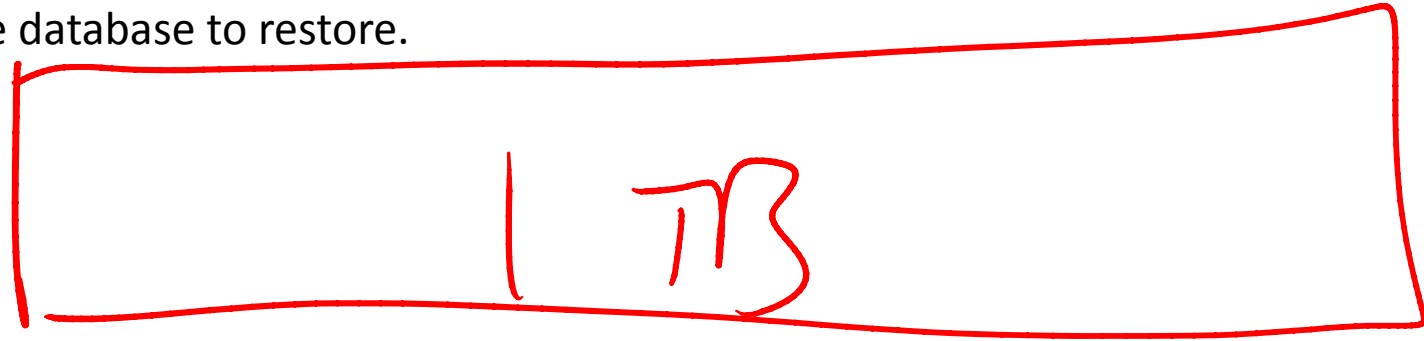
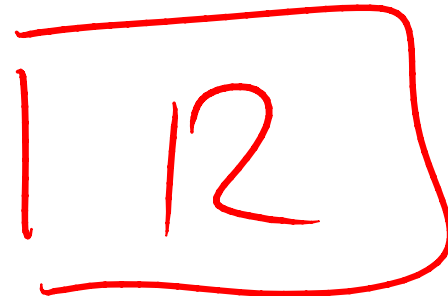
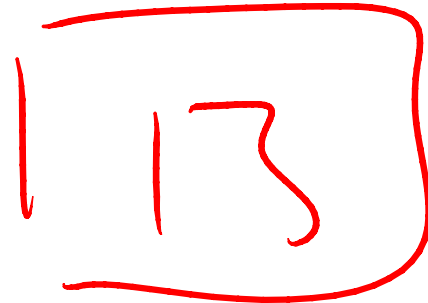
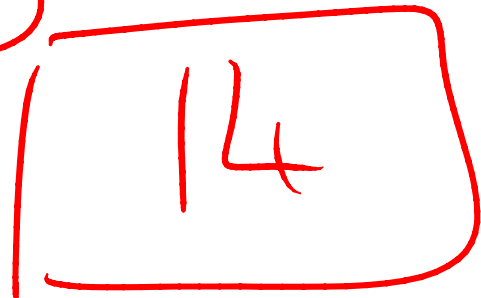
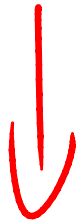


M1S7 Explaining that dividing a large database into filegroups allows targeted restores during disaster recovery, rather than having to wait for the entire single file database to restore.

