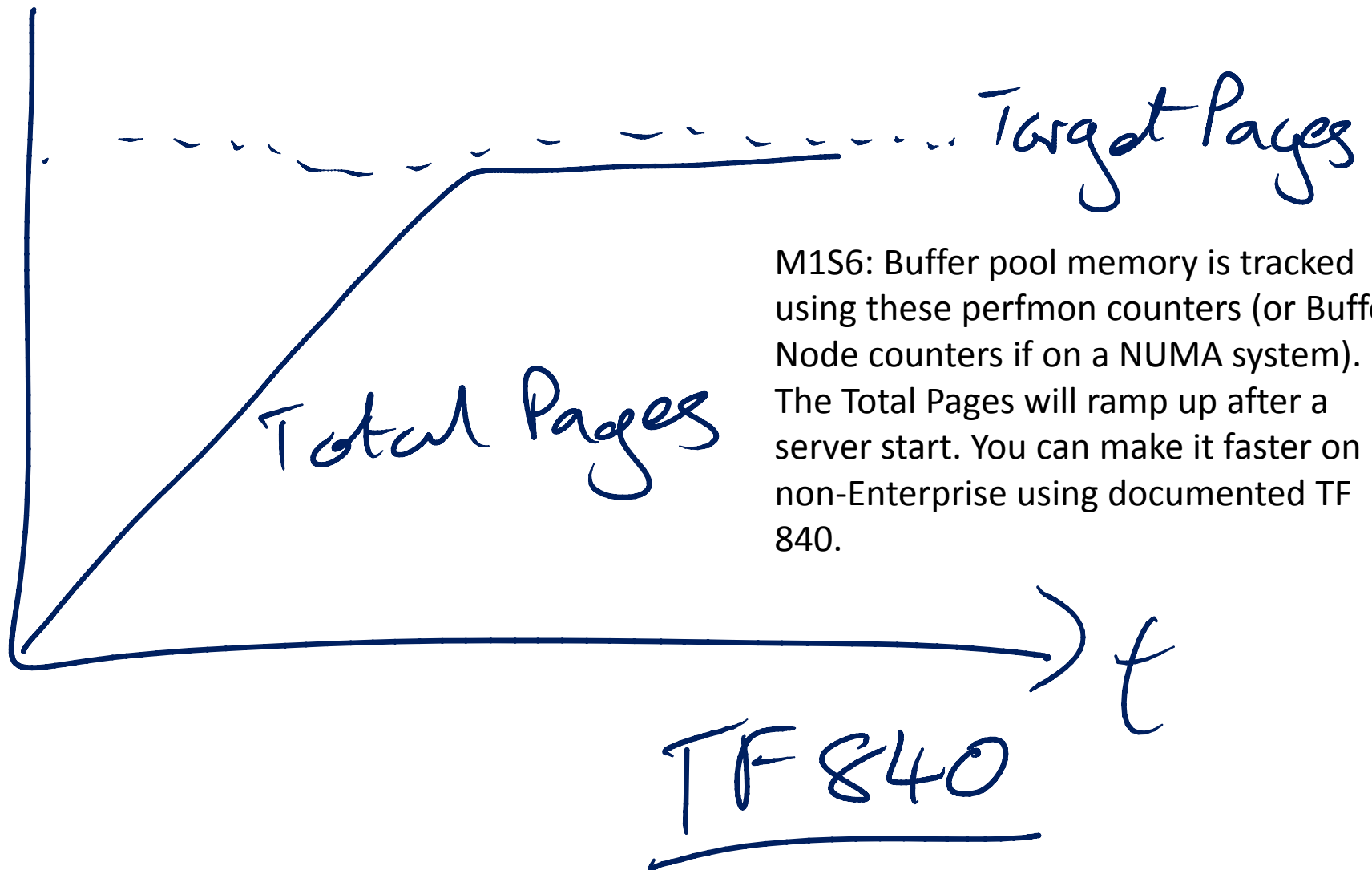
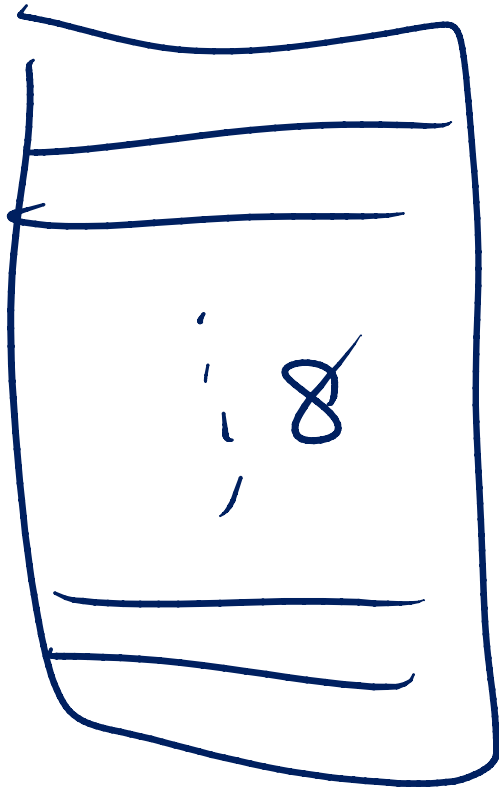
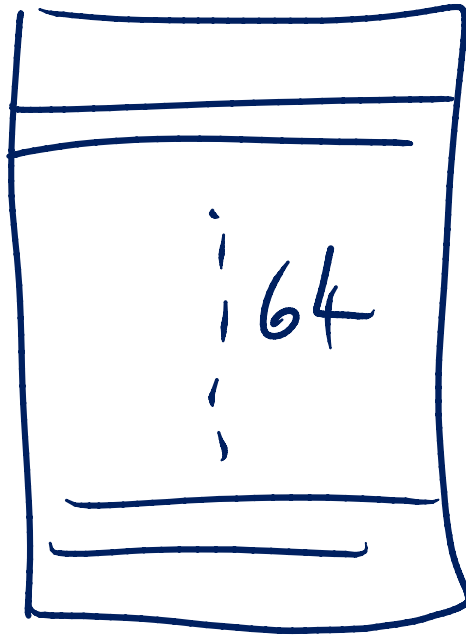




