Diagnose and resolve lock problems with DB2 for Linux, UNIX, and Windows

Bill Wilkins

Yasir Warraich

DB2 Business Partner Enablement

IBM Canada

wilkins@ca.ibm.com

warraich@ca.ibm.com

Notices and Trademarks

Introduction 4
Overview of the lock issues 4
Lock escalations 4
Lock waits 5
Deadlocks 5
The tools for monitoring locks -- what do they tell you? 6
(1) `LIST APPLICATIONS` command 6
(2) Snapshot monitoring 7
(3) Snapshot table functions 13
(4) Event monitoring 14
(5) Administration notification log ("notify log") 15
(6) Health Center 16
(7) Operating system facilities 17
Resolving the lock issues 17
Reducing locking 17
Avoiding lock escalations 18
Resolving lock waits 19
Resolving deadlocks 21
Locking notes specific to partitioned databases 22
Conclusion 23
References 24
Appendix A -- Case Study 25
Appendix B -- Sample Scripts 31
Appendix C -- Sample Lock Snapshot Output 34

Introduction

This article is a "how-to" guide for dealing with issues related to locking in DB2 for Linux, UNIX, and Windows. It's not intended to be a complete reference on locking, but rather a supplement to existing information. Before reading this article, you should be familiar with the following material, which is available from the locations given in the "References" section:

- The section on locking in the *DB2 Performance Guide*. This covers details on the types of locks, when they are obtained, and compatibility between them.
- The Problem Determination Tutorial sections on locking. These provide basic instruction on the steps in diagnosing lock issues.

To quote the DB2 Engine Problem Determination tutorial: "DB2 is a multi-user database and as such the database engine provides a locking mechanism to avoid resource conflicts and to provide data integrity." For many applications, the locking mechanism works completely transparently, but for others, issues such as lock waits, deadlocks, or lock escalations can occur.

In this article, we discuss lock waits, deadlocks, and escalations, and what to do about them. Other lock-related issues, such as access to uncommitted data, nonrepeatable reads, and phantom rows, are not addressed here. Their implications -- unexpected or undesirable results -- are entirely different from the performance impacts of the issues discussed here. Those issues can be addressed by choosing an isolation level appropriately, as described in the *Performance Guide*.
This article focuses on application development environments. Most of the material also applies to production environments, but in that environment there is usually much less opportunity to deal with lock issues through application changes.

Overview of the lock issues

Let's briefly review the various locking issues, their meanings and implications. Later we'll talk about how to resolve each of them.

Lock escalations

Each lock uses a certain amount of memory. Lock escalations occur when DB2 automatically replaces many row locks on a table with one table-level lock, thereby freeing up lock memory. Although escalations represent a waste of CPU (from not only the cost of releasing the locks, but the cost to acquire them in the first place), a concurrency issue may occur in one of these ways:

- Escalation may be prevented, because the table lock would conflict with existing row or table lock(s) held by other transactions; or
- The creation of the table lock may cause lock contention for other transactions.

Lock waits

A *lock wait* occurs when one transaction tries to acquire a lock whose mode conflicts with a lock held by another transaction, according to the Lock Type Compatibility table in Chapter 5 of *Performance Guide*. Two transactions can have lock conflicts with one another even if they are running the same application under the same authorization name, and even if they are threads of the same process. As long as the DB2 work is being done in separate connections (separate units of work), there can be lock contention.

On the other hand, if only one application is connected to the database (such as a batch application), lock waits cannot occur, and any locking that's done is a waste of resources, so you should try to avoid locking entirely through the techniques given below under the heading "Avoiding Lock Escalations".

A lock wait continues until the locks held by the transaction causing the conflict are released. Most commonly this occurs when the transaction is committed, but it also happens when the transaction is rolled back, such as by being forced through the FORCE APPLICATION command, or through a lock timeout. There can potentially be a long time before these occur, with the obvious effect on the response time of the waiting transaction, so lock waits are usually the most important type of lock issue to be dealt with.

A lock timeout is one way of dealing with lock waits. By changing the value of the database configuration parameter LOCKTIMEOUT from its default value of -1 (meaning "wait forever") to 0 or higher, you cause transactions to be rolled back (SQLCODE -911 with reason code 68) when they are in a lock wait for that number of seconds. Changing the value of LOCKTIMEOUT can avoid lengthy lock waits, but:

- It does not solve the problem of why the lock waits occur
- The application may not deal properly with the -911's and rollbacks, and
- If the application retries the rolled back transaction, it may encounter the same lock wait.

For these reasons, if you're developing or testing an application, we suggest that you try to eliminate lock waits before changing LOCKTIMEOUT, and if you do change it, be sure to test how the application responds. LOCKTIMEOUT can also be set at the session level using:

```
SET CURRENT LOCK TIMEOUT WAIT X
```

Where x is the number of seconds. This allows you to override database level LOCKTIMEOUT level value and have more control. You could use different LOCKTIMEOUTS for different applications and sessions.

Deadlocks

A *deadlock* is a special case of a lock wait in which two or more applications cannot proceed because each is waiting for a lock held by one of the others. They are resolved automatically by DB2's deadlock detection, which is activated with the frequency specified by the value (in milliseconds) of the database configuration parameter DLCHKTIME.

Deadlocks are certainly undesirable, but they are sometimes viewed as disastrous occurrences that must be eliminated at all costs. Unless deadlocks occur frequently or in long-running transactions, it is more productive to treat them like any other type of error that an application can encounter: ensure that the application is written to be aware of them, and take appropriate action (usually retrying the rolled-back transaction).

The tools for monitoring locks -- what do they tell you?

A wide variety of tools can be used to obtain information on locking. In this section, we'll take a look at some of the tools that are available and what they provide. These tools can be used as part of regular monitoring of a test or production application, or specifically to analyze response time issues reported by users, testers, or performance analysts.

There are a number of ways to perform most monitoring tasks. This article focuses on command line and file-based approaches, but I'll also mention the available alternatives.

(1) `LIST APPLICATIONS` command

This DB2 command, when issued with the SHOW DETAIL clause, shows the status of each application. This is a good first step to take when you suspect that a lock wait condition exists.
(The GUI alternative is: **Control Center > instance name** > **Applications**. However, not all columns from LIST APPLICATIONS SHOW DETAIL are reported by the Control Center.)

Because the command output is over 240 bytes wide (with SHOW DETAIL), there's no example shown here. For the purposes of this article you should focus on these output columns:

- "**Status**": A value of "Lock-wait" means the application is blocked by a lock held by a different application. Don't be confused by a value of "UOW Waiting": it means that the application (unit of work) is in progress, *not* blocked on a lock, but is just not doing any work at the moment. However, when a lock conflict does occur, an application holding locks that others are blocked on will usually have a status of "UOW Waiting", so that status is not necessarily an innocuous one.
- "**Status Change Time**": This is of particular interest for an application with Lock-wait status: it shows when the lock wait began. Note that the UOW monitor switch must be on for this time to be reported.
- "**Appl. Handle**": The handle is an integer value that serves two main purposes:

  - It allows the LIST APPLICATIONS information to be correlated with the output of snapshot and event monitors discussed below.
  - It provides the value that you can use in the FORCE APPLICATION command to force an application that's holding locks which are causing contention problems.
- Various other columns identify the connection and application, including the database name.

(2) Snapshot monitoring

Snapshot monitoring provides the majority of the useful information for dealing with lock issues. Since our focus here is on test environments, use the following steps in your initial monitoring to obtain all of the available information, ignoring the minor performance impact:

- Use the UPDATE MONITOR SWITCHES command to turn on *all* of the switches.
- Run RESET MONITOR ALL to reset counters. This makes it easier to compare different snapshots and look for differences over the course of a test. For example, are there more lock waits at the end of the run than at the beginning?
- Wait for a standard length of time, such as one minute or five minutes, and then issue GET SNAPSHOT FOR ALL ON <database> .
- Repeat the previous two steps to get multiple snapshots to compare.

Now let's talk about what's available in the output of GET SNAPSHOT FOR ALL ON <database>. This command will bring back the following snapshot monitoring components in sequence. The ones in bold are the most relevant to locking, so we'll focus on those:

- **Database snapshot**
- Bufferpool snapshot
- **Dynamic SQL snapshot result**
- **Application snapshot** (one per connected application)
- Tablespace snapshot
- **Database lock snapshot**
- Table snapshot

(You can choose to obtain specific components via separate GET SNAPSHOT commands, but the less useful components tend to be the shortest, so there's not much to save by being selective.)

