SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 5: Logging, Recovery, and the Transaction Log

Paul S. Randal
Paul@SQLskills.com

immerse yourself in sql server

Overview

- Transaction log architecture
- Log records
- Checkpoints and recovery
- Transaction log operations
- Recovery models
- Log file provisioning and maintenance

Basic Terminology

- Transaction: a set of changes to a database
- Commit: finalize a transaction
- Roll back: abort a transaction and undo its effects
- Logging: writing a durable description of changes to a database
- Log record: a description of a single, small change to a database
- Transaction log: the file (or files) where log records are stored
- Crash: when SQL Server shuts down unexpectedly
- Recovery: making a database transactionally-consistent, specifically after a crash has occurred (i.e. crash recovery)
- Checkpoint: writing changed data-file pages to non-volatile storage

Why is Logging Required?

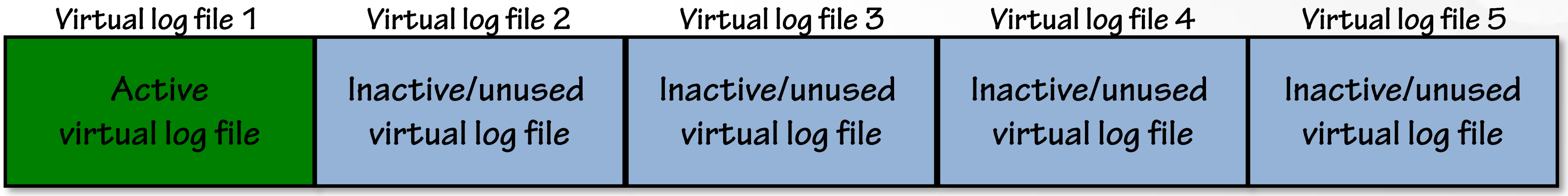
- Logging allows transactions to be made durable and recoverable in the event of a crash
- Without logging, a database would be transactionally inconsistent, and potentially corrupt after a crash
- Without logging, how would a transaction roll back?
- Without logging, how would backups work? Replication? Mirroring? Availability Groups? Log shipping?

- SQL Server 2014+ has non-logged tables, which we'll discuss later
- The version store in tempdb is completely non-logged, not even the allocations/deallocations of pages and extents

Write-Ahead Logging

- **SQL Server uses a mechanism called 'write-ahead logging'**
- **This ensures that a data-file page change cannot *EVER* be written to disk before the log records describing the change are written to disk**
    - It's an invariant, even when delayed durability is used in SQL Server 2014+
- **Without the write-ahead logging guarantee, how would recovery work?**
    - Consider: a data-file page change is written to disk before the log records describing the change, and then SQL Server crashes
    - Without the log records describing the change, how can SQL Server know whether to leave the data-file page change intact or remove the change?
    - And if it has to remove the change, how can that be done without the description of the change itself?
- **Write-ahead logging combined with periodic checkpoints also increases the efficiency of persisting changes to disk**
- **Books Online description: https://sqlskills.com/p/036**

Virtual Log Files



Virtual log file 1	Virtual log file 2	Virtual log file 3	Virtual log file 4	Virtual log file 5
Active virtual log file	Inactive/unused virtual log file	Inactive/unused virtual log file	Inactive/unused virtual log file	Inactive/unused virtual log file

- **The transaction log is divided up into chunks called virtual log files, or VLFs for short**
- **Newly created VLFs are inactive and unused**
  - An active VLF cannot be reused until it is made inactive by log clearing
- **Except that in a new database, the first VLF is always active**
  - There must always be at least one active VLF in the transaction log
- **There is an 8KB file header page at the start of the transaction log file**
  - Stores metadata about the file such as size and auto-growth settings

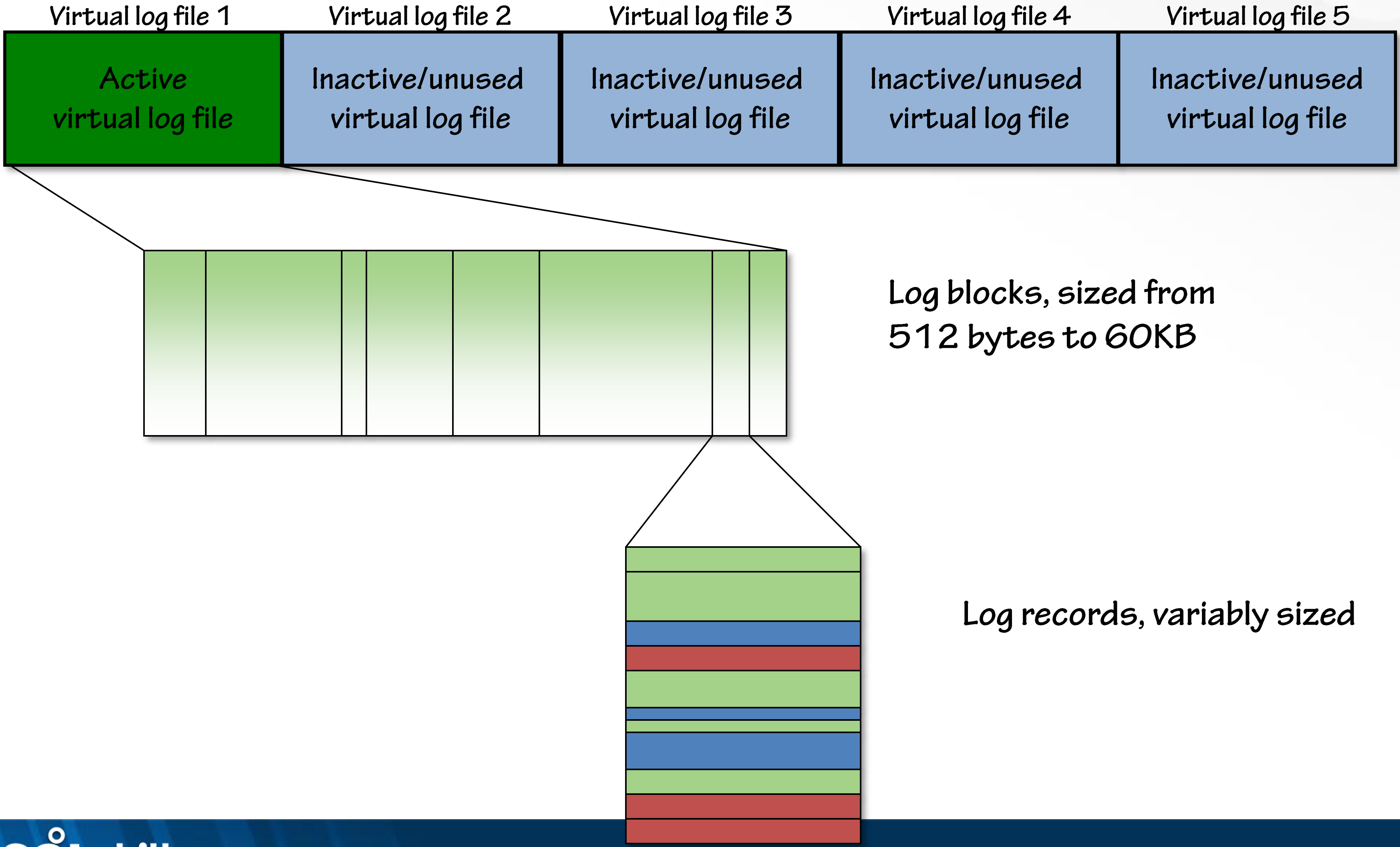
How Many VLFs Do You Get?

- **The number and size of the VLFs in a new portion of the transaction log are determined by SQL Server and cannot be configured**
  - Under 1MB is irrelevant for discussion
  - Under 64MB there will be 4 new VLFs (each ¼ of growth size)
  - 64MB to 1GB there will be 8 new VLFs (each 1/8 of growth size)
  - Above 1GB there will be 16 new VLFs (each 1/16 of growth size)
  - This formula applies to the initially-created transaction log, and for each manual or automatic growth that occurs
- **On SQL Server 2014+**
  - If growth size is less than 1/8 of current log size, only one VLF is created
  - See blog post at https://sqlskills.com/p/037 for details
- **However, knowing the formula used means you can control how many VLFs are created**
  - Too few/many VLFs can cause performance problems with log operations

VLF Sequence Numbers

- **Each VLF has a 4-byte sequence number, which uniquely identifies it**
- **Sequence numbers increase by one each time the next VLF is made active**
- **The start of the 'active' portion of the transaction log begins with the VLF that has the lowest sequence number and is still active**
- **The VLF sequence numbers for a new database do not start at 1**
  - They start with whatever the highest VLF sequence number is in the model database, plus 1
- **It is extremely unlikely you will run out of VLF sequence numbers**
  - In fact, SQL Server has code that will force the instance to shut down if a VLF sequence number ever wraps around to zero
  - E.g. 1000 VLFs in 10GB log = 2^32/1000 log wraps = 10GB * 4294967 of log = 256MB/s of log for 1300 years

VLFs and Log Blocks (1)


Virtual log file 1
Virtual log file 2
Virtual log file 3
Virtual log file 4
Virtual log file 5
Active virtual log file
Inactive/unused virtual log file
Inactive/unused virtual log file
Inactive/unused virtual log file
Inactive/unused virtual log file

Log blocks, sized from 512 bytes to 60KB

Log records, variably sized

notes for
prior slide

VLFs and Log Blocks (2)

- **Each VLF contains a VLF header, which includes:**
  - Whether the VLF is active or not
  - The log sequence number when the VLF was created
  - The current parity bits for all 512-byte blocks in the VLF
    - Start at 64 and switch back-and-forth between 64 and 128
    - These are used during crash recovery (discussed later)
- **Each VLF contains a series of log blocks**
  - Log blocks vary in size from 512 bytes, in 512-byte increments, to 60KB
  - The log block size is set when one of the following occurs:
    - A transaction generates a log record to commit a transaction
    - The log block size reaches 60KB without a transaction committing
- **Each log block contains log records**
  - Stored in the order written, in a similar manner to a data-file page
  - Log records from multiple transactions can be in a single log block

Log Sequence Numbers (LSNs)

- **LSN = <VLF sequence #>:<log block #>:<log record #>**
  - VLF sequence number is 4-bytes
  - The log block number within the VLF is 4 bytes
  - The log record number within the log block is two bytes
- **LSNs are ever-increasing**
- **Each log record has a unique LSN that allows the log record to be found in the transaction log**
- **Each data-file page has an LSN in its page header that identifies the most recent log record whose change is reflected on the page**
  - This is critical for recovery

What are Log Records?

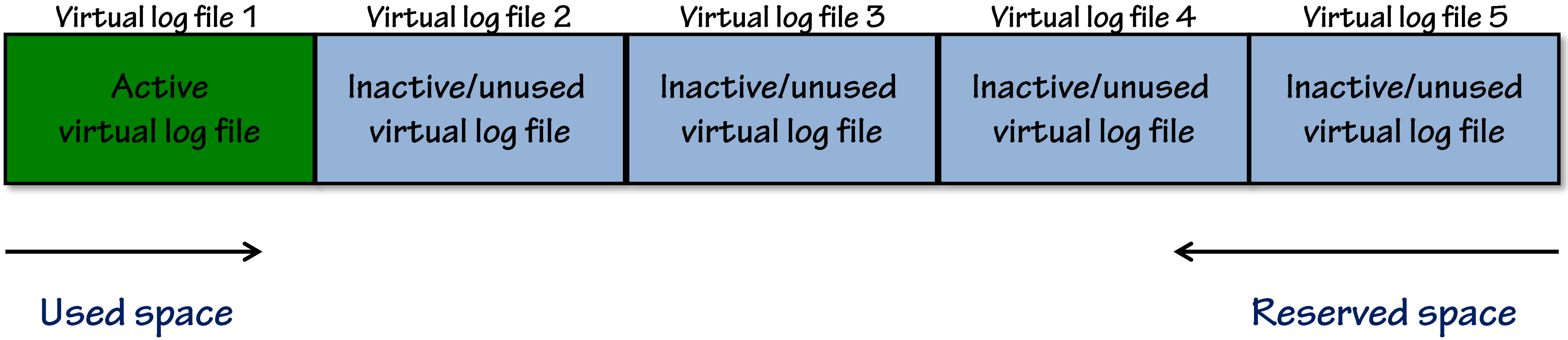
- A log record describes a single change in the database
- Each log record has a unique Log Sequence Number (LSN)
- Log records for concurrent transactions are intermingled in the transaction log according to when they occurred in time
- Log records are stored in log blocks in the buffer pool until they are flushed to disk
- There are NO non-logged operations in user/system databases
  - In tempdb, version store and workfile operations are non-logged
- Log records never move in the transaction log

Log Record Contents

- **Information in a log record allows it to be redone (rolled-forward) or undone (rolled-back)**
  - Crucial for allowing transactions to be rolled back, and for recovery to work
- **Log records contain many fields, depending on the type of log record**
- **There are some fields common between all log records, including:**
  - The log record type
  - The context of the log record, if any
  - The transaction ID the log record is part of, if any
  - The log record length
  - The LSN of the previous log record in the same transaction, if any
  - The amount of log space reserved in case the log record must be undone

Log Space Reservation

- Every log record that is generated in the forward part of a transaction must reserve free space in the transaction log to allow the log record to be rolled back, without the transaction log having to grow
- The log space reservation mechanism is very conservative, always reserving enough space, and usually more, just in case an unexpected situation occurs
  - E.g. a BEGIN TRAN operation immediately reserves 8.5KB as a fudge-factor



Log Record Types

- **There are many types of log records, including:**
  - LOP_FORMAT_PAGE
  - LOP_MODIFY_ROW
  - LOP_SET_BITS
  - LOP_INSERT_ROWS
  - LOP_DELETE_ROWS
  - LOP_SET_FREE_SPACE
- **Log records that change table/index pages include:**
  - The allocation unit ID
  - The page ID and slot ID on the page
  - The after-image, or the before-image and after-image of the changed data
    - There may be multiple sets of these in a single log record
    - After-images allow redo to occur
    - Before-images allow undo to occur

Lock Logging

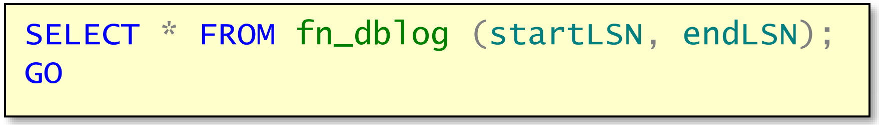
- **Some log records include a bitmap of which locks were held when the described change took place**
  - Count of the number of locks
  - What type and mode of lock
    - E.g. a page lock in X mode
  - What the lock is on
- **During crash recovery and database mirroring/availability group failovers, these locks will be acquired for all log records that are going to be undone**
- **This allows the fast recovery feature in Enterprise Edition**

