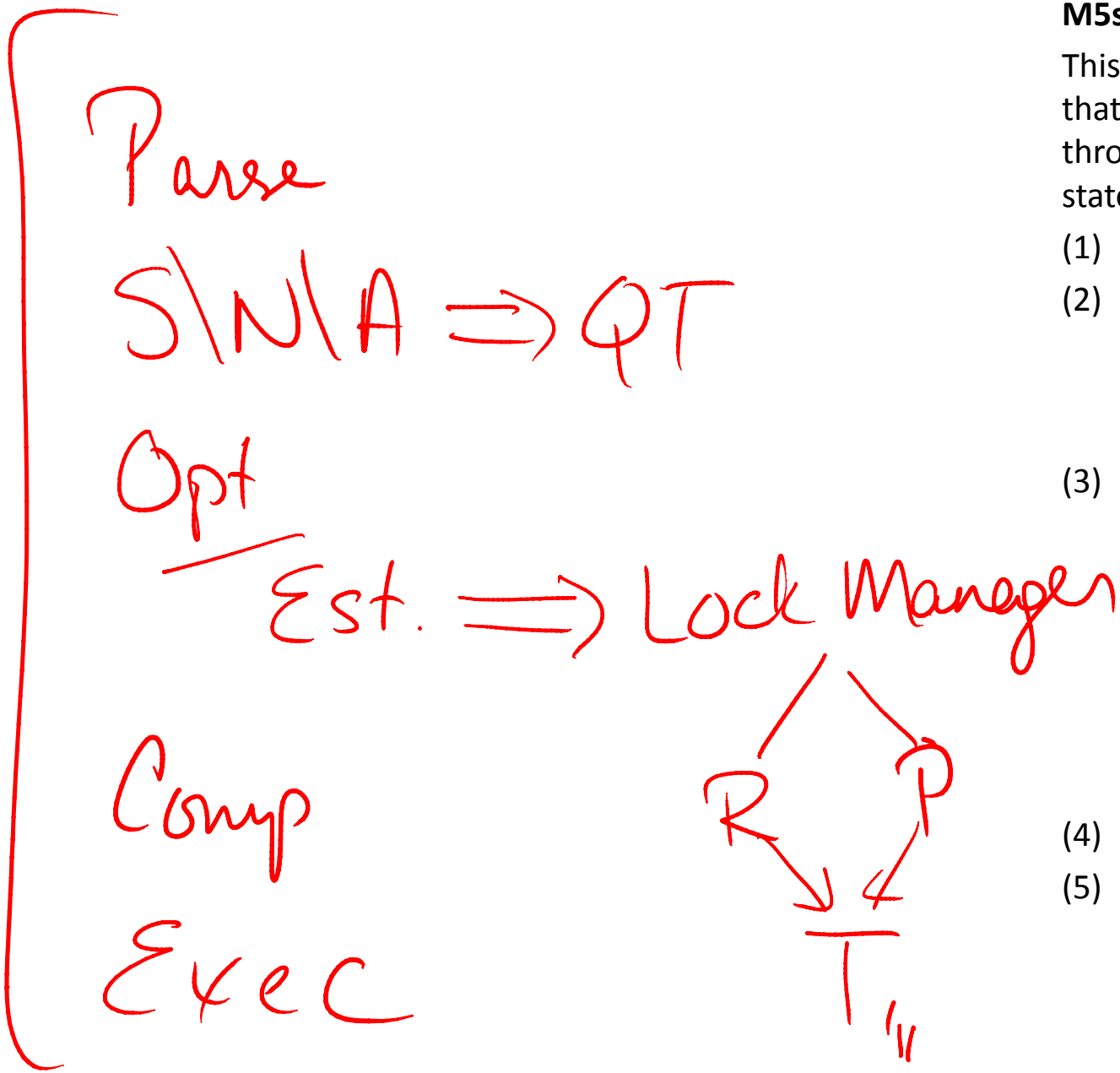


M5s15

This is showing the process that SQL Server goes through when executing a statement:

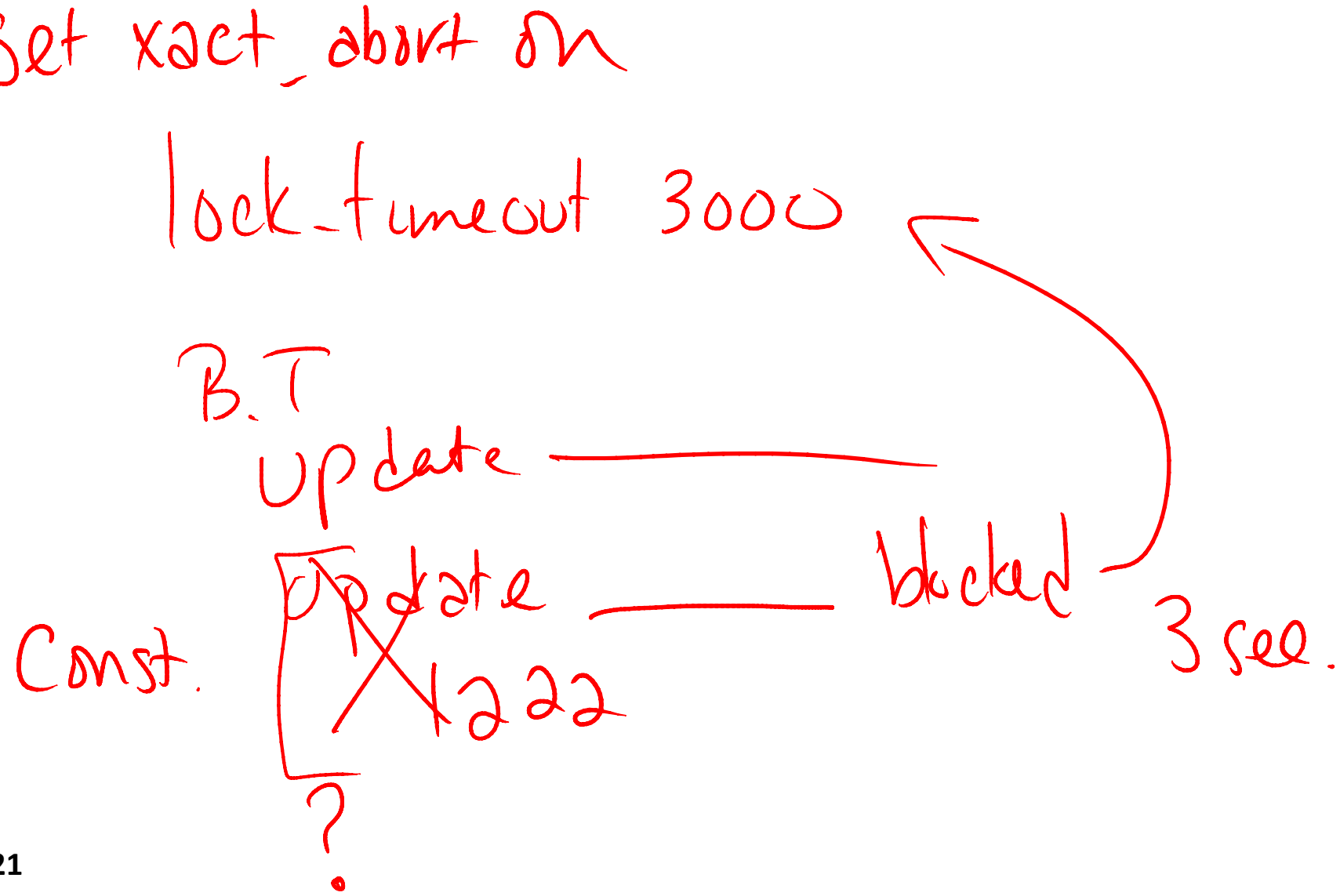
(1) Parse

(2) Standardization / Normalization / Algrebrization
    -> query tree

(3) Optimization (this is where the lock manager uses statistics to estimate/determine the best lock granularity (and best for concurrency / resources)

(4) Compilation

(5) Execution – at runtime, if the resources don't exist to support the lock level then escalation occurs – to TABLE level (by default)



M5s21

It's important to realize that the default behavior (without xact_abort on) when a lock_timeout is hit (error 1222) does NOT cause the transaction to fail – only the statement.
You must use the session setting SET XACT_ABORT ON in order to cause the transaction to fail.

M5s25 – aka Transaction Madness

The key points are:

BEGIN TRANSACTION

- Increments @@trancount

COMMIT TRANSACTION

- Decrements @@trancount

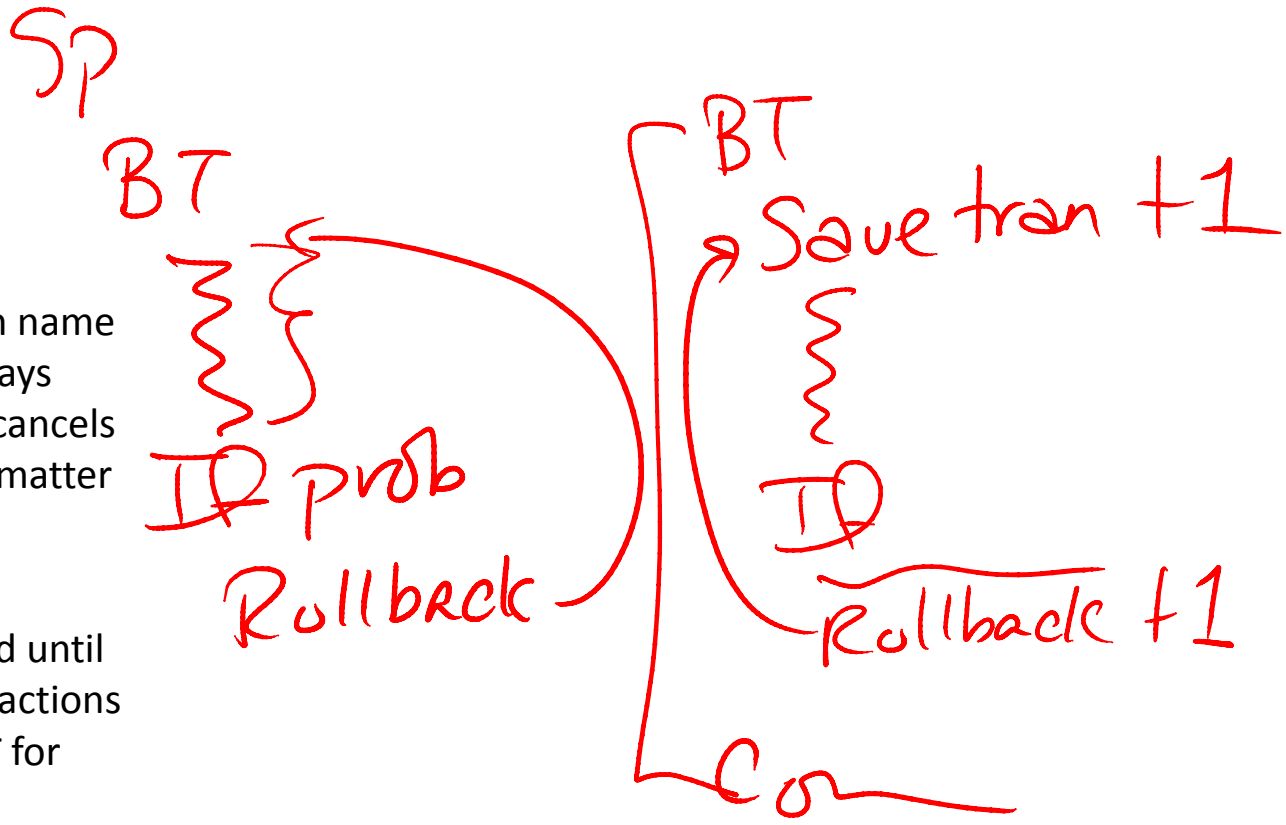
ROLLBACK TRANSACTION

- When used with a transaction name OR without a name at all – always resets @@trancount to 0 and cancels ALL pending transactions – no matter how deeply nested.

A transaction is NOT committed until @@trancount = 0. When transactions are nested you MUST COMMIT for every BEGIN

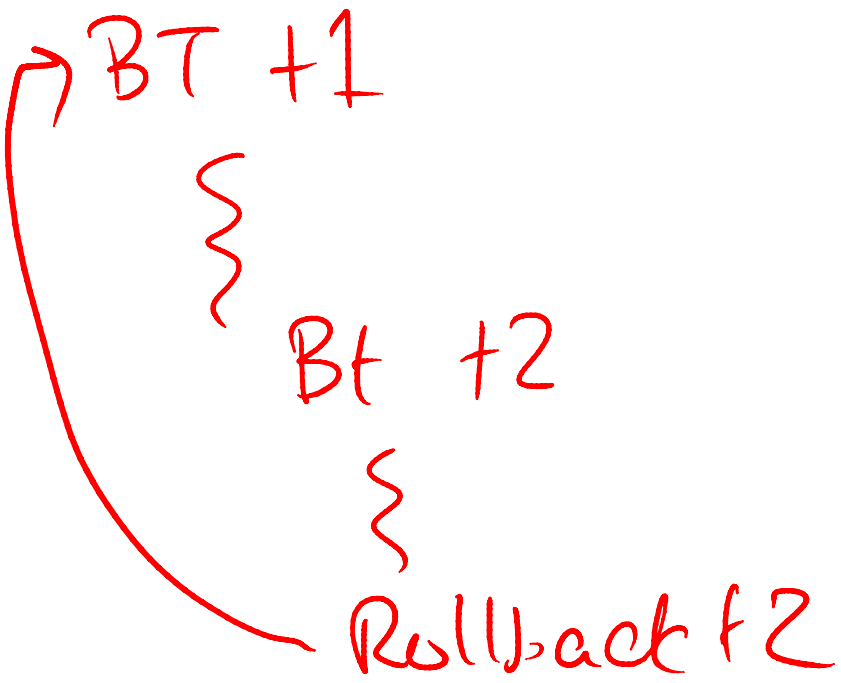
Savepoint (SAVE TRANSACTION NAME) do NOT affect @@trancount and can be rolled back to safely without affecting @@trancount.

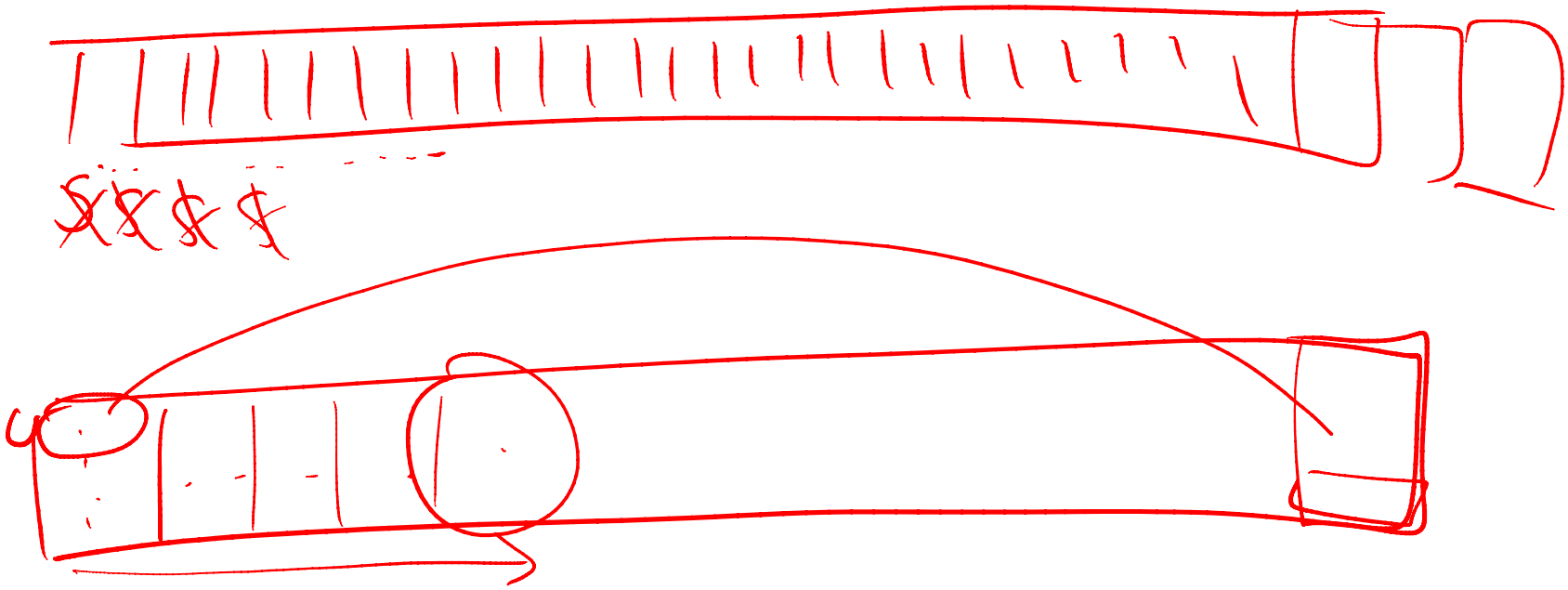
A stored procedure is NOT a transaction in and of itself. You MUST use BEGIN/COMMIT to engage the ACID properties of the statements of a stored procedure.



