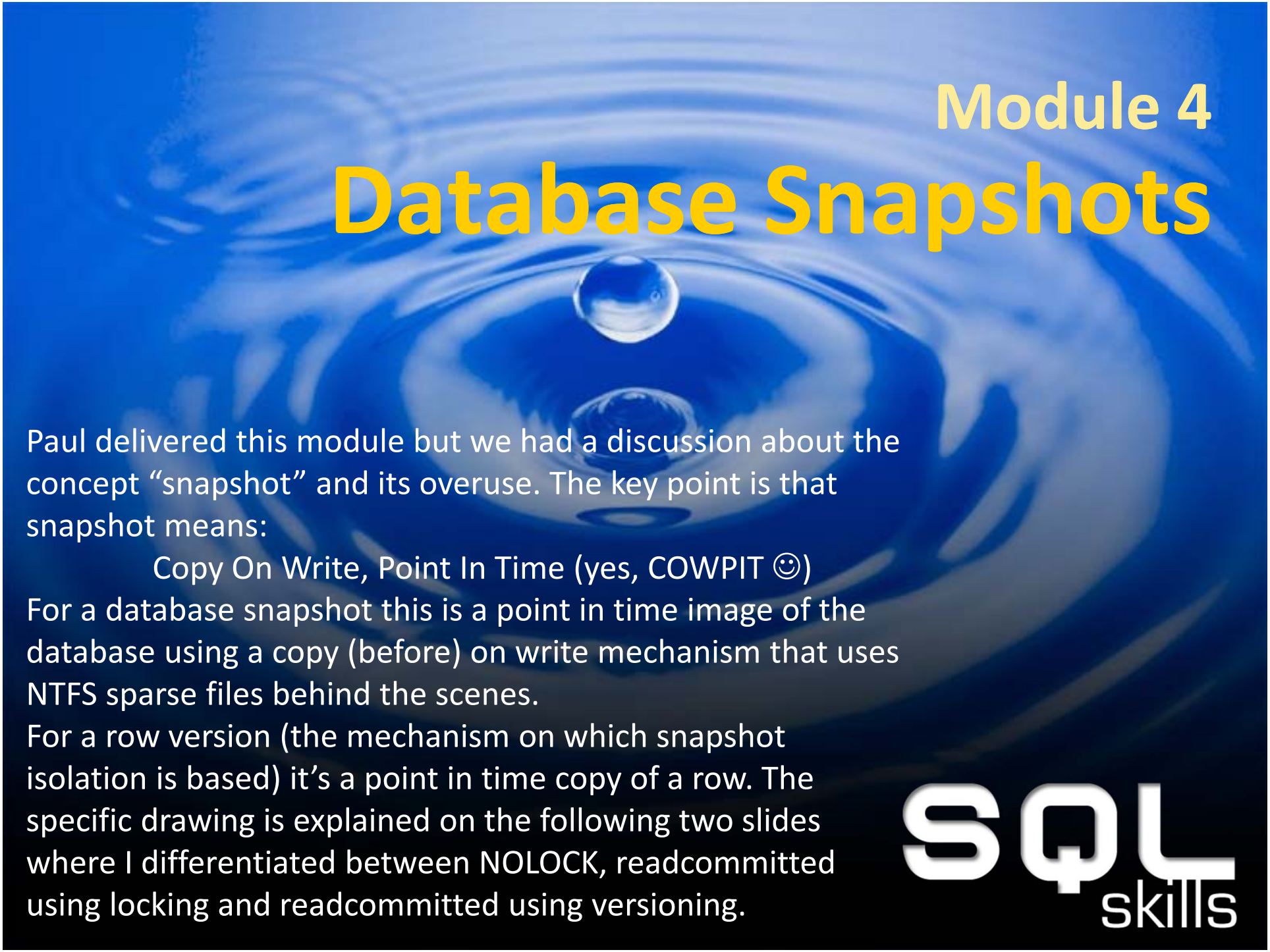


Module 4
Database Snapshots

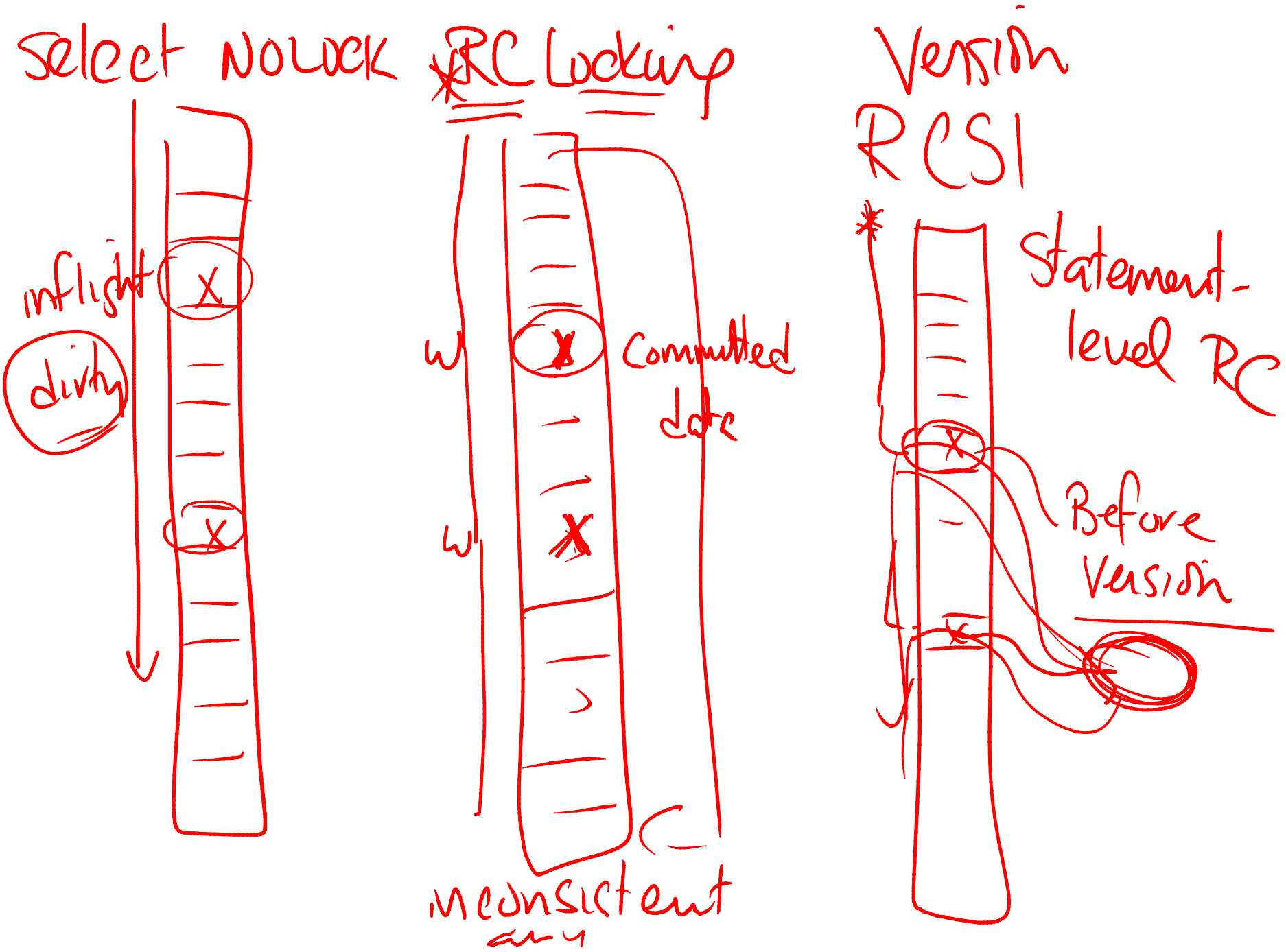
Paul delivered this module but we had a discussion about the concept “snapshot” and its overuse. The key point is that snapshot means:

Copy On Write, Point In Time (yes, COWPIT ☺)

For a database snapshot this is a point in time image of the database using a copy (before) on write mechanism that uses NTFS sparse files behind the scenes.

For a row version (the mechanism on which snapshot isolation is based) it’s a point in time copy of a row. The specific drawing is explained on the following two slides where I differentiated between NOLOCK, readcommitted using locking and readcommitted using versioning.

SQL skills


Select NOLOCK
RC Locking
Version
RCSI
inflight
dirty
W
W
Committed
data
Statement-
level RC
Before
Version
Inconsistent
analysis

The prior slide showed a table vertically as a set of pages. The idea was to compare/contrast the behaviors between:

(1) A forced NOLOCK hint

The key thing to remember here is that nolock allows the reader to read quickly – not stopping for locked rows. However, these rows may be "in-flight" and their modified data might end up getting rolled back or even be in a mid-flight state. If you're looking for only an estimate – this might be fine.

(2) The default – READ COMMITTED (using locking)

The key things to remember from the picture is that (1) has "hiccups" along the way as they encounter rows that are locked. They read… wait… read… wait. This is *partially* what slows down a statement. However, it's worse that this. It's possible that AFTER a row is read, it will appear again (if the record is relocated) and we will read it twice (non-repeatable reads). Also, it's possible that another transaction would modify a row that we've NOT yet seen (before we get there) as well as a row that we have seen (after we've read it) such that the end result of our statement has rows that aren't really transactionally consistent (with another multi-statement transaction). This is also a problem… The default (1) environment is prone to a few inconsistencies (aka. Inconsistent analysis). Only way to solve – increase isolation (or [as of 2005] consider using versioning).