Log Records in Transactions

- **Multiple log records are generated by transactions, always in the sequence:**
  - LOP_BEGIN_XACT
    - This includes information like the SPID, transaction name, start time
    - All transactions started by SQL Server are named to describe the operation
    - E.g. AllocFirstPage, DROPOBJ
  - Other records…
  - LOP_COMMIT_XACT (if the transaction commits)
  - LOP_ABORT_XACT (if the transaction rolls back)
    - These both include the end time
- **Log records in a transaction are linked together backwards by LSN**
  - This allows the transaction to be rolled back correctly
- **Some log records are non-transactional, including:**
  - PFS free space changes (impossible to reconcile with other transactions)
  - Differential bitmap changes (one-way change only)

Examining Log Records

- **Log records are most easily examined using the fn_dblog table-valued function**



```
SELECT * FROM fn_dblog (startLSN, endLSN);
GO
```

- **It is very powerful:**
  - Returns a tabular result set that can easily be manipulated
  - Allows complex predicates to be used
  - It scans all transaction log in the active portion of the log
    - From the start of the oldest uncommitted transaction to the most recent log record
    - This can be over-ridden using trace flag 2537
  - The startLSN and endLSN fields are usually passed as NULL

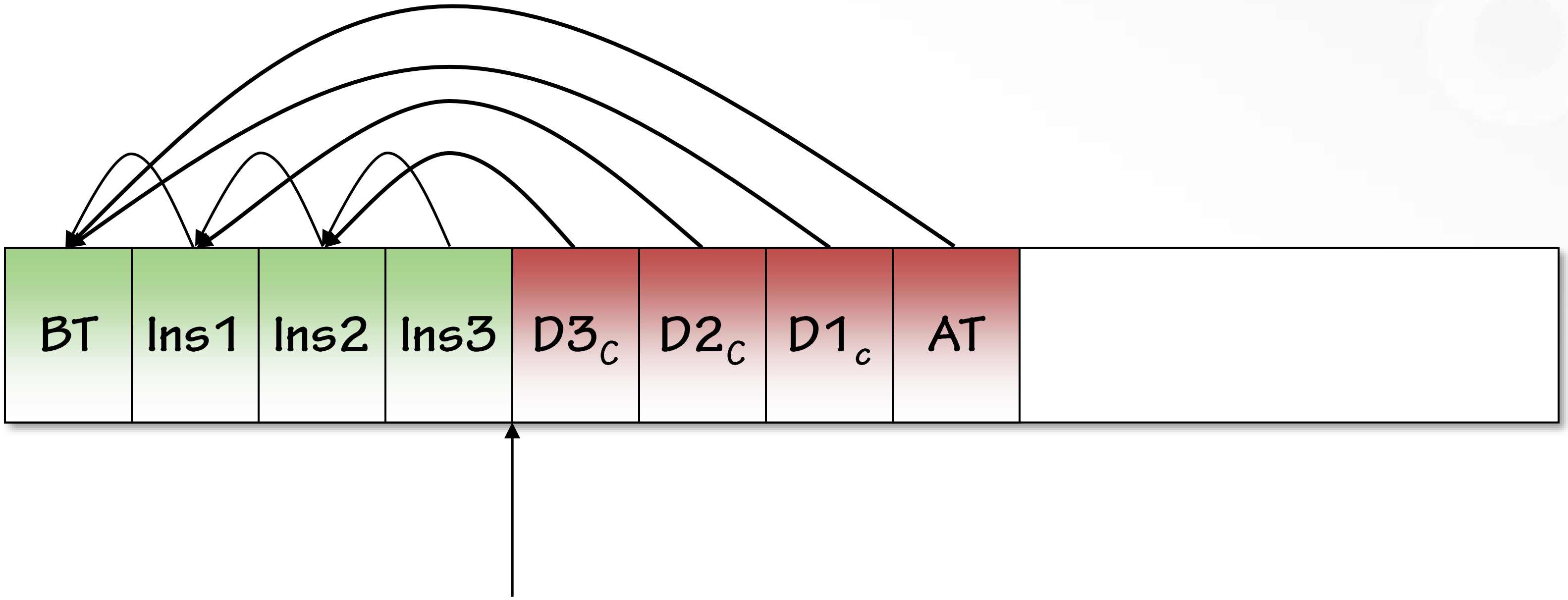
Demo

Examining log records with fn_dblog

COMPENSATION Log Records

- **When a transaction rolls back, the change described by each log record in the transaction must be undone in the database**
- **Rollback starts with the most recent log record for the transaction and follows the previous LSN links until the LOP_BEGIN_XACT log record**
- **For each log record:**
  - Perform the 'anti-operation' that will negate the effects of the log record
  - Generate a log record, marking it as a COMPENSATION log record
    - As it is compensating for the log record in the forward part of the transaction
  - The COMPENSATION log record's previous LSN points to the log record prior to the one it is compensating for
    - I.e. it essentially causes the log record to no longer be part of the chain of log records for the transaction
  - The reserved log space for the log record is released
- **COMPENSATION log records cannot be undone, only redone**

Rolling Back a Transaction



```
ROLLBACK TRAN;
GO
```

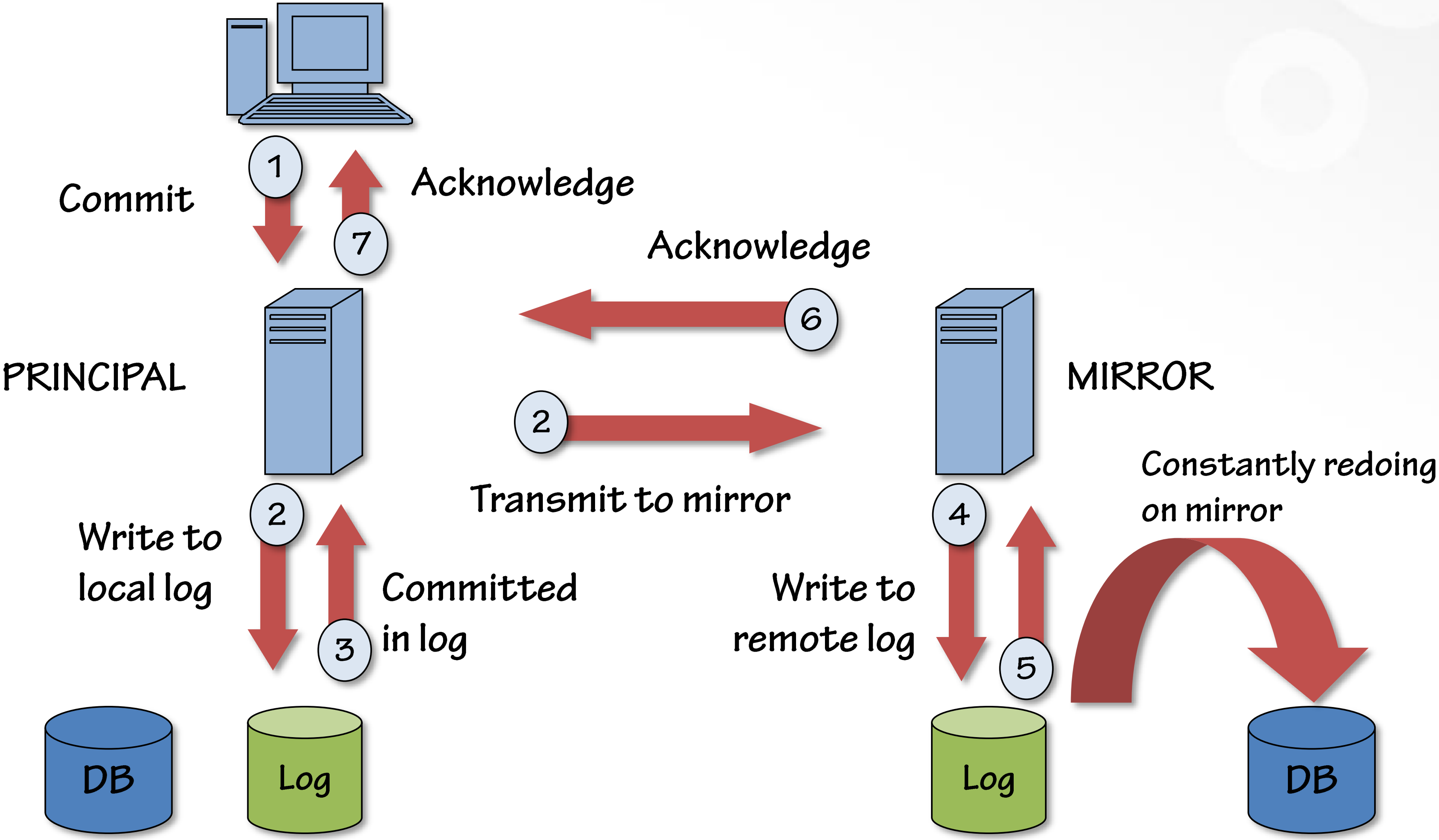
Example Data Modification

- **User/app sends an UPDATE query**
  - Update is highly selective (say, 5 rows on 3 data pages)
  - No transaction specified – implicit transaction
- **The three data pages are read into the buffer pool**
- **All necessary locks are acquired**
  - 1 intent-exclusive table-level lock
  - 3 intent-exclusive page-level locks
  - 5 update row-level locks
- **For each row:**
  - Update lock is converted to exclusive lock
  - Change is made on the data page in memory
  - Log record is generated describing the change

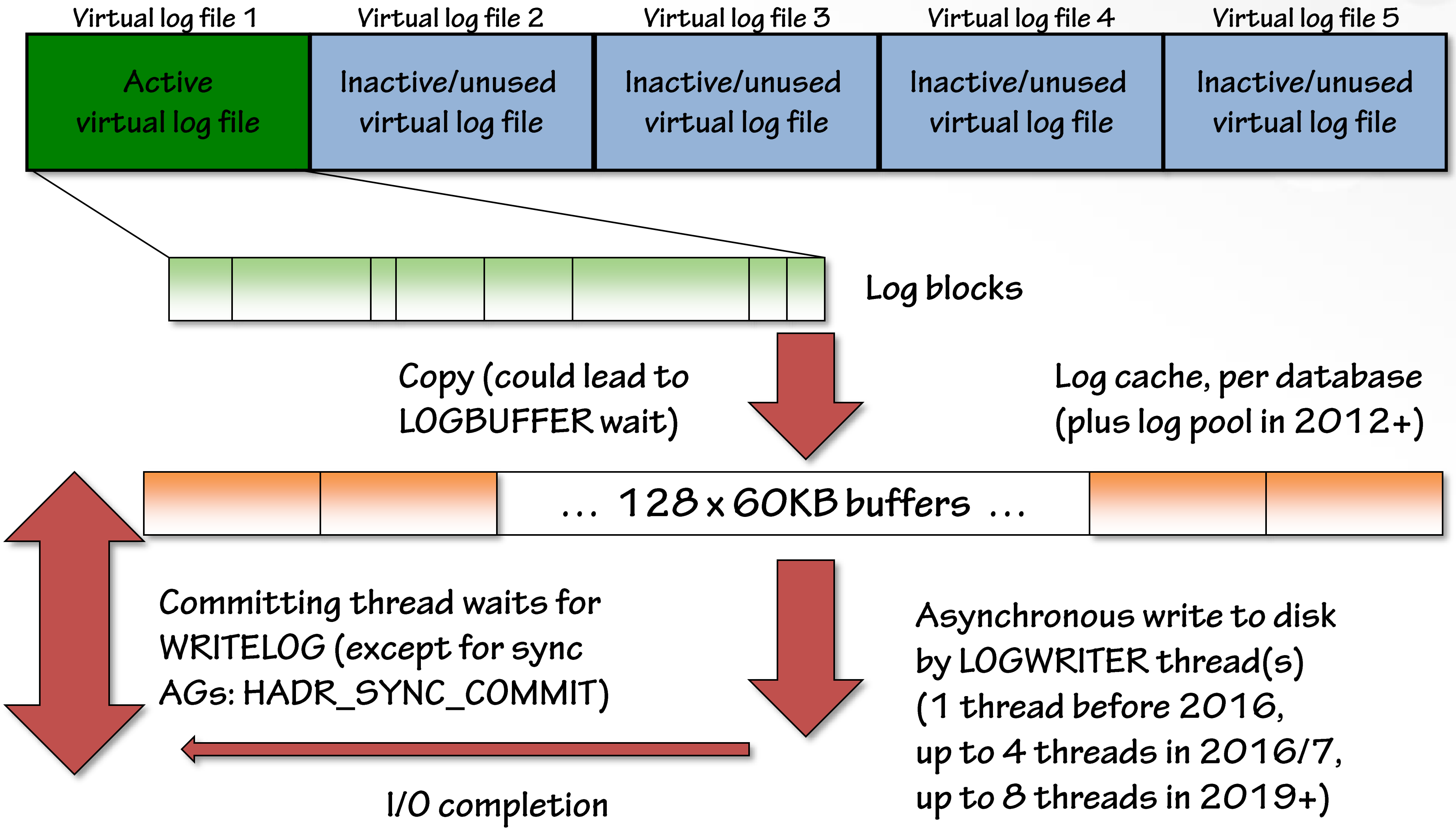
Example Data Modification

- **At this point the updates are complete**
- **The transaction is ready to commit**
- **Steps are:**
  - Ensure all log records generated by the transaction are forced write-through to the transaction log file on disk
    - This forces all of the transaction log up to the point of the LOP_COMMIT_XACT log record to be written to disk, regardless of which transaction it is for
    - Write-through is by design and cannot be over-ridden
    - Doesn't happen if the transaction is delayed durable in SQL 2014+
  - (Wait for synchronous mirroring/Availability Group if necessary)
  - Release all locks held by the transaction
  - Acknowledge the commit to the user/application:
    - (5 Rows Affected)

Log Flush to Synchronous Mirror or AG Replica



Transaction Log Flushes



Virtual log file 1

Active virtual log file

Virtual log file 2

Inactive/unused virtual log file

Virtual log file 3

Inactive/unused virtual log file

Virtual log file 4

Inactive/unused virtual log file

Virtual log file 5

Inactive/unused virtual log file

Log blocks

Copy (could lead to LOGBUFFER wait)

Log cache, per database (plus log pool in 2012+)

… 128 x 60KB buffers …

Committing thread waits for WRITELOG (except for sync AGs: HADR_SYNC_COMMIT)

Asynchronous write to disk by LOGWRITER thread(s) (1 thread before 2016, up to 4 threads in 2016/7, up to 8 threads in 2019+)

I/O completion

Log Tail Caching in SQL Server 2016+

- 2016+ has ability to have tail of the log (the blocks being written to) stored in a file on an NVDIMM drive on Windows 2016
  - By adding a second log file that is detected to be on an NVDIMM drive
- Log blocks are written to in memory and mirrored on the NVDIMM
- When a transaction commits, threads consider the log block hardened and do not need to wait for a flush to occur
- Blocks are written from the NVDIMM file to the log file when full
- If crash, log tail reconstructed from NVDIMM file before crash recovery
  - Beware, there's only one NVDIMM so potential single point of failure
  - Not a problem for tempdb
- Provides huge perf boost for workloads bottlenecked on log flushing
  - Even more of a boost than Delayed Durability
- See https://sqlskills.com/p/038 for more details on how to enable it

The Transaction has Committed… Now What?