M5s25 – aka Transaction Madness

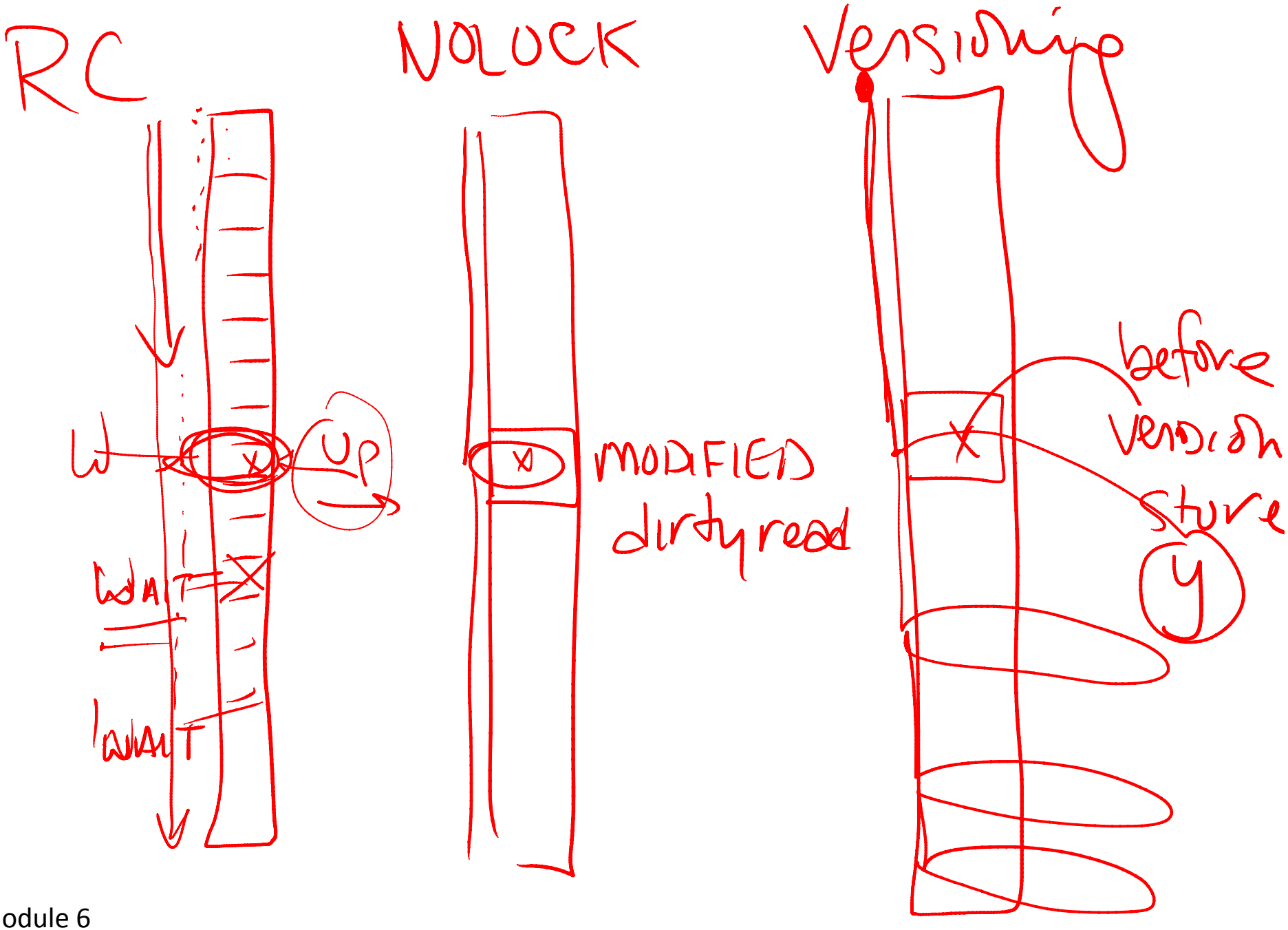
A rollback (without a savepoint name) always rolls back to the beginning of the OUTERMOST BEGIN.





M6s6

This is showing how an update to a CL key value that causes record relocation (after another query has already read [and released the lock on] the row allows the reader to read the row TWICE (non-repeatedly).



Module 6

See the notes on the following page

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

(1) The default – READ COMMITTED

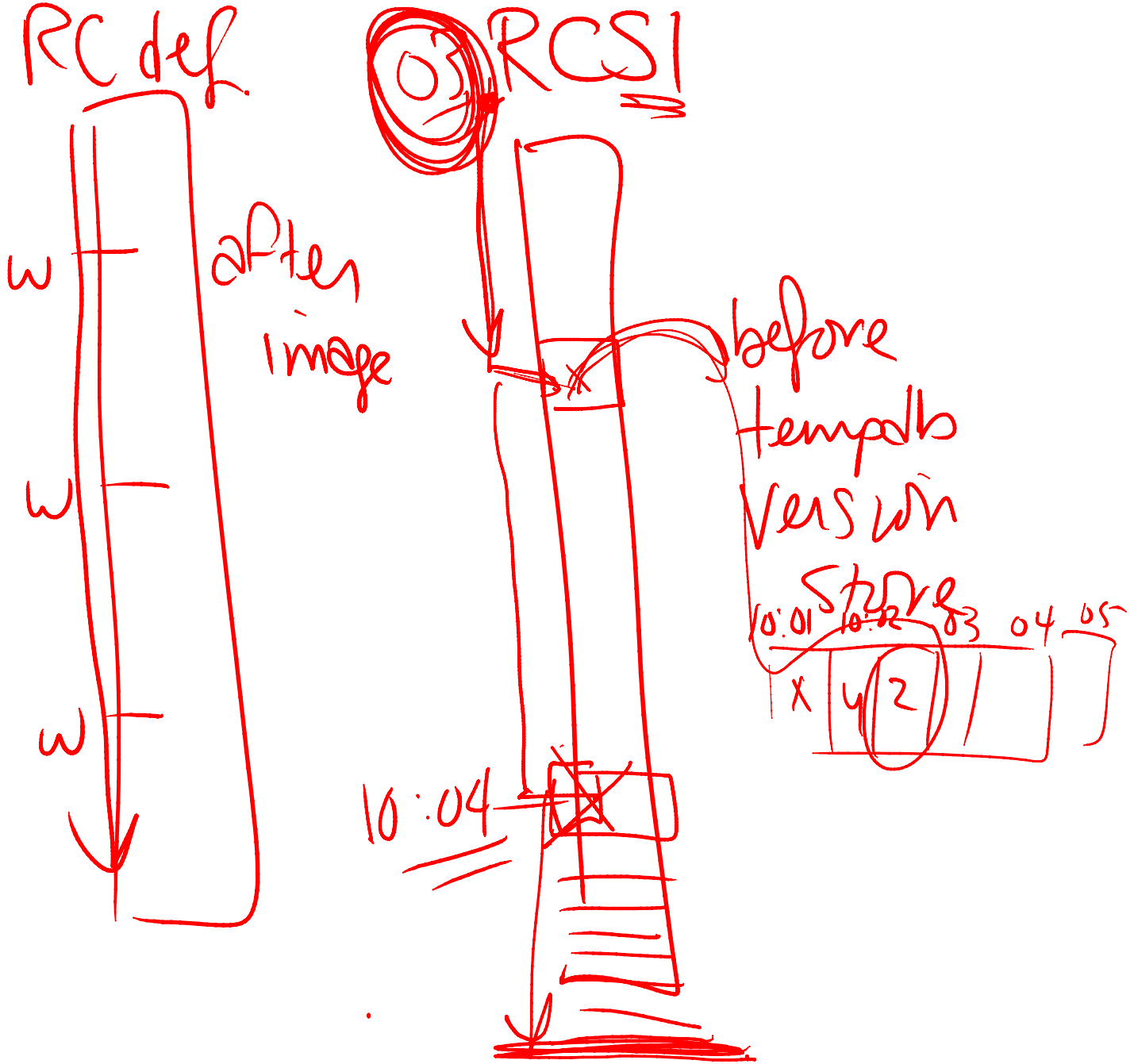
The key things to remember from the picture is that (1) has "hiccups" along the way as they encounter rows that are locked. They read… wait… read… wait. This is *partially* what slows down a statement. However, it's worse that this. It's possible that AFTER a row is read, it will appear again (if the 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).

(2) A forced NOLOCK

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.

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

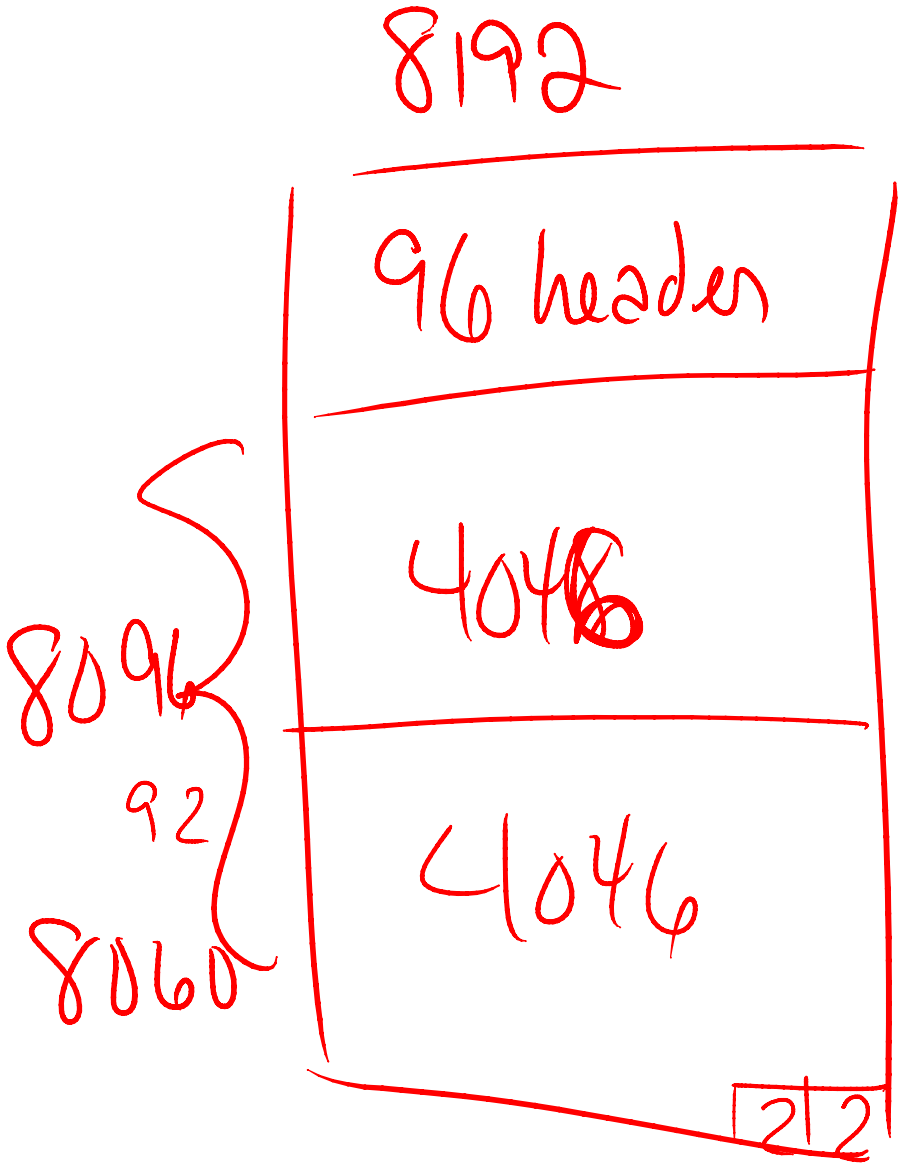
When a versioned query runs the reader will not stop but will have to go to the version store for any row that's in flight. This is a tiny bit slower but guarantees consistency of the ENTIRE read to the point in time when the statement started. This gives you better concurrency as well as a definable 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.



M6 – RSCI/Snapshot Isolation

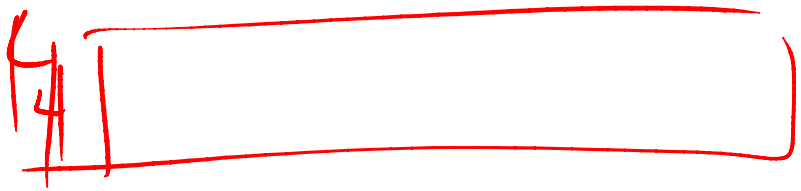
This is just to remind you that the default behavior of RC can lead to inconsistent analysis. (LEFT)

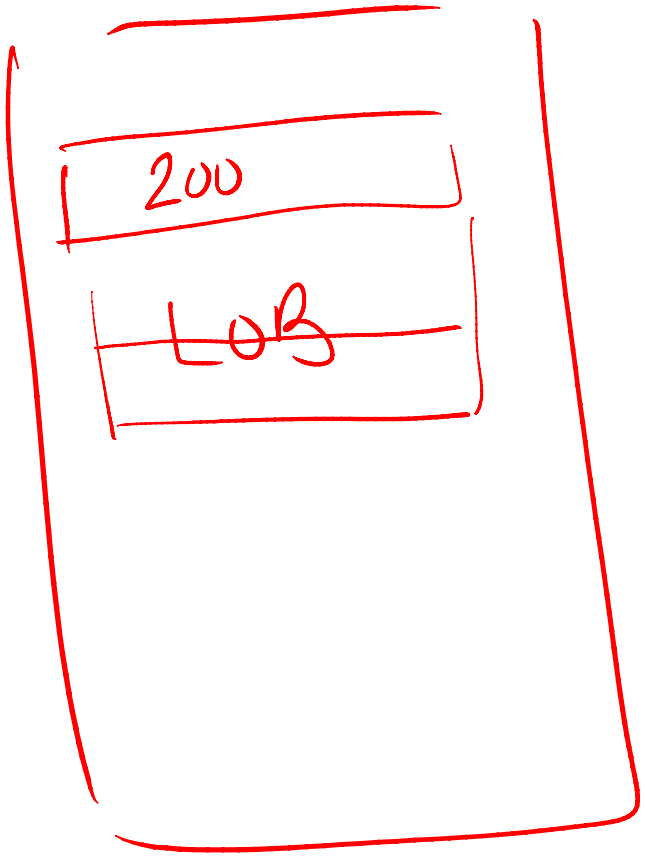
On the right – the reminder is that SQL Server may need MULTIPLE versions to keep many readers consistent/accurate if the data is very volatile.



M7s4

This was a reminder that when you're calculating row size (as well as page density) that you can use 8096 for data rows. Every row requires 2 bytes in the slot array so two rows could each be 4046. However, a single row cannot be more than 8060. The 8060 comes from the maximum amount of space for data (8096) minus a row header (4 bytes) minus "32 bytes" for future growth. Some of which they're already using in SQL Server 2005 and higher. A versioning offset is 14 bytes.





LOB

100 - 200 bytes

majority

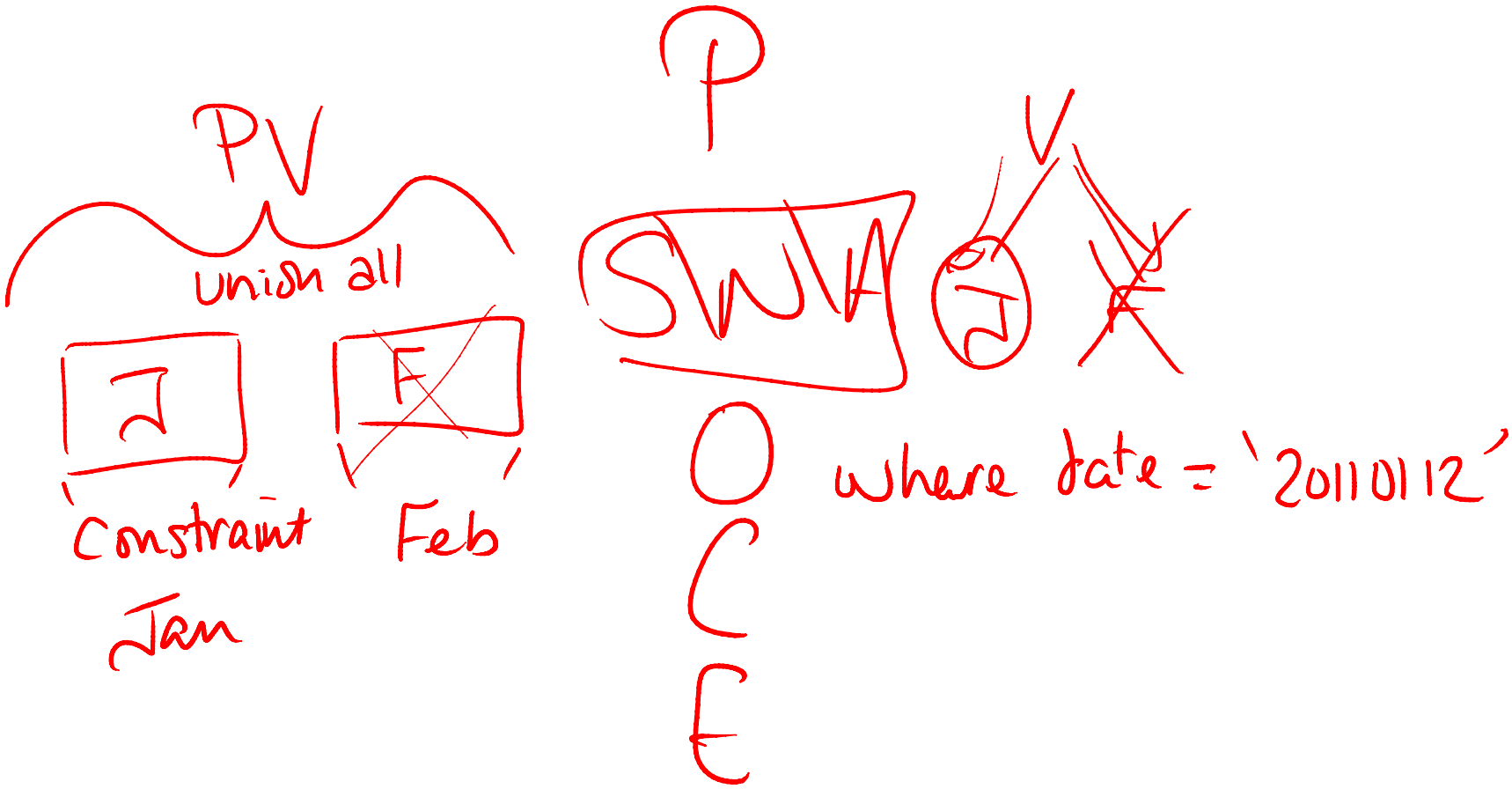
< 2000 bytes

M7s7

Default behavior of a *new* LOB (max or XML) is that they will be put “in_row” if they fit (if the total row size is under 8060).