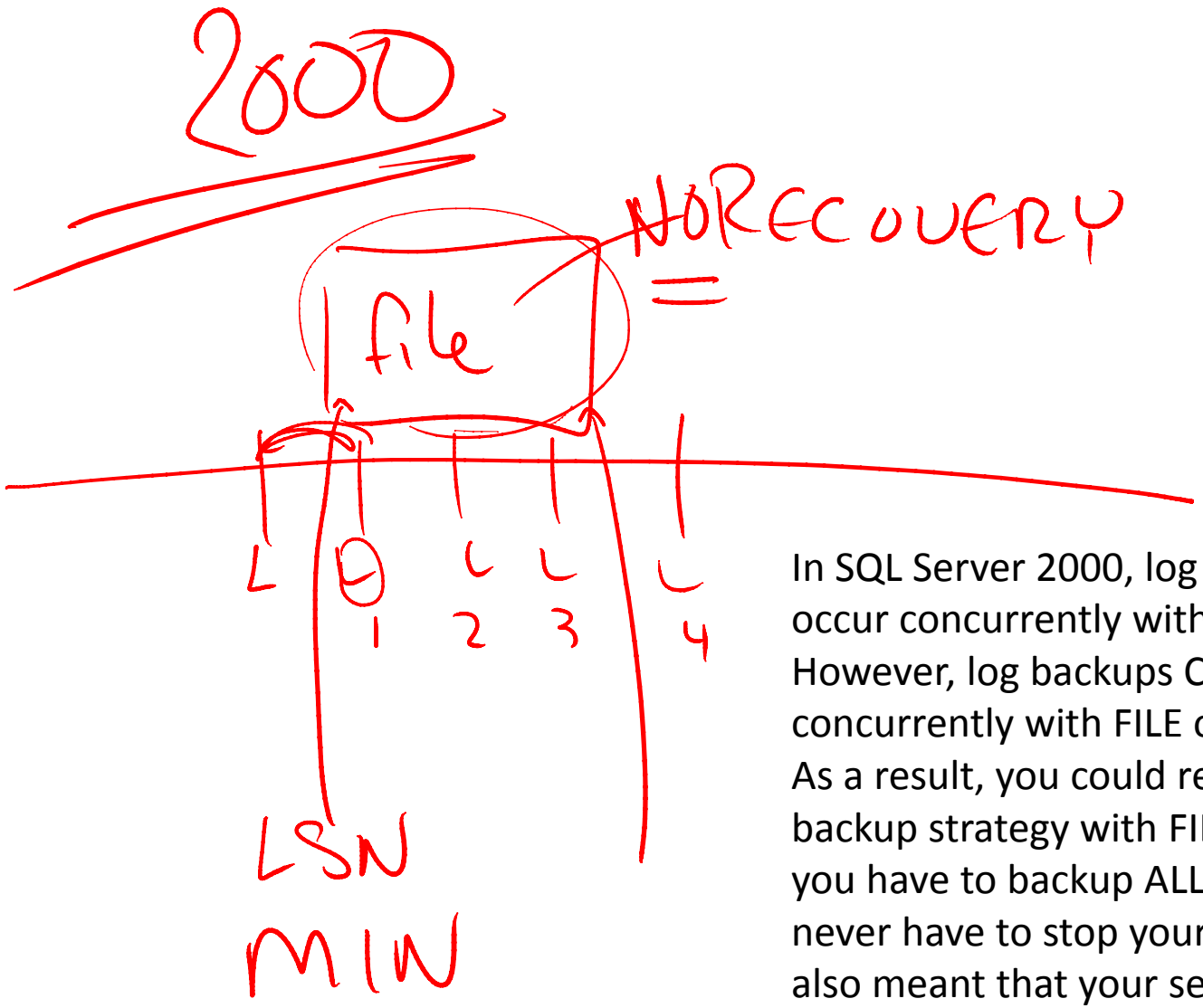
(3) Versioning (and it was implied that it was statement-level RCSI: read_committed_snapshot)

When a versioned query runs the reader will not stop but will have to go to the version store for any row that's in flight. This is a tiny bit slower but guarantees consistency of the ENTIRE read to the point in time when the statement started. This gives you better concurrency as well as a definable point in time to which your statement reconciles. Of course, it isn't free – the overhead of versioning is for every writer and it occurs within tempdb.

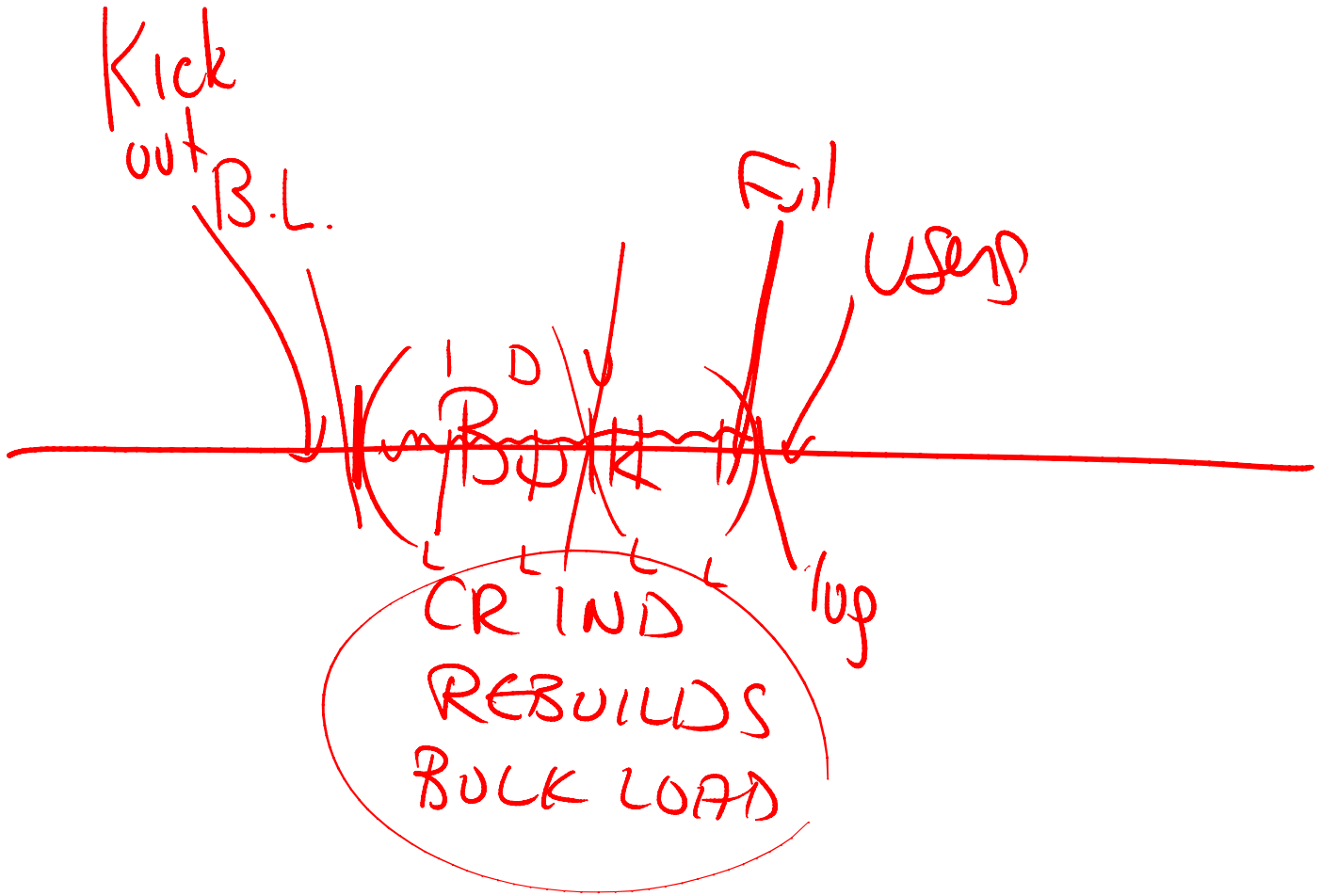

Module 5
Backup Strategy and Internals
SQL skills

Here we were discussing how log backups can occur concurrently with full database backups. They can even run numerous times during a single full. Each log backup will run but it will NOT clear the log. When the backup completes the data copy phase, the necessary log information will be backed up as well. This will allow the no-longer-needed log information to be cleared but it is not the full backup that caused the log to clear – it's the log backups that have already been done!

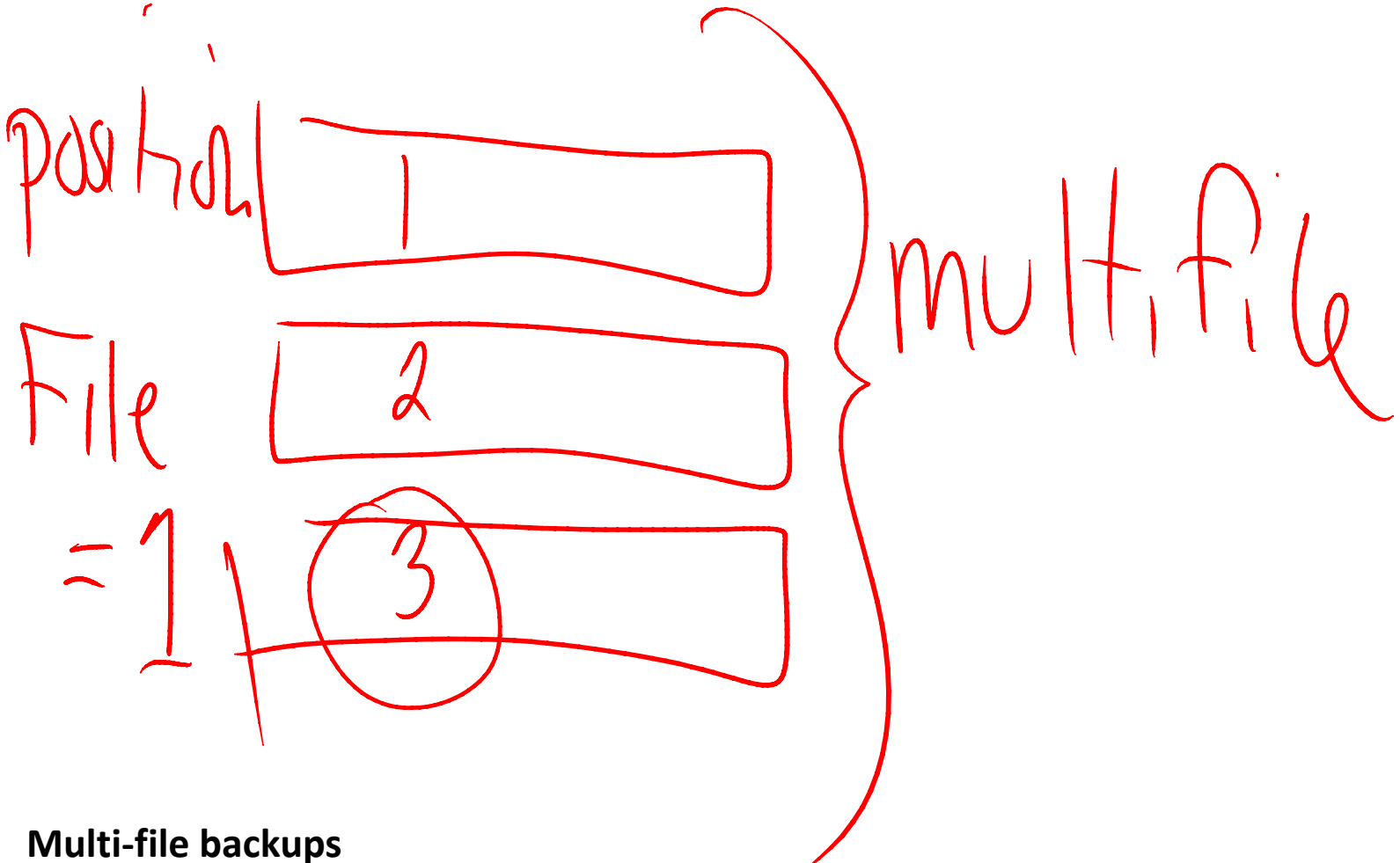
Also, if a full backup becomes damaged, SQL Server can use the logs as if the full backup had never occurred. The restore process would be:
Use the most-recent (but working ☺) full database backup
Use the last differential based on that full backup
Use any logs from the last differential and until current (up-to-the-minute)



In SQL Server 2000, log backups could NOT occur concurrently with full database backups. However, log backups COULD occur concurrently with FILE or FILEGROUP backups. As a result, you could replace your full database backup strategy with FILEGROUP backups (and you have to backup ALL files/filegroups) and never have to stop your log backup job. This also meant that your secondary sites would NOT fall far behind (because log backups would NOT be blocked during FILE/FILEGROUP backups).

