More Mysteries of the IBM MQ Distributed Logger

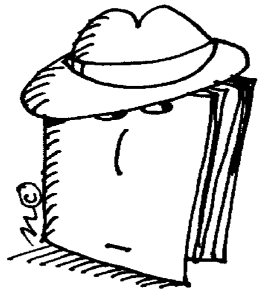
Christopher Frank

IBM Hybrid Cloud Integration

Agenda

- What’s the Mystery?
- Logging 101
- Decisions, Decisions
- The Great Debate
- Numbers and Sizes
- Log Write Integrity!
- Log Operation
- AMQ7469!
- Behind the Curtain
- What's New
- Wrap-Up

What’s the Mystery?



- **The Recovery Log is one of those things in MQ that’s “just there”**
- ***Why* it’s there is no mystery, really**
  - Enables MQ to recover data after a failure
- ***How* it does what it does – That can be somewhat mysterious**
  - Resulting sometimes in misconceptions
  - Can be overlooked when performance suffers
  - Misconfiguration possible if how it works is not understood
- **Questions abound!**
  - When is it running well? When is it not? How can I tell?
  - If running poorly, how can I tell why? And how can I make it better?
  - How can the total log size be smaller then the number of queued persistent messages?
  - Why do I sometimes run out of log space even when the system is lightly loaded?

What's the Mystery? - Notes

N O T E S

- Most if not all MQ admins know that MQ has this thing called a Recovery Log. And most if not all know *why* it is there – to enable MQ persistent data and state information following a failure. But *how* it does what it does – that tends to be more mysterious, as the logger tends to be viewed as a black box that just does what it does with little help or monitoring needed.
- But like all black boxes, the fact that you cannot easily look inside makes it mysterious and can lead to misconceptions about how it operates. It can also be easily overlooked when performance issues are encountered. And if it is suspect when throughput is suffering, it can be difficult to tell if the problem is with the logger or with the underlying OS, storage, virtualization or containerization layer, network, etc.
- As the systems hosting a queue manager become more complex, with network-attached storage, virtualization, containerization, and so on, it becomes much more important to understand how the logger does what it does, and how to tell if things have gone wrong. When is the logger running well, and how can I tell? If it is not running well, how can I tell that? What can I do to make things better? Other questions arise as well, such as: How can the total log size be smaller then the number of queued persistent messages? Why do I sometimes run out of log space even when the system is lightly loaded? It's when questions like these arise that the more mysterious nature of the Recovery Log comes to the fore.
- In this presentation we'll explore some of these "Mysteries". What do the logging parameters really mean? What considerations do I use when deciding what values to use when setting them? How do I tell when a throughput problem might be due to the logger running suboptimally? And how can I look outside of MQ to determine what might be causing poor logger performance, and what can I do about it?

What's This Presentation About?

- **In this presentation we'll explore some of these mysteries**
  - What do the logging parameters really mean?
  - What considerations do I use when deciding how to set them?
  - How to tell when the logger is behaving suboptimally
  - And how to tell when to look outside of MQ to improve logger performance
- **Some misconceptions I've heard about the Recovery Log include:**
  - When putting a single message it's faster if the put is outside syncpoint
  - Rollbacks (MQBACK calls) cause the log to be read backwards, hurting performance
  - When checkpoints are taken all data is moved from queues to the log
  - Logs must be large enough to hold all messages that might possibly be queued at one time
  - When persistent messages are gotten (MQGet) they are physically deleted from the log file
- **We'll speak to some to these misconceptions because:**
  - Knowing how transactions are actually rolled back is important
  - Understanding what happens when a checkpoint is taken is important
  - Knowing how to properly size the log is important
  - Understanding when the log is written to and read from is useful

What's This Presentation About? - Notes

N

O

T

E

S

- Over time I've heard a number of misconceptions of how MQ logging works. Some of these include things like:
  - Syncpoints are unnecessary, or even a bad thing, if used when putting a single message.
  - Rollbacks (MQBACK calls) cause the log to be read backwards, so you should avoid this
  - At checkpoint time all data is moved from queues to the log
  - Logs must be large enough to hold all messages that might possibly be queued at one time
  - When persistent messages are gotten (MQGet) they are deleted from the log file
- While misconceptions, it is understandable why people might make assumptions like these. One goal of this presentation is to clarify when and how applications interact with the log, how to properly size the log, etc, and get the most out of your queue managers.

How is the Log Used by MQ?

- **The Recovery Log captures all the information needed by MQ to recover from a failure**
- **What does this include? Things like:**
  - Persistent message updates (usually!)
  - Changes to transaction states
  - When MQ objects are created and deleted
  - Changes to MQ object attributes
  - Various channel activities
- **However, the MQ logger does not exist in a vacuum**
  - It relies on the underlying file system
  - If this is unreliable, information, including recovery information, can still be lost

How is the Log Used by MQ? - Notes

N

O

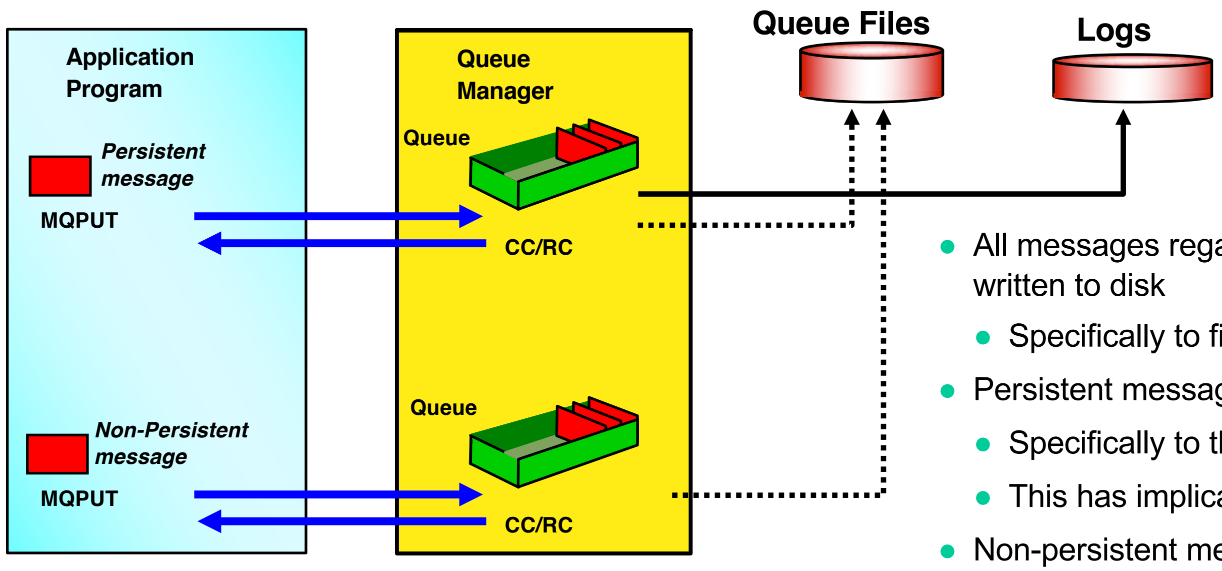
T

E

S

- The purpose of the Recovery Log is to capture and persist all the information needed by a queue manager to enable it to recover from a failure. First and foremost this includes persistent messages data. Persistent messages are always written to the recovery log, except in one unique circumstance – when it is put outside syncpoint to a waiting getter that is also not using syncpoint.
- But other information is written to the recovery Log as well, including things like changes to transaction states (Begin, Commit, Rollback, etc), when MQ objects are created and deleted, when the attributes of MQ objects are altered, as well as various channel activities.
- So following a failure, the log contains the information needed to restore recoverable updates to MQ queues. But it is important to keep in mind that the MQ logger does not exist in and of itself - it must rely heavily on the underlying file system that holds the recovery log – and if the file system (or components that make it up) are not reliable, information, including recovery information, can still be lost.

MQ 101 - How are Messages Persisted?



- All messages regardless of persistence *may* potentially be written to disk
  - Specifically to files that back the queues
- Persistent messages are *always* written to disk
  - Specifically to the Recovery log
  - This has implications for performance
- Non-persistent messages can “persist” as well:
  - NPMCLASS=HIGH on queue definition
  - Still never written to Recovery Log!
    - Even if put under syncpoint
    - Thus are not “recoverable”

MQ 101 - How are Messages Persisted? - Notes

N O T E S

- In MQ, all message data may be written to the file system that backs the queues. The queue manager will attempt to avoid this by holding the messages in buffers, only spilling to disk if the queue buffers fill or when messages sit on the queues for a long period of time.
- There is a core difference in MQ between how persistent messages are handled verses non-persistent messages:
  - Persistent messages will **always** be written to the Recovery Log, except in one very special case (as mentioned earlier). Once hardened to the log, messages should **never** be lost or discarded by MQ, unless expiry was specified on the message (or imposed on the message using capped expiry), or…if the application fumbles them, by, for example, getting persistent messages outside syncpoint.
  - Non-persistent messages will **never** be written to the Recovery Log – thus they can (and will) be lost or discarded, even in non-failure situations.
  - Using NPMCLASS=HIGH on a queue definition can allow non-persistent messages to persist across a normal restart, but the messages are still not written to the transaction log and so are not recoverable in a failure situation.
  - This is true even if the non-persistent messages are put under syncpoint – even if the UOW includes both persistent and non-persistent messages. All transaction state for non-persistent messages is held in memory – never written to the recovery log. So including non-persistent messages in a transaction will not result in any log I/O – and thus will not be recovered after a failure.
- Messages are always put to a **local** queue. Messages destined for remote queues are therefore written to a local intermediate queue (called a transmission queue).
- Messages sent by a connected queue manager that cannot be delivered to the target queue are written to a special queue called the Dead Letter Queue (DLQ), or an application-level equivalent such as a backout queue.

So to set the stage…

- **The Recovery Log is essential for persisting messages**
  - And thus enabling recovery following a failure
- **But nothing is free…**
  - Logging does have a cost
- **You want to keep the cost to a minimum**
  - But it can be hard to tell when the Logger is “happy”
  - …or when it’s unhappy, for that matter!
  - But it’s very important to understand how best to configure the logger
    - …and how to tell if the configuration is not meeting your objectives
- **MQ Logger essential for persisting messages**
  - But don’t take it for granted
  - Understand what the logging parameters really mean
  - Know what considerations are needed when deciding how to set them
  - Know how to tell when the logger itself is behaving suboptimally
  - And know how to tell when to look outside of MQ to improve logger performance

What’s the Mystery? - Notes

N O T E S

- So it’s really no mystery why the Recovery Log is there – it is essential for persisting messages across restarts and in failure situations.
- But nothing is free…Logging does have a cost, and you want to keep the cost to a minimum. Unfortunately, it can be difficult to tell when the Logger is “happy”…or for that matter, when it’s unhappy, because often when the logger is unhappy, it will not manifest itself in an obvious way. For example, your persistent message throughput may be degraded, but if the problem is related to an issue with the logger it may not be obvious. And MQ historically has not told you very much about its operation (although this is changing).
- So this presentation aims to answer these questions and other questions that arise regarding the MQ logger. Understanding its purpose, what it does and how it does it, and how you can understand and influence its operation, is something that every MQ admin should know. We’ll explore these and other mysteries associated with this critical component of MQ, including
  - What do the logging parameters really mean? What happens when I change this or that parameter? How does changing one affect others, and affect performance?
  - How does one tell when the logger is behaving suboptimally? How does this manifest itself? And how can I tell when an apparent problem with logger performance is outside of the logger itself, and what can I do about it in such a case?

Logging 101…

What Does the Log Record?

- **For each persistent update a log record is written**
  - The log record describes the update
  - Messages, message state, transaction state, queue manager status, etc

- **Log files are written to sequentially**
  - Sequential I/O is much quicker than random
  - Single point of writing rather than to individual queue files

- **A Write-Ahead Strategy is used**
  - Meaning the log is always more up-to-date than the actual data
  - Log and actual data are reconciled during strmqm

- **Log and actual data are reconciled during Queue Manager start-up**
  - Progress information displayed
  - Periodic reconciliation is done using checkpoints

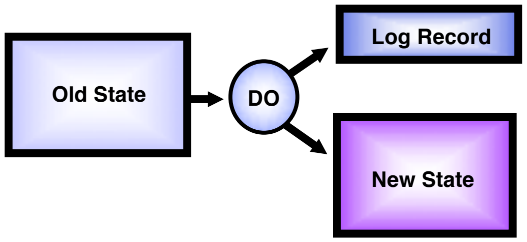
What Does the Log Record? - Notes

N O T E S

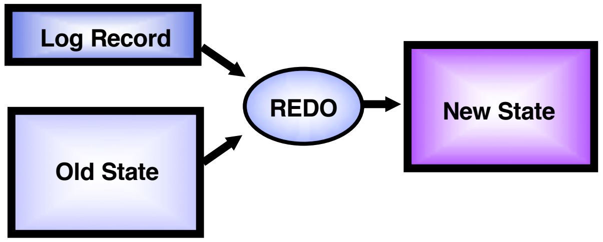
- Each update to persistent data is written to disk at least once – first to the recovery log (which always happens) and (potentially) to the file that backs the queue. This may sound like needless overhead but there is good reason for using a recovery log:
  - Log records are always appended at the end of the log, rather than to various locations as is the case with the queue files, This means the data is written in a continuous stream, minimizing disk head movement when writing to the log. A continuous stream of log data speeds up both logging itself as well as recovery, by minimizing disk head movement.
  - Writing to a sequential file makes it much easier to deal with things like power failures.
  - The log makes it easy to keep track of the operations which make up transactions.
- Message operations under syncpoint do not result in synchronous I/O until commit or rollback.
  - This slight-of-hand minimizes physical I/O without compromising recovery.
    - This I/O optimization is a major reason why persistent messages should always be put under syncpoint!
  - The log record(s) describe updates in enough detail for the update to be recreated.
  - The log records are written using a strategy called Write-Ahead Logging.
    - The log record describing an operation is guaranteed to arrive on disk before the data being updated.
    - The log is never less up-to-date than the actual data.
    - The contents of the log records can be used to perform the updates on the real data.
- Checkpoints are used periodically to keep the log and queue files in step. This eliminates the need for the log to hold all the messages that might be queued at one time.

