

www.DBTechNet.org



Education and Culture DG  
Lifelong Learning Programme

# On SQL Concurrency Technologies

## - for Application Developers

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

Martti Laiho<sup>1</sup>, Fritz Laux<sup>2</sup>

<sup>1</sup> Dpt. of Business Information Technology, Haaga-Helia University of Applied Sciences, Ratapihantie 13, 00520 Helsinki, Finland

<sup>2</sup> 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 Technologies of Real DBMS Products

The locking schemes of DB2 and SQL Server are based on the single-version, hierarchical, multi-granularity (for example tablespace/table/page/row) locking using lock modes S, X, and U on selected lock granule, and corresponding intent mode locks IS, IX, and SIX on the upper granular levels, always acquiring the (possible intent) locks on the upper levels first. S-lock durations are configured based on the selected isolation level as discussed later.

Early DB2 versions used page-level locking and still single tables can be configured to use it, but the default locking level is row-level locking. SQL Server uses only row-level locking.

### LSCC in SQL Server

Microsoft has implemented the locking scheme as presented by Gray [8], but included many new lock modes such as intent update (IU), key-range locks preventing from phantoms, bulk update locks preventing others accessing the table during a bulk load of data, and schema locks preventing from schema modifications [21].

The locking units used in this multi-granular implementation include, for example, the following (for more details see Appendix 2):

- RID – rowid of a single row in a table which doesn't have clustered index
- KEY- key value of a row in a table with clustered index
- PAGE – page of a table or index
- EXTENT – continuous group of 8 table or index pages
- TABLE – whole table including the data and its indexes
- FILE – a single file instead of the whole tablespace (which is called filegroup)
- DATABASE – an accessed database in the SQL Server instance

Transact-SQL does not have an explicit LOCK TABLE command, but the cost-based optimizer of SQL Server decides of the execution plan and locking plan based on command type, cardinality (statistics), the requested isolation level and the table hints. Depending on the estimated number of rows to be processed, SQL Server will dynamically select row-level or page-level locking, and in case of too many locks, occasionally may decide on lock escalations to table level.

We will now have a look at what kind of locks the optimizer of SQL Server will request based on different isolation levels:

For READ UNCOMMITTED isolation level no S-locks are requested.

READ COMMITTED is the default isolation level of SQL Server. For read operations it will requests a short S-lock just for the time of the command and the lock will be released, according to Microsoft MSDN SQL Server 2008 Books Online documentation [21], depending on the granularity of the lock as follows: "Row locks are released before the next row is processed. Page locks are released when the next page is read, and table locks are released when the statement finishes." We have tested this in Appendix 2 using Transact-SQL cursor processing, and the test proves that the row lock is released immediately after cursor has fetched the row. Against the SQL standard, Read Committed isolation in SQL Server does not imply READ ONLY mode of the transaction.

For REPEATABLE READ isolation level, SQL Server keeps all S-locks and the multi-granular intent locks up to the end of the transaction. In the following example we use a simple table T

```
CREATE TABLE T (  
  id INT NOT NULL PRIMARY KEY,  
  s CHAR(10)  
);
```

with the following contents

```
INSERT INTO T VALUES (1,'a');  
INSERT INTO T VALUES (2,'b');  
INSERT INTO T VALUES (3,'c');
```

Current locks in SQL Server can be monitored using system stored procedure `sp_lock` or the new dynamic management view `sys.dm_tran_locks` which would allow more focused monitoring. In the following test we simply use `sp_lock` to monitor the locks.

We start 3 query windows accessing the same database, and in the first window, SQLQuery1 (process 53), we run the following transaction

```
-- session A  
BEGIN TRANSACTION  
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;  
SELECT * FROM T ;  
UPDATE T SET s = 'value1' WHERE id = 1 ;
```

and in the second window, SQLQuery2 (process 55), we run the following transaction

```
-- session B  
UPDATE T SET s = 'value2' WHERE id = 1 ;
```

which seems to be blocked by some other transaction. Now, in the third window, SQLQuery3 (process 57), we use `sp_lock` procedure to look at the locks

```
EXEC sp_lock
```

spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
53	5	0	0	DB		S	GRANT
53	5	1157579162	1	KEY	(8194443284a0)	X	GRANT
53	5	1157579162	1	PAG	1:121	IX	GRANT
53	5	1157579162	0	TAB		IX	GRANT
53	5	1157579162	1	KEY	(98ec012aa510)	S	GRANT
53	5	1157579162	1	KEY	(61a06abd401c)	S	GRANT
54	5	0	0	DB		S	GRANT
55	5	1157579162	0	TAB		IX	GRANT
55	5	1157579162	1	PAG	1:121	IX	GRANT
55	5	1157579162	1	KEY	(8194443284a0)	X	WAIT
55	5	0	0	DB		S	GRANT
56	5	0	0	DB		S	GRANT
57	5	0	0	DB		S	GRANT
57	1	1131151075	0	TAB		IS	GRANT

In the report we see locks on databases (`DB`), tables (`TAB`), pages (`PAG`), and keys (`KEY`) of rows. Our database has internal number 5 in our SQL Server instance, and table T has object ID 1157579162. Process 57 is accessing also some table from database 1 for the lock report. For primary key of table T we have used the defaults in SQL Server, so the table T has clustered index <sup>1</sup> for the primary key. This means that the rows are included on the leaf level pages of the index. Therefore the key locks appear as locks on the index (`IndId` = 1). (Processes 54 and 56 are some internal processes of SQL Server 2008 R2 Express.)

---

<sup>1</sup> Clustered indexes are explained for example in the Index Design workshop of DBTechNet

After repeating the test using isolation level `SERIALIZABLE` in session A, we get the following lock report:

spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
53	5	0	0	DB		S	GRANT
53	5	1157579162	1	KEY	(8194443284a0)	RangeX-X	GRANT
53	5	1157579162	1	PAG	1:121	IX	GRANT
53	5	1157579162	1	KEY	(fffffffffffff)	RangeS-S	GRANT
53	5	1157579162	0	TAB		IX	GRANT
53	5	1157579162	1	KEY	(98ec012aa510)	RangeS-S	GRANT
53	5	1157579162	1	KEY	(61a06abd401c)	RangeS-S	GRANT
55	5	1157579162	0	TAB		IX	GRANT
55	5	1157579162	1	PAG	1:121	IX	GRANT
55	5	1157579162	1	KEY	(8194443284a0)	X	WAIT
55	5	0	0	DB		S	GRANT
57	1	1131151075	0	TAB		IS	GRANT
57	5	0	0	DB		S	GRANT

SQL Server requests and keeps row level S-locks as RangeS-S locks on index, and X-locks as RangeX-X. The RangeS-S lock on the high-end KEY value `(fffffffffffff)` in the clustered index (`IndId = 1`) prevents concurrent transactions from inserting new key values on the primary key, i.e. from inserting new rows in the table.

For a complete list of SQL server locking modes and also implementation of MVCC concurrency control, we refer to Appendix 2 and SQL Server Books Online.

We present examples on use of U-locks by SQL Server in our "RVV Paper" [12].

The isolation level defined for the scope of transaction is the default isolation used by optimizer to sort out proper locking strategy for the commands in the transaction, but the strategy can be changed for any table in a single command by writing `Transact-SQL table options` in a `WITH` clause after the table name.

The table options include, for example,

- reconfiguring the isolation level: `READUNCOMMITTED`, `READCOMMITTED`,  
`REPEATABLEREAD`, `SERIALIZABLE`
- `READPAST` to bypass exclusively locked rows instead of waiting on reading  
implementing read-past locking (see Gray [7], p 402)
- configuring the lock mode: `UPDLOCK`, `XLOCK`
- configuring the lock granule: `PAGLOCK`, `ROWLOCK`, `TABLOCK`
- configuring the lock options: `NOLOCK`, `NOWAIT`, `HOLDLOCK`.

For full documentation of table options, see SQL Server Books Online / table hints.

*Lock timeout* can be set in milliseconds, for example, to a half second as follows:

```
SET LOCK_TIMEOUT TO 500 ;
```

A powerful transaction tuning option is `DEADLOCK_PRIORITY` which can be used to set transaction priority to low (down to -10) to become a potential victim in case of a deadlock, or high (up to 10) to win the deadlock competition.