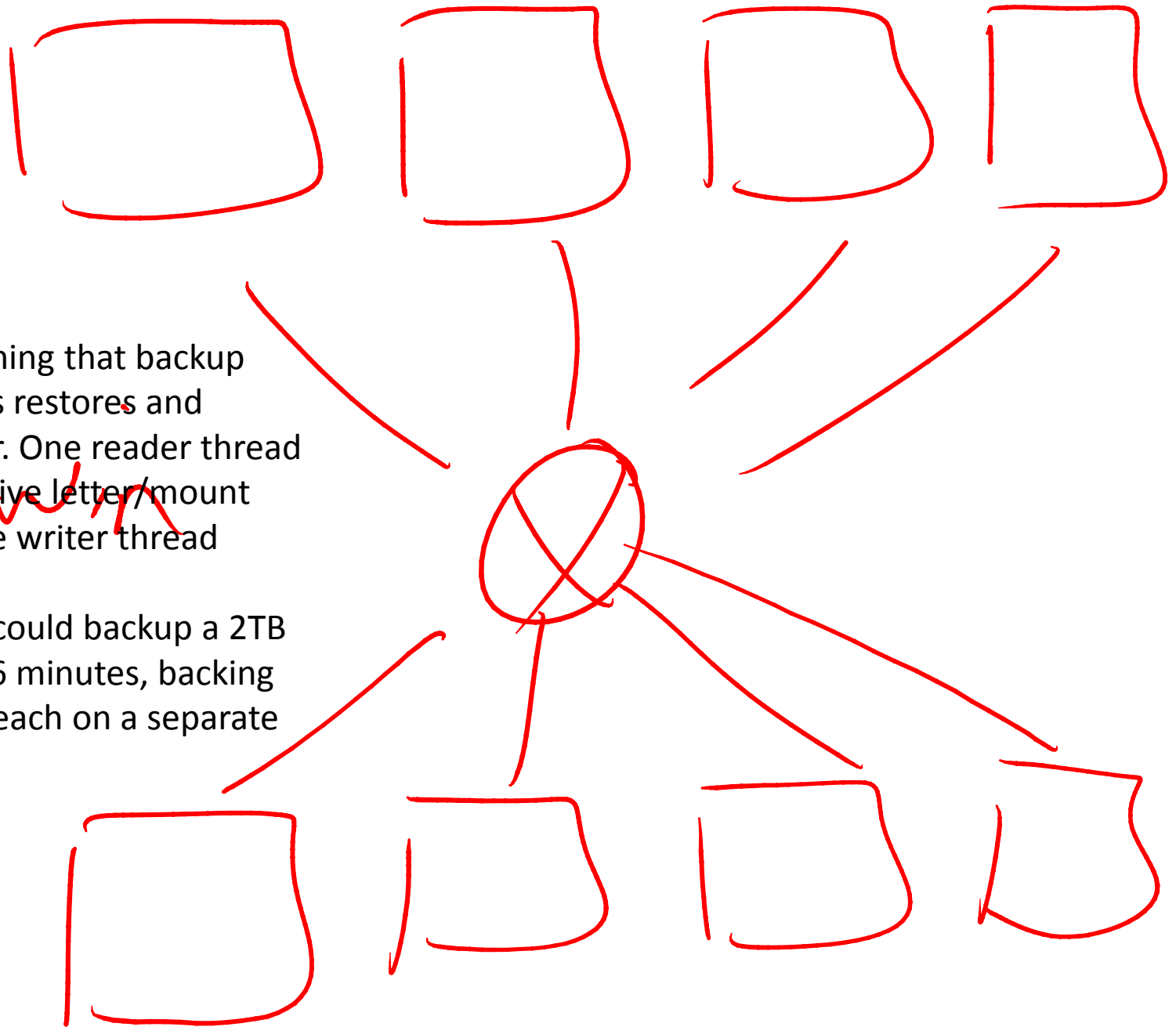


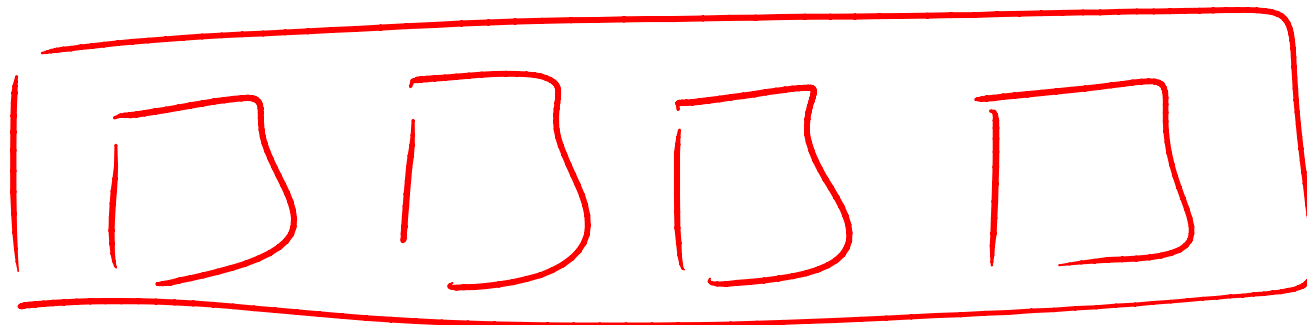
SALES



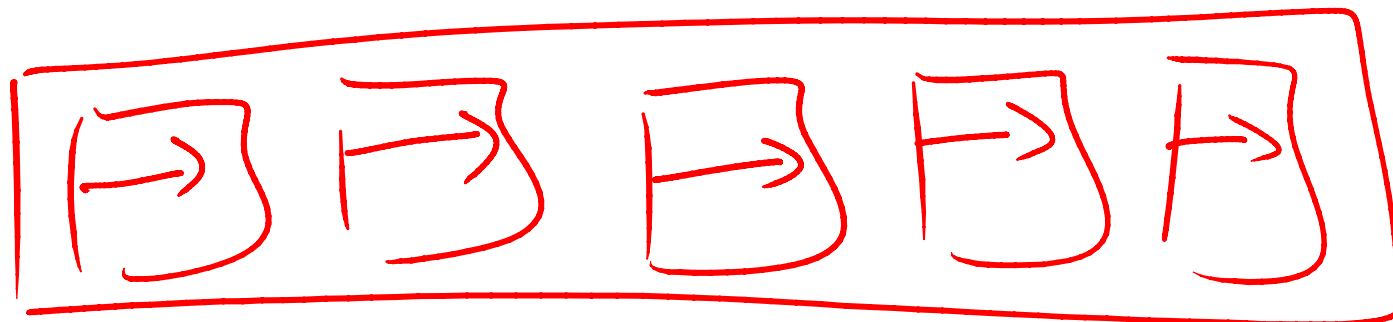
M1S33 Explaining that backup striping makes restores and backups faster. One reader thread per distinct drive letter/mount point, and one writer thread similarly.

Bwin in 2005 could backup a 2TB database in 36 minutes, backing up to 12 files each on a separate drive.