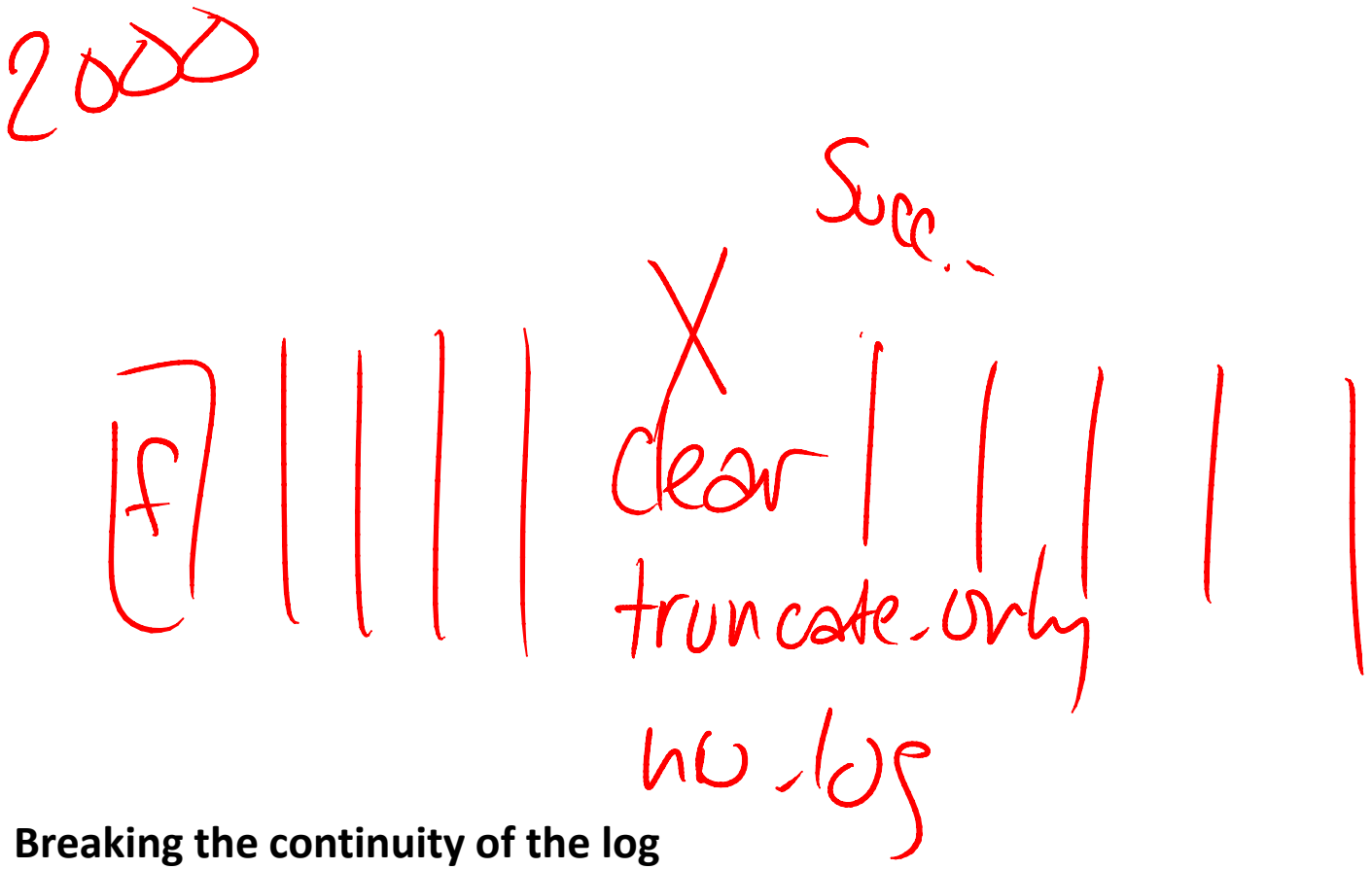


If you plan to use the BULK_LOGGED recovery model during certain batch/bulk operations, you should be sure to minimize the potential data loss but performing log backups before the change TO bulk_logged as well as AFTER the change back to the FULL recovery model. Remember, if a disaster were to occur while SQL Server is in BULK_LOGGED you cannot access the tail of the log (if any bulk operations have occurred). This is what you would lose if there were a disaster during the batch process.



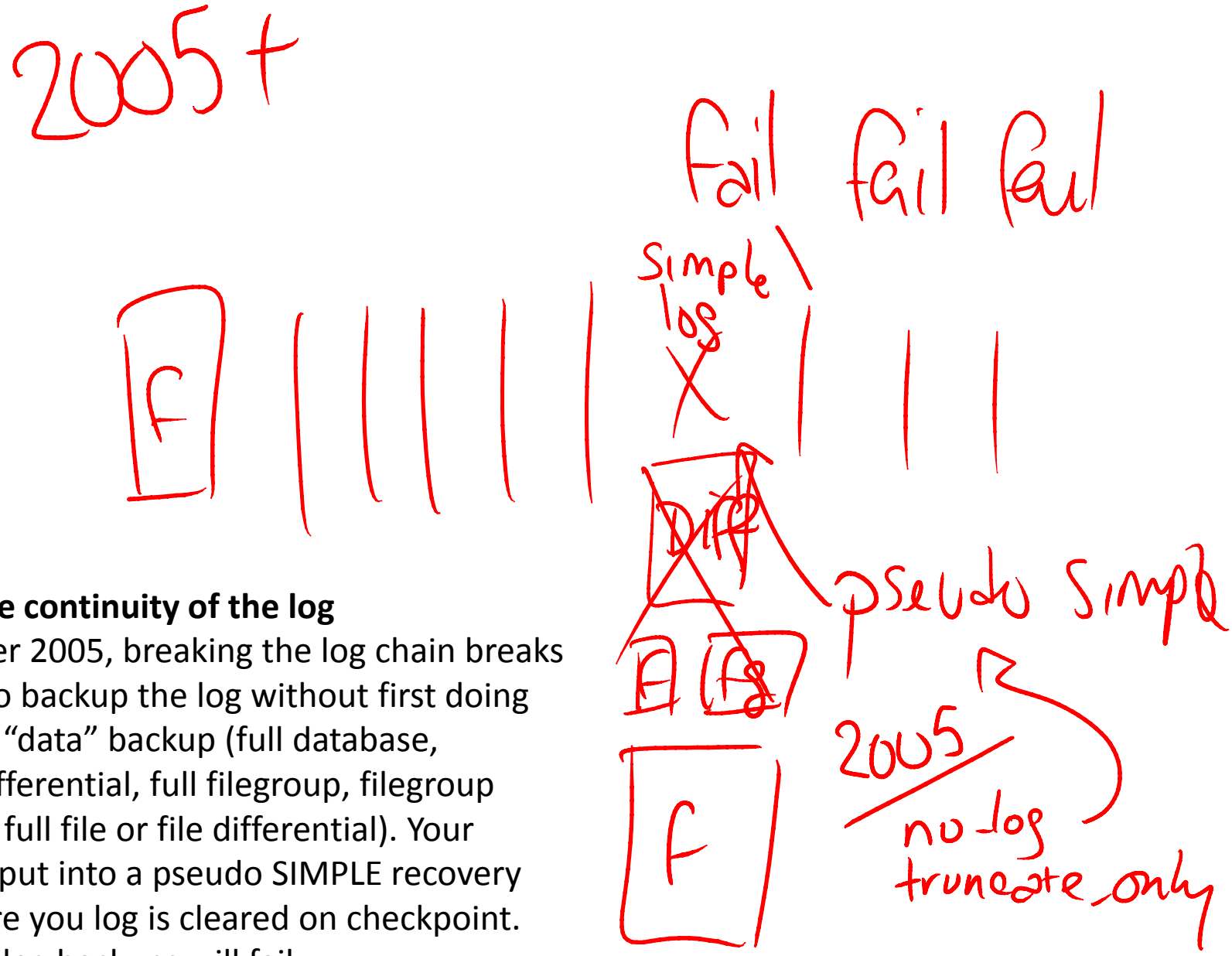
Multi-file backups

Are multiple backups to one or more devices (know as a media set). Once a media set has been defined, all backups to that media set must match in number as well as compression setting. If you want to change compression or use a different number of backup devices – you must use the WITH FORMAT option. When specified the media set is overwritten and any remaining “fragments” are completely and totally unusable.