## LSCC in DB2 LUW

Concurrency control in DB2 is based only on the single-version locking scheme. Default locking granularity is on row-level, but can be configured by `LOCKSIZE` parameter of `CREATE / ALTER TABLE` commands to `ROW` or `TABLE`. Page-level locking is available only for special editions (see [20]). In this tutorial we focus on the DB2 Express-C edition only.

Beside the basic and intent lock modes, DB2 uses some other lock modes which are described in [20].

Locking plan is decided by the DBMS depending on the execution plan generated by the optimizer and the isolation level requested by the application as presented in Figure 9. Application can also lock a table or view by an explicit `LOCK TABLE` command as follows:

```
>>-LOCK TABLE---+table-name---+IN---+SHARE-----+MODE-----><
      '-nickname---'      '-EXCLUSIVE-'
```

(source: DB2 Information Center)

However, the explicit locking is usually not recommended.

Listings with tests on various isolation levels on the locks held by DB2 are presented in Appendix 3.

Isolation level `UNCOMMITTED READ (UR)` corresponds to the `READ UNCOMMITTED` of SQL Standard, and like SQL Server, DB2 does not cover reading with S-locks for these transactions.

Isolation level `CURSOR STABILITY (CS)` corresponds to the `READ COMMITTED` of SQL Standard, and like SQL Server, DB2 requests S-locks for rows to read committed data and releases the locks immediately after reading. However, in case of cursor processing, the lock on current row is kept.

Isolation level `READ STABILITY` holds IS-lock on table and S-lock on every row of the result set.

Isolation level `REPEATABLE READ` holds an S-lock on the table allowing the reading of table rows without row-level locking.

**Figure 9. Basic locking mode selection by the DB2 optimizer**

Note: In case of Read Stability or Cursor Stability, on row-level, DB2 uses Next Key Share (NS) locks instead of S-locks.

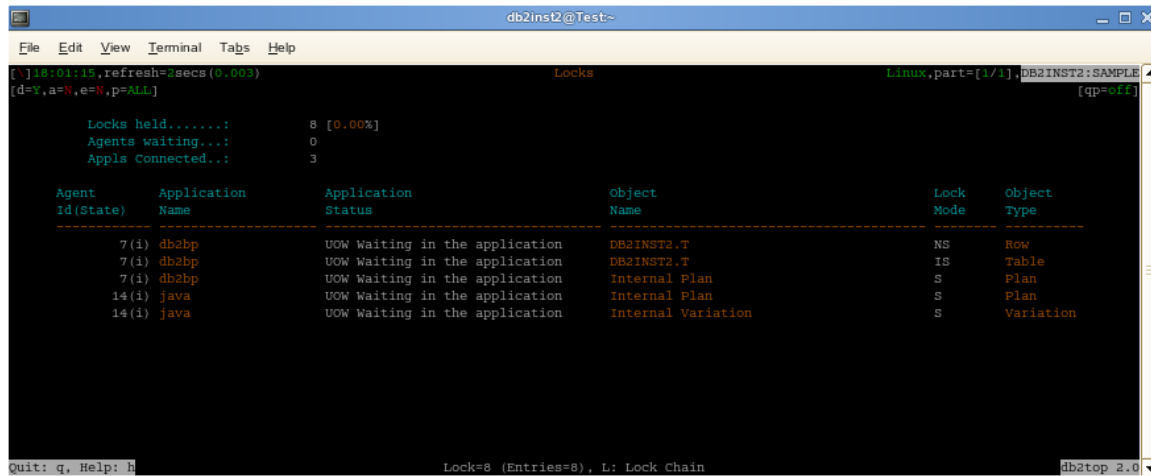
There are many possibilities to watch the current locks in a DB2 database providing different kinds of information, but unfortunately we have not yet found any as usable for a novice user as in SQL Server. There are various utilities such as `db2dp` and `db2top` (the later only on Unix and Linux platforms) and system views `SYSIBMADM.SNAPLOCK` and `SYSIBMADM.SNAPLOCKWAIT`. The following examples demonstrate these alternatives using similar table T like we used in SQL Server after following command sequence

```
SET ISOLATION TO RS;  
SELECT * FROM T;
```