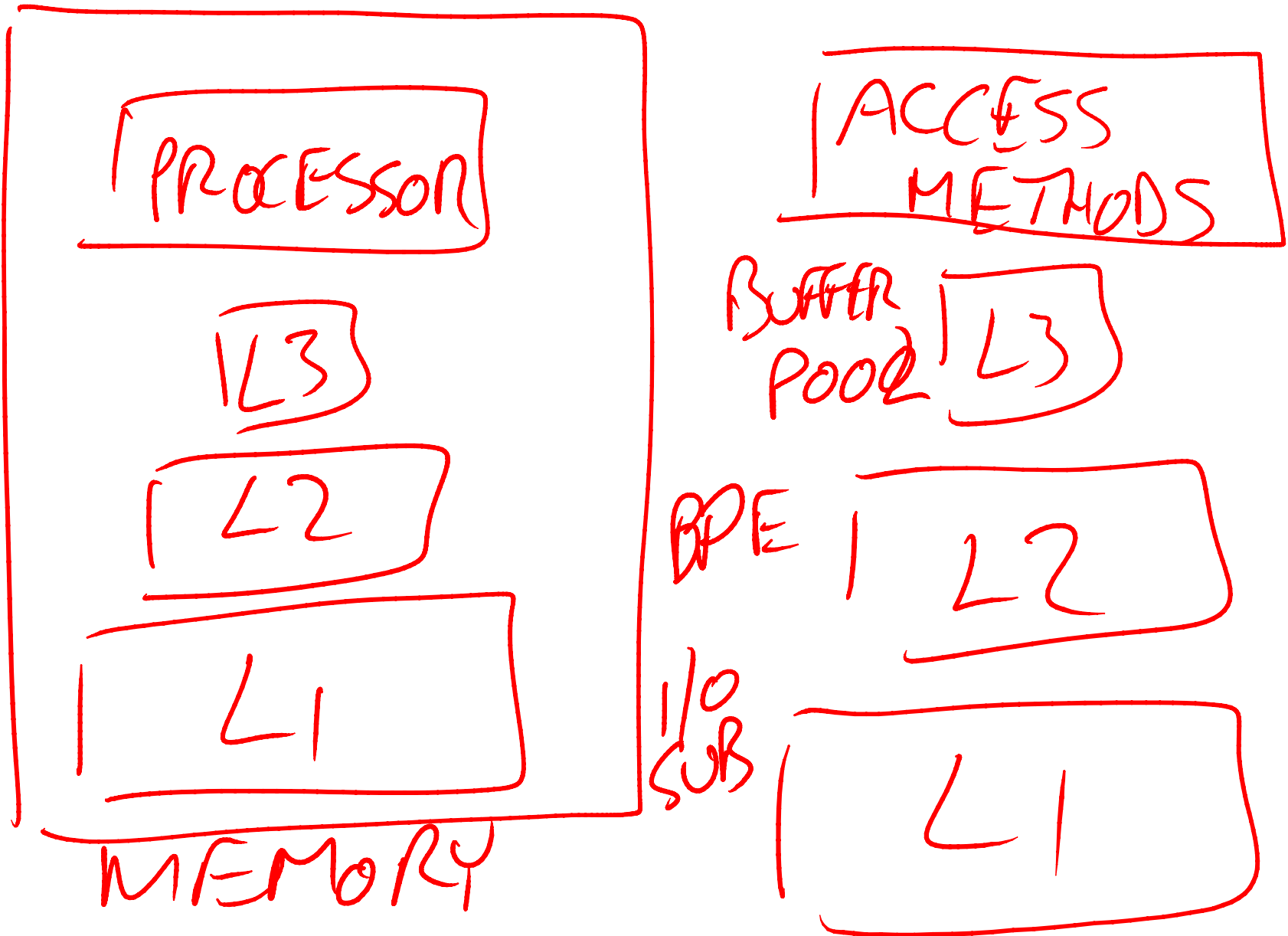




M1S7 Be careful about putting all the log files on a single volume. If they're all having activity, that creates multiple write points on the volume, making it a random I/O workload, even though each log is sequential-write only.