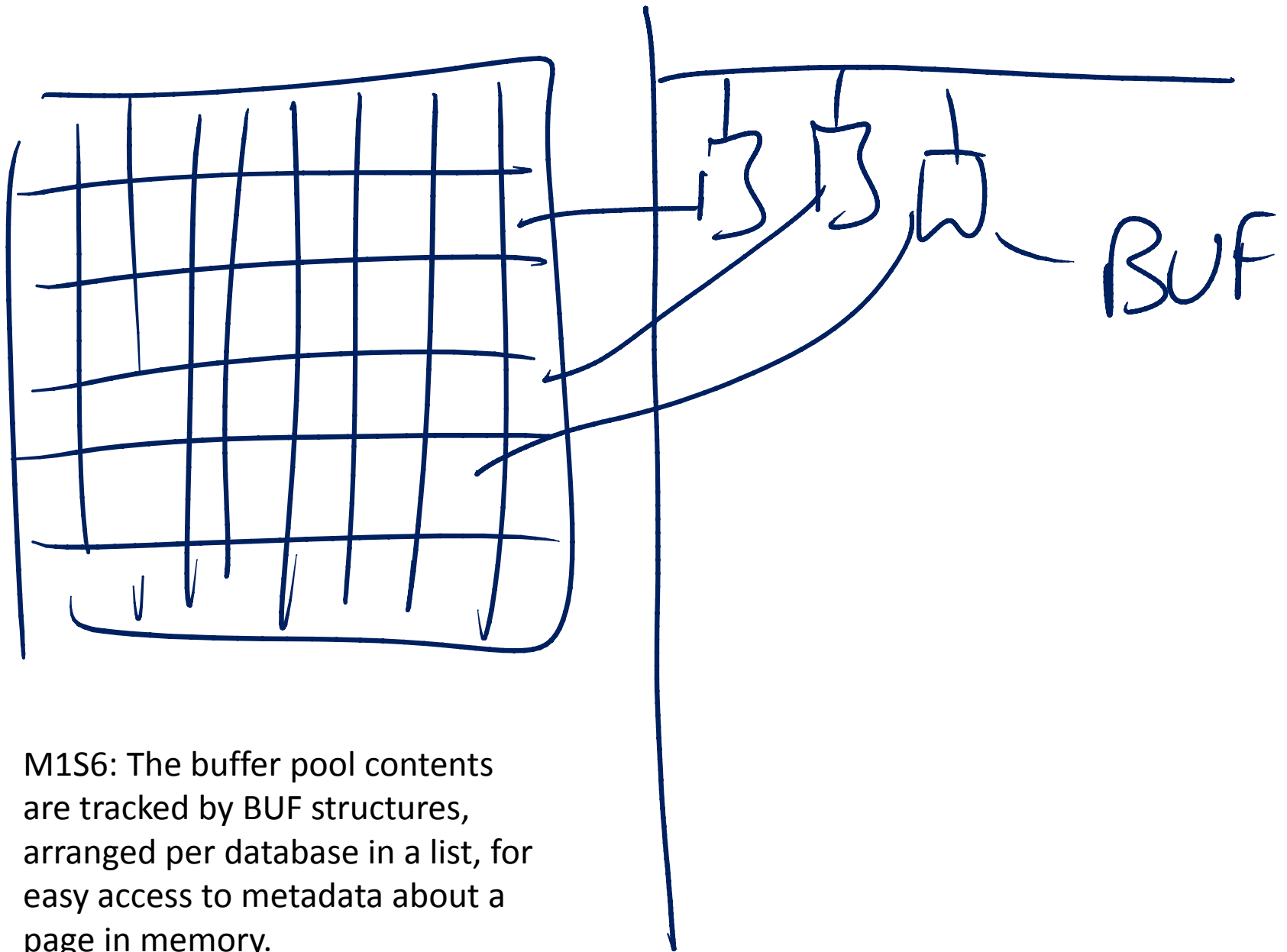
M1S6: Buffer pool memory is tracked using these perfmon counters (or Buffer Node counters if on a NUMA system). The Total Pages will ramp up after a server start. You can make it faster on non-Enterprise using documented TF 840.