- **All the log records for the transaction have been 'hardened' on disk in the transaction log**
  - The transaction is durable
  - The effects of the transaction can be replayed in the event of a crash
- **The data-file pages with the changes on them are still in memory only**
  - These are called 'dirty' pages
- **Why are the changed data-file pages not written to disk when the transaction commits?**
  - They do not need to be written to disk as the description of the changes applied to them is already on disk in the transaction log
  - Eventually they will be written to the data files to bring the data files up-to-date with what is in the transaction log
- **Changed data-file pages are written to the data files periodically by a mechanism called 'checkpoint'**

Why Do Checkpoints Exist?

- **To reduce crash-recovery time**
  - By having as many up-to-date data-file pages in the database as possible, this reduces the amount of redo that must be performed
- **To batch I/Os to disk and improve performance**
  - Imagine 1000 update transactions to a single data page
  - Is it more efficient to write the data page image to disk after each change (i.e. 1000 times) or just once, during a checkpoint?

Clean vs. Dirty Pages

- **The buffer pool maintains a set of data-file pages in memory**
- **Each page has a control structure (called a BUF) associated with it that tracks 'state' for the page, including:**
  - The database ID the page is part of
  - The amount of free space on the page
  - Information to support the LRU (Least-Recently-Used) algorithm
  - Whether the page is clean or dirty
  - A latch structure for in-memory access protection
- **'Clean' page <u>has not</u> been changed since last read from/written to disk**
- **'Dirty' page <u>has</u> been changed and changes are not on disk**
- **This information can be seen using sys.dm_os_buffer_descriptors**
- **Checkpoints are concerned with dirty pages**

Checkpoint Mechanism

- **A checkpoint writes ALL dirty pages of a database to disk**
  - Pages are marked as ‘dirty’ as soon as they are changed
  - It doesn’t matter if the transaction that made the change is committed or uncommitted at the time of the checkpoint
  - Pages have their ‘dirty’ bit cleared when written during a checkpoint
- **Steps taken are:**
  - Log that a checkpoint started, and any necessary checkpoint information
  - Write all dirty pages for the database to disk, flushing log records if necessary
    - For non-indirect checkpoints, scan all BUFs for dirty pages
    - For indirect checkpoints, only flush known dirty BUFs from special list
  - Write the LSN of the checkpoint in the boot page of the database in the dbi_checkptLSN field
  - If in the SIMPLE recovery model, try to clear the log (discussed later)
  - Log that a checkpoint ended

Checkpoint Mechanism (2)

- **Checkpoints can occur in parallel for multiple databases**
  - SQL Server 2000 was limited to one checkpoint at a time
- **Buffer pool does gather-writes of up to 32 contiguous, dirty pages (see https://sqlskills.com/p/039) and up to 128 on 2016+**
- **Checkpoint throttles I/O if I/O latency exceeds 20ms (50ms on 2016+)**
  - During shutdown, the throttling threshold increases to 100ms
  - More in-depth explanation at https://sqlskills.com/p/040
- **Documented "-kXX" startup option can be used to set the checkpoint I/O rate at XX MB/s**
- **Note that for minimally-logged operations, data-file pages are written out continuously as they are allocated and formatted with the data on**
  - This is called 'eager writing' and has same sizes as for checkpoint writes
  - Minimally-logged operation cannot commit until all eager writing is completed otherwise the transaction would not be durable

Checkpoint and the Log

- **When a data-file page is written to disk by a checkpoint, write-ahead logging guarantees that all log records affecting that page must be written to the transaction log on disk first**
- **All log records up to and including the last one that affected the page are written out, regardless of which transaction they are part of**
  - Cannot just flush all current log records at the start of the checkpoint as the checkpoint could take a while and a data-file page may have more log records affecting it before the checkpoint reaches it to write it to disk
- **This means log records are written out in three ways (plus next slide):**
  - When any transaction commits
    - All log records up to and including the commit transaction log record
  - When a data-file page is written to disk
    - All log records up to and including the last log record to affect the page
    - This may also happen from lazywriter activity too
  - When a log block hits the maximum size of 60KB and is forcibly ended

SEQUENCE Objects

CREATE SEQUENCE dbo.Seq1 AS BIGINT MINVALUE 1 CACHE 50

- **Current SEQUENCE value is persisted in the log for crash recovery**
- **If NO CACHE is used, every NEXT VALUE causes a log block flush**
  - Don’t do this!
- **Otherwise, using CACHE x means a log block flush every x uses of NEXT VALUE**
  - Default value for CACHE is 50
  - Be careful not to choose too small a number
  - Balance between log block flushes and your desired recovery of SEQUENCE values if a crash occurs

Types of Checkpoint: Automatic

- **Based on recovery interval in minutes (default 1 minute)**
  - Under SIMPLE recovery model, also when log is 70% full
- **Checkpoint will complete in time that minimizes impact to performance (via I/O throttling discussed earlier)**
- **Tracked using Buffer Manager: Checkpoint pages/sec**
- **Can be extremely slow for systems with very large buffer pools as scans all buffers for database looking for dirty ones**
  - E.g. when creating a new database, adding a file, VDI-based backup

Types of Checkpoint: Indirect

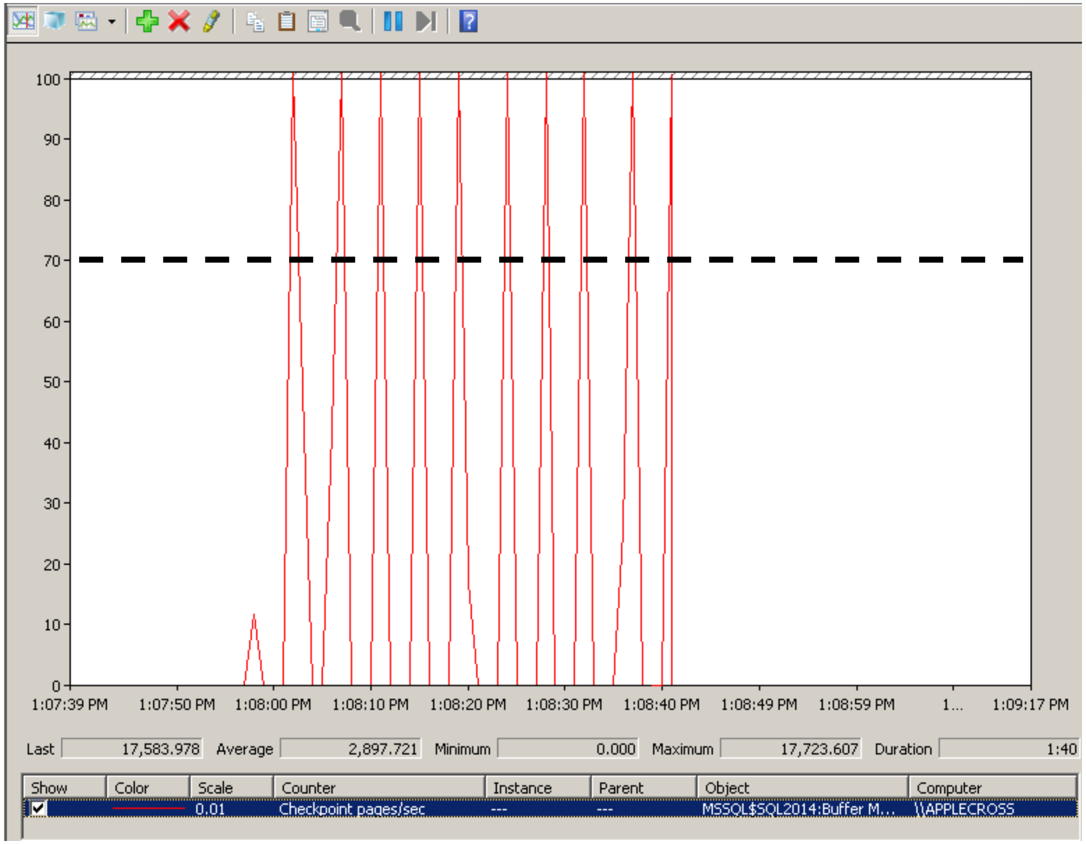
- **Default for new databases and tempdb for 2016+**
  - For 2012 and 2014, see https://sqlskills.com/p/042
    - Install CU, enable indirect checkpoints, enable TF 3449 to get improvement
  - More info in blog post at https://sqlskills.com/p/043
- **Finer-grained control of recovery time and stops periodic I/O spikes**
  - Think of it as a kind of ongoing, rolling checkpoint
- **Set using *ALTER DATABASE … SET TARGET_RECOVERY_TIME***
  - Takes precedence over recovery interval setting and is per-database
- **Faster than automatic checkpoints as it scans list of known dirty pages**
- **Tracked using Buffer Manager: Background writer pages/sec**
- **Can cause bottleneck with very hot tempdb – see https://sqlskills.com/p/041 (fixed in latest builds of 2016 and 2017)**

Types of Checkpoint (2)

- **Internal checkpoints**
  - Performed when:
    - Data files added or removed
    - A database shutdown occurs (for whatever reason)
    - Database snapshot created
    - Database backup performed (full or differential)
- **Manual checkpoints**
  - Using the CHECKPOINT command
  - CHECKPOINT supports a duration parameter
    - CHECKPOINT [checkpoint_duration]
    - checkpoint_duration is an integer used to define the amount of time in which a checkpoint should complete
    - Governs how many resources are assigned to checkpoint operation
  - When not specified checkpoint will complete in the time that minimizes impact to performance

Checkpoint I/O Spikes

- The I/O spike from traditional checkpoints might overload the I/O subsystem



Last 17,583.978 Average 2,897.721 Minimum 0.000 Maximum 17,723.607 Duration 1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
☑	🟥	0.01	Checkpoint pages/sec	---	---	MSSQL$SQL2014:Buffer M...	\\APPLECROSS

Checkpoint Monitoring

- **It can be useful to correlate checkpoints occurring with spikes in I/O so that changes can be made to specific database (or the I/O subsystem) to alleviate the I/O spike if it overloads the I/O subsystem**
  - For instance, doing more frequent, manual checkpoints, or configuring a lower recovery interval on SQL Server 2012+ with indirect checkpoints
  - This will produce a more constant I/O load without high spikes that overload the I/O subsystem
  - However, the root cause may be more I/O being performed because of a change somewhere so do not just accept sudden increase in checkpoint activity without investigating why it occurred
- **Buffer Manager/Checkpoint pages/sec counter is not database specific so identifying which database is involved requires trace flags or extended events**
  - Remember this doesn't track indirect checkpoints

Checkpoint Monitoring (2)

- **Trace flags:**
  - Trace flag 3502 writes messages to the error log about which database a checkpoint is occurring for
  - Trace flag 3504 writes more detailed information about how many pages were written out and the average write latency
  - Trace flags 3502 and 3504 can be used together
  - Trace flag 3605 is also required otherwise no messages are printed
- **These trace flags are safe to use in production for a limited time**
  - All they do is print messages in the error log
- **In SQL Server 2012+, the info printed by 3504 is also printed if a checkpoint takes longer than the recovery interval**
- **Extended events:**
  - checkpoint_begin and checkpoint_end events

Demo

Monitoring automatic checkpoints

Tempdb Behavior

- **Tempdb is recreated on each instance restart so no crash recovery is ever run**
- **This means:**
  - SIMPLE recovery model (unchangeable)
  - Redo information is not included in transaction log records
  - Log records are not flushed to disk on transaction commit
  - Checkpoints do not occur based on recovery interval
    - They're triggered when the log file becomes 70% full
  - Checkpoints don't always flush tempdb data pages to disk (varies by version)
- **Changes to tempdb must be logged to allow transactions to roll back**
- **Beware of indirect checkpoint for tempdb**
  - Can cause bottleneck – see https://sqlskills.com/p/041

SQL Server 2014+: Delayed Durability (1)

- **Ability of transaction commit not to cause a log flush**
  - See MSDN at https://sqlskills.com/p/044
- **Trades off reduction in log contention and I/O against possibility of data loss in the event of a crash through less frequent log flush I/O**
- **Log is flushed when:**
  - Internal log buffer fills up to 60KB, or
  - A durable transaction commits, or
  - You execute sp_flush_log, or
  - 1ms internal timer expires and I/O subsystem is not overloaded
- **Must be configured at the database level:**
  - ALTER DATABASE … SET DELAYED_DURABILITY =
    - DISABLED: the default, all transactions are durable
    - ALLOWED: transaction are delayed-durable if set for the transaction
      - DELAYED_DURABILITY = OFF | ON (OFF is the default)
    - FORCED: all transactions are delayed-durable regardless of per-transaction setting

SQL Server 2014+: Delayed Durability (2)

- **There is a risk of data loss, so be careful when using this**
  - If you're allowed any data loss, consider whether it can help
  - If not, be extremely wary about enabling this feature
- **Synchronous mirrors and Availability Groups behave differently:**
  - When the delayed durable transaction commits, it DOES NOT WAIT for the log records to be hardened on the mirror/secondary
  - 'Synchronous' only applies to fully durable transactions
- **Log backups behave differently:**
  - A log backup only backs up transactions that are durable on disk at the time of the backup
- **Not supported with transactional replication**