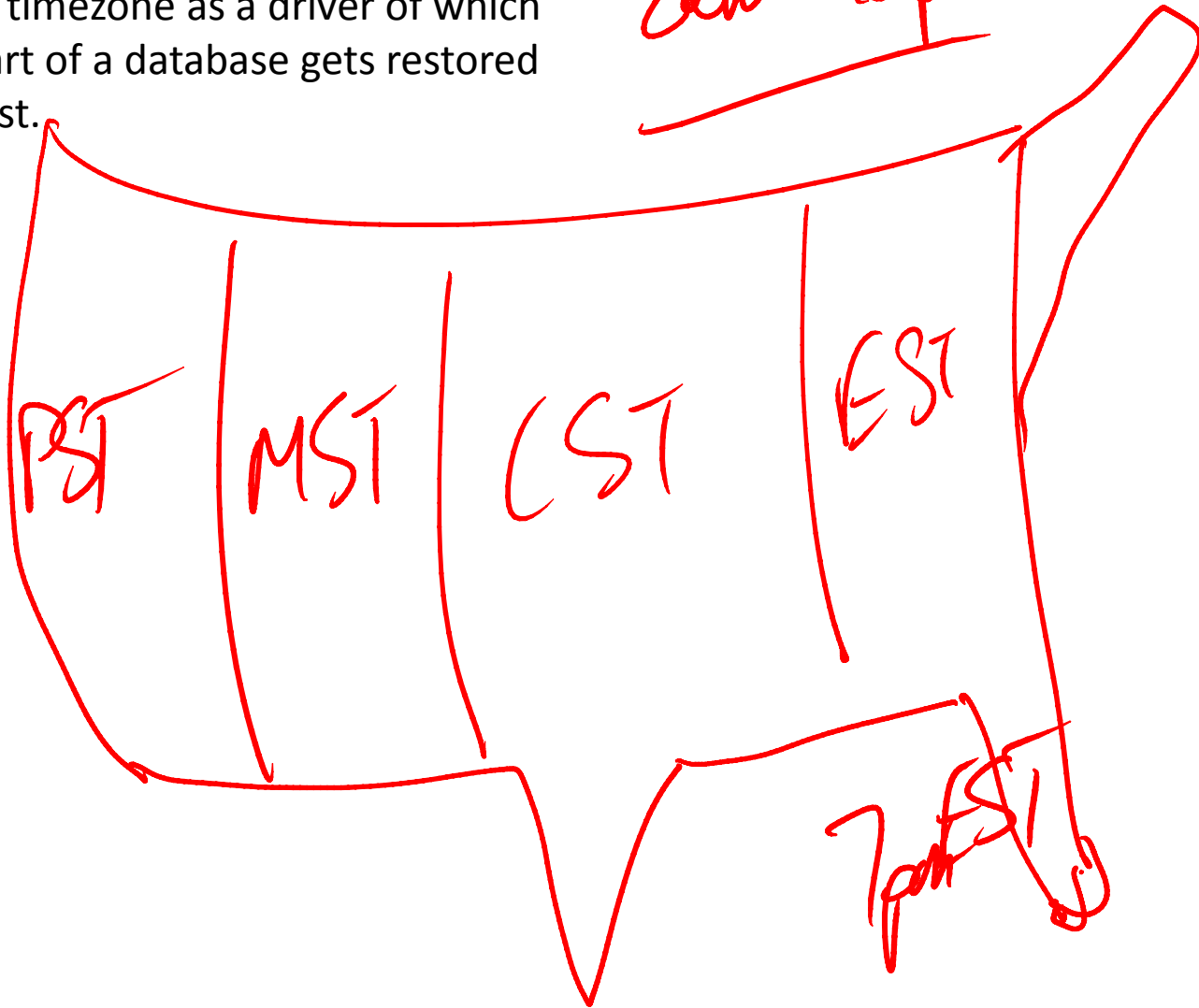


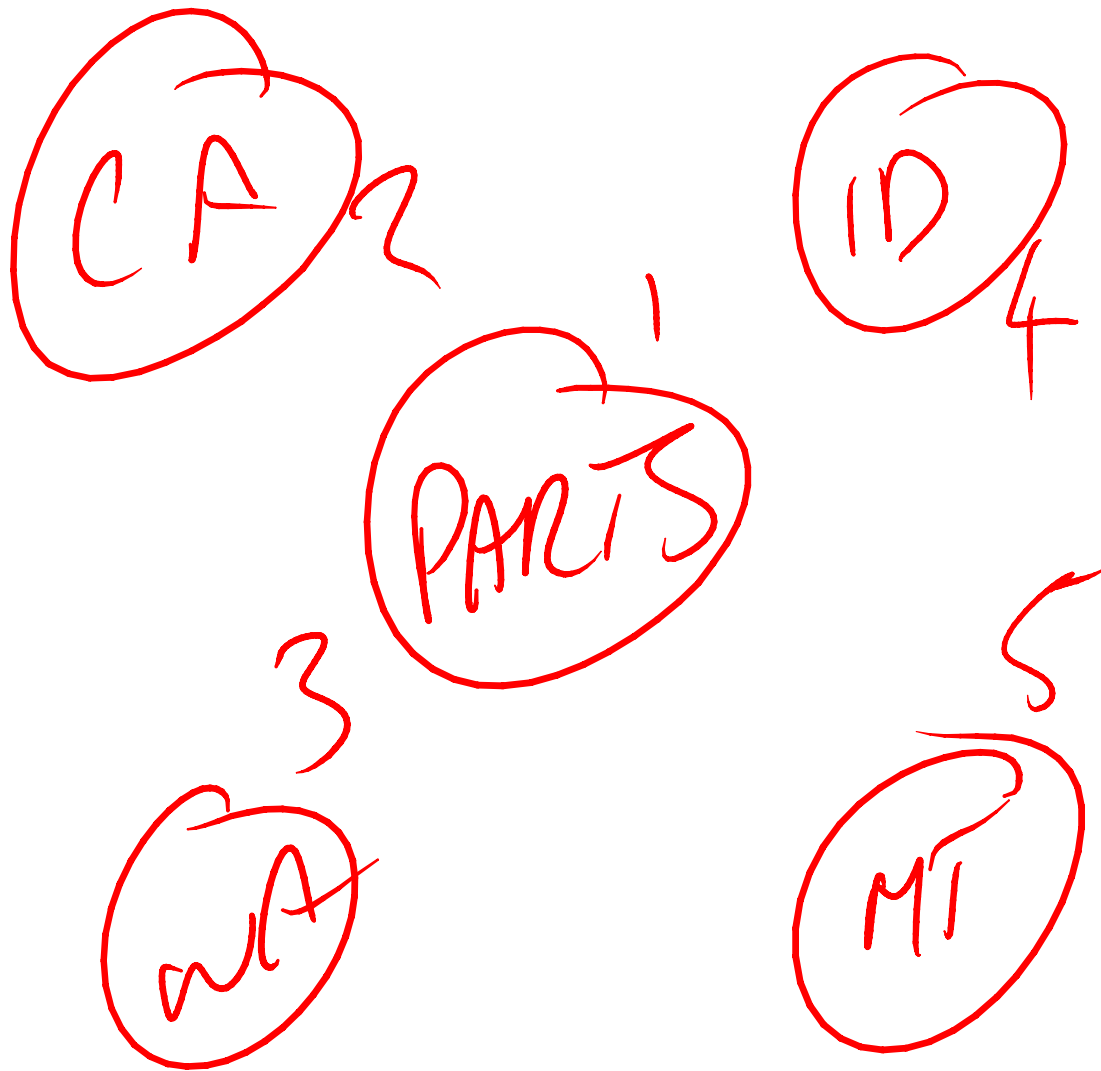
M1S13 Explaining 2014 Buffer Pool Extension. It's like have L1-L3 caches for your data in memory. BPE sits in between memory and the I/O subsystem.



M1S7 My fabulous drawing of the US. Showing consideration of timezone as a driver of which part of a database gets restored first.

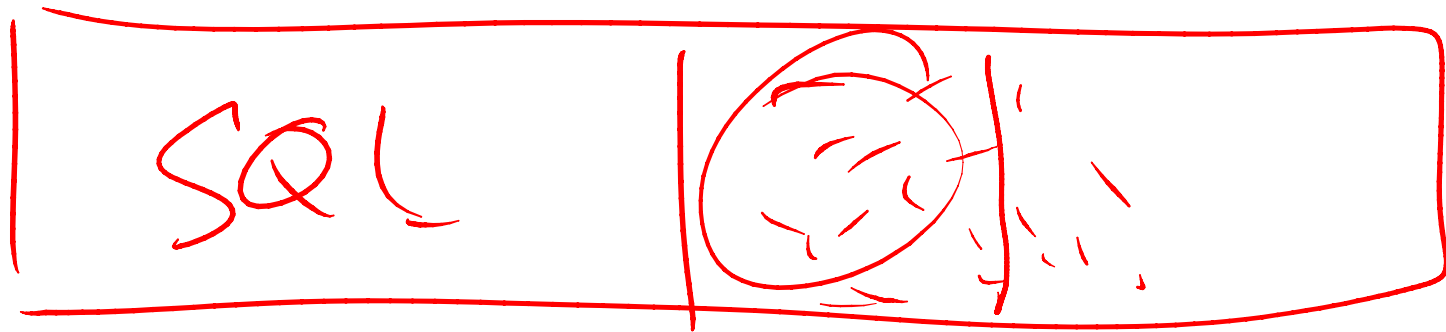
Sam 6pm





M1S7 Explaining a different partitioning scenario – auto parts sales to WA/CA/ID/MT.

If things go down, it makes sense to restore the portion of the database that will bring in the most sales quickly – CA, as it has the most population.

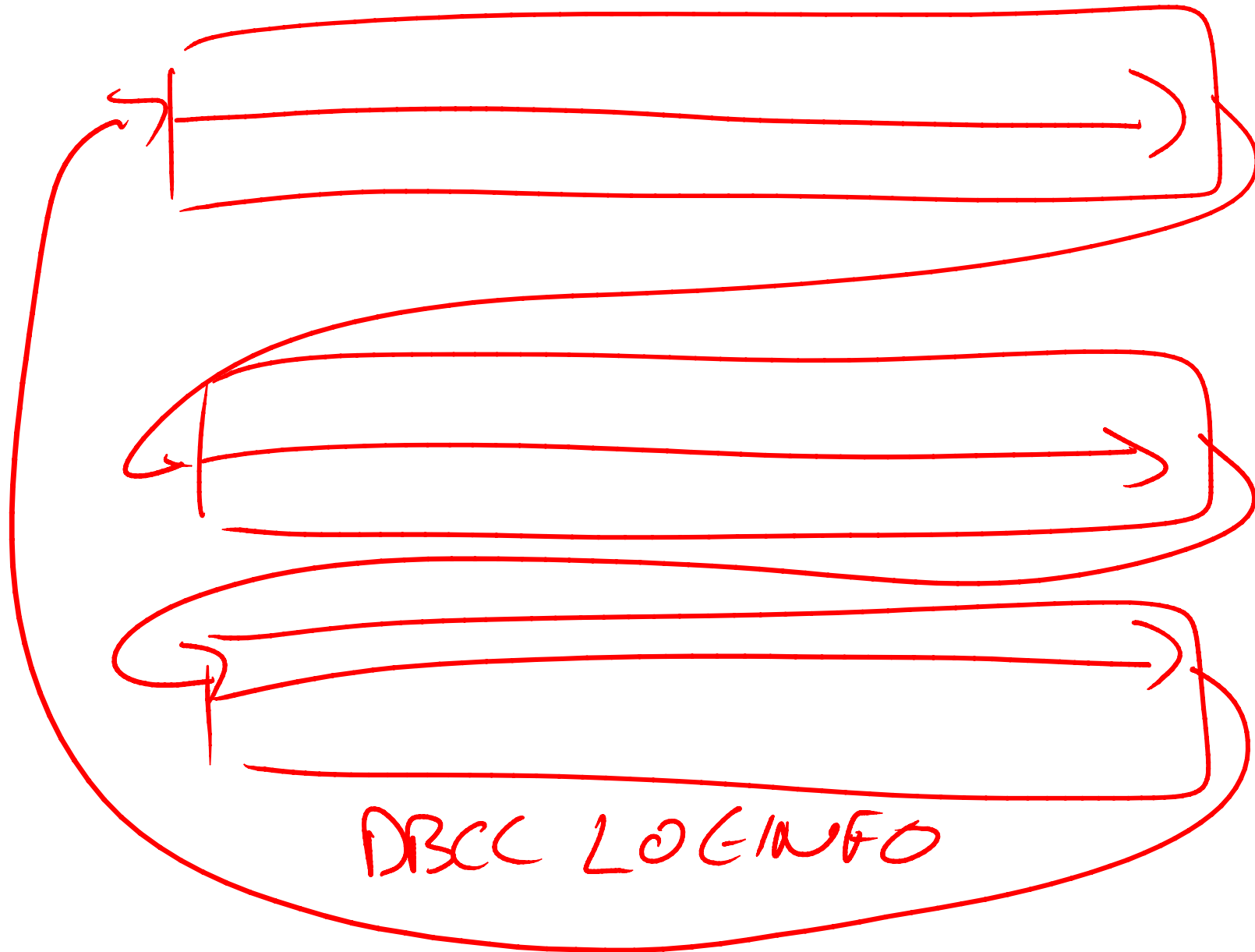


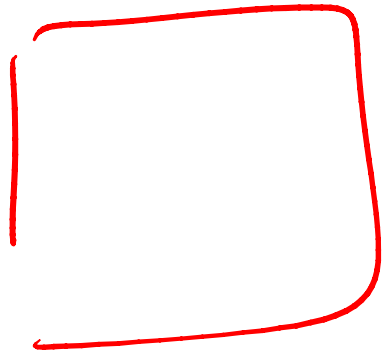
SA

DBCC PAGE

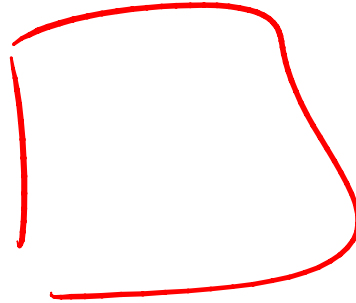
M1S14 Explaining why instant file initialization is off by default. Sharing a volume between SQL Server and 'secure files' can lead to information leakage if the secure files are deleted, then SQL grows a data file, and an SA can use DBCC PAGE to get a hex dump of the bits/bytes that comprised the deleted secure file. If you don't have this situation, turn instant file initialization on.

M1S18 Multiple log files will always be written to in sequential order, except for periodic updates to the file header pages.