On Unix or Linux platforms we can watch the locks by `db2top` on terminal window as follows

```
db2top -d sample
```

and selecting “U” for Locks we get the following view

The screenshot shows the db2top utility running in a terminal window titled 'db2inst2@Test~'. The main display shows lock statistics: 'Locks held.....: 8 [0.00%]', 'Agents waiting...: 0', and 'Apps Connected...: 3'. Below this is a table of locks. The table has columns: Agent Id(State), Application Name, Application Status, Object Name, Lock Mode, and Object Type. The data rows show agent 7 (db2bp) holding NS locks on rows of table DB2INST2.T, and agent 14 (java) holding IS locks on table DB2INST2.T. There are also internal locks on plans and variations. The bottom status bar indicates 'Lock=8 (Entries=8), L: Lock Chain' and 'db2top 2.0'.

Unfortunately we cannot configure this view of `db2top`. In our case it just shows us that there is IS-lock on table T and at least 1 NS-lock on rows of table T for agent 7 which is our Command Editor connection. The “Internal” locks are technical locks: S-locks on Internal Plans are locks on the used application plans (packages), and S-locks on Internal Variations are locks used for dynamic SQL in cache.

Utility `db2pd` presents us a bit more information as follows

```
db2inst2@Test:~> db2pd -d sample -locks

Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:55:20

Locks:
Address   TranHdl  Lockname                                     Type      Mode Sts Owner      Dur HoldCount Att      ReleaseFlg rriID
-----
0xA3A7E280 2      03001400040000000000000000000052 Row        .NS  G  2      1  0      0x00000000 0x00000001 0
0xA3A7E580 2      53514C43324832307F4760B841 Internal P  ..S  G  2      1  0      0x00000000 0x40000000 0
0xA3A80480 8      53514C4445464C5428DD630641 Internal P  ..S  G  8      1  0      0x00000000 0x40000000 0
0xA3A7E880 2      03001400050000000000000000000052 Row        .NS  G  2      1  0      0x00000000 0x00000001 0
0xA3A80180 8      010000000100000000100C00756 Internal V  ..S  G  8      1  0      0x00000000 0x40000000 0
0xA3A7E980 2      03001400060000000000000000000052 Row        .NS  G  2      1  0      0x00000000 0x00000001 0
0xA3A80380 8      5359535348323030DDECECF2841 Internal P  ..S  G  8      1  0      0x00000000 0x40000000 0
0xA3A7E780 2      03001400000000000000000000000054 Table      .IS  G  2      1  0      0x00002000 0x00000001 0
.. ..
```

At least we now know that some 3 rows have NS locks, and since there are locks (the IS lock) only on table T, these are rows of the table T.

We can get more useful lock reports configuring the MONITOR switch for locks ON for the database as follows:

```
db2 "update monitor switches using lock on"
```

and then applying queries to performance snapshot system views `SNAPLOCK` and `SNAPLOCKWAIT`. (Note: keeping MONITOR switches ON will generate extra overhead, so this should usually be avoided in a production environment.)

Trying to keep our lock monitoring as simple as in case of SQL Server, we create a monitoring view combining the snapshot system views as follows:

```
CREATE VIEW Locks (proc,tabname,type,mode,status,count,holder)
AS
SELECT CAST(agent_id AS SMALLINT),
       CAST(TRIM(tabschema)||'.'||TRIM(tabname) AS CHAR(30)),
       CAST(lock_object_type AS CHAR(12)),
       CAST(lock_mode AS CHAR(6)),
       CAST(lock_status AS CHAR(6)),
       CAST(lock_count AS SMALLINT),
       CAST(NULL AS SMALLINT)
FROM sysIBMadm.SnapLock
WHERE lock_object_type NOT LIKE 'INTERN%'
UNION ALL
SELECT CAST(agent_id AS SMALLINT),
       CAST(TRIM(tabschema)||'.'||TRIM(tabname) AS CHAR(30)),
       CAST(lock_object_type AS CHAR(12)),
       CAST(lock_mode AS CHAR(6)), 'WAIT',
       CAST(1 AS SMALLINT),
       CAST(agent_id_holding_lk AS SMALLINT)
FROM sysIBMadm.SnapLockWait
COMMIT;
```