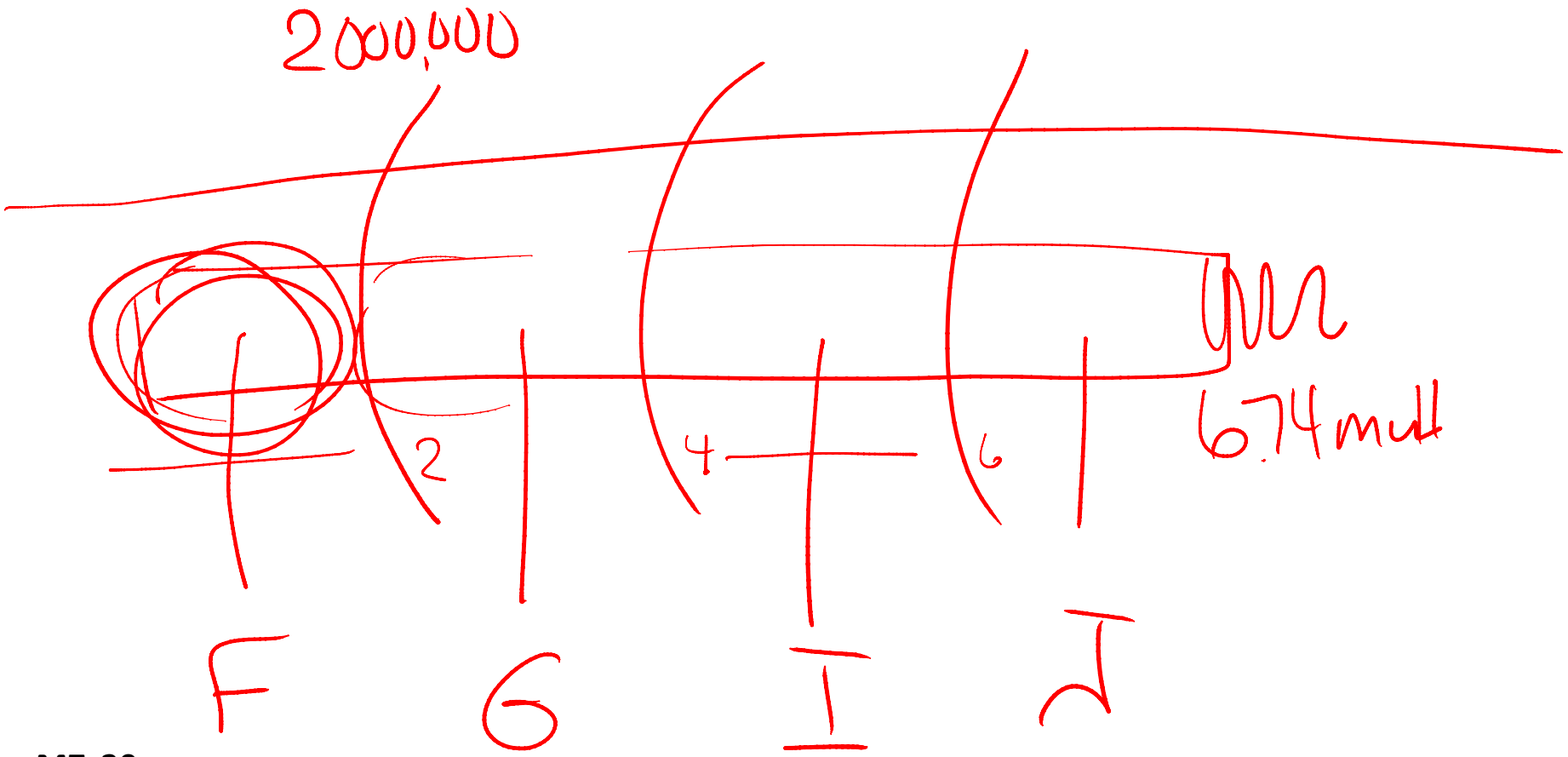
If you have a lot of small LOBs (for example 2K LOBs) then all of your rows will be really wide. If you’re doing a lot of scans and NOT interested in always returning the LOBs then you’re going to waste a lot of pages/IOs to get the LOBs into cache when you don’t really need them.

Consider pushing these small LOBs off-page instead!



M7s27

Constraints are validated during optimization. SQL Server is able – when querying through a view – to generate the query tree and then "prune" the tree. This partition elimination removes any of the redundant tables from access. All of this is as long as the constraint is trusted (or, should I say as long as it's NOT untrusted. ☺)



M7s30

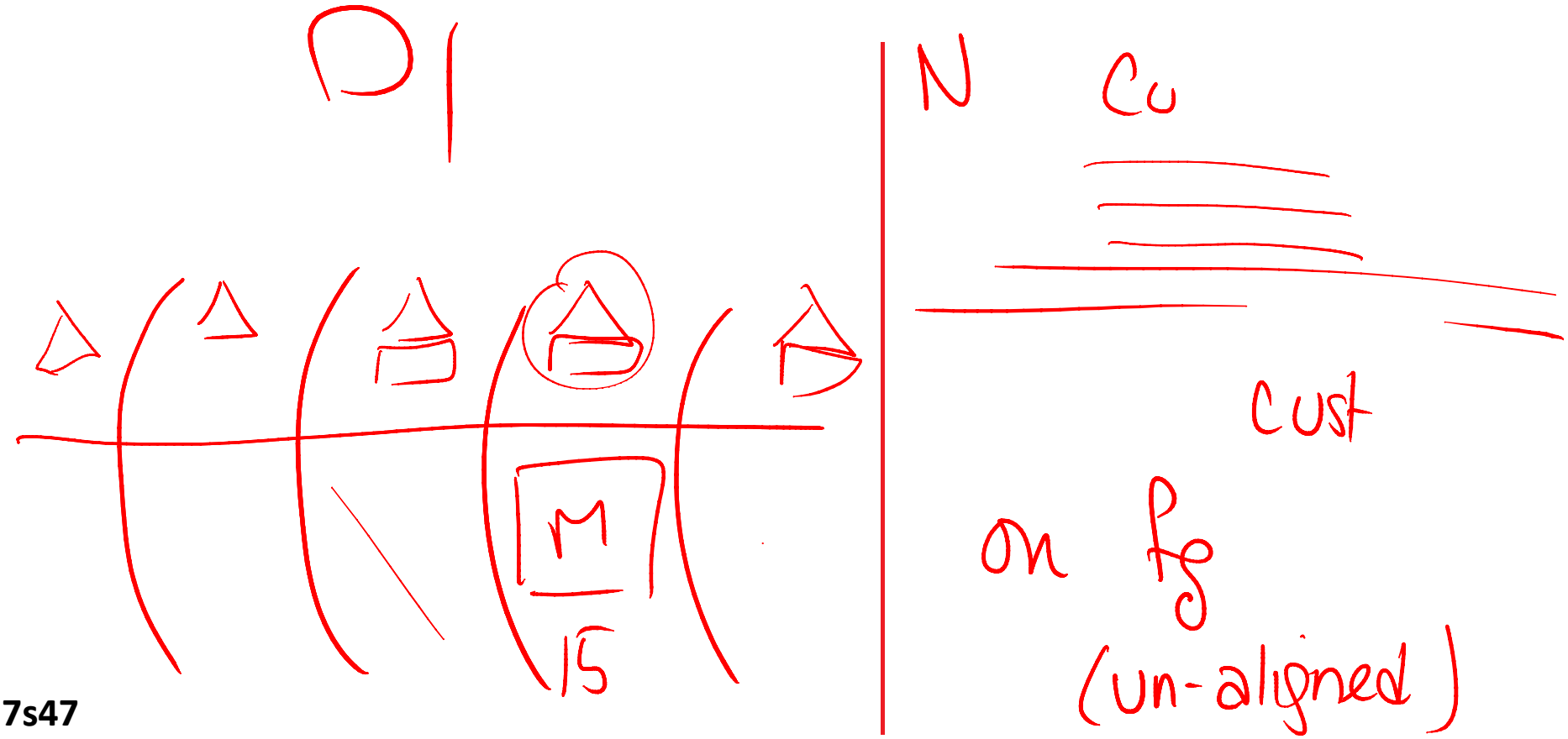
This example is a partitioned table with 3 boundary points: 2million, 4 million and 6million. It's a RIGHT-based partition function so the rows that match the boundary point will be to the RIGHT side. Specifically this translates into:

In partition 1: all rows less than 2 million

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

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

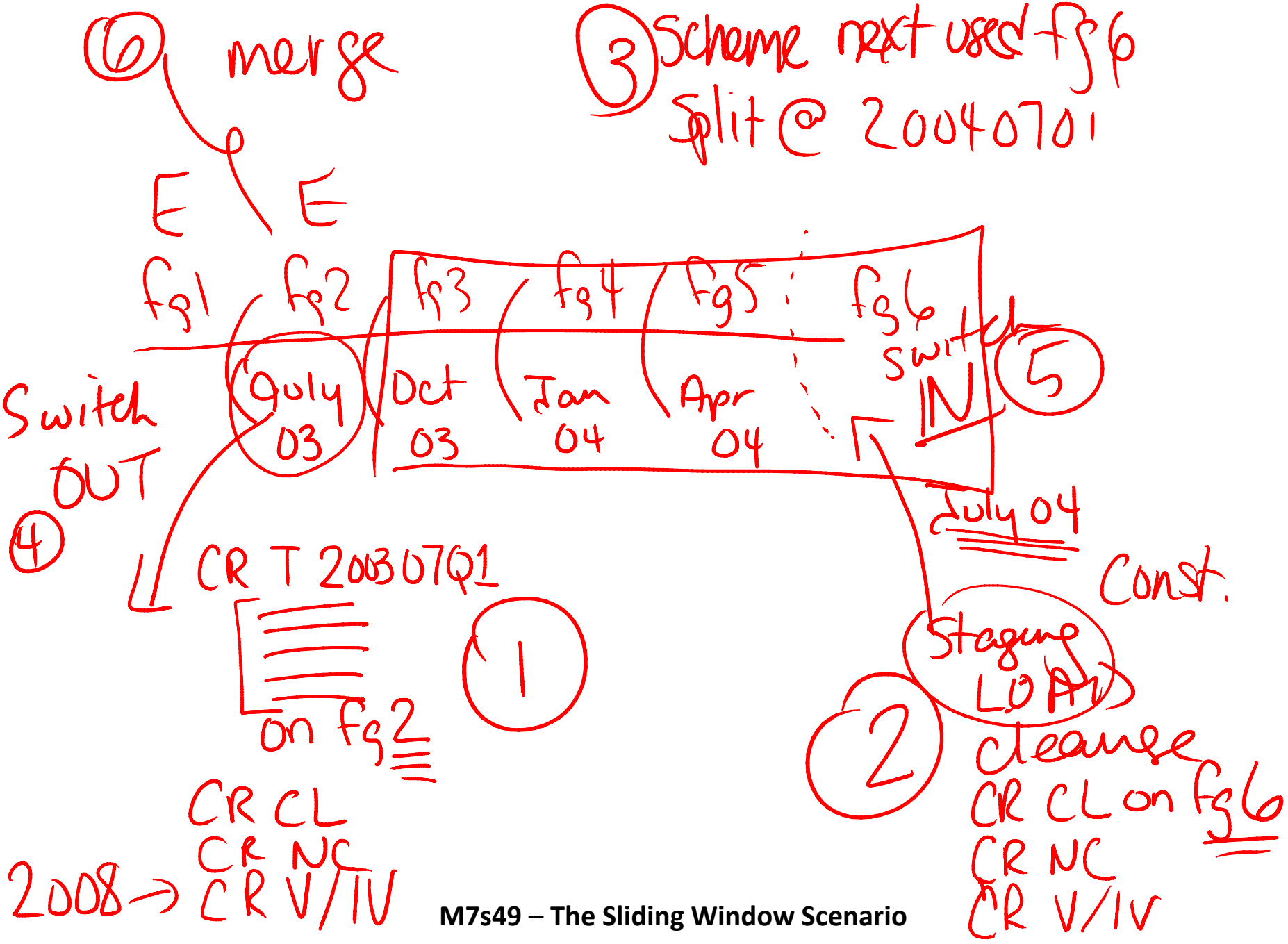
In partition 4: all rows equal to or greater than 6million



M7s47

For fast switching to be allowed – you must have ONLY aligned indexes. For an index to be aligned – there are rules. If the index is going to be unique (and aligned) then it MUST include the partitioning key. Indexes will default to being created on the same scheme as the table (they will default to being aligned – and therefore have all of these requirements).

However, if you're not interested in fast switching – then you can have un-aligned indexes. In this case you create these indexes on a specific filegroup (or on a different scheme). These un-aligned indexes can be unique and do NOT have to include the partitioning column.



M7s49 – The Sliding Window Scenario

See the next slide for full details

The prior slide showed the sliding window scenario – but we went through it slowly:

(1) Preparing the table into which we'll switch OUT our old partition

Review all of the slides for the requirements here but the key point is that this "staging" table MUST be on the same filegroup as the partition you are going to switch out.

(2) Preparing the table for our data load and what will become our new partition to switch IN

Again, be sure to review all of the slides for the requirements here but one thing you need to make sure of is that there's a TRUSTED constraint on this table before you switch in.

(3) Change the partitioned table to support the new filegroup and data range

Always set the NEXT USED filegroup with ALTER SCHEME

Once you have the correct filegroup specified then ALTER FUNCTION…SPLIT

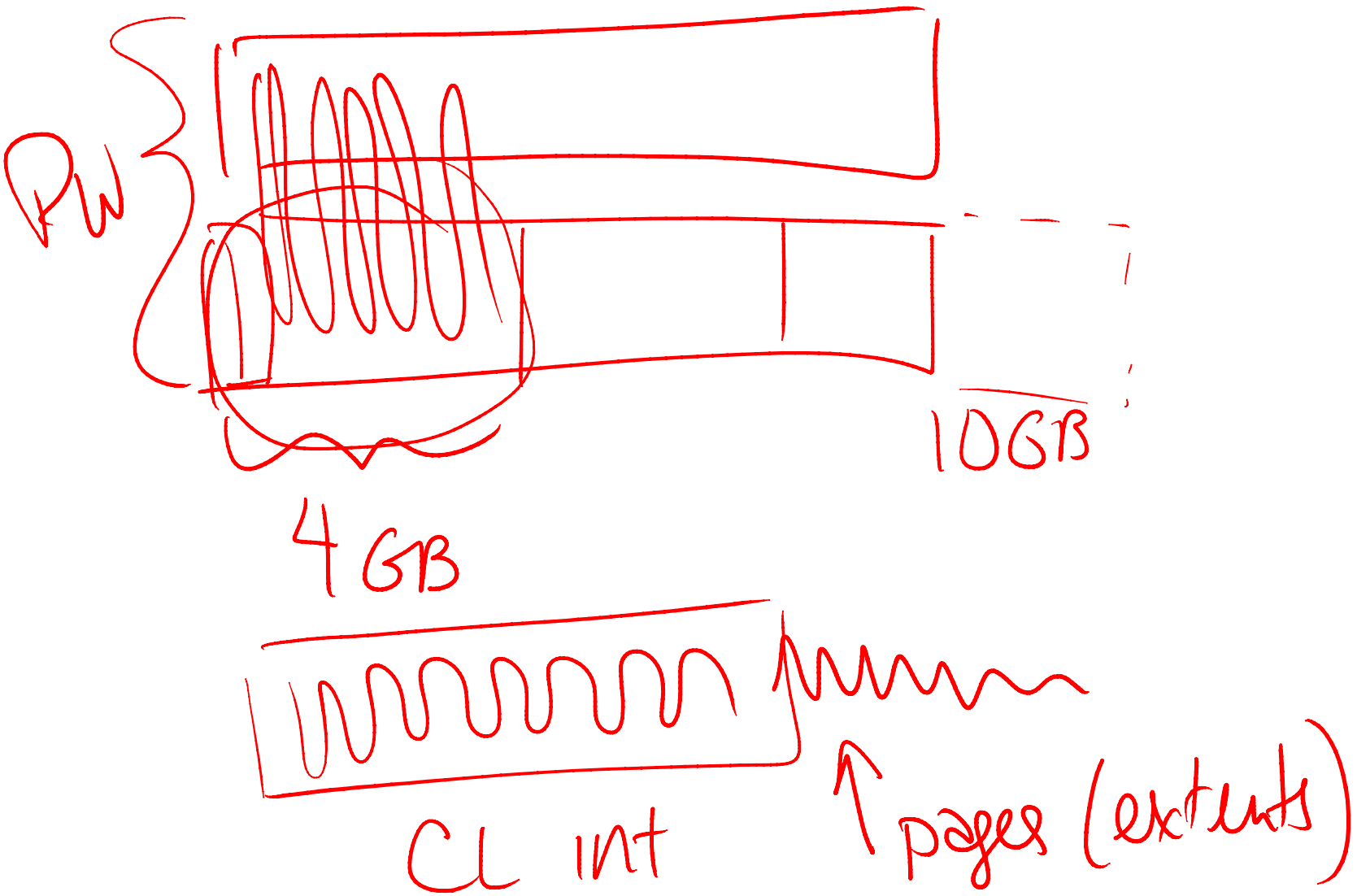
- Now, you're ready

(4) and (5) can be in any order. SWITCH IN the new partition and SWITCH OUT the old.