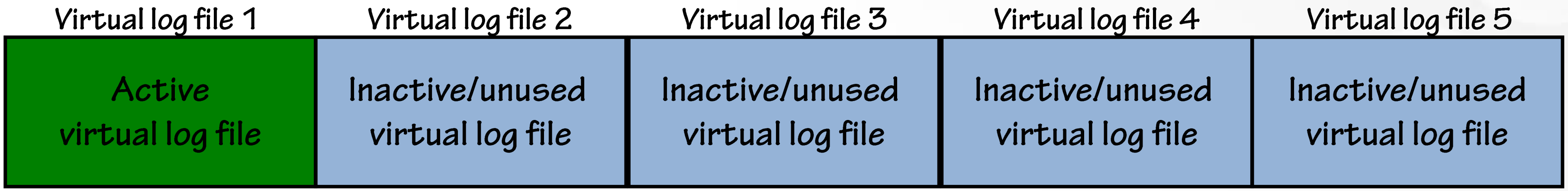
Demo

Delayed durability effect on performance

SQL Server 2014+: In-Memory OLTP/Hekaton

- **Gives the ability to have tables stored entirely in memory, with optional logging to give durability, and very fast performance**
  - See comprehensive MSDN at https://sqlskills.com/p/045
  - Test: 1 million individual inserts in 7 seconds compared to 6 minutes
- **Can be complicated feature requiring a lot of development work**
  - Depends what you want to use it for
- **Where we've seen it used successfully so far with clients:**
  - Intermediate load database during large ETL processes
  - Configured with non-durable in-memory tables for extremely fast load times
  - Alleviating PAGELATCH_EX contention (insert hotspots)
  - Replacing temporary table usage
  - Tempdb in-memory system tables in 2019+
- **Advice: read through the comprehensive MSDN section**
  - Also whitepapers at https://sqlskills.com/p/046, https://sqlskills.com/p/047

Making a VLF Active



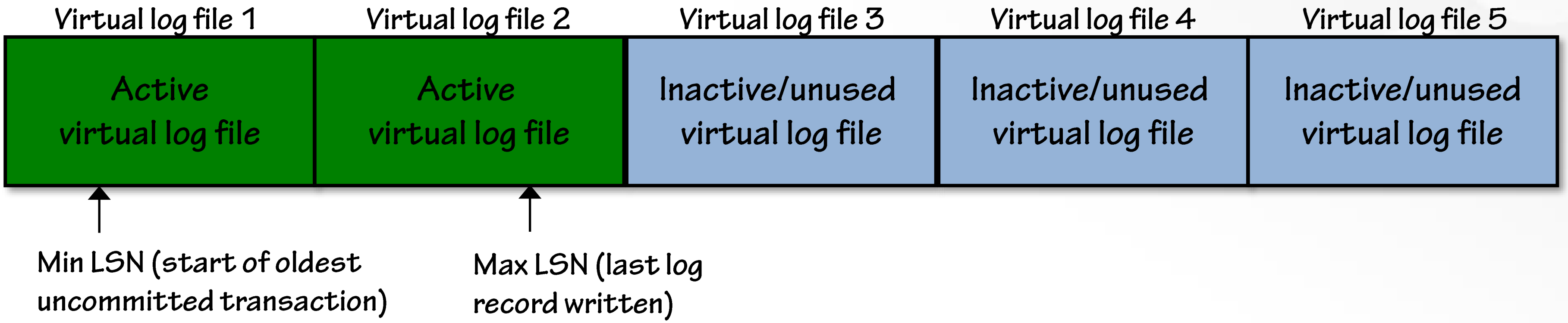
Virtual log file 1	Virtual log file 2	Virtual log file 3	Virtual log file 4	Virtual log file 5
Active virtual log file	Inactive/unused virtual log file	Inactive/unused virtual log file	Inactive/unused virtual log file	Inactive/unused virtual log file

- **When a transaction modifies the database, a log record is written to the transaction log**
- **The first log record written to a VLF makes it become active**
  - A VLF is either wholly active or inactive
- **An active VLF cannot be overwritten until it becomes inactive**
  - A VLF remains active until all log records in it are not needed for:
    - Log or data backups
    - Transactional replication
    - Change data capture
    - Database mirroring/availability groups
    - Long-running transaction rollback or crash recovery

Log Space Reservation

- **As we discussed earlier, each log record that is written to the transaction log also reserves some free space in the transaction log in case a compensation log record is required**
- **The Log Manager guarantees that all uncommitted transactions in the database can roll back without requiring the transaction log to grow**
  - As transaction log growth may not be possible, and so the database will be transactionally inconsistent and hence SUSPECT
- **The reserved log space does not make VLFs become active but will trigger transaction log auto-grows if there is no more space**
  - As an analogy, consider your bank account with $1,000 in it
  - If you know there's the possibility of having to make a $300 payment on Wednesday, you cannot spend more than $700 until Wednesday passes
  - If Wednesday passes and there was no payment, the $300 is now available
  - The reserved log space behaves in the same way in that as soon as the transaction commits, the reserved space is discarded and is available for use

Moving Through the Transaction Log

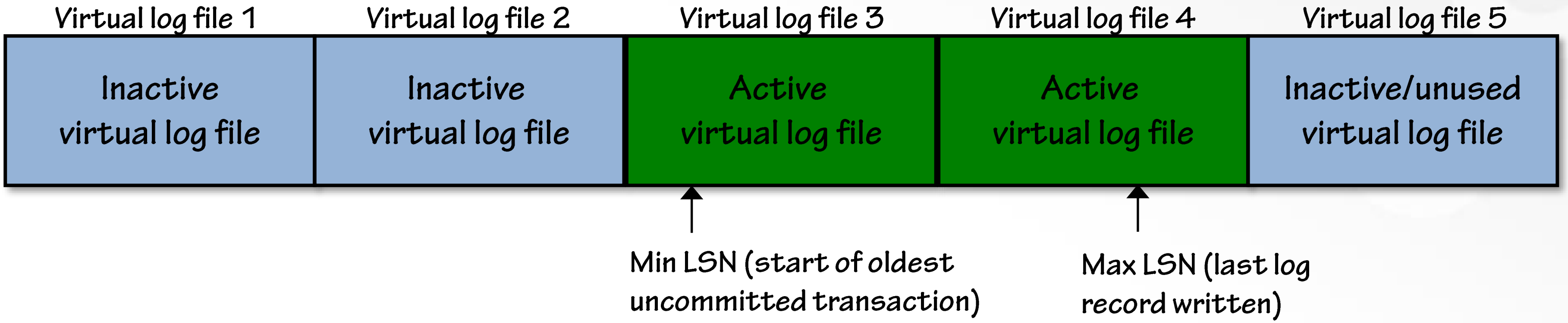

Virtual log file 1
Virtual log file 2
Virtual log file 3
Virtual log file 4
Virtual log file 5
Active virtual log file
Active virtual log file
Inactive/unused virtual log file
Inactive/unused virtual log file
Inactive/unused virtual log file
Min LSN (start of oldest uncommitted transaction)
Max LSN (last log record written)

- **As more log records are written, more VLFs become active**
- **SQL Server tracks:**
  - Which portion of the transaction log is still required
  - The start of the oldest uncommitted transaction
  - The most recent log record written
- **The oldest uncommitted transaction defines the oldest active LSN and therefore the oldest active VLF**

Tracking Uncommitted Transactions

- **DBCC OPENTRAN returns the oldest uncommitted and unreplicated transaction**
- **sys.dm_tran_database_transactions returns all uncommitted transactions for all databases**
  - Can see how much transaction log space has been reserved for a transaction
  - Can see how much total transaction log space is required for a transaction
  - Can join with other DMVs to get more comprehensive information
  - Blog post with a script: https://sqlskills.com/p/048

Transaction Log Clearing



Virtual log file 1 | Virtual log file 2 | Virtual log file 3 | Virtual log file 4 | Virtual log file 5

Inactive virtual log file | Inactive virtual log file | Active virtual log file | Active virtual log file | Inactive/unused virtual log file

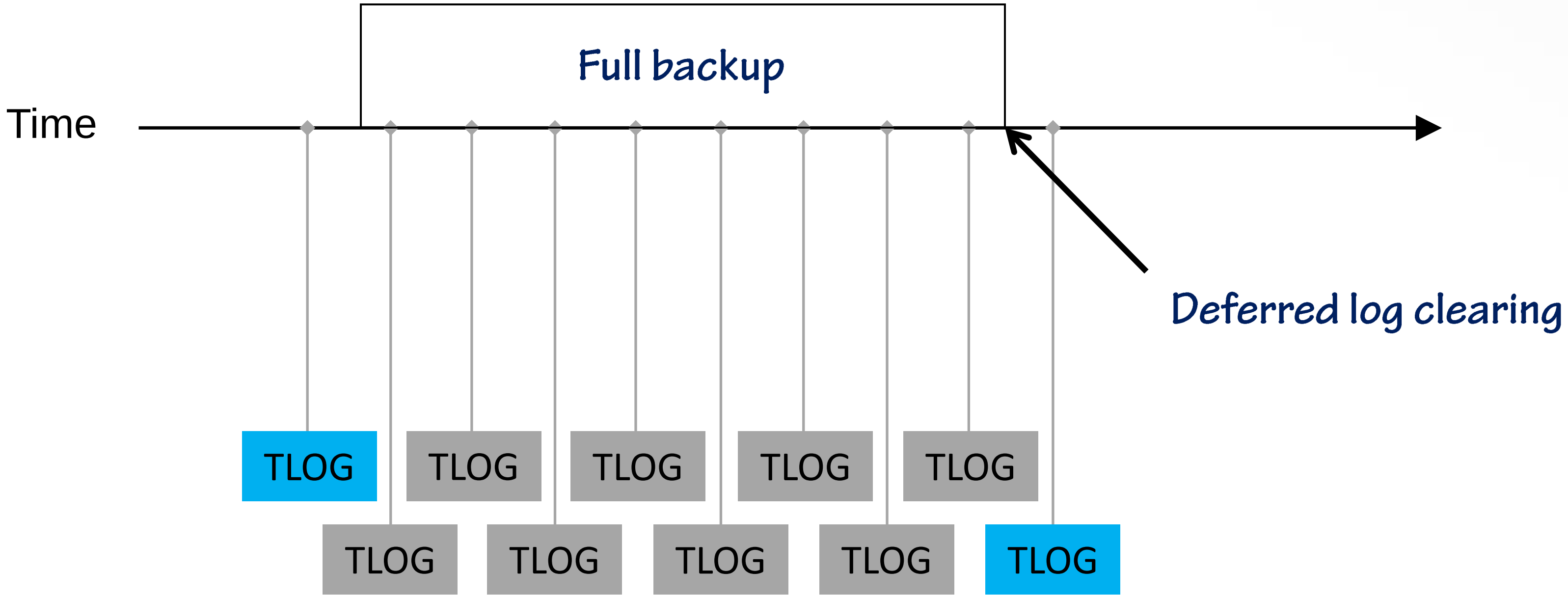
Min LSN (start of oldest uncommitted transaction)

Max LSN (last log record written)

- **A VLF can be made inactive once all log records are not required**
  - This is called clearing (or truncating) the transaction log
- **Log clearing is done by a log backup in FULL or BULK_LOGGED recovery models, or by a checkpoint in SIMPLE recovery model**
  - 2014+: Exception occurs when in-memory tables used(see slide 59)
- **All that happens is that zero or more VLFs are marked inactive**
  - Nothing is zeroed, removed, overwritten (“clearing” is a misnomer)
  - The transaction log file size does not change (“truncating” is a misnomer)
- **If log clearing does not occur, the transaction log will grow forever**

Concurrent Log and Data Backups

- Log backups that occur while a full or differential backup are running cannot clear the log
- Deferred log clearing happens at the end of the data backup



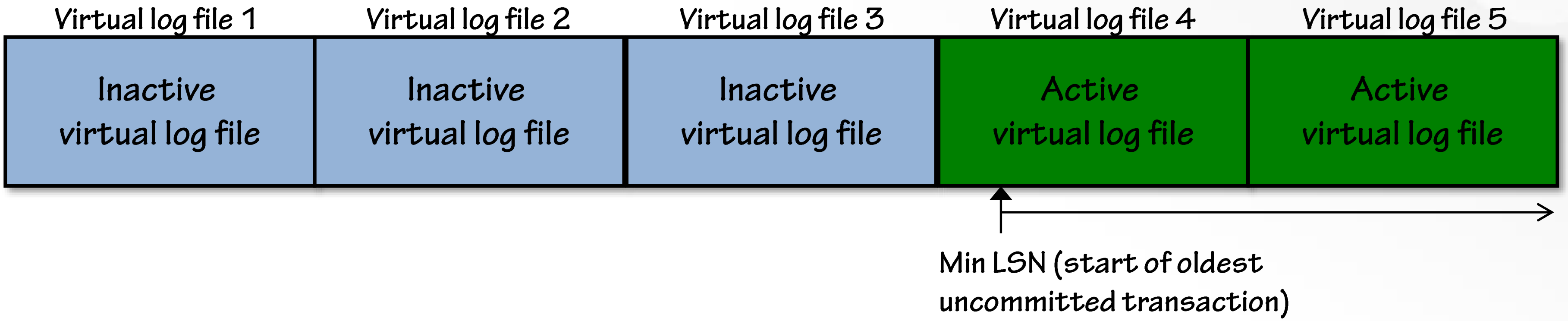
Tracking Transaction Log Space Usage

- **Using Performance Monitor, from the Databases performance object:**
  - Log File(s) Size (KB)
  - Log File(s) Used Size (KB)
  - Percent Log Used
  - Log Growths
  - Log Shrinks
- **DBCC SQLPERF (LOGSPACE)**
  - Returns information for all databases including the size and the percentage used
  - Processing of the output requires creating a table and then using INSERT … EXEC to capture the results
- **sys.dm_db_log_space_usage**
  - Returns information for the current database only

DBCC LOGINFO

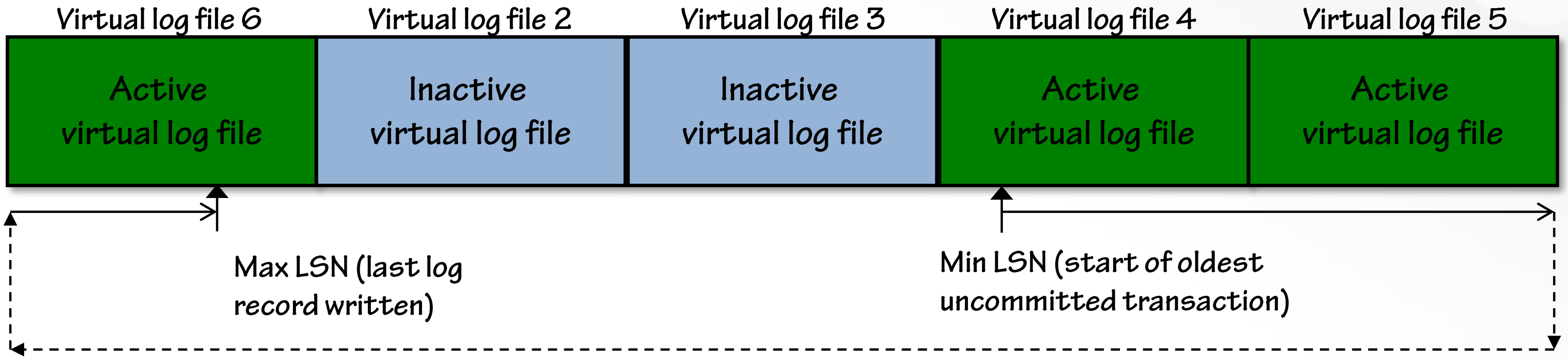
- **Undocumented DBCC LOGINFO command to examine VLFs in all versions of SQL Server**
  - dbcc loginfo [({'dbname' | dbid})]
- **It returns a result set with one row per VLF giving info including:**
  - FileSize: VLF size, in bytes
  - FSeqNo: the VLF sequence number
  - Status: whether the VLF is active or not (0 = inactive, 2 = active, 1 is not used)
    - Also 4 = ghost VLF on an AG secondary that hasn't yet replayed a log growth
  - CreateLSN: the LSN when the VLF was created (0 = the VLF was created when the transaction log file was initially created)
- **Also new DMVs, but only in SQL Server 2016 SP2+**
  - sys.dm_db_log_info – info on VLFs, the same as DBCC LOGINFO
  - sys.dm_db_log_stats – size and usage info for the log
  - These two DMVs allow easier programmatic monitoring of the log

