

DB2[®] Locking for Performance

Version/Revision# 1

Module ID 10305

Length 1 hour



Table of Contents

1	Objectives	3
2	Lab Setup	3
2.1	Login to the Virtual Machine	3
2.2	Create the SAMPLE Database.....	4
2.3	Modifying Lab Environment.....	4
3	Identifying Locking Issues.....	4
3.1	Enabling a locking event monitor (LOCKWAITS)	4
3.2	Using Snapshot Administrative Views	5
3.3	Review Details of Locking Report	7
4	Event Monitor for Locking – Deadlock detection and formatting to relational tables	8
4.1	Create Event Monitor	9
4.2	Simulating deadlocks	10
5	Using Currently Committed Isolation Level	12
5.1	Currently Committed Scenario	13
5.2	Without Currently Committed Isolation	13
5.2.1	<i>Turning off Currently Committed</i>	<i>13</i>
5.2.2	<i>Execute an update query</i>	<i>14</i>
5.2.3	<i>Execute a read query against the updated row.....</i>	<i>15</i>
5.2.4	<i>Releasing the lock.....</i>	<i>15</i>
5.3	With Currently Committed Isolation	15
5.3.1	<i>Turning on Currently Committed</i>	<i>15</i>
5.3.2	<i>Execute an update query</i>	<i>16</i>
5.3.3	<i>Execute a read query against the updated row.....</i>	<i>16</i>
6	Summary	17
7	Scripts provided for this lab	17
7.1	snapdblocks.sql	17
7.2	lockwaits.sql	17
7.3	createEventMonitorLocking.db2.....	18
7.4	call_EF.sh.....	18

1 Objectives

After completion of this lab, the student should be able to:

- § Use snapshots, event monitors, administrative views and db2pd to identify locking issues through the reporting provided by key monitoring elements.
- § Be able to identify the type of lock event causing a performance slowdown, identify the SQL statement(s) involved and the locks being requested and encountered by applications.

2 Lab Setup

2.1 Login to the Virtual Machine

To access the lab environment, perform the following steps:

1. Login to the VMware virtual machine using the following information:

User: **db2inst1**
Password: **password**
2. Open a terminal window as by right-clicking on the **Desktop** area and choose the “**Open Terminal**” item.

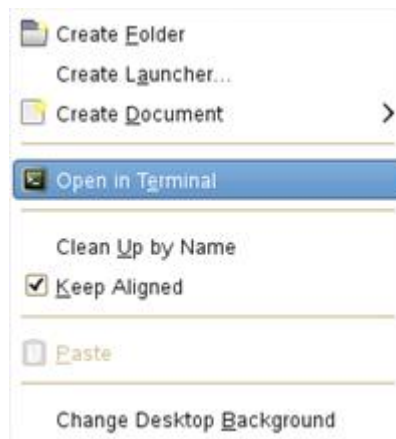


Figure 1 – Opening a Terminal

3. Start up DB2 Server by typing “**db2start**” in the terminal window.

2.2 Create the SAMPLE Database

Before beginning the exercise, let's create DB2's SAMPLE database using the following command:

```
db2sampl -force
```

2.3 Modifying Lab Environment

It will be necessary to modify the default monitor switches at the instance level, so all snapshot administrative views will report all monitor elements accurately.

```
db2 update dbm cfg using DFT_MON_LOCK on
```

The database used in this lab will need to have CUR_COMMIT turned off. CUR_COMMIT provides significant relief from locking issues in a variety of scenarios.

Newly created databases will have the feature turned on by default. Any database updated from V9.5 or earlier will not have CUR_COMMIT enabled.

```
db2 update db cfg for sample using CUR_COMMIT off
```

Let's stop and restart the instance so are modified parameters are in effect before proceeding.

```
db2stop force  
db2start
```

3 Identifying Locking Issues

3.1 Enabling a locking event monitor (LOCKWAITS)

Lab Tip: Use the up arrow on your keyboard to recall commands.