100 24 → OFF Row ^{0.5%}



M1S5 Having a large LOB value in row that's not used much makes for large rows and low data density. Choose how to store LOB data wisely.



M1S6: The buffer pool contents are tracked by BUF structures, arranged per database in a list, for easy access to metadata about a page in memory.

BUFFER
DISFAVORING

LRU-K, K=2



M1S6: Buffer pool implements a least-recently-used algorithm, to determine which pages are dropped when there's memory pressure. When memory pressure, lazy writer sweeps through buffers to find LRU ones. Disfavoring is discussed [here](#).

CPU

L1

L2

L3

RAM

I/O CACHE

B. Pool

64/128CB

B.P.E.

I/O SUBSYSTEM

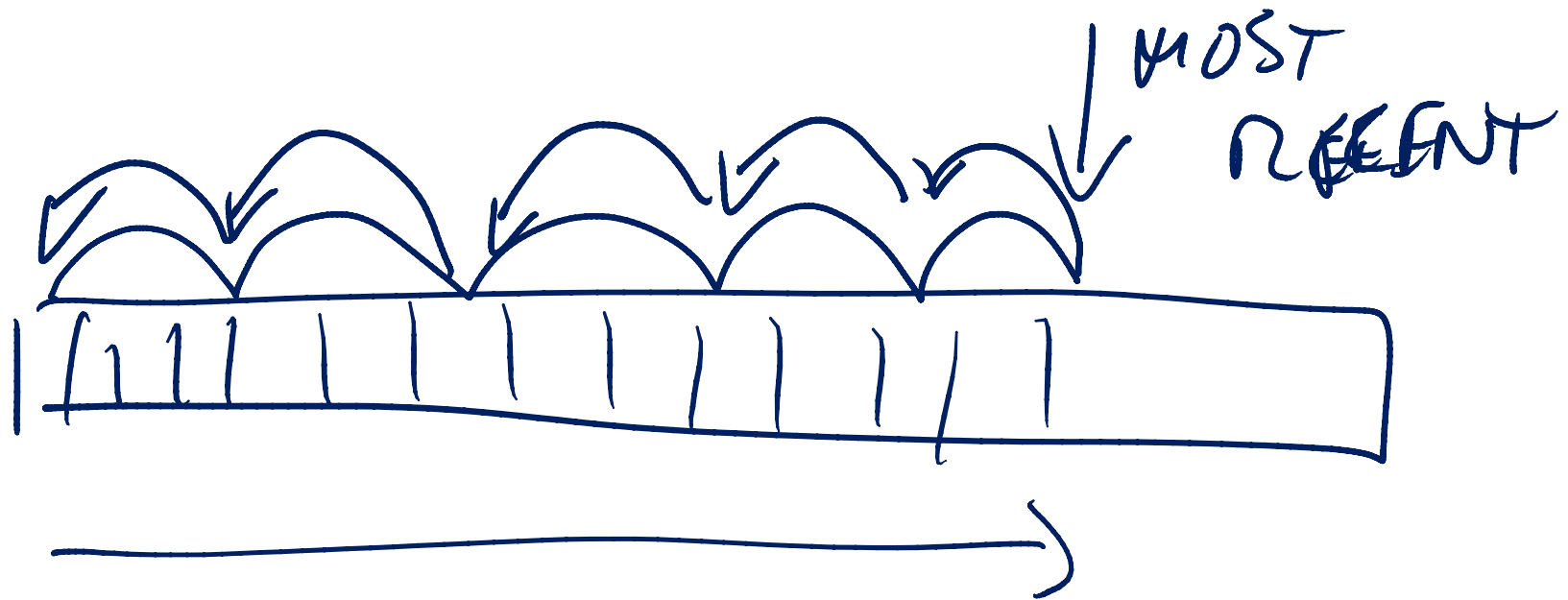
M1 BPE is a faster storage for unchanged pages than being on disk, but slower than being in memory in the buffer pool proper.