Database snapshot information

The database snapshot is the best place to start when investigating lock issues. The following are the pertinent lines:

```
Locks held currently                       = 8
Lock waits                                 = 0
Time database waited on locks (ms)         = 315704
Lock list memory in use (Bytes)            = 1692
Deadlocks detected                         = 0
```

```
   Lock escalations                          = 0
   Exclusive lock escalations                = 0
   Agents currently waiting on locks         = 1
   Lock Timeouts                             = 0
```

These are quite self-explanatory (see the *System Monitor Guide and Reference* if necessary), but let's discuss the most important of them:

"**Agents currently waiting on locks**"
If this number is greater than zero, there are that many applications in lock wait, and for each of them you should see an Application Snapshot with a status of "Lock-wait".

"Lock waits" and "Time database waited on locks (ms)"
These tell you how big an impact lock waits had during the measured period. (The measured period is the difference between the times in the following lines from earlier in the snapshot -- here the counting was for about 13 seconds.

```
   Last reset timestamp                      = 08-15-2003
   14:30:30.648433
   Snapshot timestamp                        = 08-15-2003
   14:30:43.414574 )
```

Remember that the "waited on" time is a total for all connected applications, so its importance is dependent on the number of users. The effort you should put into dealing with lock waits is dependent on the number of lock waits, the "waited on" time, and your application's response time requirements. If there are a few lock waits, with a short duration, and there were as many connections to the database as there will be in a production environment, then spending time on locking is probably not a good use of your time. On the other hand, the figures might indicate a clear problem or at least signal a trend that lock contention may become increasingly important as more users run the application.

Application snapshots

Each application snapshot reflects the current status of a connected application. Here's the most important subset of information for an **application in a lock wait** situation, with commentary following. The lines are shown in the order that they appear in the snapshot.

```
Application handle                        = 14
Application status                        = Lock-wait
Status change time                        = 08-15-2003 14:30:36.907312
Snapshot timestamp                        = 08-15-2003 14:30:43.414574
Time application waited on locks (ms)     = 6507
Total time UOW waited on locks (ms)       = 6507
UOW start timestamp                       = 08-15-2003 14:30:36.889356
Statement start timestamp                 = 08-15-2003 14:30:36.890986
Dynamic SQL statement text:
select * from org

  ID of agent holding lock                = 13
  Application ID holding lock             = *LOCAL.DB2.011905182946
  Lock name                                = 
0x02000200000000000000000054
  Lock attributes                          = 0x00000000
  Release flags                            = 0x00000001
  Lock object type                         = Table
  Lock mode                                = Exclusive Lock (X)
  Lock mode requested                      = Intention Share Lock (IS)
  Name of tablespace holding lock          = USERSPACE1
  Schema of table holding lock             = WILKINS
  Name of table holding lock               = ORG
  Lock wait start timestamp                = 08-15-2003 14:30:36.907318
```

The application handle is a unique identifier for the connection. It matches the handle number reported by the LIST APPLICATIONS command, and along with other information not shown here, allows the application to be matched to its originating program or user.

The "Application status" and "Status change time" values have the same meanings as in the output of the LIST APPLICATIONS command, as described above.

The timestamps allow a picture of the waiting application to be obtained:

- The unit of work started at `14:30:36.889356`
- Very shortly thereafter, at `14:30:36.890986`, the waiting statement (`select * from org`) started executing. In this simple case, the waiting statement is the first one in its unit of work.
- The unit of work entered its current status (Lock-wait) at `14:30:36.907312`
- The snapshot was taken at `14:30:43.414574`
- The "Time application waited on locks" and "Total time UOW waited on locks" figures, both 6.5 secs., match the difference between the status change time and the snapshot time, because the current lock wait is the first one in the current UOW, and there were no earlier UOW's in the current connection. This would not be the case if there had been earlier lock waits in the unit of work (the UOW wait time would be higher than the wait time for the current lock wait), or if there were earlier units of work (with lock waits) executed in the same connection (the application wait time would be higher than the UOW wait time).

The set of twelve indented lines provide information related to the application's lock wait:

- The application (14) is waiting for a lock held by agent (application) 13, and the latter's ID is provided. You should look at the application and lock snapshots for application 13 and study them for clues as to the cause of the lock wait -- more about this later.
- The lock name is a hexadecimal value that provides a unique name for a lock. It's not essential that you know the components of a lock name (they're not documented), but in Appendix C there's a brief discussion of this.

- The lock attributes and release flags are defined in `sqlmon.h` (in `sqllib/include`), but are not usually valuable to study.
- The remaining values are quite self-explanatory.
- All of the values in the set, except one, describe the lock already held. The exception is “Lock mode requested”, which is the mode requested by the application this snapshot is for (14). In the above example, application 13 has an X lock on table ORG, and application 14 is unable to acquire the IS lock that it requires on the table to do its SELECT.

The above discussion of the application snapshot focused on applications in lock wait. Just as important are *snapshots for applications that others are waiting for*. The primary reason to look at these snapshots is to find whether the application is holding more locks than it should, or for a longer time. The most useful items in these snapshots are somewhat different from those for lock wait applications, and they are discussed following this subset of the application snapshot for handle 13 -- the application that handle 14 (above) was waiting for:

```
Application handle                         = 13
Application status                         = UOW Waiting
Status change time                         = 08-15-2003 14:30:34.954543
Application idle time                      = 9
Snapshot timestamp                         = 08-15-2003 14:30:43.414574
Locks held by application                  = 4
Lock waits since connect                   = 0
Time application waited on locks (ms)      = 0
Deadlocks detected                         = 0
Lock escalations                           = 0
Exclusive lock escalations                 = 0
Number of SQL requests since last commit   = 2

UOW start timestamp                        = 08-15-2003 14:30:34.651601
UOW stop timestamp                         =

Most recent operation                      = Execute Immediate
Most recent operation start timestamp      = 08-15-2003 14:30:34.652262
Most recent operation stop timestamp       = 08-15-2003 14:30:34.954535

Dynamic SQL statement text:
lock table org in exclusive mode
```

- **UOW start timestamp:** When subtracted from the time of the snapshot (“Snapshot timestamp”), this tells you how long the UOW has been in progress, and therefore how long it may have been holding locks. Obviously, the longer locks are held, the more chance there is of another application running into conflicts with them. In this example the UOW started at 14:30:34.651601 and the snapshot was taken at 14:30:43.414574, so the UOW had been running for almost nine seconds by the time of the snapshot.

- **Most recent operation stop timestamp:** When subtracted from the time of the snapshot, this tells you how long the UOW (unit of work) has been idle since finishing its previous SQL statement. If the gap is relatively large, a major cause of lock contention may be that too much time is spent inside the application, causing locks to be held too long. In this example the most recent statement ("lock table org in exclusive mode") stopped at 14:30:34.954535, almost nine seconds before the snapshot. This nine-second difference is also reflected in the value of "Application idle time". Nine seconds is much too long for a real application to sit idle while holding locks in a concurrency-sensitive situation. You can also compare the "Most recent operation stop timestamp" here (14:30:34.954535) to the "`Status change time`" in the waiting application's snapshot (handle 14's,14:30:36.907312) and see that the LOCK TABLE statement was executed about two seconds before the start of the lock wait.

- The problem with both of the above timestamps is that the timing of the snapshot may conceal an issue: the UOW may continue (and therefore hold locks) for a long time after the snapshot is taken, by executing many more SQL statements, or spending a lot of time in the application itself, or both. Taking one or more additional snapshots may be useful, but the way to guarantee that you have the start, stop, and elapsed time of each statement and UOW is to use the event monitor for statements, which I'll discussed later. Another possibility is to use CLI tracing on the client; this isn't quite as direct as event monitoring and is not discussed in this document.

- **Number of SQL requests since last commit:** This is an indication of how much work has been done in the UOW, which may be unreasonably large for a single UOW. In this example the number is only two, but LOCK TABLE IN EXCLUSIVE MODE is deadly for concurrency. They are not shown above, but an application snapshot also shows the number of rows updated, inserted, deleted, and read, which are also good indicators of the work done in the UOW. In particular, if isolation level CS is in use, the number of rows read will tell you how many row locks were obtained and later released prior to the snapshot (except if the rows were updated or deleted, in which case they'll still have X locks on them).

- **Locks held by application:** This is an obvious factor in evaluating how blameworthy an application is in creating lock conflicts. When *table* locks are held, however, there can be contention issues despite few locks, so it is usually necessary to look at the lock snapshot for the application causing the wait; here there are only four locks held, but one of them is from the LOCK TABLE statement. Other lock-related information, such as "Time application waited on locks" and "Lock escalations" will suggest whether locking was a problem for this application earlier, even though it might not be at the time of the snapshot.

- **The statement being executed** (identified by the "Dynamic SQL statement text" or the static SQL package and section ID): In general you shouldn't put too much focus on this statement. It's just as likely that a statement executed earlier in the UOW is the one holding the lock that the contention is on. However, knowing what's being run can help you understand the context of the problem, and in this example it really is the LOCK TABLE statement that's the problem.

The above discussions have covered the cases of a waiting application and a waited-on application, but it's not uncommon for the same application to have both of those roles. For example, application A can be waiting on application B, which is itself waiting for C; thus application B is both a waiter and a causer of a wait, so you would want to look at its application snapshot from both of those standpoints.

Another situation to look for is several applications all waiting for the same application, possibly even for the same lock. A quick way to check for this is to use `grep` or `findstr` against the snapshot file. Search for "ID of agent holding lock" and see if the same handle number appears multiple times. Such an application is obviously a good candidate to start your analysis with.

Lock snapshot

Like the application snapshot, the lock snapshot has a section for each connected application, and it's where you find detailed information on the locks held. There are two or three types of entries for each application, and the entries are in this sequence:

1. A header identifying the application and giving lock summary information, notably the number of locks held and the total lock wait time.
2. If the application has lock wait status, there will be a block describing the lock wait. This block is usually exactly the same as what was presented earlier in the application snapshot, except in rare cases where lock information changes after the time of the application snapshot and before the lock snapshot.
3. Details on each lock that was successfully acquired and is currently held. These details are virtually the same as what's provided for a lock wait, except that there is no contention-related information. The main additional piece of information is the Lock Count: this will usually be 1, but can be greater than 1 if different applications hold the same lock. (For example, two applications can each have an NS lock on the same row.) Also, 255 is a special case reflecting a lock held for a long time, such as for a LOCK TABLE statement.

It's very important to keep in mind that the isolation level in use for an application plays a critical role in what locks are held and will therefore show up in the lock snapshot. For example, with isolation level UR in use, there should be no read locks (NS or S) on user tables, but the same application running with isolation level RR or RS could have a large number of read locks on user tables. (Locks on catalog tables are taken when required by DB2 and are not directly affected by the application's isolation level.) With the default isolation level of CS in use, you'll typically see one read lock held per table at any given time, but be aware that there may have been a large number of read locks acquired and freed prior to the time of the snapshot.

If isolation level RS or RR is in use, or there are updates, inserts, or deletes performed in the transaction, there may be enough locks held that you can get a good idea of what the transaction has done. Start at the bottom of the Lock Snapshot and work your way up to the top, since locks are reported in reverse chronological order. Typically the first (bottom) few entries will be internal locks related to the application package being used. Above those will usually be an IS or IX lock on a table, followed by one or more NS or X locks on rows in that table, and then a repetition of this pattern for other tables. It is left as an exercise for the reader to see how the lock pattern for a SELECT statement with joins varies with the access plan type (nested loop join *vs*. merge scan or hash join). For an application in lock wait, the top (final) lock was the last one successfully acquired. It will often be an IS or IX lock, with the lock wait occurring on the subsequent attempt to lock a row in the same table.

Let's look briefly at internal locks, which show up in lock snapshots but don't have details on them in the DB2 manuals. These locks are issued by DB2 for a variety of reasons, but primarily on application packages and sections while they're being executed, to prevent them from being dropped by another connection. You may rarely see lock waits occurring on internal locks, and normally only if a DROP PACKAGE is attempted at a bad time. So you don't really need to worry about internal locks.

Dynamic SQL snapshot

A dynamic SQL snapshot shows each dynamic SQL statement that's currently in the DB2 package cache, and provides usage and performance information for all users collectively over the counting period. The main use of this snapshot for lock issues is in confirming that certain SQL statements are being affected by lock waits. In such a case, the statement's total execution time will be much higher than the CPU time expended for it, and unless a lot of I/O or network traffic is occurring for the statement, lock wait time is the usual explanation for the discrepancy. An example of this is shown in the case study in Appendix A.

(3) Snapshot table functions, routines and views

You can use administrative SQL functions, procedures and views that produce information related to locks. These are:

- snap_get_lock table function and snaplock view
- snap_get_lockwait table function and snaplockwait view
- locks_held view
- lockwaits view
- am_get_lock_chn_tb procedure
- am_get_lock_chns procedure
- am_get_lock_rpt procedure

Note that in V9 snapshot_lock has been deprecated and replaced by snap_get_lock table function and snaplock view. Similarly, `snapshot_lockwait` has been deprecated and replaced by snap_get_lockwait table function and snaplockwait view. Usage Details and output details are all described in *Administrative SQL Routines and views* manual.

`snap_get_lock` produces one row for each lock held, and `snap_get_lockwait` one row per lock wait condition. Each row contains the same data that's provided in multiple lines of "get snapshot for locks" output. Sample scripts with `snapshot_lockwait` usage are in Appendix B.

Important note: `snap_get_lockwait` will not return any rows unless the DBM configuration parameter DFT_MON_LOCK is set to ON.

Whether to use "get snapshot for locks" or the table functions (snap_get_lock table and snap_get_lockwait) or administrative views (snaplock and snaplockwait) is largely a matter of personal preference. The first option makes it easier to get the "big picture", while the second and third yield results that are easier to deal with programmatically. These table functions can be run against any database because database name needs to be specified when using either of the table functions. Views are objects in a database, therefore database name need not be specified, but you need to have a database connection to be able to access the view. Using these views produces the equivalent of "db2 get snapshot for locks on <database alias>."

Snaplock and snaplockwait views are very similar to locks_held and lockwatis views respectively. Application name(appl_name) and auhorization id (AUTHID) are only available in locks_held and lockwaits views. If these elements are of interest, then locks_held/lockwaits should be used.

In V9, three lock related stored procedures have also been built into the product to help with lock monitoring and investigation. Am_get_lock_rpt takes gives all locking related information for a given application in one place. Output is split into 3 parts: the first section has general application information, the second details all the locks that theapplication is holding, and the third section details all the locks it is waiting on. While this is useful it does not show if the given application is part of a complex lock chain.

Am_get_lock_chn_tb and am_get_lock_chns provide information on lock chain maps. Both are essentially the same, former provides output in a tabular format and latter provides info in plain text format. Consider a scenario where application A is waiting on application B, Application B is waiting on C and C is waiting on D. Prior to V9, one would have to take several snapshots and hand draw all this information to figure out the whole chain. In V9, you could call either of the procedures with the application of handle of one of applications in the chain and it would output the entire chain. An example of this is shown in "Resolving Lock waits" section in this article.

(4) Event monitoring

A DB2 event monitor can be used to obtain performance information on events as they occur on the server, such as statement or transaction completion, or deadlock resolution. For DB2 locking issues, event monitoring serves two essential purposes:

- It gives you transaction flow and timing information related to lock waits. While a snapshot can tell you that an application is in lock wait state, it usually will not provide enough information on what the application was doing prior to the lock wait, nor on what the other application(s) involved in the lock wait were doing. Creating an event monitor for statements is the ideal tool to use, because it will show the activity for each transaction, including when each SQL statement started and ended. Each end time will reflect any delay from lock waits.

  Note that for each dynamic SQL statement there will usually be multiple event monitor events with different operation types: for a SELECT, there will be Prepare, Open, and Close; for an INSERT, UPDATE, or DELETE, a Prepare and Execute. For the purposes of lock wait issues, it's the Close and Execute events that are most interesting, because their elapsed time values ("Exec Time:") will reflect any lock wait delays in executing the statement.

  For OLTP-like applications, the overhead of running event monitoring for statements is quite high, and the output voluminous, so be careful not to have the monitor running for too long. Even a few seconds can produce megabytes of output.

- It gives you details on deadlocks. Snapshots can provide counts of the numbers of deadlocks that occur, and, if you time them properly, can give you the application details for deadlock situations before they are identified by the deadlock detector and rolled back. However, the only way to guarantee that you obtain detailed information on each deadlock is to create and activate an event monitor "for deadlocks with details".

  The overhead of an event monitor for deadlocks is very small, so this type of monitor can be left running for long periods, and you can use FLUSH EVENT MONITOR to flush all of the deadlock information gathered to that point for examination (while leaving the monitor running).

You have two choices for obtaining event monitor output in readable form:

- The `db2evmon` command provides ASCII output. To prepare for `db2evmon`, create the event monitor with the "write to file" option. `db2evmon` formats the binary data from the monitor into a file with events presented in chronological order.
- The `db2eva` command provides a graphical display. To prepare for `db2eva`, create the event monitor with the "write to table" option. db2eva provides the same information as db2evmon, but with sorting, drill-down, and other capabilities.

The choice of which tool to use is as much a matter of personal preference as anything else. An advantage of `db2evmon` is that it's easier to see how one application's events coincide with another's, but analysis of `db2evmon` output can be difficult unless you are familiar with commands such as `grep` and `sort`. Examples of the use of `grep` and `sort` are in the case study in Appendix A.

(5) Administration notification log ("notify log")

It's used to store diagnostic information that's useful to customers, as compared with the information in `db2diag.log`, which is intended for IBM Service personnel.

On UNIX® platforms, the notify log is a text file called `<instance_name>.nfy` in the directory specified by the DIAGPATH parameter. On Windows®, DB2 administration notification messages can be seen through the Event Viewer (in the Application Log).

By setting the database manager configuration parameter *notifylevel* to 4, you can ensure that all available lock escalation information is recorded in the notify log. (Refer to Chapter 5 in Performance Guide for details on which information is available at lower values of notifylevel.

Here are two sample entries from the notify log for a lock escalation. Some less important lines have been omitted from this sample.

```
====================================================================

Event Type:    Information
Description:

2003-09-10-15.32.23.058000   Instance:DB2   Node:000
PID:1156(db2syscs.exe)   TID:2088   Appid:*LOCAL.DB2.011980193214
data management  sqldEscalateLocks Probe:1   Database:SAMPLE

ADM5501I  DB2 is performing lock escalation.  The total number of
locks
currently held is "47", and the target number of locks to hold is
"23".  The current statement being executed is "insert into org
select * from org".

====================================================================


Event Type:    Warning
Description:

2003-09-10-15.32.23.098000   Instance:DB2   Node:000
PID:1156(db2syscs.exe)   TID:2088   Appid:*LOCAL.DB2.011980193214
data management  sqldEscalateLocks Probe:3   Database:SAMPLE

ADM5502W  The escalation of "43" locks on table "WILKINS .ORG" to
lock intent "X" was successful.

====================================================================
```

(6) Health Center

The Health Center is primarily for production databases, but you may wish to use it during application development. It provides four indicators in the Application Concurrency category: lock escalation rate, lock list utilization, percentage of applications waiting on locks, and deadlock rate. With the Health Center you can set warning and alarm levels for the indicators, enable the indicators, and define an action to be taken when the thresholds are reached, such as taking a snapshot.

(7) Operating system facilities

While it doesn't provide direct information on lock issues, the use of operating system tools on the database server can sometimes provide useful indications. In particular, if the CPU utilization is unexpectedly low, the explanation can be that some applications are sitting idle due to lock waits. You can use `vmstat` on UNIX or Linux, or the Task Manager on Windows, to find CPU utilization.

Resolving the lock issues

Now that we've looked at what the issues are, and the tools available to get information about them, how do we go about resolving the issues? To start with, one thing that helps with all of these lock issues is this: reduce the amount of locking that takes place. Let's look at ways of doing this, and later I'll discuss techniques that are specific to each of the issues.

Reducing locking

There are numerous ways to reduce locking. Not all will be applicable to your particular situation, but there's almost always something you can do.

1. Use a weaker isolation level for packages or when running applications. From weakest to strongest, the isolation levels are: UR, CS, RS, RR. Remember that the choice of isolation level must be made in conjunction with the application design, so as not to introduce integrity exposures (if the level is too weak for the design). Here are the techniques to use to set the isolation level for different application types:
   - JDBC: Connection.setTransactionIsolation()
   - CLI: SQLSetConnectAttr
   - CLP: Use "change isolation to <level>" (before connecting to the database).
   - Static packages: Use the "isolation" clause in the BIND or PREP/PRECOMPILE commands.
   - SET CURRENT ISOLATION <ISOLATION LEVEL> can be used to set the isolation level at the session level. It can be set in CLP sessions or embedded in applications.
2. If it's not appropriate for you to change the isolation level for an entire application or connection, you may be able to do so for particular statements, by setting the isolation

level at the statement level using the WITH clause. These SQL statements support statement-level isolation: SELECT, SELECT INTO, Searched DELETE, INSERT, Searched UPDATE, and DECLARE CURSOR. For example: “SELECT * FROM org WITH UR”.

3. Using such means as creating indexes or adding predicates, cause access plan changes that will reduce the number of rows searched by queries. The fewer the rows searched, the fewer the locks that will be taken. It is beyond the scope of this document to fully describe the process and possibilities for improving access plans. Suffice it to say that the common methods of accessing a table, from most to least preferred with respect to locking, are:

    - Index scan with start key and stop key (ideally, the start and stop keys being the same)
    - Index scan with start or stop key
    - Full index scan (no start or stop key)
    - Table scan

Avoiding lock escalations

There are two approaches to take in avoiding escalations: (a) reduce the amount of locking; (b) increase the memory resources available for locking.

Approach (a) is preferable, because it produces CPU savings that may measurably improve performance, but it can take more effort to implement. Although any of the above techniques for reducing locking can help, there is another possibility: force DB2 to use table locking instead of row locking. In effect, the goal is to achieve the benefit of an escalation without actually going through one. However, table locking can solve an escalation problem but in doing so cause lock waits or deadlocks (for example, when a table has an S or X table lock on it, no other unit of work can update a row in it), so it should usually be used only in one of these situations:

- A batch environment where there is only one unit of work accessing the table(s) in question
- A table being accessed by multiple connections concurrently, but exclusively read-only.

The two ways to directly force table locking are:

- Use the LOCK TABLE statement in the same unit of work where a large number of row locks would normally be acquired. Using LOCK TABLE <table> IN SHARE MODE will allow other connections to have read-only access to the table. If you’re using CLI or CLP, remember that autocommit is on by default, so the table lock from LOCK TABLE is released immediately unless autocommit is turned off.

- Use ALTER TABLE <table> LOCKSIZE TABLE to cause future accesses of the table to acquire a table lock. This remains in effect until a subsequent ALTER TABLE is run to change locksize back to ROW.

Another way to avoid escalations is to commit more frequently. This does not reduce the total number of locks taken for a workload, but spreads them over more units of work, such that there may never be enough at one time to trigger an escalation.

Approach (b), increasing the memory resources available for locking, is done by raising the value of the database configuration parameters LOCKLIST (total memory for all locks in the database) or MAXLOCKS (percent of LOCKLIST available for a single transaction before triggering a lock escalation). Two problems with this approach are the additional memory that's tied up if LOCKLIST is increased, and the extra CPU taken to acquire and free more row locks, but in some cases escalations must be avoided because of their concurrency impact. In such a case, you can use information in the administration notification log to guide you in adjusting LOCKLIST or MAXLOCKS. There are quite detailed instructions supplied in the Performance Guide, under the heading "Correcting lock escalation problems". When tuning LOCKLIST, one can estimate the number of locks a given LOCKLIST would support. Here is a summary of different lock sizes:

For 32 bit platforms in v8, each lock requires:

- 80 bytes to hold a lock on object that has no locks on it.
- 40 bytes to record a lock on object that has an existing lock.

For 32 bit platforms in v9, each lock requires:

- 96 bytes to hold a lock on object that has no locks on it.
- 48 bytes to record a lock on object that has an existing lock.

For 64 bits platforms in v8 and v9, each lock requires:

- 128 bytes to hold a lock on object that has no locks on it.
- 64 bytes to record a lock on object that has an existing lock.

New in V9, by default LOCKLIST and MAXLOCK are set to AUTOMATIC. With this setting, Self Tuninng Memory Manager(STMM) adjusts different memory heaps automatically by taking memory from under utilized heaps and giving to maxed out heaps. STMM will try to prevent lock escalations by automatically tunning LOCKLIST and MAXLOCK where possible. If there are no under utilized heaps, then STMM may not be able to give more memory to LOCKLIST but it may still be able to tune MAXLOCK to prevent lock escalations. A detailed discussion of STMM can be found in *Perfromance Guide*.

Resolving lock waits

Let’s assume that through monitoring or response time observations, someone has conjectured that lock waits are a problem. Your first steps should be to use LIST APPLICATIONS SHOW DETAIL and look at database snapshots with counters accumulated over a relatively long time, to see if lock waits are frequent or significant enough to worry about. Be careful you don’t miss potential future locking issues if only a small number of applications are currently running.

Once you’ve determined that lock waits are a problem, you should take enough snapshots to capture several lock waits in progress. Tools such as those discussed in Appendix B can be used. Look at the application and lock snapshots to determine how many different issues there are, and prioritize them using factors such as the number of applications involved and the wait durations. One helpful technique is to look at the applications in lock-wait and draw a picture relating the application handles, such as in this example, where the numbers are handles and an arrow means “is waiting for”:

```
13 --> 10 --> 5
11 --> 16 --> 10 --> 5
24 --> 30
35 --> 30
37 --> 30
```

In this case, there are apparently two separate issues, one involving chains of applications waiting for handles 5 and 10, and another with waits for handle 30. An automated way to produce information similar to the above “graph” is to use the am_get_lock_chns procedure. Let us take the a simple example where application handle 15 is waiting on application handle 11. We want to see there are more members in this lock chain:

```
C:\>db2 call sysproc.am_get_lock_chns(15,?)

  Value of output parameters
  --------------------------
  Parameter Name  : LOCK_CHAINS
  Parameter Value : >db2bp.exe (Agent ID: 15) (Auth ID: YASIR   )

<db2bp.exe (Agent ID: 15) (Auth ID: YASIR   )
   <db2bp.exe (Agent ID: 11) (Auth ID: YASIR   )
      <db2bp.exe (Agent ID: 7) (Auth ID: YASIR   )

  Return Status = 0
```

This confirms that there are more members in the lock chain:

15 --> 11 --> 7

Appl handle 15 is waiting on 11, which in turn is waiting on 7.

If you’re familiar with the application, you may be able to figure out the problem from the above steps. Usually, however, it will take a bit more work to determine the sequence of events leading up to each lock wait. You should create and activate an event monitor for statement and deadlock events, and while the event monitor is active, reproduce the lock wait situation(s) and take several snapshots that capture lock waits in progress. Then, for each lock wait in a snapshot, use the snapshot time, the application handle, and the statement text to find the waiting statement in the event monitor output (assuming, that is, that the lock wait was resolved; otherwise, the statement will still be waiting and there will be no Close or Execute event for it, in which case you should find the previous event (usually Open or Prepare) for the statement). Work backward from that statement, looking at the previous events for the waiting application handle and the handle of the application it was waiting for, going only as far as the previous Commit or Rollback for each (since any locks taken before then will have been released). This procedure will allow you to determine all of the statements and tables involved in the lock wait situation, and should let you figure out why the lock wait occurred. Remember to use the *Performance Guide* for information on the locks taken by different statements, and the compatibility between them.

Even if you know the cause of the lock wait, you have limited means to fix it unless you have control over the application source. The main exception to this is the LOCKTIMEOUT parameter discussed earlier. Assuming that you can change the application, here are some design and coding tips (in addition to those listed earlier under "Reducing Locking"):

- Avoid lock escalations, using the techniques given above.
- Commit more frequently, but not so frequently that transactions no longer match the definition of what a business transaction consists of.
- Specify the FOR UPDATE clause in the SELECT statement whenever appropriate. FOR UPDATE avoids problems when two different applications select the same row and then try to update it. Without FOR UPDATE, depending on the isolation level, either
  - Both applications can read and lock the row, but neither can update it because of the lock the other holds on the row (X conflicts with NS), or
  - One application can read the row and extract values from it, but before it updates the row, another application might update the row and commit, causing the first application's subsequent update to store values that conflict with the second application's values.

  With FOR UPDATE, SELECT takes a U lock on the row, and the second one to execute will wait for the lock taken by the first to be released.
- Different connections should process different sets of rows as much as possible. Avoid hotspots such as rows that are updated in every transaction to store a value like "last ID used". When such hotspots are needed, move the update to the end of the transaction and commit as soon as possible afterwards.
- Release read locks established under RS or RR isolation level by using the WITH RELEASE option of the CLOSE <cursor name> statement, instead of waiting to have them released by the eventual Commit or Rollback.

Resolving deadlocks

Most deadlock issues can be resolved by avoiding lock waits using the techniques I've already discussed. If you don't have lock waits you can never have deadlocks. In addition, consider these ideas:

- Each application connection should process its own set of rows to avoid lock waits, but if that isn't always possible, the application should be designed such that every transaction type accesses the various tables in the same order.
- A lock timeout is not much better than a deadlock, because both cause a transaction to be rolled back, but if you must minimize the number of deadlocks, you can do it by ensuring that a lock timeout will usually occur before a potential related deadlock can be detected. To do this, set the value of LOCKTIMEOUT to be much lower than the value of DLCHKTIME (both of these being database configuration parameters, and keep in mind that the units of LOCKTIMEOUT are seconds, and DLCHKTIME milliseconds). Unless LOCKTIMEOUT is set to 0, however, the deadlock detector could wake up just after the deadlock situation began, and detect it before the lock timeout occurs.
- Avoid concurrent DDL operations if possible. For example, DROP TABLE statements can result in a large number of catalog updates as rows may need to be deleted for the table's indexes, primary keys, check constraints, and so on, as well as the table itself. If other DDL operations are dropping or creating objects, there can be lock conflicts and even occasional deadlocks.
- An application, particularly a multithreaded one, can have a deadlock involving a DB2 lock wait and a wait for a non-DB2 resource such as a semaphore. For example, connection A can be waiting for a lock held by connection B, and B can wait for a semaphore held by A. DB2's deadlock detector is unable to know about and resolve such a situation, so the application design must guard against this.

Locking notes specific to partitioned databases

From the standpoint of locking, a partitioned database can be viewed as a collection of near-independent databases, in each of which locks are obtained as though the database were not partitioned. The special issue to be addressed, however, is that on partition 1, application A could wait for a lock held by application B, and on partition 2, B could wait for a lock held by A, and this deadlock situation could never be detected by a deadlock detector running independently on each partition. To resolve this issue, there is a *global deadlock detector*. In a partitioned database, each partition sends *lock graphs* to the database partition that contains the system catalog views. Global deadlock detection takes place on this partition, but a deadlock is not flagged until after the global deadlock detector wakes up for the second time after the deadlock occurred.

As with all database configuration parameters, make sure that the lock-related parameters have the same value on every partition. You can use the `db2_all` command to send "`db2 update db cfg`" commands to every partition.

Locking Registry variables

Here is a list of registry variables that would improve concurrency if the changed behavior still meets your business requirements. To enable following registry variables you need to issue:

```
db2set <registry variable>=on
```

and then recycle the instance(db2stop/db2start).

1) DB2_SKIPINSERTED

The default value is off. When a table scan is taking place with CS or RS isolation levels, and cursor hits a row that is being inserted but not committed yet, the scanning application would go into lock-wait and wait for the row to commit or rollback. There is a large class of applications that do not need such behavior and lock wait for uncommitted inserts is not useful. For such applications you can enable this variable. When this variable is enabled, the cursor would go over the inserted row, without waiting, and not present the skipped rows in the result set.

