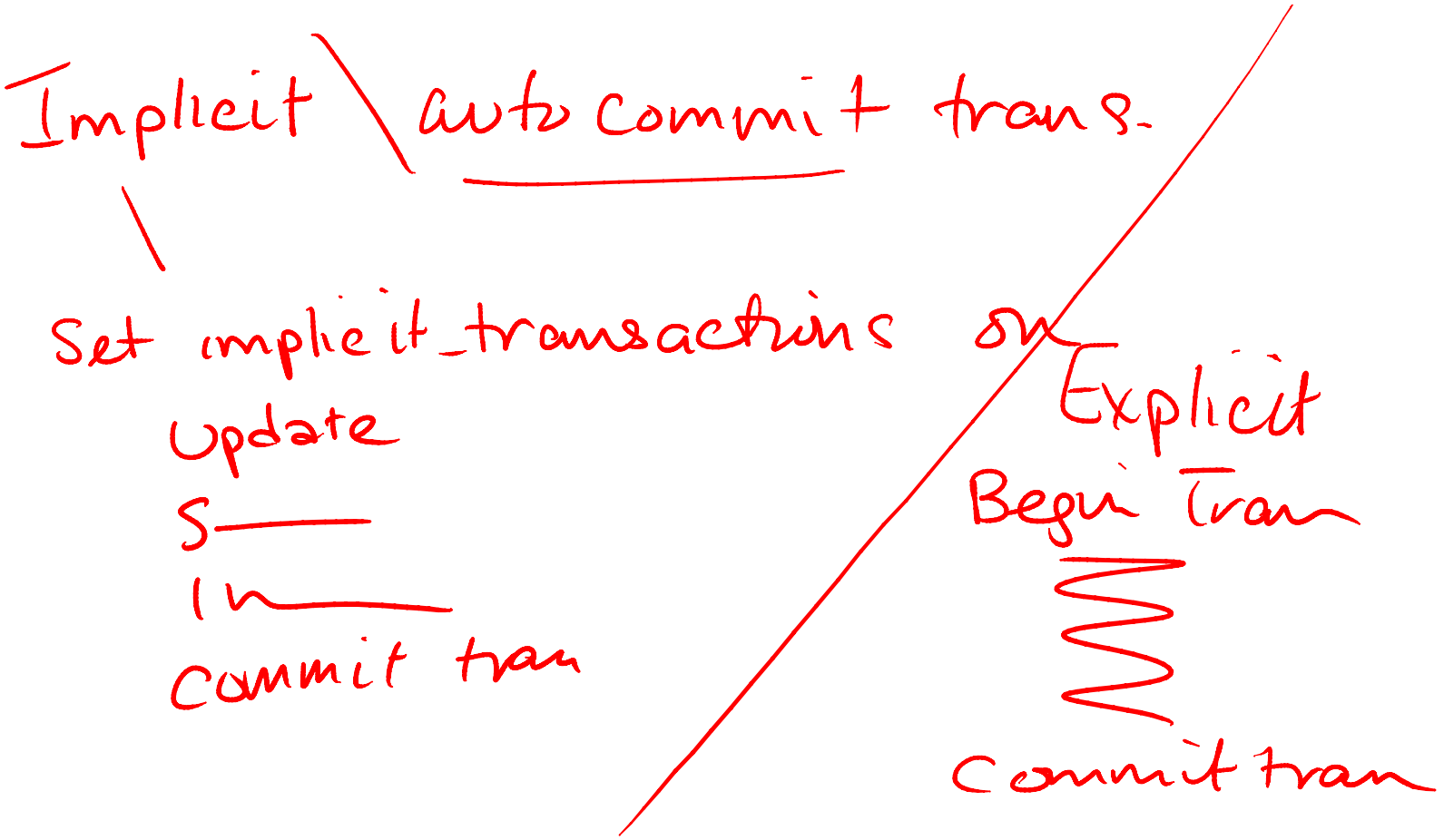

Module 3
Locking & Blocking
SQL
skills

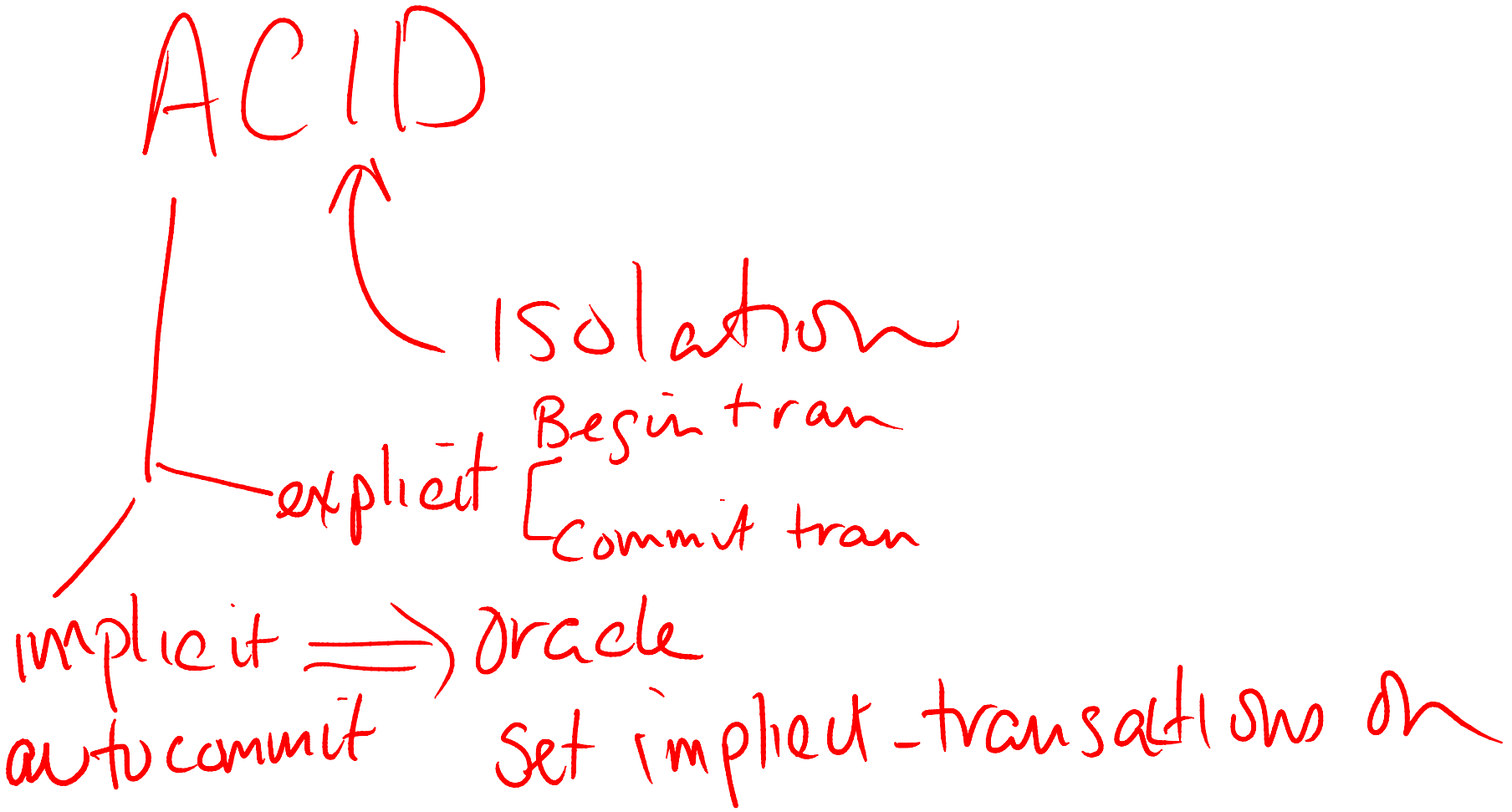


M3s3

While discussing locking basics, we talked about ACID properties.

We also started to discuss transactions:

- *Implicit* is now auto-commit. In most people's minds implicit used to mean that SQL Server implicitly treats your statements as a transaction. However, Oracle's implicit begin a transaction for you and so these should no be confused. To reduce confusion, most people now use the term auto-commit instead.
- Explicit are user-defined transactions where you've explicitly defined the begin/commit.

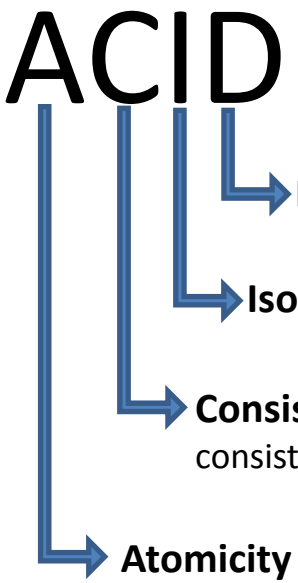


M3s3

While discussing locking basics, we talked about ACID properties.

