

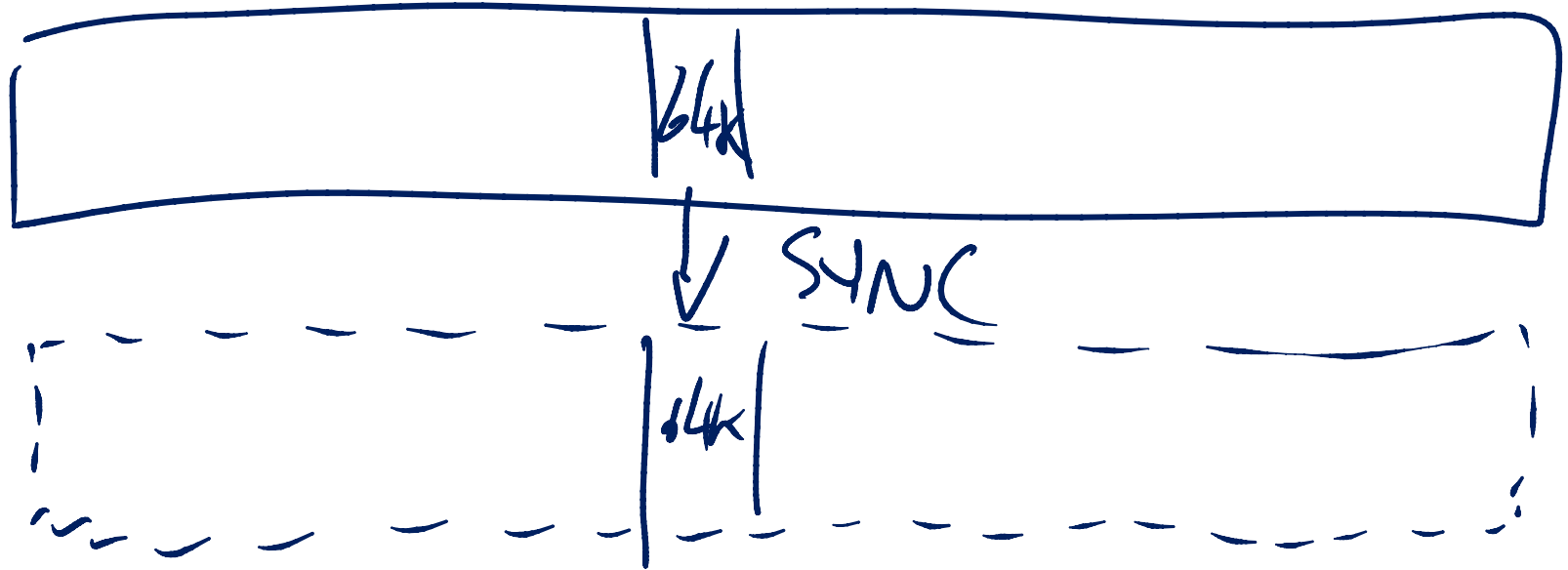
int c[100000000] sparse;

c[104] = 6; [104][6]