2) DB2_EVALUNCOMITTED

If it is acceptable in your application to evaluate data in uncommitted rows then you may be able to improve concurrency using this registry variable. With a CS or RS scan, each row is locked and evaluated to see if it qualifies. If there is an existing lock on the given row, scanning application goes into lock wait. With this option enabled, application can evaluate uncommitted data without having to wait for a lock. Consider an ORDERID column where one of the rows has been updated from 20 to 5, but not committed. Along comes a scan: “select ... where orderid > 10”. With default behavior, this scan will wait for the commit to happen and then evaluate the committed value. With this registry variable enabled, scan would skip over this row because it does not satisfy the predicate. However, if the ORDERID had been updated to 15 and not committed, scan would still wait on this row because the value satisfied the predicated and it needs to lock it to include in result set.

3) DB2_SKIPDELETED

With CS or RS scan cursor waits to get a lock on any uncommitted deleted row. With this variable, you can make the cursor skip over the uncommitted deletes. These uncommitted deletes would not show up in your result set and later if the deletes were rolled back and another scan was done, rows that were missing in first pass would show up now. Again, you have to be cautious that such behavior is acceptable in your environment.

Conclusion

We hope that this document has given you a good idea of how to diagnose and resolve most DB2 locking issues. Watch for even more DB2 concurrency improvements in the near future.

References

- DB2 manuals can be downloaded or viewed at:
  http://www.ibm.com/software/data/db2/udb/support/manualsv9.html

- *DB2 V9 Performance Guide*; Chapter 5, Concurrency issues. At the beginning of this chapter there's an extensive description of concurrency and locking.
- *DB2 V9 Command Reference*. It provides details on all of the DB2 commands, such as GET SNAPSHOT and get evmon.
- *DB2 V9 Administrative SQL Routines and views.* This manual documents the various administrative snapshot functions and views such as SNAP_GET_LOCK. *V9 SQL Reference Vol 2*. "Statements" section. This covers the ALTER TABLE, CREATE EVENT MONITOR, FLUSH EVENT MONITOR, LOCK TABLE, and SET EVENT MONITOR STATE statements.
- DB2 V9 *System Monitor Guide and Reference.* It describes the various monitor tools and the information they provide. Part 4 covers the Health Center and some approaches to dealing with the issues monitored by the concurrency indicators.
- DB2 Technical Support -- DB2 Problem Determination Tutorial Series. These Web tutorials provide a good introduction to dealing with locking issues.
  - Tutorial 5: Database Engine Problem Determination; Section 4: Troubleshooting locking problems.
  - Tutorial 6: Performance Problem Determination, Section 5: Locking - lock waits, timeouts, escalations and deadlocks.

  These tutorials are available at:
  http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/c0011982.htm

Appendix A -- Case Study

This appendix discusses an actual situation encountered during the testing of an application developed by a DB2 Business Partner. The analysis began with an examination of why response times were poor, then evolved into a study of where lock waits were occurring and why, and finally a resolution of the problem. The purpose here is not to focus on the exact problem, but to show usage of the various tools and the thought process being used.

Lock wait issue: Step 1 = Why were response times so poor?

For discussion purposes we view this issue as having three steps, but the three steps were repeated several times as the biggest problems were resolved and others then came to the surface.

The Lock Wait issue was discovered and addressed as part of the overall task of improving the response times. The first indication that lock waits were an issue was in the output of `vmstat`, which showed very low CPU consumption and relatively little I/O wait time (more on that later). Confirmation of the lock waits quickly followed in the output of the snapshot monitor (“`get snapshot for all on <database>`”). Over a counting period of about 8.4 minutes, the following were reported in the Database Snapshot portion of the output:

```
Locks held currently                       = 934
Lock waits                                 = 438
Time database waited on locks (ms)         = 2770173
Lock list memory in use (Bytes)            = 82692
Agents currently waiting on locks          = 17
```

There were 37 applications connected to the database, and at the time of the snapshot, 17 of them were in a lock wait state. Furthermore, there was an average of ((2770 secs./ 60 secs./min.) / 37 applns.) = 1.24 minutes of lock wait time per application over the 8.4 minutes; if the time each application was inactive were omitted, the lock wait time during activity must have been very high, perhaps 50%.

Further confirmation of the lock wait problem came from two other sources:

(a) The following shows the Dynamic SQL Snapshot information for one query that uses the VR_VIEW view. The counting period was once again 8.4 minutes. Notice that the total execution (elapsed) time was 2511 secs. for 1150 executions, an average of about 2 secs. per execution, but the total CPU time expended was less than 1.5 secs. for all 1150 executions. So what was DB2 doing in all that time? Waiting for locks would be the likeliest explanation, even if we hadn't looked at the lock information above.

```
Number of executions              = 1150
Number of compilations            = 0
Worst preparation time (ms)       = 81
Best preparation time (ms)        = 81
Internal rows deleted             = 0
Internal rows inserted            = 0
Rows read                         = 101121
Internal rows updated             = 0
Rows written                      = 0
```

```
 Statement sorts                      = 3405
 Total execution time (sec.ms)        = 2511.616892
 Total user cpu time (sec.ms)         = 1.340000
 Total system cpu time (sec.ms)       = 0.080000
 Statement text                       = SELECT * FROM VR_VIEW
                                        WHERE DATAVALUE = ? AND
P_ID = ?
                                        ORDER BY A_ID
```

(b) The following is the event monitor (db2evmon) information provided for one execution of the same statement for which we just looked at summary information in the Dynamic SQL Snapshot. Note that there were actually two previous events for the same execution of this statement, a Prepare and an Open, but their elapsed times are negligible and they are not interesting for the lock wait discussion.

The "Exec Time:" line shows that the statement took 6.28 secs. to execute, but only took 0.01 secs. of CPU.   Once again we wonder why it took so long, and lock wait is really the only explanation.

```
4083) Statement Event ...
  Appl Handle: 10
  Appl Id: 0A0A1071.DDA4.030429211640
  Appl Seq number: 0001

  Record is the result of a flush: FALSE
  -------------------------------------------
  Type     : Dynamic
  Operation: Close
  Section  : 4
  Creator  : NULLID
  Package  : SQLLF000
  Cursor   : SQLCUR4
  Cursor was blocking: FALSE
  Text     : SELECT * FROM VR_VIEW
             WHERE DATAVALUE = ? AND P_ID = ?
             ORDER BY A_ID
  -------------------------------------------
  Start Time: 04-29-2003 16:12:20.951335
  Stop Time:  04-29-2003 16:12:27.238560
  Exec Time:  6.287225 seconds
  Number of Agents created: 1
  User CPU:   0.010000 seconds
  System CPU: 0.000000 seconds
  Fetch Count: 25
  Sorts: 1
  Total sort time: 0
  Sort overflows: 0
  Rows read: 200
  Rows written: 0
  Internal rows deleted: 0
  Internal rows updated: 0
  Internal rows inserted: 0
  SQLCA:
   sqlcode: 0
   sqlstate: 00000
```

To find the above problem statements I used commands such as the following, which print the longest 25 total elapsed times (from a Dynamic SQL Snapshot) and the longest 10 elapsed times for individual executions (from `db2evmon` output). Once you have the times it's easy to search for them to find the block of information for the statement.

```
grep "Total execution time" snapshot_file.txt | sort -k 6,6rn | head -n 25
grep "Exec Time:" event_monitor_file.txt | sort -k 3,3rn | head -n 10
```

(Another source of confirmation would have been LIST APPLICATIONS SHOW DETAIL.)

Lock wait issue: Step 2 = Where were lock waits occurring, and why?

Once we knew lock waits were occurring, we needed to find details about them, and such information is available through Application Snapshot and Lock Snapshot output. The problem with snapshots, however, is that they only provide detailed lock wait information if a snapshot is taken while a lock wait is actually in progress, so they need to be timed appropriately. Two approaches we used were:

(a) A brute force approach of taking snapshots very frequently, say every five seconds, and then looking inside the files taken when lock waits were in progress. One way is to take the snapshots and then run this command:

```
      grep "Agents currently waiting on locks" snapshot_file_*.txt
```

which might show lines such as these:

```
      snapshot_file_1.txt:Agents currently waiting on locks        = 0
      snapshot_file_2.txt:Agents currently waiting on locks        = 17
      snapshot_file_3.txt:Agents currently waiting on locks        = 2
```

We would then focus our efforts on `snapshot_file_2.txt`.

(b) A more systematic approach of repeatedly using "`db2 list applications show detail | grep -i lock-wait | wc -l`", which prints the number of applications with lock-wait status. We could therefore take a snapshot when the count was > 0. Out of this approach evolved a script to automate the process, as shown below in Appendix B.