So we may review detailed information about locking events we will create an event monitor for locking called lockevmon:

```
db2 connect to sample  
db2 "CREATE EVENT MONITOR lockevmon  
    FOR LOCKING WRITE TO UNFORMATTED EVENT TABLE (TABLE lockwaits)"  
db2 "set event monitor lockevmon state 1"
```

Update the DB CFG parameter MON_LOCKWAIT to control the amount of trace data and history to be captured. HIST_AND_VALUES will include the contents of a history buffer. Alternatively, you can set the parameter to WITHOUT_HIST to trace only the data about lock events when the lock event occurs and not any past activity history:

```
db2 update db cfg for sample using MON_LOCKWAIT hist_and_values
```

Update the DB CFG parameter to establish a threshold time limit of 5 seconds for how long the application is suspended in a lock-wait status before being included in the report.

```
db2 update db cfg for sample using MON_LW_THRESH 5000000
```

3.2 Using Snapshot Administrative Views

Let's take a look at a typical lock wait scenario. To simulate database activity open a terminal window and enter the following commands:

```
db2 -c "UPDATE EMPLOYEE SET SALARY=1000 WHERE EMPNO = '000010'"
```

Open another terminal, and issue these commands:

```
db2 connect to sample  
  
db2 -c "UPDATE EMPLOYEE SET SALARY=1000 WHERE EMPNO = '000010'"
```

You will notice that there is no indication of any successful execution. It seems to 'hang'. Do not close this window. You can minimize it since it will not be used until the end of the exercise.

After the scenario has been running for about 5 seconds, you should have collected enough data in the lock event monitor that you started in section 3.1. To avoid having a large amount of data to browse through in the output file, you can turn the event monitor off by setting it to 0. (Do NOT stop the Workload Expert scenario running in the first terminal window. You will need it running for the next several exercises). Open a **third terminal** window, connect to the database, and enter:

```
lockdir (aliases cd /home/db2inst1/Documents/LabScripts/db2pt/Locking)  
db2 connect to sample  
db2 "set event monitor lockevmon state 0"
```

You can always query the database to see your event monitors, and whether they are on or off, by entering:

```
db2 "select evmonname, event_mon_state (evmonname) from  
syscat.eventmonitors"
```

A database snapshot is a good place to start when investigating locking issues. The snapshot administrative view SYSIBMADM.SNAPDB allows database level monitor elements to be returned via SQL SELECT statements. The use of queries to retrieve snapshot monitor data allows filtering to focus on specific monitor data.

In the second terminal window, connect to the database and execute the following:

```
lockdir (aliases cd /home/db2inst1/Documents/LabScripts/db2pt/Locking)  
db2 connect to sample  
db2 -tf snapdblocks.sql
```

LOCK_WAITS	DEADLOCKS	LOCK_ESCALS	LOCK_TIMEOUTS
1	0	0	0
1 record(s) selected.			

i Note: Your results may be a little different.

This query will return database level monitoring elements for each of the locking scenarios; lock waits, deadlocks, lock escalations and lock timeouts. It is easy to see that LOCK_WAITS are occurring. Why?

This can be explained by reviewing the database configuration.

```
db2 get db cfg for sample | grep LOCKTIMEOUT
```

Lock timeout (sec)	(LOCKTIMEOUT) = -1
--------------------	--------------------

The DB CFG parameter LOCKTIMEOUT has not been changed from the default value of -1.

When LOCKTIMEOUT is set to -1 all LOCK_WAIT conditions will wait indefinitely for the encountered lock to be released. Thus all queries will become suspended in a lock-wait state until the lock encountered is released. If you plan on setting this parameter to a specific time period, make sure that the application is prepared accordingly. DB2 will return an error code to the application (SQL0911). From the command line interface it looks like this:

```
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 "68". SQLSTATE=40001
```

```
db2 "select locks_held, lock_waits from sysibmadm.snapdb"
```

Sample Output:

```
db2inst1@db2v10:~/Documents/LabScripts/db2pt/Locking> db2 "select locks_held, lock_waits from
sysibmadm.snapdb"
```

LOCKS_HELD	LOCK_WAITS
13	14

i Note: Your results may be a little different.

If the number reported under `LOCK_WAITS` is greater than zero, then that is the number of times that applications or connections waited for locks.