CHECKPOINT

DBCC DROPCLEAN BUFFERS

RAMP-UP SCRIPT

M1S6 Discussing ramp-up scripts to warm up the buffer pool, specifically wanting to reset the buffer pool after a large ETL process. Checkpoint forces all dirty pages to go to disk, DBCC DROPCLEANBUFFERS empties the buffer pool, then the ramp-up script refills the buffer pool with the 'working set' of data.



M1S12 Discussing how VLF fragmentation (too many VLFs) can cause slow down of rollbacks. This is because of having to search through the VLFs for a particular VLF sequence number when rolling back log records in the reverse order in a transaction. The more VLFs there are, the longer the search for each log records takes.

PFS

2:1:1

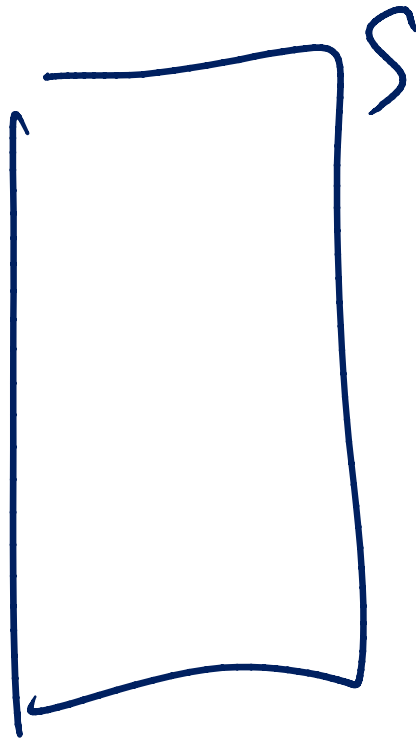
~~SGAM~~

2:1:3

IF 1118

PFS

M1S25 Explaining PFS and SGAM contention in tempdb and how adding multiple files spreads the contention over multiple files, thus reducing it.



