

Improve Retrospectives

Measure your team's performance against
focus your efforts.

[Home](#) [Articles](#) [Tools](#) [Newsletters](#) [Subscribe](#) [Magazine](#) [Links](#) [Search](#) [Contact](#)

Ad ▾

Database Locking: What it is, Why it Matters and What to do About it

Justin Callison, Principal Consultant, Peak Performance Technologies Inc

Introduction

"Know your enemy and know yourself and you can fight a hundred battles without disaster."
Sun Tzu (The Art of War)

Database locking is a varied, evolving, complicated, and technical topic. As testers, we often think that it belongs in the realm of the developer and the DBA (i.e. not my problem). But to both functional and performance testers, it is the enemy and has led to many disasters (as the presenter can personally attest).

However, there is hope. This paper will shed light on the nature of database locking and how it varies between different platforms. It will also discuss the types of application issues that can arise related as a result. We will then look at ways to ferret out these issues and to resolve them before they sneak out the door with your finished product. Armed with this understanding of the enemy and how it relates to your application, you'll be much better able to avoid disaster.

1. Scope

The breadth and depth of this topic necessitates that scope be constrained. The scope of this paper has been chosen with the following considerations in mind:

- The audience is QA professionals
- The audience does not have significant database experience
- Core concepts can be generalized once understood

Specifically, the scope will be constrained to:

- "Transactional locking" (not all types of locking)
- Oracle and SQL Server

2. Why do Databases Lock?

So why does locking occur in a database? As in other systems, database locks serve to protect shared resources or objects. These protected resources could be:

- Tables
- Data Rows
- Data blocks
- Cached Items
- Connections
- Entire Systems

There are also many types of locks that can occur such as shared locks, exclusive locks, transaction locks, DML locks, and backup-recovery locks. However, this paper will focus on one specific type of locking that I will call "transactional locking".

2.1 ACID Properties of Transactions

Most of what we're calling transactional locking relates to the ability of a database management system (DBMS) to ensure reliable transactions that adhere to these ACID properties. ACID is an acronym that stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*. Each of these properties is described in more detail below. However, all of these properties are related and must be considered together. They are more like different views of the same object than independent things.

2.1.1 Atomicity

Improve Retrospectives

Measure
performance
the competition
focus your

[Software Testing Magazine](#)

[The Scrum Expert](#)

[Subscribe](#)

Atomicity means all or nothing. Transactions often contain multiple separate actions. For example, a transaction may insert data into one table, delete from another table, and update a third table. Atomicity ensures that either all of these actions occur or none at all.

2.1.2 Consistency

Consistency means that transactions always take the database from one consistent state to another. So, if a transaction violates the databases consistency rules, then the entire transaction will be rolled back.

2.1.3 Isolation

Isolation means that concurrent transactions, and the changes made within them, are not visible to each other until they complete. This avoids many problems, including those that could lead to violation of other properties. The implementation of isolation is quite different in different DBMS'. This is also the property most often related to locking problems.

2.1.4 Durability

Durability means that committed transactions will not be lost, even in the event of abnormal termination. That is, once a user or program has been notified that a transaction was committed, they can be certain that the data will not be lost.

3. Examples of Simple Locking

The example below illustrates the most common and logical form of transactional locking. In this case, we have 3 transactions that are all attempting to make changes to a single row in Table A. U1 obtains an exclusive lock on this table when issuing the first update statement. Subsequently, U2 attempts to update the same row and is blocked by U1's lock. U3 also attempts to manipulate this same row, this time with a delete statement, and that is also blocked by U1's lock.

When U1 commits its transaction, it releases the lock and U2's update statement is allowed to complete. In the process, U2 obtains an exclusive lock and U3 continues to block. Only when U2's transaction is rolled back does the U3's delete statement complete.

This example shows how the DBMS is maintaining consistency and isolation.