(6) Clean up

Merge the boundary point but ONLY if it's empty (the next picture will remind you of what happens when you don't MERGE an empty boundary point)

Backup the filegroup where the partition resides that you just switched out. OR, drop the table.



M7s53

To remove contention at the system allocation map level (GAM/SGAM) create two files for read-write filegroups. When objects are requesting space during allocations – your system will be able to handle more due to the ability to round robin through BOTH files instead of having to stay solely within one files first 4GB (a GAM interval).

M7s53

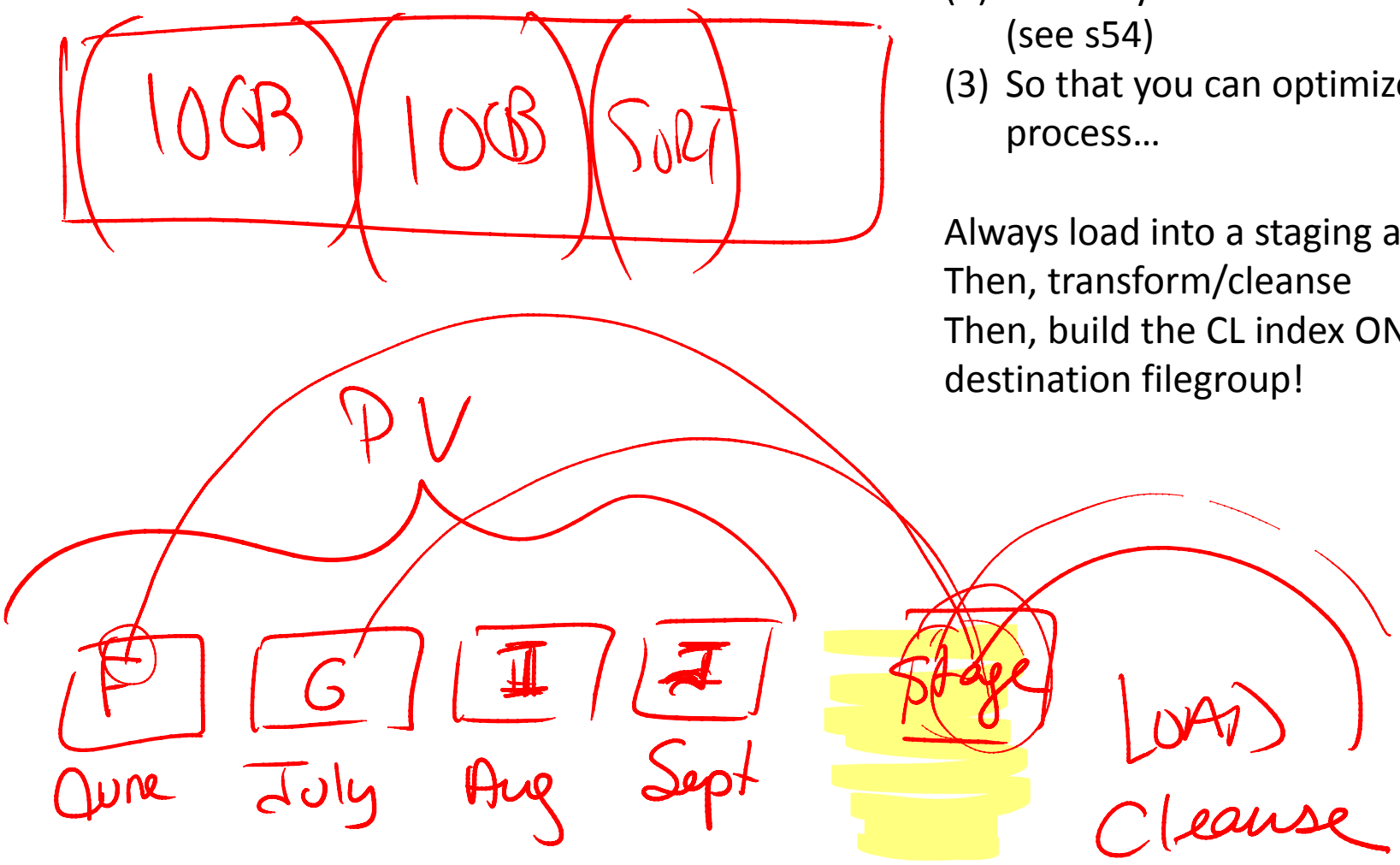
Why do you need a staging area?

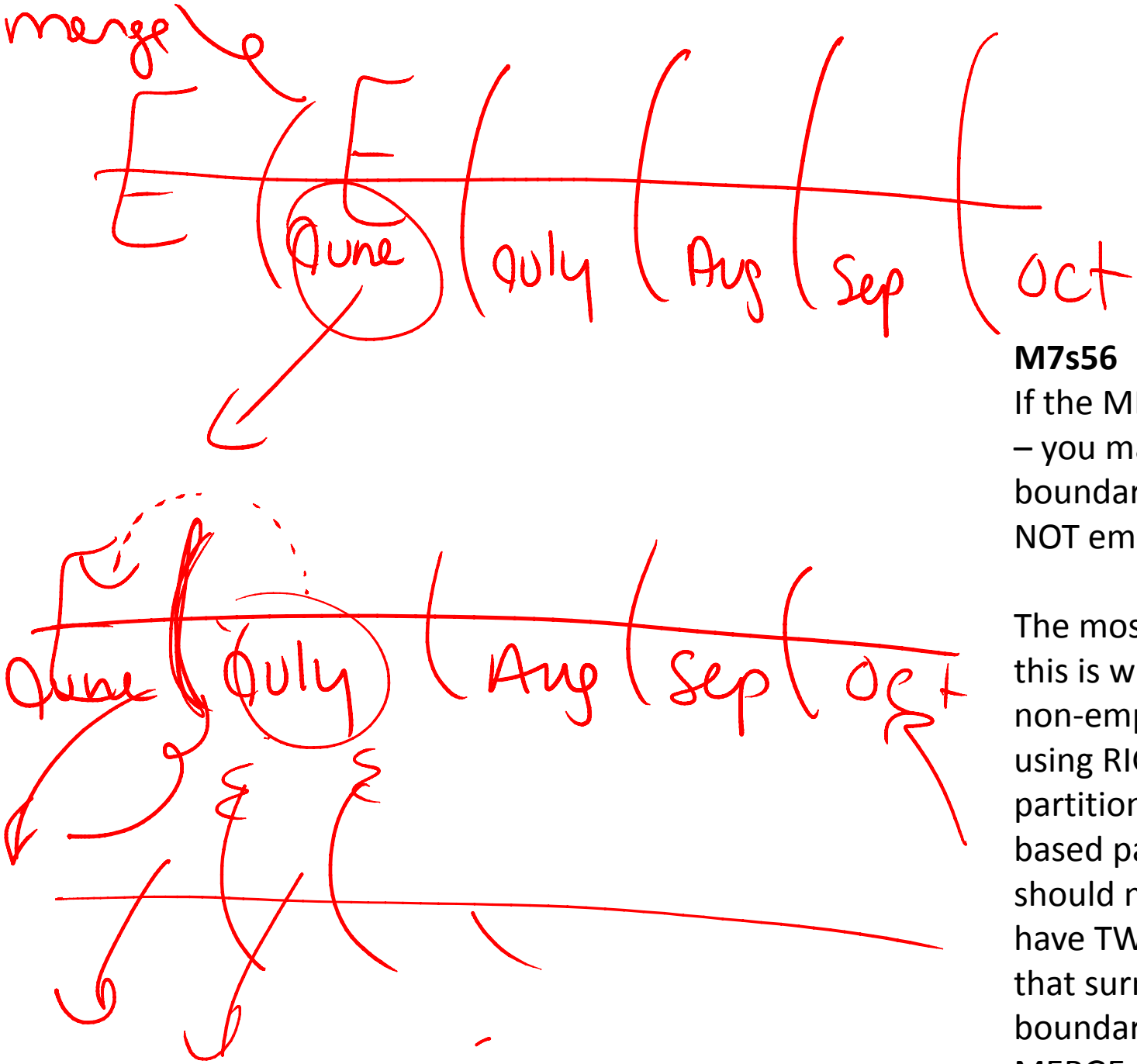
(1) So that you don't need to overallocate space within destination filegroups

(2) So that you don't have to shrink (see s54)

(3) So that you can optimize the process…

Always load into a staging area first.
Then, transform/cleanse
Then, build the CL index ON the destination filegroup!





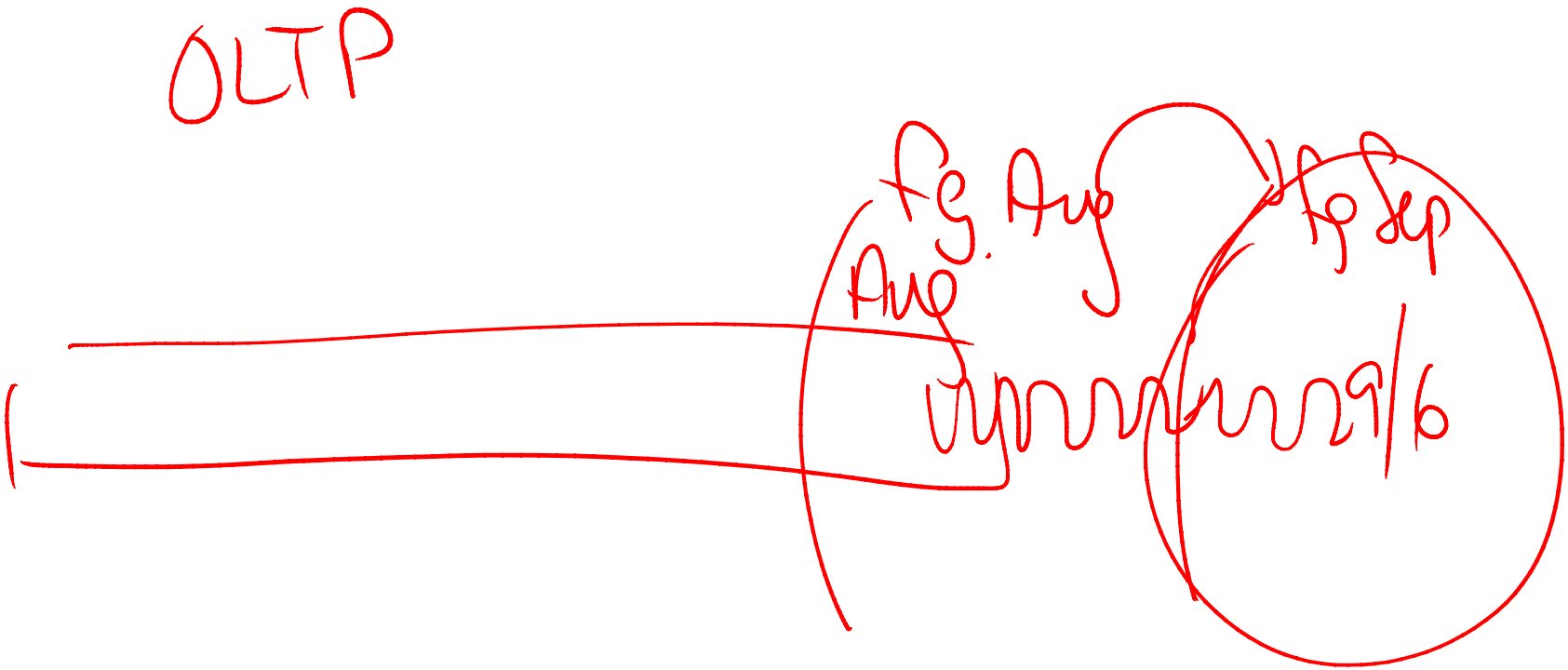
M7s56

If the MERGE process is slow – you may have merged a boundary point that was NOT empty.

The most common case of this is when you start with a non-empty first partition using RIGHT-based partitioning. With RIGHT-based partitioning you should not MERGE until you have TWO empty partitions that surround the emptied boundary. Then, you can MERGE (top diagram).