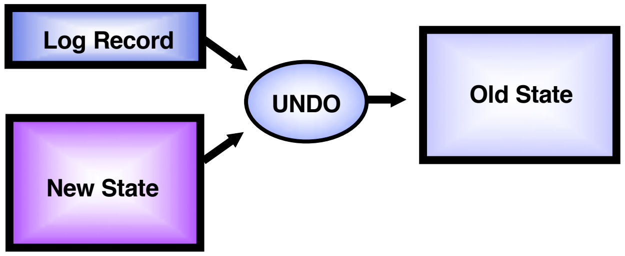
Log Processing – DO, REDO and UNDO

Normal processing



Recovery processing





DO, UNDO and REDO - Notes

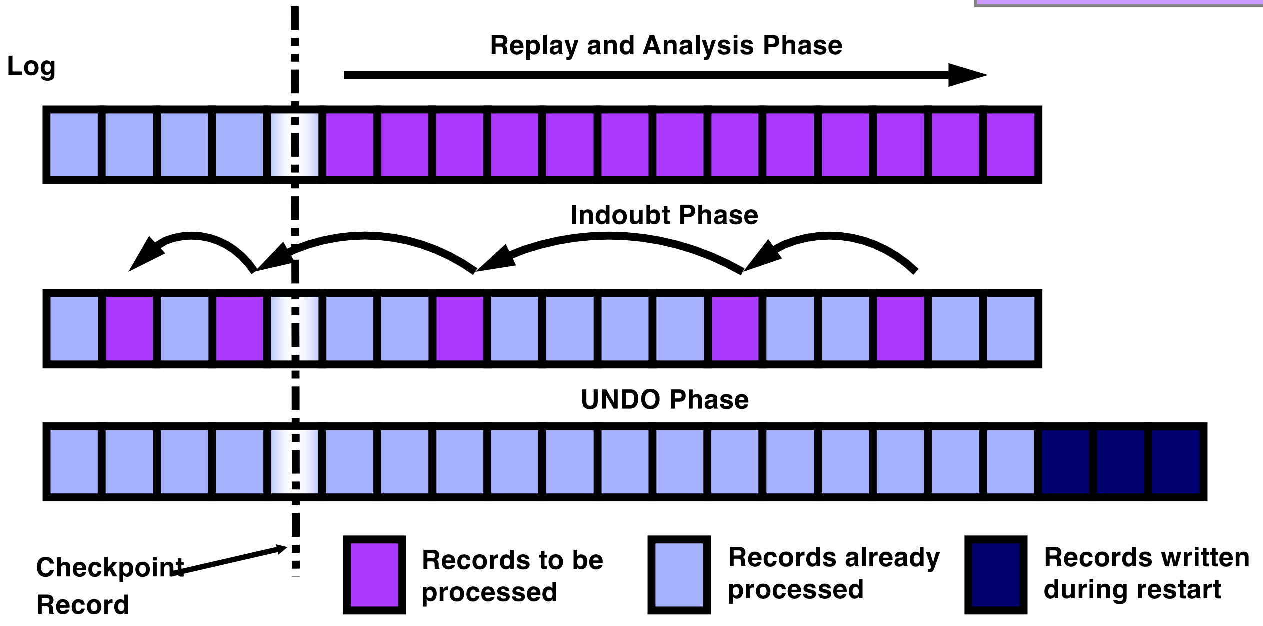
N
O
T
E
S

- All operations on recoverable data are split into three operations – DO, UNDO and REDO:
  - During normal operation:
    - **DO**: While the queue manager is running, each recoverable operation is performed and an associated log record is generated. The log record contains an encapsulated version of the operation – it might be message data if processing an MQPut, or a notation that a message has been gotten (MQGet), or that a transaction has completed, etc. Nothing is ever physically removed from the log during normal operation – transaction rollbacks during normal operation do not cause the recovery Log to be read..
  - During recovery operations:
    - **REDO**: In a recovery scenario, resource managers may need to reapply changes which were originally made. The contents of the log record and the old copy of the resources affected by the operation can be used to recreate the updated state of the resources from the most recent checkpoint.
    - **UNDO**: Following the REDO phase there may be certain operations which need to be undone, e.g. incomplete transactions. The contents of the log record and the updated copy of the resources affected by the operation can be used to recreate the state of the resources as they were before the operations were performed.
- The DO-REDO-UNDO protocol is a commonly used and well-established approach used by many resource and transaction managers. It enables a program, separate from the original applications and with no knowledge of how those original applications work, to properly reconstruct the state of the application updates following a failure. The important point is that during restart, the log must contain all the information necessary to allow the resource to be recovered without the intervention of any code other than the resource manager. The applications do not have to be involved.

Phases of Restart Recovery

- **"Backup" Queue Managers**
  - Only do REDO

```
$ strmqm QMC
WebSphere MQ queue manager 'QMC' starting.
9 log records accessed on queue manager 'QMC' during the log replay phase.
Log replay for queue manager 'QMC' complete.
Transaction manager state recovered for queue manager 'QMC'.
WebSphere MQ queue manager 'QMC' started.
```


Log
Replay and Analysis Phase
Indoubt Phase
UNDO Phase
Checkpoint Record
Records to be processed
Records already processed
Records written during restart

Phases of Restart Recovery (1) - Notes

N O T E S

- Each time a queue manager restarts, three phases of recovery are performed to restore the queue manager to a consistent state:
  - Replay and Analysis Phase
    - All log records after the last checkpoint are REDONE. This is known as repeated history.
    - The checkpoint is the last known point of consistency between the log and the object files.
    - The transaction table is rebuilt during this phase. Any transactions mentioned in log records are added to a list as candidates for rollback in the final phase of restart. The state of all of the transactions found is also maintained during this phase. When the log records for the end of a transaction are found, the transaction can be removed from the list.
    - If the queue manager stopped cleanly (no in-flight transactions), the only processing required is to replay the most recent checkpoint. There will be no work to do in the next two phases.
  - Indoubt Phase
    - We have a list of transactions in-flight at the time that the queue manager ended. For each transaction, we scan backwards through the log from the last record the transaction wrote following the links between records in the same transaction building up a picture of what the transaction was actually doing at the time the queue manager stopped.
  - Undo Phase
    - Any transactions in the list which are not prepared are rolled back. This involves writing special undo log records called Compensation Log Records (CLRs). A CLR contains an after image only which corresponds to the before image of the log record it is undoing.
    - Once the CLR has been written, the operation which it describes will be ignored by the indoubt phase if we get into a situation where restart is interrupted and restarted at second time.

Phases of Restart Recovery (2) - Notes

N O T E S

- At the end of restart, we may have some prepared transactions. The indoubt phase will have reconstructed the list of operations making up the transaction so we can subsequently commit or rollback the transactions when called by the transaction manager.

What are Recovery Logs?

- **Log contains record of all recoverable activity in MQ**
- **Enables recovery of messages and message state *after a failure***
  - Means that transactions must be used for certainly of recoverability
  - This is true whether messages are put or got
- **Under normal conditions the log is only written to**
  - MQ will *only* read the log file:
    - During restart
    - When explicitly requested to recover a damaged object (**rcrmqobj**)

What are Recovery Logs? - Notes

N

O

T

E

S

- It's easy to think of the logs as being used to log persistent *messages*, and that is certainly true. But it's more accurate to say that the logs are used to record each *recoverable activity* in MQ. This certainly would include persistent message data, but also includes things like when persistent messages are gotten, when transactions are committed or rolled back, when checkpoints are taken, etc.
- The recovery log is not involved in message recovery during normal operation. The log only comes into play when recovering in-flight *transactions* following a queue manager failure. This means that transactions must be used for certainly of recoverability – there are points in processing where messages that are put or got outside syncpoint can be lost, even though they are persistent messages.
- Recoverable data is written to the log files sequentially. Even though a typical queue manager will have many application threads putting and getting messages in parallel, the logger is merging all this activity into a single stream of data. This means the logger can become a bottleneck (which we'll discuss in more detail), so considerable effort has been made to optimize the logger to handle very high volumes of data.
- Sequential I/O is used to maximize throughput to disk. MQ under normal conditions will never read the log, as this would mean interrupting the stream of data being written - the log is only written to during normal operation. The log will only be read in recovery situations: During queue manager restart, for example, or, when using linear logging, when explicitly requested to recover a damaged object using the **rcrmqobj** command.
- The logger was significantly enhanced in MQ V7.1 to improve its performance. Further performance enhancements were introduced in MQ V8.

What do the Recovery Logs Contain?

- **Record of Recoverable Activity**
  - Persistent messages
    - Non-persistent messages are never written to the log
  - Internal data about queue manager objects
  - Persistent channel status

- **Recovery logs consist of a collection of files**
  - Three or more files of log data
    - S0000000.LOG – S9999999.LOG
  - Log control file
    - amqhlctl.lfh (in log directory)
  - Checkpoint file
    - amqalchk.fil (in qmgr directory)

What do the Recovery Logs Contain? - Notes

N

O

T

E

S

- Any activity for which recovery might be needed following a failure is captured in the Recovery Log. This includes persistent message data certainly, but also state information about those messages – have they been committed, have they been gotten, etc. This makes it possible for the queue manager following a failure to restore things back to their last-known "good" state following a failure. The log data can also be used for purposes of "reconstituting" a queue manager elsewhere, if you are using Backup queue managers.

- It is sometimes not realized that the Recovery Log is not involved in backing out transactions that fail. If it were involved, it would be necessary to read the log backwards (remember the phases of recovery discussed previously) – but this would seriously impact performance of the queue manager as a whole. Rolling back transactions is considered part of a queue managers normal operations, and as mentioned, the Recovery Log is never read during normal operations – transaction rollbacks are accomplished using a different mechanism.

- When messages are gotten (or for that matter, when they are rolled back following a transaction failure) they are not physically removed from the Recovery Log. MQ Uses a "Write-Ahead" strategy with the log, meaning the log is always more up-to-date than the actual data. When a message is gotten, that fact is noted in the log, so that should things need to be restored later following a failure, the recovery process can tell which messages on the log can be safely ignored. This reconciliation of data takes place during strmqm

- Recovery logs consist of a collection of files
  - Three or more files of log data (S0000000.LOG – S9999999.LOG)
  - Log control file (amqhlctl.lfh in the *log* directory)
  - Checkpoint file (amqalchk.fil in the *qmgr* directory)

Decisions, Decisions…

Logging Parameters

- **A number of logging parameters at your disposal**
  - Used to control the type, size and behavior of logging
  - Some can be specified <u>only</u> at queue manager creation!
- **Use (or misuse) of these can have a significant effect on performance**
  - For good or for bad
  - And when the latter, this can be difficult to diagnose
- **Some can only be set at the time a queue manager is created**
  - Type of logging
  - Size of log files
  - Log file location
- **Those that are changeable always require a queue manager restart**
  - Number and type of log files
  - Size of log buffer

Logging Parameters - Notes

N

O

T

E

S

- There are several MQ parameters at your disposal for controlling the type of logging, the size of the log, and various aspects of the logger’s behavior.
- Choosing these values wisely is important, as the values chosen can have a significant effect on performance, for good or for bad. And in the latter case, it can sometimes be difficult to determine that poor choice was the cause of performance problems later.
- Some of these are critical for another reason - some can only be set at the time a queue manager is created. Values such as the type of logging (circular or linear) and the size and placement of log files can only be changed later deleting and recreating the queue manager with the desired values. So getting these right the first time is no small matter.
- Other logger parameters can be changed later, but be aware that even those that are changeable still require a queue manager restart. These include the *number* (not the *size*) of both primary and secondary log files, as well as the size of the log buffer.

What happens if you do nothing?

- **Defaults used if log parameters not specified on crtmqm**
  - These will work for many installations, but don't assume that they will be right for you, or that they are "ideal"
- **Default values assigned include:**
  - Once a queue manager is created, the defaults for that queue manager can be found in the *Log* stanza of the *qm.ini* file
    - *Log:*

```
LogPrimaryFiles=3
 LogSecondaryFiles=2
 LogFilePages=4096
 LogType=CIRCULAR
 LogBufferPages=0
 LogDefaultPath=/var/mqm/log
```

- **In practice this results in:**
  - A log buffer size of 2MB
  - A log file size of 16MB
  - A maximum active log of 80MB (of which you can use 64MB)
- **Very possible you will want to use different values**
  - So we'll be discussing these in detail

What happens if you do nothing? (1) - Notes

N O T E S

- If you do not specify log parameters when creating a queue manager, MQ will assign default values for these. You can consider these values as a good “start set” of parameters, and may choose to go with the defaults, initially at least. There are many customers who run in production with the default values and do just fine – especially if their load is small-to-medium and if their applications are “well behaved”.

- The default Log parameters as they will appear in the *Log* stanza of the *qm.ini* file are:
  - *Log:*

```
LogPrimaryFiles=3
LogSecondaryFiles=2
 LogFilePages=4096
 LogType=CIRCULAR
 LogBufferPages=0
LogDefaultPath=/var/mqm/log
LogWriteIntegrity=TripleWrite
```

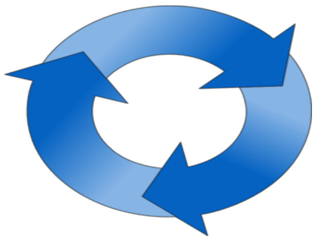
What happens if you do nothing? (2) - Notes

N O T E S

- Some of these are non-intuitive – it's not obvious what you are going to get just by looking at the parameters themselves. In practice, what you will have as a result of using the default values will be:
  - A log file size of 16MB: The value specified for *LogFilePages* is expressed as the number of 4K pages, and 4K * 4K = 16MB.
  - A maximum active log of 80MB: This is the number of ((*LogPrimaryFiles* + *LogSecondaryFiles) * LogFilePages)*
  - A log buffer size of 2MB: The value "0" in a modern queue manager results in an actual default of 512, which multiplied by the page size of 4096 results in a buffer size of 2MB.

- These are not unreasonable values, and work well for many customers. But there may be situations where these values will not suffice, and so you may want to consider choosing values that make it easier for you to increase the size of the log later without running into constraints that will force you to delete and recreate the queue managers, as well as values that might result in better performance, particularly if you know you will be running with a heavy messaging load, with applications that have long-running units-of-work, etc. For that reason we are going to be discussing these values in more detail.

The Great Debate…



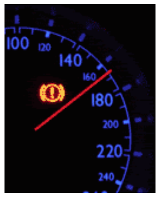
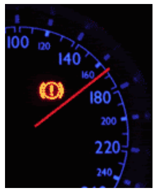
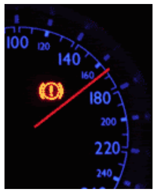
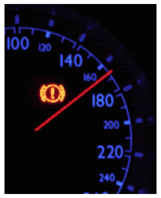
Circular

Linear

Two Types of Logging Available

Circular and Linear

- **Specified at queue manager creation**
  - Changing requires the queue manager to be deleted and recreated!
- **Each has advantages**
  - Neither is “right” or “wrong”
  - Though each has been the subject of religious wars!
- **Each involves trade-offs**
  - Neither is “free”
  - Views of each have evolved over time
  - Largely due to improvement in technology

Performance		**Circular** **Linear**
Administration		**Circular** **Linear**
Ability to archive		**Linear only**
Media recovery		**Linear only**

Two Types of Logging Available - Notes

N O T E S

- Few topics in the MQ universe can elicit a livelier discussion than the topic of circular vs linear logging. We'll be exploring both over the next few slides.
- Probably the most important thing to emphasize is that the choice you make will be with you for as long as that queue manager exists! You must choose one of the other when creating a queue manager, and once decide, it cannot be changed without deleting and recreating the queue manager.
- Neither option is "right" or "wrong" – thus the spirited debates which can result. Each has advantages, as well as disadvantages. Similarly, neither is "free" – each comes with a cost that must be weighed against the advantages each options presents.
- As will be seen, the advantages of one vs the other have evolved over time, as the underlying technology that MQ relies on has matured.