To narrow the scope, and filter database locks to only the applications in a lock wait status, use the Snapshot administrative view `SYSIBMADM.MON_LOCKWAITS`:

```
db2 -tf lockwaits.sql
```

TABSCHEMA	TABNAME	REQ_AGENT_TID	LOCK_OBJECT_TYPE	LOCK_MODE	LOCK_MODE_REQUESTED	HLD_APPLICATION_HANDLE
DB2INST1	EMPLOYEE	43	ROW	X	X	19

1 record(s) selected.

i Note: Your results may be a little different.

Some administrative views like `SYSIBMADM.LOCKWAITS` have been deprecated in DB2 10, so it is necessary to use new administrative views like `SYSIBMADM.MON_LOCKWAITS`. You can review the following link:

<http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/topic/com.ibm.db2.luw.sql.rtn.doc/doc/r0023171.html>

There is one application waiting. It has encountered an exclusive lock (X) placed by another application. One is attempting to acquire a X-lock, which would be expected when issuing an INSERT, UPDATE or DELETE.

3.3 Review Details of Locking Report

To obtain the locking event report, use the `db2evmonfmt` tool. This tool is included with DB2 but is not compiled. You will find the source in the Instance's home directory under `samples/java/jdbc`.

For more details refer to the “**db2evmonfmt tool for reading event monitor data**” topic in Information Center v10.

For this lab, the utility has already been compiled and is in the Locking directory. Redirect the output to a file name to ease review using a text editor.

```
~/sqllib/java/jdk64/jre/bin/java db2evmonfmt -d sample -ue lockwaits -ftext  
-u db2inst1 -p password > db2evmonfmt.out
```

```

db2evmonfmt.out
=====
Activity ID : 3003
User ID : 1
Package Name : SYSDIAG200
Package Schema : SYSDIAG
Package Version : 1
Package Token : SYSDIAG101
Package Epoch : 65
Reset value : none
Incremental Bind : no
Eff isolation : CS
Eff timeout : 0
Eff lock level : -1
Reset timeout : 0
Reset query th : 0
Reset nesting level : 0
Reset invocation ID : 0
Reset source ID : 0
Reset package ID : SYSDIAG200
State type : dynamic
State operation : SQL, Insert/Update/Delete
State text : update employee set lastname = 'PORTER' where empno = '000042'

```

Figure 2 – db2evmonfmt output

i Note: You can find the db2evmonfmt.out output file in ~/sqllib/java/jdk32/jre/bin/java directory.

As good practice, so that the lab can be rerun, turn off and drop the event monitor lockevmon, and drop the lock event table lockwaits:

```

db2 set event monitor lockevmon state 0
db2 drop event monitor lockevmon
db2 drop table lockwaits

```

In order to continue with the next exercise is necessary to update the DB CFG parameter MON_LOCKWAIT with its previous value:

```

db2 update db cfg for sample using MON_LOCKWAIT none
db2stop force
db2start

```

4 Event Monitor for Locking – Deadlock detection and formatting to relational tables

In previous releases of DB2, pre-V9.7, if you wanted to capture deadlock events, you had to issue the CREATE EVENT MONITOR FOR DEADLOCKS statement or check the output files written by the DB2DETAILDEADLOCK event monitor.

Now if you want to monitor deadlock events in DB2 10, using the `CREATE EVENT MONITOR FOR LOCKING` statement is the recommended method. This event monitor not only monitors for deadlocks but also lock timeouts and excessive lock wait occurrences.

In the previous exercise, you used this same approach (event monitor for locking) when monitoring `lockwaits`, now we will focus on deadlocks. Previously we used the `db2evmonfmt` tool to generate formatted output and details about the `lockwaits`, in this exercise you will use a system based stored procedure to store this information in a set of relational tables then query these tables.

In this exercise you will:

1. Create the event monitor for locking. Output for this monitor is generated in the unformatted event (UE) table structure. The create statement can specify placement of the UE table in a separate tablespace, a preferable technique to use when you want to reduce I/O contention.
2. Generate transactions that will create deadlocks. The deadlock events will be written out to the UE tables
3. Format UE table data into relational table structure using a stored procedure developed expressly for this purpose.
4. Use SQL, via a script, to query the relational tables.

