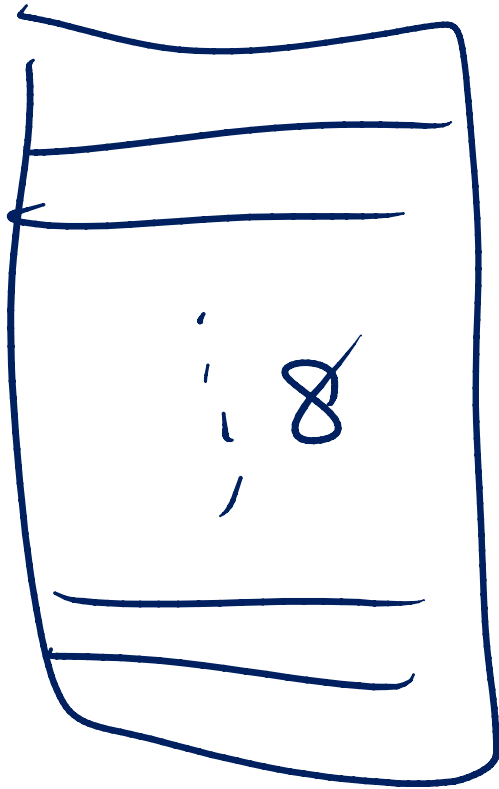
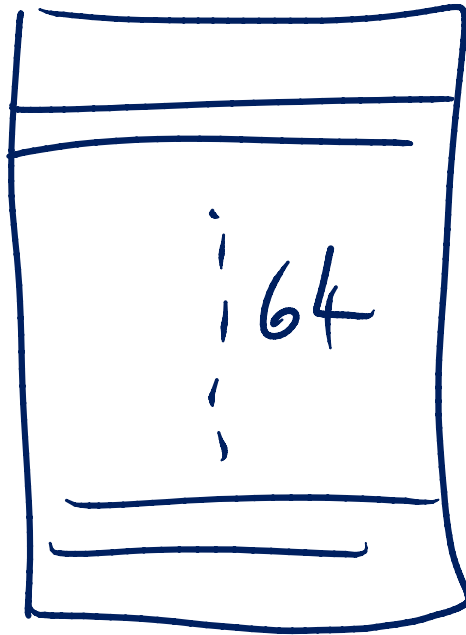
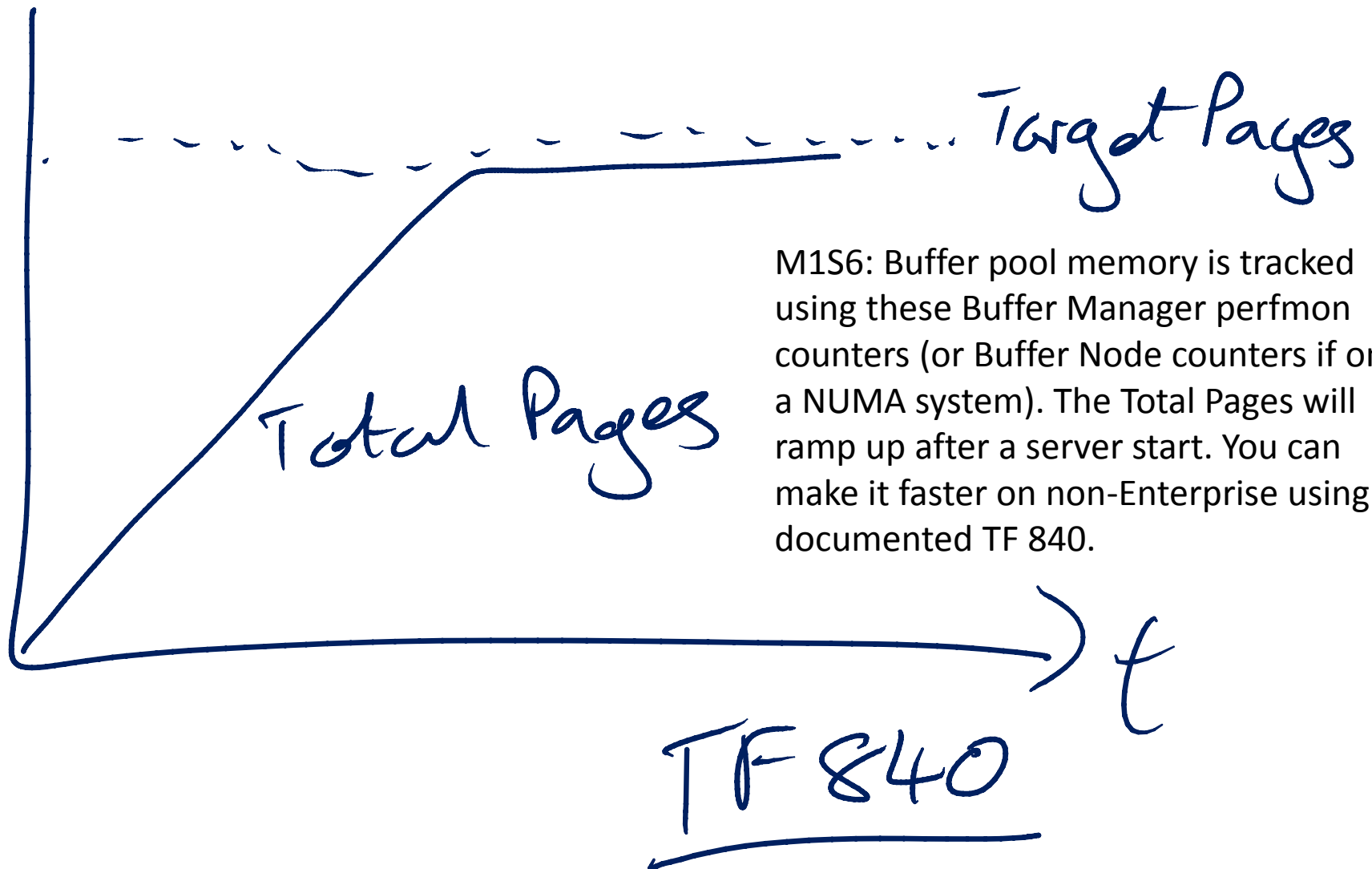




100 24 → OFF Row ^{0.5%}



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: Buffer pool memory is tracked using these Buffer Manager 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.

TABLE

INDEX A

10,000

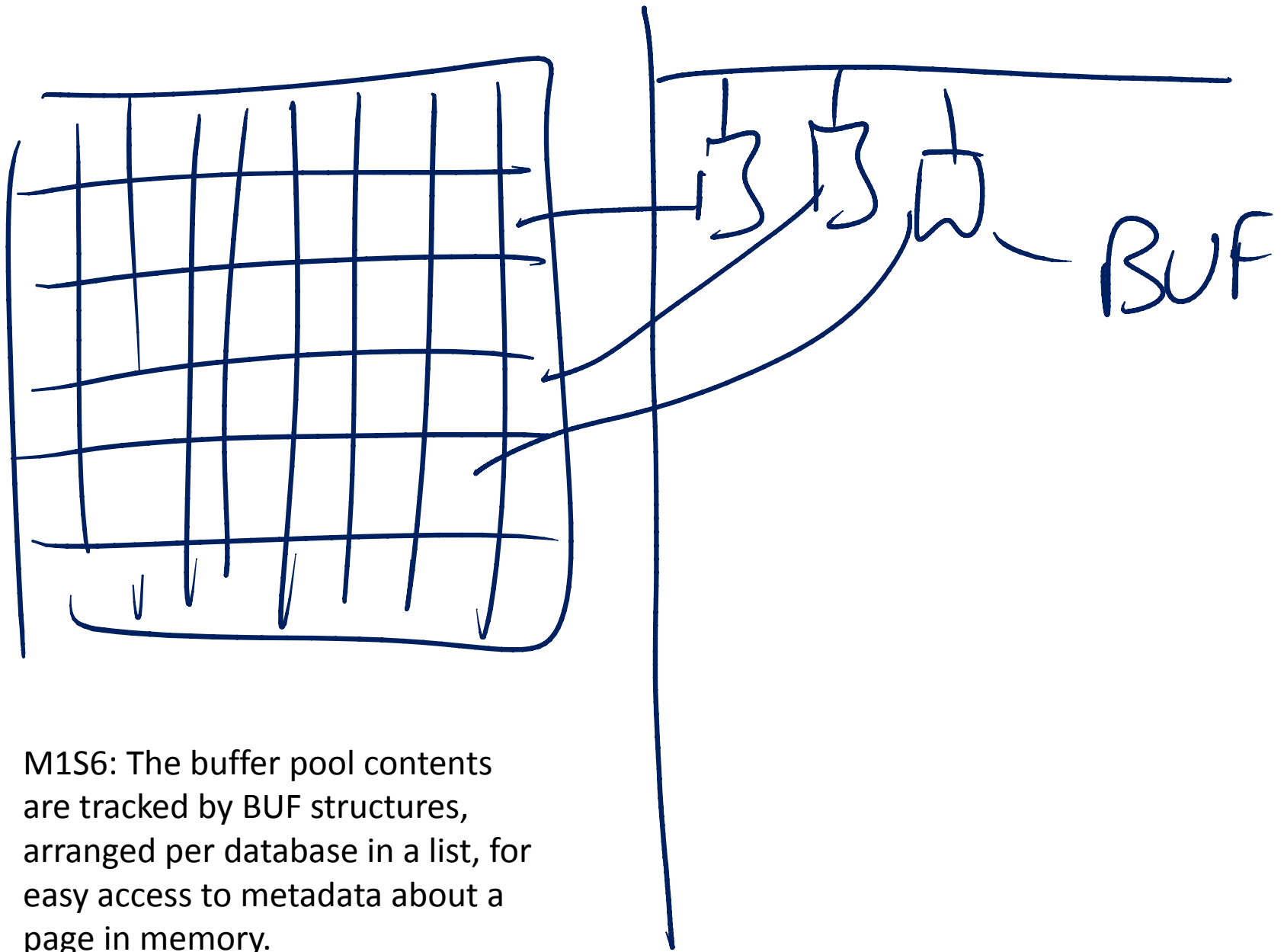
INDEX B

100,000

IN MEM

M1S6 The Query Optimizer does not know what's in memory, and makes plan choices based on index size, not what's in memory. See editorial in

<https://www.sqlskills.com/insidercontent/201609/20160926newsletter.pdf> for example



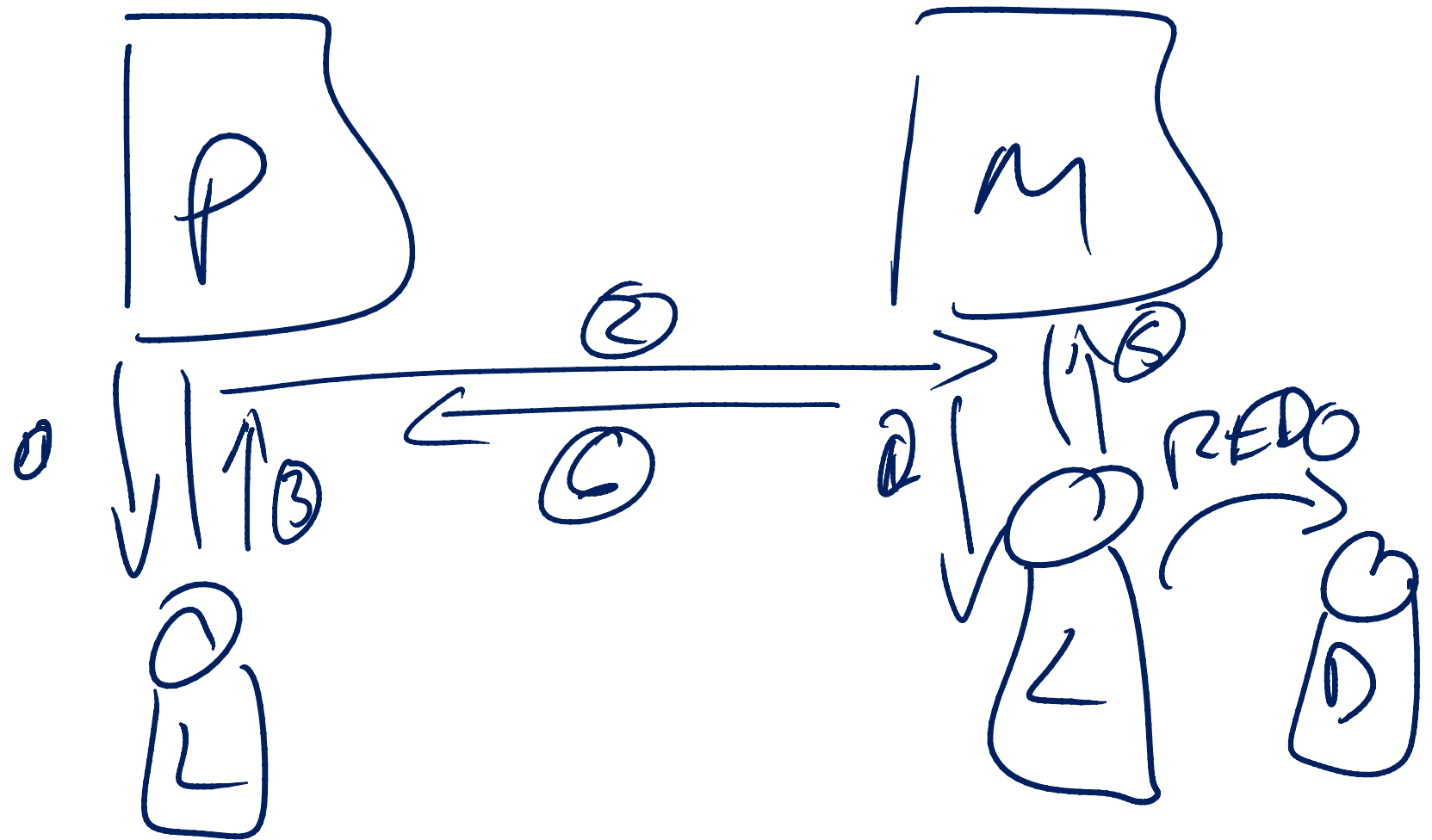
M1S6: The buffer pool contents are tracked by BUF structures, arranged per database in a list, for easy access to metadata about a page in memory.

BUFFER
DISFAVORING

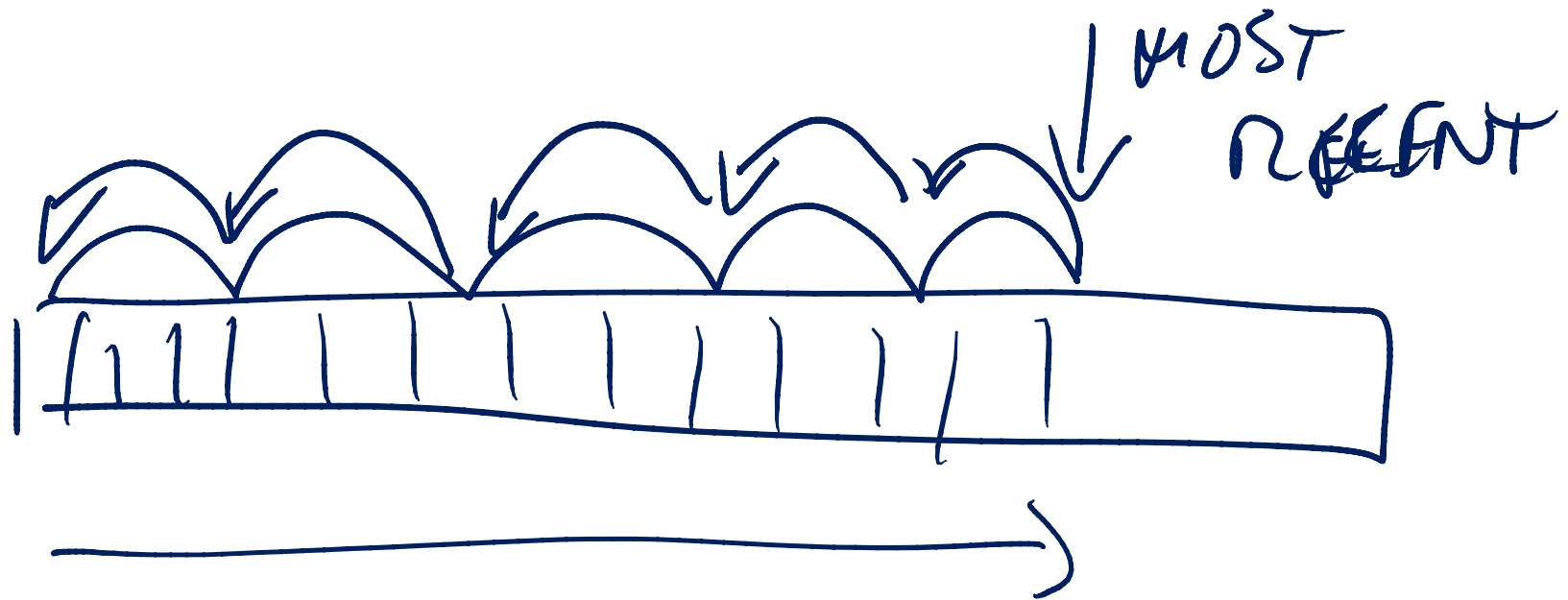
LRU-K, K=2



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](#).



M1S12-13 Showing what happens when a transaction commits and the final log block for it is flushed to disk, when sync DBM or AG is involved. The local I/O will complete first but the transaction commit cannot complete until the mirror acknowledges the remote I/O.



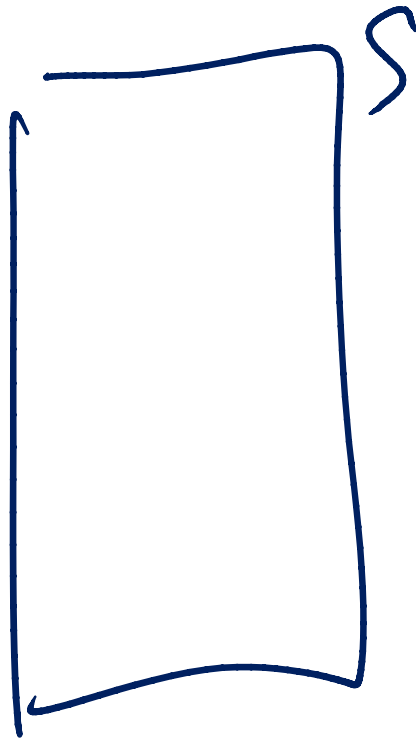
M1S12 Discussing how VLF fragmentation (too many VLFs) can cause slow down of rollbacks. This is because of having to search through the VLFs for a particular VLF sequence number when rolling back log records in the reverse order in a transaction. The more VLFs there are, the longer the search for each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

PFS 2:1:1

~~SGAM~~ 2:1:3 IFULL

PFS

M1S26 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
-----	------	-----

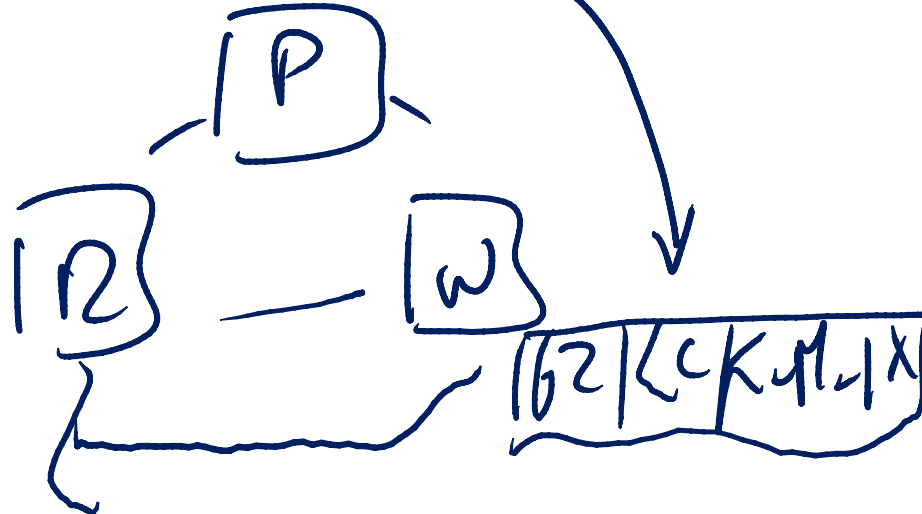
GRANTED
LIST

61	X
62	X

PENDING
QUEUE

62	X
64	X

M6: see next slide



M6 Explaining how the lock pending queue works.

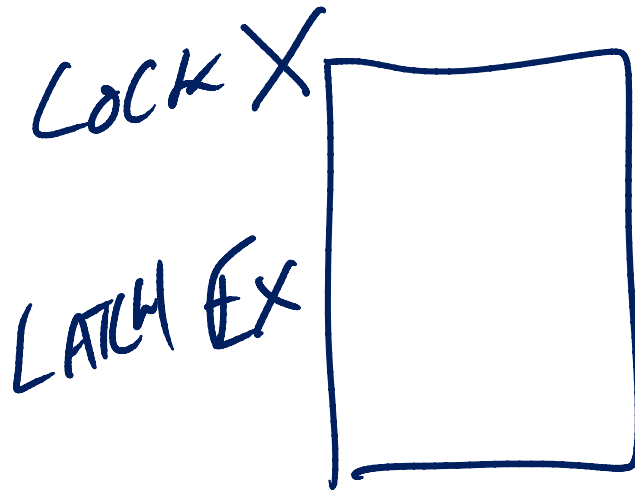
At time:

X: Thread 61 holds the lock in S mode

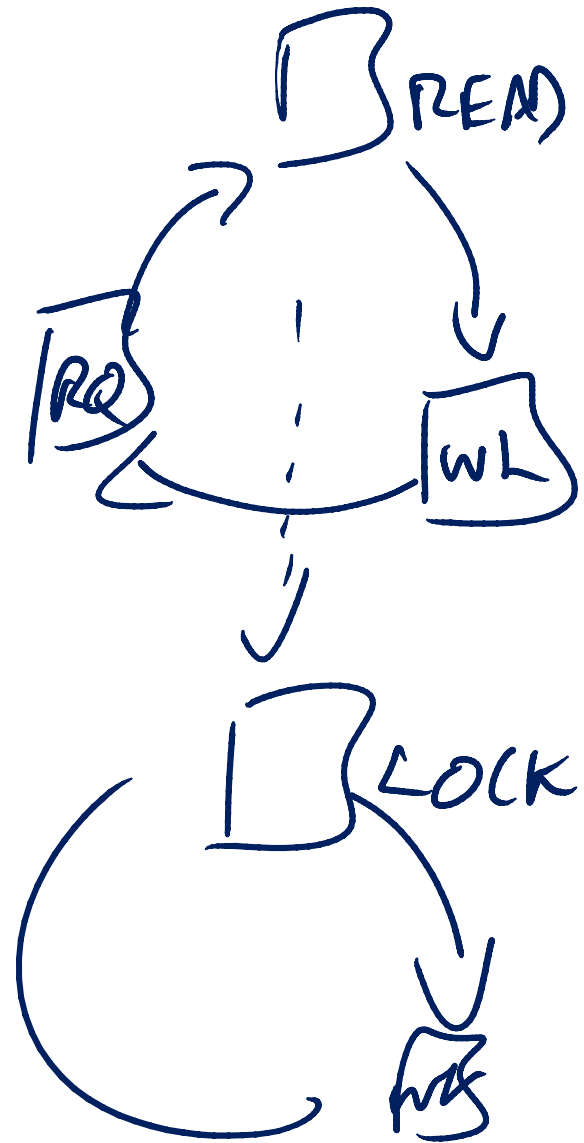
X+1: Thread 62 tries to acquire the lock in IX and can't. It registers a callback routine for itself on the pending queue in the lock value block. Then it suspends itself, moves off the CPU and onto the waiter list.

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.

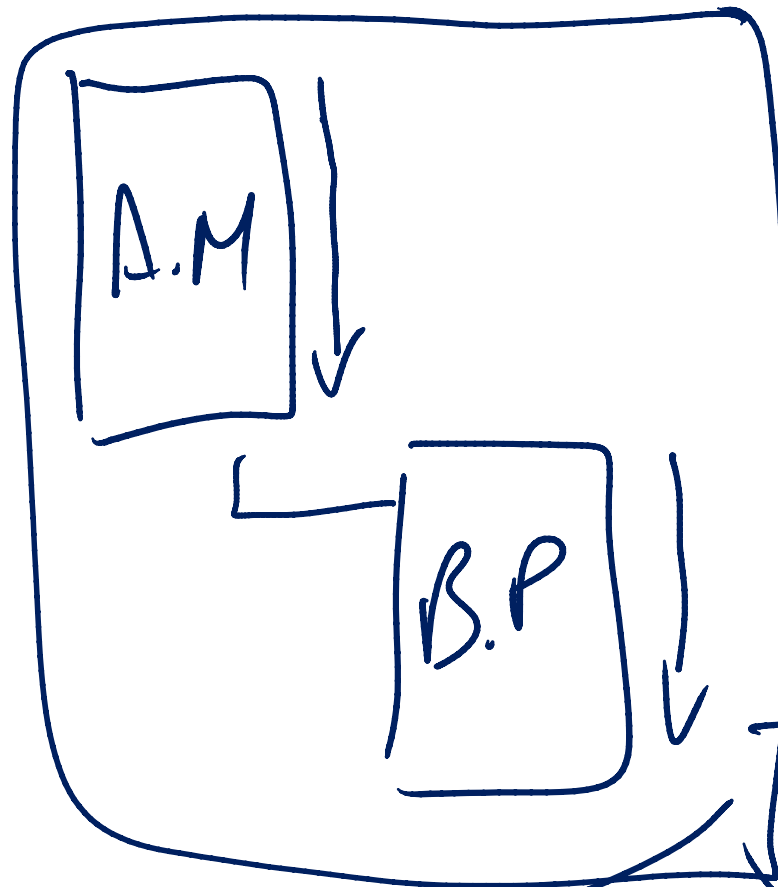
PAGLOCK
UPDATE



READ
LOCK
LATCH



M6: A thread may need to acquire multiple resources in a short amount of time. E.g. changing a page will require a lock, reading the page into memory, and a latch. Each of these could mean a trip to the Waiter List while the resource is waited for.



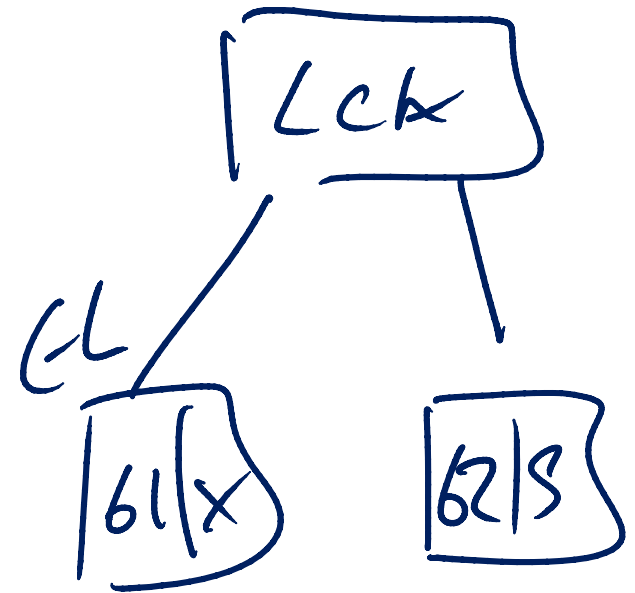
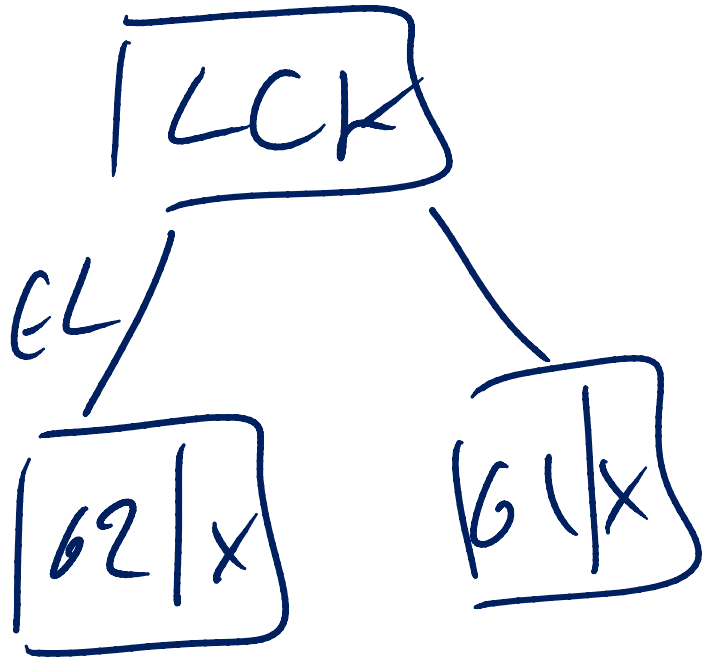
PROCESSOR

M6S17 Showing how a thread on a scheduler that's just about to be suspended will help out 'scheduling' by looking in the buffer pool to see if any I/Os from other threads on that scheduler have completed. If so, it

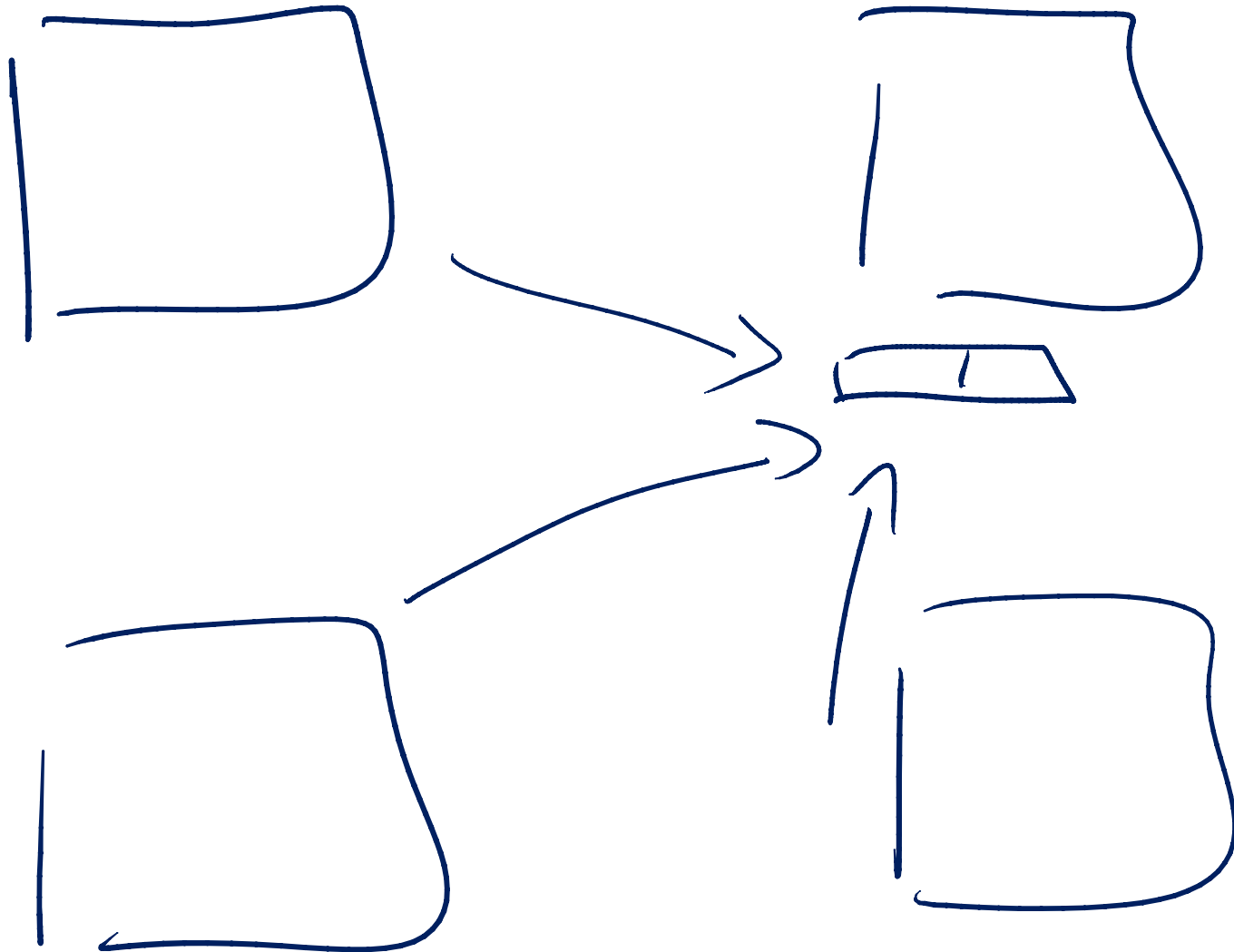
→ I/O

will signal one thread before suspending itself)



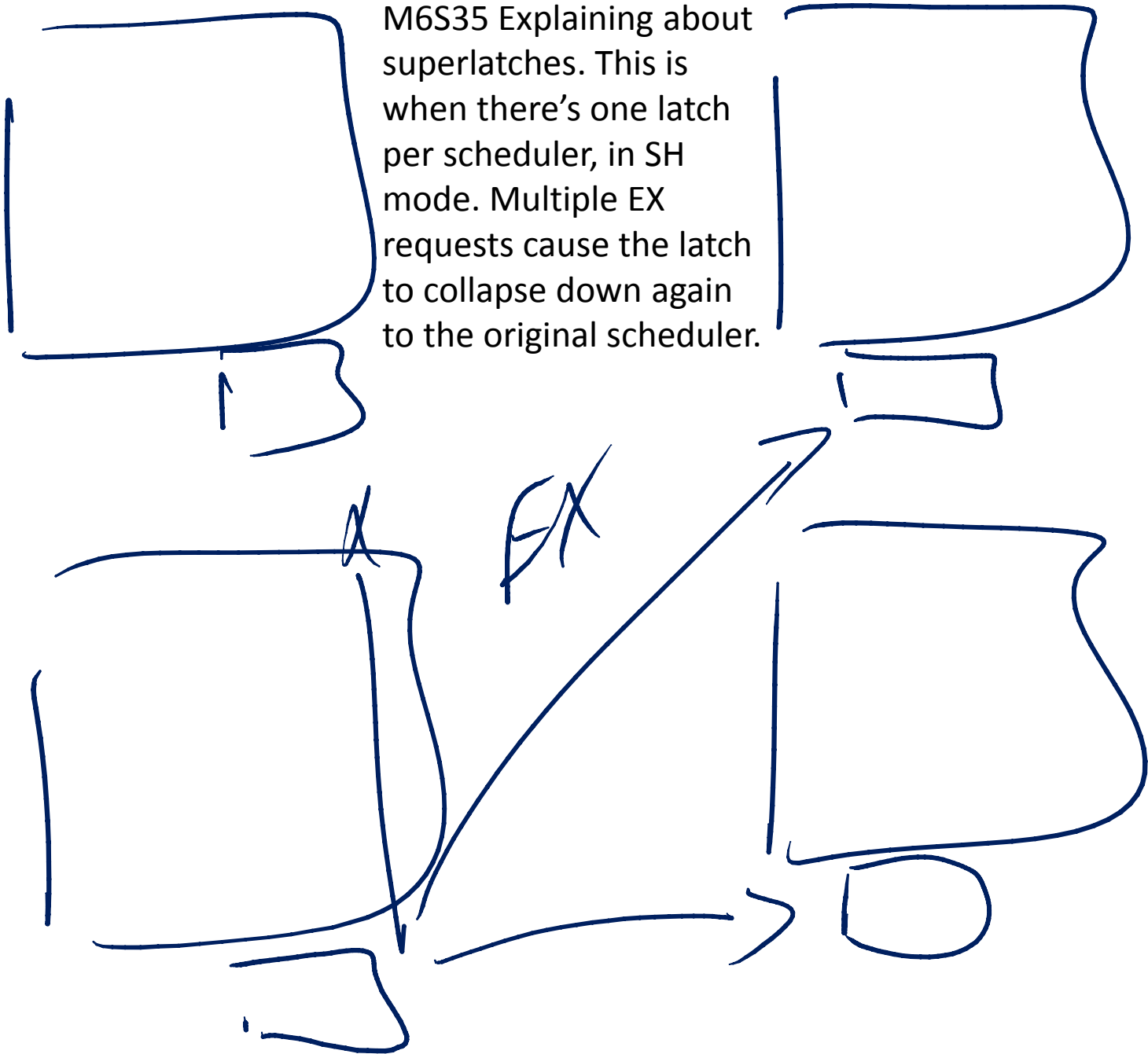


Deadlock scenario: Spid 62 X locks the first page. Spid 61 X locks the second page. (GL = Granted List of locks for that resource.) Spid 61 then tries to X lock the first page. Spid 62 then tries to S lock the second page. That's a deadlock. The deadlock monitor notices and picks a victim, based on who's done the least amount of work. Let's say it's 62. Thread 62 (waiting for the S lock) is signaled (moves to the runnable queue) and the deadlock monitor has set a bit in Thread 62's memory that indicates it's been chosen as a deadlock victim. Thread 62 then initiates rolling back the transaction that it's part of.