What are the Differences?

	Circular	*Linear*
➢ Recovery	Can reconcile in-flight UOWs that were incomplete	Same. And can also recover damaged queues
➢ Performance	Minimum overhead. Logs allocated once and reused	Logs must be allocated on an ongoing basis, potentially causing resource contention
➢ Admin	Basically no administrative effort required during normal operations	Administrators must ensure inactive log files deleted or archived
➢ Risk	Loss of a queue file means loss of all messages on that queue	A normally running queue manager will eventually exhaust available disk space if log files are not managed regularly

Reference: http://www.ibm.com/developerworks/websphere/techjournal/0904_mismes/0904_mismes.html

What are the Differences? - Notes

N

O

T

E

S

Recovery

- Circular logs are used to reconcile in-flight UOWs that were incomplete at the time of failure. They cannot be used to recover damaged queues or the messages they contain.
- Linear logs contain a copy of all persistent messages ever queued. So in addition to reconciling incomplete UOWs, they also support rebuilding of damaged queues.

Performance

- Circular logs are allocated once and then reused. Thus overhead is minimal. Linear logs must be allocated on an ongoing basis, impacting performance. Inactive logs must be deleted or moved, causing I/O contention.

Administrative Overhead

- With Circular logging no administrative overhead is required during normal operations. Linear logs contain a copy of all persistent messages ever queued, and so administrative action is needed to manage the log files or they will eventually fill the file system.

Operational Risk

- With Circular logging the loss of a queue file results in loss of all messages on that queue. Loss of a file system hosting the
- queue files results in loss of all messages on that queue manager. This is the single most important justification for using Linear logging. On the other hand, with linear logging a normally running queue manager will eventually exhaust available disk space if log files are not mana

Which to use?

- **Today, there really aren't many good reasons for using Linear logging**
  - Media failure? Modern RAID disk redundancy can address this
    - Media recovery can still provide some benefit in certain cases
      - e.g. Damaged objects can be recovered
      - But not always if the cause was administrative errors or a damaged log
  - Audit trail? Yes, but not a very user-friendly one
- **Linear logging imposes a cost**
  - The performance penalty sometimes is not always noticed
  - But the administrative cost usually is:
    - Media images…
      - Must be frequent enough to make recovery time acceptable after a failure
      - But not so frequently that it impacts performance
    - Archiving and/or deleting old log extents
  - These costs can negate the benefits and should be weighed carefully

Which to use? - Notes

N

O

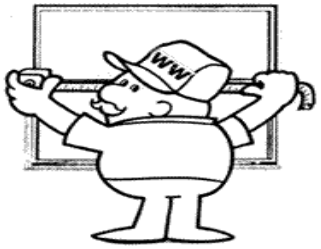
T

E

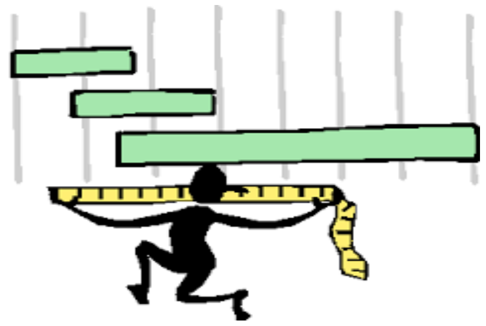
S

- Today, there really aren't many good reasons for using Linear logging. Many years ago when disks were much less reliable it was necessary for MQ to "go the extra mile" to ensure that data was not lost, but today with modern RAID disk redundancy this is much less of a concern than in the past.
- Linear logging has also sometimes been used to provide an audit trail, but the logs are designed for recovery purposes, not for audit purposes – as anyone trying to use them for that purpose can probably attest to. There are better ways today for capturing a record of transaction activity then using linear logging.
- Media recovery can still provide some benefit in certain cases – for example, damaged objects can be recovered and messages salvaged. But this does not work in all cases (such as when the "damaged" object was due to an administrative error, or when the log itself is damaged) – so this is at best a partial solution.
- At the end of the day, the choice really becomes one of balancing risk. Circular logging is generally recommended because it gives better performance and reduced administrative effort while carrying very little risk in terms of lost messages – especially given the reliability of disk storage systems today, the existence of RAID drives, etc. On the other hand, there are situations where messages can be lost due to a damaged queue that can only be recovered when using Linear logging. What degree of risk that actually represents, and whether that small additional protection warrants the additional performance and administrative cost, is a decision each customer must make. As stated earlier, there is no "right or wrong" answer.

Numbers and Sizes…



Log Buffer – How Big?



- **Memory space MQ uses to buffer log writes**
  - Not specified on the crtmqm command
    - Value specified in qm.ini (LogBufferPages)
- **Specified as number of 4KB pages:**
  - Minimum: 18 (72KB)
  - Default: 0 (which really means “512 4K pages”, or 2MB)
  - Maximum: 4096 (16MB)
- **Default is often used, but might be insufficient**
  - What are your typical message sizes?
  - What kind of volume do you expect?
  - How many concurrent transactions (esp. Putters) do you expect?
- **Bigger is usually better**
  - Significant performance penalty can result if too small
  - And if oversized…who cares?

Log Buffer – How Big? - Notes

N O T E S

- The log buffer is a location in shared memory that MQ uses to buffer log writes. Because this is a memory area it is allocated each time the queue manager is started.
  - Which means it can be changed more easily than the log file size. Although changing it does require a queue manager restart.
- Unlike other logger parameters, the log buffer size cannot be specified on the crtmqm command. To override the default you change the *LogBufferPages* parameter of the Log stanza in the qm.ini file.
- The log buffer is specified as number of 4KB pages. Values:
  - Minimum: 18 (72KB), Default: 0 (which really means "512", or 2MB), Maximum: 4096 (16MB)
- Default is often used, but might be insufficient
  - What are your typical message sizes?
  - How much concurrent activity (putters in particular) do you expect the queue manager to handle?
  - How many messages per transaction do applications typically put?
- Bigger is usually better
  - If undersized, can impose a significant performance penalty, although changes in the logger in MQ V8 reduce this penalty slightly.
  - If oversized…who cares? The cost if minimal and the potential benefit large.

Log Files – How Big?

- **Specified as a multiple of 4KB. Values:**
  - Default: 4096 (16MB)
  - Minimum: Unix and MQ Appliance: 64 (256KB) – Windows: 32 (128KB)
  - Maximum: 65535 (256MB)
- **Should give careful thought to this**
  - What are your typical message sizes?
  - What kind of volume do you expect?
  - How long do transactions typically run?
  - How much headroom needed for future growth?
- **Larger file size generally better**
  - Less frequent log file switches – slight performance gain
    - Linear – More efficient to format/archive one large file vs many smaller ones
  - Allows for larger log overall if needed in future
    - Since log file size cannot be altered after queue manager creation

Log Files – How Big? - Notes

N

O

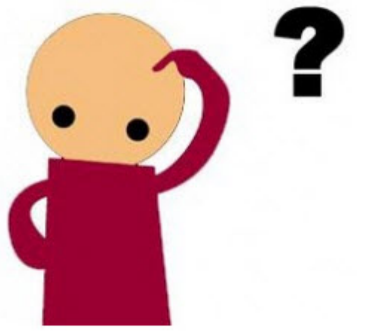
T

E

S

- The Recovery Log is really a collection of files that together comprise a single, logical log.
- File size can be specified on **crtmqm**, or in qm.ini (*LogFilePages* parameter of the Log stanza). The file size is specified as a multiple of 4KB.
  - The default log file size is 4096 (x 4K = 16MB). You can specify a smaller size if this makes sense – the minimum log file size on Unix, Linux and the MQ Appliance is 64 (x 4K = 256KB) and on Windows is 32 (x 4K = 128KB). You can also specify a larger size if needed or preferred – the maximum log file size is 65535 (x 4K = 256MB).
- What factors should you consider when deciding whether to use non-default values for the log file size? Considerations include things like what are your typical message sizes? What kind of volume do you expect? How long do transactions typically run? How much headroom needed for future growth?
- Careful thought should be given to this, given the fact that you cannot alter the size of the log files without deleting and recreating the queue manager. For this reason, it's generally better to overallocate than to underallocate. Reasons for this include:
  - There is a small delay incurred when switching log files. Larger log files mean fewer log switches.
  - If using linear logging, files need to be preformatted, and oftentimes archived as well. It's more efficient to do both of these things to a single file of 256MB than to 16 files of 16MB each.
  - A larger log file size means you can start with fewer log files initially, and increase these as needed. Increasing the number of files can be done without deleting and recreating the queue manager. So larger log files give you the potential of a much larger log without having to delete and recreate the queue manager.

Primary Log Files – How Many?



- **Specifies the initial, minimum number of log files**
  - In effect this specifies the minimum size of the *active log*
  - **-lp** <num> on **crtmqm**
- **Values:**
  - Minimum: 2, Default: 3, Maximum: 510 (Unix and MQ Appliance) or 254 (Windows)
- **Considerations for Primary log files:**
  - What are your typical message sizes?
  - What kind of volume do you expect?
  - How long do transactions typically run?
    - This is especially important!
- **Can be altered after queue manager creation**
  - *LogPrimaryFiles* parameter in qm.ini
    - But note that change will not necessarily take effect immediately

Primary Log Files – How Many? - Notes

N O T E S

- The Primary log files represent the minimum number of files that can be in the active log, meaning the active log will never be made up of fewer than this number of files. They are allocated when the queue manager is created – and the number can be specified on the **crtmqm** command.
- There must always be at least two (2) logs files in the active log. By default three (3) are allocated. The maximum number of primary log files varies by platform - for Unix, Linux and the MQ Appliance, the maximum is 510; for Windows the maximum is 254.
- You should give careful thought to this number, although it can be changed after the queue manager is created. It can be tempting to say “I’ll make it the maximum so that I’ll never have to worry about it”, but doing so might create other problems. File system space is not infinite, and having too few primary files is infinitely better than running out of file system space.
- But for best performance, you want to have the number of primary files set to a value that takes into account things like: What are your typical message sizes? What kind of volume do you expect? How long do transactions typically run?
- The active log must be able to hold all the message data associated with uncommitted transactions. In many systems that means the number of active log files is often fairly low, especially if the log file size is large. We’ll see later how you can periodically check the size of the active log and tune this number as your workload changes. The number can be changed using the *LogPrimaryFiles* parameter of Log stanza in qm.ini. Be aware that if you do change this number, it may not take effect immediately.

Secondary Log Files

- **Specifies number of "spare" log files**
  - **-ls** <num> on **crtmqm**
  - Additional log files to be created should primaries be exhausted
- **Considerations for Secondary log files:**
  - Occasional spikes in volume?
  - Long-running transactions?
    - This is especially important!
  - Headroom needed for future growth?
    - Or unanticipated spikes in volume?
- **You do take a "hit" when allocated**
  - Fortunately MQ keeps them around once allocated
- **Values:**
  - Minimum: 1, Default: 2, Maximum: 509 (Unix and MQ Appliance) or 253 (Windows)
- **Can be altered after queue manager creation**
  - *LogSecondaryFiles* parameter in qm.ini
  - Secondaries can be freed if no longer needed

Secondary Log Files – How Many? (1) - Notes

N O T E S

- Secondary log files represent spare log capacity you want in reserve should the active log need to be expanded. You specify the number of secondary log files you want when the queue manager is created – it too is an option on the **crtmqm** command.
- Secondary log files differ from primaries in that they are not allocated when the queue manager is created – they are only allocated when needed, and when no longer needed, they can be freed. Thus you must ensure the needed space is available on the file system.
- You must specify at least one (1) secondary log; by default two (2) will be assigned. The maximum number varies by platform - for Unix, Linux and the MQ Appliance the maximum is 509; for Windows the maximum is 253.
- You should give careful thought to this number, although it can be changed after the queue manager is created. It can be tempting to say "I'll make it the maximum so that I'll never have to worry about it", but doing so might not work as you expect, and might create other problems – we'll discuss why shortly.
- Conversely, you take a "hit" when a secondary file is allocated, as it must be formatted and this is not done in advance. Fortunately, once a secondary is allocated, the logger is very reluctant to release it.
- What should you consider when choosing this number? Some examples would be: Do you have occasional spikes in volume? How long do transactions typically run? How much headroom might be needed for future growth or unanticipated spikes? effect immediately.

Secondary Log Files – How Many? (2) - Notes

N O T E S

- The active log must be able to hold all the message data associated with uncommitted transactions. The availability of secondary files means you can reduce the number of primaries without needing to worry that you might run out of log space. But how to decide how many primary vs secondary files to allocate? We'll see later how you can periodically check the size of the active log and tune this number as your workload changes.

- The number can be changed using the *LogSecondaryFiles* parameter of Log stanza in qm.ini. Be aware that if you do change this number, it may not take effect immediately.

Log File Constraints

- **You cannot have as many log files as the numbers imply!**

- **There is a limit on the number of *active* log files**
  - No fewer than 3, no more than:
    - Unix/MQ Appliance: 511, Windows: 255

- **Why does MQ let you specify more log files than you can use?**
  - Allows you to decide the mix of primaries and secondary log files
  - Usually not a good idea to have a very large number of log files
  - Better to have fewer, but larger, log files
    - Better performance, room for growth

- **This maximum of 511/255 active log files is a *key constraint* for an active queue manager**
  - Max size on Windows: 64GB
  - Max size on Unix, Linux, MQ Appliance: 128GB
  - The good news is, rarely if ever should you need to come anywhere close to that size
    - But there are always exceptions

Log File Constraints - Notes

N O T E S

- It is important to keep in mind that you cannot have as many *active* log files as the previous numbers imply – you might think you could have 1019 files in the active log on Unix/Linux or the MQ Appliance (or 507 on Windows) but in actuality MQ will limit the number of active log files to less than this. The active log can never have fewer then three (3) files, and never more than 511 files on Unix/Linux and the MQ Appliance, or 255 on Windows. These numbers are unchanged in MQ V9.

- MQ lets you decide what the mix of primaries and secondaries should be. It's never a good idea to have a very large number of log files – if you need to have a large active log, better to have fewer, but larger, logs files, as it's easier to manage as well as providing better performance while allowing you more room for growth.

- This maximum of 511 active log files (255 on Windows) is a *key constraint* for an active queue manager; it means that on Windows, the largest active log in total cannot exceed 64GB, and on Unix/Linux and the MQ Appliance the largest active log in total cannot exceed 128GB. The good news is, it should never have to come anywhere close to that size except in very unusual circumstances.