RES	G.L.	P.Q
-----	------	-----

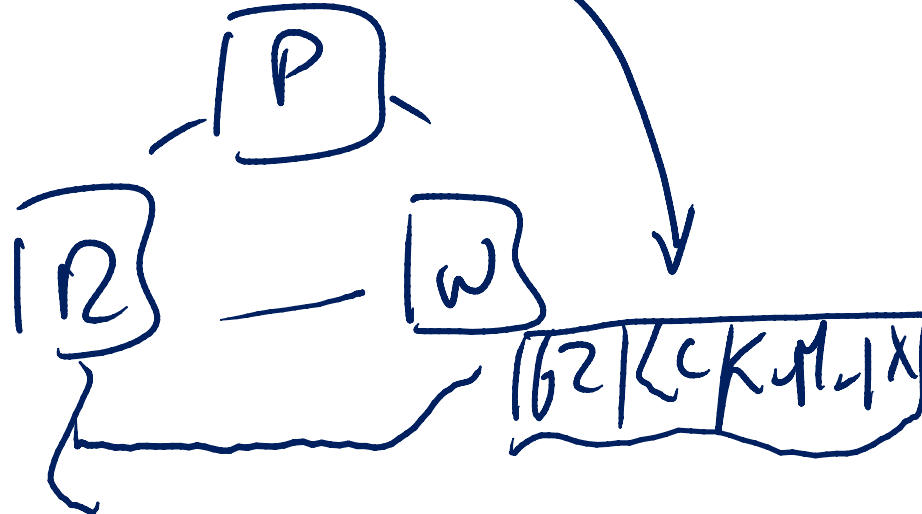
GRANTED
LIST

61	X
62	X

PENDING
QUEUE

62	X
64	X

M7: see next slide



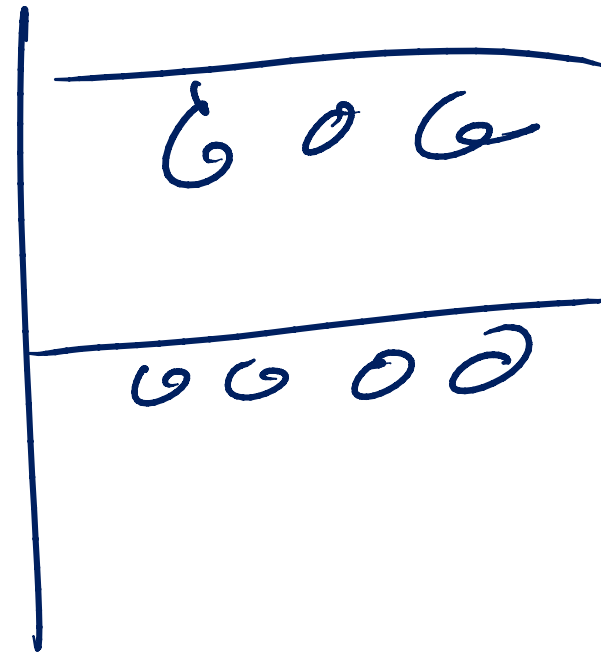
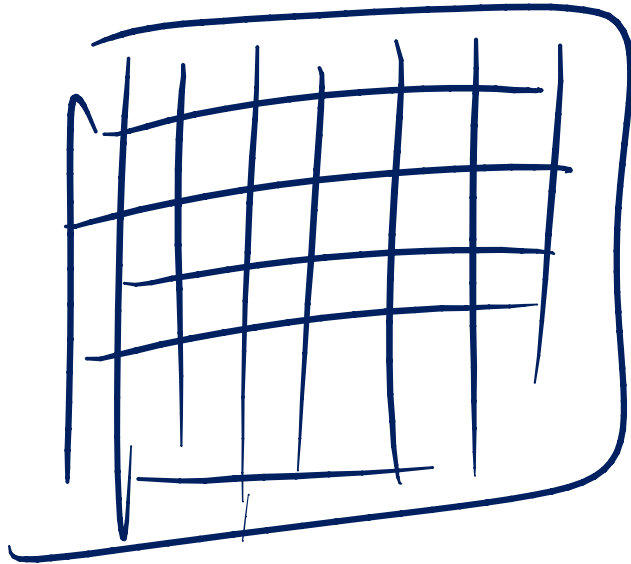
M7 Explaining how the lock pending queue works.

At time:

X: Thread 61 holds the lock in S mode

X+1: Thread 62 tries to acquire the lock in IX 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.

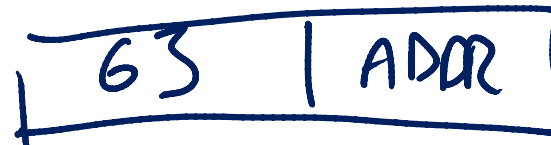
X+2: Thread 61 releases the lock. It checks the pending queue for locks that can now be granted ownership of the lock (taking into account any other threads still holding the lock). In this case, thread 62 is granted the lock in IX mode. Thread 62's callback routine is called, signaling it and moving it to the runnable queue on its scheduler.



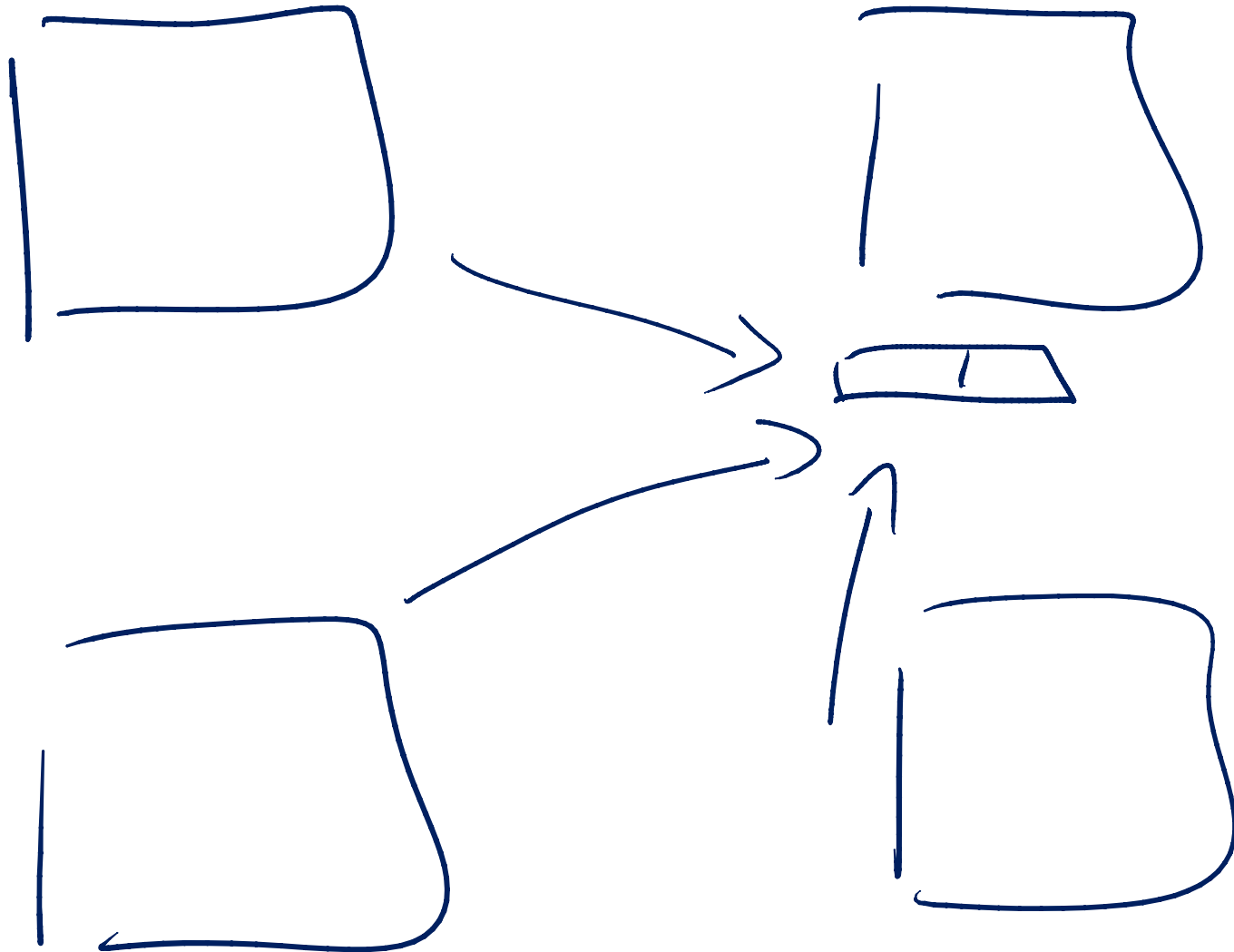
I/O COMPLETION
ROUTINE



ASYNC I/O

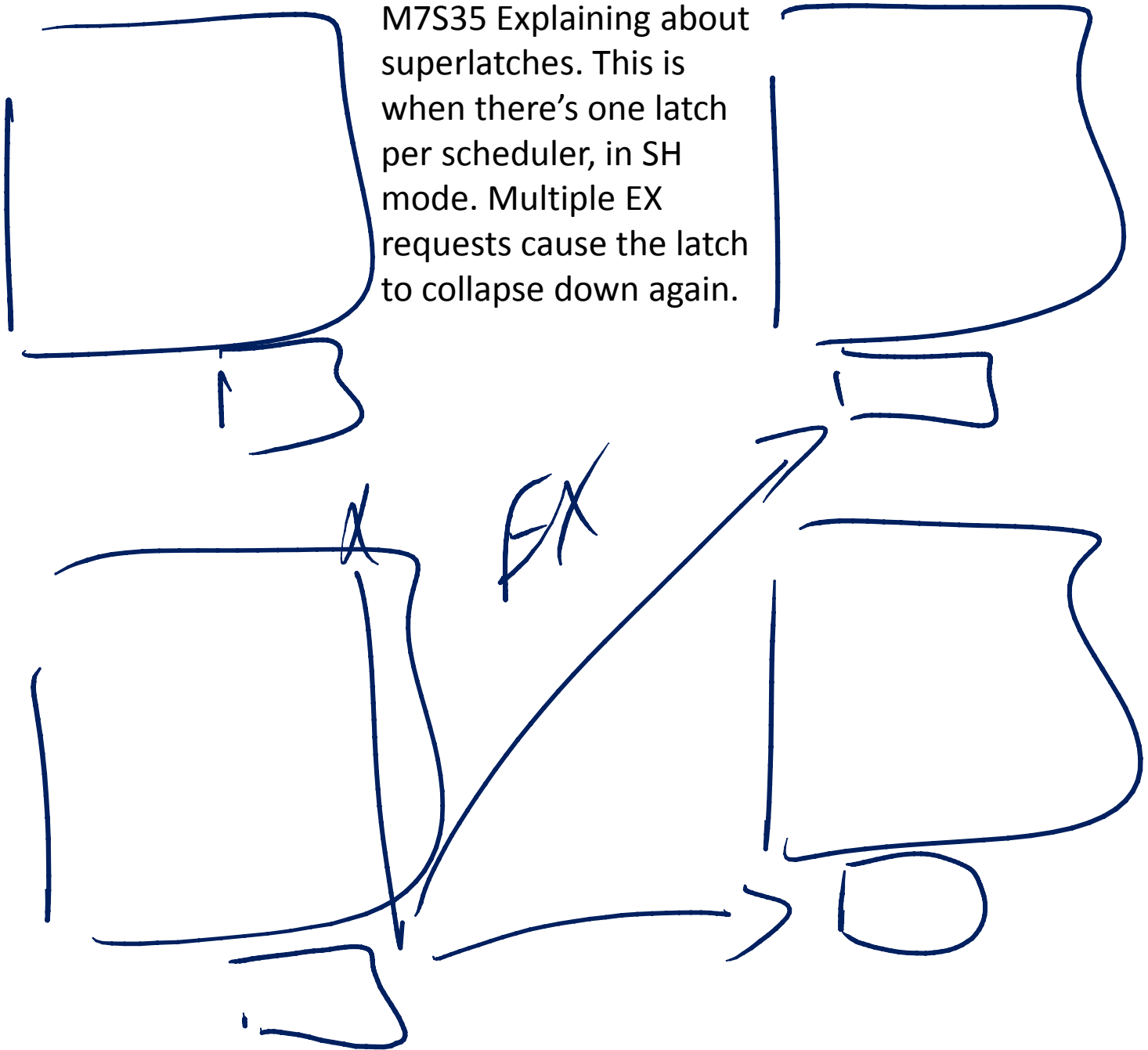


M7: A similar mechanism works for