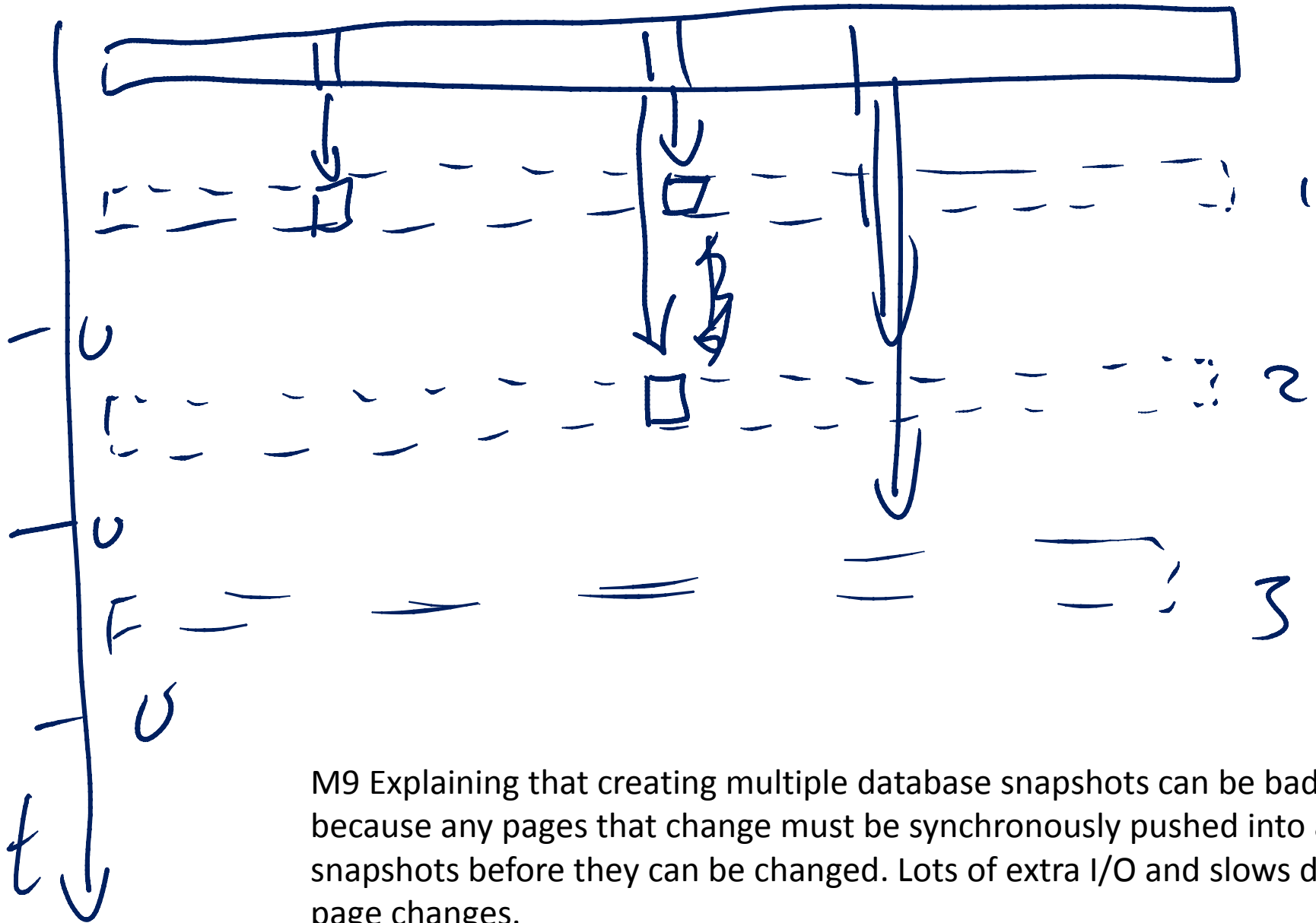
d = c[103]; [103][56]

M9 Drawing an analogy between sparse files in NTFS and sparse arrays in various programming languages. You can define an array of arbitrary size but only the populated elements take up memory. Any elements you ask for that haven't been populated result in an error.