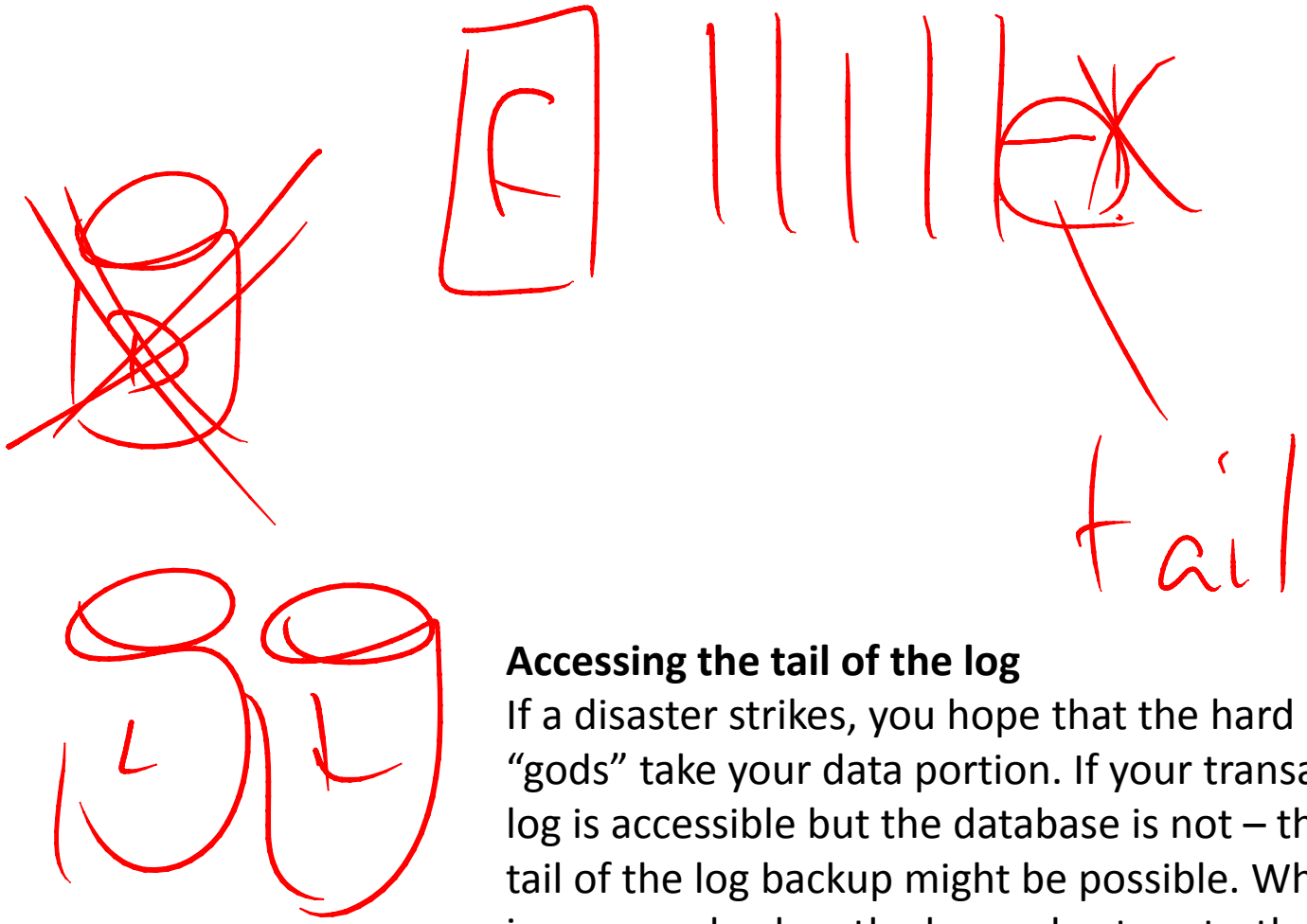
Breaking the continuity of the log

In SQL Server 2000, you can continue to successfully perform log backups even after the continuity of the log has been broken (YIKES!). To stop this possibility of this happening when someone uses BACKUP LOG WITH TRUNCATE_ONLY or NO_LOG, use trace flag 3231 (see the slides for more details). These commands are turned into a NOOP (null operation) in SQL Server 2000 and 2005 with 3231. On SQL Server 2005, 3230 turns them into a CHECKPOINT (which has no [negative] effect on databases in the FULL or BULK_LOGGED recovery models).



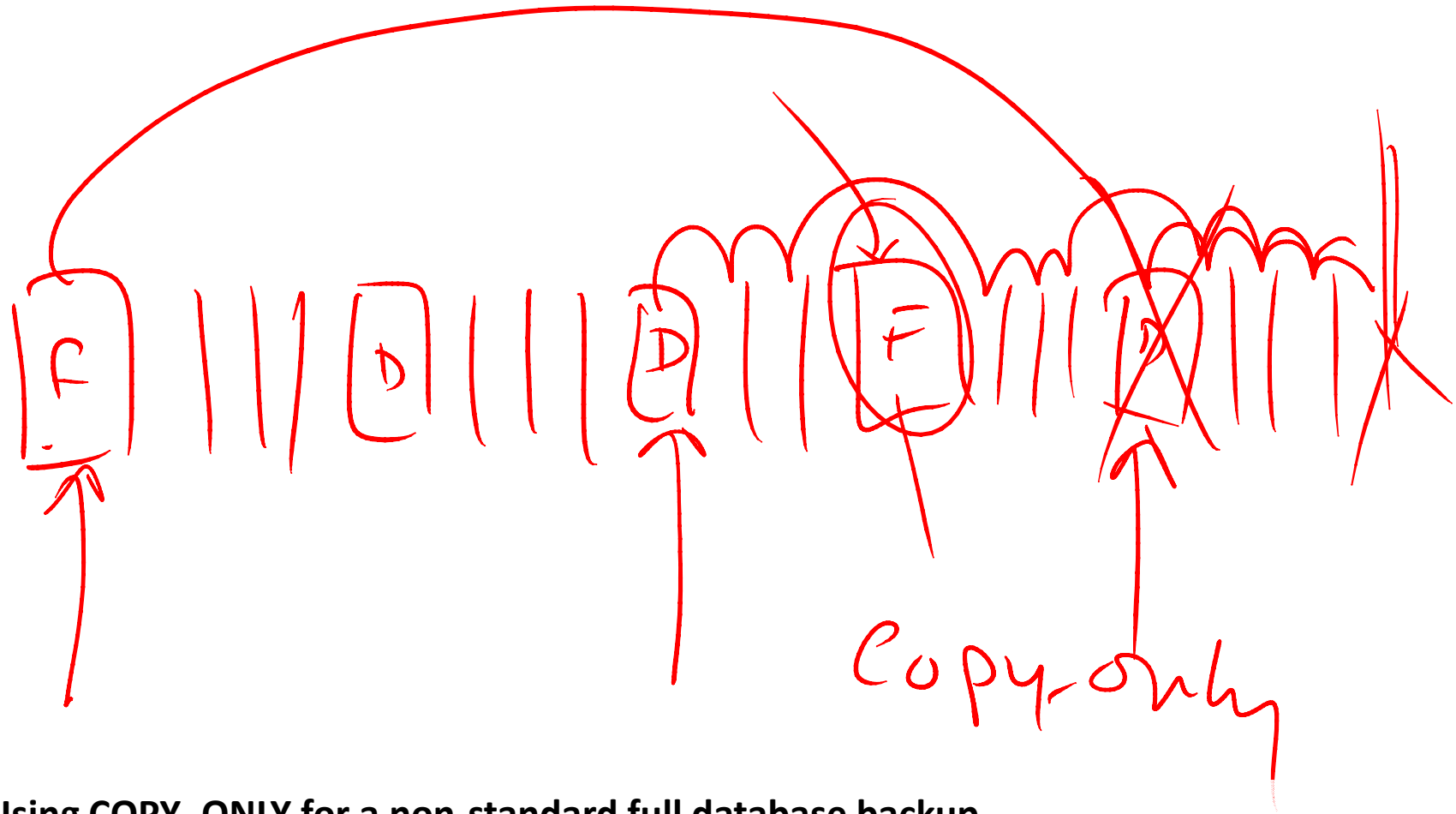
Breaking the continuity of the log

In SQL Server 2005, breaking the log chain breaks the ability to backup the log without first doing some other “data” backup (full database, database differential, full filegroup, filegroup differential, full file or file differential). Your database is put into a pseudo SIMPLE recovery model where you log is cleared on checkpoint. Transaction log backups will fail.



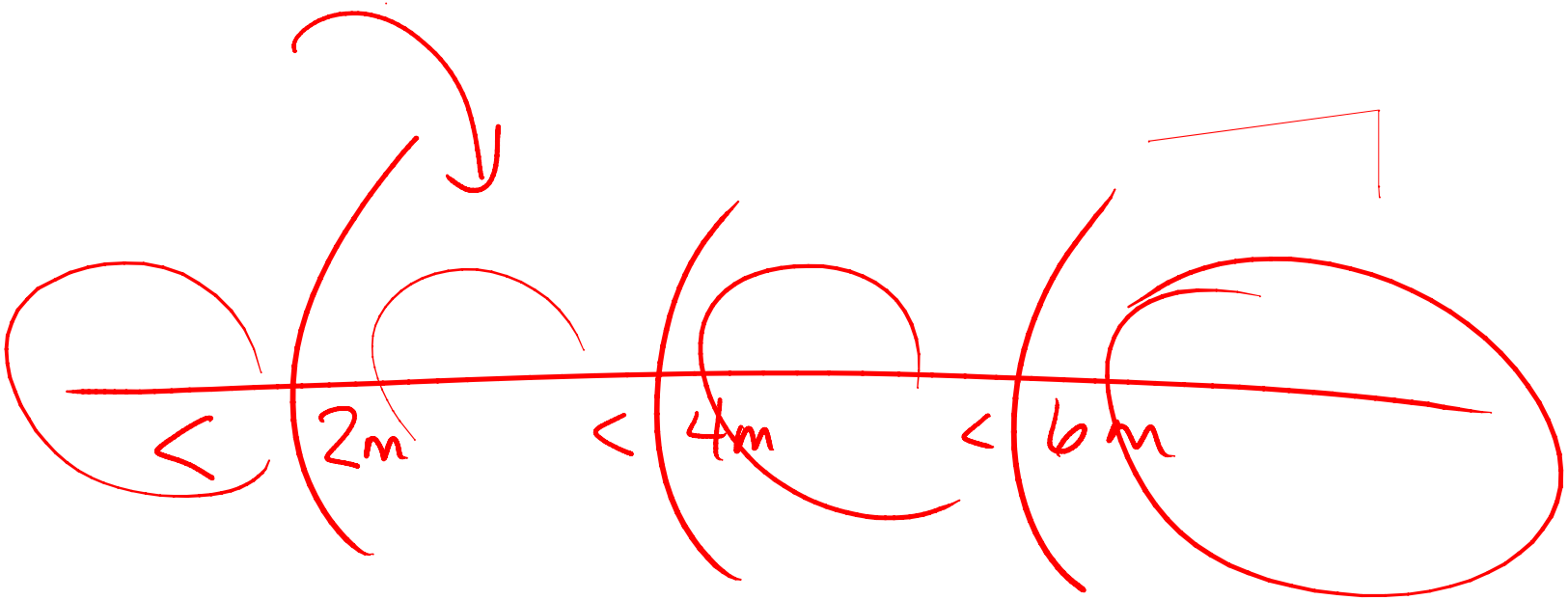
Accessing the tail of the log

If a disaster strikes, you hope that the hard drive "gods" take your data portion. If your transaction log is accessible but the database is not – then a tail of the log backup might be possible. When it is, you can backup the log and get up-to-the-minute recovery of your database (with ZERO data loss). This is why redundancy for the log is the most important – even more so than for the data portion. However, you need redundancy there as well to minimize downtime.



Using COPY_ONLY for a non-standard full database backup

If a developer/tester or another team needs a backup of a database, you do not just want to back it up and give it to them. Performing a full database backup resets the differential bitmap. All differentials would then be tied to that most recent backup and not the one that was performed during your regularly scheduled backups. As a result, your restore process might take quite a bit longer (and require quite a few more log backups to be applied). Use COPY_ONLY instead and the diff bitmap will not get reset!