This picture is described better on the following new picture/slide.

Transaction Properties



Durability – covered by write-ahead logging.

Isolation – this is what you can control the most. What do readers see?

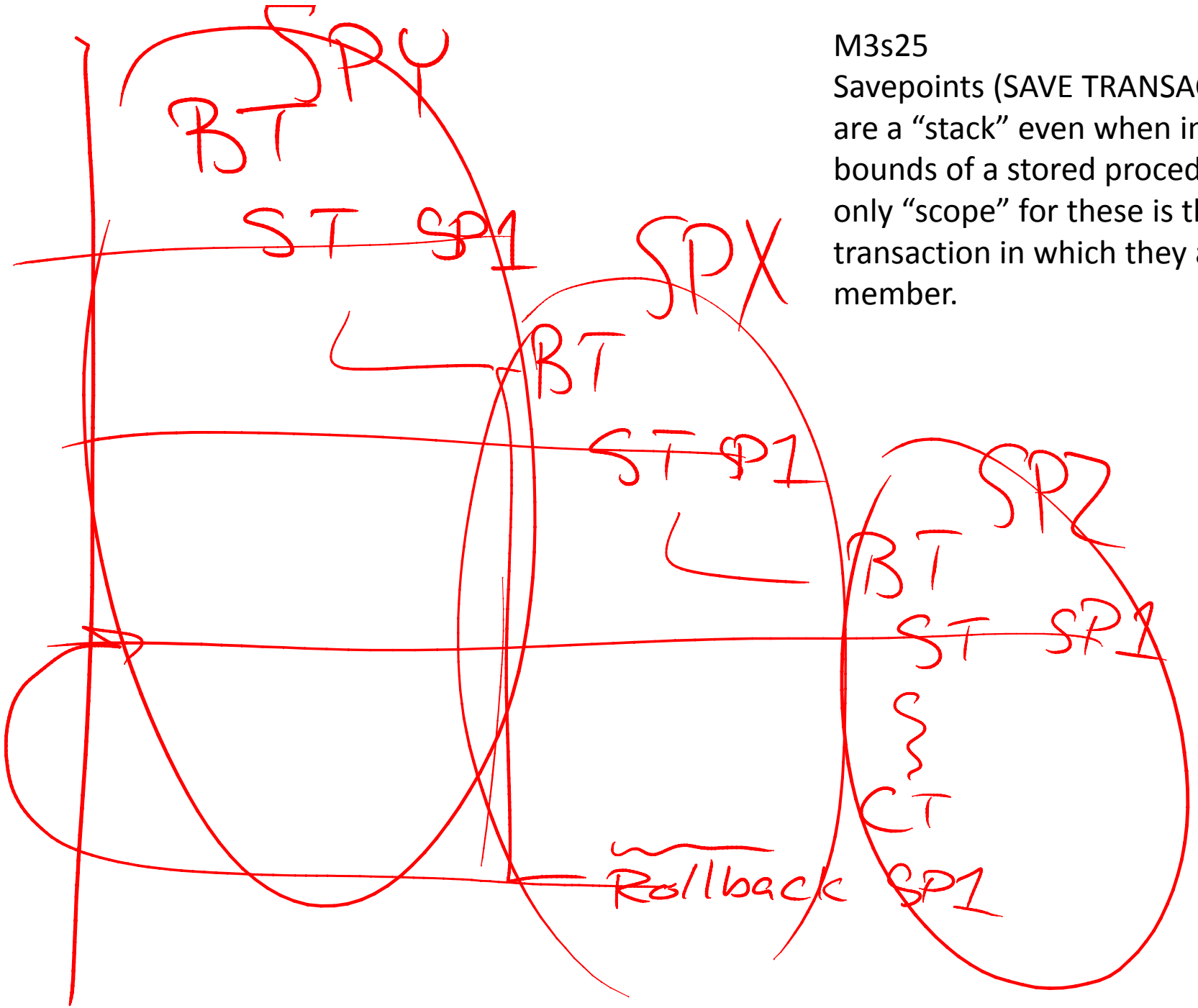
Consistency – everything affected by the transaction (triggers, indexes, constraints) will remain consistent as transactions take data from one state to the next.

Atomicity – A transaction is a single, logical, unit of work. "All or nothing."

- Implicit (or auto-commit) transactions – each statement is a transaction. If a single statement affects multiple rows then ALL rows shall be affected or NONE of those rows will be affected. SQL Server treats EVERY statement as an auto-commit (or implicit) transaction without requiring a BEGIN or COMMIT. If the same behavior as Oracle is desired: SET IMPLCIT_TRANSACTIONS ON. Any statement will begin the transaction; COMMIT TRANSACTION is required to complete.
- Explicit transactions – can widen the boundaries of your transaction by explicitly stating a BEGIN TRANSACTION and a COMMIT TRANSACTION. Be careful though – some statements errors only cause the statement to be rolled back and not the transaction (unless SET XACT_ABORT ON).

  From BOL:

  - When SET XACT_ABORT is ON, if a Transact-SQL statement raises a run-time error, the entire transaction is terminated and rolled back.
  - When SET XACT_ABORT is OFF, in some cases only the Transact-SQL statement that raised the error is rolled back and the transaction continues processing. Depending upon the severity of the error, the entire transaction may be rolled back even when SET XACT_ABORT is OFF. OFF is the default setting.



M3s25
Savepoints (SAVE TRANSACTION) are a “stack” even when in the bounds of a stored procedure. The only “scope” for these is the “one” transaction in which they are a member.

M5s27 – aka Transaction Madness

The key points are:

BEGIN TRANSACTION

- Increments @@trancount

COMMIT TRANSACTION

- Decrements @@trancount

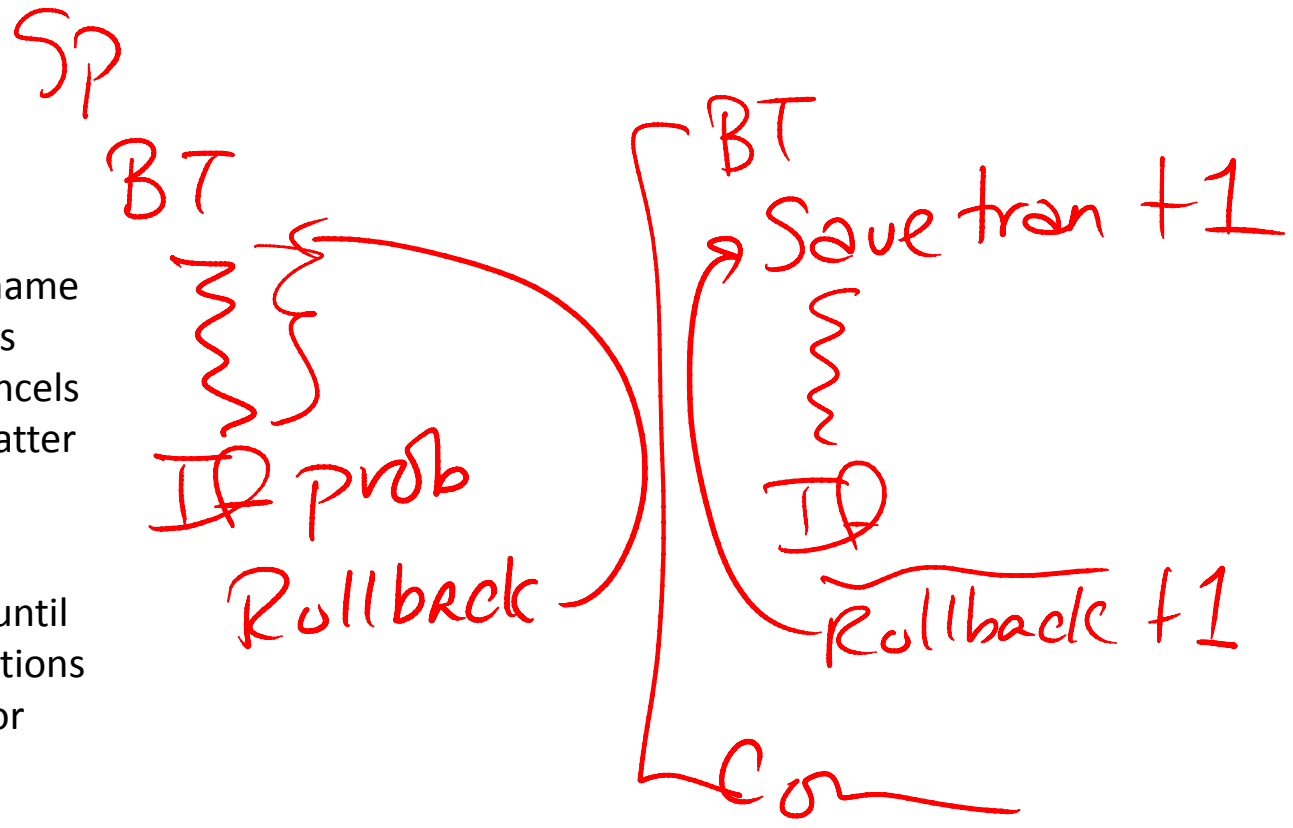
ROLLBACK TRANSACTION

- 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.

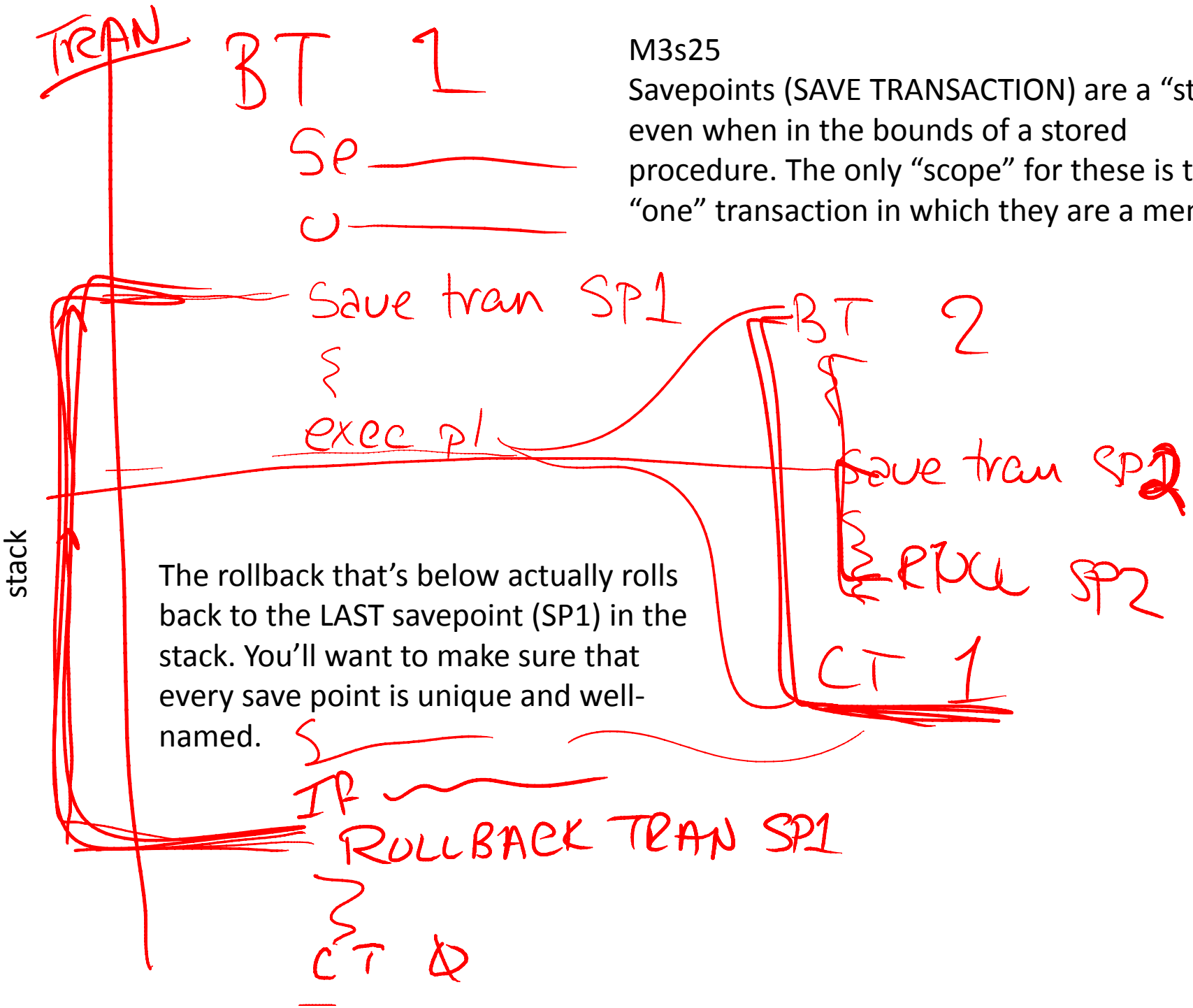
A stored procedure is NOT a transaction in and of itself. You MUST use BEGIN/COMMIT to engage the ACID properties of the statements of a stored procedure.

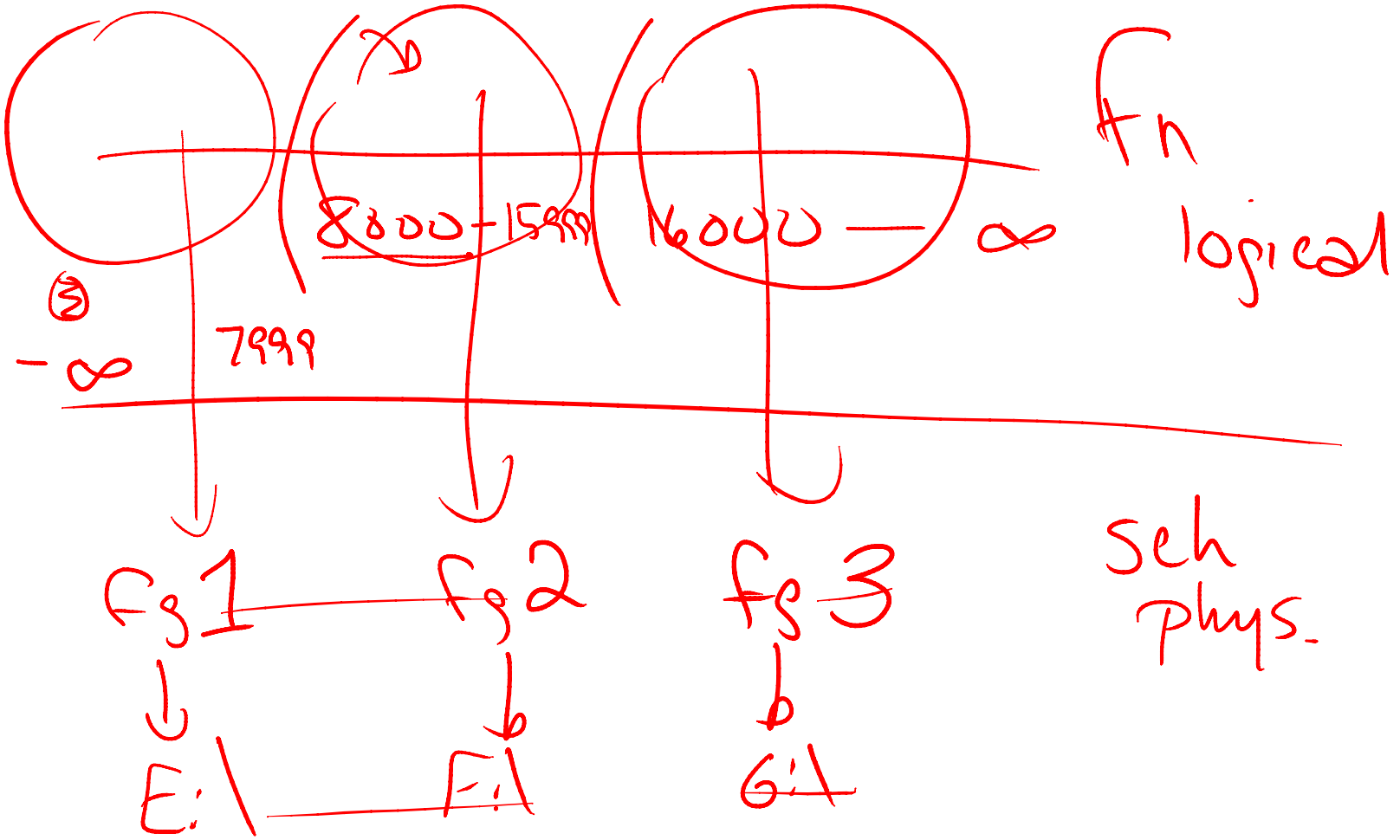


Be sure to go through the demo scripts if you need more insight into transactions!