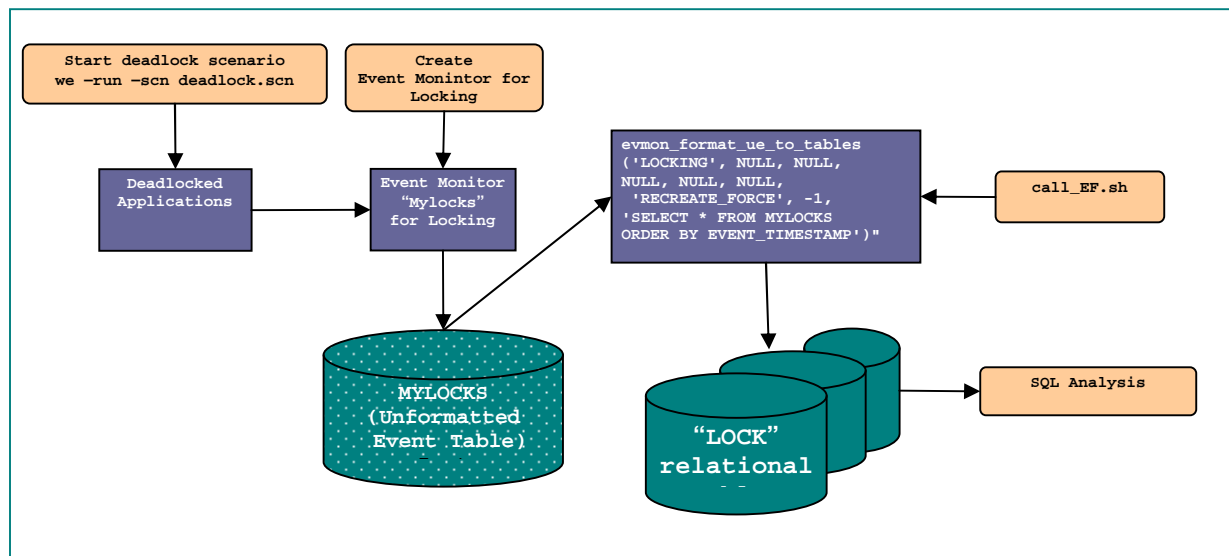


Figure 3 – Exercise diagram

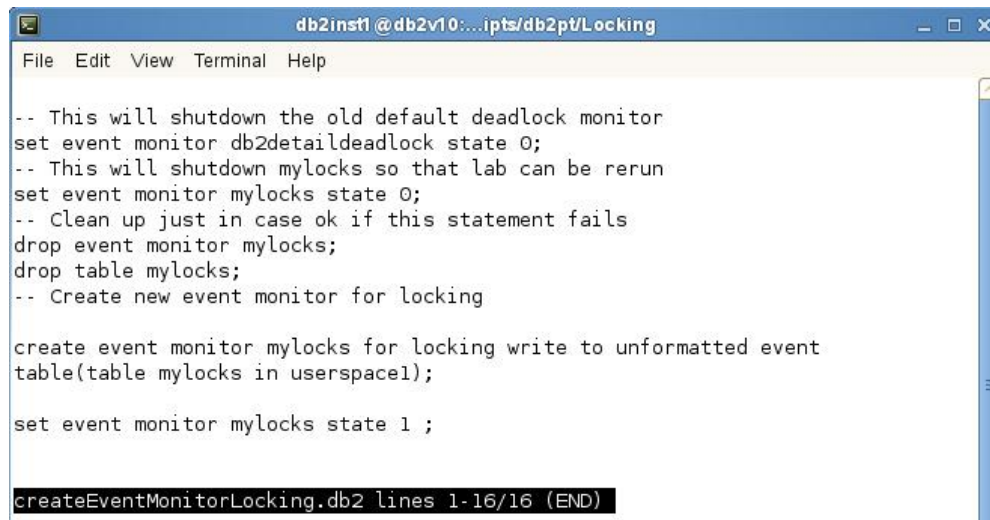
4.1 Create Event Monitor

First you will execute a DB2 script file to create the event monitor for locking. Open up a terminal window and change your current directory to:

```
lockdir (aliases cd /home/db2inst1/Documents/LabScripts/db2pt/Locking/)
```

Before executing, take a look at the script; first the old event monitor is disabled, then an event monitor for locking is created with specific designations for the table name (mylocks) and table space location (userspace1) before the new monitor is turned on.

```
less createEventMonitorLocking.db2
```