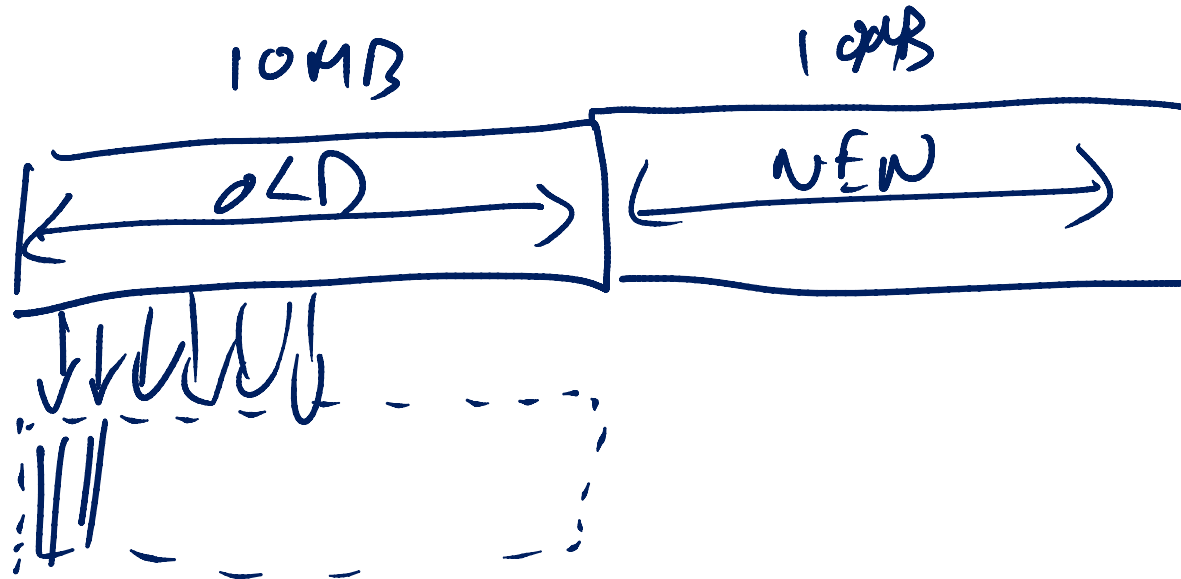
200CB



M9 Explaining that the NTFS sparse file keeps track of which file offsets have been written to and for how much. Also, the sparse file is not kept sorted by offset that's been written to. The index at the start takes care of that.

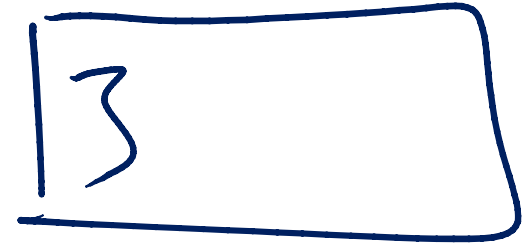
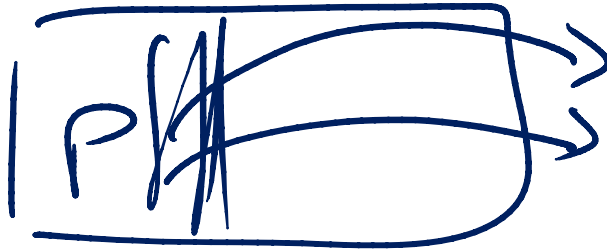


M9 Explaining that creating multiple database snapshots can be bad because any pages that change must be synchronously pushed into all snapshots before they can be changed. Lots of extra I/O and slows down page changes.



M9 Explaining that if a snapshot is created when the source database is 10MB, and then a 10MB index is rebuilt, the source datafile will grow by 10MB but none of the new pages will be pushed into the snapshot, as they didn't exist when the snapshot was created. A few metadata pages will be pushed into the snapshot. When the original pages occupied by the index are overwritten, they will be pushed into the snapshot.