Let's now apply the same test we had in SQL Server using concurrent sessions. In session A (proc 1463) we start the following transaction

```
-- session A
SET ISOLATION = RS ;
SELECT * FROM T ;
UPDATE T SET s = 'value1' WHERE id = 1 ;
```

and in session B (proc 1493) the following command will be blocked by session A

```
-- session B
UPDATE T SET s = 'value2' WHERE id = 1 ;
```

which we can see by running the following query in a third concurrent session

```
SELECT * FROM Locks ORDER BY proc, tabname, mode;
```

PROC	TABNAME	TYPE	MODE	STATUS	COUNT	HOLDER
1463	TIKO.T	TABLE_LOCK	IX	GRNT	2	-
1463	TIKO.T	ROW_LOCK	NS	GRNT	1	-
1463	TIKO.T	ROW_LOCK	NS	GRNT	1	-
1463	TIKO.T	ROW_LOCK	X	GRNT	2	-
1493	TIKO.T	TABLE_LOCK	IX	GRNT	1	-
1493	TIKO.T	ROW_LOCK	X	WAIT	1	1463

Note: This has been tested using DB2 V9.7 Express-C on Windows XP workstation. While running the very same test on Linux platforms we have not got the last line of the waiting process.



The lock mode NS stands for Next Key Share, which is used instead of row-level S-lock on isolation levels CS and RS.

Compared with our lock report of SQL Server, in this DB2 lock report we see the qualified names of the tables, but will not see any key values or addresses.

The most extensive lock monitoring report format we get using the following get snapshot command

```
db2 "get snapshot for locks on <database>"
```

The problem with this snapshot report is that we cannot configure the report to focus only on those details which we are interested in.

## Concurrency Control in Oracle

Oracle was the first commercial database product on the market in 1979. Even if it was partly implemented based on the public papers of the System R development team of IBM, its concurrency control technology as implemented in version 6 is quite unique, a mixture of MVCC and locking. It supports both row-level X-locking, and multiple modes of table-level locking. Instead of separate lock records in buffers of DBMS like System R, DB2 and SQL Server use, Oracle uses row locks which Gulutzan and Pelzer [10] call "marks on the wall". That is a System Change Number (SCN) of the latest writing transaction marking the page or (starting from Oracle 10 versions) a technical column on a table created with ROWDEPENDENCIES option [23] marking every row. If the "mark" can be found in the internal table of active transactions, the row is considered locked for this transaction. This eliminates the need for any lock escalations.

While any row in a table is locked, the table is marked as ROW EXCLUSIVE (RX) locked [14]. This is similar to the IX-lock on table in the LSCC systems.

No read-locking and waiting for read requests is needed due to the multi-versioning implementation of Oracle. Whenever a row is updated (or deleted), the old version is written to a separate storage (Rollback Segment of earlier or Undo Tablespace of the latest Oracle versions) and chained there to the saved history of the rows. Other transactions which want to read the row will immediately get the version from this history which fits with the isolation level of the reading transaction. See Fig. 10.

**Figure 10. Oracle Multi-Versioning**

In Oracle's MVCC solution, it is possible to delete those row versions in the history chains which are too old for any active transaction. The new storage solution using the Undo Tablespace for the version history chains instead of separate rollback segments is easier to manage. However, it is necessary to reserve space big enough for this storage, otherwise readers may get the error message of "snapshot too old".

*Note: SQL Server's MVCC solution is like Oracle's MVCC, except that version history chains are stored in the TempDB database, which should also be reserved big enough.*

The only supported isolation levels in Oracle are called READ COMMITTED and SERIALIZABLE although these should be called something like "READ LATEST COMMITTED" and "SNAPSHOT SERIALIZABLE", since Read Committed gets the latest committed version of a data item from the row history, and Serializable gets the latest row versions which have been committed before the reading transaction has started.

If a transaction with isolation level SERIALIZABLE, having read a row from the history chain instead of the current *rowid* location, tries to update the row, it will get a concurrency conflict exception.

The concurrency control described above is managed automatically. In addition to automatic locking and independent of the isolation level, the programmer can **explicitly lock rows** using a SELECT command with FOR UPDATE clause, for example,

```
SELECT * FROM T WHERE id = 2 FOR UPDATE;
```

In our RVV paper [11] we will show that this is a must - a real programmer's "life saver" in using Oracle.