Time	User 1 Actions	User 2 Actions	User 3 Actions
1	Starts Transaction		
2		Starts Transaction	
3			Starts Transaction
4	Updates row 2 in table A		
5		Attempts to update row 2 in table A	
		<i>U2 Is Blocked by U1</i>	
6			Attempts to delete row 2 in table A
			<i>U3 Is Blocked by U1</i>
7	Commits transaction		
		<i>Update completes</i>	<i>U3 Is Blocked by U2</i>
8		Rolls back transaction	
			<i>Delete completes</i>
9			Commits transaction

4. Locks Outlive Statements

It is critical to understand that locks will often remain after a statement has finished executing. That is, a transaction may be busy with different, subsequent activity but still

hold locks on a table due to an earlier statement. Transactions may even be idle. This is especially dangerous if the application allows user think time within database transactions.

This concept that locks outlive statements may seem obvious, but it is often forgotten when considering the impact of locking, so it is very important to remember.

5. Issues that can occur

So what does this mean for your application? Well, there are a number of problems that can be caused by database locking. They can generally be broken down into 4 categories: Lock Contention, Long Term Blocking, Database Deadlocks, and System Deadlocks. Each of these types of issues is discussed in more detail below, including examples.

Note: The examples used are quite extreme, but are meant to be illustrative.

5.1 Lock Contention

Lock contention occurs when many database sessions all require frequent access to the same lock. This is also often called a "hot lock". The locks in question are only held for a short time by each accessing session, then released. This creates a "single lane bridge" situation. Problems are not noticeable when traffic is low (i.e. non-concurrent or low-concurrency situations). However, as traffic (i.e. concurrency) increases, a bottleneck is created.

Overall, Lock Contention problems have a relatively low impact. They manifest themselves by impacting and limiting scalability. As concurrency increases, system throughput does not increase and may even degrade (as shown in Figure 1 below). Lock contention may also lead to high CPU usage on the database server.

The best way to identify a lock contention problem is through analysis of statistical information on locks provided by the DBMS (see monitoring section below).

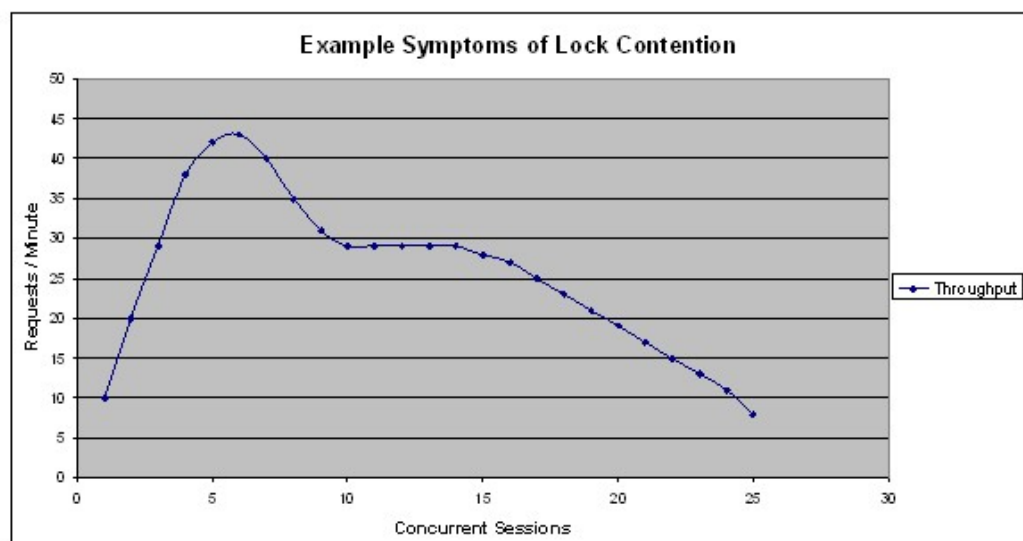


Figure 1: Example Symptoms of Lock Contention - Scalability

5.1.1 Example: Statistics Table

A web application has a requirement to record the number of times each page is loaded. This has been implemented with a database table where the access counts for each page are stored, with one row for each page. Each time that a request is made to a page, the appropriate row in the tables is updated to increase the request count number by 1.

The related update statement is not a problem for low concurrency situations, as the requests are fast and the same page is not loaded concurrently. However, as load increases, popular pages are concurrently being accessed. This leads to contention and limits scalability.

5.1.2 Example: Order ID Storage

An order processing system requires that Order IDs be unique and sequential. This is implemented through use of a database table to store the value of the next order ID. As each order is processed, the transaction gets the value for the next order ID then updates it with the next ID.

Under low concurrency circumstances, order id generation does not have a significant impact. However, as the number of concurrent orders increases, a bottleneck is created due

to contention on the associated lock, limiting order throughput.

5.2 Long Term Blocking

Long Term Blocking is similar to Lock Contention in that it involves an object or lock that is frequently accessed by a large number of database sessions. Where it differs is that in this case, one session does not release the lock immediately. Instead, the lock is held for a long period of time and while that lock is held, all dependent sessions will be blocked.

Long Term Blocking tends to be a much bigger problem than Lock Contention. It can bring an entire area of functionality or even a whole system to a stand still. The locks involved in these scenarios may not be "hot" enough to lead to Lock Contention problems under normal circumstances. As such, these problems may be intermittent and very dependent on certain coincidental activity. These are the most likely to lead to "disasters" in production due to the combination of devastating impact and difficulty to reproduce.

The consequences of long term blocking problems may be abandonment. However, these problems can also often lead to further problems as frustrated users re-submit their requests. This can compound and exacerbate the problem by leading to a larger queue and consuming additional resource. In this way, the impact can expand to consume an entire system.

5.2.1 Example: Order ID Batch Processing

Consider the same system described in 5.1.2 above. But now imagine that the system also has a batch process that is set to run at 7 p.m. This batch process is optimized to complete multiple orders within one transaction. As such, it will acquire a lock on the Order ID row at the beginning of processing. If it then takes 15 minutes to process the entire batch, that lock will be held for the duration and no other orders can be processed until it completes, commits the transaction, and releases the lock. This situation is illustrated in the graph below, where order processing times are very fast (a few seconds) during most of the day, but then shoot up to 900 seconds during the 7 p.m. hour.

Users trying to process orders during this time think that the system has gone down. Many resubmit their requests, leading to a queuing of duplicate requests which cause problems later due to duplicate orders. The repeated requests also lead to exhaustion of threads and database connections, making the system unable to respond to any requests, even those unrelated to orders.

This might not have been found during testing, because only small batches were tested or batch processing was not tested concurrently with regular usage.

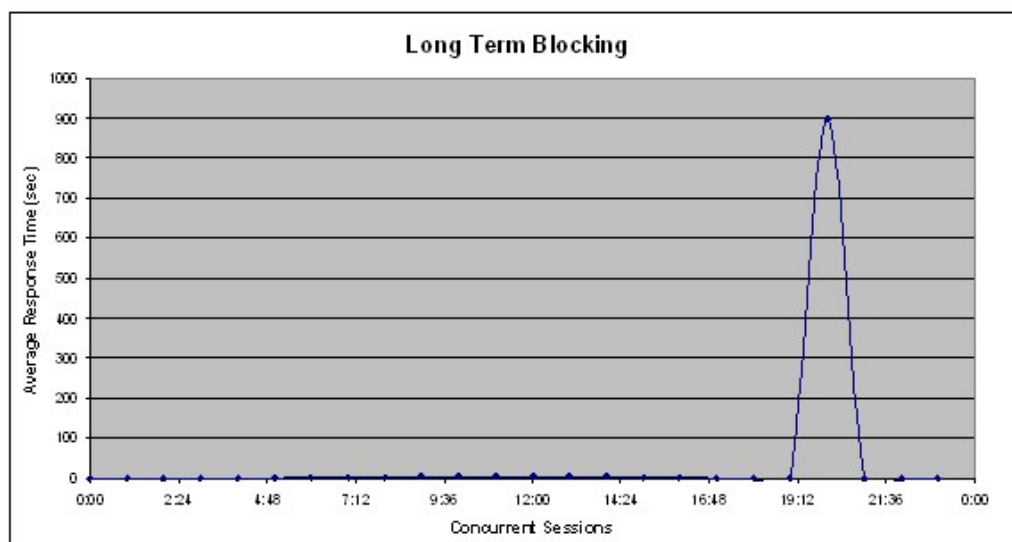


Figure 2: Example Symptoms of Batch Locking

5.3 Database Deadlocks

Database Deadlocks occur when 2 or more transactions hold dependent locks and neither can continue until the other releases. Below is a simple illustration of a deadlock.

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	
2		Starts Transaction

3	Updates row 2 in table A	
4		Updates row 10 in table B
5	Attempts to update row 10 in table B	
	<i>U1 Is Blocked by U2</i>	
6		Attempts to update row 2 in table A
	<i>U1 Is Blocked by U2</i>	<i>U2 Is Blocked by U1</i>
	DEADLOCK!!!	

This example is the simplest type. Deadlocks can be much more complicated, involving different types of locks, and involving more than 2 sessions.

The good news is that the DBMS will recognize these situations and resolve them. The DBMS will choose one "victim" and roll back their transaction. The DBMS tends to choose the "victim" by determining which transaction will be easiest to roll back. However, unless the application is designed to react to deadlock errors and retry, there will be a negative end-user impact.

Deadlocks generally lead to intermittent errors. They are *almost* always caused by application logic problems. These generally occur when the application does not access data in a consistent sequence. They are also very timing and data dependent, which can make them very hard to reproduce.

Note: Deadlocks can occur within other components as well, such a JVM or the CLR.

5.4 System Deadlocks

System Deadlocks (aka Distributed Deadlocks) are similar to a Database Deadlock, in that they involve dependent locks. However, they involve locks that are outside the database (e.g. in the JVM) as well. As a consequence, the database cannot identify the deadlock situation. It simply appears to be a normal blocking situation. The same will be true for the external component. As such, these situations are usually not resolved until a timeout occurs or one of the components is restarted.

System deadlocks are relatively rare, but they do occur. And when they do, they can lead to system outages as well. In many cases, the impact will grow because the system deadlock will lead to secondary long term holding of locks which result in the Long Term Blocking for other sessions.

5.4.1 Example: Start-up Deadlock

A J2EE application has a start-up thread that's responsible for initializing various parts of the system. A "system change number" is stored in a database table and is updated every time that key tables are changed. The first start-up task makes such a change and updates the system change number, acquiring a lock as it does.

Subsequently, the application executes an EJB method that has been configured with the "Requires New" property. As such, the application server must create a new database connection and transaction to execute that EJB method. This particular method also leads to changes that require the same system change number to be updated.

At this point the start-up thread is waiting for the EJB method to complete. However, this method is being blocked by the database lock associated with the first start-up transaction. The database sees a simple blocking situation, as does the JVM. However, this problem will never resolve and the system never starts up.

5.4.2 Example: Session Counts

A clustered .NET web application has requirements to track the number of sessions logged into each application server and the entire system. Each application server tracks the session count with in memory counters while the system wide total is tracked in a database table. The in-memory session tracking was protected with a locking mechanism such that three methods must be called in order to update it. The update of the database is completed with a single method. When implementing the login, the developers coded the following sequence:

1. acquireSessionCountLock

2. updateSessionCountValue
3. updateDBSessionCount
4. releaseSessionCountLock

This was done so that, if the database updateDBSessionCount call failed, then the updateSessionCountValue call could be reverted to ensure that a discrepancy did not occur between the two session counts.

However, another developer implemented the logout request in the following way:

1. updateDBSessionCount
2. acquireSessionCountLock
3. updateSessionCountValue
4. releaseSessionCountLock

That is, the database will be updated, after which the in-memory counter is updated (including acquisition and release of locks). This would not be a problem under low concurrency situations. However, with sufficient load and concurrency, the following situation would occur, creating a System Deadlock.

Time	User 1 Actions	User 2 Actions
1	Login Request Received	
2		Logout Request Received
3	Executes acquireSessionCountLock	
4		Executes updateDBSessionCount
5	Executes updateDBSessionCount	
	<i>U1 Is Blocked by U2's DB Lock</i>	
6		Executes acquireSessionCountLock
	<i>U1 Is Blocked by U2's DB Lock</i>	<i>U2 Is Blocked by U1's App Lock</i>
	SYSTEM DEADLOCK!!!	

Neither the CLR nor the DB would be able to detect that this deadlock and it would go unresolved. Subsequent Login requests to this app server would be blocked in the acquireSessionCountLock call. Requests to other application servers would be blocked in the updateDBSessionCount call due to the database lock. As a result, the system would become non-functional.

6. Complications and Platform Variation

The situations discussed so far have involved database locking that is quite simple, logical, and expected. Most problems can be avoided with sound development practices. However, there are further complications that can lead to additional types of locking that result in more frequent locking and locking that may seem illogical or not strictly necessary. These complications are also generally a result of DBMS differences. When the frequency of locks and locking events increases, the frequency of the problems described above also increases. And they also begin to appear in unexpected situations.

6.1 Select Blocking

Select blocking is closely related to the isolation property of ACID transactions. In order to ensure that transactions are isolated, the DBMS must be sure that changes made by one transaction are not visible to others until the transaction commits. For example, if one transaction updates a row in a table and another session tries to select the updated row, the DBMS cannot allow the changes to be seen by the second session.

The solution to this problem is variable. Oracle's implementation is to maintain separate versions of the related data blocks. The uncommitted changes made in one transaction are visible within the transaction that made them, while other sessions will see the old, unchanged version. In this way, isolation is maintained and *select statements never block*.

SQL Server and other DBMS' have traditionally implemented this differently. Instead of maintaining separate versions, they use locks to protect access to the changed data. This leads to *Select Blocking*. The difference is illustrated in the examples below.

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	

2		Starts Transaction
3	Updates row 2 in table A	
4		Attempts to read row 2 in table A
		<i>Select statement completes, returning data unchanged by U1</i>
5	Commits transaction	
6		Attempts to read row 2 in table A
		<i>Select statement completes, returning changed data</i>

Table 1: Select Behaviour in Oracle

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	
2		Starts Transaction
3	Updates row 2 in table A	
4		Attempts to read row 2 in table A
		<i>U2 Is Blocked by U1</i>
5	Commits transaction	
		<i>Select statement completes, returning changed data</i>

Table 2: Select Behaviour in SQL Server

The examples above are quite simple and involve two sessions attempting to access the same row of data. However, select blocking can also come into play in situations that might not be expected if one were not considering how the DBMS works. Consider the following situation.

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	
2		Starts Transaction
3	Updates row 2 in table A	
4		Attempts to read row 200 in table A
		<i>table A has no indexes, so the query requires a full table scan</i>
		<i>U2 Is Blocked by U1</i>
5	Commits transaction	
		<i>Select statement completes, returning data</i>

In this case, User 2 is not explicitly requesting the data that had been changed by User 1. However, in order for the DBMS to return that data, it must read all records in the table, including the one changed by User 1. As such, User 2 will be blocked until User 1 releases that lock.

This difference illustrates how isolation can be implemented differently. Both are valid solutions to that problem. But this is also an illustration of how different DBMS' can have

very different behaviour. This behaviour will then translate into much different behaviour for your application and much more or different problems.

In general, select blocking greatly increases the frequency of locking events and, as a result, of the locking problems discussed above.

6.2 Non-Row Locking

Locking does not always involve row specific locks. SQL Server stores data in pages, with each page containing the data for multiple rows. As such, locks from one transaction may impact another transaction even though they are not accessing the same row, but because the rows they are concerned with share the same data page.

6.3 Lock Escalation

For DBMS such as SQL Server, locks are managed in memory and therefore consume memory resources. As the number of locks increase, so do the memory resources required by the DBMS to track these locks. These resources can be significant and, at some point, can overwhelm the resources available to the database. In order to manage this situation, the DBMS may *escalate* locks to a higher level. For example, if one transaction is holding many locks on a single table, the DBMS can escalate that lock to the table level. This will greatly reduce the resources required to maintain that lock and isolation. However, it can greatly impact concurrency because now the entire table has been locked.

In SQL Server, Lock Escalation occurs when:

- A statement locks more than 5000 rows in a table
- Lock resources consume > 40% of total memory

Note: Things are a bit more complicated, but this explanation is sufficient for this level of discussion

In these cases, locks may be escalated to the table level, introducing different locking events and locking behaviour than would be otherwise seen. This type of locking will also be much more intermittent and difficult to reproduce. For example, they may never be seen in testing if the data accessed never leads to large updates (>5000 rows) or the load on the system never leads to sufficiently high lock memory usage. But these situations may occur in production and lead to unanticipated behaviour and problems.

6.4 ITL Locks In Oracle

As discussed above, Oracle does not have a global lock manager, but instead stores locking information at the level of the data block. Within each data block there exists a simple data structure called the "Interested Transaction List" (ITL). The initial size of this list is determined by the INITRANS storage parameter. Space permitting, the number of slots in the list is allowed to grow up to the value of the MAXTRANS storage parameter.

Often INITRANS is set to 1. This then allows a situation to occur where row data fills up the rest of the block and, even if MAXTRANS is greater than 1, there is no room for the ITL to grow. As such, if a block contains many rows and only has 1 ITL slot, then the rows in that block can participate in only one transaction at a time. If another transaction tries to lock another row (unrelated to those locked by the first), that transaction will block.

This behaviour can lead to additional locking in Oracle, which would not otherwise be expected.

7. What to do?

So now that we've gotten to know our enemy a bit better, how can we as QA professionals use this to avoid disaster? This section outlines the 4 complementary strategies that will help you to find these problems in the testing stages.

7.1 Use Appropriate Test Data

Using the right test data is crucial. Based on our discussion of locking issues above, it should be clear that data is important: different data will result in different behaviour. As such, it is vital that test data is chosen and designed specially with locking issues in mind.

When talking about test data, there are basically three aspects to consider: Active Data, Background, and Database Structure. All are important.

7.1.1 Background Data

Background data is data that's in the database, but is not necessarily accessed during testing. Its purpose is to provide "weight" to the database and ensure that testing validly reflects reality. For example, if your test database is 1 GB in size, containing tables with row

counts in the order of 10,000 rows, but production deployments are in the TB range, you can expect to miss many things in testing. As such, the following guidelines are important to ensure that locking issues are found during testing:

- The physical size (i.e. in GB) should approximate production sizes
- Row counts in key tables should approximate production sizes
- Distribution of data within tables should approximate production data

These are important because the size and distribution of data affects the way that the database accesses data (e.g. execution plans for a given query), the amount of time that actions take to complete, and the amount of resources required to complete them. These, in turn, will greatly affect the frequency of locking events and locking problems.

For example, imagine an update statement that updates a large portion of rows in a table and requires a full table scan. If a small database was used for testing, this statement would complete quickly and would not result in lock escalation. However, with a larger database, the update statement could take much longer to execute. This could lead to a Lock Contention problem if this statement was frequent enough. It could also lead to a Long Term Blocking problem if the statement caused lock escalation.

7.1.2 Active Data

Active data is the data that will be actively accessed and utilized during testing. This data is also critically important to ensuring that locking problems are reproduced in testing. Here are several areas to consider:

- The amount of data accessed
E.g. How many items do users really have on their home pages?
- The overlap of data accessed concurrently
E.g. How many users are generally in each group and how many groups are there per user in the system?
- The completeness of data accessed
E.g. How many sections are there on the homepage that could contain data?

The goal should always be to ensure this validly reflects reality whenever possible. It is generally best in a test environment to err on the "bad" side, with active data being "bigger" than 90% of real situations. But it is also prudent to not be too extreme, as this can lead to overemphasis and cause problems to appear in testing that would never occur in production.

7.1.3 Database Structure

This may seem like common sense, but it is absolutely critical that the structure of the database (including table definitions and indexes) be the same in testing as it is in production. The presence or absence of an index can completely change locking behaviour, for the better or worse.

7.2 Implement Appropriate Monitoring

Another vital strategy is to ensure that appropriate monitoring is implemented to allow identification of locking problems during testing. Without appropriate monitoring, Lock Contention and Database Deadlock problems may occur but not be noticed or identified for what they are. This is especially problematic with intermittent and timing dependent problems. Also, Long Term Blocking or System Deadlock problems may occur, but sufficient information will not be available to identify them as such and to enable diagnosis of the cause.

The exact monitoring required will depend on your application and DBMS. However, the following are crucial:

- Detect Long Term Blocking events and ensure that sufficient information is gathered. Also ensure that notification is sent out and that processes exist to ensure that these events are investigated.
- Detect Database Deadlock events and ensure that notification is sent out and that processes exist to ensure that these events are investigated.
- Capture statistical information on locking and ensure that there is a process to review this regularly and act upon problems identified.
- Capture information on application level blocking (e.g. thread contention)

The appropriate tools for such monitoring vary with DBMS and application technology.

7.3 Design Explicit Locking Tests

Designing explicit locking tests is also critical. By considering the possible locking problems and your specific application, it is possible to prepare test that will increase the chances of finding locking problems during testing. This testing, in general, will need to involve performance or load test automation in order to be successful. This testing needs to consider 3 factors.

7.3.1 Normal Activity

When designing explicit locking tests, it is best to start with tests that simulate "normal activity". This vague term is meant to indicate the workload that the application or system is most often required to support. This would generally include short requests or actions involving a wide breadth of functionality.

7.3.2 Long Running Tasks

As opposed to Normal Activity, Long Running Tasks are the less frequent components of workload that tend to take a long time to execute. These are often batch or schedule tasks which would be run infrequently during a day or week. Examples would be data synchronization events, index rebuilds, or backups. Depending on the application or system, they may also involve end-user initiated actions that are expected to take a long time to execute. Examples would be report execution, user initiated exports, or batch imports.

7.3.3 Data Overlap

Data overlap is an important concept in defining explicit locking tests. Levels of data overlap must be explicitly designed into the tests. The reason for this is that some locking would be expected. For example, if 10 users were all attempting to update the username of the same user, blocking and contention would be expected. However, if 10 users were all attempting to update the usernames for different users, then blocking and contention would not be expected or desirable.

When defining these tests, the following sequence is recommended.

1. Start with Normal Activity
2. Add in Long Running Tasks, starting with minimal data overlap
3. Increase level of data overlap

7.4 Combine Automation w/ Manual Testing

The fourth strategy is to combine automation with manual testing. In many testing processes, Performance Testing (under which much of the above would be categorized), functional automated testing, and manual testing are completed separately. They often involve different teams and different equipment.

However, this can greatly limit the ability to reproduce locking problems during testing. This is because automated testing, even that which follows an explicit locking test plan, will almost always be narrower in scope than manual testing. Automation creates a "beaten path" where problems are quickly identified and resolved. Conversely, manual testing environments often involve broader and more variable activity. This usage is also often more valid than that which results from automation.

By combining automated load testing with manual testing, it will be possible to reproduce a much broader set of locking issues than would be possible through load testing alone. And by implementing the same monitoring used to detect locking in fully automated tests, one can be sure that problems which do occur will be identified.

Combining these in a UAT environment can also help to assess the severity of locking problems that occur. Load test automation that pushes a system to extreme levels of concurrency may result in locking issues which would not occur under normal conditions. Running load tests, at realistic levels, during UAT can help to assess whether these problems will actually occur and whether they must be fixed prior to release.

8. When To Be Concerned

Locking issues and their relative impact can be very hard to predict. However, the following is a list of some scenarios where QA professionals should be particularly concerned.

1. The application has a combination of short and long running transactions
2. An application is being ported from Oracle to another DBMS
3. An application is moving from a standalone application tier to a clustered application tier.
4. The application accesses multiple databases.

How to fix - Workarounds

This paper is mainly focused on how to identify problems during testing. Diagnosing and resolving these problems often requires code changes by developers which are beyond the scope of action for QA professionals. However, there are some configuration level changes, not involving code changes, which can help. There are also some workarounds that the QA professional should know about, since the implications can be problematic.

9.1 SQL Server: NOLOCK Hints

Most DBMS's allow for *hints* to be provided in SQL statements. In general, these do not change the end result of a statement but will alter the execution plan or some other behaviour. However, one such hint which is often used in SQL Server to work around problems of Select Blocking is the NOLOCK hint.

A NOLOCK hint tells the DBMS to relax the Isolation property and leads to "dirty reads". This allows a statement to complete without placing any locks, thereby avoiding Select Blocking and reducing the overhead on the lock manager. However, this will lead to uncommitted data being returned.

QA professionals should be aware of all places where NOLOCK hints are used within applications because they can cause strange and complicated problems. This is especially true when a NOLOCK hint is used in a query that will populate a cache. In this scenario, cache corruption can occur if "dirty data" read into the cache is then rolled back. This can also lead to variable behaviour depending on whether the data is cached or results in a database query.

9.2 SQL Server: READ_COMMITTED_SNAPSHOT

SQL Server 2005 introduced the new READ_COMMITTED_SNAPSHOT isolation level. This may be enabled for an entire database and will result in behaviour similar to that seen with Oracle (i.e. select statements will generally not block). This is not enabled by default and must be explicitly enabled. Since the TEMP DB is used to store older versions, this can put additional strain on the TEMP DB. However, this option's overhead is generally manageable and well worth the benefit of reducing locking issues.

9.3 SQL Server: Disable Lock Escalation

Lock escalation can be disabled using the TraceFlag-1211 or TraceFlag-1224 trace flags. However, these should be used with caution as they can lead to other problems related to exhaustion of memory.

9.4 SQL Server: Avoid Full Table Scans with Indexes

If common queries are leading to full table scans on SQL Server without READ_COMMITTED_SNAPSHOT, this can increase the chances of locking problems. This is because the full table scan will access every row and will be impacted if any row is locked. Creating an index that is included in the execution plan will reduce the incidence of locking.

9.5 Oracle: Size INITRANS

Problems relating to ITL locks can often be avoided by setting a larger INITRANS parameter. This database level change can often be implemented during installation of your system and can be implemented after the fact (though this requires a rebuilding of the related objects). The proper size depends on many factors and a larger INITRANS will result in less compact data storage. As such, these changes should only be done in close consultation with a trained DBA.

Related Methods & Tools articles

[The Return of ACID in the Design of NoSQL Databases](#)

[Behavior Driven Database Development \(BDDD\)](#)

More Database Knowledge

[Database Tutorials and Videos](#)



[Click here to view the complete list of archived articles](#)

[This article was originally published in the Spring 2009 issue of Methods & Tools](#)

Explore a list of [Free and Open Source Scrum Tools for Agile Software Project Management](#)

Browse a selected list of upcoming [Software Development Conferences](#)

Copyright © by 1995-2019 [Martinig & Associates](#) | [Network](#) | [Advertise](#) | [Contact](#) | [Privacy](#)

Follow Methods & Tools on    

Methods & Tools uses Google Analytics for statistics and Google AdSense for advertising.
[You can consult Google Privacy Policy here.](#)