SYSTEM
TABLE

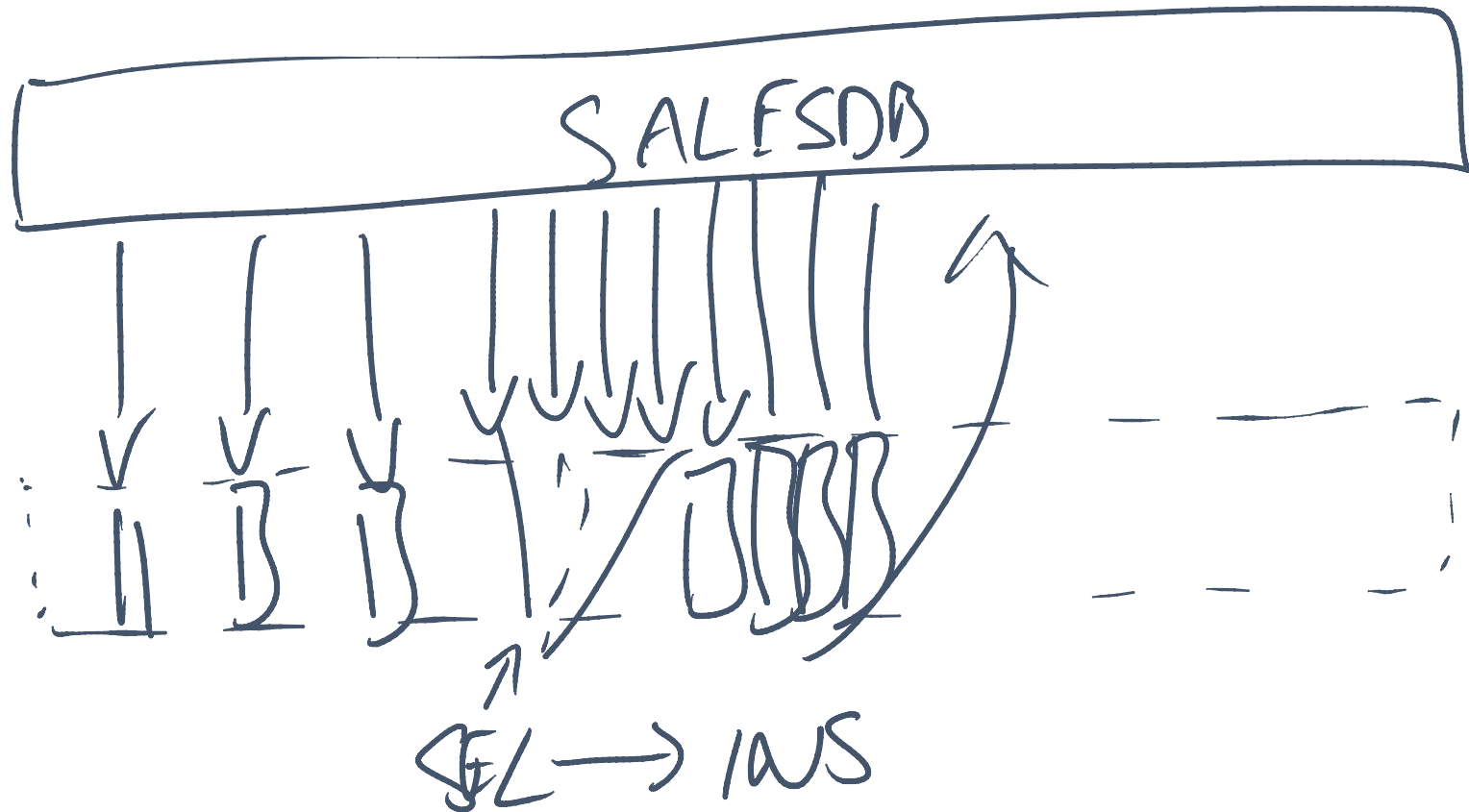


FR-1



SNAP

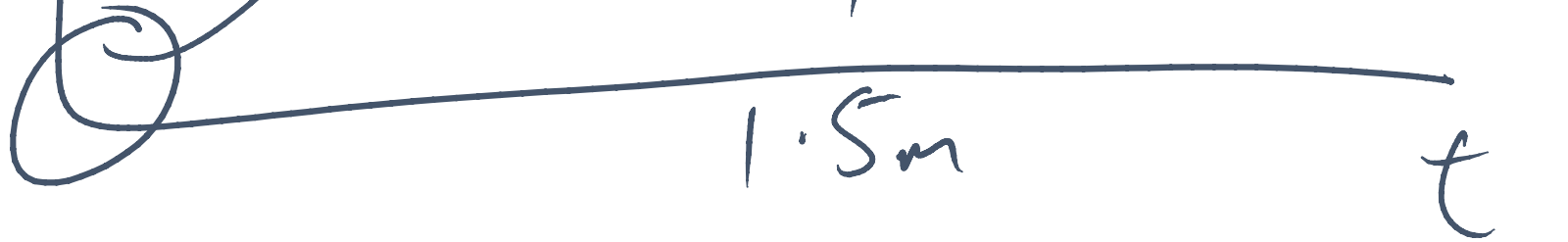
M9 Explaining that after a snapshot is created, if another file is added to the database, it doesn't exist as far as the snapshot is concerned.