**Figure 11. Syntax of Oracle's LOCK TABLE command (source [20] )**

Although it is not recommended, programmers can use also explicit table locking commands using the syntax in Figure 11. The available table lock modes are listed in table 6 and the compatibilities of these in table 7.

lockmode:	code:	description:
EXCLUSIVE	X	table locked exclusively, but others can read data
SHARE	S	table locked for READ ONLY mode. No process can update rows
ROW EXCLUSIVE	RX	default processing mode, see above
ROW SHARE	RS	other processes can read data, but no other gets exclusive lock on the table
SHARE ROW EXCLUSIVE	SRX	others can lock the table in RS mode, but not in X or S mode

Table 6. Table Lock modes of Oracle (sources [14], [23])

Lock requested:	Lock already granted to some other process				
	RS	RX	S	SRX	X
RS	grant	grant	grant	grant	wait?
RX	grant	grant	wait?	wait?	wait?
S	grant	wait?	grant	wait?	wait?
SRX	grant	wait?	wait?	wait?	wait?
X	wait?	wait?	wait?	wait?	wait?

Table 7. Compatibility matrix of Oracle table locks. Waiting depends on the WAIT parameter (sources [14], [23])

The RS, RX, and SRX are actually Oracle's implementation of intent locks on table level.

As a summary, Oracle's concurrency control mechanism is a hybrid mixture of MVCC and table-level locking.

Oracle locks can be monitored by DBA users using the system view V\$LOCK as described in Appendix 4.

This far Oracle has not implemented lock timeout parameter for local transactions, but for distributed transactions DISTRIBUTED\_LOCK\_TIMEOUT can be set as 1 or more seconds.

### **Concurrency conflicts in Oracle:**

It is possible get deadlocks also in Oracle, but Oracle deals with deadlocks differently from pure LSCC systems. Instead of automatic ROLLBACK, Oracle will rollback only the command which would lead to deadlock situation, and it will inform the application on the situation. It is then the responsibility of the application to request for the ROLLBACK.

Due to MVCC nature of Oracle, there may appear also other forms of concurrency conflicts which are not called as deadlock, but should be treated the same way by the application.

### **Statement and Cursor Level Isolations**

As a good programming practice and according to the SQL standard, it is not possible to change the *transaction isolation level* after beginning of the transaction [11] . The isolation level of transaction affects as *default on SQL statements on read accessing* to data which concurrent transactions have modified, and on how long the transaction prevents other transactions from modifying the rows it has been reading. Note that a transaction in DBMS, which allows Read Uncommitted isolation, cannot prevent concurrent transactions from reading the rows it has itself modified, which is a common misunderstanding.

ISO SQL standard does not define concurrency separately for command level, but as we learned, DB2's Cursor Stability affects cursor behavior. Starting from DB2 LUW 9.5 cursor can be configured to use *optimistic concurrency* without locking and so differing from the isolation level of the current transaction scope. We will cover this in our "RVV Paper"[12]. It is also possible to change the isolation of any SELECT statement using *isolation clause* like in the following example:

```
SELECT COUNT(*)  
FROM emp  
WHERE eyes='blue' AND hair='red'  
WITH (UR) ;
```

The isolation level in the WITH clause can be any isolation level of DB2. We will present these later.

**SQL Server** provides cursor processing in *optimistic concurrency* without locks either in WITH VALUES or WITH ROW VERSIONING mode [21]. It is also possible to define the isolation level differently from the current transaction isolation level for any *table access in a statement*, using *table hints* in a WITH clause like in the following example:

```
SELECT COUNT(*)  
FROM emp WITH (READUNCOMMITTED)  
WHERE eyes='blue' AND hair='red'
```

With the table hints, we can also change the isolation level of a Transact-SQL cursor. We will return later to these SQL Server table hints.