- What does happen if you specify too high of values for primary and secondary files?
  - Specify too high a value when creating a queue manager and either **crtmqm** will fail ("log size not valid"), or MQExplorer will warn you, then ignore it per comments above.
  - Specify too low a value and either **crtmqm** will ignore it and use the product defaults, or MQExplorer will ignore it and use the value specified in mqs.ini (or the product default).

Log Write Integrity…

Log Write Integrity

- **Another contentious logger parameter!**
  - But it really shouldn't be so
- **Represents the degree of trust (or distrust) in the underlying file system**
- **Three values:**
  - SingleWrite
  - DoubleWrite
  - TripleWrite
- **Frequently misunderstood**
  - The names probably don't help
- **Originally introduced because file systems can vary in terms of integrity**
- **Can be altered after queue manager creation**
- **We'll discuss each in detail**

Log Write Integrity - Notes

N O T E S

- LogWriteIntegrity is arguably the most contentious of all the logger parameters!
- Three options are available: SingleWrite, DoubleWrite and TripleWrite, with the latter both the default as well as the recommended value.
- The meaning of these is frequently misunderstood – due at least in part to the names chosen.
- Properly setting this should reflect the degree of trust (or distrust) you have in the underlying file system to guarantee write integrity in specific situations.
- Because file systems and the components that make them up offer different degrees of write integrity, caution must be used in specifying any value other than the default of TripleWrite.
- These values can be altered after queue manager creation
- What do these values actually mean? What is the effect of specifying one verses the other? We'll look at each of these next.

What Exactly is SingleWrite?

- **Just the name is compelling!**
- **Can occasionally provide improved performance**
  - But in practice only with low-concurrency workloads
  - And only when logger writes partial pages
    - Which since V7.1 the logger tries much harder not to do!
- **There is a huge potential downside to using SingleWrite!**
  - The underlying file system / storage device must *absolutely guarantee* that writes are atomic
  - Should a write fail for **any** reason the page being written will be one of two states:
    - Either the before image, or the after image, and nothing in between
    - With network file systems, it is very hard to guarantee this absolutely
      - As there are many components involved
- **Recommendation: Do not use SingleWrite**
  - Unless the file system / storage device can provide the above-mentioned *absolute guarantee*
  - Guarantee must apply to the entire I/O stack – not just the device
  - Must be confident that a future change to the underlying software or hardware will not negate such a guarantee

What Exactly is SingleWrite? (1) - Notes

N O T E S

- Just the term "SingleWrite" can be compelling to an MQ administrator – why would I want to write something more than once? And it is true that with MQ you can typically see a significant performance benefit from using SingleWrite in certain scenarios - in particular, scenarios where there is little to no concurrent processing going on in the queue manager.
- To illustrate: Take as an example a simple MQ application, which performs a repeating sequence of MQPUT->MQCMIT->MQGET->MQCMIT. In this example, each iteration of the application will require 2 log forces (one for each commit) and thus will incur a minimum of 2 physical writes if SingleWrite is used, but with TripleWrite 4 physical writes would be incurred. Thus the attraction of SingleWrite – because it is simple to set up a test like this, is it easy to demonstrate a non-trivial performance benefit from using SingleWrite.
- But the question really should be: How representative is this of how a typical queue manager operates in a production environment? A single thread executing a simple message sequence?
- Consider instead a more realistic production scenario: For example, 50 concurrent instances of the same application. First, it would be a better representation of a more typical production workload. But more importantly, because of the way logger attempts to optimize I/O you should expect to see minimal differences between SingleWrite and TripleWrite.
- The most important consideration, though, goes beyond performance. SingleWrite is only safe to use if the underlying file system MQ is dependent on, and all the components that comprise it can absolutely guarantee under all circumstances that 4KB pages written synchronously to the MQ recovery log are written atomically – meaning the file-system/device hosting the MQ recovery log explicitly guarantees that if a write of a 4KB page fails for any reason (regardless of the severity of the failure) the only two possible states for that 4KB page will be either the before image, or the after image, and that no intermediate state is possible.

What Exactly is SingleWrite? (2) - Notes

N

O

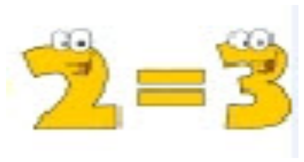
T

E

S

- To have that degree of certainty requires that you fully understand the entire I/O stack from top to bottom in some detail to safely deploy SingleWrite. In the vast majority of cases TripleWrite would be the safer option – especially since the onus would be on you to ensure that any future change to any component of the underlying file system or storage device does not invalidate this guarantee without the knowledge of the MQ administrator.
- So the recommendation is, if as the MQ administrator you are in ANY doubt that your storage infrastructure can make the required guarantee both now and in the future, in any and all circumstances, you should use TripleWrite.

What Exactly is DoubleWrite?



- **Deprecated (but can still be specified)**
  - Long, long ago, DoubleWrite was the alternative to SingleWrite
  - For when the atomicity of writing 4KB pages to the MQ recovery log could not be assured
- **Unfortunately, there was a problem with DoubleWrite**
  - A small but real potential data integrity issue with the approach was uncovered
  - So the TripleWrite algorithm was implemented to overcome that issue
- **DoubleWrite no longer serves any useful purpose**
  - If specified it's treated the same as TripleWrite (since V7.1)

What Exactly is DoubleWrite? - Notes

N O T E S

- In the early days of MQ there were only two options, SingleWrite and DoubleWrite.
- DoubleWrite was used when the atomicity of writing 4KB pages to the MQ recovery log could not be assured.
- Unfortunately, a potential data integrity issue was discovered with the approach used by the DoubleWrite algorithm, and so the TripleWrite algorithm was implemented to overcome that issue.
- DoubleWrite no longer serves any useful purpose. If specified it is (since v7.1) interpreted as TripleWrite.

What Exactly is TripleWrite?

- **The safest option to use in all cases**
  - And given that, why would you want to use any other?
- **TripleWrite does *not* mean MQ writes *all* log data three times!**
  - Only applies when the logger is forced to write a partial page
- **When writing a partial page:**
  - The logger may subsequently need to add data to the partial page
    - It could simply overwrite the page with both previous and new data
      - But if a write error occurred, the original data could be lost if the write was not atomic
  - To add data to the partial page safely, the logger will:
    - First write the new page to a different location
    - If that succeeds, it will then overwrite the original log file page with the “new” page
  - Should a failure occur the logger knows which log file page to use in a recovery situation
- **Current releases of MQ try very hard not to write partial pages**
  - So the performance cost of TripleWrite in most cases is negligible
  - But the integrity cost of not using it can be substantial

What Exactly is TripleWrite? (1) - Notes

N O T E S

- If there is any uncertainly about the file system being able to deliver the needed degree of write integrity, then TripleWrite is the safest option to use. And given that most customers that use MQ use it for the data integrity it offers, why would you want to use anything else?

- The standard answer to this is for the performance benefit, but we've already discuss that and in most cases it's a spurious justification. Why? Because TripleWrite does *not* mean MQ writes *all* log data three times! What it does is address a potential problem that may exists when writing *partial* pages. So, it applies only to the last (or only) page being written, and only if that page is not a full page. But what does TripleWrite actually do?

- Consider an example: Picture a scenario where the logger wrote 1KB of data into a log file page, so has lots of space left in the log page (which are fixed at 4KB). Thereafter, the logger needs to write more data to that log file page – perhaps it now has 3KB more data and now wants to fill the page.

- To do that, the logger will need to rewrite the page so that it contains both the original 1KB of data as well as the additional 3KB of data. But wait a minute - the logger cannot simply write over the partial page – what if a failure occurs sometime during the write? This would potentially corrupt the 1KB of data that was already out there. So, the logger will write a second page elsewhere, containing the original 1KB of data as well as the remaining 3KB of data it is appending to the page. If that succeeds, it will then write a third time, this time overwriting the original page with the "new" page. Because the logger always knows which writes were successful, it knows which page (the original page or the "elsewhere" page) should be used during the recovery step.

What Exactly is TripleWrite? (2) - Notes

N

O

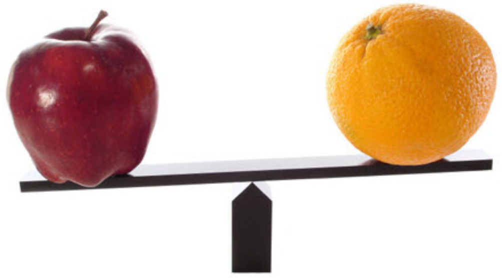
T

E

S

- In a queue manager with a high degree of concurrent activity, the need to write less than full pages to the log file should be infrequent, so this triple-writing of occasional partial pages should have little impact on performance. But most important, given that it can be difficult to reliably determine the atomicity of log file writes, and given that, even if this is reliably determined, changes to the underlying software or hardware might invalidate any such guarantee without the knowledge of the MQ administrator, the recommendation is you should use TripleWrite to ensure the integrity of log data.

Log Usage…



Side-by-Side Comparison

- **The following slides compare Circular and Linear logging in action**
- **Intent is to illustrate the following:**
  - MQ usage of Primary and Secondary log files
  - Active log files – Those needed for recovery purposes
  - Inactive log files – Those no longer needed for recovery purposes

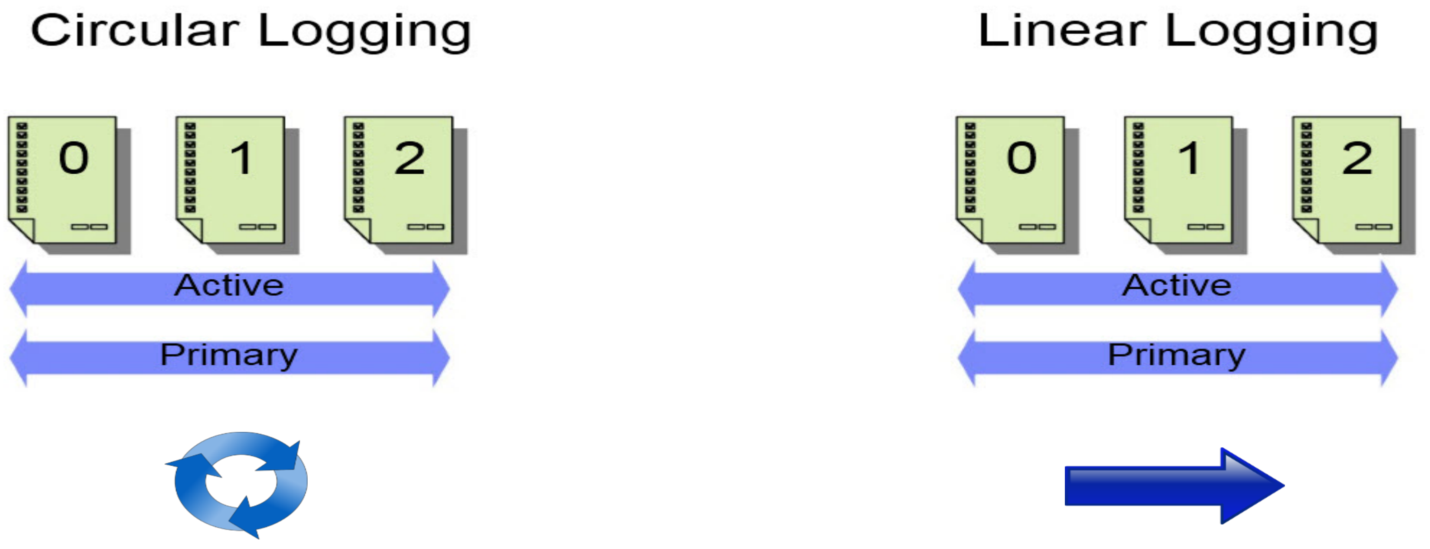
Side-by-Side Comparison - Notes

N O T E S

- The next few slides compare the behavior of Circular and Linear logging.
- The intent is to illustrate the following differences in behavior between these two logging schemes. In particular:
  - How MQ uses the Primary and Secondary log files
  - How Active log files – the ones needed recovery purposes – are handled
  - How Inactive log files – the ones no longer needed for recovery – are handled.

Log Files at initial start of Queue Manager

- **Assume Log Defaults were used**
  - LogPrimaryFiles=3, LogSecondaryFiles=2



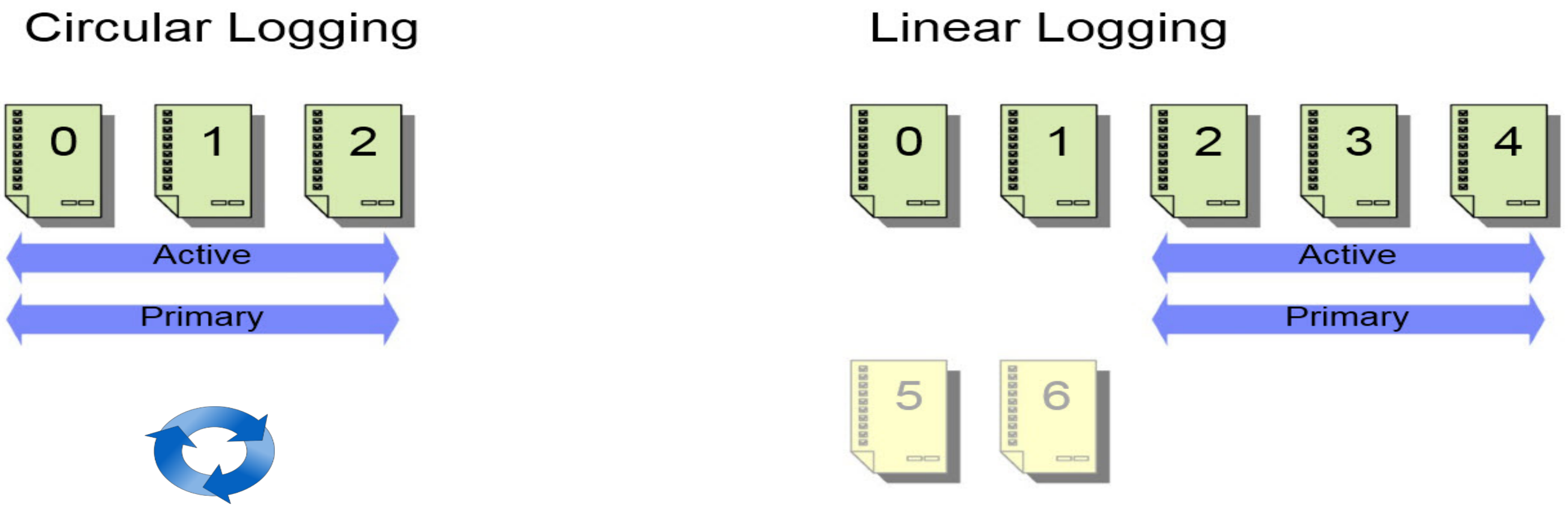
Log Files at initial start of Queue Manager - Notes

N O T E S

- Assume for purposes of illustration that the default log values were used (LogPrimaryFiles=3, LogSecondaryFiles=2).
- At initial startup, both circular and linear logging look similar - 3 primary logs, which were formatted at the time the queue manager was created.
- These 3 log files comprise the “active” log – the files that MQ might potentially use in a recovery situation.
- Secondary log files are not made use of initially.
- What follows shows the fundamental difference between how log files are used between Circular and Linear logging.