when a page must be read from disk.

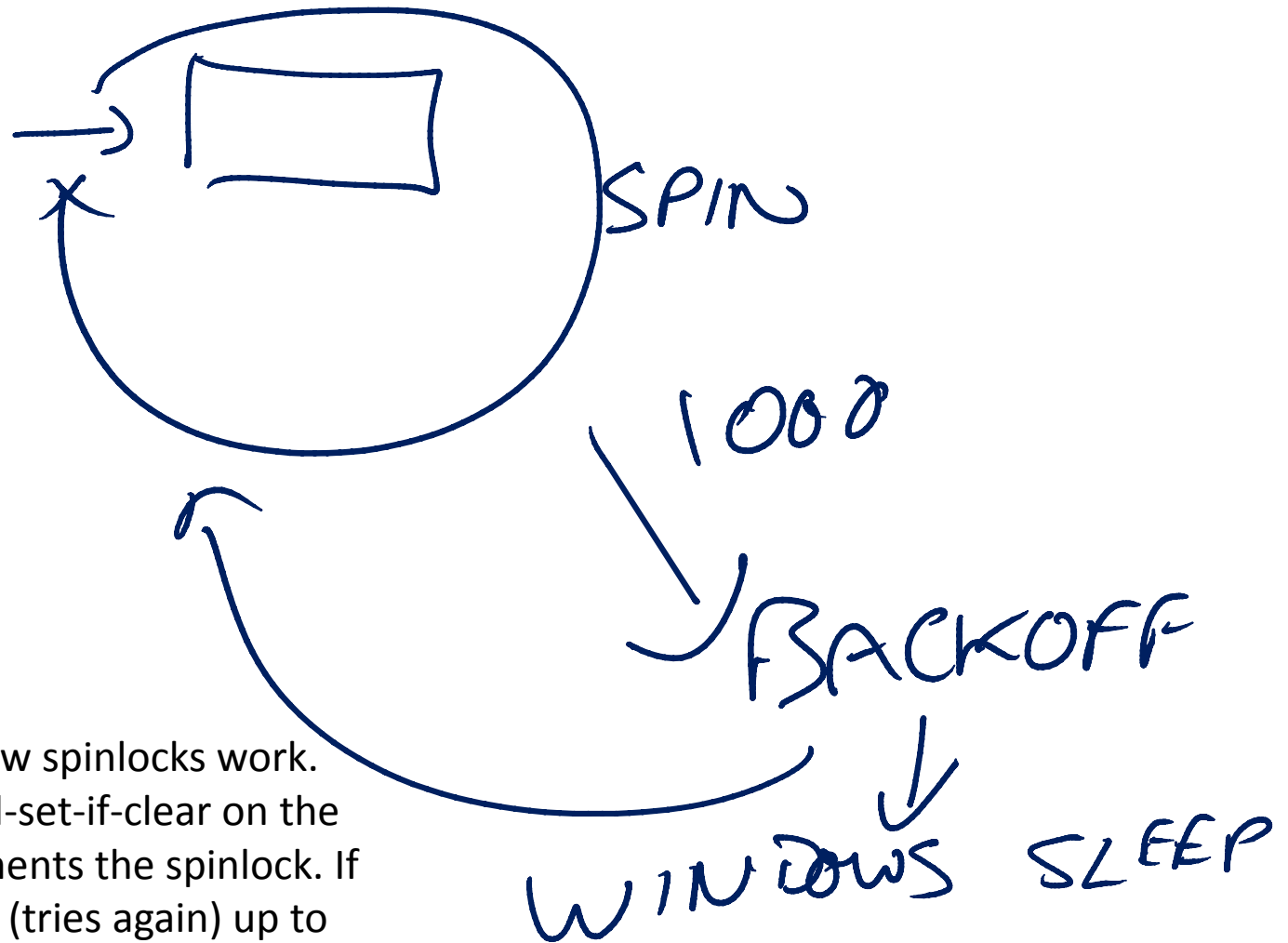


M7S35 Explaining about
superlatches. This is when
there's only one latch...