**Oracle's** transaction isolation levels of are based on multi-versioning so that a transaction can always only read committed data and without need to wait. Oracle's SERIALIZABLE is actually a SNAPSHOT from the database based on the timestamp of the transaction start time. With this transaction isolation level, Oracle cannot prevent other transactions from updating or deleting some of the rows seen in the snapshot. This problem can be solved only by *locking* selected rows using the FOR UPDATE clause as follows:

```
SELECT *  
FROM emp  
WHERE eyes='blue' AND hair='red'  
      FOR UPDATE ;
```

## Holdable Cursor and Locks

By default a cursor will be closed automatically at the end of the transaction in which it has been opened, but if the cursor is defined using the WITH HOLD option, the cursor remains open on COMMIT, and in some DBMS systems also on ROLLBACK. On locking systems the locks acquired for the cursor will be released on COMMIT, except the lock of the current row of the cursor. However, update or delete by the WHERE CURRENT OF form of the cursor succeeds only on rows fetched in the current transaction only. A holdable cursor must be closed explicitly.

## Some Contention Scenarios

Following concurrency scenarios may lead to concurrency conflicts, and the typical lesson in textbooks is “Don’t do that!”. However, our pedagogical point is to experiment with these conflicts, so that we better understand these cases. There are many other

potential scenarios for concurrency conflicts than those which we present below, and we need to cope with them.

### **Scenario 1: Contention on SELECT-UPDATE accessing the same resource**

- *The concurrency control can affect the final results*

In Appendix 1 we present a simple scenario of accessing the same row by SELECT-UPDATE command sequences in two concurrent SQL sessions and using OCC implementation of Pyrrho DBMS. This can be considered as a baseline scenario of a true ACID transaction, fully isolated from concurrent sessions, - and failing in case of conflicts found on validation at the COMMIT phase. We encourage the readers to test the case with different isolation levels of DB2, Oracle, and SQL Server.

Since concurrent sessions usually don't proceed synchronously, it is possible that for example steps 2 and 3 will be done in different order, and this will affect the final result.

If we test the SELECT-UPDATE scenario of Figure 1 which leads to Lost Update problem without any concurrency control mechanism as presented by Date, we will notice that the results will vary depending on the DBMS system, the concurrency control mechanism (LSCC or MVCC), and the isolation levels used. We encourage the readers to test the case with different isolation levels of DB2, Oracle, and SQL Server.

### **Scenario 2: Contention when updating multiple resources in different order**

- *The concurrency control affects the behavior on exceptions*

A typical piece of advice for trying to avoid deadlocks while using some LSCC based system is to arrange transaction logic so that updates of tables/rows are always done in the same order. Of course this is not always possible, and we may not even know of the competing transactions.

In Appendix 1 we also present a basic test of updating two rows in different order by concurrent sessions using OCC implementation of Pyrrho DBMS. The scenario leads to a concurrency conflict, and the first transaction to commit will win.

The readers are encouraged to test this scenario using DB2, Oracle, and SQL Server. The final result may be the same, but service provided by the DBMSs will differ.

For more detailed experimentation on SQL exceptions in two concurrent JDBC sessions which update two rows in opposite order can be conducted using the Java program of Appendix 5. The readers are encouraged to test how does DB2, Oracle, or SQL Server behave in case of concurrency conflicts, and will the presented Retry data access pattern solve the conflict for the application needs.

The contention can appear even if one transaction is applying updates and the others read accesses like in our example in Figure 8.

### Scenario 3: Contention as side effect of referential integrity checks

- *The concurrency control can affect the final results*

Referential integrity checking means that DBMS need to do some extra reading. This may affect also concurrency control. For example, try to test the behavior of SQL Server using the following tables and concurrent transactions:

```
CREATE TABLE Parent (  
  pid    INT NOT NULL PRIMARY KEY,  
  pv     INT  
)  
CREATE TABLE Child (  
  cid    INT NOT NULL PRIMARY KEY,  
  cv     INT,  
  fk     INT  
          FOREIGN KEY REFERENCES Parent  
)  
GO  
INSERT INTO Parent VALUES (1, 0)  
INSERT INTO Child  VALUES (1, 4, 1)  
GO  
-- Process A  
BEGIN TRANSACTION  
UPDATE Parent SET pv= pv+1 WHERE pid = 1  
-- Process B  
BEGIN TRANSACTION  
SET TRANSACTION ISOLATION LEVEL READ COMMITTED  
INSERT INTO Child  VALUES (2, 5, 1)
```

What will happen?

What if this were applied to DB2 or Oracle?

You can find these hands-on exercises and some more in the hands-on exercise collection in the accompanying file CCLab1.pdf.