Once one of these approaches produced a file with lock waits in progress, there were a few things to be done:

(i) Find a waiting application. To do this, look for "Lock-wait", which can show up at the start of an Application Snapshot, as follows:

```
                  Application Snapshot

      Application handle                         = 21
      Application status                         = Lock-wait
```

If we continue down to the bottom of that Application Snapshot, we see lines such as the following:

```
      Dynamic SQL statement text:
```

```
SELECT * FROM VR_VIEW WHERE DATAVALUE = ? AND P_ID = ? ORDER BY
A_ID
    Agent process/thread ID                           = 933992
  Agent process/thread ID                    = 933992

  ID of agent holding lock                   = 265
  Application ID holding lock                =
0A0A1071.AEBC.030428180321
  Lock object type                           = Row
  Lock mode                                  = Exclusive Lock (X)
  Lock mode requested                        = Next Key Share (NS)
  Name of tablespace holding lock            = USERSPACE1
  Schema of table holding lock               = SOURCE
  Name of table holding lock                 = OBJECTS
  Lock wait start timestamp                  = 04-28-2003
13:46:44.571922
  Lock is a result of escalation             = NO
```

There's a lot of useful information here. First of all, there’s the text of the statement that was being executed when the lock wait occurred. It turns out that this is the same statement we saw above. Also, we see that this statement was trying to obtain an NS (next key share) lock on a row in the OBJECTS table, and that lock could not be obtained because the application with handle 265 was holding an X (exclusive) lock on the same row. (Note that details on DB2 locking and conflicts between different lock modes can be found in the Administration Guide.)

During the early stages of our investigation into the lock waits, we found that when lock waits occurred, almost all of them had the same characteristics: an NS lock requested on an OBJECTS row, conflicting with an X lock held by one transaction that was causing virtually all of the lock waits. We now knew at a high level why the waits were occurring, but not the exact reason(s) behind them. Our next steps were to get more information on the bad transaction that "everyone" was waiting for, and what the waiting transactions were doing.

There can be many applications in a given lock chain. In this case, there were several applications waiting on one application. Once you have application handle of one of the application in a lock chain, rest of the elements in a lock chain can easily be identified by using the sysproc.am_get_lock_chns. Usage of this procedure is discussed in detail back in “Resolving Lock waits” section.

The bad transaction

The starting point for finding out about the "bad" transaction that caused a lot of waits was to use the handle number from the lock wait information above (265) and find the Application Snapshot for that handle. Here are some lines of particular interest extracted from that snapshot (not in sequence):

```
UOW start timestamp                          = 04-28-2003
13:46:27.646738
Snapshot timestamp                           = 04-28-2003
13:46:47.138356
Application status                           = UOW Waiting
Locks held by application                    = 694
Dynamic SQL statement text:
UPDATE OFFERS SET STATUS=?, [ more columns omitted ] ,
TDATIMEUPDATED = (SELECT DBDATE FROM DATABASEDATE)  WHERE
OFFER_ID=?
```

We see that the current transaction started at 1:46:27 pm, and that the snapshot was taken 20 secs. later, at 1:46.47. The application has "UOW Waiting" status, which means that DB2 is waiting for more work to be sent to it by this application. The previous statement it ran was the update of the OFFERS table, which is not particularly interesting, because the lock wait is on the OBJECTS table, but it helps identify what the transaction was doing. The worst part is that the application is holding 694 locks and since the transaction has been running for 20 secs., those locks have been held for an average of about 10 secs. already, and we don't know when the transaction will eventually commit and release the locks.

We can find out more about this transaction by looking at its Lock Snapshot. In the case of this bad transaction, we saw a sequence of nine tables being locked, some of them with hundreds of X locks on rows in them:

(four tables with relatively few rows locked in each)
OBJECTS (X lock on 1 row)
OFFERS (a bunch of X locks on rows)
(three tables, each with hundreds of X locks on rows by the time of the snapshot)

The developer was able to identify the transaction from the sequence of tables involved in the locks. The identification could also have been done by looking at the sequence of SQL statements in event monitor output for the transaction.

The largest number of locks recorded as held by a single application at any time was an enormous 16563. The developer identified this as a second transaction. It had a different pattern of lock acquisition, including some tables with over 1000 X locks:

(two tables with relatively few rows locked in each)
OBJECTS (X lock on 1 row)
(two tables, each with thousands of X locks on rows)
(two tables with a few rows locked in each)
(one table with hundreds of X locks on rows by the time of the snapshot)

Each of these transactions features an update of a row in the OBJECTS table relatively early in the transaction, which means that a lot of time is spent holding the OBJECTS row lock before the subsequent commit eventually releases it. I'll discuss the solution to this in Step 3 below.

OK, we've now seen that OBJECTS rows are being locked in these bad transactions, and held for long durations. But why are other transactions being held up by those locks? Shouldn't each transaction be dealing with an independent OBJECTS row and thus avoid lock conflicts? The answer to this question was obtained by first identifying the waiting SELECT statements, which as stated above, were initially almost always:

```
SELECT * FROM VR_VIEW WHERE DATAVALUE = ? AND P_ID = ? ORDER BY A_ID
```

This query expands to a 4-way join of the INFO, OBJECTS, PROPERTIES, and PRINCIPALS tables. To find out what role the OBJECTS table played in the query, we did an Explain of the query, which revealed that a table scan of the OBJECTS table was being performed. Since every row of the OBJECTS table needed to be accessed in those table scans, it was inevitable that if any row had an X lock on it, the SELECT would wait until that lock was released, which, as we've seen, could be many seconds later.

Lock wait issue: Step 3 = Fixing the problem

We've now seen that there were two aspects to the main locking problem: (i) long-running transactions holding X locks on rows in OBJECTS, and (ii) queries doing table scans that blocked on those X locks. Each of these had a different solution and we implemented both of them. Further study may have revealed that only one of them was actually necessary to achieve acceptable performance, but it was clearly desirable to do both.

For the X locks on OBJECTS, the developer was able to change the code so that the update of OBJECTS was done at the very end of each transaction. This meant that the OBJECTS X locks were held for a very short time and thus conflicts with SELECT statements were minimized.

Regarding the table scan on OBJECTS, often a table scan is chosen because no appropriate index exists, but that wasn't the case for the OBJECTS table. There were a few different circumstances involved:

(a) The PROPERTIES table was being accessed through a non-optimal index. We defined a new index on it and this index was chosen.
(b) The INFO table had a table scan, and we altered it to be VOLATILE (to make the optimizer aware that the table is likely larger than when the previous Runstats was done), and index access started to be used.
(c) Application performance test runs were typically preceded by a restore of the test database to get back to a consistent starting point. We found that the statistics for some tables indicated that the tables were empty (CARD=0 in SYSCAT.TABLES), which makes table scans very attractive to the optimizer (and which becomes increasing problematic as the tables grow over time). By the end of the analysis we had a established a new backup which incorporated all of our changes and had statistics updated to reflect non-empty tables.

Through the combination of (a)-(c) we made the access to OBJECTS change to a primary key (A_ID) lookup. Because only a single row now needed to be accessed in OBJECTS, the chances of having a lock conflict were almost eliminated.

Other lock wait issues

The above discussion focused on the OBJECTS table and the lock waits encountered on it by the "SELECT * FROM VR_VIEW", because that was by far the major lock issue. However, we did encounter other lock waits, and they were investigated using the same approach as above. Most of them were dealt with through index changes. Note that not all of the problematic access plans involved table scans. Some had index scans but either the entire index was accessed or a large enough subset of it that conflicts with X locks were likely; in some cases all that can be done is to rewrite the query, because they may not be selective enough.

Appendix B -- Sample Scripts

Here are some sample UNIX scripts for various lock-related tasks. If you aren't familiar with UNIX scripts, a Windows near-equivalent is shown for the third to give you an idea of how they compare.