A screenshot of a terminal window titled "db2inst1 @db2v10:...ipts/db2pt/Locking". The window contains the following text:

```
-- This will shutdown the old default deadlock monitor
set event monitor db2detaildeadlock state 0;
-- This will shutdown mylocks so that lab can be rerun
set event monitor mylocks state 0;
-- Clean up just in case ok if this statement fails
drop event monitor mylocks;
drop table mylocks;
-- Create new event monitor for locking

create event monitor mylocks for locking write to unformatted event
table(table mylocks in userspace1);

set event monitor mylocks state 1 ;

createEventMonitorLocking.db2 lines 1-16/16 (END)
```

Figure 4 – createEventMonitorLocking.db2 script

i **Note:** You can use the arrow keys to scroll up and down in the script and type the letter “q” to exit anytime.

Connect to the database and execute the script:

```
db2 connect to sample
db2 -tvf createEventMonitorLocking.db2
```

i **Note:** Ignore the SQL0204N errors; they are a result of DROP statements which precede the CREATE and SET statements in the script so you can repeat execution.

4.2 Simulating deadlocks

We will use two terminals to create a deadlock scenario. Locking information will be collected in the unformatted event tables (UE).

In the first terminal window where you created the event monitor, enter the following command:

```
db2 +c "UPDATE EMPLOYEE SET SALARY=1000 WHERE EMPNO = '000010' "
```

Open another terminal window, enter the following commands:

```
db2 connect to sample
db2 +c "UPDATE EMPLOYEE SET SALARY=1000 WHERE EMPNO = '000020'"
db2 +c "SELECT * FROM EMPLOYEE WHERE EMPNO = '000010'"
```

You will notice that the last statement will 'hang'. Return to the first terminal window, and issue this command:

```
db2 +c "SELECT * FROM EMPLOYEE WHERE EMPNO = '000020'"
```

This will also 'hang'. Wait for a few seconds, and you will see an error message SQL0911N. Now check that deadlocks were generated and captured into the mylocks table (defined in create event monitor statement).

```
db2 connect to sample
db2 "select count(*) from mylocks"
```

You should see a count of some rows in the mylocks table. (If you don't, try re-executing the deadlock scenario).

Since the mylocks table is unformatted, you will execute a stored procedure to format the table into relational tables.

Use the call_EF.sh script to invoke EVMON_FORMAT_UE_TO_TABLES stored procedure. Three tables will be created to store the unformatted data in a relational format: LOCK_EVENT, LOCK_PARTICIPANTS and LOCK_PARTICIPANT_ACTIVITIES.

```
./call_EF.sh
```

code in call_EF.sh :

```
db2 "connect to sample"
db2 "call evmon_format_ue_to_tables('LOCKING', NULL, NULL, NULL, NULL,
NULL, 'RECREATE_FORCE', -1, 'SELECT * FROM MYLOCKS ORDER BY
EVENT_TIMESTAMP')"
```

Be sure to look for "Return Status = 0".



Figure 6 – call_EF.sh output

The last step is to query the newly created (and populated) relational tables. You can use a provided script that will pick out the first ID (xmlid) that is returned from the `LOCK_EVENT` table and query the other tables based on that ID.

```
db2 connect to sample
./query_lockevent_tables.sh
```

PARTICIPANT_NO	PARTICIPANT_NO_HOLDING_LK 3	EFFECTIVE_ISOLATION	STATEMENT
2	1 db2bp	CS	SELECT * FROM EMPLOYEE WHERE EMPNO = '000020'
1	2 db2bp	CS	SELECT * FROM EMPLOYEE WHERE EMPNO = '000020'
2	1 db2bp	CS	SELECT * FROM EMPLOYEE WHERE EMPNO = '000010'
1	2 db2bp	CS	SELECT * FROM EMPLOYEE WHERE EMPNO = '000010'

4 record(s) selected.

Figure 7 – query_lockevent_tables.sh output

The query's result displays the participants in a deadlock event, the application(s) that were involved, the SQL code, and isolation levels used. This data provides a “good start” to resolving a deadlock issue.