Online operations – SalesDB partitioning

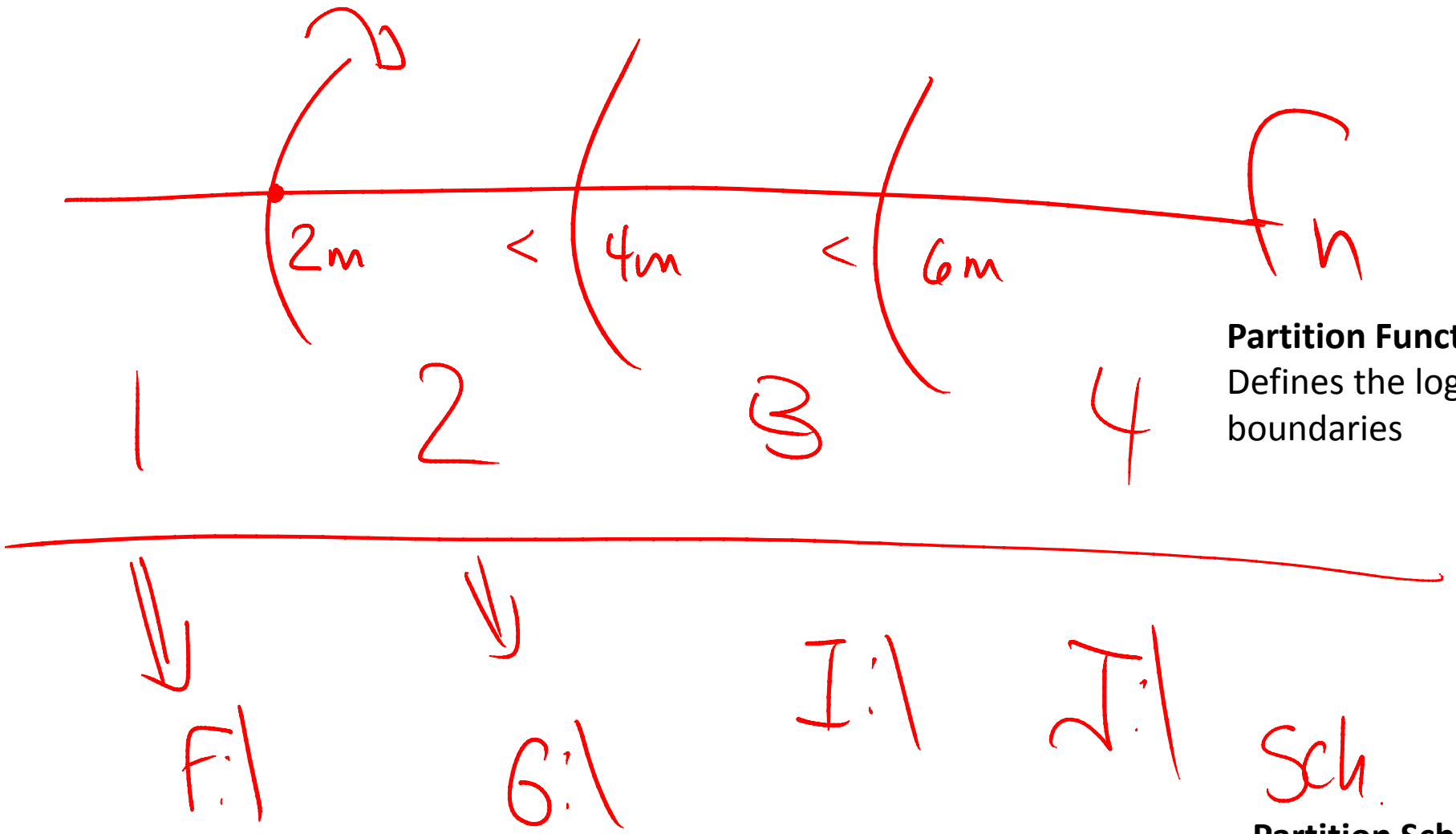
The sales table was partitioned with 3 boundary points: 2 million, 4 million and 6 million. It's a RIGHT-based partition function so the rows that match the boundary point will be to the RIGHT side. Specifically this translates into:

In partition 1: all rows less than 2 million

In partition 2: all rows equal to or greater than 2 million and less than 4 million

In partition 3: all rows equal to or greater than 4 million and less than 6 million

In partition 4: all rows equal to or greater than 6 million



Partition Function
Defines the logical boundaries

Partition Scheme
Defines the physical location for the data

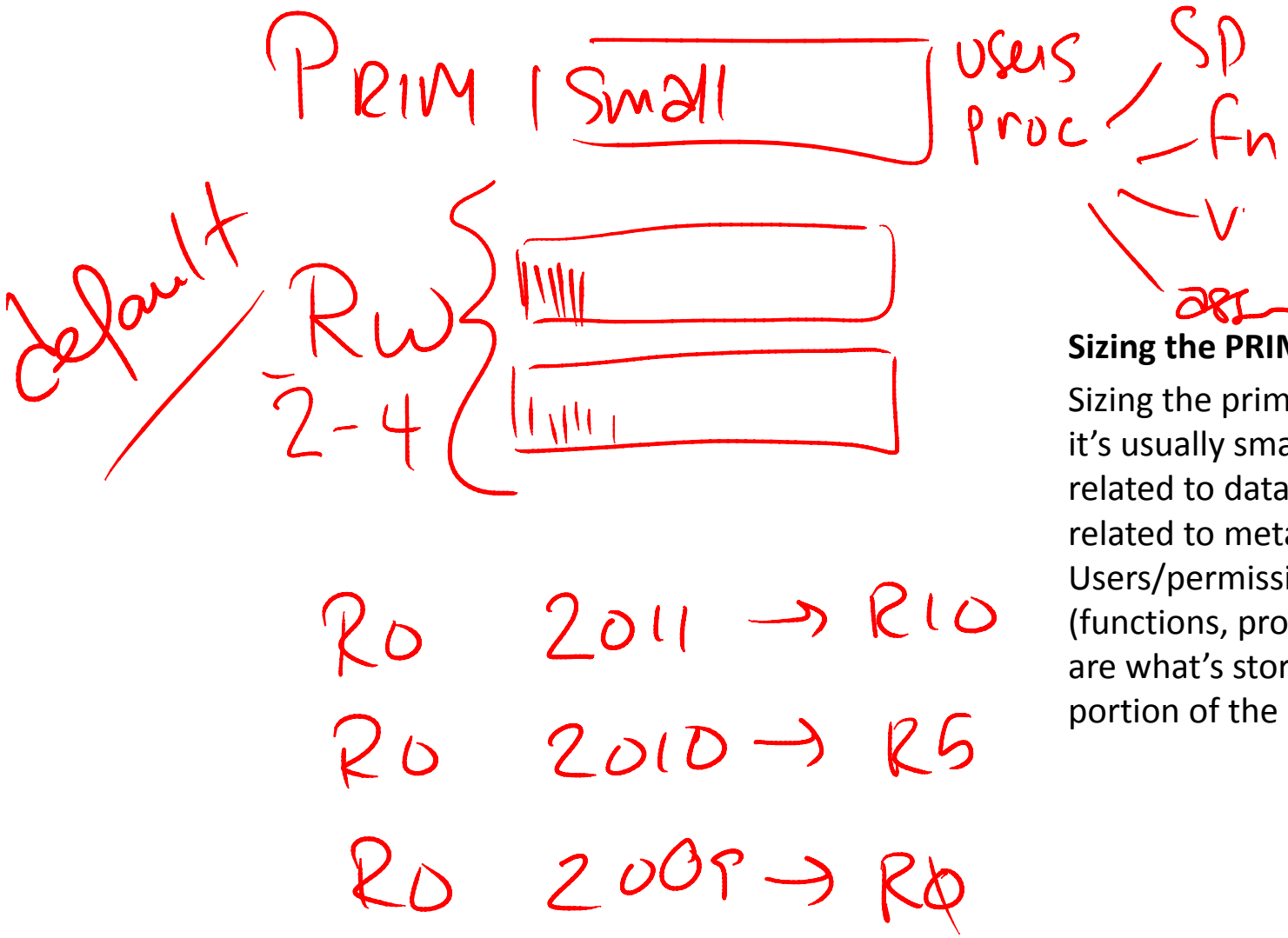
Online operations – SalesDB partitioning

When I partitioned the sales data, I ended up writing the data to the USB drives.

- Drive F:\ (partition 1): all rows less than 2 million
- Drive F:\ (partition 1): all rows equal to or greater than 2 million and less than 4 million
- Drive F:\ (partition 1): all rows equal to or greater than 4 million and less than 6 million
- Drive F:\ (partition 1): all rows equal to or greater than 6 million

Primary filegroup, Primary file?

The first file associated with a database is the MDF (main data file) which is also know as the PRIMARY DATA FILE (but Adobe beat us to PDF ☺). This file is critical for a database to be up and running. If you add files to a database without specifying a filegroup they are going to end up in the PRIMARY filegroup. Because the primary file is a member and because allocations can come from any file of the filegroup – ALL of the files of the primary filegroup end up being critical. Generally, it's important to isolate the primary filegroup (with just one mdf) and protect it (as much as you would the transaction log).

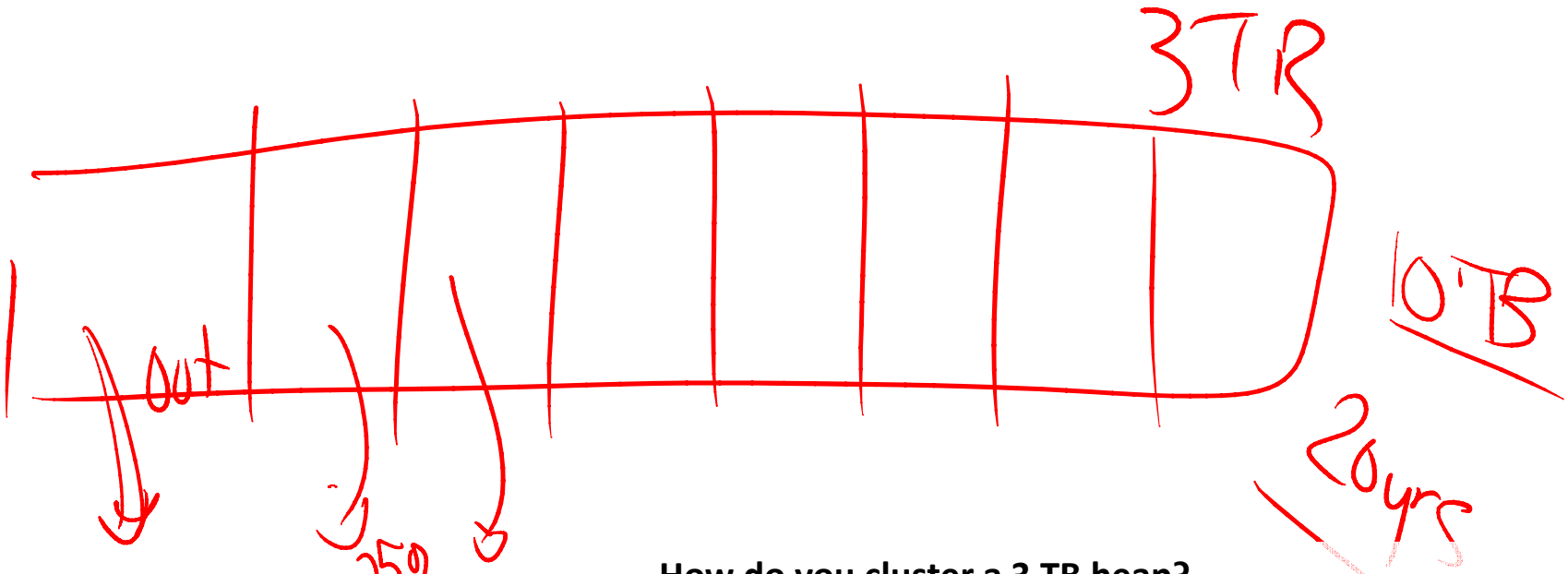


Sizing the PRIMARY filegroup

Sizing the primary is challenging but it's usually small. It's not directly related to data size but instead related to metadata complexity. Users/permissions as well as code (functions, procedures, assemblies) are what's stored in the "primary" portion of the database.

Isolating the primary filegroup

More specifically, isolate primary and separate this from your user-defined data. Create a filegroup for readwrite data that has at least 2 files and then read-only data can be in one or more filegroups (depending on availability requirements, backup/restore SLAs and hardware options/costs)



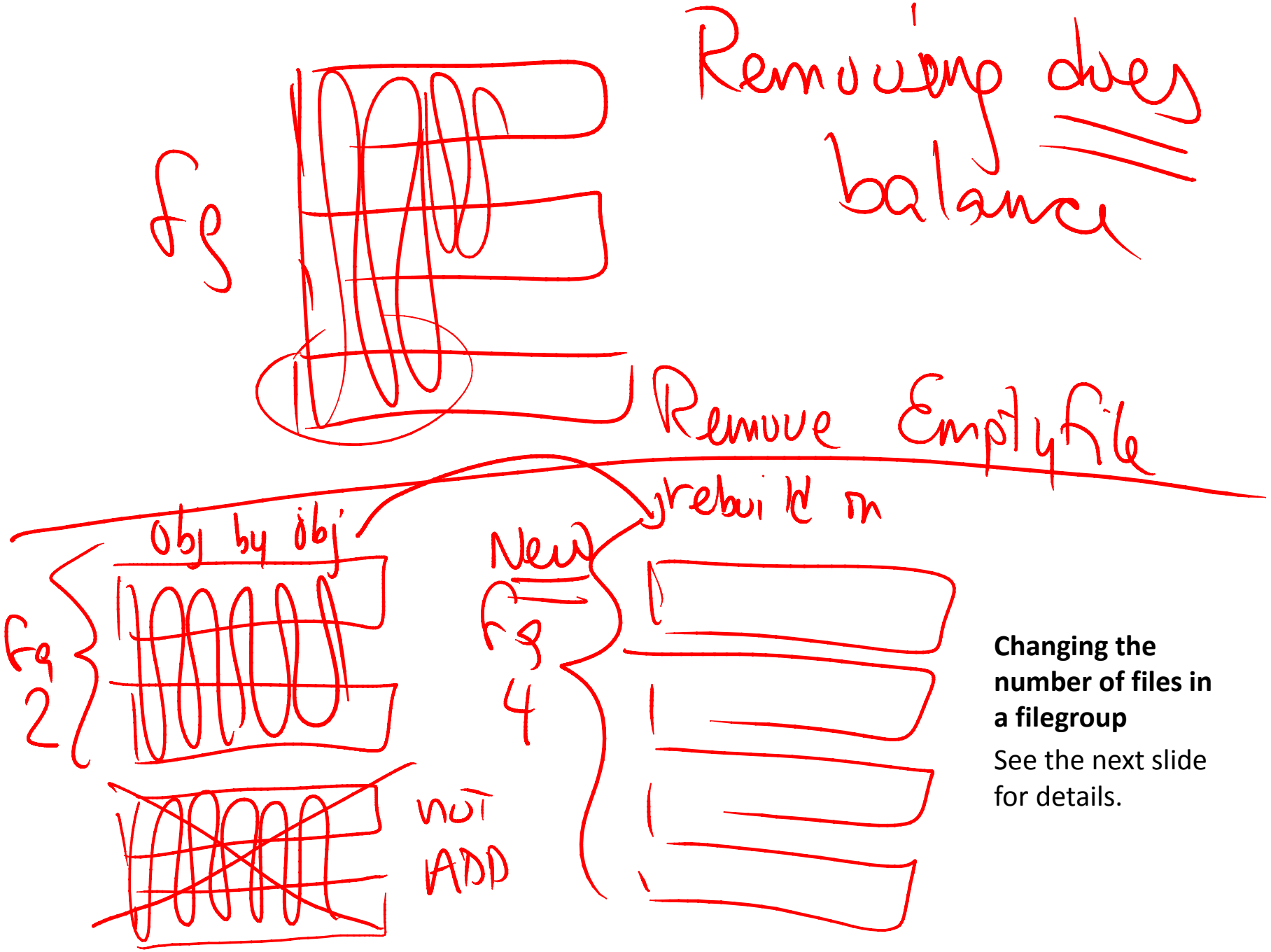
How do you cluster a 3 TB heap?

Not easily. The good news is that it was partitioned. The strategy…

1. Create a new partition function
2. Create a new partition scheme
3. Create a new partitioned table on the new scheme and cluster it
4. Switch out data partition by partition creating the correct/new indexes required by the new table
5. Switch in the partitions until the entire table is moved over

Concerns:

* primary/foreign keys
* downtime



Changing the number of files in a filegroup

See the next slide for details.

Changing the number of files in a filegroup

Removing a file must be done through EMPTYFILE (which DOES rebalance the data among the remaining files in the filegroup). However, it is not object specific.

Adding a file to a filegroup does NOT rebalance. If you want to change a filegroup from two or three files to something like four or five, your best bet is to create a NEW filegroup and then rebuild all of your indexes into it.

Use CREATE with DROP_EXISTING to MOVE an index to another filegroup.

Now, as for moving LOB data… this is the most interesting and it DOES work if you use DROP_EXISTING and rebuild on a scheme or conversely from a scheme to just a single filegroup. However, going from one filegroup to another (even when using DROP_EXISTING) does NOT move the LOB data. There's a blog post in that information for sure. I might try and do this as a SQLServerPro post in my series on Partitioning (Partitioned Tables vs. Partitioned Views).

Here are the links to the current posts:

Q1: Partitioned Tables v. Partitioned Views–Why are they even still around?

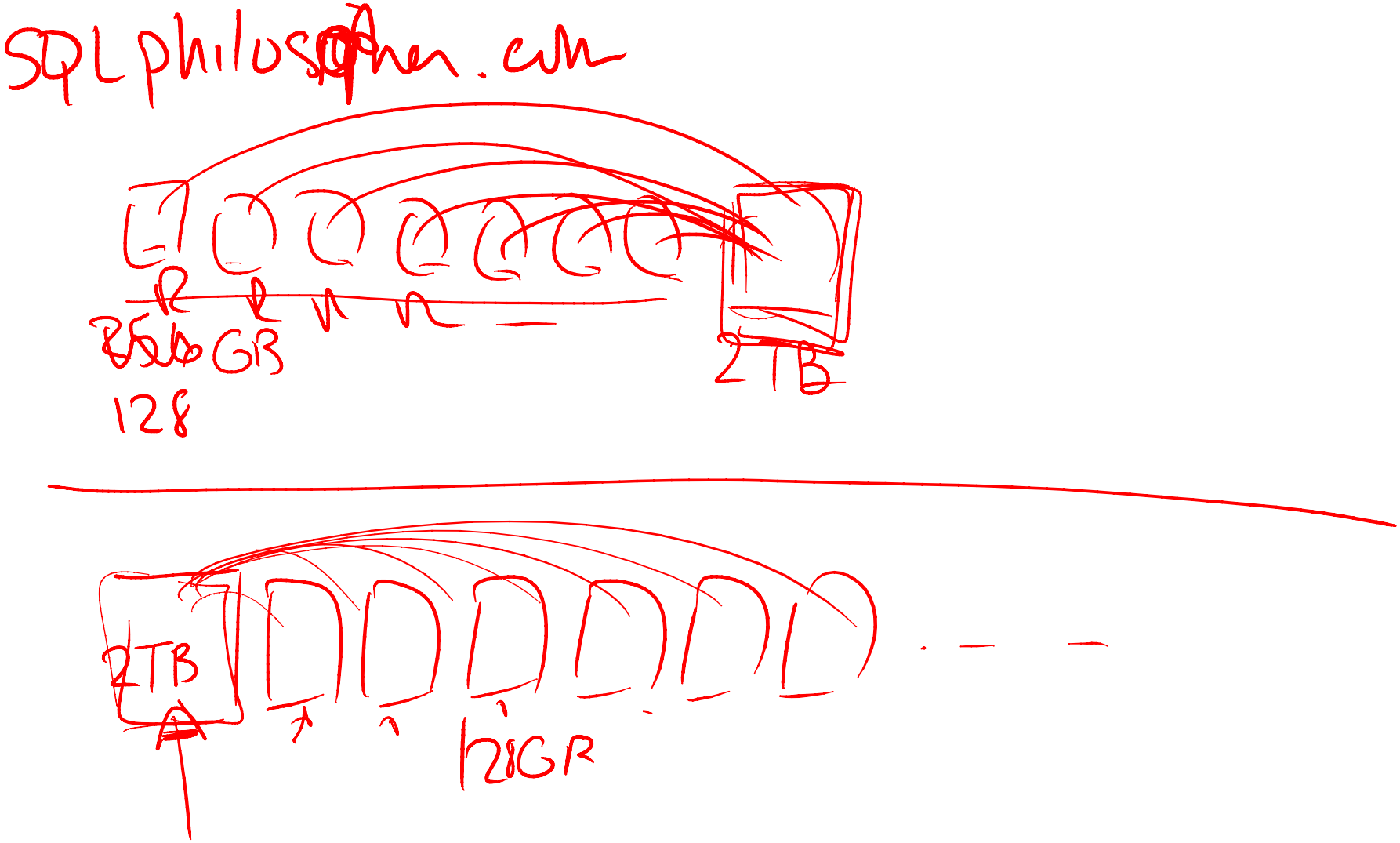
Q2: Solutions to VLT concerns around statistics and maintenance!

Q3: Did SQL Server eliminate any partitions?

Q4: How about Filtered Indexes instead of Partitioning?

Q5: Is there a way to move LOB data? (or something like that)

The final consideration is availability. See the next slide for Brad's (@SQLPhilosopher's) trick!



As an alternative, Brad (@SQLPhilosopher) came up with a way to move LOB data to a different number of files – and, do this as an ONLINE operation.
He described it fully here: http://www.sqlphilosopher.com/wp/2012/02/moving-a-filegroup-to-a-new-disk-array-with-no-downtime/.
It's not a different filegroup but the end result (restructuring the filegroup) is the same!

COPY_ONLY vs. NO_TRUNCATE

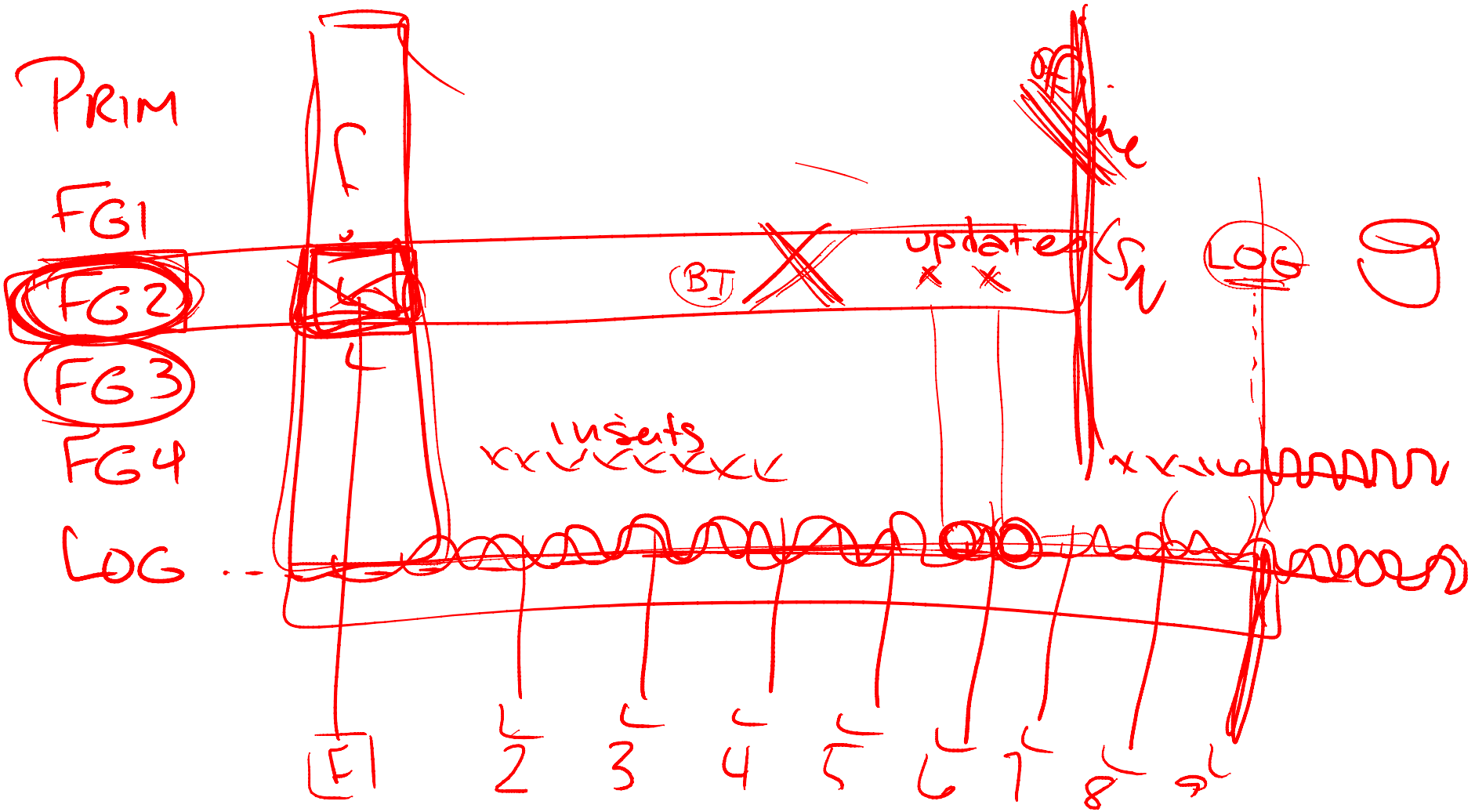
NOTE: This is a new slide that I added to the Backup module and I thought you might want to have the info as a reminder from class! fyi, kt ☺

- Sound similar… but, they're not the same!
- COPY_ONLY will only work if the backup is OK
- NO_TRUNCATE
  - A backup of the log without clearing the log
  - Designed for situations where the data portion is unavailable…
  - Works even if the log backup is damaged but you won't know this because NO_TRUNCATE does a CONTINUE_AFTER_ERROR even if you don't ask

\+ NO_TRUCATE should always work

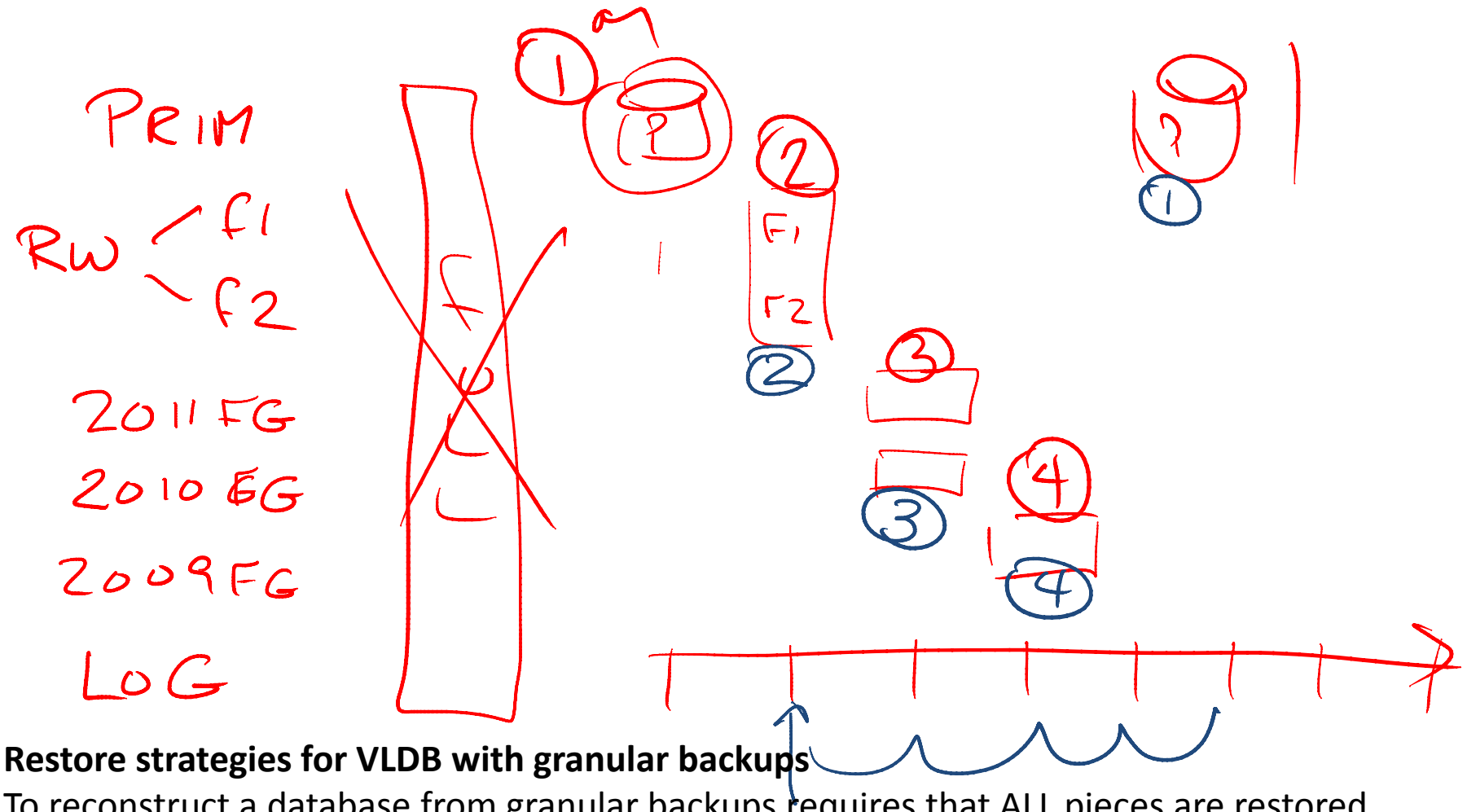
\- You won't really know if there's an error until you restore. You might be forced to use CONTINUE _AFTER_ERROR