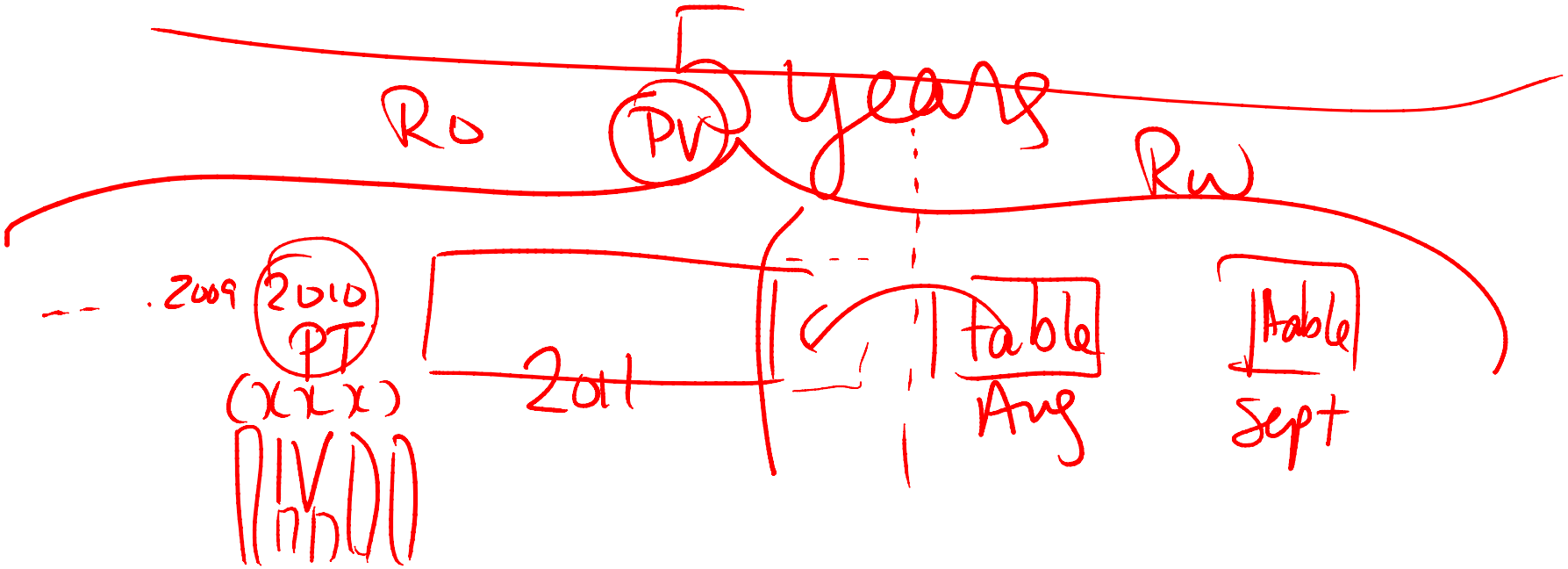
M7s58

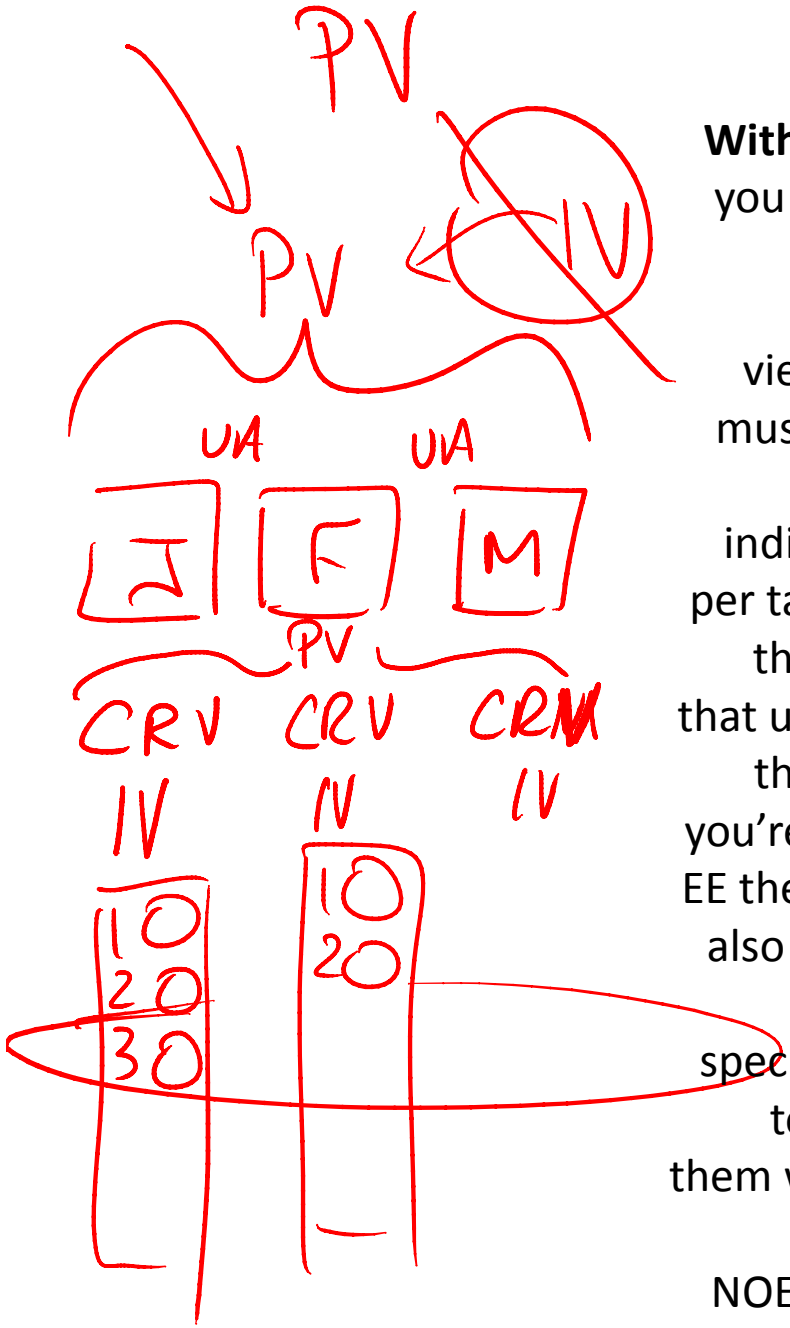
With PTs – it's likely that your last partition (often, the current data) will be active. This is also where it's most likely that you'll have fragmentation. If you want to rebuild ONLY that last partition – you'll need take it offline to do it.



M7s58 (continued)

A better option is to separate the RW from the RO and put the RW in separate tables. Then, you can do online rebuilds at the table level. How do you deal with queries – put a partitioned view over them!

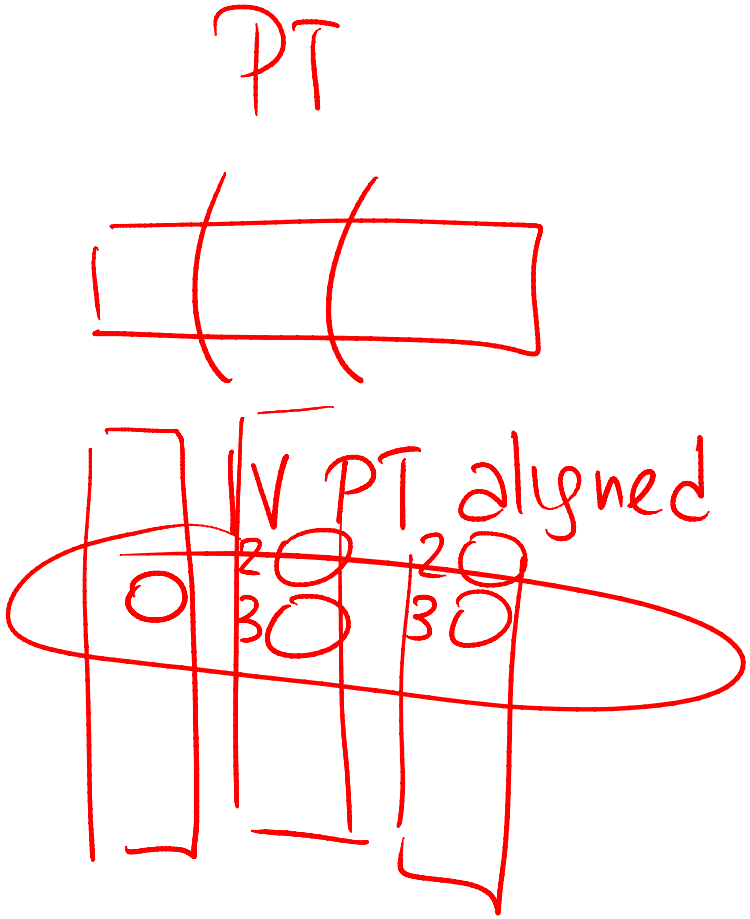


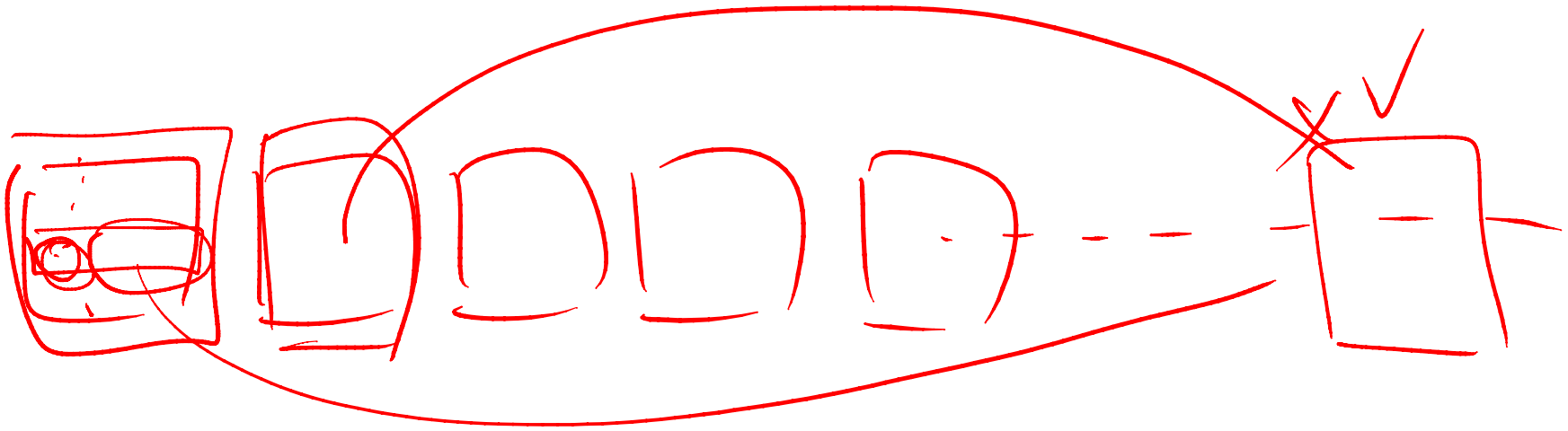


With PVs - If you want to create indexed views, you must create them individually per table (for the tables that underpin the PV). If you're not on EE then you'll also need to create a specific view to access them with the (WITH NOEXPAND) hint.

M7s61

With PTs - If you want to create indexed views, you must create them as partition-aligned (for fast switching). This is a new feature of SQL Server 2008+



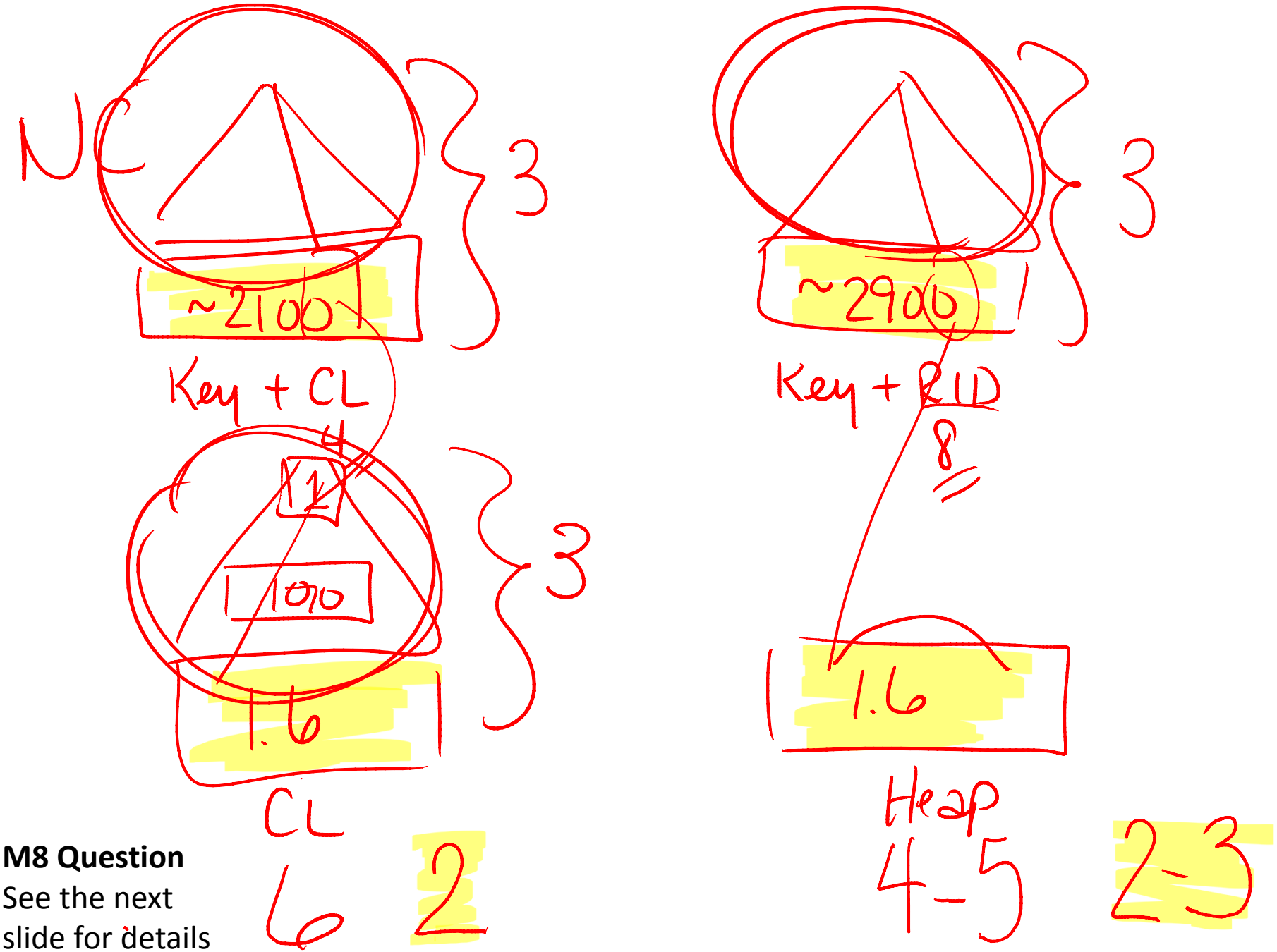


M8s7 (part of – *why heaps are evil*)
Record relocation happens when an update makes a row larger and that row then needs to be moved to another page.

The first page (far left) is where the row is originally inserted.
Then, an update occurs that causes this row to become much larger. And, there isn't enough room on the page to support the update. The forwarding process essentially does this:

1. Leaves a small header on the original page (1 byte)
2. Adds a RID = 8 byes in the format of file (2 bytes): page (4 bytes): slot (2 bytes)
   Note: the slot on the original page is still needed. The total is 11 bytes
3. Moves the row to a page that does have enough free space

The end result is that SQL Server must lock the actual row when a row is requested. This causes an excessive number of IOs to show up (in Logical Reads) when requested. This is especially bad when physical IOs need to be performed!



M8 Question
See the next slide for details

M8 Question

Olaf had asked about the costing (about IOs) for bookmark lookups – between a heap and clustered table.

It *looks* like the CL index is worse:

> Using a nonclustered to do a lookup (= 3 IOs), then – using the CL to lookup the data row = 3 more IOs for a total of 6 IOs

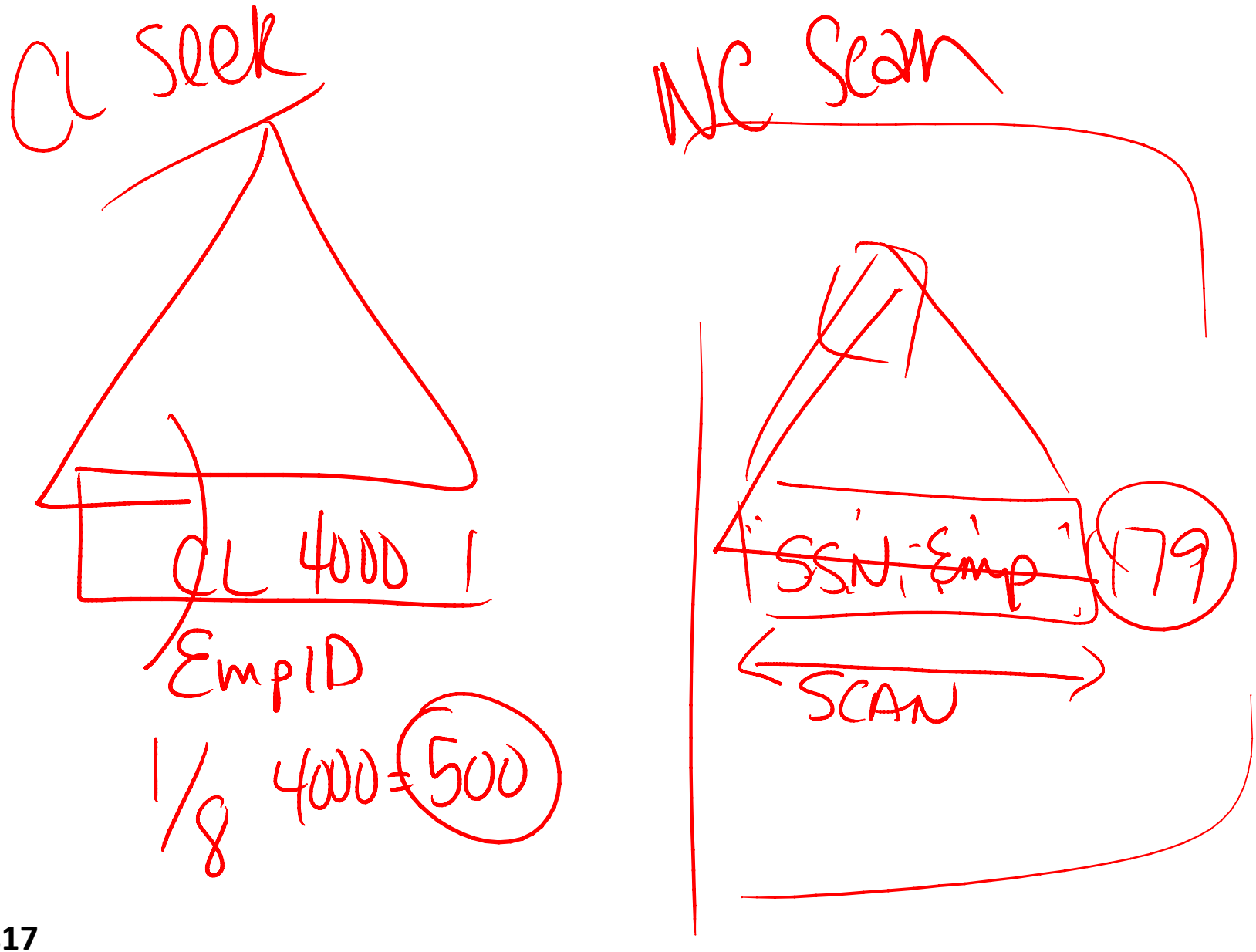
Whereas the head seems to require fewer IOs. Using the nonclustered is about the same (=3 IOs), then – using the Heap to access the data row is 1 (possibly 2 if there's been record relocation) for a total of 4-5 IOs.

The long story short is that 4-5 IOs seems less than 6 IOs. Yes, the number is less but the IOs are potentially more expensive. The yellow highlighting show where the more expensive IOs are going to be performed (which is predominantly in the leaf structures).

As a result, a bookmark lookup from a NC to a clustered has 2 potentially physical IOs.