5 Using Currently Committed Isolation Level

Lock timeouts and deadlocks can occur under the cursor stability (CS) isolation level with row-level locking, especially with applications that are not designed to prevent such problems. Some high throughput database applications cannot tolerate waiting on locks that are issued during transaction processing, and some applications cannot tolerate processing uncommitted data, but still require non-blocking behavior for read transactions.

Under the *Currently Committed* isolation level, only committed data is returned, as was the case previously, but now readers do not wait for updaters to release row locks. Instead, readers return data that is based on the currently committed version; that is, data prior to the start of the write operation.

In DB2 10, *Currently Committed* is turned on by default for new databases. This allows any application to take advantage of the new behavior without any modification. The database configuration parameter `cur_commit` can be used to override this behavior. This might be useful, for example, in the case of applications that require the pre-9.7 behavior of blocking on writers to synchronize internal logic. Upgraded databases have `cur_commit` disabled by default.

Currently Committed semantics apply only to read-only scans that do not involve catalog tables or the internal scans that are used to evaluate constraints.

Note that, because *Currently Committed* is decided at the scan level, a writer's access plan might include *Currently Committed* scans. For example, the scan for a read-only sub-query can involve *currently committed* semantics

Log data is required for retrieving the currently committed image of the row. Depending on the workload, this can have an insignificant or substantial impact on the total log space and log buffer space required. The requirement for additional log space does not apply when `cur_commit` is disabled.

5.1 Currently Committed Scenario

Consider the following scenario, in which deadlocks are avoided under the Currently Committed isolation level. In this scenario, two applications update two separate tables, but do not commit. Each application then attempts to read (with a read-only cursor) from the table that the other application has updated.

Time	Transaction A	Transaction B
1	<code>update T1 set col1 = ? where col2 = ?</code>	
2		<code>update T2 set col1 = ? where col2 = ?</code>
3		<code>select col1, col5, from T1 where col5 = ? and col2 = ? waiting for A to commit</code>
4	<code>select col1, col3, col4 from T2 where col2 >= ? waiting for B to commit</code>	

Without Currently Committed, transactions running under the Cursor Stability isolation level might create a deadlock, causing one of the transactions to fail. This happens when each transaction needs to read data that is being updated by the other.

Using Currently Committed, if the query (of either application) happens to require the data currently being updated by the other transaction, that transaction does not wait for the lock to be released, avoiding a deadlock since the previously committed version of the data is located and used.

5.2 Without Currently Committed Isolation

In this lab, we'll demonstrate the effect of the Currently Committed feature. To do so, we will create a scenario where a potential read / write block can happen when 2 queries are running concurrently. We will then compare the difference in results and execution time as we toggle the parameter `CUR_COMMIT`.

We will use command line processor (CLP) to simulate the applications accessing the database at the same time.

5.2.1 Turning off Currently Committed

First, let's examine the existing setting for currently committed. From within a Terminal window, type in the following:

```
db2 get db cfg for sample
```

The `CUR_COMMIT` parameter is located near the end of the list. It should display as `DISABLED`, since we turned it off earlier in the exercise. If it is `ON`, disable it by issuing the following command:

```
db2 update db cfg for sample using CUR_COMMIT disabled
```

After changing the value, we need to disconnect all connections, thus deactivating the database and allowing for the new value to take effect upon the next activation or connect.

In both terminal windows, issue a connect reset:

In the first terminal window:

```
db2 connect reset
```

In the second terminal window:

```
db2 connect reset
```

5.2.2 Execute an update query

In order to mimic the behavior of a long running transaction, we first need to disable CLP's auto-commit feature, which is ON by default. When auto-commit is active, CLP automatically issues a COMMIT after every executed SQL statement. Therefore, we need to disable it so we are able to specify when the transaction will be committed. In **terminal A**, enter the CLP prompt by typing the command below. The "+c" option will disable the auto-commit feature for this session.

```
export DB2OPTIONS="+c"
```

You can check that the auto-commit feature is off by executing the command below:

```
db2 list command options
```