Working with Primary Log Files

- Circular – Wraps when active log full
- Linear – Marks oldest active file as inactive, and allocated a new file



Working with Primary Log Files - Notes

N O T E S

Circular logging

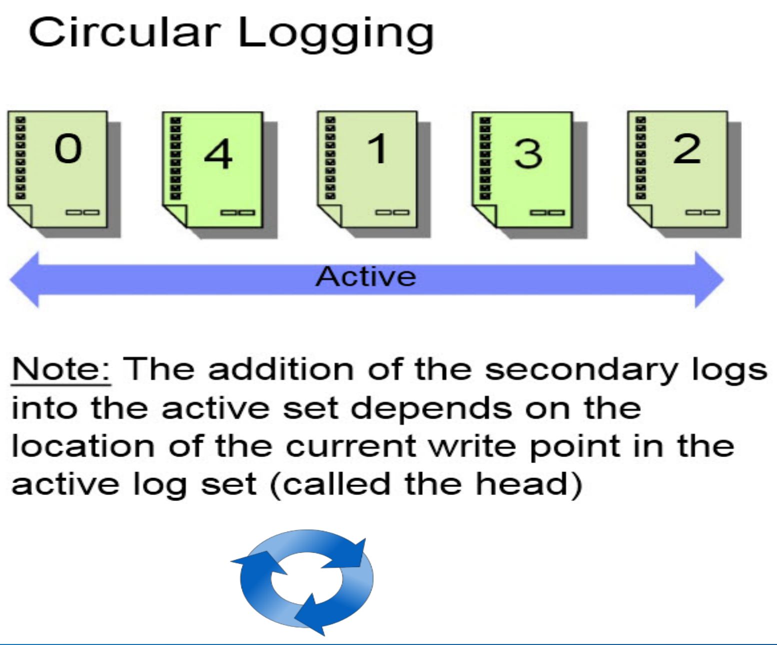
- The logger will start with what we'll call log file "0". As messages flow through the queue manager, it will progress to log file 1, then log file 2.
- Once log file 2 is full, providing no active transactions span these three log files, the logger will reuse log file 0, and the cycle repeats. This is the case even if messages that were logged have not yet been consumed. How can that be?
- With circular logging, the purpose of the recovery log is not to hold all unconsumed messages – it is to hold the state of all uncommitted messages. Messages that are committed but not yet consumed are held in the files that back the queues. Once the state of those files is in sync with the state of the log, the logger can safely overwrite the log files.

Linear logging

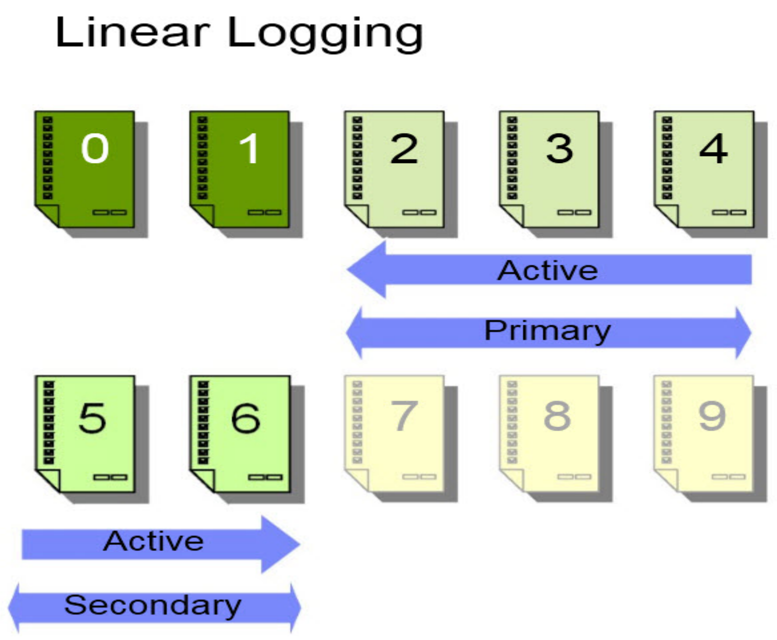
- The logger will start with log file 0. As messages flow through the queue manager, it will progress to log file 1, then log file 2.
- Notice log file 5 and 6 at bottom. With linear logging, as the logger progresses, it formats log files it knows it will need in advance of their actually being needed. This is so they are ready to be used as active log files are filled. This is a key difference between linear and circular logging – we talk about the log "ring" with circular logging, while we talk about the log "stream" with linear logging.

Expanding to Secondary Log Files

- Circular – Active log expanded by inserting secondaries into log ring
- Linear – Active log "stretched" by adding secondaries onto the end



Note: The addition of the secondary logs into the active set depends on the location of the current write point in the active log set (called the head)



Expanding to Secondary Log Files (1) - Notes

N O T E S

- Here's where things start to get interesting. We'll talk about linear first.

Linear logging

- As the logger continues to write to log file 4, it may need to move on to log file 5, but finds there is an uncommitted transaction that spans all the way back to log file 2. That presents a problem, because there are only 3 log files in the active log (2, 3 and 4). For that reason the logger cannot "retire" log file 2.

- But in the configuration some slack was allowed by allocating some number of secondary log files (2 in this example). So the logger can continue to write to log file 5 and 6 – by the time log file 6 is full that transaction had better commit, but for now there is sufficient headroom for the logger to continue.

- Thus with linear logging, the active log "stretches" to include the secondary log files if needed. In this example the active log now spans log files 2 – 6.

- You might be wondering - what about log files 0 and 1? They are shown here as outside the log as a whole – they are now considered "inactive" log files. Does that mean they are no longer needed and can be deleted? We'll discuss that shortly.

Expanding to Secondary Log Files (2) - Notes

N O T E S

Circular logging

- We see a difference here between how secondary files are used with circular verses linear logging. In this case, where a secondary log file is inserted into the log ring depends at what point we need to add additional log capacity. In the example shown, the logger has written to log file 0, 1, 2, wrapped and reused log files 0 and 1, and now wants to reuse log file 2 – but it determines that log file 2 still has data associated with an uncommitted transaction. To preserve that active transaction the logger will expand the active log by "inserting" a secondary log file (3 in this example) at that point in the log ring (between log files 1 and 2). So the active log now consists of log files 0, 1, 3, 2.

- Assume the transaction ends and log file 2 can now be reused. The logger fills log file 2 and wants to reuse log file 0. – but it determines that log file 0 has data associated with an uncommitted transaction. To preserve that active transaction the logger will again expand the active log, inserting another secondary (log file 4) at that point in the log ring (between log files 0 and 1). So now the active log consists of log files 0, 4, 1, 3, 2.

- Thus with circular logging we see that we the logger does not necessarily use the log files in their "natural" progression – they are inserted and used if and when they are needed.

- A reasonable question now becomes – how long will the logger hang only these secondary log files as part of the active log? Perhaps the active log needed to be expanded because of higher workload due to end-of-month processing – once that has passed you likely aren't going to need that additional log space for another month. So when will the logger give back these secondary extents?

Expanding to Secondary Log Files (2) - Notes

N O T E S

Circular logging (continued)

- The answer is that they will not be released immediately – after all, the logger does not know *why* the additional log space was needed, or when that additional space might be needed again. So it will continue to keep the secondary extents as part of the active log for a period of time such that it does not think they will be needed again anytime soon. This is to avoid the overhead of reducing the size of the active log only to have to turn around and expand it again - there is a cost involved in doing so. Better to hang onto the file once it is part of the active log until such time as it is seen to likely be superfluous – and then release it at that time.

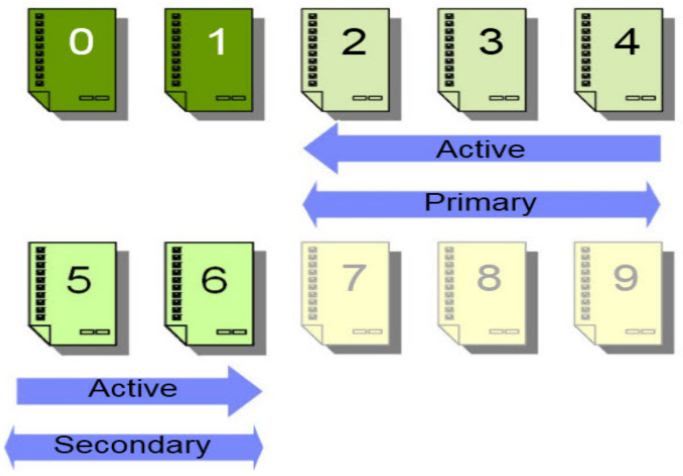
- So over time, secondary file(s) that were pulled into the active log will be freed. And assuming the primary allocation was adequate for normal processing, eventually the active log will be back to the mirroring the primary allocation (in this case, log files 0, 1, 2).

- Now extend this scenario a bit – because the logger did use the secondaries (log files 3 and 4) you decide you need additional primary log files – so you decide to change LogPrimaryFiles=3 to LogPrimaryFiles=10, then stop and restart the queue manager – what happens? Will there immediately be 10 primaries in the active log?

- No. Remember that the primary logs were allocated when the queue manager was created. If you increase the number of primary logs later, that increase is not put into effect immediately. But the logger will now see log files 1,4,1,3,2 as all primary log files, and so will not retire 3 and 4 even if no longer needed. This means you will have 5 as-yet-unused primary log files, which will only be added to the active log if and when they are needed. So the additional primaries are effectively treated as de facto secondaries. The remaining 2 (remember you added 7) will be managed as secondaries. Should any of the 5 as-yet-unused primary log files need to be pulled into the active log, they will remain permanently. Should those last 2 need to be pulled in as well, they will be treated as secondaries, and available for release if no longer needed.

Linear Logging Inactive Files

- **What about log file 0 and 1?**
- **Although not required for restart, they may be required for full recovery**
- **Messages logged periodically to identify:**
  - Oldest file needed for restart recovery
    - Oldest active log file - S0000002.LOG
  - Oldest file needed for media recovery
    - Oldest media image – S0000000.LOG
- **Can also request this via mqsc**
  - DISPLAY QMSTATUS ALL
- **Fundamental difference between linear and circular logging!**



AMQ7467: The oldest log file required to start queue manager QM1 is S0000002.LOG

AMQ7468: The oldest log file required to perform media recovery of queue manager QM1 is S0000000.LOG

Linear Logging Inactive Files (1) - Notes

N

O

T

E

S

__Linear logging__

- Let’s go back now and discuss what happens with files 0 and 1. With circular logging the files are reused and overwritten – but not so with linear logging. In the slide you see them shown as outside the log as a whole – they are now considered “inactive” log files. Does that mean they are no longer needed and can be deleted?

- Possibly. You will see the MQ logger periodically write messages like shown at the bottom of the slide - AMQ7467 and AMQ7468. These messages tell you a) The oldest log file needed to restart the queue manager (AMQ7467) – in other words, the oldest active log file, and b) The oldest log file that serves any useful purpose (AMQ7468) – the most important useful purpose being to perform media recovery, which we’ll discuss shortly.

Linear Logging Inactive Files (2) - Notes

N O T E S

Linear logging (continued)

- You can also see this same information using runmqsc, when using linear logging only. The mqsc command DIS QMSTATUS ALL will return the following:

```
DISPLAY QMSTATUS ALL
     3 : DISPLAY QMSTATUS ALL
AMQ8705: Display Queue Manager Status Details.
   QMNAME(LINEAR_QM)              STATUS(RUNNING)
   CONNS(22)                      CMDSERV(RUNNING)
   CHINIT(RUNNING)                INSTNAME(Installation1)
   INSTPATH(C:\Program Files (x86)\IBM\WebSphere MQ)
   INSTDESC( )                    STANDBY(NOPERMIT)
   CURRLOG(S0000006.LOG)          RECLOG(S0000002.LOG)
   MEDIALOG(S0000000.LOG)         STARTDA(2015-09-01)
   STARTTI(09.32.48)
```

- What do the highlighted properties mean?
  - **CURRLOG(S0000006.LOG)** Log file currently being written to
  - **RECLOG(S0000002.LOG)** Oldest log file needed for queue manager recovery
  - **MEDIALOG(S0000000.LOG)** Oldest log file needed for media recovery

What are Checkpoints?

- **Synchronize queue manager data and log files**
  - Mark a point of consistency from which log records can be discarded
  - Frequent checkpointing makes recovery quicker
- **When are checkpoints taken?**
  - Every a defined number of recoverable log operations
  - Or every 30 mins if at least 100 recoverable operations have occurred
  - And when endmqm or rcdmqimg commands issued
- **You can tune checkpointing**
  - CheckPointLogRecdMax - Checkpoint taken after this many logged operations
  - CheckPointLogRecdMin - Checkpoint taken every *CheckPointWaitLen* only if this many logged operations have occurred
  - CheckPointWaitLen - Default: 30 minutes. Minimum: 5 minutes. Maximum: 60 minutes
- **Note:** Not advisable to use these unless required
  - Doing so will have an impact on the performance – perhaps not the one you intend!

What are Checkpoints? (1) - Notes

N
O
T
E
S

- Checkpoints are used periodically to keep the log and queue files in step. This eliminates the need for the log to hold all the messages that might be queued at one time. By establishing a point of consistency between the data and log files, log records that are no longer needed can be discarded.
- Taking frequent checkpoints can make recovery time shorter after a failure. By default, checkpoints are taken when the endmqm or rcdmqimg commands are issued. In addition, checkpoints are taken periodically, either:
  - After every 50,000 recoverable log operations (10,000 in releases prior to V8)
  - Or, every 30 minutes, if at least 100 recoverable log operations have occurred

  Whichever of these occurs first.
- Checkpoint processing is a three stage process:
  1. The first stage is only carried out if the log is full or getting full (has passed an internal logger threshold). If either of these is true, the queue manager's Transaction Manager component is invoked to release log space one transaction at a time until sufficient log space has been freed or the log space is being held because the last checkpoint is pretty old. This is accomplished by rolling back active transactions or taking soft log images of prepared transactions. One transaction is processed at a time and the space threshold is rechecked with each iteration.
  2. The second stage is the checkpoint itself. If the log is full or getting full, the checkpoint is only taken if it will release log space. If the log is not getting full, a checkpoint is taken unconditionally.
  3. The third stage is to attempt to reduce the log head once more. Over time, some of the log records become unnecessary for queue manager restart – with circular logging, doing this reclaims freed space in the log files. This activity is transparent to the user and you do not see the amount of disk space used reduce, because the space allocated is quickly reused. A checkpoint removes messages put in completed transactions from the circular log.

What are Checkpoints? (2) - Notes

N O T E S

- To increase checkpoint frequency or to force checkpointing, you can use the following tuning parameters.
  - **Note:** It is not advisable to change these values unless required as they will have an impact on the performance – perhaps one that you did not intend!

- These tuning parameters correspond to a, b, and c mentioned above:
  a. CheckPointLogRecdMax - A checkpoint is taken after this many logged operations. Default Value is 50000, maximum value is 100000, minimum value is 1000.
  b. CheckPointWaitLen - Default value is 30 minutes, minimum value is 5 minutes and maximum value is 60 minutes.
  c. CheckPointLogRecdMin - Every 'CheckPointWaitLen' a checkpoint is taken if this many logged operations have occurred. Default value is 100, minimum value is 0 and maximum value is 2000
  - During normal running, checkpoints are taken either every 30 minutes provided there are at least 100 log records, but also driven when 50000 log records (10000 in releases prior to V8) have been written.

- The log and data are reconciled at queue manager start-up. You can see this happening through messages that are produced as the queue manager goes through the phases of reconciliation

The Active Log and Long-Running UOWs

- **The Active Log has a further constraint**
  - A running UOW cannot span the entire active log
- **Logger holds a portion of the potential active log “in reserve”**
  - ~20% of the active log space is held in reserve
  - This is to allow a cushion for the logger to take checkpoints
  - Remaining 80% represents the maximum “distance” between the head and the tail of the log
- **If encroached on…**
  - The logger will abort (“roll back”) the longest-running UOW
    - Better that than risk a stall if log space is exhausted
  - More than one UOW may be rolled back (or “rolled forward) if necessary
    - When this happens you will see one or more AMQ7469 errors
    - Sample program amqsblst (“MQ Blast”) can be used to demonstrate

The Active Log and Long-Running UOWs (2) - Notes

N O T E S

- The Active Log has a further constraint – a transaction that is running does not have the entire log available for use. Around 20% of the active log space is held by the Logger as a reserve, to ensure space is available if needed when taking checkpoints. Thus only about 80% of the active log is available for transaction logging – this represents the maximum “distance” between the head and the tail of the log.
- If this reserved 20% is encroached on, the Logger will start aborting (“rolling back”) active transactions, beginning with the oldest transaction that is running. It sacrifices this transaction as doing so is better than taking the chance that the queue manager as a whole will stall at checkpoint time if there is not sufficient log space available. If rolling back the oldest transaction is not sufficient, more than one transaction may be rolled back (or “rolled forward”) if necessary. For each of these you will see one or more AMQ7469 errors
- You can reproduce this behavior very easily, using the amqsblst (“MQ Blast”) sample program, if you want to demonstrate this.

Speaking of Long-Running Transactions…

- **With both circular and linear logging, transactions cannot be infinite**
  - Entire transaction must be contained in the active log
  - Reasons differ but result is the same
- **Active log must be sufficiently large to contain all active transactions**
  - Can be addressed by increasing the number of log files
  - But this is not necessarily the right thing to do
- **MQ must take action if this situation arises**
  - Reported in the queue manager error logs. Can take two forms:
    - AMQ7469: Transactions rolled *back* to release log space
    - AMQ7469: Transactions rolled *forward* to release log space
- **Often caused by applications not committing transactions**
  - Also can be caused by in-doubt channels
  - And by inappropriate use of rcdmqimg command
- **What are these messages saying?**

The Active Log and Long-Running UOWs - Notes

N O T E S

- Transactions cannot be infinite, even with linear logging. An entire transaction must be contained in the active log. That's why it is called the "active" log – it contains the recoverable actions for all the transactions that are in-flight, or active.

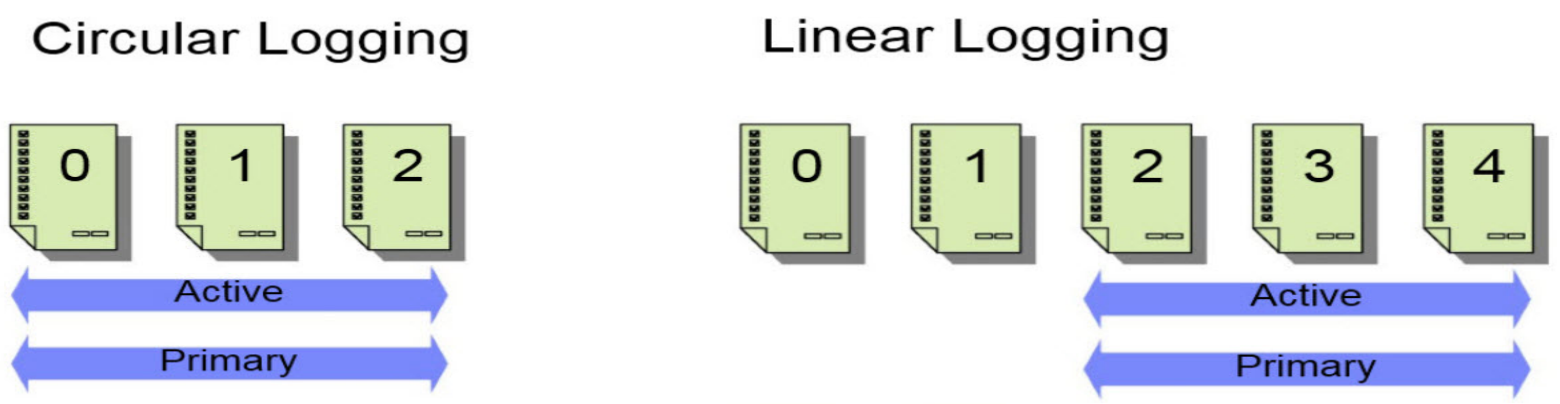
- You must therefore ensure that the Active log is sufficiently large to contain all the transactions that might be active at any given point in time. If you do not, you will see the dreaded AMQ7469 message in the error log, indicating MQ was forced to take action:
  - AMQ7469: Transactions rolled *back* to release log space
  - AMQ7469: Transactions rolled *forward* to release log space

- Often you will see these messages when an application is putting a very large number of messages in a single unit-of-work. But you can also see this if an application gets or puts a handful or even a single message, and then does not commit immediately. And three are other causes as well – in-doubt channels will cause this error, as well as running the rcdmqimg command against very deep queues.

- One way to address this is by increasing the number of log files. But this is not necessarily the right thing to do, since there is a limit to the size of the active log. Better to try and determine why transactions are running long enough to exhaust the active log space. It may be that the active log is too small. But it may also be an application has forgotten to do a commit, or is using very large units of work (UOWs). If the reason is application-related, making the log bigger is often not the best course of action – better to have the application corrected to avoid this problem, rather than use the band-aid of making the active log larger.

Why are Some Transactions Rolled Back?

- **In-flight transactions must be held in the active log only**
  - Active log has a finite size
  - Because MQ cannot arbitrarily commit a transaction, it must be rolled back
- **Why does this happen?**
  - Active log is too small for current workload
  - One or more applications may be poorly designed



Why are Some Transactions Rolled Back? - Notes

N O T E S

- As mentioned in the previous slide, transactions cannot be infinite, even with linear logging. An entire transaction must be contained in the active log. If an application takes some recoverable action that requires MQ to record it in the log, and there is not sufficient space in the active log for recording the action, MQ cannot arbitrarily commit a transaction – that might seem to be the "nice" thing to do, but it may adversely impact the application behavior, as it would essentially mean MQ is taking a single transaction and splitting it into two or more separate transactions. This could affect transaction recovery in a failure situation. Transactions must be atomic ("all or nothing"), and the only way MQ can assure that in this situation is to abort the transaction.

- What do you do in this situation? Well, it could have been a transient situation, for example a burst of workload, and if the transaction is rerun it might complete just fine. More likely, though, the cause is one of two things:
  1. The Active log is too small for current workload, perhaps because the message volume has been increasing over time and the logging parameters chosen previously need to be revisited and the size of the active log increased. If the log cannot be made bigger, perhaps because you are already at the maximum, it may be necessary to split the workload across more than one queue manager.
  2. One or more applications may be poorly designed or erroneously coded, putting too many messages in a single transaction, or perhaps putting one or more messages and then delaying the commit of the transaction for an excessive period. In this case, increasing the size of the active log is rewarding poor design or coding – Better to have the application change its behavior to conform to MQ Best Practices.

Why can Some Transactions be Rolled Forward?

- **On occasion you may see transaction rolled *forward***
- **This represents a special case**
- **A transaction may be In-Doubt, or complete ("Prepared") but not committed**
  - Could be waiting for the "commit" flow from an external TM
  - In these cases MQ rolling it back could lead to problems
  - So MQ "rolls it forward", maintaining its state in the active log until resolution
- **Why does this happen?**
  - Usually not a log space issue
  - Could be an in-doubt transaction, or an issue with an external TM

Why can Some Transactions be Rolled Forward?- Notes

N O T E S

- Occasionally you might see that the AMQ7469 message report that a transaction was *rolled forward* rather than rolled back. Why does MQ occasionally allow a long-running transaction to continue?
- As mentioned, if an application takes some recoverable action that requires MQ to record it in the log, there must be space in the active log to add a log record. But there are cases where a transaction is still active, but MQ knows the application will not be adding any additional log records.
- One example is where the application has committed the transaction, and the transaction is complete (“Prepared”) from MQ’s standpoint, but the commit has not been finalized – for example, if MQ is waiting for the “commit” flow from an external Transaction Manager.
- Now you have a quandary – the active log needs to progress to the next log file, but MQ can neither commit the transaction (because it has not been told to do so by the external TM) nor can it back out the transaction (because it’s already told the external TM that it was Prepared and cannot now “change its mind” and back out). So what MQ does with the transaction is “roll it forward”, maintaining its state in the active log, until the commit is requested and can be completed.
- Why does this happen? Usually it’s not due to a log space issue. More commonly it could be due to a transaction being in an in-doubt state, waiting for resolution. Or, it could be that there is a delay in receiving the Commit flow from an external TM.

What if I Don't Use Transactions? (1)

- **Persistent messages should always be put as part of a transaction**
  - Ensures the message is not potentially lost
  - Also has performance benefits – why?
- **Transactions intuitively would seem to slow down, not speed up applications**
  - Require at least one extra MQI call
- **How is it throughput improves when using transactions?**
  - Messages being put are not immediately hardened to the log
    - This allows messages from concurrent putters to be batched
    - This in turn allows the queue manager to handle a higher number of concurrent putters
  - Since messages may only have to be written to memory, the putter gets control back much more quickly
    - This can speed up applications that put multiple messages before issuing a commit
  - Since control is returned much more quickly, the queue lock may be held for a shorter period of time
    - This enables a larger number of concurrent getters and putters to operate against the same queue.

What if I Don’t Use Transactions? (2)

- **Contrast this with not using Syncpoint:**
  - A persistent message put outside of syncpoint is immediately forced to the log
    - The log buffer may contain few if any other messages when it must be forced to disk
    - Thus many more file writes may be needed for the same number of messages
    - The queue manager will operate much less efficiently.
  - Each MQPut will result in at least one physical write to the log file
    - The putting application will be blocked during this time
    - For applications putting multiple messages this can significantly degrade their performance.
  - Because each MQPut will take longer, the queue lock will be held longer
    - If multiple putters are using the same queue, they will “line up” behind one another
    - Result can be sequential processing of putters
    - If putting threads outnumber getters (e.g. when putting to an XMITQ) the latter can be “crowded out” by the many putting threads, and messages will pile up in the queue
- **Moral – Always put persistent messages as part of a transaction!**

What if I Don’t Use Transactions? - Notes

N O T E S

- How can using transactions improve persistent message throughput? Consider the difference in behavior with the Logger when putting messages inside vs outside of syncpoint:
  - Messages put as part of a transaction do not have to be physically written to the log at MQPut time; this means messages from many putters can be buffered up and not physically written to the log file until the buffer is full or a commit is issued. This in turn means the queue manager can handle a higher number of concurrent putters.
  - Since these messages only have to be written to memory at put time, control is returned to the putter much more quickly then if the message had to be physically written to disk before returning to the application. This can speed up applications that put multiple messages before issuing a commit.
  - Since control is returned to the putter much more quickly in this circumstance, the queue lock may be held for a much shorter period of time, enabling a larger number of concurrent getters and putters to operate against the same queue.
- Contrast this with what happens when persistent messages are put outside of syncpoint:
  - When a persistent message is put outside of transaction control, it must be forced to the log immediately. This means the log buffer will likely be holding fewer messages when it must be written to the log file, resulting in many more file writes for the same number of messages. The queue manager will operate much less efficiently.
  - Because each MQPut will result in at least one (and possibly several) physical writes to the log file, more time will be required to complete the put and return control to the application. For applications that put many messages this can significantly degrade their performance.
  - Because each MQPut will take longer, the queue lock will be held longer. If multiple threads are putting to the same queue, the result will be that the putters effectively “line up” behind one another, with the performance of all suffering. And if there are more putting threads than getting threads (such as when putting to an XMITQ) the getting threads can be “crowded out” by the many putting threads, and messages will pile up in the queue.

Linear Logging – Media Images (1)

- **Media Recovery requires periodic media images be recorded**
- **The rcdmqimg command captures an image of MQ object(s)**
  - Image is written to the MQ log for use in media recovery
  - Linear logging only
  - Moves the LSN (log sequence number) forward - freeing up log files for archival or deletion
- **MQ does not run this automatically!**
  - Must be run manually or from an automated task
  - You must decide when and how images are taken
- **Some times are good for taking a media image:**
  - When a queue is empty (or nearly so)
  - When the system is quiet (or nearly so)
  - When the number of log files required for media recovery is large
  - When a lot of time and/or activity has passed since the last image was captured

Linear Logging – Media Images (1)

- **Conversly, there are times that are not so good:**
  - When queue(s) are large or growing
  - When there is a lot of activity in the system
  - When a recent image already exists
- **Using rcdmqimg during not-so-good times can be deleterious**
  - Applications can experience slowdowns as queues are locked while being recorded
  - AMQ7469 errors ("Transactions rolled back...") and AMQ6150 ("Semaphore busy...") errors possible
  - File system usage can increase as queue data is being copied from the queue files to the log
- **Recommendations:**
  - Try to use during good times only
  - May mean running once daily (probably at night)
  - Means sufficient file system must be available

Linear Logging – Media Images - Notes

N O T E S

- The rcdmqimg command captures an image of MQ object(s), which are then written to the MQ log and are available for use in media recovery – for example, if an MQ object becomes damaged.
- With Circular logging the image would be overwritten when the log wrapped – for this reason, media recovery is only available if you are using Linear logging.
- The rcdmqimg command moves the Log Sequence Number (LSN) forward, freeing up log files for archival or deletion.
- It is important to note that MQ cannot be configured to run this automatically – it must be run manually or from an automated task. This means it it up to you to decide when and how images are taken.
- Some times are better than others for taking a media image - When a queue is empty (or nearly so), when the system is quiet (or nearly so), when the number of log files required for media recovery is large, and when a lot of time and/or activity has passed since the last image was captured. This implies that there are times that are not so good for taking media images - When queue(s) are large or growing, or when there is a lot of activity in the system, or, if a recent image already exists.
- Running rcdmqimg during not-so-good times can be deleterious to your applications. They may experience slowdowns as queues are locked while being recorded. You may also see errors such as AMQ7469 ("Transactions rolled back to release log space") and AMQ6150 ("Semaphore busy...Long Lock Wait"). Also, file system usage can increase as queue data is being copied from the queue files to the log. These things can make MQ appear to be unstable.
- Recommendations: Try to use during "good times" only, if at all possible. This may mean running once daily, most likely at night. Less frequent use will also mean sufficient file system space must be available to hold the inactive but not-yet-deletable log files.