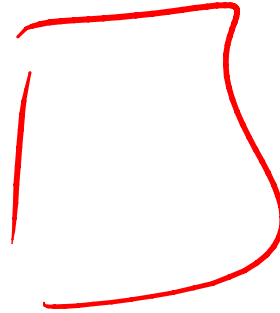




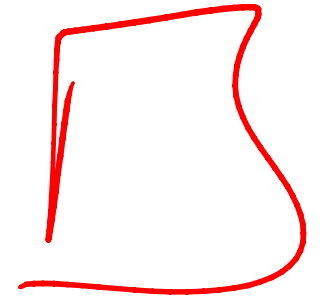
4000



4000



4000



4000

Bpm: PLE = 4000 $\rightarrow$ 3606

$\Rightarrow 3625$

1000 $\uparrow$

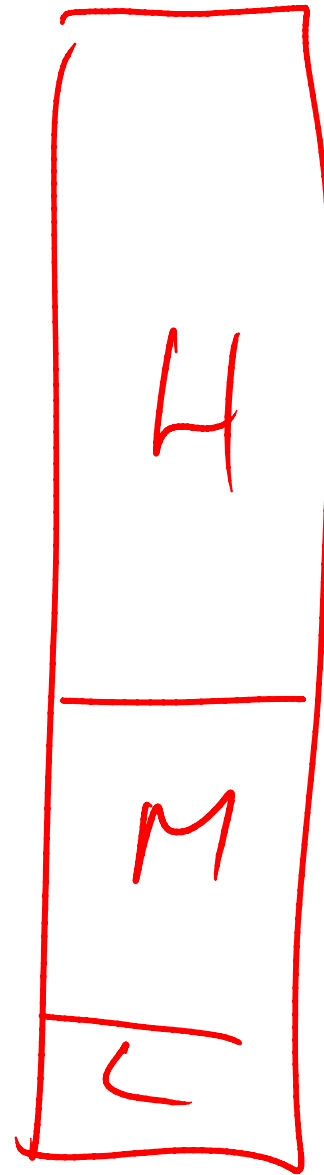
RESOURCE POOL

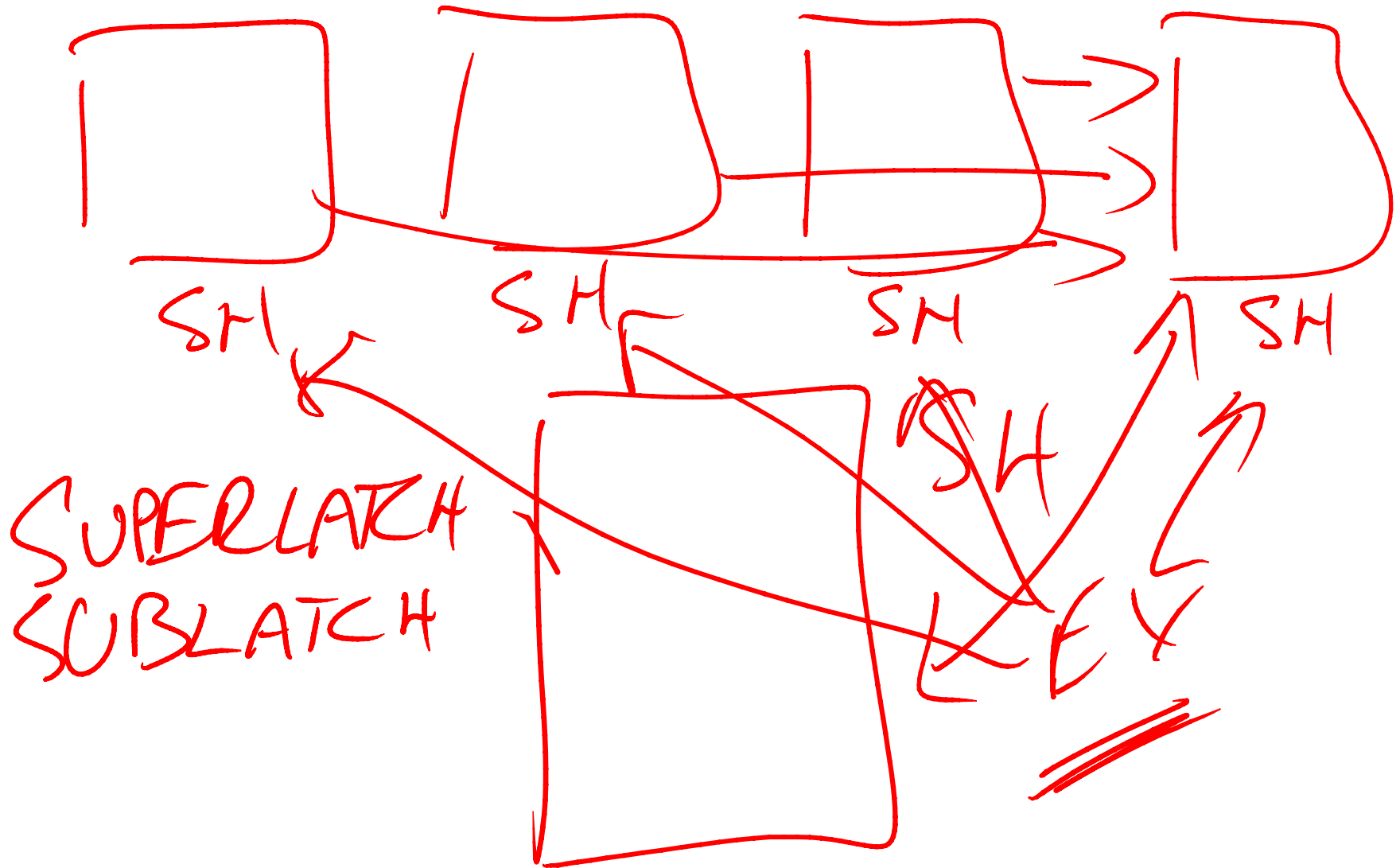
> / WORKLOAD GROUPS

H / M / L

9 : 3 : 1

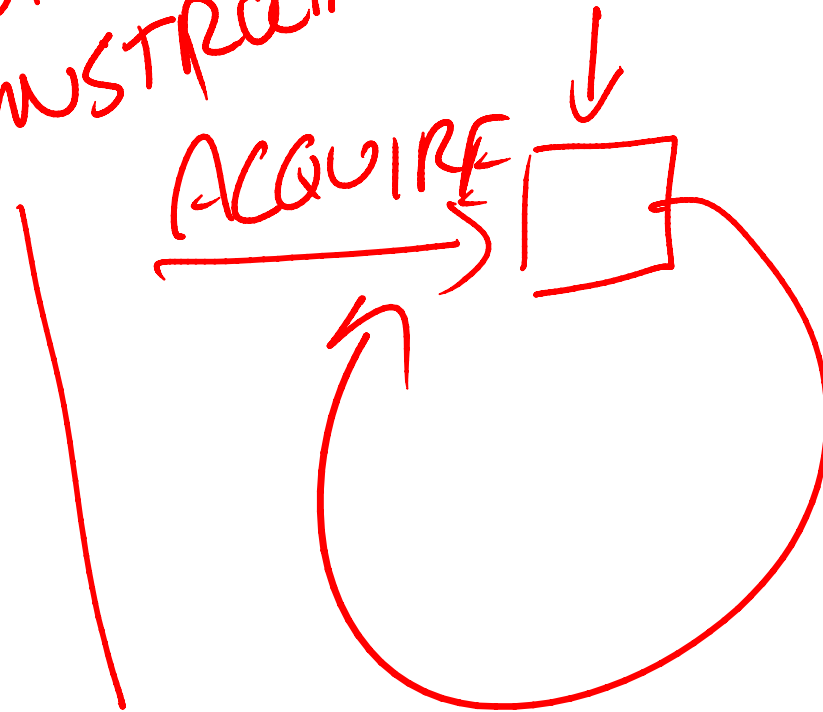
M7S17 Explaining how the Runnable Queue is a true FIFO (First-In, First-Out) queue unless Resource Governor workloads groups and group priorities are being used. High-Medium-Low priority equals 9-3-1 threads through the Runnable Queue.