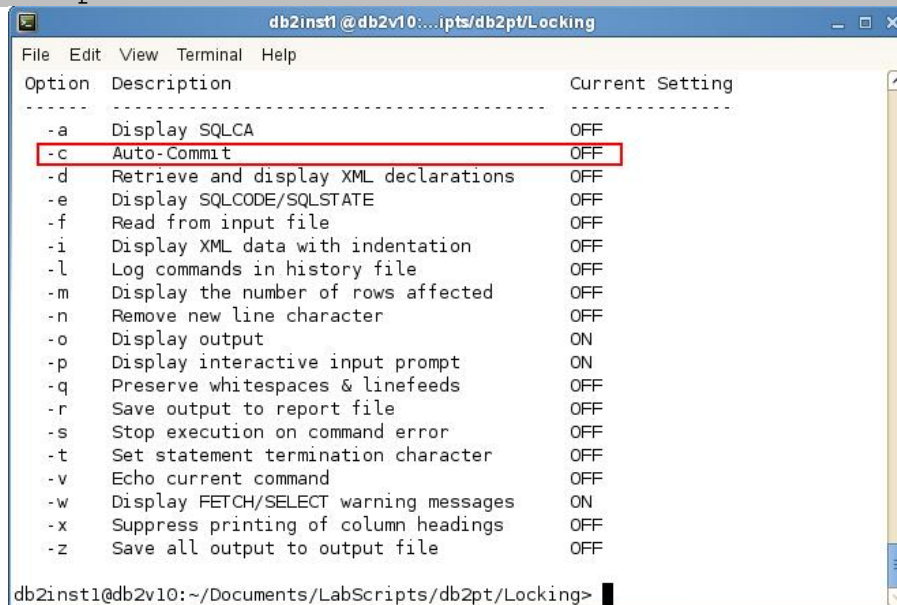


Figure 8 – List command options

Since auto-commit is OFF, from now on all SQL statements that you execute will be part of the same transaction until you issue a "commit" or "rollback". Now, connect to the database:

```
db2 connect to sample
```

We will then execute an update query which will put a lock on the rows for as long as the transaction is not committed.

```
db2 "update employee set lastname = 'Porter' where empno = '000340'"
```

5.2.3 Execute a read query against the updated row

Terminal B will act as the second application trying to access the table. Similar to the first terminal, we will connect to the database "sample" as user "db2inst1" with password "password" by typing in the command:

```
db2 connect to sample
```

Next, we will launch a query that will read the data locked by **Terminal A**.

```
time db2 "select empno, firstnme, lastname from employee"
```

The time command will allow us to quantify the wait time. We can see that the query waits and does not return any result. In fact, it is waiting on the locks of the first terminal window's UPDATE statement to be released.

5.2.4 Releasing the lock

With the two terminal windows open beside each other, we will observe the effect of committing the UPDATE of the first terminal window. In Terminal A, commit the transaction by executing the following command:

```
db2 commit
```

You can see the second terminal windows query is instantly returned with the updated values. The locks placed by the first terminal window's update have been released with the issuance of the commit statement.

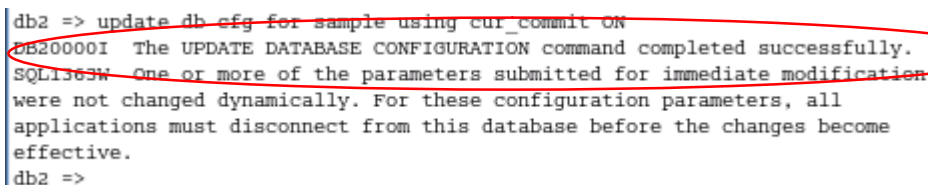
5.3 With Currently Committed Isolation

We will repeat the procedure again but this time with the Currently Committed feature turned on. The objective is to see the difference in the time it took for the second query to return and the actual values being returned.

5.3.1 Turning on Currently Committed

In **Terminal A**, we will use the command to turn on currently committed:

```
db2 update db cfg for sample using CUR_COMMIT on
```



```
db2 => update db cfg for sample using cur_commit on
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
SQL1363W  One or more of the parameters submitted for immediate modification
were not changed dynamically. For these configuration parameters, all
applications must disconnect from this database before the changes become
effective.
db2 =>
```

Figure 9 – Update parameter output

After changing the value, we need to disconnect all connections thus deactivating the database allowing for the new value to take effect upon the next activation.