Circular Nature of the Transaction Log



- Once the end of the transaction log is reached, it would like to wrap around and start using the first VLF again, as long as log clearing has occurred, without having to grow the transaction log file
- The log manager checks the active status of the first VLF in the transaction log

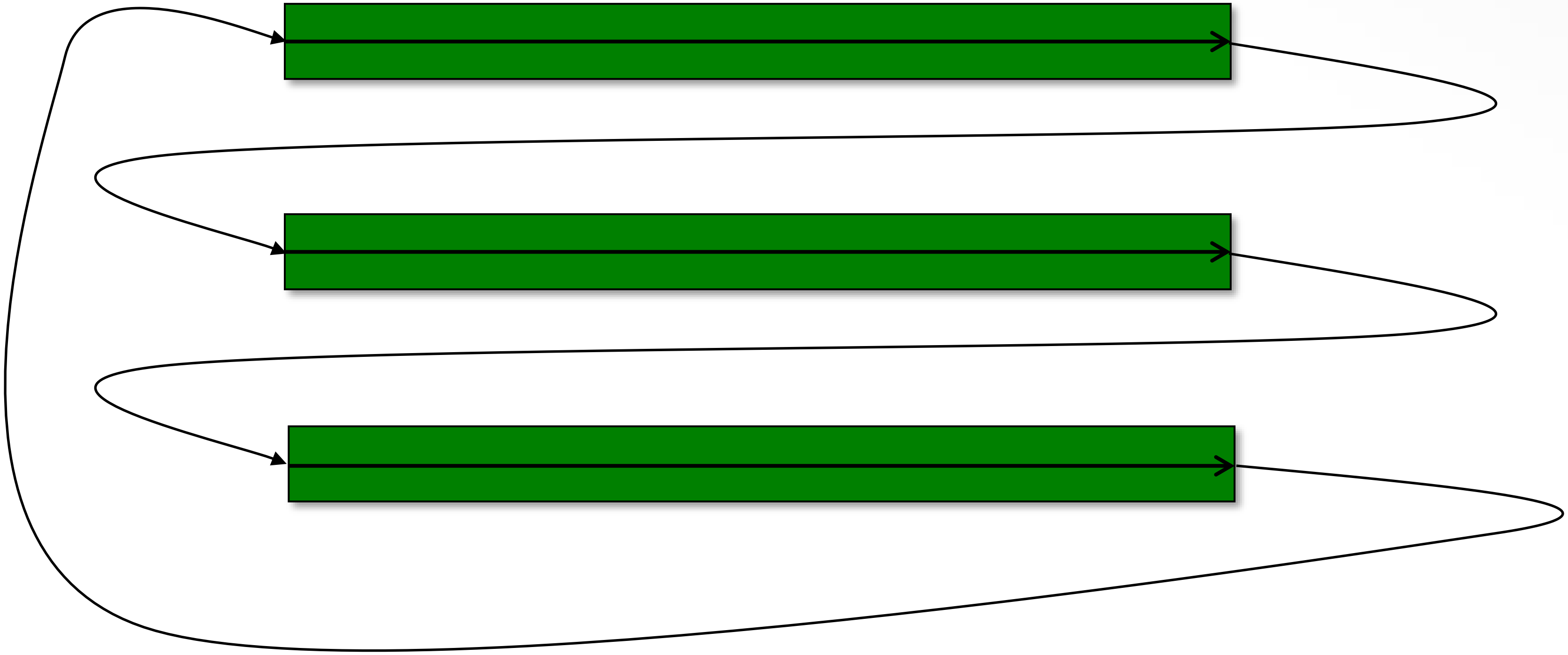
Circular Nature of the Transaction Log (2)



- **If the first VLF is inactive, the transaction log can wrap**
  - Note that VLF 1 is reactivated and becomes VLF 6
- **Log records continue to be written**
- **This post has an example: https://sqlskills.com/p/049**

Wrapping with Multiple Files

- If multiple transaction log files are configured, they will be used in order before wrapping back to the start of the first file



Demo

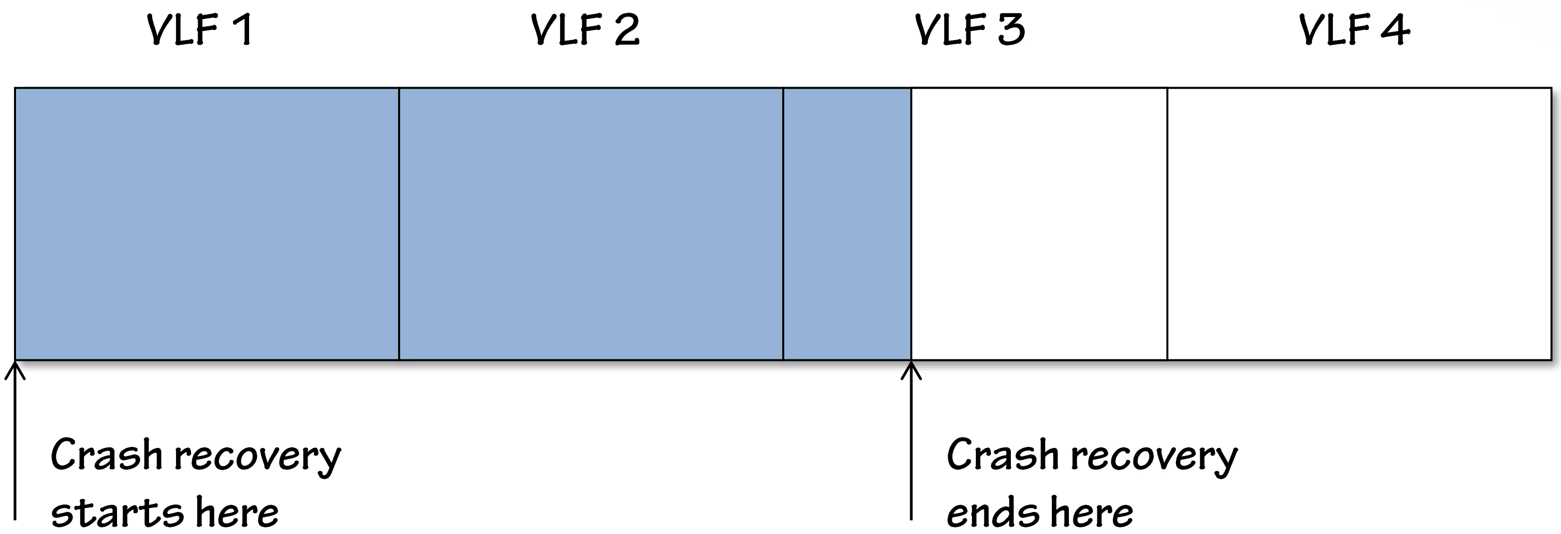
Circular nature of the log

Crash Recovery

- **For each database:**
  - The boot page is examined to get the LSN of the most recent checkpoint
  - The checkpoint log records are examined to get the list of uncommitted transactions at the time the checkpoint occurred
  - Recovery starts from the LSN of the LOP_BEGIN_XACT log record of the oldest uncommitted transaction at the time of the most recent checkpoint
- **Three passes are made through the transaction log:**
  - First pass examines the log records to see which data-file pages will be required, and these pages are read into the buffer pool
    - Also builds a list of committed and uncommitted transactions and discovers the 'end' of the log (described on the next two slides)
  - The second pass does redo of log records for committed transactions
  - The third pass does undo of log record for uncommitted transactions
- **Track recovery in 2016+ with XEvents: see https://sqlskills.com/p/050**

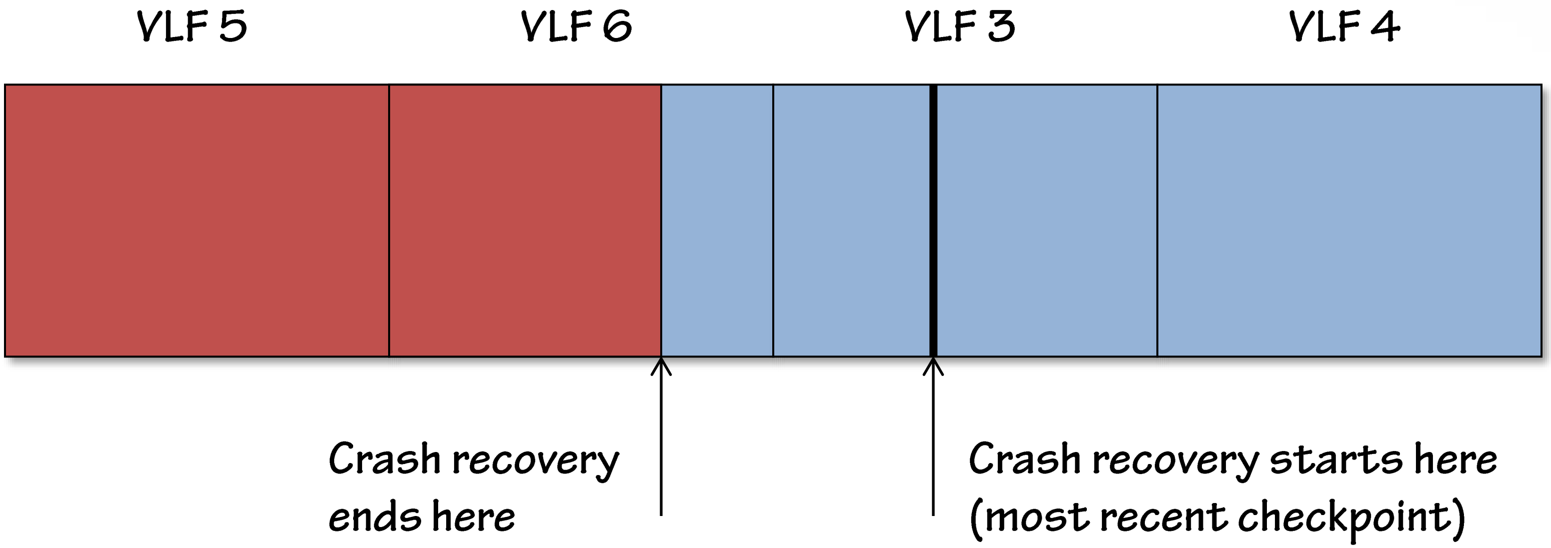
Finding the End of the Transaction Log

- In this case, the transaction log has not wrapped, so as part of the first pass through the transaction log, crash recovery will encounter a 512-byte block filled with 0x00s (or 0xC0s in 2016+)
  - The white section of VLF 3
- All log blocks that have been written will have parity bits 64



Finding the End of the Transaction Log (2)

- **In this case, the transaction log has wrapped, so as part of the first pass through the transaction log, crash recovery will encounter a 512-byte block with the old parity bits**
  - The blue section of VLF 6 will have parity bits 64, whereas the VLF header for VLF 6 states that the current parity bits are 128

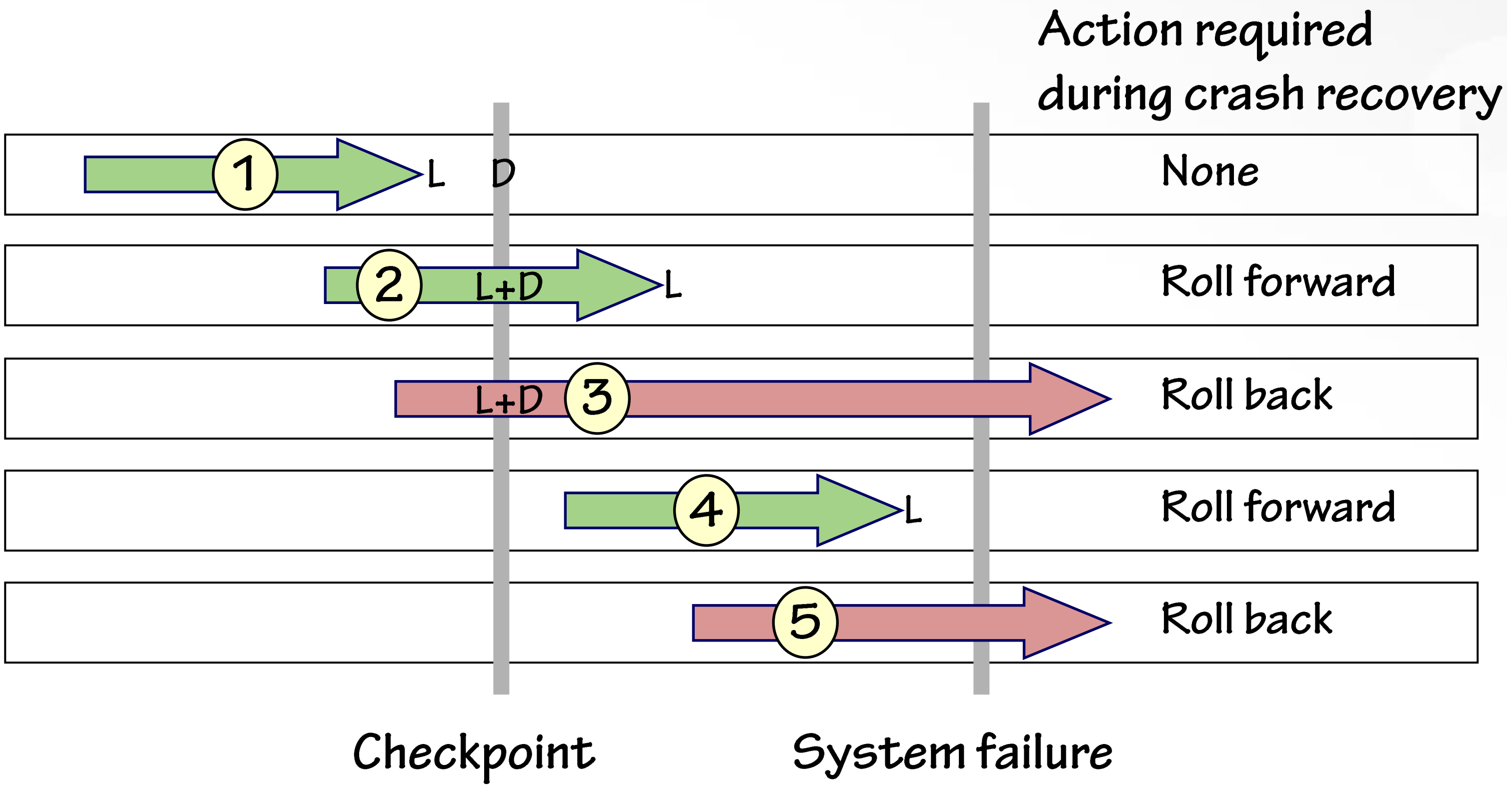


notes for prior slide

Where Does Crash Recovery Stop?