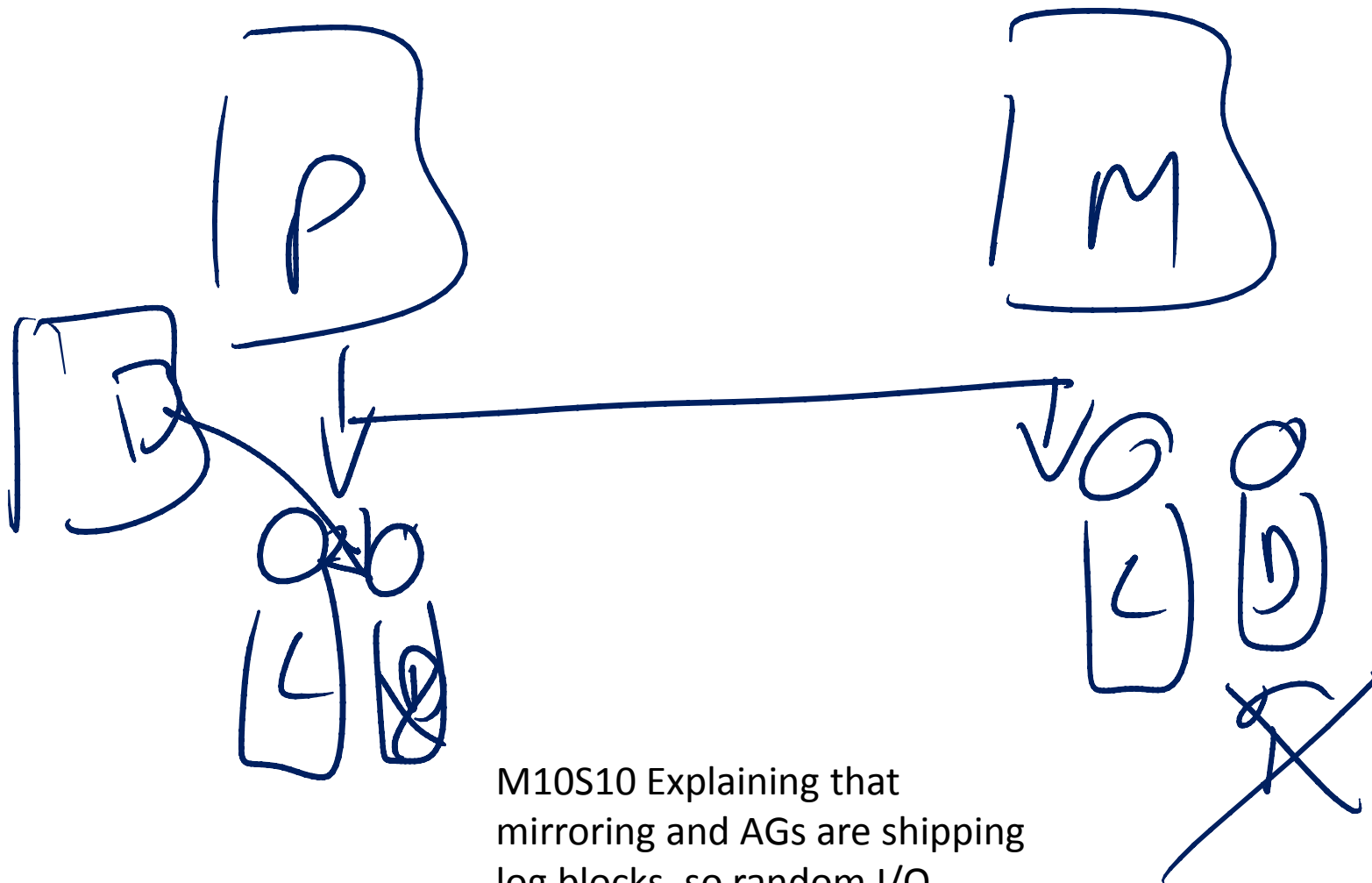


M9 Explaining the demo on recovering data from a snapshot. After dropping the table, the snapshot doesn't grow because the pages from the dropped table haven't been changed yet. As we repopulate the newly-created table though, it's pulling unchanged pages from the source database through the context of the snapshot to populate the new table, thus changing pages and pushing them into the snapshot. A bit of a mind-bender.