In **both** terminal windows issue a connect reset: In the first terminal window:

```
db2 connect reset
```

In the second terminal window, execute:

```
db2 connect reset
```

5.3.2 Execute an update query

Now, let's create the exact same scenario. Similar to the previous section, update a row of data in the table in the first terminal window and attempt to access the same row in the second terminal window.

In the first terminal window:

```
db2 connect to sample  
db2 "update employee set lastname = 'Selleck' where empno = '000340'"
```

5.3.3 Execute a read query against the updated row

In the second terminal window, reconnect to the database and try to retrieve the values from the table:

```
db2 connect to sample  
time db2 "select empno, firstnme, lastname from employee"
```

Notice the amount of time the query took to return this time. The query returned instantly because there was no access block to the data. Also, notice the values returned were not from the most recent update since the update has not yet been committed, but from the last committed update.

In Terminal A, commit the update by typing in the command

```
db2 commit
```

Switch the focus back to Terminal B. We want to execute the selection query again by pressing the up arrow button once to retrieve the last executed command, and then press Enter. If you cannot find the last command, type in

```
time db2 "select * from employee"
```

Notice the values returned this time reflects our last update since the transaction in terminal A has ended and the updates committed to the database.

To finalize the lab, enable auto-commit so that CLP will automatically issues a `COMMIT` after every executed SQL statement. Do that by typing the command below. The `-c` option will enable the auto-commit feature for this session.

```
export DB2OPTIONS="-c"
```

You can check that the auto-commit feature is off by executing the command below:


```
db2 list command options
```

Terminate the database connections in the first terminal window:

```
db2 connect reset
```

Then, terminate the database connection in the second terminal window:

```
db2 connect reset
```

6 Summary

You can now identify and diagnose locking issues that can affect the performance of your databases and applications.

Now you have some tips which you can apply in your environment in order to design better databases to avoid locking issues.

Now you can use DB2 tools such as event monitors and administrative views to monitor your databases.

7 Scripts provided for this lab

The source for the scripts used in this workshop is provided here for you to copy/paste into your own environment. You might need to modify them to meet your specific environment.

7.1 snapdblocks.sql

```
#-----  
SELECT LOCK_WAITS, DEADLOCKS, LOCK_ESCALS, LOCK_TIMEOUTS FROM SYSIBMADM.SNAPDB;
```

7.2 lockwaits.sql

```
SELECT SUBSTR(TABSCHEMA,1,8) AS TABSCHEMA,  
       SUBSTR(TABNAME,1,15) AS TABNAME,  
       REQ_AGENT_TID,  
       LOCK_OBJECT_TYPE, LOCK_MODE,  
       LOCK_MODE_REQUESTED,  
       HLD_APPLICATION_HANDLE  
FROM SYSIBMADM.MON_LOCKWAITS;
```

7.3 createEventMonitorLocking.db2

```
-- This will shutdown the old default deadlock monitor
set event monitor db2detaildeadlock state 0;
-- This will shutdown mylocks so that lab can be rerun
set event monitor mylocks state 0;
-- Clean up just in case ok if this statement fails
drop event monitor mylocks;
drop table mylocks;
-- Create new event monitor for locking
create event monitor mylocks for locking write to unformatted event
table(table mylocks in userspace1);
set event monitor mylocks state 1 ;;
```

7.4 call_EF.sh

```
#-----
db2 "connect to db2pt97"
db2 "call evmon_format_ue_to_tables('LOCKING', NULL, NULL, NULL, NULL, NULL,
'RECREATE_FORCE', -1, 'SELECT * FROM MYLOCKS ORDER BY EVENT_TIMESTAMP')"
exit
```



© Copyright IBM Corporation 2014
All Rights Reserved.

IBM Canada
8200 Warden Avenue
Markham, ON
L6G 1C7
Canada

IBM, the IBM logo, ibm.com and Tivoli are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at ibm.com/legal/copytrade.shtml

Other company, product and service names may be trademarks or service marks of others.

References in this publication to IBM products and services do not imply that IBM intends to make them available in all countries in which IBM operates.

No part of this document may be reproduced or transmitted in any form without written permission from IBM Corporation.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT.

IBM products are warranted according to the terms and conditions of the agreements (e.g. IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided.