Optimizing Concurrency Through Automated Lock Memory Tuning in DB2

Sam S. Lightstone  Chris Eaton  Yun Han Lee  Adam J. Storm
light@ca.ibm.com  ceaton@ca.ibm.com  yunhanl@ca.ibm.com  ajstorm@ca.ibm.com
IBM Canada

Abstract

Lock memory consumption can be difficult to project and can vary rapidly in short amounts of time. This volatility makes lock memory tuning difficult and can result in either significant memory waste if systems are configured for peak requirements, or lock escalation and lock wait if under configured, either of which can cause significant performance penalties. This paper describes an algorithm for adaptive tuning of database lock memory. The DB2 technique adapts the locking memory in real time to mitigate the occurrence of lock escalations. The technique uses a combination of synchronous and asynchronous modification to the locking structures so that it can respond well to rapid immediate growth in locking requirements. The adaptive algorithm also relaxes the locking memory over time so that peak requirements in lock memory will not result in a permanently large allocation of memory to locks. Experimental tests have shown this technique to work well in a number of benchmark and adaptive workloads, converging almost immediately to optimal settings which avoid lock escalations and achieve optimal throughput. The solution has been implemented in DB2 9.

1. Introduction

Since the development of the relational model by E.F. Codd at IBM [1], relational databases have become the de facto standard for managing and querying structured data. The rise of the Internet, online transaction processing, online banking, and the ability to connect heterogeneous systems have all contributed to the massive growth in data volumes and highly concurrent systems over the past 15 years. Modern databases frequently need to support hundreds if not thousands of concurrent users. This requires locking logic to maintain the data integrity while still providing the optimal availability to the data. RDBMSs require memory to store the locking structures and optimal tuning for this memory allocation can be non-trivial.

Within the relational database industry, vendors typically support row locking on data, as well as table locking. Some vendors also offer page locking. Row locking provides the best granularity, but also consumes the most memory (one lock per row, plus counters and locking modes that describe the number of clients holding each lock). When lock memory is constrained a RDBMS will “escalate the lock”, promoting one or more row level locks to either a page level lock or a table level lock. Lock escalation dramatically reduces the memory requirements for locking (since a single table lock can be held instead of thousands or millions of row locks) but may have an extremely detrimental impact on system concurrency. Therefore, lock escalation is an extremely poor alternative to lock memory tuning, and in general should be avoided in situations where high concurrency is required. Lock memory requirements vary widely by application. A brief informal review of database users revealed typical allocation rate between 1% and 10% of available database memory, and in some cases (including cases with some of the most widely used enterprise applications) allocation rates are as high as 15-20% of system memory. Therefore the problem of optimal lock memory selection becomes non-trivial. Under-allocation can prove disastrous due to lock escalation. Over-allocation of memory for locking reduces the available memory for other purposes such as the main memory cache (bufferpools), sorting, and communication buffers, etc. Some vendors offer implicit or automatic tuning of these memory structures, discussed below, obviating the need for human tuning, but existing techniques appear to be either very wasteful of memory or have problematic qualities. The adaptive lock memory tuning described here is part of DB2’s R&D efforts in autonomic computing [2].

The rest of the paper is organized as follows: section 2 gives an overview of related literature, section 3 describes the approach used to tune memory in DB2, and section 4 illustrates the process using a worked example. Section 5 describes experiments performed to demonstrate some of the qualities of the self-tuning lock memory algorithm, and section 6 concludes the paper with a brief summary of results and future work.

2. Background

2.1 DB2’s memory management

DB2 v8.2 introduced the notion of database overflow memory, which represents a small reserve of memory that can be used on demand by heaps within the database shared memory set. Overflow memory is simply memory allocated to the database but not yet in use by a memory consumer. When memory heaps need to exceed their configured allocation they can automatically expand into the overflow memory on a first come-first-served basis.
In addition to the overflow memory, DB2 9 includes autonomic self-tuning of database memory. The lock memory tuning described in this paper is part of the new autonomic self-tuning memory feature referred to as the Self-tuning Memory Manager (STMM)[3]. STMM tunes several heaps at runtime in order to remove the administrative burden of manual tuning. This has two benefits: first, reduced TCO, since several complex tasks are removed or greatly simplified and second improved performance in many cases when the self-tuning algorithm yields a superior memory distribution than could be produced with manual tuning.

STMM will determine the following at runtime:

- The total amount of memory allocated to a DB2 database, \( \text{databaseMemory} \)
- The value of specific heap sizes, such as bufferpools, sorts, hashjoins, packagecache and lock memory.
- The size of the memory overflow area.
- The tuning interval (time between adjustments)

Within STMM memory consumers (heaps) are divided into two major categories:

1. Performance related memory consumers (PMCs) have the strong potential to affect system performance, but not usually query success or failure, by the amount of memory they are allocated.
2. Functional memory consumers (FMCs) require memory to store data, without which database operations will fail.

Examples of PMCs include: bufferpools, sort, hash join, compiled statement cache. Examples of FMCs include memory for internal control blocks, SQL/XQUERY compilation memory and statistics collection memory.

The lock memory is being modelled as a functional memory consumer since lock escalations can have an effect on the system that is similar to denial of service.

2.2 The DB2 Lock Manager

DB2’s lock manager memory is allocated from the database shared memory set. The amount of memory initially allocated is controlled by the LOCKLIST configuration parameter. This parameter specifies the number of 4KB pages that will be used for lock memory.

When the lock memory is allocated, the allocations are made in 128 KB blocks with one allocation is made for every 32 pages of LOCKLIST memory required. Each 128 KB memory block is enough memory to store approximately 2000 locks. As the memory blocks are being allocated they are added to a linked list.

When lock requests arrive at the lock manager, memory must be taken from the linked list to store the requested locks in memory. This is done by taking locks from the first memory block on the list. Once the locks from the first memory block (A) have been exhausted, that empty memory block is taken off the head of the list and put on the “empty block” list. This causes the second memory block (B) to be the new head of the list. As a result, when the next lock request arrives, the requested locks will be taken from memory block B. When the locks that were allocated from block A are freed, the memory block is returned to the linked list at the head. When this occurs, any new lock requests will again be satisfied from block A.

Since this algorithm places memory block A back at the head of the list when some of its lock memory becomes available for new requests, it is evident that if the locking demands of the database require only half of the allocated lock memory, memory blocks towards the end of the list will always be entirely free. This is advantageous when a request arrives to decrease the amount of memory being used by the lock manager.

When a request arrives to decrease the amount of lock manager memory some of the allocated memory blocks must be freed. When freeing memory blocks, the lock manager first looks for empty memory blocks at the end of the list since, if there are any free memory blocks it is likely that they will reside at the end of the list. Starting from the end of the list, the memory blocks are scanned and the blocks with no outstanding lock requests, are set aside to be freed. If the scanning of the list reveals that there are enough freeable memory blocks, the list of freeable memory blocks is deallocated and the request succeeds. If the scan reveals that there are not enough freeable memory blocks, any blocks that have been set aside to be freed are reintegrated into the list and the request fails.

DB2 also includes a parameter named MAXLOCKS which controls the percentage of lock memory that any single application can hold. For example, if MAXLOCKS is 10, then a single application may hold no more than 10% of the system lock memory (LOCKLIST). If an application acquires enough locks to saturate this portion of the lock memory, further requests for locks will result in lock escalation. Throughout the remainder of this paper we will
refer to the MAXLOCKS parameter as lockPercentPerApplication to improve the readability of the text.

Section 3 will describe how both total lock memory and the per-application constraints on lock memory consumption are adaptively tuned.

2.3 Existing industrial approaches

Current implementations for row level locking in commercial databases typically fall into one of two categories. The first is employed by both Microsoft SQL Server[9] and DB2 LUW 8.x in which lock structures are stored in memory. A lock record is stored in memory when an application acquires a row level lock and that lock record is removed from memory when the lock is released. Other application that request a lock on the same row in a compatible mode (i.e. two applications both locking the row in share mode) would share the same lock record while lock requests on the same row that are incompatible would be represented by a chain of lock requests. For example, assume that there are four applications (app_1, app_2, app_3, app_4) all trying to access a given row (row_x). The first application reads the row and acquires a share lock on that row. This results in a lock object being stored in memory. If application 2 then asks for a share lock on the same resource, it would be granted that lock object would now have two applications associated with it. If the third application requests the lock in an incompatible mode, it would result in a lock chain being formed to indicate that when app_1 and app_2 release their locks, app_3 would be able to acquire that lock object. If application 4 then requests a share lock on that row, it will queue up behind application 3 as shown in Figure 3.

![Figure 3. Lock queuing](image)

Microsoft SQL Server 2005 allows the lock memory to grow dynamically. SQL Server 2005 will initially allocate enough memory for 2500 locks. Should this amount of memory be consumed by lock requests, the database manager will automatically allocate additional lock memory but only up to a maximum of 60% of the total database server memory. In the SQL Server 2005 case, a lock escalation occurs when the memory consumed for locks reaches 40% of the total database engine memory. This is not a configurable parameter. In addition, if a single application acquires 5000 row level locks an automatic lock escalation is triggered regardless of the amount of memory available for locks. As a result, a single reporting query can easily result in lock escalation. This too is not configurable. While the SQL Server memory manager allows memory heaps to grow and shrink, we failed to find a clear statement in the SQL Server documentation that the SQL Server lock manager has the ability to return memory to the global pool.

There is however a distinction to be made between the implementations available from IBM and from Microsoft. DB2 9 allows greater flexibility in self-tuning the lock memory, allowing it to:
- Grow lock memory allocation synchronously and asynchronously
- Adaptively control the number of row locks an application can acquire before an escalation occurs
- Avoid changes to lock memory allocation caused by minor variation on lock requests
- Asynchronously reduce lock memory allocation during periods of low locking requirements.

The Oracle method for lock management is significantly different[10][11]. Rather than storing locks in distinct memory, Oracle stores locks on the actual data pages in which the row objects reside. This has the side affect that locks also consume memory when data pages are brought into the buffer cache regardless of whether the data on any of the rows is being accessed. With this approach every row has a lock byte associated with it. If a row is locked this lock byte is set to inform other applications that the row is locked in exclusive mode. In addition there is also an interested transaction list (ITL) stored on the data page in which transactions store their identifiers, the rows on the page they have locked as well as a pointer to the undo tablespace section in which they have stored past images of this data page (for undo processing) as shown in Figure 4.

![Figure 4. Oracle page memory](image)

The advantage of this strategy is that lock memory is pre-allocated. However, this method has several disadvantages. Firstly, extra disk space is consumed for lock information. This consumption is permanent. In fact as more transaction register concurrent interest in rows on a page, the ITL section of that page increases and is not decreased until the table is reorganized. Secondly, the applications in the ITL section which are waiting on a lock to be released go into a sleep, wake, check, sleep cycle to determine if the row lock they are waiting on is released. The previously described memory chaining method uses a post method so that requesters are serviced in the order in which they request
locks. The Oracle method results in the possibility that another transaction can acquire a lock which is being waited on by a sleeping application, thus “jumping” the queue. A second disadvantage is that the space on each data page is finite. Should the allocation of ITL space be consumed, any new application wishing to acquire space in the ITL in order to lock a row on that page must wait. This is true even if the row that this new application wants to lock is not locked by any other application. In effect, the exhaustion of ITL space results in page level locking. A third disadvantage is that the data page may be flushed from the buffer cache prior to the transaction committing which means that the data page on disk still has the associated lock byte set. At some future point in time, when that page is read back into the buffer cache another application will be required to consume extra CPU cycles to check if this row is really locked and to clear out the lock byte and ITL slots if the old transactions are no longer active. This method employed by Oracle does not allow for any dynamic allocation (either increase or decrease) of lock memory as it is fixed with the amount of space consumed on the page.

3. Design overview

3.1 High level description

The lock memory will be designed to consume memory on-demand (i.e. in real-time) from the DB2 database shared memory set in order to avoid lock escalation, up to some reasonable limits (described below). The proposed algorithm uses a combined asynchronous and synchronous allocation model to achieve stable behaviour. Lock memory will be tuned as a deterministic heap, meaning specifically that a cost-benefit model will not be created for lock memory.

The basic tuning model is as follows.

- STMM will asynchronously (at each memory controller tuning interval) aim to maintain the lock memory allocation such that a set percentage of all lock structures are allocated but unused by any application (i.e. minFreeLockMemory).
- When lock consumption slumps (i.e. much more than maxFreeLockMemory empty) STMM will asynchronously reduce lock memory size by δreduce (a small percentage) each tuning interval until it achieves the maxFreeLockMemory empty state. The slow reduction stabilizes the control of the heap allocation.
- By keeping at least minFreeLockMemory of the lock memory free during any single tuning interval the system can accommodate a large growth in locks during one STMM interval without allocating any new memory synchronously to lock memory.
- Sudden spikes in lock memory usage (that exceed the freelist of lock structures) synchronously consume memory from database overflow memory allowing lock memory to grow on demand and avoid lock escalations.
- Reasonable limits on minimum and maximum lock memory size constrain the growth pattern. These limits are based on the number of applications using the database and percentage of total databases memory.
- For massive spikes when database overflow memory is constrained (a real but rare case), locks will escalate, but lock memory will double each tuning interval while escalations are continuing, trending towards a well tuned allocation despite the temporary presence of lock escalations.

lockPercentPerApplication will be kept high provided lock memory is ample. As lock memory approaches its maximum size, lockPercentPerApplication will decrease precipitously.

Sections 3.2-3.6 describe the algorithm in more detail. Table 1 in section 3.7 summarizes the modelling parameters.

3.2 Minimum and maximum allocations for lock memory

Allowing the lock memory to grow indefinitely is a poor strategy, since massive locking requirements are often the result of poorly designed applications and runaway transactions. It would be unreasonable to grow lock memory in an unbounded fashion, potentially penalizing numerous clients executing transactions due to the unreasonable behaviour of a small number of client transactions. Therefore, we have designed DB2 with a reasonable upper bound on lock memory. This upper bound was designed based on a causal analysis of past customer requirements rather than through theoretical analysis.

Lock memory will be kept at no smaller than minLockMemory:

\[
\text{MAX}(2MB, 500*\text{locksize}*\text{num_applications})
\]

Where num_applications is the number of application connections to the database and locksize is the size in bytes of a single lock structure. In other words, the larger of 500 lock structures per application connection, or 2MB. This minimum value will be evaluated by the STMM controller at each tuning interval (generally between 0.5min and 10min).

The maximum possible self-tuned value for lock memory is constrained to be no larger than maxLockMemory, defined as:

\[
0.20 * \text{databaseMemory}
\]
This 20% constraint is based around our informal observations, mentioned earlier, that most database systems use less than 10% of database system memory for lock structures, and in some cases this can grow as high as 20%.

When consuming memory from database overflow memory lock memory will be constrained to a percentage \( C_1 \) of the available database overflow memory, specifically \( LMO_{\text{max}} \):

\[
C_1 \times (\text{database overflow memory})
\]

…where database overflow memory is the overflow memory available in the database including plus any lock memory currently allocated from the database overflow memory. More specifically, if LMO represents the lock memory allocated from database overflow memory at the time of the calculation, then \( LMO_{\text{max}} \) is the maximum allowable size of LMO, is defined as:

\[
C_1 \times (\text{database memory} - \sum \text{heapsizes} + \text{LMO})
\]

\( C_1 \) is chosen as a number less than 100% so that lock memory cannot consume all of the available database overflow memory which represents the last available memory reserve for the database. Since the database overflow memory may be needed for other purposes at any moment, we considered it unwise (risky) to allow the lock memory to consume the entire database overflow memory, even momentarily. In practice we have used a value of 0.65 for \( C_1 \). This value is admittedly somewhat arbitrary, but it allows the lock memory to consume the majority of the database overflow memory, while leaving a moderate portion available for other purposes if required.

The consequence of these two constraints \( \text{maxLockMemory} \) and \( LMO_{\text{max}} \), is that lock memory will not be allowed to grow beyond a predefined percentage of database shared memory, and will no be allowed to consume more than a fixed percentage of free space within the overflow area.

All increments and decrements to the lock memory will be performed in integral units of lock memory blocks, which are 128KB, consistent with the lock allocation scheme described in section 2.1 above.

### 3.3 Lock memory combined synchronous and asynchronous self-tuning growth algorithm

Real-time growth of lock memory is critical in order to avoid lock escalation. To achieve this DB2 uses a combined synchronous and asynchronous algorithm.

In this design we distinguish between the actual allocation of memory to the locking structures within the system, and the value that is stored on disk as part of the system configuration definition. The on-disk configuration will be denoted by LMOC (for Lock Memory On-disk Configuration). The in-memory allocation is allowed to grow beyond the LMOC as a transient effect to support sudden growth requirements needed to avoid lock escalation.

When lock memory is constrained at each tuning interval STMM will attempt to configure both the lock memory and LMOC so that a set amount of the lock memory is free for use. This free amount will be in the range between \( \text{minFreeLockMemory} \) and \( \text{maxFreeLockMemory} \) respectively. The value for \( \text{minFreeLockMemory} \) is:

\[
50\% \quad \text{(i.e. 50\% of LMOC)}
\]

\( \text{maxFreeLockMemory} \) is defined slightly higher as:

\[
60\% \quad \text{(i.e. 60\% of LMOC)}
\]

During normal operations, with lock memory size being nominally maintained between \( \text{minFreeLockMemory} \) and \( \text{maxFreeLockMemory} \) free (a small spread between the two values avoids constant modification of the lock memory). By keeping the lock memory roughly half free the system can absorb as much as 100% growth in lock structure requests without requiring a synchronous increase in the lock memory size. However, if lock requests grow beyond 100% during a single tuning interval and lock memory becomes constrained (i.e. 100% used) lock manager will allocate new lock structure blocks to the lock structure chain from the database overflow memory, as required, up to the constraints described above for lock memory maximum. The consumption of memory from the overflow memory is only a transient effect until the subsequent STMM tuning interval. At the next STMM tuning interval the memory controller will adjust memory distributions to return overflow memory to its allocation goal (i.e. by reducing other memory heaps), and resize the lock memory and the LMOC.

Asynchronous adjustment to the lock memory is performed by STMM at each tuning interval. The lock manager defines the new goal for the lock memory as \( \text{targetSize} \), which is communicated to the STMM controller. \( \text{targetSize} \) is defined to satisfy the \( \text{minFreeLockMemory} \) objective. However, in the case where the new \( \text{targetSize} \) falls between \( \text{minFreeLockMemory} \) and \( \text{maxFreeLockMemory} \) then \( \text{targetSize} \) is defined as the \( \text{targetSize} \) from the previous STMM tuning interval so that no change will be made in the lock memory allocation levels.

STMM defines a goal for the size of database overflow memory, so that a moderate but small amount of memory is usually available for any purpose by memory consumers in the system (not limited to lock memory). At each tuning interval, if the overflow memory has decreased in size because some heaps, such as lock memory, have grown into it during the last interval, STMM will reduce the memory consumption of the heaps it controls in order to increase the overflow memory towards its goal. While overflow memory can become reduced/constrained
The decrement rate of the lock memory is called constant adjustment of the lock memory size. This avoids escalations. This is a rare but real scenario.

3.4 Lock memory asynchronous shrinking

The motivation for shrinking lock memory comes from the recognition that occasional batch processing of updates, inserts and deletes (rollout), poor access plans, occasional statements from poorly designed applications, and high concurrency times can lead to a time limited need for a very large number of locks that are not required during other operational periods. Because these peak pressures on lock memory may only be short lived (e.g. a few hours per week or per month) it is worthwhile not to permanently reserve the memory for lock memory. The lock memory is only reduced when there are more than maxFreeLockMemory free in the lock structure allocation chain (implying that the lock memory is grossly underutilized).

For lock memory we suggest a slow asynchronous reduction policy as follows:

- On each STMM tuning interval (generally between 0.5 min. and 10 min.), a small percentage of the entirely free blocks (i.e. the chain of 128KB block used to allocate lock structures) on the lock structure chain will be freed back to other consumers tuned by STMM or back to databaseMemory if overflow memory is below it’s objective, down to a minimum of maxFreeLockMemory.
- The decrement rate of the lock memory is called \( \delta_{\text{reduce}} \) and is defined as 5% of the current lock memory size rounded to the nearest number of 128KB blocks.

Note that there is a spread between minFreeLockMemory and maxFreeLockMemory between which STMM will not adjust either the lock memory or LMOC. This avoids constant adjustment of the lock memory size.

3.5 Tuning client constraints of lock memory use

The goal of the self-tuning lock memory algorithm is to avoid lock escalation at all times by adjusting the lock memory. Therefore we wish to allow a single application to hold a reasonably high percentage of the lock memory if that will avoid escalations in the system. At the same time we want the system to be well behaved in the case of multiple heavy lock consumers. We propose a runtime-adaptive model for lockPercentPerApplication which is initially hardly unconstrained (98%), but which begins to throttle as the lock memory approaches its maximum (20% of database memory, or has grown to LMO max percentage of overflow memory). The closer the lock memory grows to its maximum, the lower lockPercentPerApplication is set, dropping down to 1 when lock memory is 100% of its maximum size.

The adaptive model of lockPercentPerApplication uses a continuous function to provide smooth behaviour. The curve is calculated using:

\[
P(1-(\frac{X}{100})^{\frac{1}{\text{lockPercentPerApplication}}})
\]

where \( P \) is the maximum percentage of the lock memory that a transaction can consume when lock memory is not near its maximum size. This form was chosen because it provides very large value for lockPercentPerApplication while memory is ample, and aggressive attenuation when lock memory is more than 75% used.

lockPercentPerApplication will be re-computed every time the lock memory is resized, or every refreshPeriodForAppPercent requests for new lock structures by an application (each request typically retrieves several dozen lock structures). The current in-memory value of lockPercentPerApplication can change rapidly, though it is externalized in the configuration parameter at STMM tuning intervals. The refresh period for lockPercentPerApplication, namely refreshPeriodForAppPercent is defined so that the lockPercentPerApplication is recalculated after the number of the lock requests is equivalent to the number of lock structures that can be stored in the in-memory blocks described in section 2.1. As a result lockPercentPerApplication is recalculated roughly on the same interval that memory blocks can possibly be allocated.

3.6 Stabilizing the impact on SQL access plan selection

The DB2 SQL query optimizer uses information about the available lock memory when selection query execution
plans for statements. With self-tuning on the lock memory parameters are fluid, and can change during the course of statement execution (either on behalf of the statement or on behalf of other activity on the system). This would pose a serious problem since the query optimizer may compile an SQL statement at a moment in time when both lock memory and lockPercentPerApplication happen to be low, and therefore construct a query execution plan that assumes lock escalation. The resulting access plan would pre-empt the self-tuning lock memory from having an opportunity at runtime to avoid escalation, since the coarser locking scheme is already defined in the runtime access plan of the statement.

This problem is resolved by observing that the SQL query optimizer is essentially only interested in a specification of how much memory is available for locks for the application. Therefore, the values for lockPercentPerApplication and lock memory exposed to the query optimizer should not be the instantaneous values of lockPercentPerApplication and allocated lock memory, but rather a reasonable approximation of how much lock memory can be dynamically allocated if required by a statement (or set of statements).

For simplicity we approximate the SQL query optimizer’s view of lock memory, sqlCompilerLockMem as 10% of total database memory, databaseMemory. While this approximation is crude, testing has shown it appears to be adequate, providing the query optimizer a stable and reasonably large estimation of how much lock memory may be available for the statement being compiled. If the estimate is excessively large, escalation will occur at runtime which would have been unavoidable regardless of the locking chosen at query compilation time.

### 3.7 Summary of modelling parameters

The following table summarizes the key parameters used in the algorithm:

<table>
<thead>
<tr>
<th>Param.</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>databaseMemory</td>
<td>Total shared memory allocated to the database.</td>
<td>N/A</td>
</tr>
<tr>
<td>minLockMemory</td>
<td>Smallest value for lock memory that STMM will allow.</td>
<td>MAX(2MB, 500 * locksize * num_applications)</td>
</tr>
<tr>
<td>maxLockMemory</td>
<td>Largest value for lock memory that STMM will allow.</td>
<td>(0.20 * databaseMemory)</td>
</tr>
<tr>
<td>sqlCompilerLockMem</td>
<td>The SQL query compiler’s view of the amount of lock memory available.</td>
<td>(0.10 * databaseMemory)</td>
</tr>
<tr>
<td>LMOmax</td>
<td>Maximum amount of overflow memory that can be consumed for lock memory.</td>
<td>65% of database overflow memory</td>
</tr>
<tr>
<td>maxFreeLockMemory</td>
<td>Maximum % of the lock memory that can be unused before asynchronous shrinking is required.</td>
<td>60%</td>
</tr>
<tr>
<td>minFreeLockMemory</td>
<td>Minimum % of the lock memory that can be free before asynchronous lock memory growth is required.</td>
<td>50%</td>
</tr>
<tr>
<td>lockPercentPerApplication</td>
<td>Representing the % of total lock memory a single application may consume. Once lock memory has reached its maximum. Exceeding this constraint results in lock escalation.</td>
<td>98(1 − (x / 100)³), where x is % of maxLockMemory that is currently used</td>
</tr>
<tr>
<td>refreshPeriodForAppPercent</td>
<td>The refresh period for lockPercentPerApplication to be re-computed.</td>
<td>0x80</td>
</tr>
</tbody>
</table>

**Table 1. Key parameters**

### 4. A descriptive example of lock memory tuning

The following example will illustrate some of the main ideas in the lock memory resizing mechanism.

![Figure 6. Examples of combined synchronous & asynchronous lock memory tuning](image)

Each bar in the graph represents the state of memory at a point in time. At time T0 the system is in steady state, with 4% of the memory allocated to lock memory, half of which is unused (because in this example we are assuming minFreeLockMemory is 50%).

At time T1, a surge in lock requests increases the lock structure consumption from 2% to 3%, which is contained within the 4% of memory already allocated to lock memory.
memory. No overflow memory is required to satisfy the new lock structure requests.

At time T2 the next STMM tuning interval begins. STMM increases lock memory to achieve the \( \text{minFreeLockMemory} \) free objective (50% of the lock structures, in this example) making decreases in sort memory (the least needy consumer) to achieve this without consuming overflow memory.

At time T3 there is a 267% surge in the lock structure requests, increasing lock structure consumption from 3% to 8%. A large part of this surge is contained by the free space in the lock memory (6%). The additional required 2% of lock memory is allocated synchronously from overflow memory, synchronously reducing overflow memory from 10% to 8%.

At time T4, the next STMM tuning interval begins. Memory heaps (bufferpools and sort) are reduced in order to meet the lock memory objective of \( \text{minFreeLockMemory} \) free lock structures, and to reclaim the overflow memory allocation objective (10% in this example).

At time T5 pressure on lock structures has returned to the previous steady state level that was seen at T0. Most of the lock memory is now empty (87.5% empty). A slow incremental freeing of lock memory will begin at the next STMM tuning interval. The freed memory is given to the neediest heaps, based on the usual STMM allocation algorithm.

At time T6 the next STMM tuning interval begins. STMM detects that lock memory is more than \( \text{maxFreeLockMemory} \) empty, and reduces the lock memory size by \( \delta_{\text{reduce}} \) (5% in this example). The slow incremental reduction continues at subsequent STMM tuning intervals T7 and T8, etc, until lock memory size reaches its goal of \( \text{maxFreeLockMemory} \) free, at time Tn. The memory freed from lock memory is given to the most beneficial heaps, as usual.

5. Experimental results

The following four experiments test the effectiveness of the algorithm under steady state and varying workloads for increasing and decreasing lock memory demands.

These experiments were run on an IBM P660 6M1 4-way server with 4 x 750 MHz Power 3 CPUs. The server storage included 96 x 18GB SSA disks, using RAID5. The operating system was AIX 5.2F with 64bit kernel. The databases used a combined TPCC and TPCH schema in a single database; each schema included 25GB of raw (pre-load) data. The database memory was configured to use a maximum configuration of 5.11GB for all memory heaps, including lock memory. For all of these experiments the STMM tuning interval, the time interval on which asynchronous modification to lock memory will be made, was fixed at 30 seconds.

5.1 Impact of lock escalation

The following experiment demonstrates how a constraint of lock memory leads to lock escalation which can have a catastrophic impact on system concurrency and therefore on system throughput. We begin with a database configured with a static allocation for lock memory that is inadequate for the application workload. Lock memory is defined as 0.4MB for a 130 client system running an OLTP workload. As the system ramps up in the first several seconds of execution the lock requests also rise, eventually leading to lock escalations. As shown in Figure 7, the escalation results in a reduction of the lock memory requirements.

However, as seen in Figure 8 the lock escalations result in a severe detrimental impact on concurrency. Following escalation only a small number of the 130 application clients are able to make forward progress and the system throughput drops practically to zero. This catastrophe is unfortunately not-uncommon in the absence of self-tuning code, and emphasizes the need for self-tuning lock memory.

5.2 Tuning responsiveness

In this experiment we perform a simple test of the self-tuning lock algorithm by starting with a minimal
configuration for lock memory and ramping up an OLTP workload. The workload rises quickly from 1 to 130 application clients all performing OLTP transactions. Figure 9 shows the resulting system throughput and lock memory allocation combined in the same graph. As the client pressure increases the transaction throughput also increases, as does the requirement for row locks. The self-tuning lock memory adapts immediately to a stable allocation level. Very significantly, no lock escalations were observed during the experiment despite the drastic increase in clients from 0 to 130, and the resulting increase in lock memory by 10.5x.

Figure 9. Rapid lock memory adaptation to steady-state OLTP load

In the next experiment we evaluate the ability of the self-tuning lock memory to adapt to a workload surge in lock requests. An OLTP workload is executed with 50 clients and run in steady state for 25 minutes. After 25 minutes, the workload is switched to 130 clients which results in higher throughput, but also demands increased row lock requirement on the system. Figure 10 shows that the increase in lock memory is practically instantaneous, as the lock memory increases to just more than double its previous allocation at the 25 minute mark. Throughout this experiment no lock escalations occur.