Behind the Curtain…

Operational View of the Logger

- **Gaining insight into how the Logger is operating**
  - Knowledge is power!
  - Not easy to come by - although this is improving!
- **MQ Appliance, MQ V9 makes this information easier to obtain**
  - Monitoring interface provides access to Log data
  - We'll look at this in the next section
- **Service aid "amqldmpa" allows a peek behind the curtain**
  - Not easy to interpret
  - Much of what's reported is not useful outside L3/Development
  - But we'll look at some useful tidbits that can prove insightful
- **Realize this is undocumented for a reason**
  - What is reported can change at any time
  - Examples that follow are from MQ V8

Operational View of the Logger - Notes

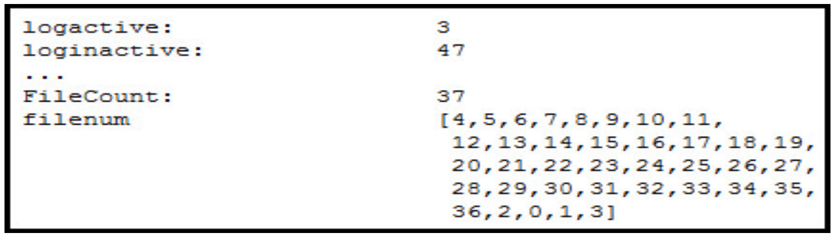
N O T E S

- Insight into how the Logger is operating can be of use in understanding MQ performance as well as how optimal the logger is being used, and what kind of tuning might be of value.
- Not a lot of information is available specific to the operation of the Logger, although this is improving – The MQ Appliance as well as IBM MQ V9 both make this information easier to obtain. We'll see that in the next section.
- If you are not using either of those as of yet, there is an alternative - MQ does ship with a service aid (amqldmpa) that can provide some insight into the operational state of the Logger. This tool is not intended for general use - much of what's reported is not useful outside L3/Development, so it is not easy to interpret. But you can glean some useful nuggets of information, and we'll look at some useful tidbits that can prove insightful in the next few slides.
- Be aware that this tool is undocumented for a reason – what it reports is subject to change at any time, and so it is not a good idea to include it in scripts, etc that are run on a regular basis, as even the application of fixpacks can cause the output it generates to change.
- The examples that follow are from MQ V8, as that is presumably the most common version in use today. But most of what we'll explore can be found in prior MQ releases – although the format may be different depending on the version.

How to tell if you are using Secondary Logs?

- **You may want better insight into log file usage**
  - You may see messages that your log is almost full, or that transactions are being rolled back
  - You may want to more intelligently choose the number of Primary and Secondary logs

- **You can use the amqldmpa command to dump the state of the logger**
  - *amqldmpa -m <qmgr> -c H -f <file>*
  - In <file> look for the following



```
logactive:                    3
loginactive:                  47
...
FileCount:                    37
filenum                       [4,5,6,7,8,9,10,11,
                               12,13,14,15,16,17,18,19,
                               20,21,22,23,24,25,26,27,
                               28,29,30,31,32,33,34,35,
                               36,2,0,1,3]
```

  - ***logactive*** tells you the number of *Primary* log files
  - ***loginactive*** tells you the number of *Secondary* log files
  - ***FileCount*** tells you the number of log files in the *Active* log
  - ***Filenum*** is the list of log files that are currently in use

- **See the following Technote:** http://www-01.ibm.com/support/docview.wss?uid=swg21623541

How to tell if you are using Secondary Logs? - Notes

N O T E S

- One question that has come up repeatedly with customers is “How do I know how many log files to allocate?”.
- You may be seeing messages to the effect that the log is almost full, or that transactions are being rolled back. We’ve covered why you do not want to run out of log space. But an excessive log allocation wastes file system space. How can you get better insight into log file usage, so that you can more intelligently configure primary and secondary logs? This is specially the case when using Circular logging.

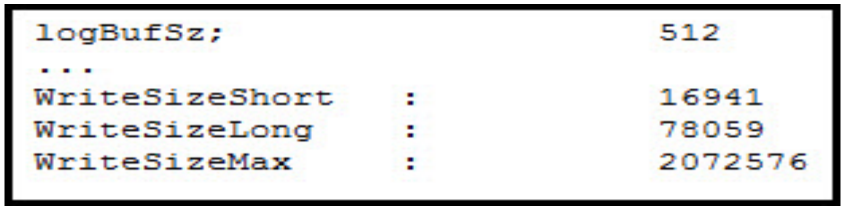
- You can use the amqldmpa command to dump the state of the logger
  - *amqldmpa -m <qmgr> -c H -f <file>*
  - In the <file> that is generated by the command, look for the following:

– logactive:	Number of primary log files in use
– loginactive:	Number of secondary log files in use
– FileCount:	Number of active log files – if greater than logactive than using secondaries
– Filenum:	List of log files currently in use

- One strategy is to allocate a small number of Primary files and a large number of secondary files, and then monitor the above over time. Use FileCount as the basis for setting the number of Primaries, and add some number of secondaries for a safety margin.
- See the following Technote: http://www-01.ibm.com/support/docview.wss?uid=swg21623541

Looking at Efficiency of Log Buffering

- **How efficiently is log I/O being buffered?**
- **For a given amount of data, fewer larger writes are more efficient than many small writes**
- **The amqldmpa command can be used to see the amount of data on average being written**



```
logBufSz;                         512
...
WriteSizeShort    :               16941
WriteSizeLong     :               78059
WriteSizeMax      :               2072576
```

  - ***logBufSz*** is the size of the log buffer in 4K pages (512 = 2MB buffer)
  - ***WriteSizeShort*** is the short-term average number of bytes written per log write
  - ***WriteSizeLong*** is the longer-term average number of bytes written per log write
  - ***WriteSizeMax*** is the largest number of bytes written to the log in a single write

- **In this example…**
  - Averages are pretty decent (~16KB to 76KB)
  - But the buffer is almost maxed out (2MB)
  - So might be worthwhile to increase log buffer in this case

Looking at Efficiency of Log Buffering- Notes

N

O

T

E

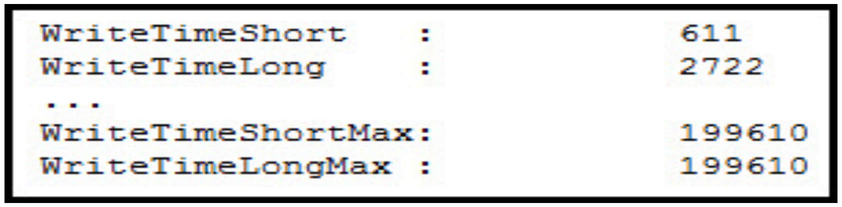
S

- Poor performance due to delays in writing to the log can be difficult to diagnose. One item to look at is how efficiently log I/O is being buffered. With persistent messaging, the log can be a bottleneck, so the goal should be to have as much data as possible buffered before the log is forced to do a write to disk. The more data that can be written per log write, the better, since for a given amount of data, fewer larger writes are more efficient that many smaller writes.
- You can use the amqldmpa command to see the amount of data on average being written. Run command "***amqldmpa -m <qmgr> -c H -f <file>***", *and in* <file> look for the following:
  - logBufSz: Size of log buffer. This is expressed in number of 4K pages - 512 * 4096 means a 2MB buffer
  - WriteSizeShort: Short-term weighted average size (in bytes) of 64 most recent writes
  - WriteSizeLong: Longer-term weighted average size (in bytes) of 1024 most recent writes
  - WriteSizeMax: Largest single write (in bytes)
- In this example, assuming small messages sizes, the averages are pretty decent - ~16KB to 76KB. If the average message was 2K, then on average you are getting between 8 and 38 messages written to the log in a single write.
- But interestingly, the buffer was at least once almost maxed out (2MB). That might be saying that you have at least some messages that are very large, or perhaps the average message size is larger than you had thought. Either way, if might make sense in this case to increase the size of the log buffer and observe the results in a subsequent test.

Log Write Latency

- **Poor persistent message throughput can result from write latency to the disk**
  - For these must look outside MQ for resolution - but MQ can provide some clues

- **The amqldmpa command can be used to obtain the write latency the Logger is seeing**



```
WriteTimeShort    :              611
WriteTimeLong     :              2722
...
WriteTimeShortMax:               199610
WriteTimeLongMax :               199610
```

  - ***WriteTimeShort*** is the short-term average latency (in µs)
  - ***WriteTimeLong*** is the longer-term average latency (in µs)
  - ***WriteTimeShortMax*** is the high-water-mark for the short-term weighted average
  - ***WriteTimeLongMax*** is the high-water-mark for the long-term weighted average

- **In this example…**
  - Average write latency is decent (0.6 to 2.7ms)
  - But there have been spikes (~200ms)
  - May be worth monitoring over time to see longer term averages

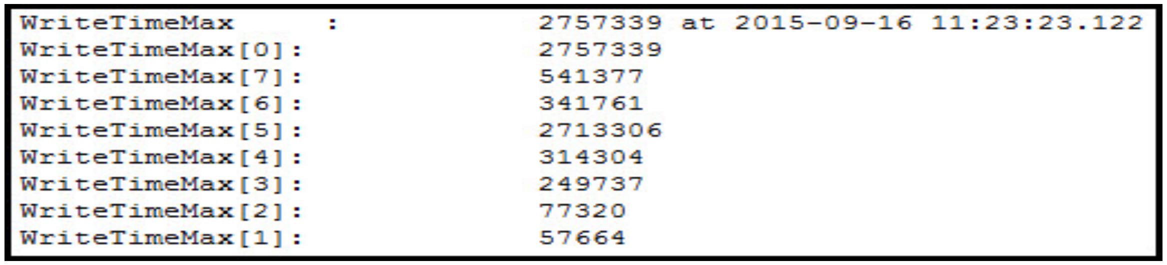
Log Write Latency - Notes

N O T E S

- A more common issue when seeing poor persistent message throughput can be high write latency to the disk where the log resides. Many factors can contribute to this – few customers (outside of MQ Appliance users) use local storage to host the logs anymore, so writing to the log means going across the network to a storage device, perhaps with the I/O virtualized, etc, etc. This journey can take more time than is realized, and so some indication that this might be occurring can be useful.
- You can use the amqldmpa command to see the amount of latency on average when writing to the log. Run command "***amqldmpa -m <qmgr> -c H -f <file>***"*, and in* <file> look for the following:
  - ***WriteTimeShort*** is the short-term average latency (in microseconds) of the 64 most recent log writes
  - ***WriteTimeLong*** is the longer-term average latency (in microseconds) of the 1024 most recent log writes
  - ***WriteTimeShortMax*** is the high-water-mark for the short-term average
  - ***WriteTimeLongMax*** is the high-water-mark for the long-term average
- In this example, average write latency is decent (between 0.6 to 2.7milliseconds) but there have been spikes as high as 200 milliseconds. So this might be something that is worth monitoring over time to see the averages and pattern of write latency over longer periods of time.

Log Write Latency over Time

- **High latency high-water-marks may be more useful when monitoring over time**
- **The amqldmpa command also reports the log write latency over time**



```
WriteTimeMax      :          2757339 at 2015-09-16 11:23:23.122
WriteTimeMax[0]:             2757339
WriteTimeMax[7]:             541377
WriteTimeMax[6]:             341761
WriteTimeMax[5]:             2713306
WriteTimeMax[4]:             314304
WriteTimeMax[3]:             249737
WriteTimeMax[2]:             77320
WriteTimeMax[1]:             57664
```

  - ***WriteTimeMax***: Longest log write time since queue manager started (in µs)
    - V8 will report the date/time this occurred, as well as the highest latency log write for the 8 most recent log files

- **In this example…**
  - HWM > 2.757 seconds – ouch! (I've seen customer examples > 60 seconds!!!)
  - Of the last 8 log files none were very good (from 57ms to 2713ms)
  - Capture over time and sit down with SAN/NAS team to resolve

- **A trap can also be set to capture this – see link in note.**

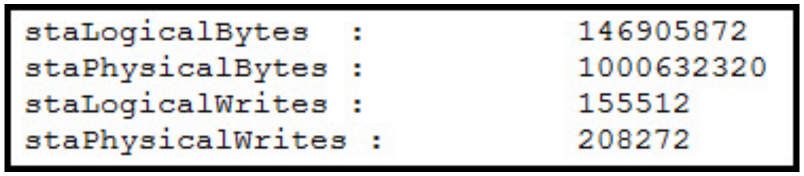
Log Write Latency over Time - Notes

N O T E S

- MQ V8 makes it easier to see log write latency over time. Prior to V8 the amqldmpa command would report the longest log write latency (“WriteTimeMax“) since the queue manager was started. But this was of limited use, as you could not easily see whether that one high-latency write was a one-off event or an ongoing problem; nor could you tell with any precision exactly when that log write occurred.
- With MQ V8 the amqldmpa command will now report the longest log write latency since the queue manager started, but will also tell you the date and time that occurred. In addition, you can also see the highest latency log write for each of the 8 most recent log files used, making it easier to tell whether high-latency writes are occurring frequently.
- Run command “***amqldmpa -m <qmgr> -c H -f <file>***”*, and in* <file> look for the following:
  - ***WriteTimeMax*** will report the highest latency log write time since queue manager started (in µs), as well as the date/time this occurred. In addition, the highest latency log write for the 8 most recent log files used is reported.
- In this example from the slide, the latency high-water-mark is greter than 2.757 seconds – eons of computer time! And of the last 8 log files, each had at least one event of high write latency (from 57ms to 2713ms)
- It is also possible to set a trap to have an FDC generated when log write latency exeeds a predefined level. See this link: *https://developer.ibm.com/answers/questions/190102/how-can-i-identify-possible-io-delay-for-mq-perfor.html*
- Armed with this information you can sit down with SAN/NAS team and state with precision when high latency was seen, and hopefully resolve the problem more quickly.