This script can be used to loop and take a snapshot whenever there's an application with lock-wait status on a specific database.

```
#!/bin/ksh

# loop (until Ctrl-c) and take a snapshot if there are applications in lock
wait
# Arguments: (1) database name
#            (2) number of secs. between checks of application status
#            (3) number of secs. to wait before taking snapshot (should usually
be 0)
# Snapshot files are named like this:  snap_0618-194724.out  (MMDD-HHMMSS)

while (true) do
   WAITERS=`db2 list applications for db $1 show detail | grep "Lock-wait" | wc
-l`
   if [ $WAITERS -ge 1 ]
     then
       SNAPFILE=snap_`date +"%m%d-%H%M%S"`.out
       db2 +o update monitor switches using lock on statement on uow on
       sleep $3
       db2 get snapshot for locks        on $1 >  $SNAPFILE
       db2 get snapshot for applications on $1 >> $SNAPFILE
       db2 +o update monitor switches using lock off statement off uow off
       echo Number of applications in lock-wait: $WAITERS   -- see $SNAPFILE
     else
       echo Number of applications in lock-wait: $WAITERS
   fi
   sleep $2
done
```

This UNIX shell script shows how applications causing lock waits can be identified by the snap_get_lockwait function and forced. This automatic forcing is not necessarily such a good thing to do, but the script gives a simple example of how the snap_get_lockwait function can be used.

```
#!/bin/ksh

# force applications holding lock(s) that lock-wait applications are waiting
for
# Argument: database name

# DBM lock switch must be on for snap_get_lockwait to return rows; SYSADM
authority is
# required to set it here!
db2 update dbm cfg using dft_mon_lock on
```

```
db2 connect to $1
APPLIST=`db2 -x "select agent_id_holding_lk from table(snap_get_lockwait( '$1',
-1)) \          as slw"`

for APP in $APPLIST
  do
    db2 -v "force applications ($APP)"
done
db2 terminate
```

This UNIX shell script can be used to loop (until Ctrl-C) and show, for each lock conflict, the application handles involved, the table name, and the lock modes requested and held. The purpose is to get a quick idea of the pattern of lock conflicts over time through output that's more readable than the entire set of snap_get_lockwait output columns.

The script could be enhanced to only produce output when lock waits are detected -- see the first script above.

```
#!/bin/ksh

# loop forever and show subset of information from snap_get_lockwait
# Arguments: (1) database name
#            (2) number of secs. to sleep between displays

# DBM lock switch must be on for snap_get_lockwait to return rows
db2 update dbm cfg using dft_mon_lock on

db2 connect to $1
while (true) do
db2 "select agent_id as WAITING_FOR_LOCK, agent_id_holding_lk as HOLDING_LOCK,
table_name, cast(lock_mode_requested as smallint) as WANTED, cast(lock_mode as
smallint) as HELD  from table(snap_get_lockwait( '$1', -1)) as slw"

  sleep $2
done
db2 terminate
```

Sample output (the heading and zero or more rows will be displayed each time through the loop):

```
WAITING_FOR_LOCK     HOLDING_LOCK         TABLE_NAME                           WANTED HELD
-------------------- -------------------- ------------------------------------ ------ ----
--
                  75                   72 ORG                                       1
5
                  64                   72 ORG                                       1
5

  2 record(s) selected.
```

Here's a Windows command file version of the above script. The main differences are that the “$1” variable has been replaced by “%1%” in the Connect and Select statements, and, since sleep is not a Windows command, it has been replaced by pause (requiring the user to press Enter to start each new pass through the loop).

```
@echo off
rem loop forever and show subset of information from snap_get_lockwait
rem Argument: (1) database name

rem DBM lock switch must be on for snap_get_lockwait to return rows
@echo on
db2 update dbm cfg using dft_mon_lock on

db2 connect to %1%
@echo off
:startloop
  db2 "select agent_id as WAITING_FOR_LOCK, agent_id_holding_lk as
HOLDING_LOCK, table_name, cast(lock_mode_requested as smallint) as WANTED,
cast(lock_mode as smallint) as HELD  from table(snap_get_lockwait( '%1%', -1))
as slw"
  pause
goto startloop
db2 terminate
```

Appendix C -- Sample Lock Snapshot Output

Here's a sample Lock Snapshot which shows the pattern of locks when every row of a table is locked as part of a table scan done under isolation level RS. Notes:

- As usual, the locks are presented in order of newest to oldest. There can be variations to this if locks are escalated or upgraded.
- The first two locks taken were internal package locks related to the statement being executed.
- The next lock, an IS lock on the table, was obtained before accessing rows in the table.
- The remaining locks are all NS locks on the rows of the table. The first three elements in the hexadecimal lock name for a row lock are the tablespace ID (2, for userspace1), the table ID (2, the value of syscat.tables.tableid for wilkins.org), and an identifier for the row (RID), which maps to the lock object name. In the example below, the highest lock object name, 11, is mapped from 0x000B00. As another example, 0x000001 in the lock name would be reported as a lock object name of 256. Having said this, the lock name is not really essential to understand: the important components are reported separately. It's useful to know that if consecutive rows in a table are locked, they will usually have consecutive lock object names.
- The locks and lock snapshot were obtained through these steps in DB2 CLP:
```
db2 connect reset
db2 change isolation to rs
db2 connect to sample
db2 +c select * from org
db2 get snapshot for locks on sample
```

```
            Database Lock Snapshot

Database name                             = SAMPLE
Database path                             = D:\DB2\NODE0000\SQL00002\
Input database alias                      = SAMPLE
Locks held                                = 11
Applications currently connected          = 1
Agents currently waiting on locks         = 0
Snapshot timestamp                        = 08-21-2003 12:19:06.468584

Application handle                        = 44
Application ID                            = *LOCAL.DB2.00B7C1161826
Sequence number                           = 0001
Application name                          = db2bp.exe
CONNECT Authorization ID                  = WILKINS
Application status                        = UOW Waiting
Status change time                        = Not Collected
Application code page                     = 1252
Locks held                                = 11
Total wait time (ms)                      = 0

List Of Locks
 Lock Name                   = 0x020002000B0000000000000052
 Lock Attributes             = 0x00000000
 Release Flags               = 0x00000001
 Lock Count                  = 1
 Hold Count                  = 0
 Lock Object Name            = 11
 Object Type                 = Row
 Tablespace Name             = USERSPACE1
 Table Schema                = WILKINS
 Table Name                  = ORG
 Mode                        = NS

 Lock Name                   = 0x020002000A0000000000000052
 Lock Attributes             = 0x00000000
 Release Flags               = 0x00000001
 Lock Count                  = 1
 Hold Count                  = 0
 Lock Object Name            = 10
 Object Type                 = Row
 Tablespace Name             = USERSPACE1
 Table Schema                = WILKINS
 Table Name                  = ORG
 Mode                        = NS

*****  lines omitted here for ORG lock object names 6 to 9  *****

 Lock Name                   = 0x02000200050000000000000052
 Lock Attributes             = 0x00000000
 Release Flags               = 0x00000001
 Lock Count                  = 1
 Hold Count                  = 0
 Lock Object Name            = 5
 Object Type                 = Row
 Tablespace Name             = USERSPACE1
 Table Schema                = WILKINS
 Table Name                  = ORG
 Mode                        = NS

 Lock Name                   = 0x02000200040000000000000052
 Lock Attributes             = 0x00000000
 Release Flags               = 0x00000001
 Lock Count                  = 1
 Hold Count                  = 0
 Lock Object Name            = 4
 Object Type                 = Row
 Tablespace Name             = USERSPACE1
 Table Schema                = WILKINS
 Table Name                  = ORG
 Mode                        = NS

 Lock Name                   = 0x02000200000000000000000054
 Lock Attributes             = 0x00000000
 Release Flags               = 0x40000001
 Lock Count                  = 1
 Hold Count                  = 0
 Lock Object Name            = 2
 Object Type                 = Table
 Tablespace Name             = USERSPACE1
 Table Schema                = WILKINS
 Table Name                  = ORG
 Mode                        = IS

 Lock Name                   = 0x41414141414A485253334E4441
 Lock Attributes             = 0x00000000
 Release Flags               = 0x40000000
 Lock Count                  = 1
 Hold Count                  = 0
 Lock Object Name            = 0
```

```
Object Type                 = Internal P Lock
Mode                        = S

Lock Name                   = 0x434F4E544F4B4E3153544E4441
Lock Attributes             = 0x00000000
Release Flags               = 0x40000000
Lock Count                  = 1
Hold Count                  = 0
Lock Object Name            = 0
Object Type                 = Internal P Lock
Mode                        = S
```