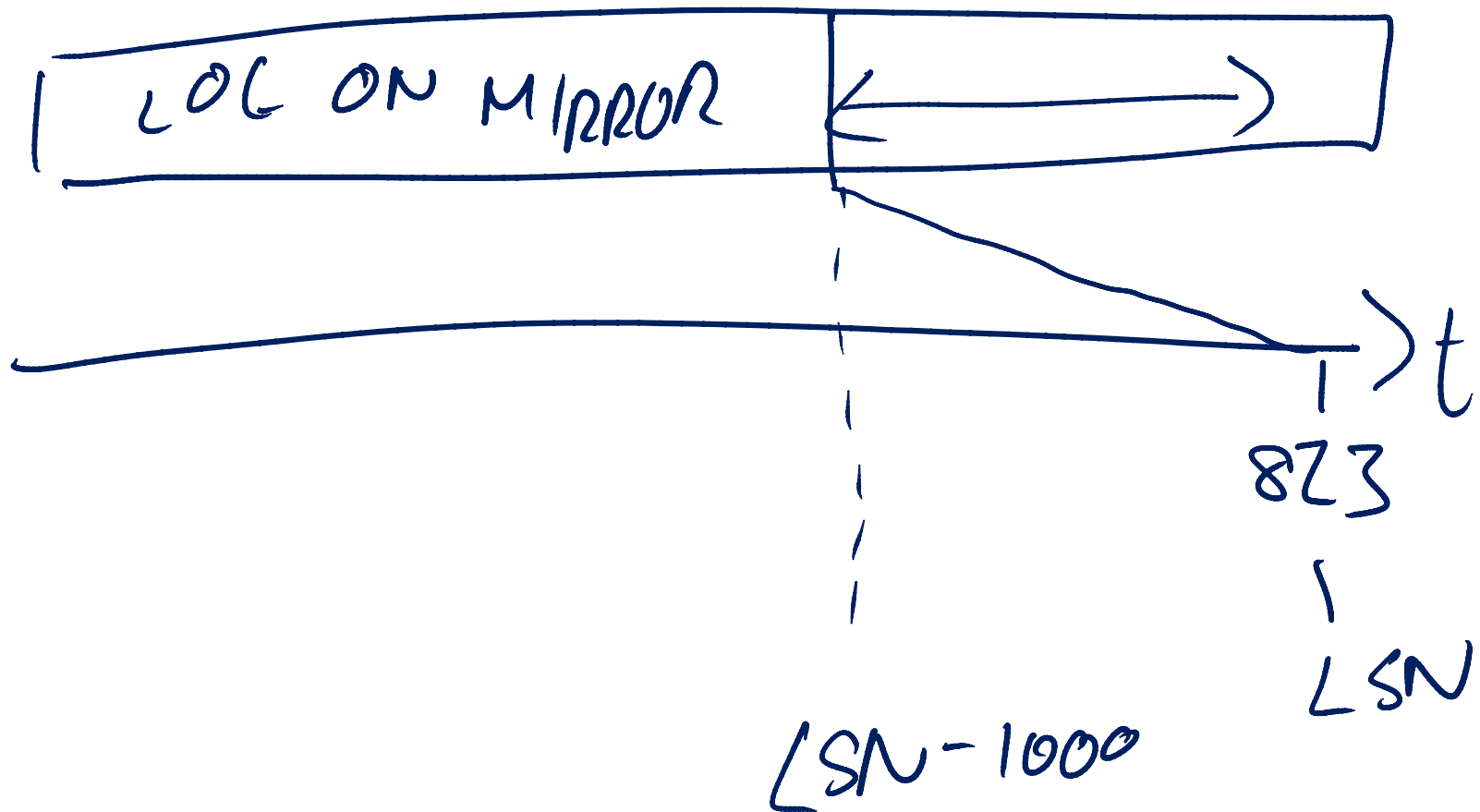
FAILURES

M10S8 The MTBF curve of drives.
Someone's going to be in the low
MTBF section on the left...