- **Crash recovery does NOT know what the most recent log record is**
- **It continues reading through the transaction log, following the VLF sequence, until it encounters one of two things:**
  - A 512-byte block full of zeroes
    - Remember that the transaction log is always zero initialized when created, grown, or auto-grown
    - This is to avoid the possibility of random bytes in the file system looking like a valid log block and causing recovery to fail
  - A 512-byte block that has the parity bits of the previous incarnation of the VLF
    - Remember that the parity bits for a VLF flip back-and-forth between 64 and 128 each time the VLF is made active
    - This allows SQL Server not to have to zero out a VLF when it becomes active again
- **This mechanism is necessary to avoid SQL Server having to persist the end of the transaction log somewhere**

Crash Recovery Visualized



Each arrow is a transaction

L = Log records written to disk

D = Data file pages written to disk

Accelerated Database Recovery (ADR)

- **New feature in SQL Server 2019**
- **Targeted at systems with:**
  - Large transactions with long rollback times, and potential for downtime on crash/failover
  - Large transactions causing excessive transaction log growth

- **Basically instantaneous rollback of large transactions and aggressive log truncation, with a cost…**
- **Performance, from anecdotal evidence and testing:**
  - Inserts and deletes may be up to 10% slower
  - Updates, depending on column types, may be 2-3x slower
- **Space always taken up on the page being changed**
- **Space may also be taken up in a per-database 'version store'**
  - Possibility of page splits from enabling this – see Module 7

ADR Mechanism (1)

- **Persistent Version Store (PVS)**
  - PVS is an append-only table in the database
    - Can control which filegroup it is created in
  - If row change is small, stored on-page as a delta version otherwise stored as a row in the PVS, and creation is logged so PVS is recoverable
    - These are where the performance hit comes from
  - Different from regular versioning as PVS is crash-recovered
  - Might become large, hence ability to place on a filegroup
- **Secondary Log**
  - In-memory log stream of operations that cannot be versioned
    - E.g. system table changes, allocation bitmap updates
  - Flushed to disk on checkpoint and transaction commit

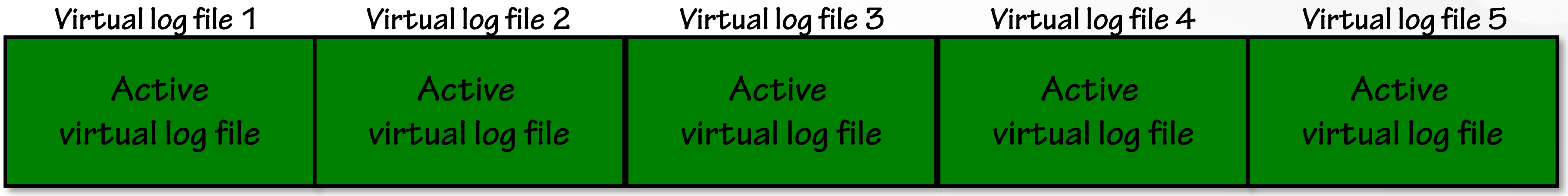
ADR Mechanism (2)

- **Aborted Transaction Map**
  - Which transactions need to be rolled back
- **Background processes:**
  - PVS cleanup and the asynchronous rollback itself (called 'logical revert')
  - Several other more targeted cleanup processes for the ATM and SLOG

- **Queries use ATM and version mechanism to determine what they see**
- **Aborting a transaction happens instantly**
- **Crash-recovery/failover appear to skip the long undo phase, so higher availability**

- **So it's a trade off, and make sure to test under load before enabling**
- **Good explanation at https://sqlskills.com/p/117**

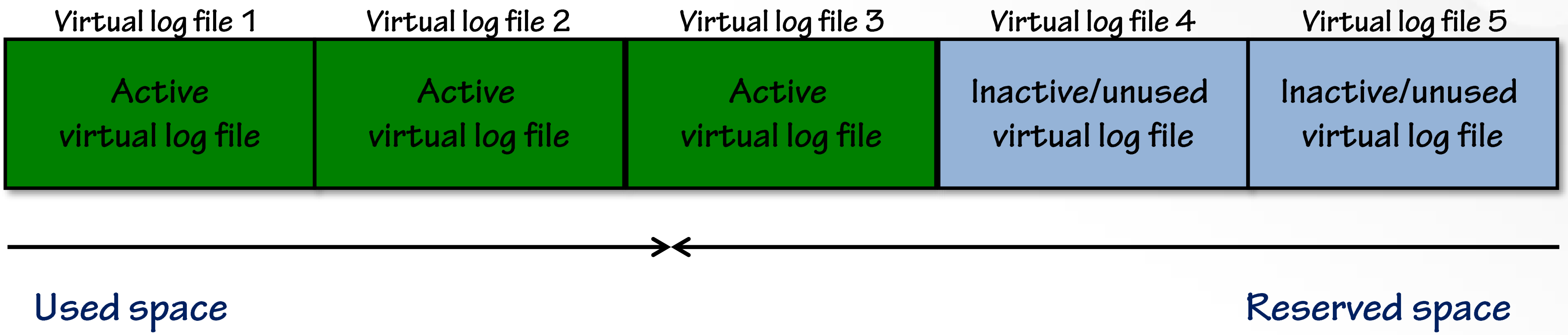
If the Transaction Log Fills Up…



Virtual log file 1	Virtual log file 2	Virtual log file 3	Virtual log file 4	Virtual log file 5
Active virtual log file	Active virtual log file	Active virtual log file	Active virtual log file	Active virtual log file

- **What if all VLFs are active and more log records are written?**
- **The transaction log will auto-grow (if configured and possible)**
  - More VLFs will be added and log records will be written into them
  - Transaction log auto-grow means a pause for zero-initialization
- **If the transaction log cannot auto-grow, write activity stops**
  - Uncommitted transactions will roll back
  - Until the transaction log is grown, another transaction log file is added, or some action is taken to clear the transaction log
- **Avoid this situation by allowing the transaction log to clear!**

If the Transaction Log Fills Up…



- What if only some VLFs are active but with a long-running transaction?
- Eventually used space + reserved space will reach the size of the log
- And then same thing as on previous slide; auto-growth or rollback

Why Did the Transaction Log Fill Up?

- **See log_reuse_wait_desc in sys.databases for why the log can't clear**
  - Examples are LOG_BACKUP, DATABASE_MIRRORING, NOTHING
    - Full list in Books Online at https://sqlskills.com/p/051
  - This is the reason why log clearing failed the last time it was attempted
    - E.g. it might be ACTIVE_BACKUP_OR_RESTORE but the backup already finished
  - Comprehensive blog post on all of them: https://sqlskills.com/p/052
- **Reason is printed as part of 9002 error in the error log**
  - *"The transaction log for database 'MyDB' is full due to 'LOG_BACKUP'"*
- **Take whatever corrective action can be done:**
  - Take a log backup (lack of backups is the #1 cause of full transaction logs!)
  - Kill a long-running transaction
    - List all transactions using code from Ian Stirk at https://sqlskills.com/p/053
  - Manually grow the existing log file(s) or add another log file (temporarily)
  - Switch to the SIMPLE recovery model as last resort, as it breaks log chain

AGs/Mirroring and Log Clearing

- AGs and mirroring complicate the log clearing mechanism
- A VLF on the primary cannot be made inactive until the equivalent VLF on all secondaries has been processed for redo on the secondaries
- Otherwise, if the VLF were made inactive and a log block was written to it on disk, it would cause the log block to be overwritten on the secondary as well, breaking redo
- This means that slow secondaries can cause primary log growth to occur

- AG secondaries also use parallel redo in 2016+, which can cause a known performance issue
  - Look for high waits on DIRTY_PAGE_TABLE_LOCK
  - Use trace flag 3459 to disable parallel redo

In-Memory OLTP and Log Clearing

- In-memory checkpoint is performed by a separate background thread
- With in-memory tables, in-memory checkpoint does not occur until 1.5GB of log has been generated since last checkpoint
- This means transaction log will not clear until 1.5GB of log has been generated since last checkpoint

- Latest builds of 2014+, log_reuse_wait_desc shows XTP_CHECKPOINT
  - Some builds of 2016/2017 didn’t, which was confusing
- More info at https://sqlskills.com/p/054

Multiple Transaction Log Files

- **If the addition of an extra transaction log file is necessary, remove it again as soon as possible**
  - It may seem like an extra transaction log file is not problematic, but it is
- **Consider the case of a restore after a disaster**
  - If the database files are missing, they must be recreated during the restore
  - All transaction log files must be zero-initialized after creation, adding to the downtime while recovering from the disaster
    - And then zero-initialized again if a differential backup is restored
  - A second transaction log file that was not removed adds further downtime
- **Be careful of shrinking multiple transaction log files as small as they will go as this will prevent the extra file being dropped**
  - If each file is shrunk to only have a single VLF, the transaction log has to have at least two VLFs so the second file cannot be dropped
  - Fix this by growing both files and then dropping the extra file

Minimize the Impact of Logging

- **Recovery will take longer if there were a lot of long-running transactions pending at the time of failure, failover, or restore**
- **Always try to avoid long-running transactions**
  - Transactions which span more than one batch
  - Transactions that require user interaction
  - Nested transactions (these do not actually exist in SQL Server)
  - Consider using the BULK_LOGGED recovery model during index maintenance
- **There may be multiple reasons that log clearing cannot occur**
- **Some operations may generate surprisingly more log records than expected which can contribute to the transaction log filling up quickly**
  - E.g. updating an index key column is not an in-place update
  - E.g. inserting a record into a table that causes a page split

Demo

Runaway log file

Recovery Models

- **FULL: everything is fully logged**
  - Log truncation usually not possible until log backup
  - Operations like creating or rebuilding an index creates as much log as the size of the index being created/rebuilt
- **BULK_LOGGED: minimal logging for SOME things**
  - Log truncation usually not possible until log backup
  - NOT non-logged, just minimally-logged
  - Limited set of operations are (see next slide)
  - ALL other operations (i.e. updates, inserts, etc.) take the same log space and time as the FULL recovery model
  - Log backups will NOT be smaller then when in FULL!
- **SIMPLE: same logging as for BULK_LOGGED**
  - Log is cleared/truncated on checkpoint
  - No log backups possible

Minimally-Logged Operations

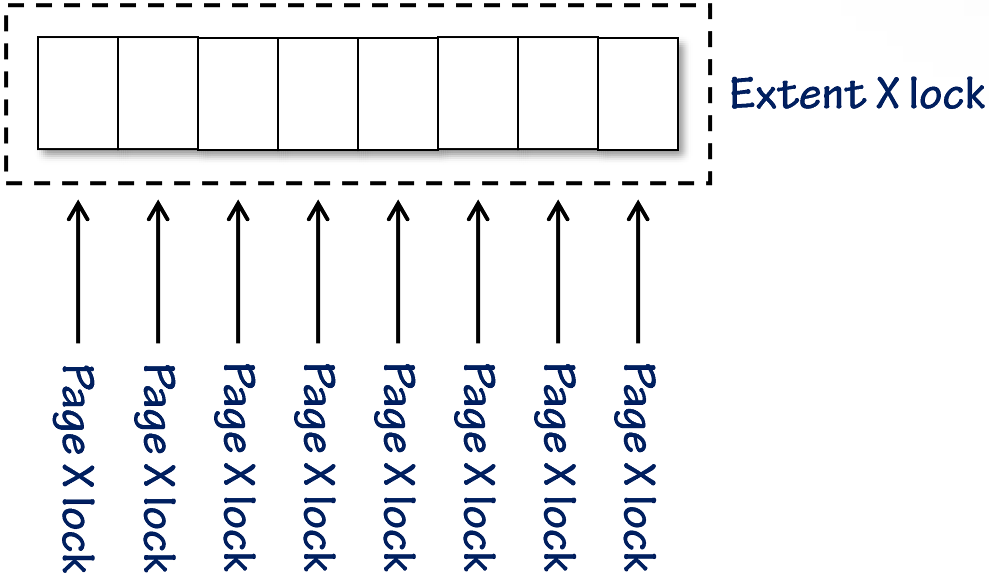
- **There is a small list of operations that can be minimally-logged, with the main ones listed below**
  - The complete list is in the Books Online topic 'The Transaction Log (SQL Server)' at https://sqlskills.com/p/051
- **Creating, dropping or rebuilding indexes**
  - Reorganizing indexes is always fully logged
- **Bulk-load operations**
  - BCP, OPENROWSET (BULK, …), BULK INSERT
- **SELECT INTO a new permanent table**
- **LOB data changes with the .WRITE option for an UPDATE statement**
  - UPDATETEXT/WRITETEXT also, but these are deprecated commands
- **Whitepaper: Data Loading Performance Guide**
  - https://sqlskills.com/p/055
  - Complete guide to all necessary conditions for minimal logging

Deferred Drop and DROP/TRUNCATE TABLE

- **There is a common misconception that these operations are non-logged or minimally-logged**
- **They're both fully-logged but very efficiently logged**
  - TRUNCATE TABLE can be performed in a transaction and rolled-back
  - PFS and GAM updates total 0.35-0.4% of table size being logged
- **Either the entire DROP/TRUNCATE occurs immediately**
  - Only for allocation units less than 128 extents
  - Pages and extents are de-allocated and no individual rows are deleted
- **Or the operation is performed using the deferred-drop mechanism**
  - Allocation units moved from table metadata to deferred-drop work queue
  - Deferred-drop background task de-allocates pages extents in small chunks
  - Added in SQL Server 2000 SP3, avoids running out of memory
- **Beware: TRUNCATE resets identity values, DELETE does not**