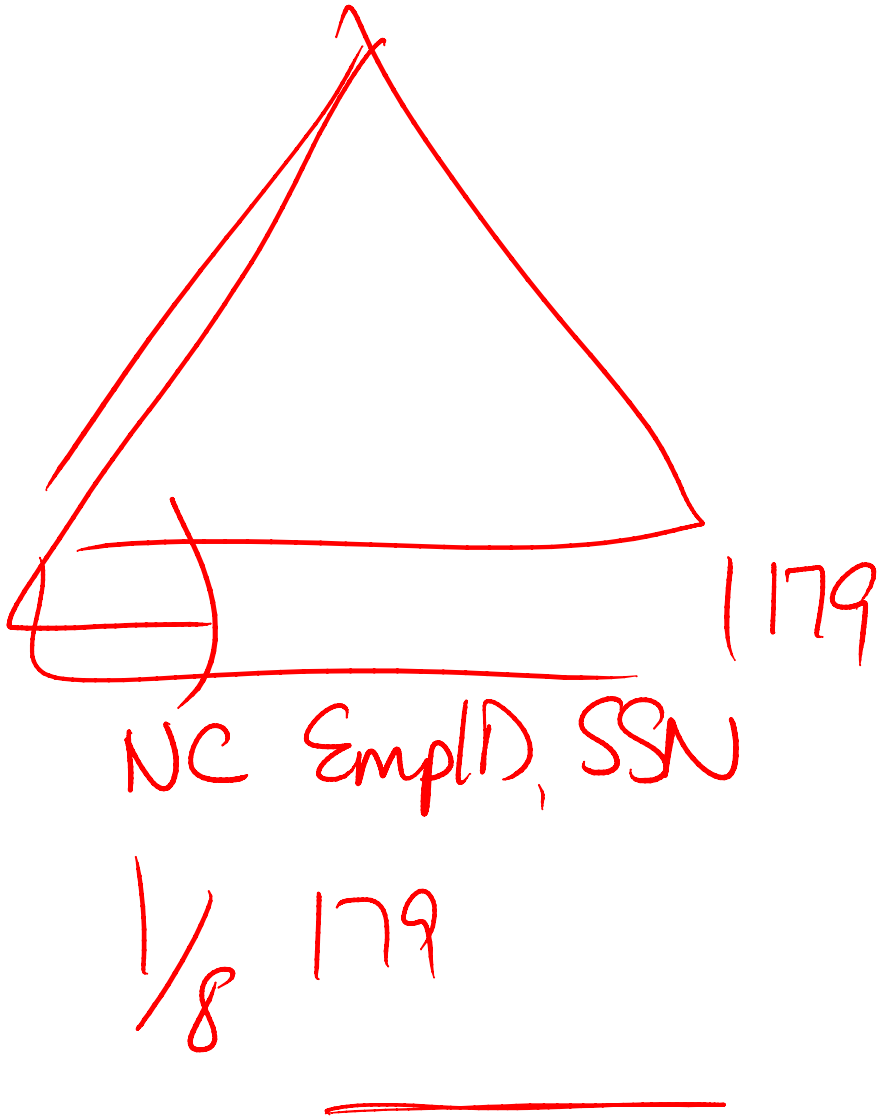
Whereas the lookup from a NC to a heap has 2-3 potentially physical IOs.

While many lookups might be the same – there are still OTHER reasons for why heaps are not ideal. This is just yet-another-one. ☺



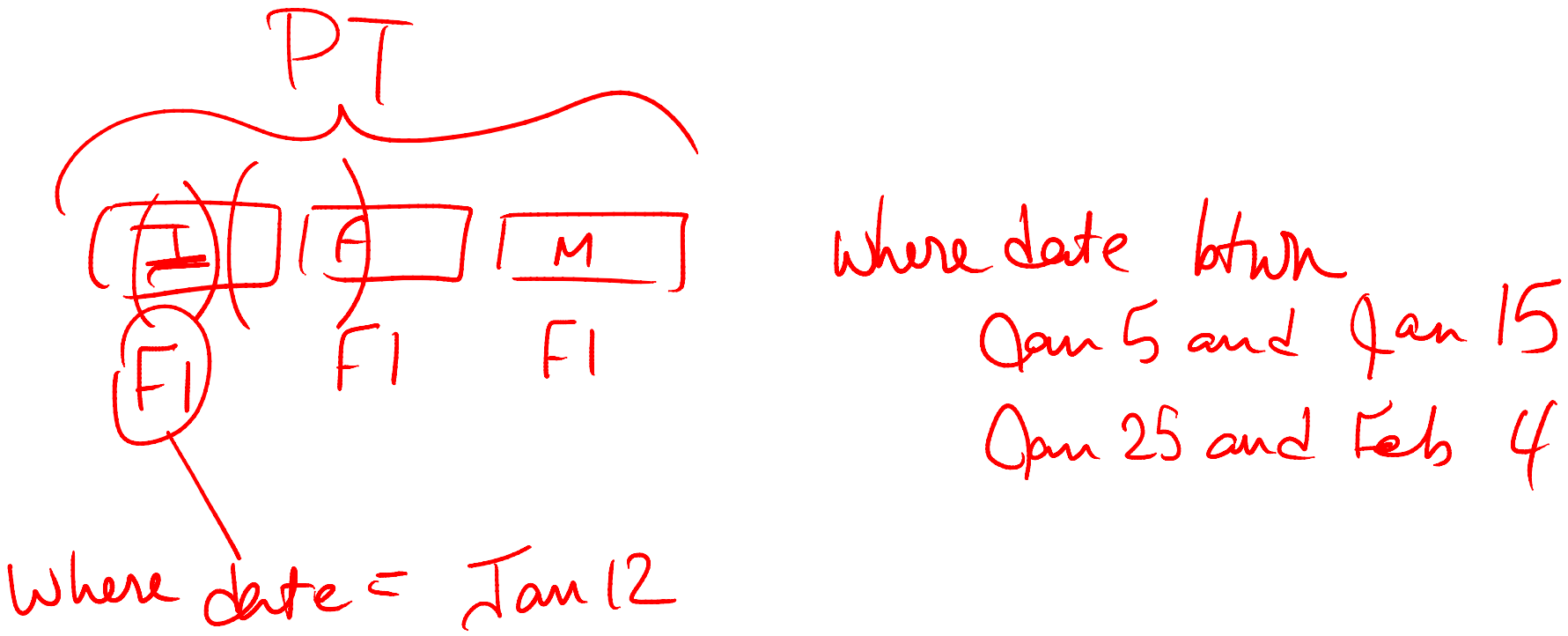
M9s17

This was just highlighting the access patterns for the example that I showed where the nonclustered scan is fewer IOs than the clustered index seek.



M9s18

If this query were VERY important and critical – I'd consider a nonclustered covering index that's seekable… the read would only require 1/8 of the 179 pages.



M9s42

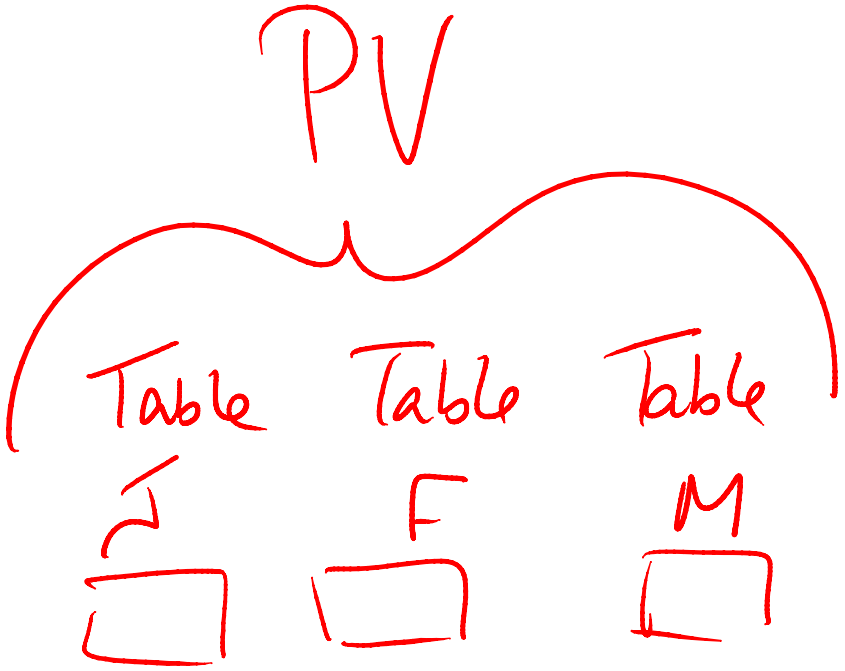
If you were to create a filtered index on each partition – SQL Server could use it for any query whose predicate was a subset of the filtered index.

For example, you have a filtered index for January and a second filtered index for February.

If a query were:

`SELECT * FROM PT WHERE date = '20110112'` then the FI/FS could be used OR if the query were **`SELECT * FROM PT WHERE date BETWEEN '20110105' AND '20110115'`** that's OK too.

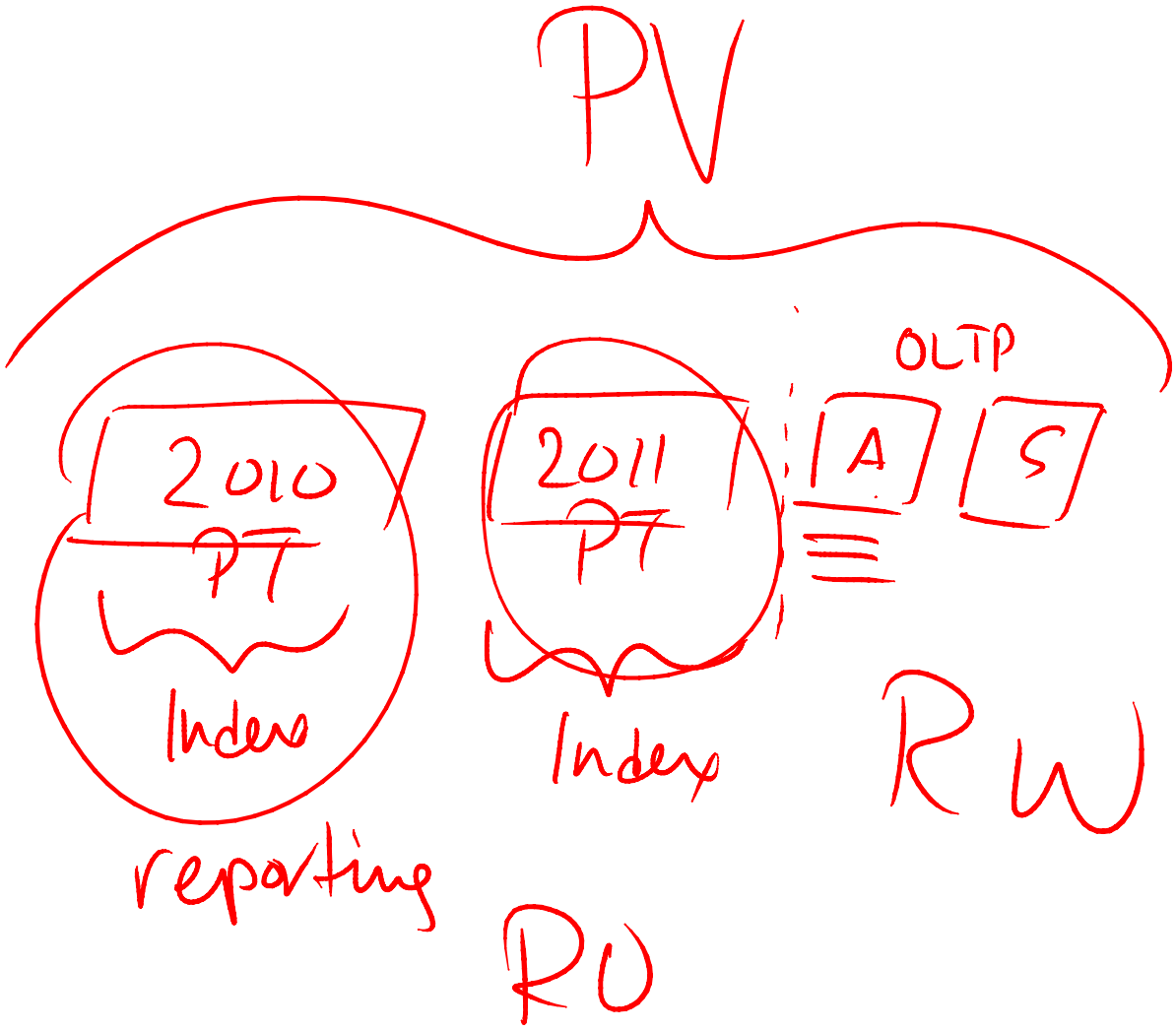
However, if query were to request **`SELECT * FROM PT WHERE date BETWEEN '20110125' AND '20110204'`** then SQL Server CANNOT use the two filtered indexes/stats and instead will have to use the table's indexes/statistics.



M9s42 – Separate tables can be better than filtered indexes over partitions

With separate tables being accessed through a view there's no other option but to use BOTH table's statistics/indexes together. This is (yet another) reason to break large tables down into smaller tables. All of the following queries would work through the PV – as long as all of the base tables of the view had real/full-table (and unfiltered) indexes:

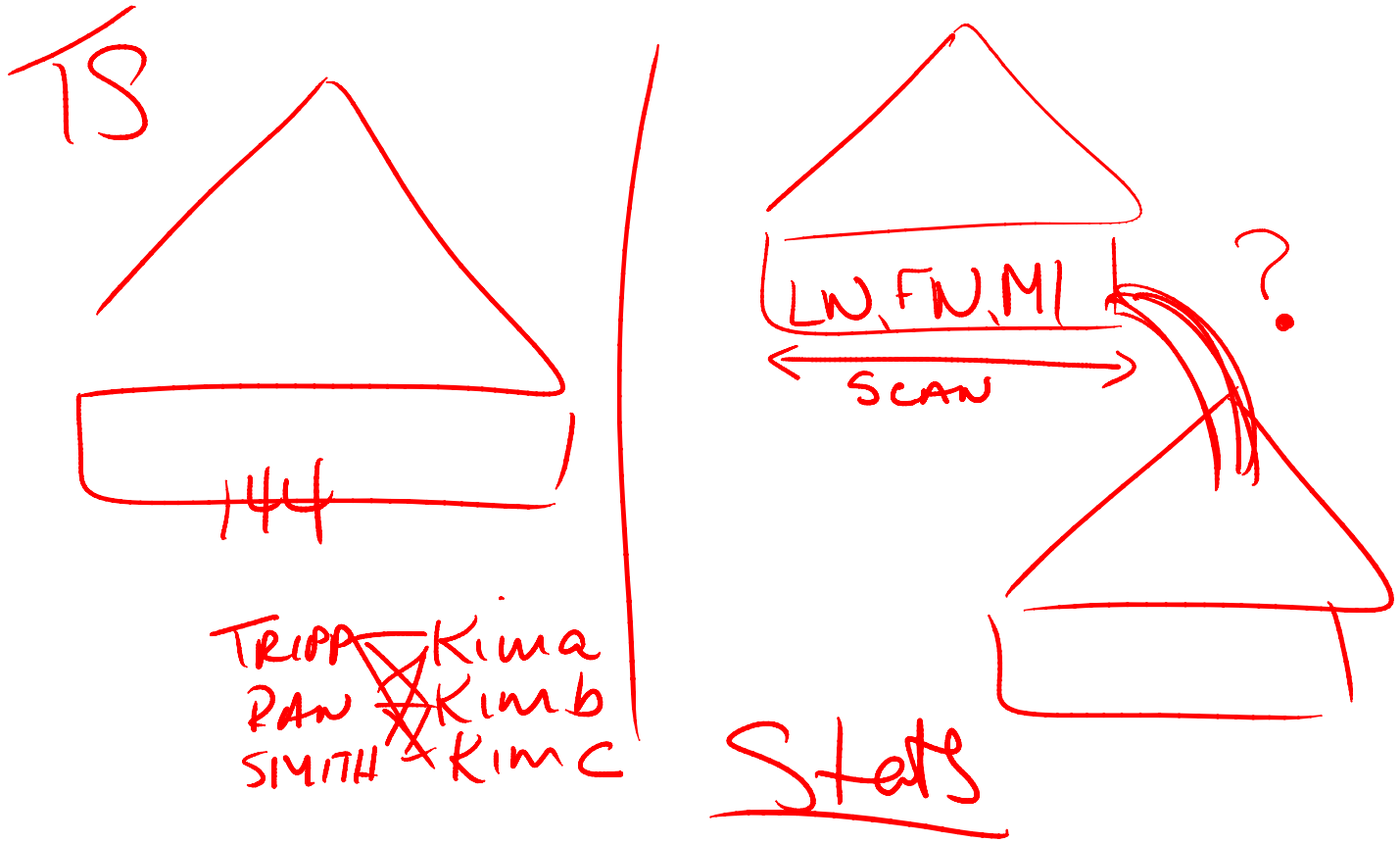
```
SELECT * FROM PV WHERE date = '20110112'
SELECT * FROM PV WHERE date BETWEEN '20110105' AND '20110115'
SELECT * FROM PV WHERE date BETWEEN '20110125' AND '20110204'
```