Figure 10. Lock memory with 2.6x workload surge

5.3 Drastic workload variation

In this experiment a reporting query is introduced into a steady state OLTP workload as shown in Figure 11, which runs initially for 5.5 minutes. Through this period the lock memory is self tuned to a mere 2048 pages, or 8MB, representing 0.15% of the database system memory. After running in steady state for 5.5 minutes a reporting query is introduced into the system which has high requirements on locking, CPU and I/O. Over the first 25 seconds following the introduction of the reporting query the lock memory grows by 60x. At its peak the lock memory rises to over 500MB, roughly 10% of the database memory. The introduction of a single reporting query with massive row locking requirements did cause a reduction in the throughput of the ongoing OLTP workload. However, deeper analysis showed that the reduction in performance was entirely due to increased competition for CPU and disk controller bandwidth as would be expected from the introduction of significant new work on the server. No exclusive lock escalations were observed throughout the experiment.

Figure 11. Lock memory adaptation for OLTP system with sudden injection of DSS queries

The ability for the algorithm to adapt lockPercentPerApplication has critical importance in this experiment. Despite the fact that the system includes 131 active connected applications, clearly the decision support query is the single largest consumer of lock memory by far. The adaptive algorithm for lockPercentPerApplication allows even a single user to dominate lock memory consumption provided total lock memory on the system is significantly far from the allowable maximum maxLockMemory. Had two or more heavy lock consumers (queries or updates) been simultaneously introduced the adaptive algorithm for lockPercentPerApplication would have attenuated the percentage of total lock memory that each query would be allowed to consume as global lock memory began to approach maxLockMemory, per the lockPercentPerApplication formula in Table 1. Had the lock manager used a fixed number of row locks (measured in thousands) or a fixed value for lockPercentPerApplication such as 10% (the previous default value used by DB2 in past product releases) to trigger lock escalation lock escalations would occurred in this experiment, grinding the OLTP workload to a halt.
5.4 Lock memory reduction

This experiment validates the ability of the self-tuning lock manager to release memory back to the RDBMS for other operational needs. An OLTP workload is run in steady state for 1500 sec. using 130 clients, and 4.2MB of memory for locking. Suddenly the database load is reduced to 30 clients, a reduction of 76.9%. Since each client performs transactions that acquire locks, the drop in application clients results in a significant decrease in the locking requirements on the system. With much fewer locks in use than the allocated memory, the self-tuning system proceeds to gradually reduce the allocated memory for locks, reducing roughly 5% per STMM tuning interval following the $\delta_{reduce}$ parameter described in section 3.4 above. Figure 12 shows the variation in lock memory allocation. After a gradual consistent reduction over 10 STMM tuning intervals, the lock memory settles into a new steady state allocation approximately half of its earlier steady-state allocation.

![Figure 12. Gradual lock memory reduction](image)

6. Conclusions & future work

6.1 Future work

The effectiveness of self-tuning lock memory can be improved by exploring the following enhancements:

- Learning in query optimization to better estimate locking decisions that are made at query optimization time.
- The introduction of application policies to bias when lock escalations are a preferred strategy over lock memory growth. Selective lock escalation would reduce memory requirements for locking providing more memory for caching and sorting etc.

6.2 Conclusions

In this paper we have described one of DB2’s self-managing features, adaptive lock memory tuning. We have shown experimentally the feature has excellent adaptive qualities both in terms of real-time responsiveness to system surge, and in terms of gradual relaxation of memory over time. The DB2 algorithm self-tunes lock memory allowing it to grow rapidly if needed, and gradually decrease if possible. The algorithm also adapts the percentage of available memory that individual client can consume enabling aggressive consumption of the lock memory when only a small number of clients are heavy lock users, and gradually adapting to a more conservative strategy when several clients have significant lock memory needs. The algorithm was shown to work well in avoiding lock escalation in constant workload, as well as in mixed OLTP and DSS workload and under rapidly changing lock requirements. The tuning model for lock memory described in this paper has been implemented in DB2 9.

Acknowledgements

The authors would like to acknowledge Sarah Posner, Roger Zheng, Wojciech Kuczyński, Berni Schiefer, Matthew Huras from IBM Canada for their help in writing and editing this paper.

References


Trademarks: AIX, DB2, DB2 Universal Database, IBM, and products are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries. SQL Server is a registered trademark of Microsoft Corp. Oracle is a registered trademark of Oracle Corp. Other company, product, and service names may be trademarks or service marks of others.