M7S32 Explaining super-latches, where a SH latch is made per NUMA node to speed things up.

INTERLOCKED
INSTRUCTION SPINLOCK



SPINNING

1000

COLLISION

TST_SET_IF_CLEAR

BACKOFF

M7S37 Explaining how spinlocks work. Code does a test-and-set-if-clear on the memory that implements the spinlock. If it can't get it, it spins (tries again) up to 1000 times and then backs off. SOS_SCHEDULER_YIELD is the wait type when a backoff occurs.

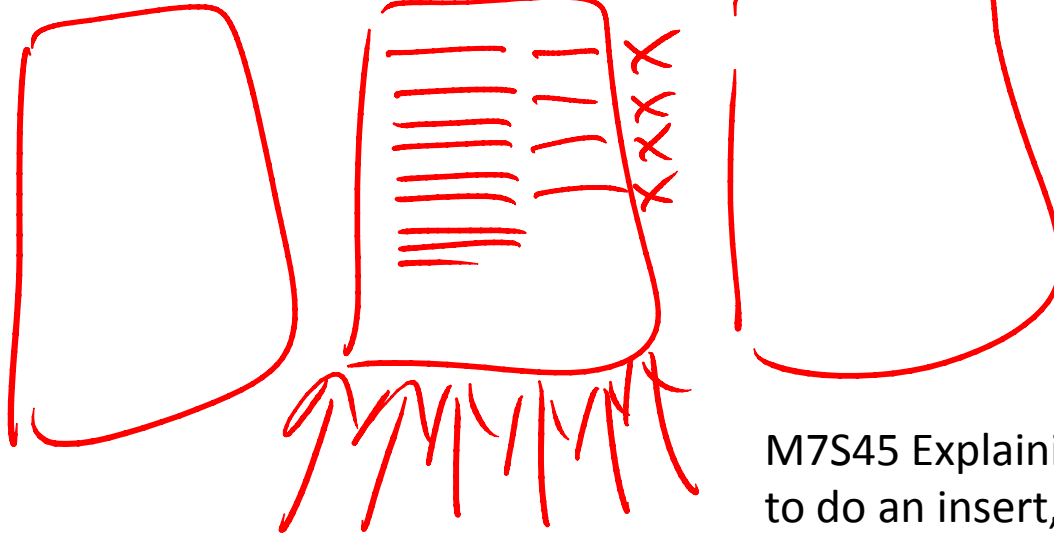
TIX

TIX = table IX lock

PIX = page IX lock

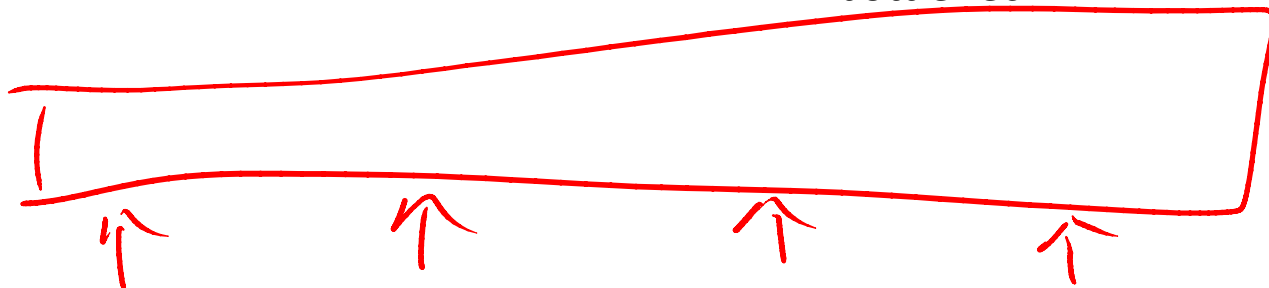
And there are row X locks

SOPIX



X X X
LATCH

M7S45 Explaining that to access a page to do an insert, multiple concurrent threads can hold non-conflicting row X locks, but they all need to acquire the X latch on the page, which becomes a bottleneck.



Solve that problem by creating multiple insert points, with a GUID or hash partitioning.



M7 Explaining how the lock pending queue works.

At time:

X: Thread 1 holds the lock

X+1: Thread 4 tries to acquire the lock and can't. It registers a callback routine for itself on the pending queue in the lock value block. Then it suspends itself, moves off the CPU and onto the waiter list.

X+2: Thread 7 tries to acquire the lock and can't. It registers a callback routine etc etc, and it behind thread 4 in the pending queue.

X+3: Thread 1 releases the lock, which grants the lock to thread 4. Thread 4's callback routine is called, signaling it and allowing it to move to the runnable queue on it's scheduler. Thread 7 is now at the head of the pending queue for the lock.

X+4: Thread 4 releases the lock, which grants the lock to thread 7. Etc etc. There is now no pending queue for the lock.

X+5: Thread 1 tries to acquire the lock again and can't, so registers the callback routine, goes on the pending queue, and suspends onto the waiter list.

M7S47 Another cause of blocking could be erroneous use of SERIALIZABLE isolation level. This could be from distributed transactions or incorrect use of .Net TransactionScope objects.

LCK_M_RS_S
_X
_U

~~DA~~ * dm_exec_sessions