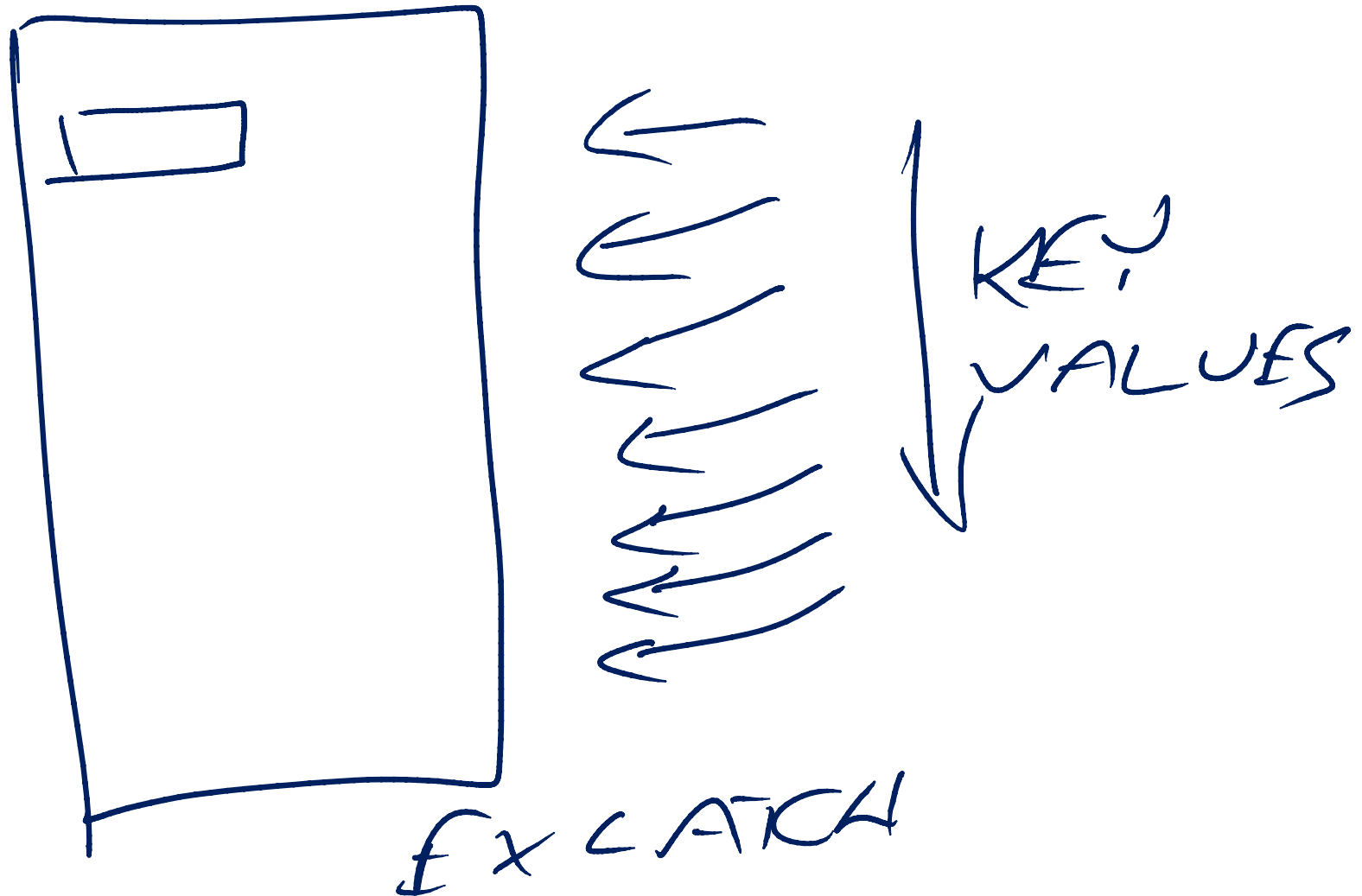
M7S35 Explaining about
superlatches. This is
when there's one latch
per scheduler, in SH
mode. Multiple EX
requests cause the latch
to collapse down again.



INTERLOCKED COLLISIONS



M7S38 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.



M7S48 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 EX latch on the page, which becomes a bottleneck. If the clustered index has an int/bigint identity, and the row size is small enough, it could result in an insert hotspot.