M9s42 – Separate tables can be better than filtered indexes over partitions

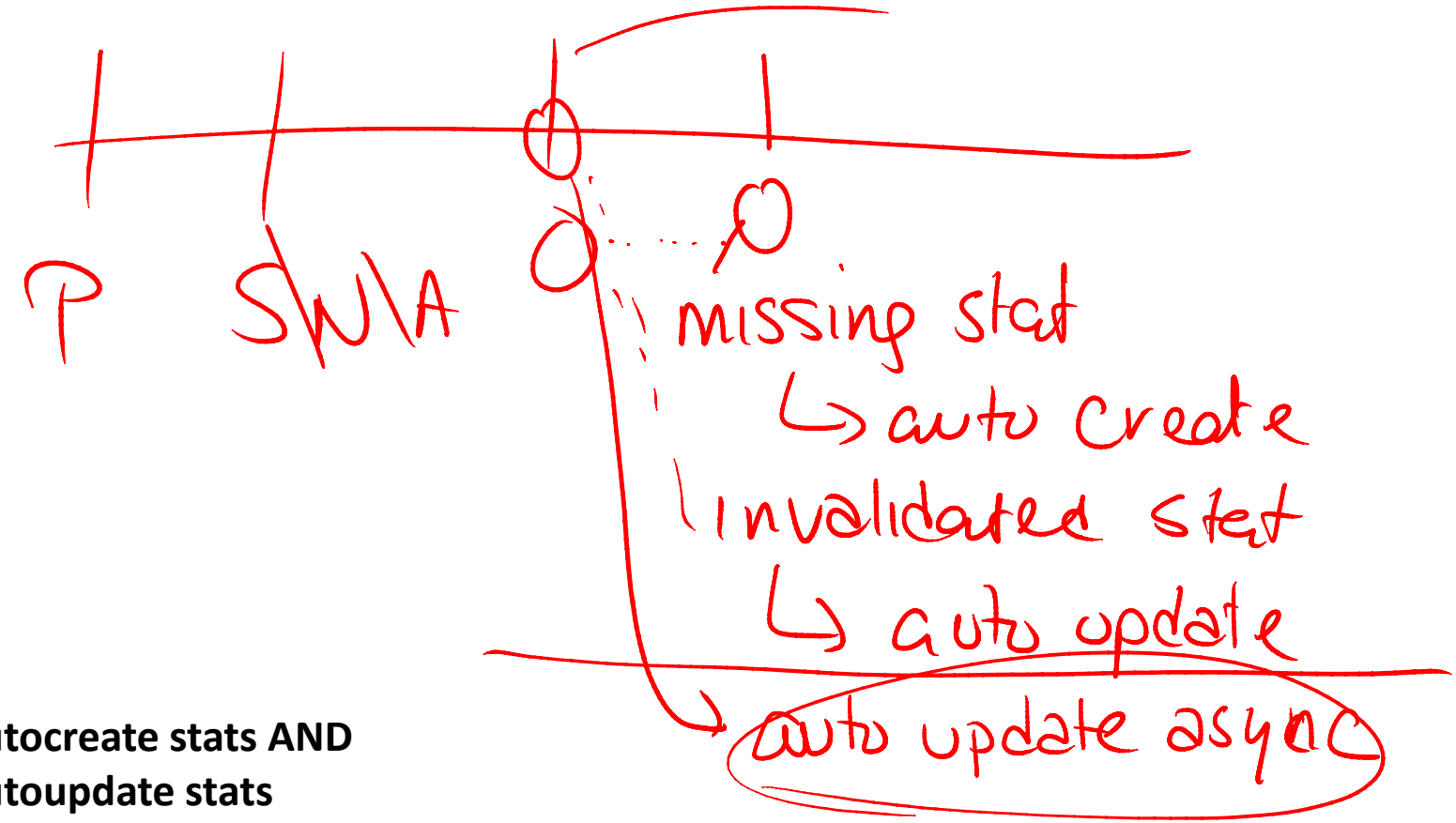
And, you can end up indexes them differently (which is the benefit of filtered indexes/filtered stats.

I'm not saying NOT to use filtered indexes/stats but I am saying that for some applications – PVs with separate base tables (that have different indexing strategies) might be better!



M10s10 – Column-level distribution from left-based density subsets…

This relates to the query on slide 10 and the idea that I was trying to show here is that even though the left-based density shows that last names are horribly NOT unique and the combination of last name & first name IS *almost* unique – that doesn't imply anything about first names alone. I could have created a data set of firstnames of Kima, Kimb, Kimc and then multiplied that with last names (Tripp, Randal, Smith) and I would have had similar statistics for last name alone and for the combination of last name, first name. SQL Server does NOT gamble on this – SQL Server creates column level statistics on first name.



M10s13 – autocreate stats AND
M10s16 – autoupdate stats

During optimization – SQL Server might hit one of two events:

Missing statistics event

If “auto create stats” is on then the optimizer will wait until the statistics have been created to optimize.

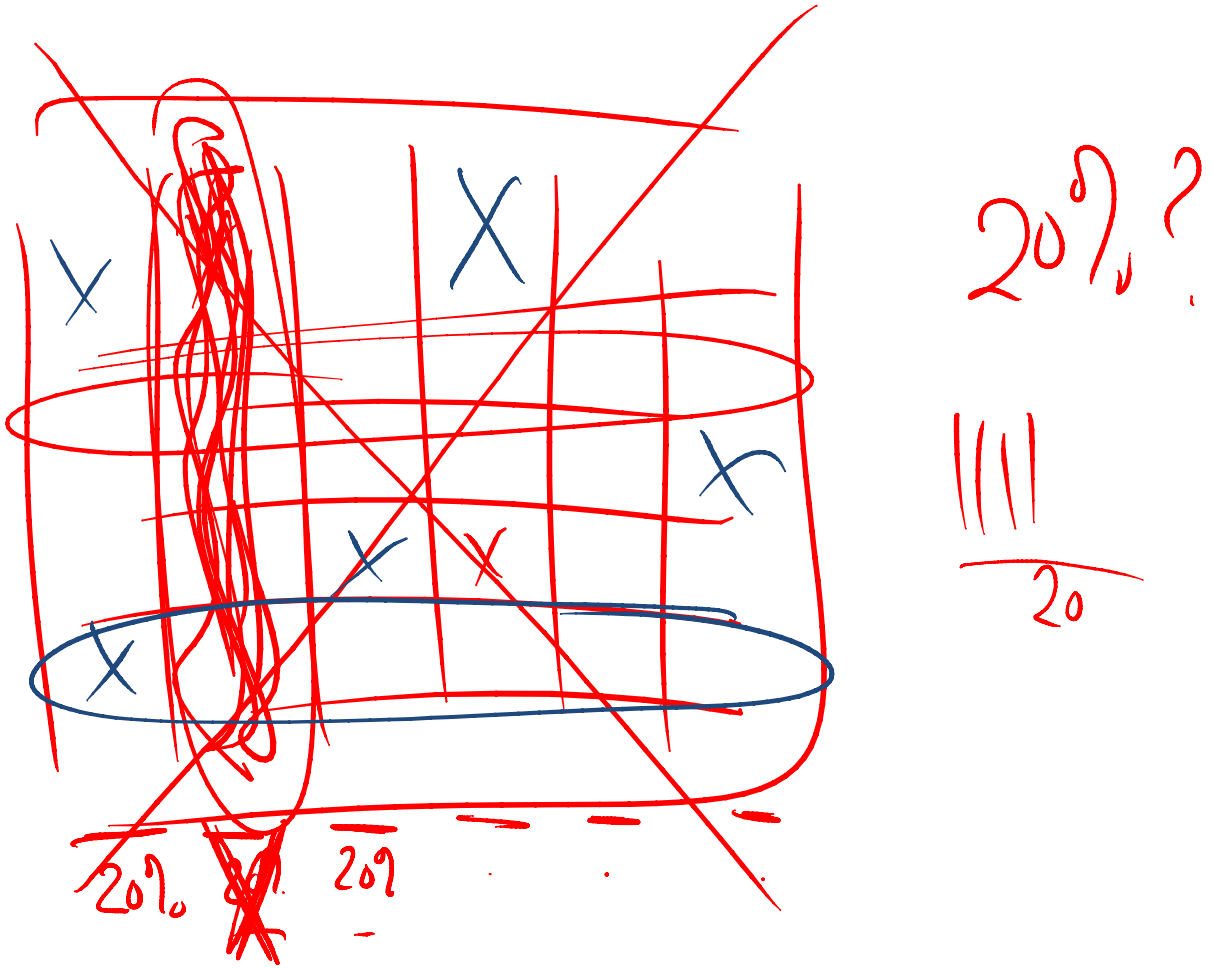
Invalidated statistics event

If “auto update stats” is on then the optimizer will wait until the statistics have been updated to optimize.

If “auto updates stats async” are on then SQL Server will optimize using the invalidated statistics. Then, asynchronously, SQL Server will update the stats.

M10s13 – autocreate stats (same as prior slide)

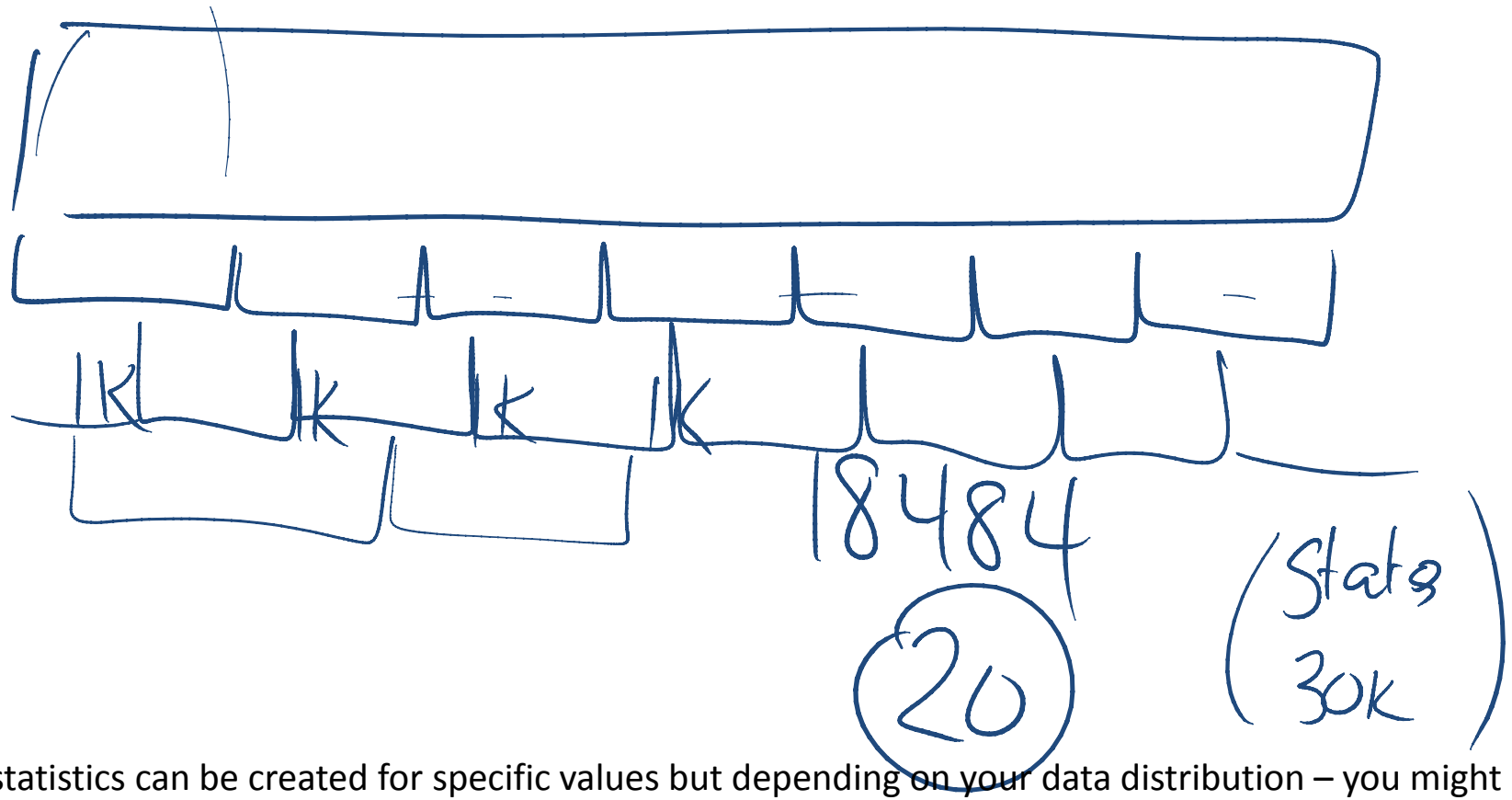
This is just the reminder of what happened during the demo. When the query ran with FIRSTNAME LIKE 'Kim%' and SQL Server did not have enough information AND auto create stats is ON – then, SQL Server creates the statistic, optimizes and executes.



M10s17

This is a diagram showing the pros/cons of the methods for statistics invalidation. SQL Server 7.0 and 2000 used a rowmodctr (sysindexes.rowmodctr) to do invalidation. Even though SQL Server doesn't use this anymore – you can (through the backward compatibility view sysindexes). In 2005+ they moved to an internally defined colmodctr:

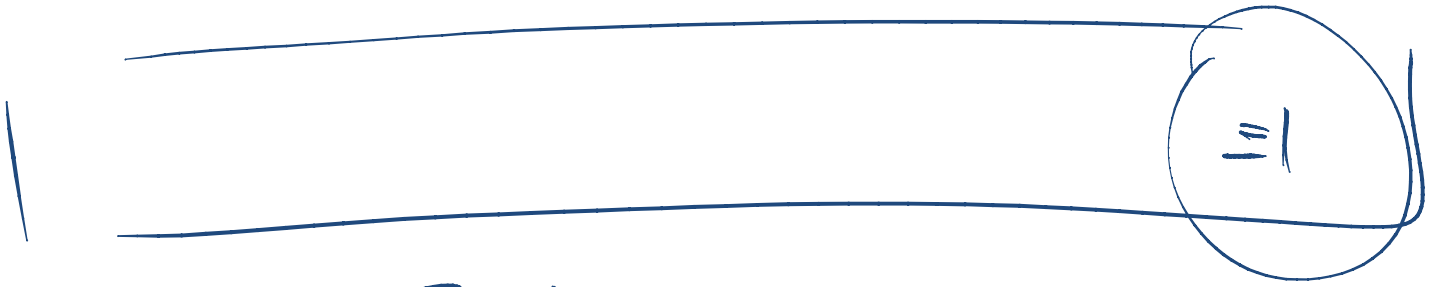
- The pro is that you don't invalidate too soon (when you have a highly volatile column)
- The con is that you might not invalidate soon enough if your modifications are reasonably distributed.



M10s23

Filtered statistics can be created for specific values but depending on your data distribution – you might want to divide the data into buckets and then create a [filtered] statistic for each of those buckets. The example was 31 million sales over ~18K customers. By creating a statistic for each 1K customers you have statistics that are effectively 19x more detailed. Even just 10 buckets gives you 10 times more detail. However, it still might not be detailed enough. You'll need to test it and check the histogram. Because of potential ***interval subsumption**** issues– some have asked if it would be beneficial to create additional stats at different intervals… you could but it would really depend on the queries. Having said that – the bigger the range that the query is interested the more the averages are more accurate. So, really, this issue is to significantly help queries that are more targeted (where the stats just weren't good).

* The optimizer can detect whether interval conditions in a filtered index cover, or "subsume" interval conditions of a query.



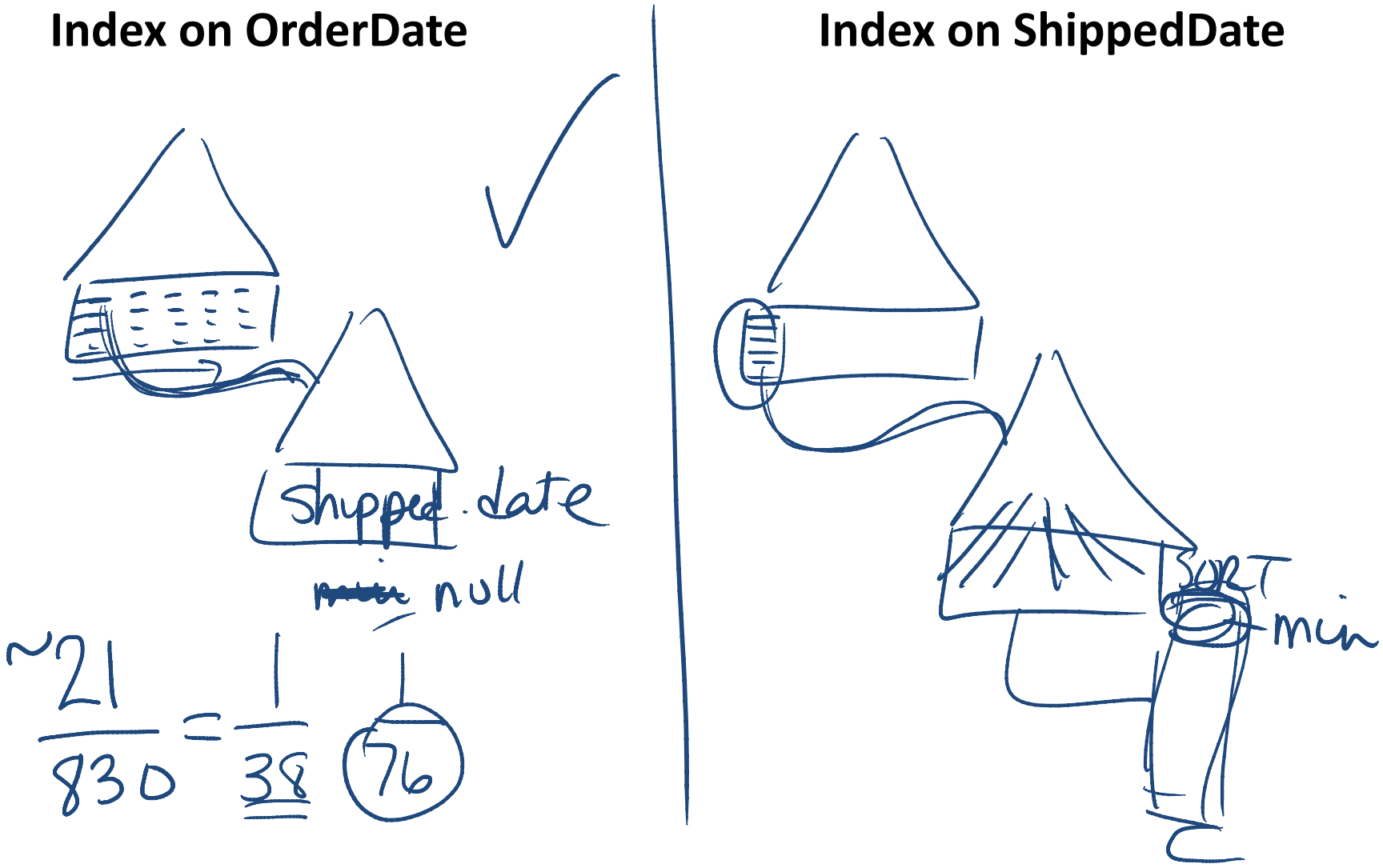
M10s26

The bad news… there's always bad news.