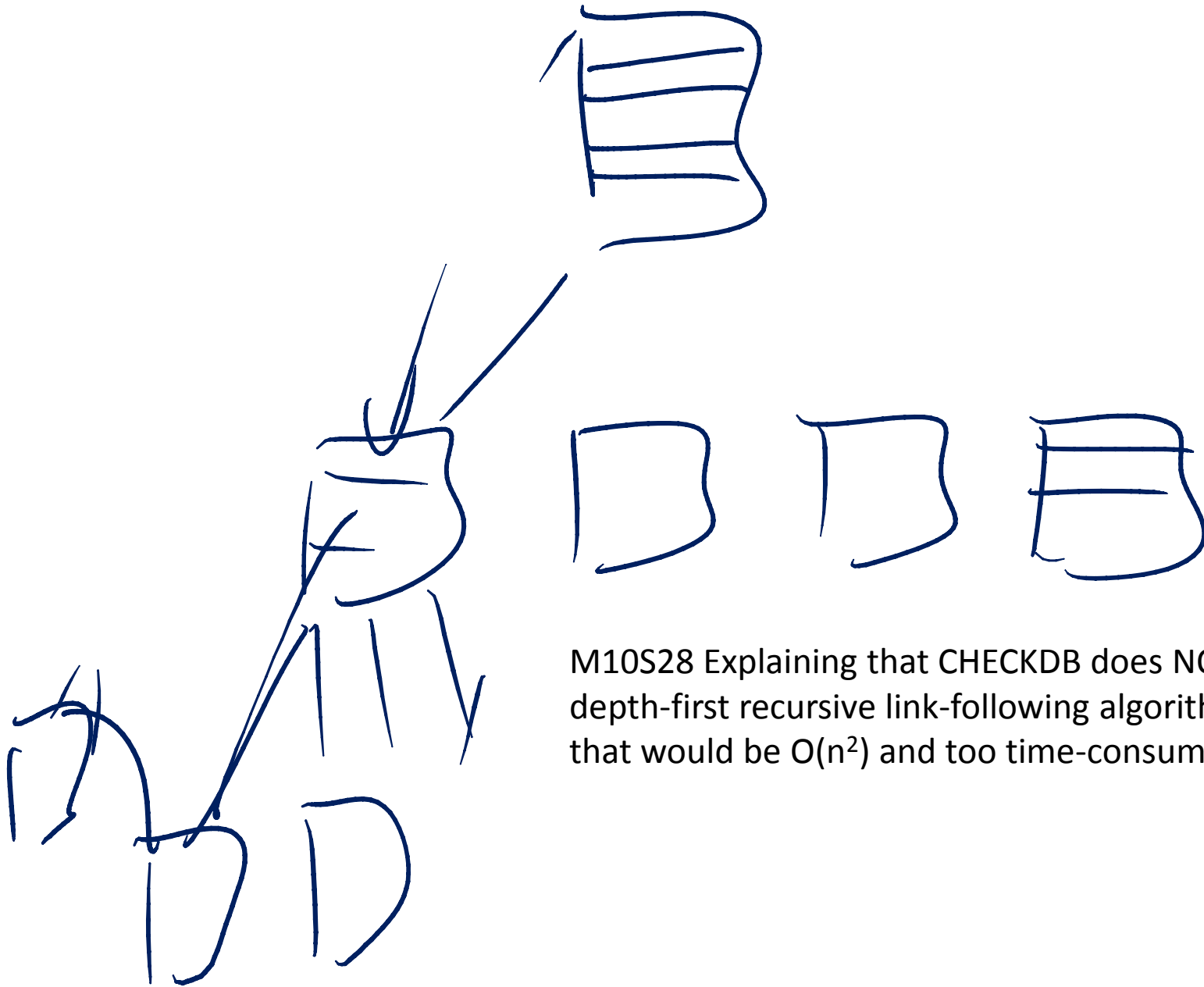




M10S10 Explaining that mirroring and AGs are shipping log blocks, so random I/O subsystem corruption will not be propagated.