Looking at Efficiency of Log File Usage

- **How efficiently is log space being used?**
- **Log writes are always some number of 4K pages**
  - But some of that space will be “empty” - not usable for recovery purposes
  - More recovery data in the log pages means the logger is being used more efficiently
- **The amqldmpa command can be used to gauge how efficiently log space is being utilized usage**



```
staLogicalBytes   :          146905872
staPhysicalBytes  :          1000632320
staLogicalWrites  :          155512
staPhysicalWrites  :         208272
```

  - ***staLogicalBytes*** is the number of bytes written that are useful for recovery purposes
  - ***staPhysicalBytes*** is the total number of bytes written to the recovery log
  - ***staStaLogicalWrites*** is the number of log writes with data needed for recovery purposes
  - ***staPhysicalWrites*** is the total number of log writes that were performed
- **Let’s look at a practical example…**

Looking at Efficiency of Log File Usage - Notes

N O T E S

- One consideration when sizing the log is how efficiently the log space is being used. Log writes are always a multiple of 4K pages; depending on the degree of concurrency in the queue manager, the frequency of log forces occurring, etc, there will usually be some amount of unused or “empty” space in the log file – space that does not contain any data needed for recovery.

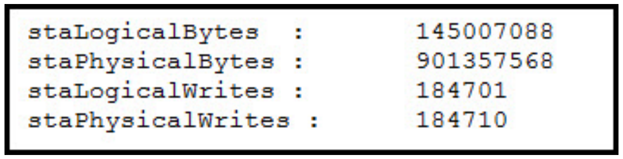
- You can use the amqldmpa command to gauge how efficiently log file space is being utilized:
  - *amqldmpa -m <qmgr> -c H -f <file>*
  - In the <file> that is generated by the command, look for the following:
    - ***staLogicalBytes***: The number of bytes written that are useful for recovery purposes
    - ***staPhysicalBytes***: The total number of bytes written to the recovery log
    - ***staStaLogicalWrites***: The number of log writes containing data needed for recovery purposes
    - ***staPhysicalWrites***: The total number of log writes that were performed

- The next few slides show a practical example of how you can use this information.

Log Efficiency – SingleWrite vs TripleWrite (1)

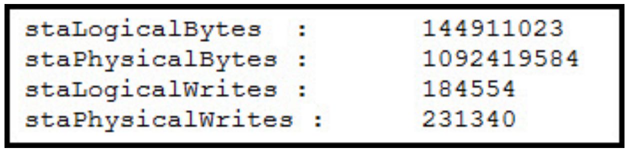
- **TripleWrite will result in some redundant data being written to the log**
- **Wanted to see how much, under different workloads**
- **Tested using a <u>single</u> putter and getter**
- **Observations:**
  - In both cases the logger was not making very efficient use of the log file (13% - 16% utilized)
  - Both logged about the same amount of recovery data (~145MB)
  - Both used about the same number of Logical Writes to log that data (~184K)
  - But look at the number of Physical Writes:
    - SingleWrite had almost identical values for Logical and Physical Writes
    - TripleWrite had ~25% more Physical Writes

➢ **SingleWrite**



```
staLogicalBytes   :        145007088
staPhysicalBytes  :        901357568
staLogicalWrites  :        184701
staPhysicalWrites  :       184710
```

➢ **TripleWrite**



```
staLogicalBytes   :        144911023
staPhysicalBytes  :        1092419584
staLogicalWrites  :        184554
staPhysicalWrites  :       231340
```

Log Efficiency – SingleWrite vs TripleWrite (1) - Notes

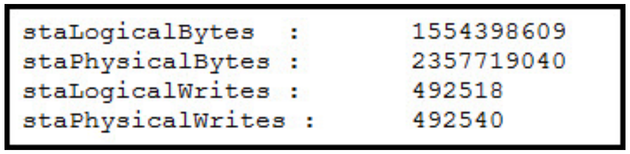
N O T E S

- We've discussed how using LogWriteIntegrity=TripleWrite will result in some redundant data being written to the log. We've also discussed how the logger tries very hard to minimize the additional writes, particularly when there is a lot of concurrent persistent workload. I wanted to see if I could demonstrate that.
- I ran two use cases. In the first use case, I used a single putter and getter processing persistent messages. I ran this once using LogWriteIntegrity=SingleWrite, and a second time using LogWriteIntegrity=TripleWrite, and compared the results.
- In both cases roughly the same amount of recoverable data was written to the log – around 145MB. And in both cases, it took around the same number of Logical Writes to log that data – around 184,000.
- But two things stood out:
  - Comparing the number of logical bytes written (the amount needed for recovery) with the total number of physical bytes written revealed that in both cases the log was being very underutilized – only about 13% (SingleWrite) to 16% (TripleWrite) of the data written to the log was actual recoverable data.
  - Comparing the number of logical verses physical log writes was even more interesting – With SingleWrite the number of physical and logical writes were almost identical (around 184,000), but with TripleWrite the number of physical writes was much higher – over 231,000, or about 25% more log writes.

Log Efficiency – SingleWrite vs TripleWrite (2)

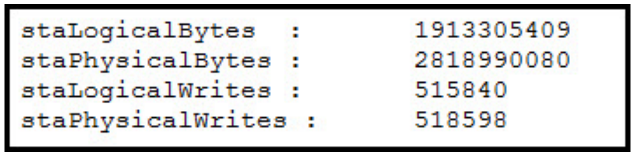
- **Ran a second test using <u>ten</u> putter and getter pairs**
- **Observations:**
  - Log file now much more efficiently used (66% - 68% utilized)
    - But now we see the number of Logical and Physical Writes much the same regardless of LogWriteIntegrity
    - In the single putter test TripleWrite had ~25% more Physical Writes
    - In this test TripleWrite resulted in <1% more Physical Writes
- **Conclusion:**
  - With a single getter and putter, SingleWrite can significantly reduce the number of physical writes
    - But does not use the log file very efficiently
  - With ten concurrent getters and putters, LogWriteIntegrity value makes very little difference
    - With more concurrent activity the log file is much more efficiently utilized
    - And TripleWrite is safer when there is any doubt as to write integrity

- **SingleWrite**



```
staLogicalBytes   :          1554398609
staPhysicalBytes  :          2357719040
staLogicalWrites  :          492518
staPhysicalWrites  :         492540
```

- **TripleWrite**



```
staLogicalBytes   :          1913305409
staPhysicalBytes  :          2818990080
staLogicalWrites  :          515840
staPhysicalWrites  :         518598
```

Log Efficiency – SingleWrite vs TripleWrite (2) - Notes

N O T E S

- The second use case was using ten putting and ten getting threads, with each putter/getter having their own queue to minimize lock contention. I again ran this test once using LogWriteIntegrity=SingleWrite, and a second time using LogWriteIntegrity=TripleWrite, and compared the results.
- Comparing the number of logical bytes written (the amount needed for recovery) with the total number of physical bytes written revealed that in this test the log was being much more efficiently utilized – now we see 66% (SingleWrite) to 68% (TripleWrite) of the data written to the log was actual recoverable data.
- Most interesting, though, were the number of log writes. In the use case with a single putter/getter, I saw about 25% more log writes with using TripleWrtie verses SingleWrite. In this test I observed that the number of Logical and Physical Writes was more or less the same regardless of LogWriteIntegrity - in the single putter test TripleWrite had about more Physical Writes, but in this use case TripleWrite resulted in less than 1% additional Physical Writes.
- Conclusion:
  - With a single getter and putter, SingleWrite can significantly reduce the number of physical writes. But it can put data at risk (as previously discussed) and most important, it's unlikely many MQ customers have queue manager used by a single putter and getter.
  - With a higher number of concurrent putters and getters, ten in this case, the value specified for LogWriteIntegrity makes very little difference. With more concurrent activity the log file is much more efficiently utilized. And TripleWrite is safer when there is any doubt as to write integrity.

What's New?

Recent Changes to Software MQ

- **Enhancements to improve Logger performance in MQ V8**
  - When possible V8 uses writev() when writing to the log
    - Allows writing continuous data from the end and the start of the log buffer in a single I/O operation
    - Older MQ versions suffered an additional log force when log buffer wrapped
  - Default checkpoint frequency is now every 50,000 recoverable operations
    - Older versions of MQ default to every 10,000 operations
    - If you used *CheckPointLogRecdMax* you may want to revisit
  - Default checkpoint delay length is now 0.5 seconds
    - Older releases of MQ used 0.25 seconds
- **And in MQ V9…**
  - Some of the logger statistics discussed here are more easily obtained
  - Sample program amqsrua can be used to capture and report this (or you can write your own!)
  - See this developerWorks article:
    - **https://www.ibm.com/developerworks/community/blogs/messaging/entry/Statistics_published_to_the_system_topic_in_MQ_v9?**

Recent Changes to Software MQ - Notes

N

O

T

E

S

- MQ V8 saw a number of small but useful enhancements to improve Logger performance. The writev() call is now used when possible when writing to the log, which allows writing continuous data from the end and the start of the log buffer in a single I/O operation - older MQ versions suffered an additional log force when log buffer wrapped.

- Some of the log defaults were also increased with MQ V8. The default checkpoint frequency is now every 50,000 recoverable operations – in older versions of MQ the default was every 10,000 operations. This was a tunable frequency, so you you have used the *CheckPointLogRecdMax* tuning parameter in qm.ini you may want to revisit this. Also, the default checkpoint delay length is now 0.5 seconds, rather than 0.25 seconds as was the case in previous version of MQ.

- With MQ V9 comes an easier method for obtaining the logger internal information we discussed earlier. Rather than use the amqldmpa tool, a new sample (amqsrua) is provided that can report the same information. This sample can be much easier to use than the amqldmpa service aid, and as a sample, can also be used as the basis for writing your own program to collect and report these (and other) statistics. More information about amqsrua can be found at these locations:
  - https://www.ibm.com/developerworks/community/blogs/messaging/entry/Statistics_published_to_the_system_topic_in_MQ_v9
  - https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.mon.doc/mo00013_.htm
  - https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.mon.doc/mo00014_.htm.

…And the MQ Appliance

- **Appliance-hosted queue managers also use recovery logs**
  - Circular logging only
  - Same defaults as Installable MQ
- **Appliance HDD a finite resource**
  - Monitoring important
  - Especially if hosting multiple queue managers
  - M2001 “disk” larger, uses SSD technology
    - Much better performance, particularly with large messages
- **New event generation scheme**
  - Published on well-known topics
  - Can create your own alert monitor
  - Also in MQ V9
- **Chart widgets instead of amqldmpa!**
  - Health of Logging much easier to monitor!
  - Free space, Log usage, Write Latency, etc
  - amqsrua sample can also be used

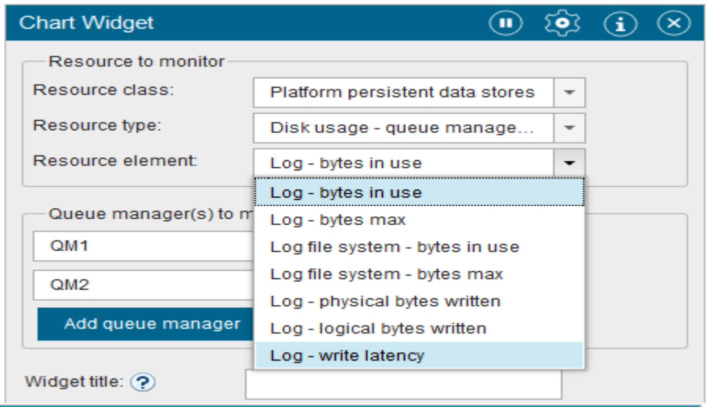


Chart Widget

Resource to monitor

Resource class: Platform persistent data stores

Resource type: Disk usage - queue manage…

Resource element: Log - bytes in use

Log - bytes in use
Log - bytes max
Log file system - bytes in use
Log file system - bytes max
Log - physical bytes written
Log - logical bytes written
Log - write latency

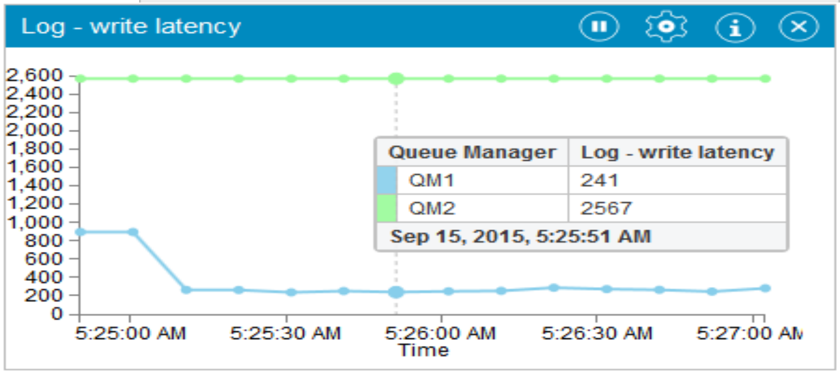
Queue manager(s) to m

QM1

QM2

Add queue manager

Widget title:


Log - write latency

Queue Manager	Log - write latency
QM1	241
QM2	2567

Sep 15, 2015, 5:25:51 AM

…and the MQ Appliance - Notes

N O T E S

- Queue managers hosted on the IBM MQ Appliance use recovery logs as well. There are some special considerations with logging on the appliance.
- Foremost of these is that only Circular logging is supported – Linear logging is not an option. This is in keeping with the MQ Appliance being a self-contained appliance. So other approaches must be used if object recovery (object "restoration" would be a better term) is desired. One driver for the use of Linear logging is media failure, but the appliance contains two SDD drives in a RAID configuration, minimizing the likelihood of media failure. The use of HA and/or DR replication can minimize this further.
- Being an appliance, disk storage is more of a finite resource than with other environments, so more care needs to be taken in determining the size and number of log files – especially if there will be a number of queue managers hosted on the appliance. The M2001 appliance triples the amount of available storage, making this less of a concern.
- The log defaults are the same with the MQ Appliance as with software MQ. And as with software MQ, these can be changed – in the case of the appliance, you would do this using the supplied setmqini command rather than editing a qm.ini file.
- A large advantage of the MQ Appliance is the MQ Console, a browser-based tool that supplies Chart widgets, which make monitoring of log usage much easier. Information can be collected on disk space available on the appliance, as well as statistics on Log usage as well as Log write latency.

Summary

Summary

- **Mysteries and Decisions**
  - Circular and Linear
  - Understanding Logging Parameters
  - Log Write Integrity!
- **Log Operation**
- **AMQ7469!**
- **Behind the Curtain**
  - Tools to help you better understand the Logger
- **What's New**
  - MQ V9
  - MQ Appliance
- **Wrap-Up**

Questions & Answers