Module 6
Restore Internals and Procedures
SQL skills



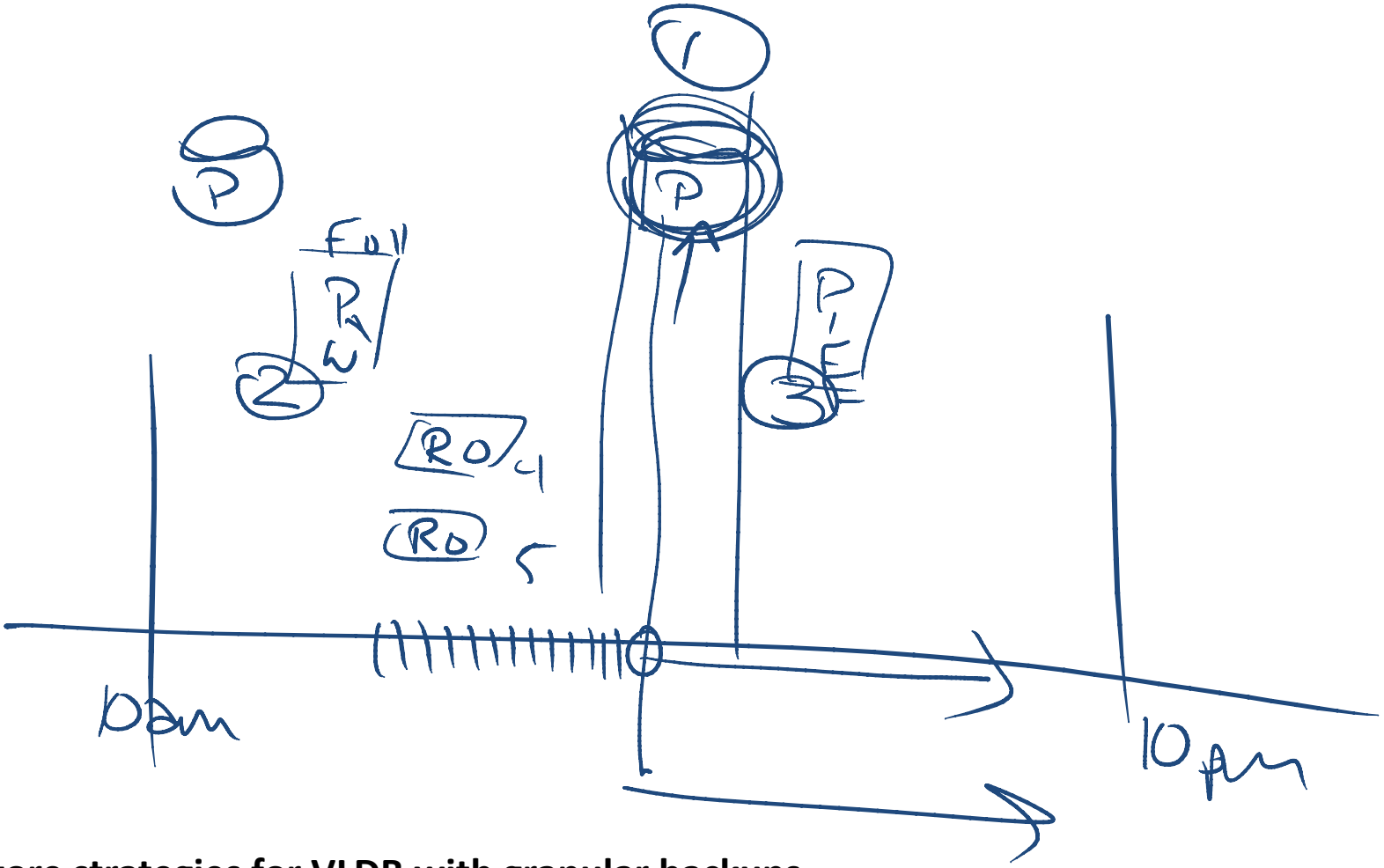
Taking a filegroup OFFLINE

This shows a multi-file database and a series of different types of backups. What was most important here is the understanding of what happens when a filegroup is taken OFFLINE. This is the point in time when SQL Server knows that no knew transactions can occur. This is the point in time to which this filegroup must be restored (reconciled) in order for this filegroup to be brought online.



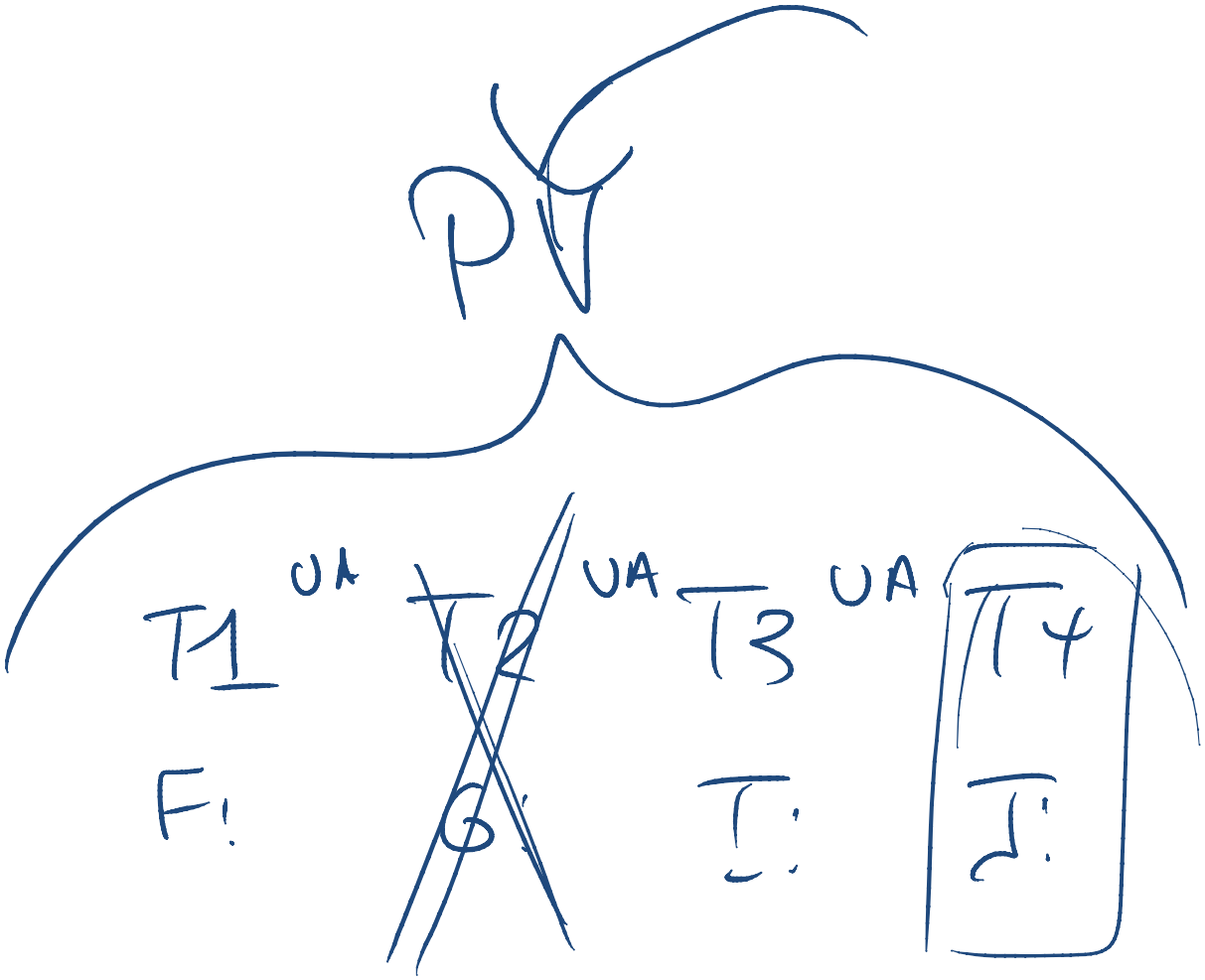
Restore strategies for VLDB with granular backups

To reconstruct a database from granular backups requires that ALL pieces are restored. However, since they were backed up at different times – a transaction log series that starts with the oldest filegroups minimum (or effective) LSN must be used. If you have a current primary and all of the filegroups (labeled in blue as 2, 3, 4) then you only need logs starting with the bottom blue arrow. If the blue backup #1 did not exist and you had to go back to the red #1 then you'd need all of the logs back to that point. The KEY point is that the database will be completely offline until the entire set of filegroups being restored are recovered to the SAME point in time. If you restore PARTIAL then you could bring it online more slowly/granularly.



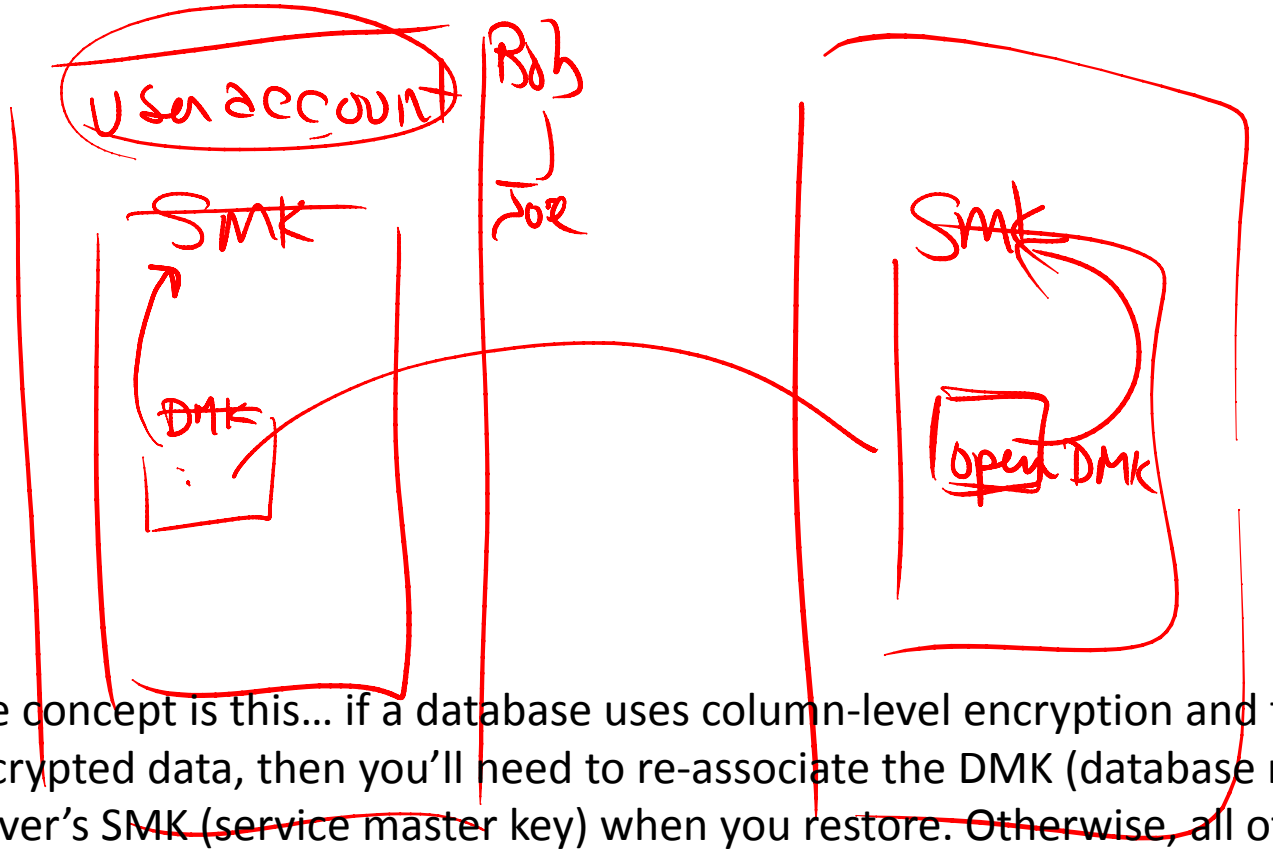
Restore strategies for VLDB with granular backups

This is a continuation of the last picture. If you have a primary filegroup backup (#1) then a RWFilegroup backup (#2) then you could restore that with logs to bring the entire database up to current (restoring the RO filegroups at virtually any time – possibly a lot later if you really just want to get the RW up to current). If you have a differential of the RWFilegroup then you could have restored it and then you only would have needed the logs from the backup time of the Primary filegroup.



Querying data when a filegroup is damaged

If you query a partitioned table when a filegroup is damaged then you'll only receive an error if you (try to) select from the damaged partition. This is also NOW true of a partitioned view. It was not always true (in 2005 a view that was damaged would fail). I *think* this was fixed in 208 but for sure in 2008 R2.



The concept is this… if a database uses column-level encryption and therefore stores encrypted data, then you'll need to re-associate the DMK (database master key) with the new server's SMK (service master key) when you restore. Otherwise, all of the data will be inaccessible!

```
USE database;
go
-- Open the DMK with the SAME password used when created:
OPEN MASTER KEY DECRYPTION BY PASSWORD = 'original password';
go
-- Reassociate the DMK with the SMK of the new instance:
ALTER MASTER KEY ADD ENCRYPTION BY SERVICE MASTER KEY;
go
```