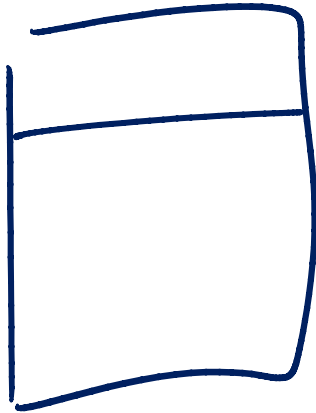


M10S21 Explaining that for automatic page repair, the mirror cannot give the principal the requested page until the mirror's redo queue has passed the LSN at which the principal discovered the broken page.

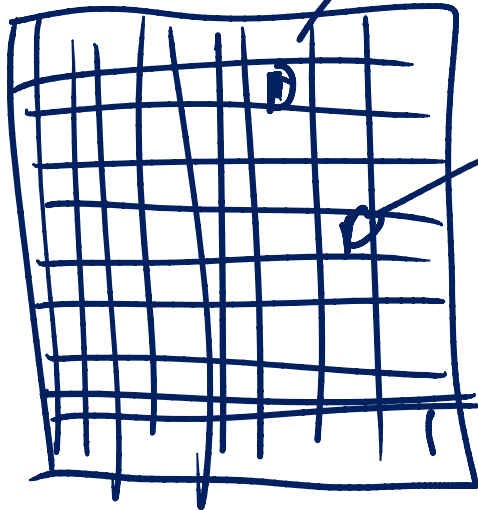


M10S28 Explaining that CHECKDB does NOT do a depth-first recursive link-following algorithm, as that would be $O(n^2)$ and too time-consuming.

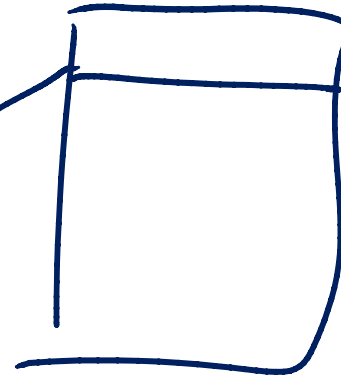
DATA c_1, c_2, c_3



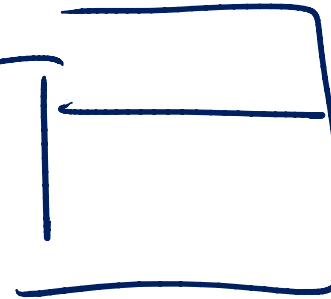
2133\$1



NC 1 c_2



NC 2 c_3



See next slide...

M10 Explaining how the nonclustered index checking algorithm works.

For 2005-2008R2:

Each data record is used to construct all NC index records, then each is hashed to a value and a corresponding bit is flipped in a bitmap. Each actual NC index records is hashed and a corresponding bit is flipped in a bitmap. At the end of the checks, the bitmap should be full of zeroes. Lossy algorithm, so any inconsistencies mean a deep-dive to figure out which records are incorrect by rehashing the records until one is found that matches the erroneous bit, and then the actual records are looked up.

Post 2008R2:

Each data record is used to construct all NC index records, then each is hashed to a value and the value is folded into a checksum for the nonclustered index. Each actual NC index records is hashed and folded into a second checksum for the nonclustered index. At the end of the checks, for each nonclustered index, the checksum created from the data records should match the checksum created from the index records. Lossy algorithm, so any inconsistencies mean a deep-dive to figure out which records are incorrect by looking up all the records for the nonclustered index with the bad checksum.