For Updates:

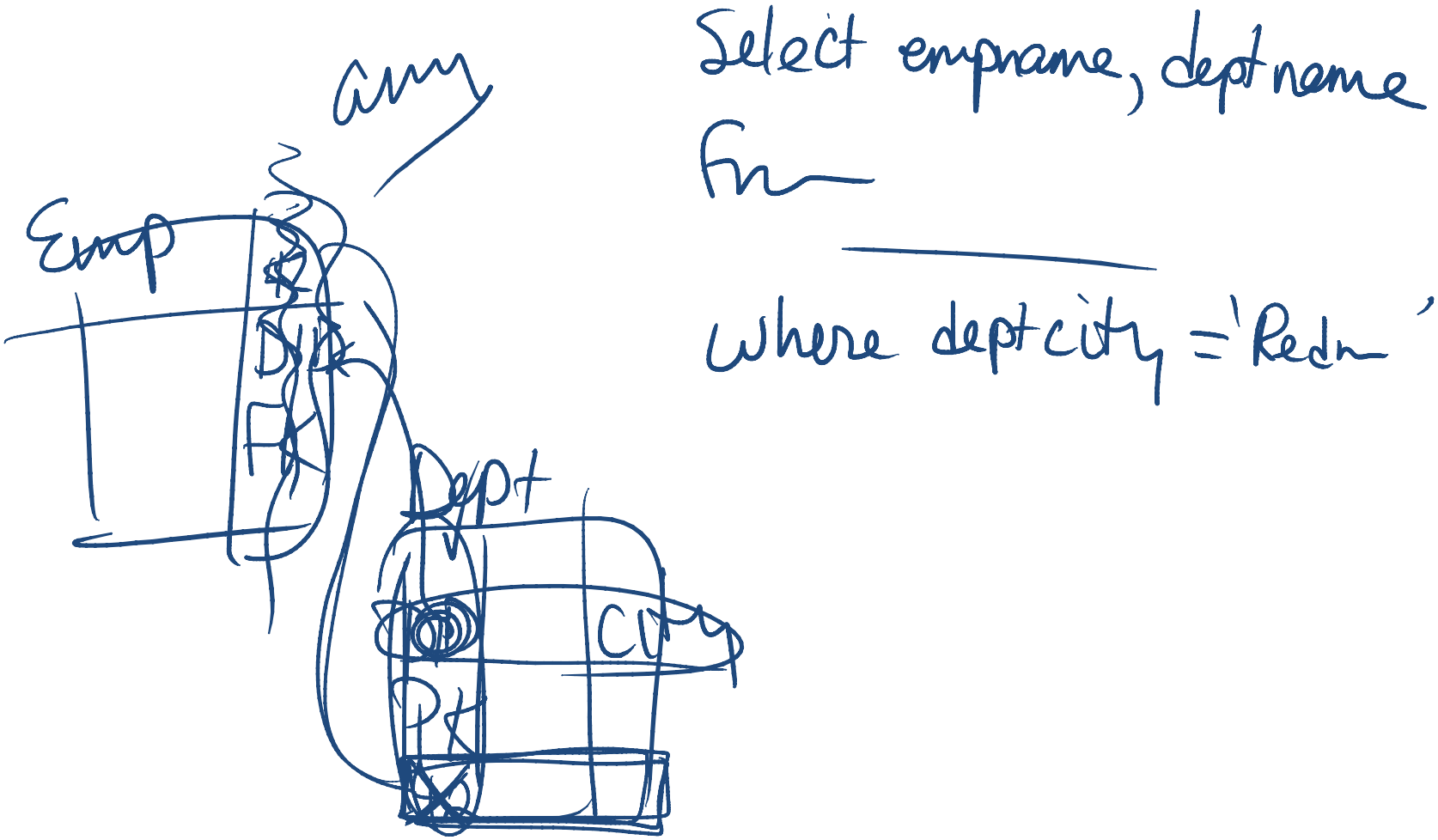
statistics for a filtered index OR a filtered stat – do NOT get updated until that column's statistics get updated (which is when the colmodctr) is reached

Forced parameterization: even when SQL Server would NOT have parameterized the statement (because it had deemed the statement's plan as "unsafe" to re-use), forced param will force it. In systems where plans are very stable (but A LOT of adhoc) then this could be great. However, the example of status = 1 (in your query) is converted to status = @1 so the filtered index/filtered stat cannot be used.



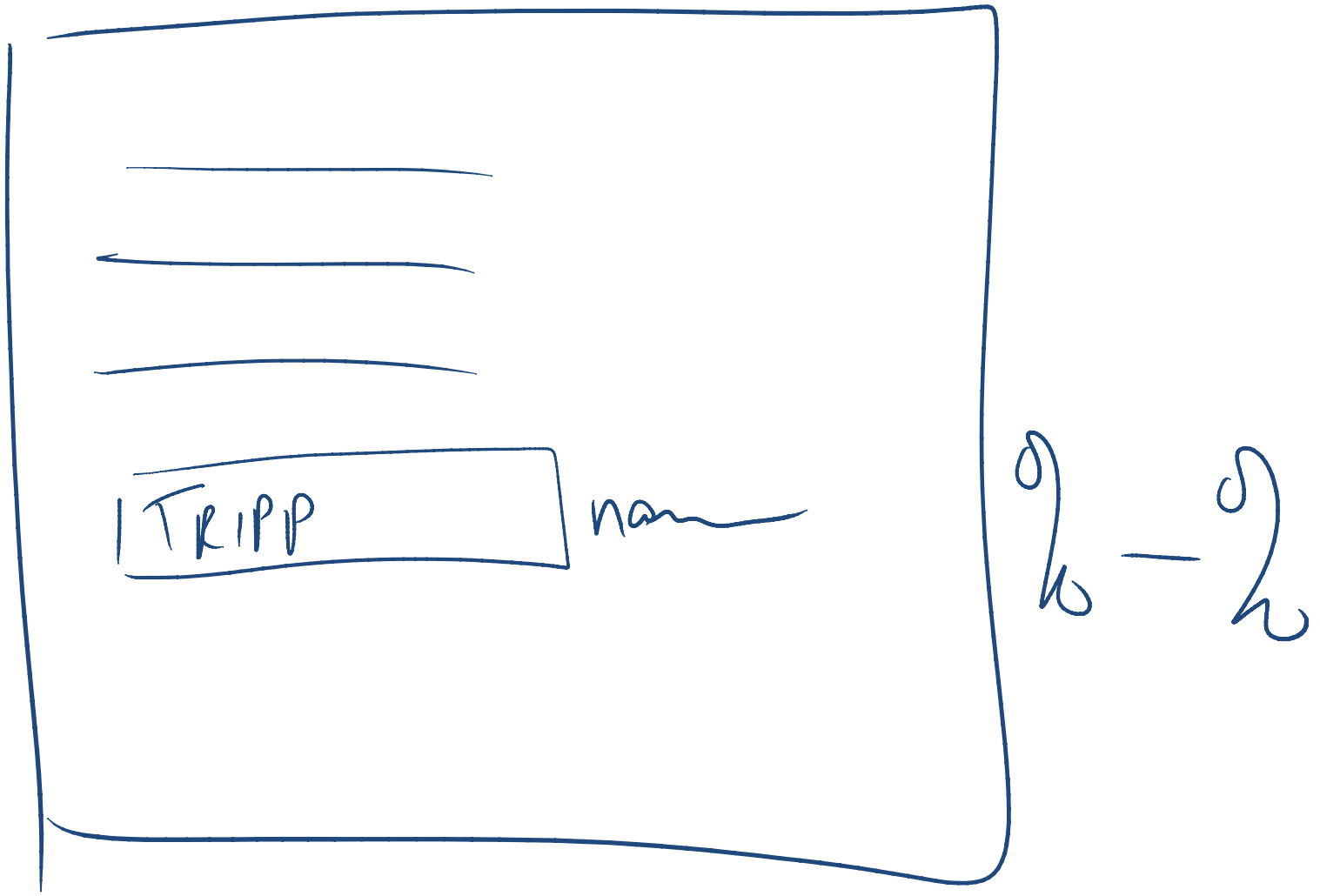
M10s28

This is the diagram to show the estimated costs using an index on OrderDate and an index on ShippedDate. The text on the slide describes the requirements.



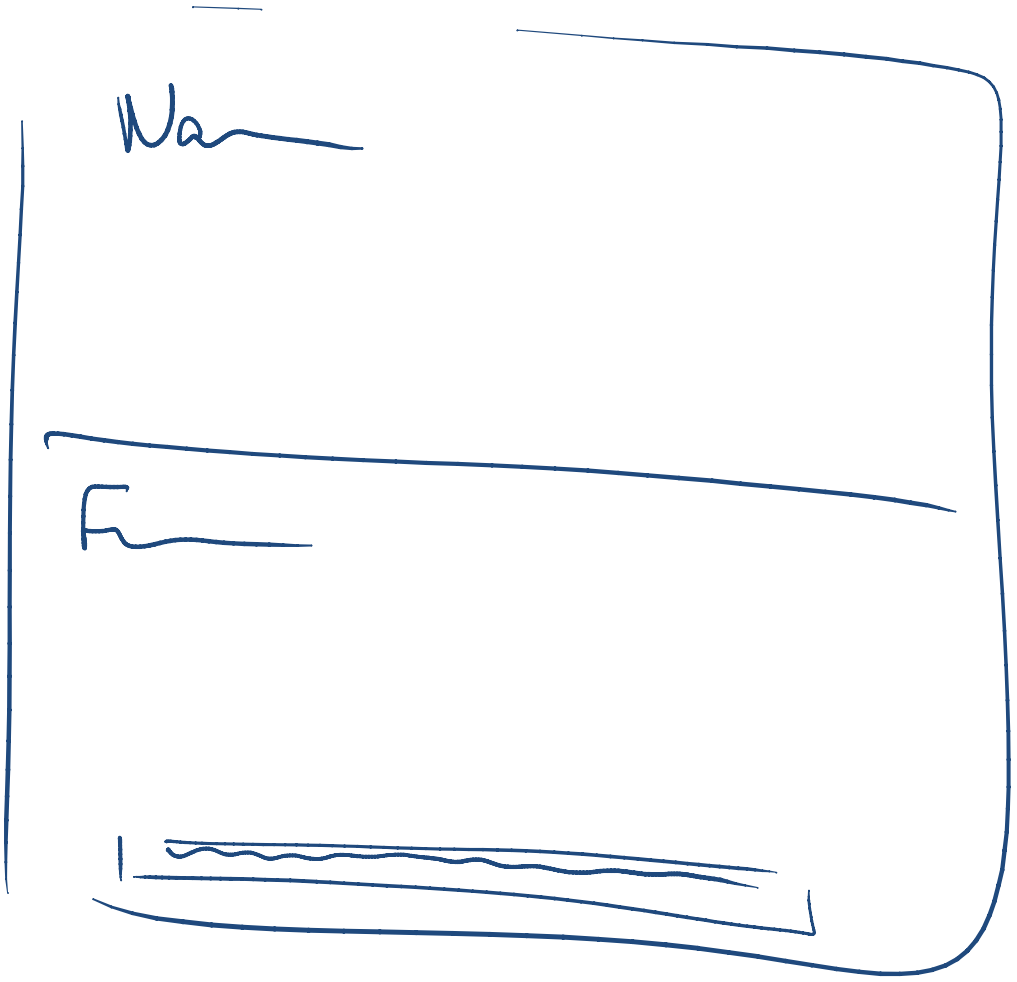
M11s22

An index on a join column (the foreign key) helps the performance for the referential integrity as well as some joins. A narrow index on the join column helps but is often superseded by wider indexes but it's a good start. (hence the phase – phase **1**)



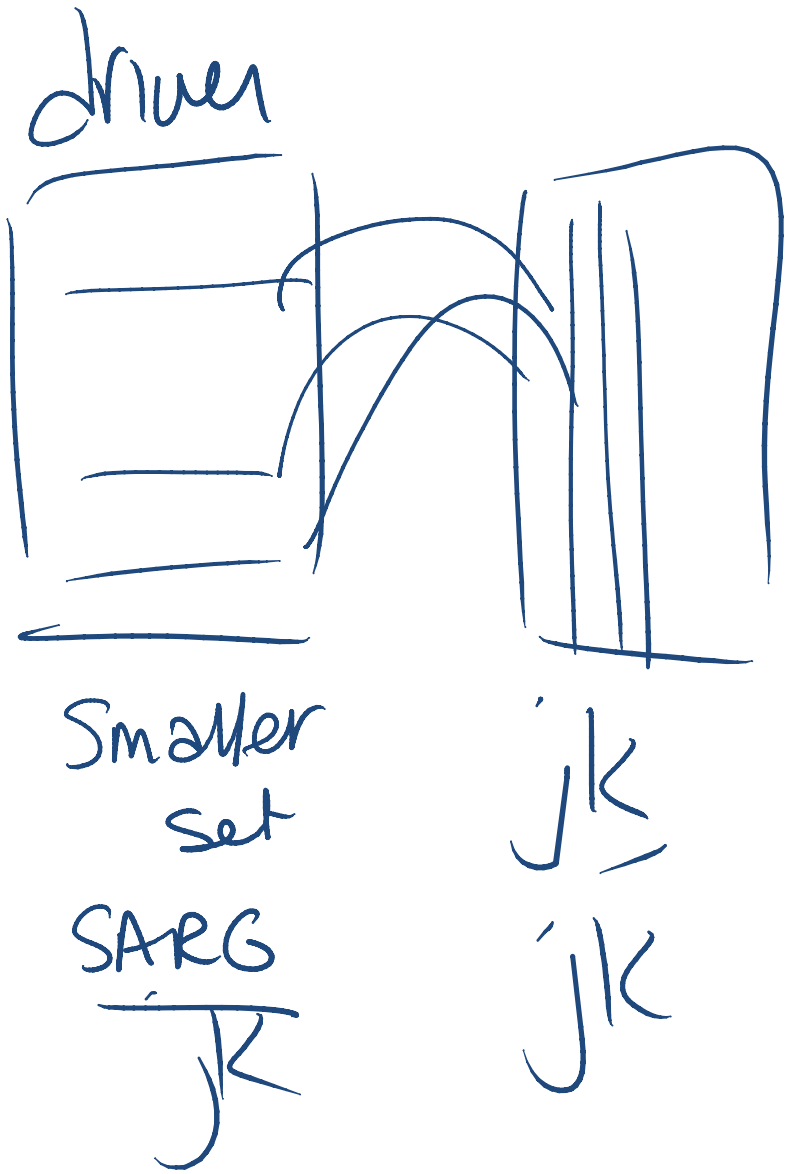
M11 – General Comment

Queries need to be specific and applications should not overly generalize client data by adding % around a value (%value%). Doing so forces a scan… better to do the following:



M11 – General Comment

Better to have two options on the dialog – one where a name is entered specifically.
A second that is a “fuzzy” match.
Add a comment on the bottom saying that these searches might take more time, etc…



M11s20

Joins are made between two tables. To optimize the join – consider the table's position within the join. If it's the "driver" in a loop join – then an index that helps to reduce the set (the SARG) is ideal. To optimize the iterative process in to the inner table – you need an index on the join key.

For merge – SQL Server needs to leverage suitably sorted sets. Two indexes that have the same high-order element (first column) are best here. This is an index on each table that begins with the join key.

For hash – there are two phases (build and probe). To support these phases (with different hash strategies) different combinations of SARG or JK indexes can be used.

Regardless, you need to give SQL Server as many options as possible so that it has all of these options for processing the join.