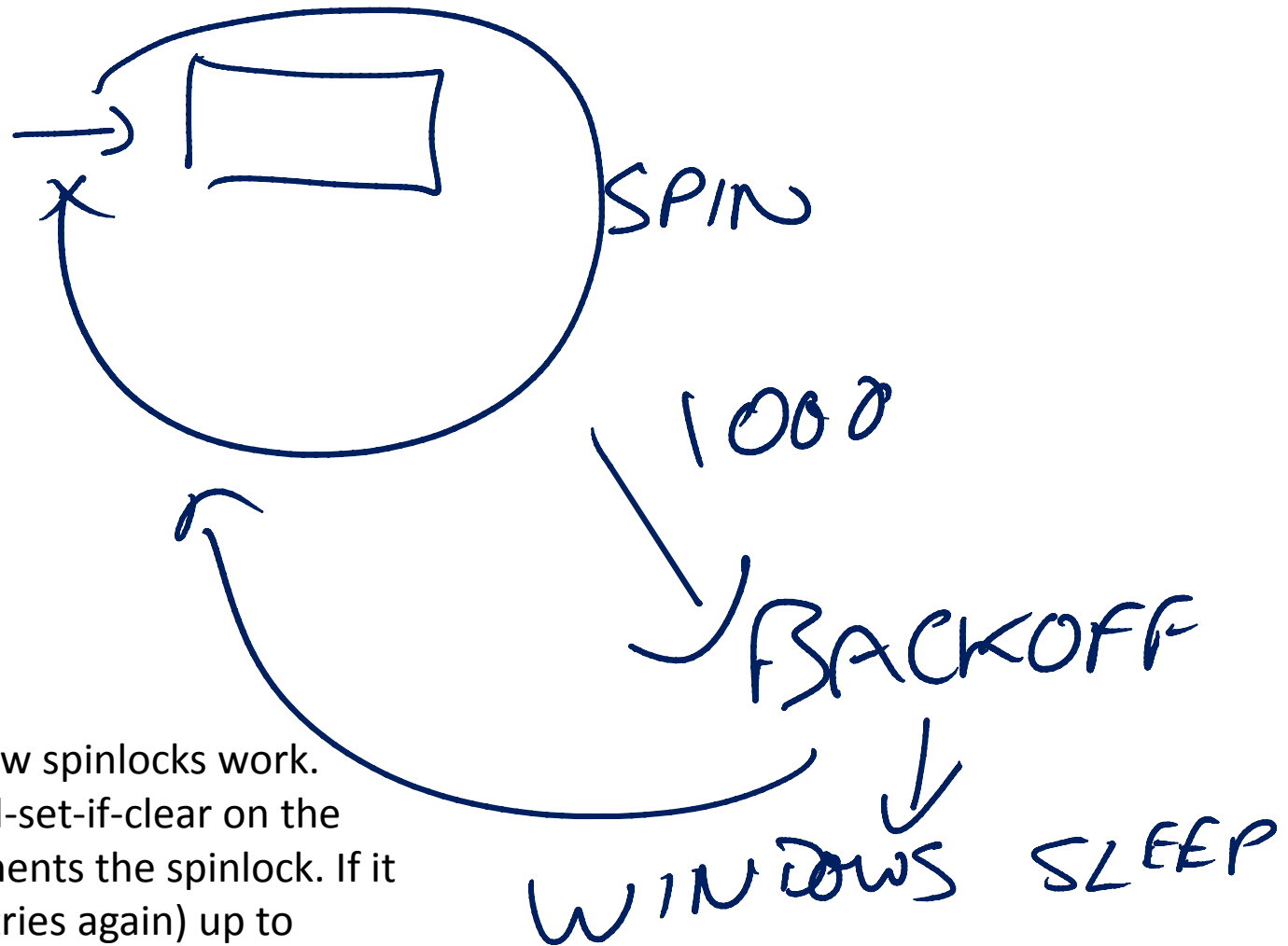


M6S35 Explaining about superlatches. This is when there's only one latch and it's owned by one scheduler (big boxes are scheduler)...

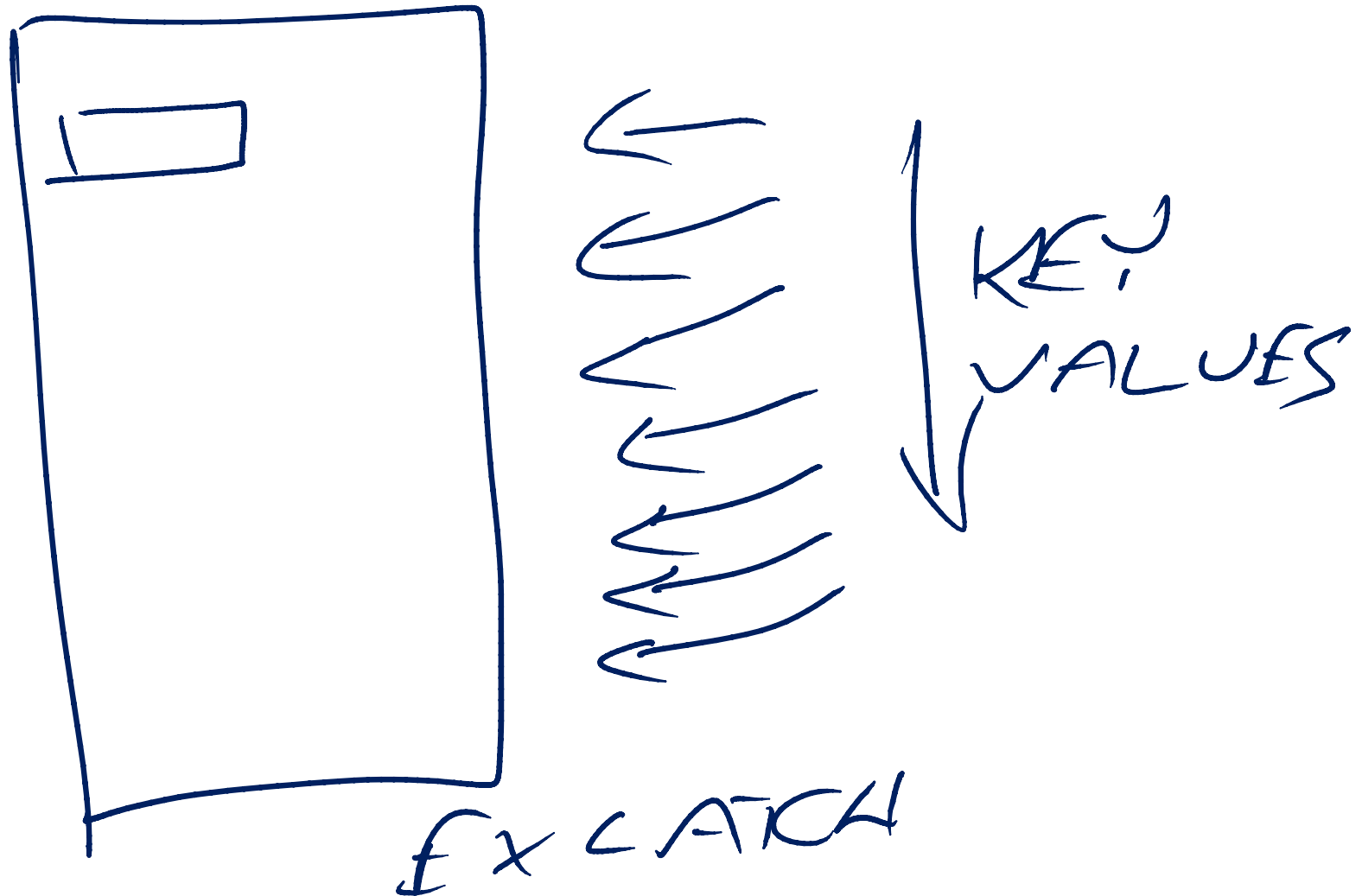
M6S35 Explaining about
superlatches. This is
when there's one latch
per scheduler, in SH
mode. Multiple EX
requests cause the latch
to collapse down again
to the original scheduler.



INTERLOCKED COLLISIONS



M6S40 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
(generally) 1000 times and then backs off.



M6S49 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.