, but the first detected DML command starts implicitly next new transaction after beginning of SQL session or after end of the previous transaction.

The default transaction isolation level in Oracle is READ COMMITTED, but it is possible to configure the default isolation level for the open SQL session using the command

```
SET SESSION ISOLATION_LEVEL = {SERIALIZABLE | READ COMMITTED}
```

Especially in case of stored procedures or subroutines of transaction code it might be useful to be able to check what is the current isolation level. This can be checked from the dynamic system views of Oracle, but any easy means of getting this information has not been implemented in the SQL or PL/SQL languages of Oracle.

Note:

*The current isolation level is available in Java programs by the getTransactionIsolation() method of Connection interface of JDBC drivers of Oracle, as well as JDBC drivers of other DBMS products. However, even if we can extend PL/SQL language by storing Java classes in the database and to define methods of these classes as stored procedures and functions, which should appear in the scope of the calling SQL transaction, we have not yet managed to get the right isolation level as return value of a stored function in Oracle 11g2.*

**SQL Server** operates as default in *autocommit* mode, so that commands cannot be rolled back. Instead of START TRANSACTION command, Transact-SQL transactions can be started explicitly by BEGIN TRANSACTION command, which can also define a name for the transaction (to be applied for kind of 'nested transactions'), and a mark to be applied in transaction log records. SQL Server version 2008 allows also implicit

transactions to be turned on as follows:

```
SET implicit_transactions ON
```

The isolation level can be defined by the SET TRANSACTION command, which can be declared outside the transaction scope or as the first command in a transaction, and it remains as the isolation level for the subsequent transactions of the SQL session until changed by a new SET TRANSACTION command. All isolation levels of ISO SQL standard are supported as defined in the standard, and in addition a new isolation level called SNAPSHOT, which corresponds to the isolation level SERIALIZABLE of Oracle. Default isolation level is READ COMMITTED. The current isolation level can be checked using the command

```
SELECT transaction_isolation_level
FROM sys.dm_exec_sessions
WHERE session_id = @@spid
```

The code values of transaction_isolation_level values are explained in table 2. The values 0-2 are the same as the numeric values of the transaction level codes in JDBC programming, but the codes > 2 follow different logic.

| **transaction_isolation_level codes in SQL Server** | **Isolation Levels of SQL Server** | **TransactionIsolation codes in JDBC** |
|---|---|---|
| 0 | unspecified | 0 |
| 1 | Read Uncommitted | 1 |
| 2 | Read Committed | 2 |
| 3 | Repeatable Read | 4 |
| 4 | Serializable | 8 |
| 5 | Snapshot | |

Table 2. Isolation level codes of SQL Server compared with JDBC codes

Following commands test the SQL Server behavior in terms of SET TRANSACTION as explained above:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
COMMIT
SELECT transaction_isolation_level
FROM sys.dm_exec_sessions
WHERE session_id = @@spid

transaction_isolation_level
---------------------------
1
```

A Transact-SQL transaction cannot be declared as READ ONLY, but this can be declared as a mode for a cursor, a file group (i.e. tablespace), or a database in the server instance.

**DB2** operates also in autocommit mode as default for interactive SQL, but can be configured to use implicit transactions. For start of transactions, DB2 complies with the SQL-92 implicit transaction mode. The isolation level is a property of the connection and needs to be defined before the SQL session starts using the SET ISOLATION command, or during execution of dynamic SQL using the SET CURRENT ISOLATION command presented in Fig. 6.

**Figure 6. Syntax of SET ISOLATION of DB2**
(Source: DB2 LUW V9.5 SQL Reference, Vol 2)

Names of the isolation levels are different, but the semantics matches with the ISO SQL standard isolation levels as presented in table 3.

| Isolation level of DB2 | explanation | corresponding isolation level of SQL standard |
|---|---|---|
| UR | Uncommitted Read | Read Uncommitted |
| CS | Cursor Stability | Read Committed |
| RS | Read Stability | Repeatable Read |
| RR | Repeatable Read | Serializable |

Table 3. Isolation levels of DB2

In the transaction scope, the Cursor Stability of DB2 matches the behavior of Read Committed of the SQL standard, but in addition, it affects concurrency control on cursors so that no concurrent transaction can change the current row of an open cursor. Cursor Stability has been the **default isolation level** of DB2, but starting from version 9.7 the **new default isolation level** in new databases is `CURRENTLY COMMITTED` (CC). This behaves like Read Committed isolation of the SQL standard, but instead of protecting the read operations by shared locks, the latest committed version of the row will be retrieved from transaction history (log) without waiting for locks and without need for getting any locks on the row.

`SET CURRENT ISOLATION` sets the isolation value to a special CHAR(2) register which will override the isolation level of the application package, and will be used up to the end of the SQL session or until a new `SET CURRENT ISOLATION` command. Application can read the current isolation level by the singleton read command of DB2 as follows

```
VALUES CURRENT ISOLATION
```

which returns the two character code of the isolation level, except spaces in case of CC. In version 9.7 it is not possible to set the current isolation level to CC, but this can be done using the command

```
SET ISOLATION RESET
```

For example,

```
VALUES CURRENT ISOLATION;

1
--


  1 record(s) selected.

SET ISOLATION RR;
DB20000I  The SQL command completed successfully.

VALUES CURRENT ISOLATION;

1
--
RR

  1 record(s) selected.


SET ISOLATION RESET;
DB20000I  The SQL command completed successfully.

VALUES CURRENT ISOLATION;

1
--
```

In DB2 a transaction cannot be declared as `READ ONLY` (but a cursor or a view can).

Note: Even if DB2 uses different names for the isolation levels than ISO SQL standard, for example in programmatic ODBC or JDBC connections the programs may use isolation level definitions based on names in the ISO SQL standard, and the requested isolation levels are mapped to the corresponding isolation levels of the DBMS by the used database driver.

**Summary**

Based on what we have found above, it is a *good and portable practice* to set the transaction isolation level *always explicitly as the first statement of every transaction* that reads data. - But why always? Doesn't isolation level affect only read operations? Yes, but even if our transaction consists only of write operations like INSERT, UPDATE, or DELETE, these commands may use subqueries or there might be included some implicit read operations of which we may not know. For example, the DBA may have created or might, in the future, create triggers which include read operations for some business rules.

The triggers will be included in the scope of the transaction and the isolation level protects also the read operations in triggers. If the isolation level after the previous transaction remains as `REPEATABLE READ` or `SERIALIZABLE`, the read operations in triggers may get blocked. On the other hand, the business rule of a trigger may be set to protect against some *write skew* anomaly (see [2], but a too low isolation level may eliminate this protection effect of the trigger. It is important to tune the isolation level according to all the needs of the current transaction, also the needs of the included triggers.

*The problem with isolation levels of triggers is that triggers are usually not programmed as part of certain transaction, but they affect in scopes and logic of different transactions which may have isolation levels of their own. So we should be able to fine tune the isolation on statement level in triggers independently from isolation levels of transactions which will be affected by the trigger. The ISO SQL standard provides no means for this, but we will return later to the solutions provided by the implementations.*