Why Deallocation Takes Lock Memory

- Before an extent can be deallocated, it must be locked and all page locks ‘probed’
- The extent lock is held until the end of the transaction

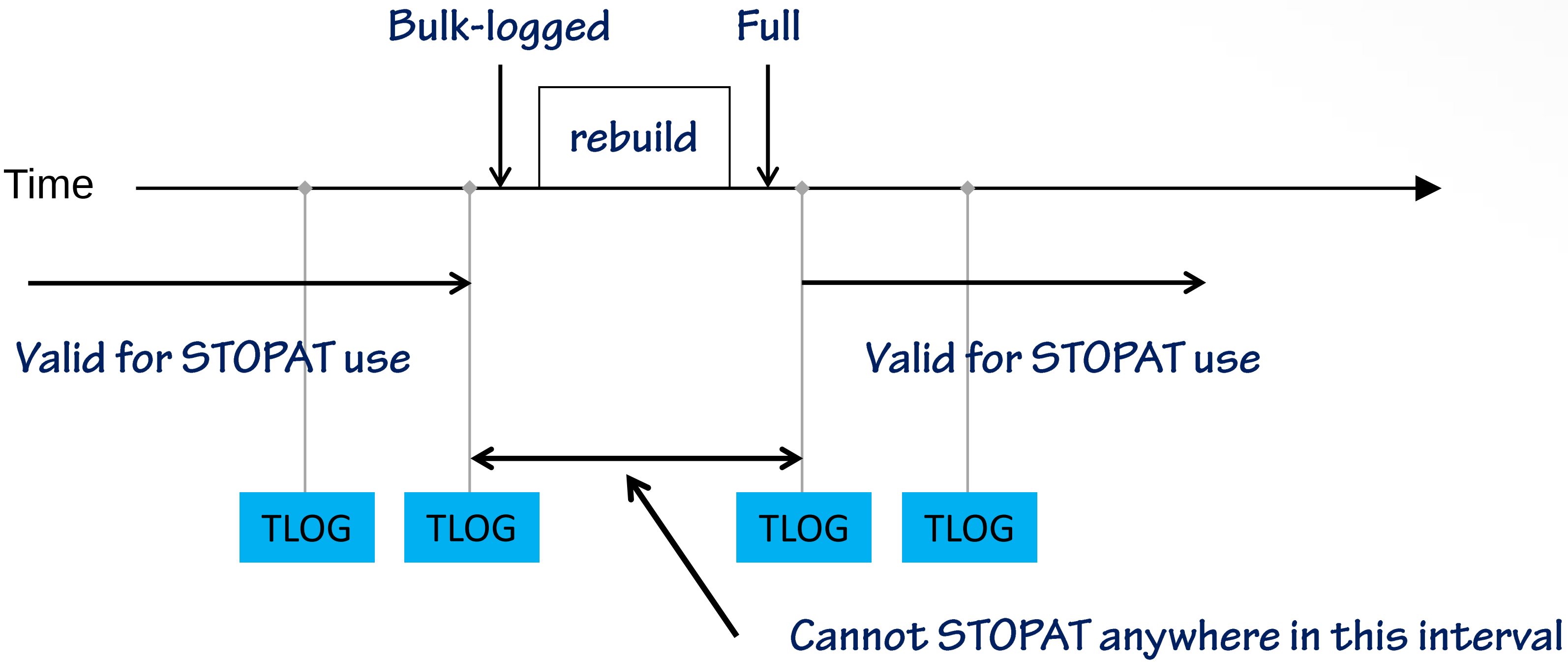


Recovery Models and Log Backups

- **FULL recovery model**
  - All changes are fully logged therefore log backups will support:
    - Up-to-the-minute recovery by backing up the tail-of-the-log
    - Point-in-time recovery with the option STOPAT on restore (marked transactions or time-based STOPAT)
- **BULK_LOGGED recovery model**
  - Bulk operations are minimally logged
    - If the tail-of-the-log contains a minimally logged operation and the data files are unavailable, the tail-of-the-log backup will succeed but will corrupt the database when restored
    - NO point-in-time recovery possible for the time covered by a log backup containing a minimally-logged operation
- **SIMPLE recovery model does not support log backups**

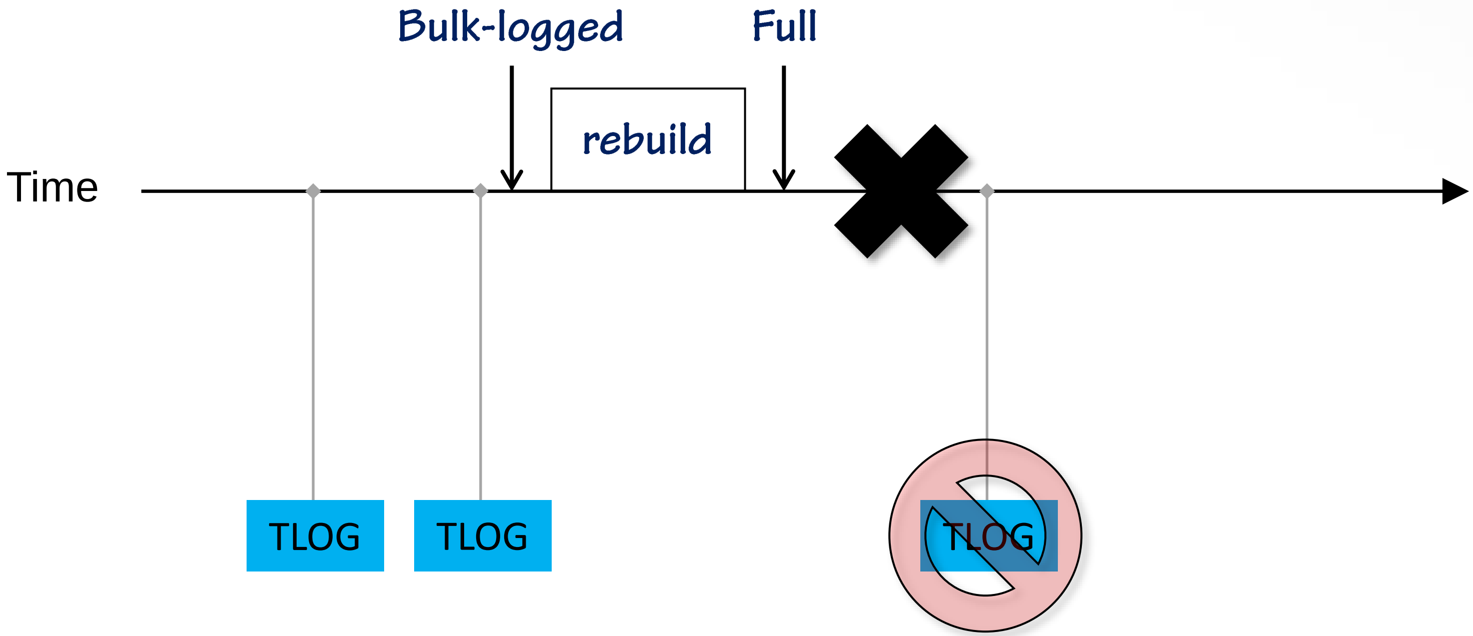
Point-in-time Restore with Minimal Logging

- A log backup containing a minimally-logged operation can be used as part of a normal restore sequence
- Restore cannot STOPAT any point covered by that log backup



Minimal Logging and Disaster Recovery

- If a crash occurs that damages or destroys the data files, and there has been a minimally-logged operation, tail-of-the-log backup will be corrupt as it cannot back up the minimally-logged data

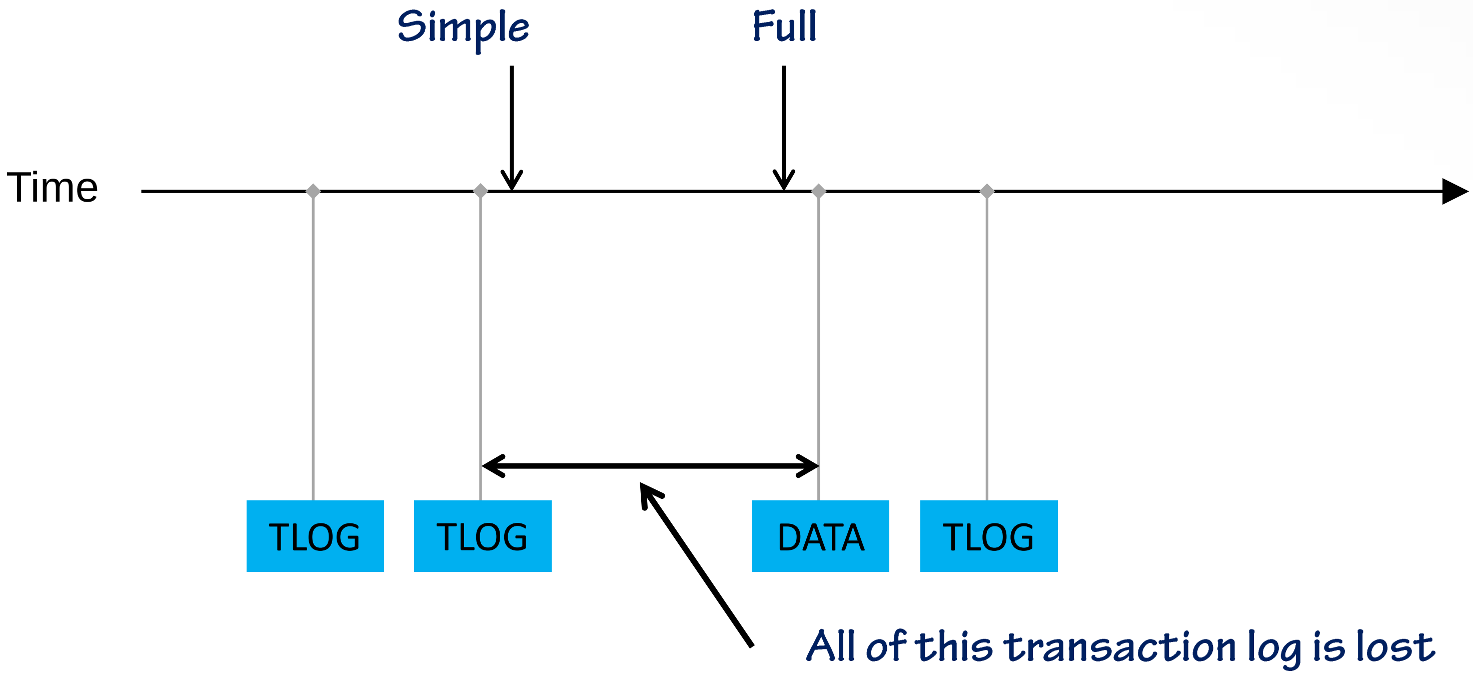


Switching Recovery Models

- **When switching to FULL for the first time, the database behaves as if it is in the SIMPLE recovery model until the first full backup is performed**
  - This is called being in 'pseudo-SIMPLE'
  - The transaction log will clear whenever a checkpoint occurs
- **The number one cause of transaction log files growing out of control is that a database has been running in pseudo-SIMPLE and then someone takes a backup, really switching the database into FULL**
  - And so it will grow forever unless log backups are also being taken
- **Switching between FULL and BULK_LOGGED does NOT break the log backup chain, and does not affect log shipping**
  - This is NOT possible with database mirroring or availability groups
- **Switching to SIMPLE breaks the log backup chain**

Switching Out of SIMPLE

- When you switch from SIMPLE to FULL or BULK_LOGGED, you cannot perform log backups until you perform a data backup
- Yes, you can use a differential backup to do this

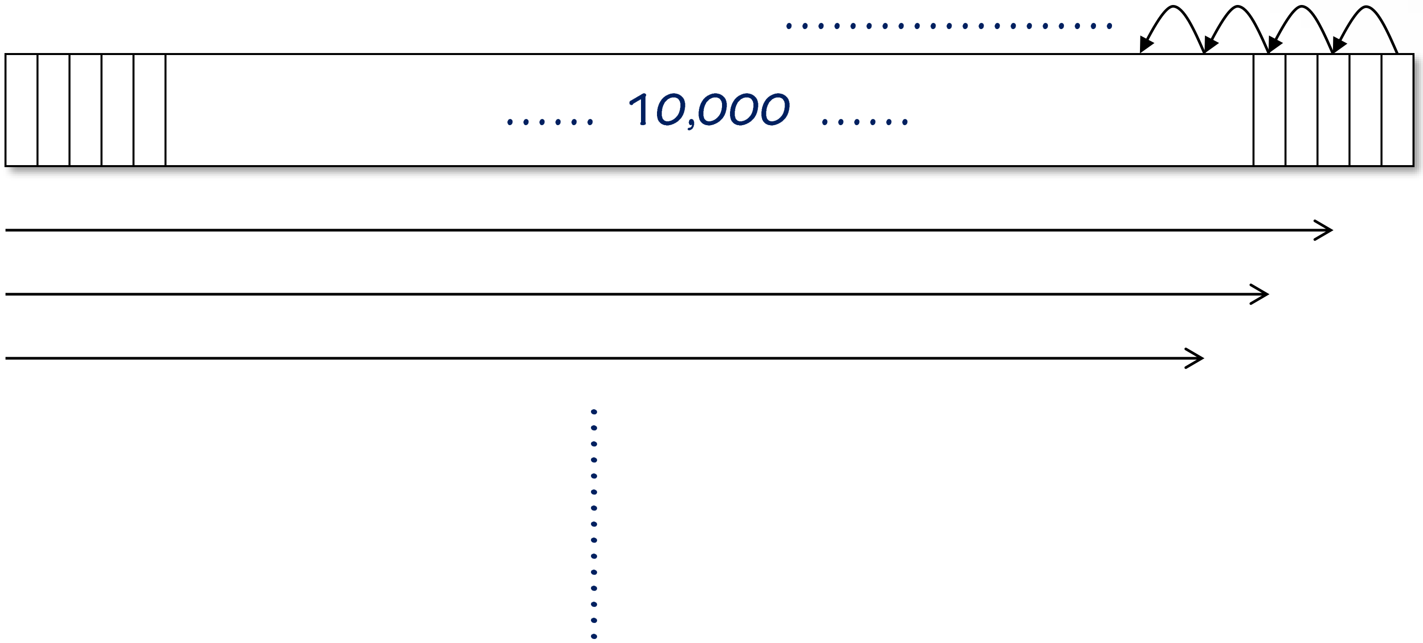


VLF Fragmentation: Too Many VLFs?

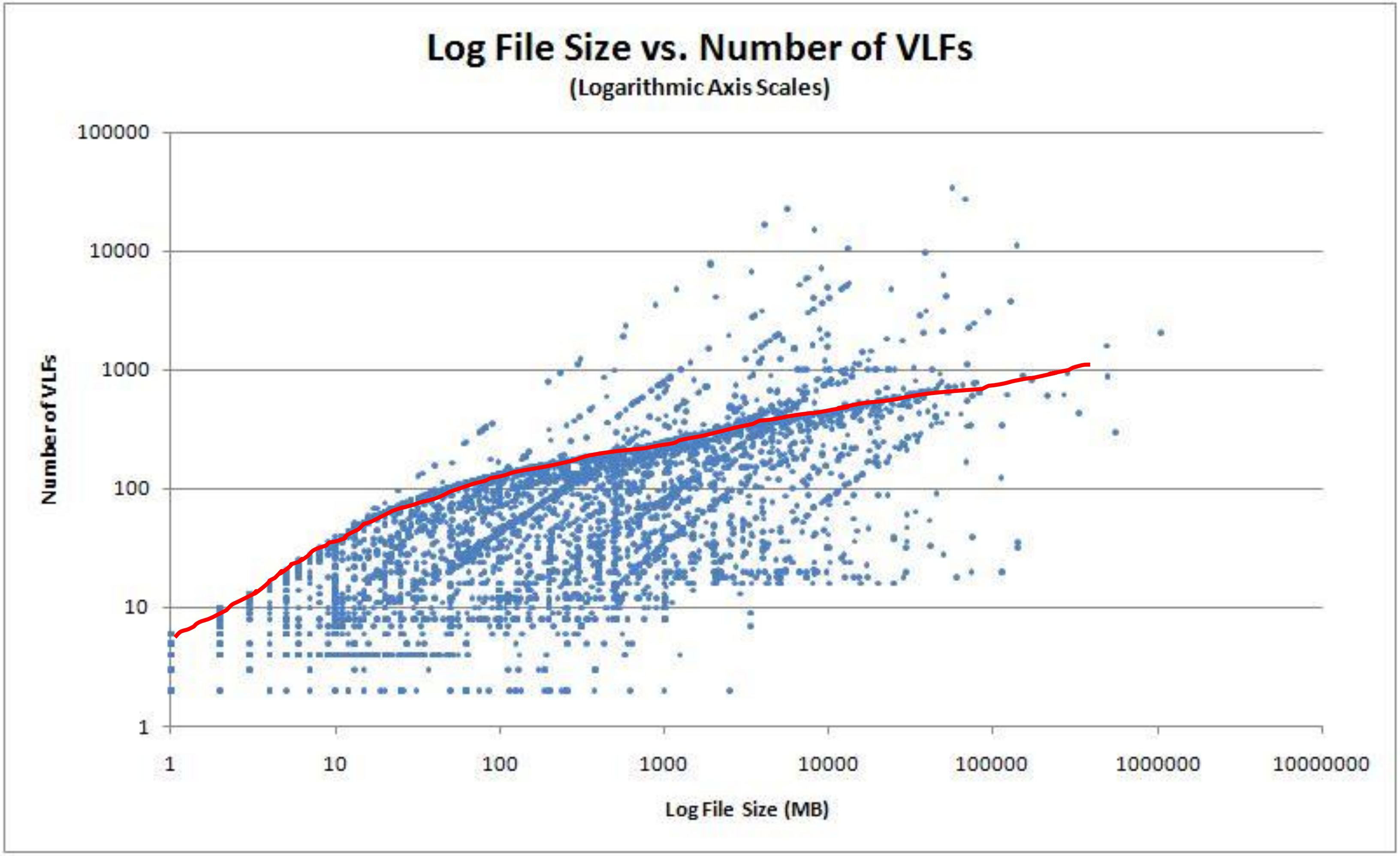
- **Earlier we saw the formula for how many VLFs are created with each additional growth/auto-growth**
- **Excessive VLFs ('VLF fragmentation') add overhead to transaction log activities because finding a VLF means traversing the list of VLFs**
  - All transaction log activity can be affected, including crash recovery
    - Log backups and log readers (replication, change data capture, database mirroring, availability groups, restoring on log shipping secondary)
  - Empirical evidence: https://sqlskills.com/p/056
  - Many bugs fixed for crash recovery in all versions
    - KB articles: https://sqlskills.com/p/057 and https://sqlskills.com/p/058
  - Reported at startup in the error log in SQL Server 2012+
  - See blog post about VLF algorithm: https://sqlskills.com/p/037
- **What is excessive?**
  - Depends on log file size: many thousands are not ok
- **Careful size management is required**

VLF Searching During Rollback

- When a transaction rolls back, log records are undone in reverse order
- Finding the next log record means searching from the start of the log for the right VLF, following the VLF sequence number chain
- The more VLFs, the longer each search takes



Survey: Transaction Log File Size vs. VLFs



- Source: https://sqlskills.com/p/005

Too Few VLFs?

- **Too few VLFs mean they may be very large and can affect ability to resize the transaction log**
- **Consider a transaction log that was created as 100GB initially**
  - It will have 16 VLFs, each 6.25GB
  - Usually not be a problem unless the log shrinks and now there are only two very-large VLFs
- **On SQL Server 2014+**
  - If growth size is less than 1/8 of current log size, only one VLF is created
  - See blog post at https://sqlskills.com/p/037 for details
- **Careful size management is required**

Log File Provisioning

- **Only ONE transaction log file is necessary**
  - Log write activity is NOT parallelized so no perf gain from multiple files
    - Jonathan Kehayias proved this and blogged about it (https://sqlskills.com/p/060)
  - Only time another log file may be needed is if transaction log runs out of space and there is no other option
- **Isolate transaction log file from data files to avoid possible I/O subsystem contention problems**
  - Potentially place the transaction log on a very fast I/O subsystem like an SSD
- **New log space stamped with 0x00s pre-2016 and 0xC0s in 2016+**
  - See blog post at https://sqlskills.com/p/061
- **Choose an appropriate RAID level**
  - RAID 5 not recommended as log is write-heavy, and poor failure protection
  - RAID 1 is good, RAID 10 is better for failure protection
    - This is especially true for SSDs, where placing the transaction log on RAID 0 on a single SSD, or RAID 1 over two sub-parts of an SSD, is not acceptable

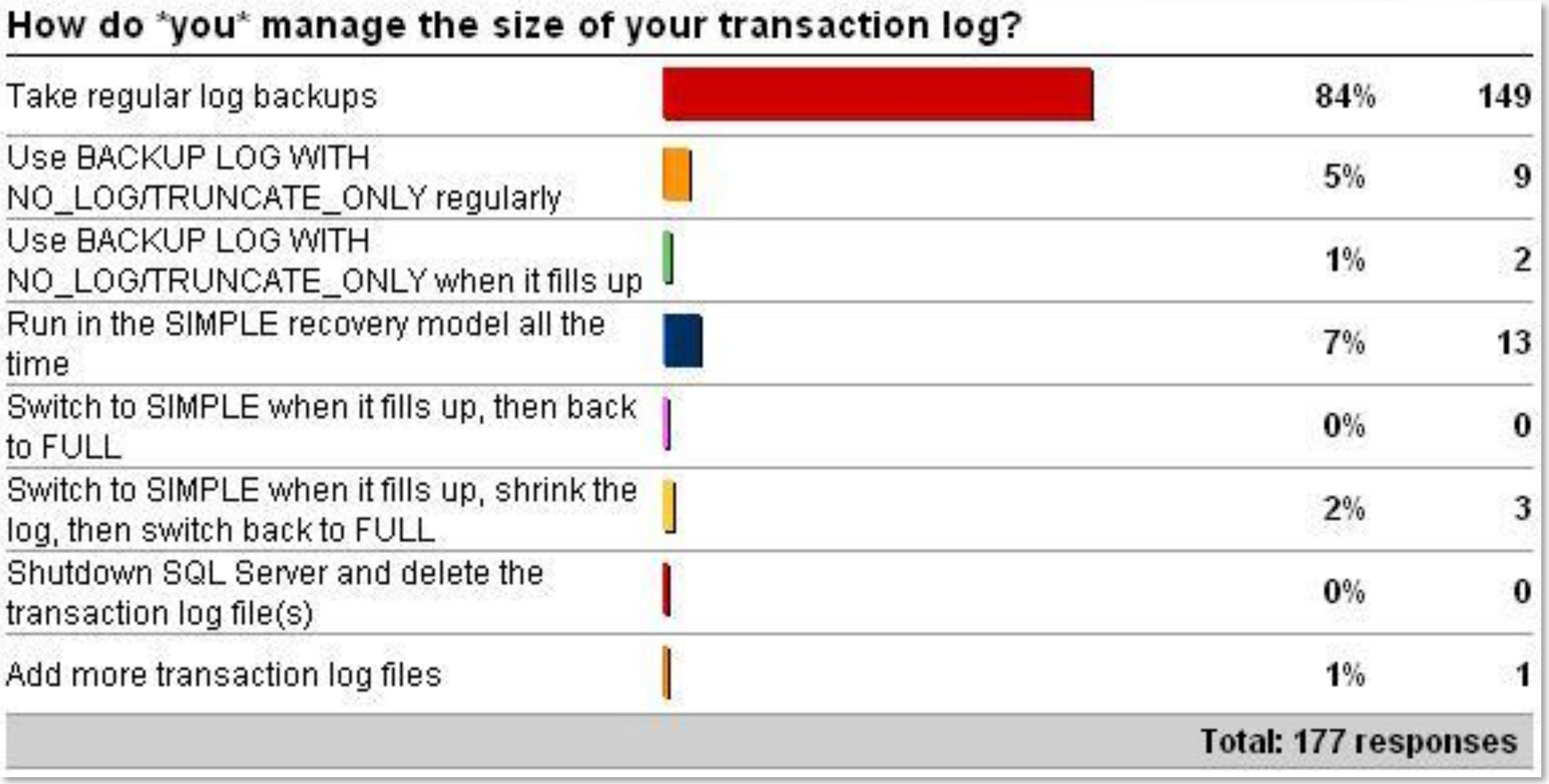
Estimating Transaction Log Size

- **Pre-allocate the log to an appropriate size**
  - What is ‘appropriate’? There is no one-size-fits-all answer or formula!
- **Estimate transaction log size based on:**
  - Transaction log generation rate from the regular workload
  - Recovery model
  - Log and data backup schedules
  - HA/DR technologies that may affect log clearing
  - The size of the largest single insert/update/delete transaction
    - E.g. a single-statement transaction affecting millions of rows
    - E.g. a multi-statement transaction performing many operations
  - The size of the largest bulk-load or index-rebuild operation
- **Create the transaction log with a moderate auto-growth size and monitor the transaction log size until it reaches a steady state**
  - Now fix any potential VLF fragmentation

Configuring Transaction Log Auto-Growth

- **Auto-growth should be configured for transaction log files for the emergency case where monitoring fails**
  - If log fills up and cannot auto-grow, all write activity in the database stops
- **Consider that new log space has to be zero-initialized**
  - This takes time and pauses write activity as no new log records can be generated if auto-grow triggered the growth
- **The auto-growth size to use needs to take into account:**
  - Potential for workload delays from having the growth amount set too high
  - Potential for VLF fragmentation from having the growth amount set too low
  - Log generation rate so it may fill again and require more auto-growth
- **Default is 10% prior to SQL Server 2016 (8MB initial and 64MB autogrow)**
- **Do not use a percentage as the growth amount will increase over time**
- **There is no one-size-fits-all answer or formula!**

Survey: Transaction Log Size Management



How do *you* manage the size of your transaction log?

Take regular log backups	84%	149
Use BACKUP LOG WITH NO_LOG/TRUNCATE_ONLY regularly	5%	9
Use BACKUP LOG WITH NO_LOG/TRUNCATE_ONLY when it fills up	1%	2
Run in the SIMPLE recovery model all the time	7%	13
Switch to SIMPLE when it fills up, then back to FULL	0%	0
Switch to SIMPLE when it fills up, shrink the log, then switch back to FULL	2%	3
Shutdown SQL Server and delete the transaction log file(s)	0%	0
Add more transaction log files	1%	1
		Total: 177 responses

- **Source:** https://sqlskills.com/p/062

Tempdb Exception…

- **Tempdb log resets to the last set size on restart**
- **Tempdb log is NOT entirely zero initialized on restart**
  - See https://sqlskills.com/p/063
- **Tempdb log VLF count is reset on restart**
  - See https://sqlskills.com/p/064

Transaction Log File Shrinking

- **The only way to reduce the transaction log size is DBCC SHRINKFILE**
  - Do not use the EMPTYFILE option as it has no effect
  - Do not use the NOTRUNCATE option as it will not shrink the file
- **Shrinking a transaction log file simply removes inactive VLFs from the end of the file**
  - Shrinking does not move or remove log records
  - Transaction log shrinking is completely different from data file shrinking in that no index fragmentation is caused
- **Shrink will terminate when it encounters an active VLF**
- **Knowledge Base article 907511 describes transaction log shrinking**
  - https://sqlskills.com/p/065
- **Do not regularly shrink the transaction log**
  - If it keeps growing again, leave it at the steady-state size
  - Otherwise it is repeatedly having to zero-initialize the newly-added space

Removing VLF Fragmentation

- **Execute DBCC LOGINFO to find the number of VLFs**
  - Number of rows = number of VLFs
- **If excessive (> many hundreds or thousands) then need to remove some to prevent performance problems**
  - Excessive is relative to transaction log size
- **Free up log space by first clearing the log**
  - If the recovery model is FULL or BULK_LOGGED then perform a log backup
    - This should remove the inactive portion of the log as long as there are no long-running transactions
  - If the recovery model is SIMPLE then perform a CHECKPOINT
    - Again, as long as there are no long-running transactions
  - If the highest active VLF after the log clearing is towards the end of the transaction log file, perform another log clearing to make the log manager wrap the log (and maybe another, and another)

Removing VLF Fragmentation (2)

- **Manually shrink the transaction log file**
  - DBCC SHRINKFILE (logfilename)
  - A second or third log clearing operation may be necessary to make the highest active VLF near the start of transaction log file
  - This can be difficult to do on a busy production system
- **Modify the transaction log file size to a more appropriate size in one or more increments, as described previously**
  - ALTER DATABASE dbname MODIFY FILE (NAME = name, SIZE = new_size);
  - Remember to keep the resulting VLF sizes around 512MB or less

Demo

Removing VLF fragmentation

Review

- Transaction log architecture
- Log records
- Checkpoints and recovery
- Transaction log operations
- Recovery models
- Log file provisioning and maintenance

- Blog posts: https://sqlskills.com/p/066
- TechNet: Understanding Logging and Recovery in SQL Server
  - https://sqlskills.com/p/067
- Pluralsight: *SQL Server: Understanding Logging, Recovery, and the Transaction Log*

Questions!

SQLskills

immerse yourself in sql server