M3s25

Savepoints (SAVE TRANSACTION) are a “stack” even when in the bounds of a stored procedure. The only “scope” for these is the “one” transaction in which they are a member.

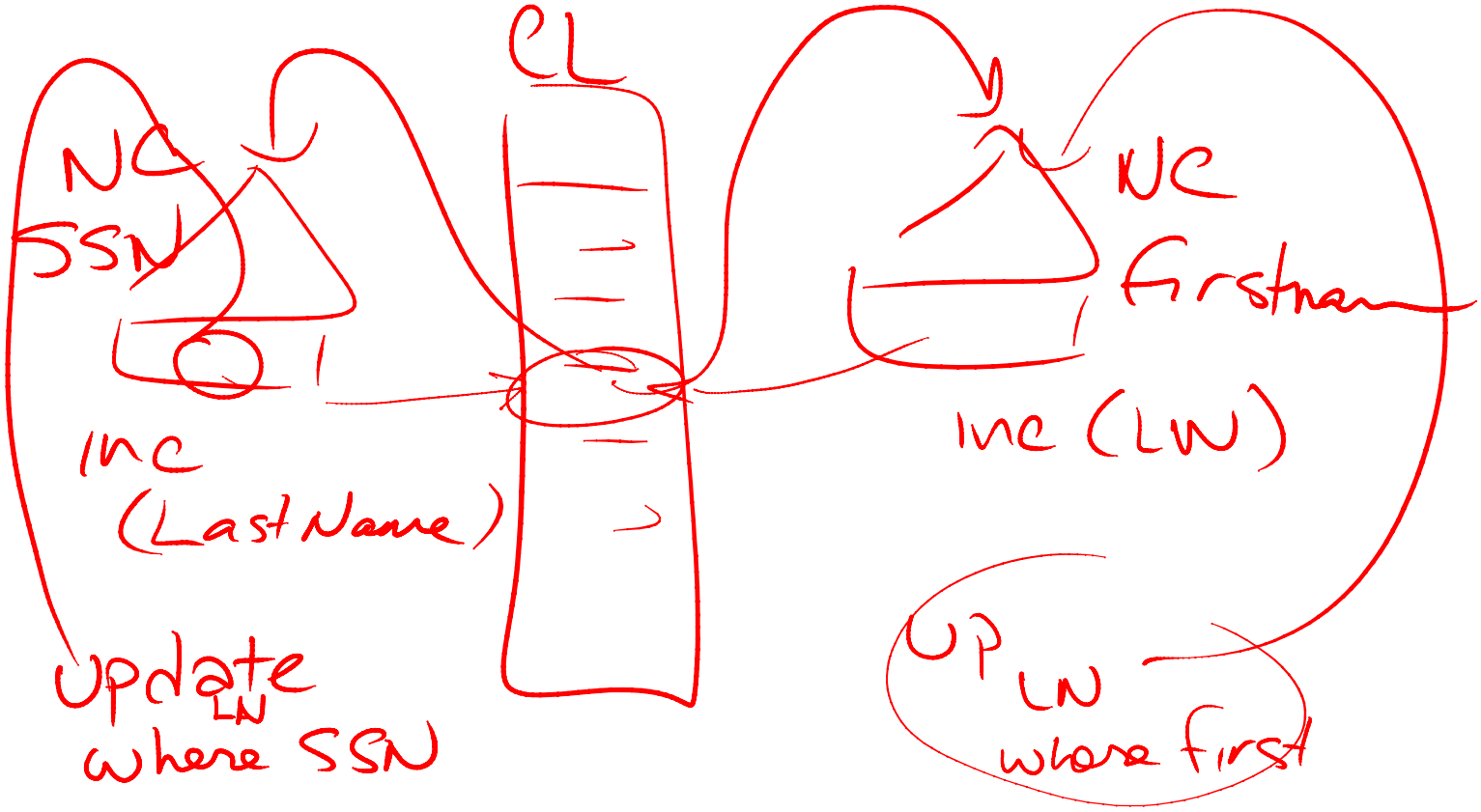




M5s41

This is from the lock escalation demo. This picture shows how the boundaries define the logical placement of data within the 3 partitions. All this pic is describing (briefly) is that the two RIGHT boundary points of 8000 and 16000 define the breakdown as follows:

```
-∞ to 7999 | 8000 to 15999 | 16000 to ∞
 Partition 1 |  Partition 2  | Partition 3
```



M3s53

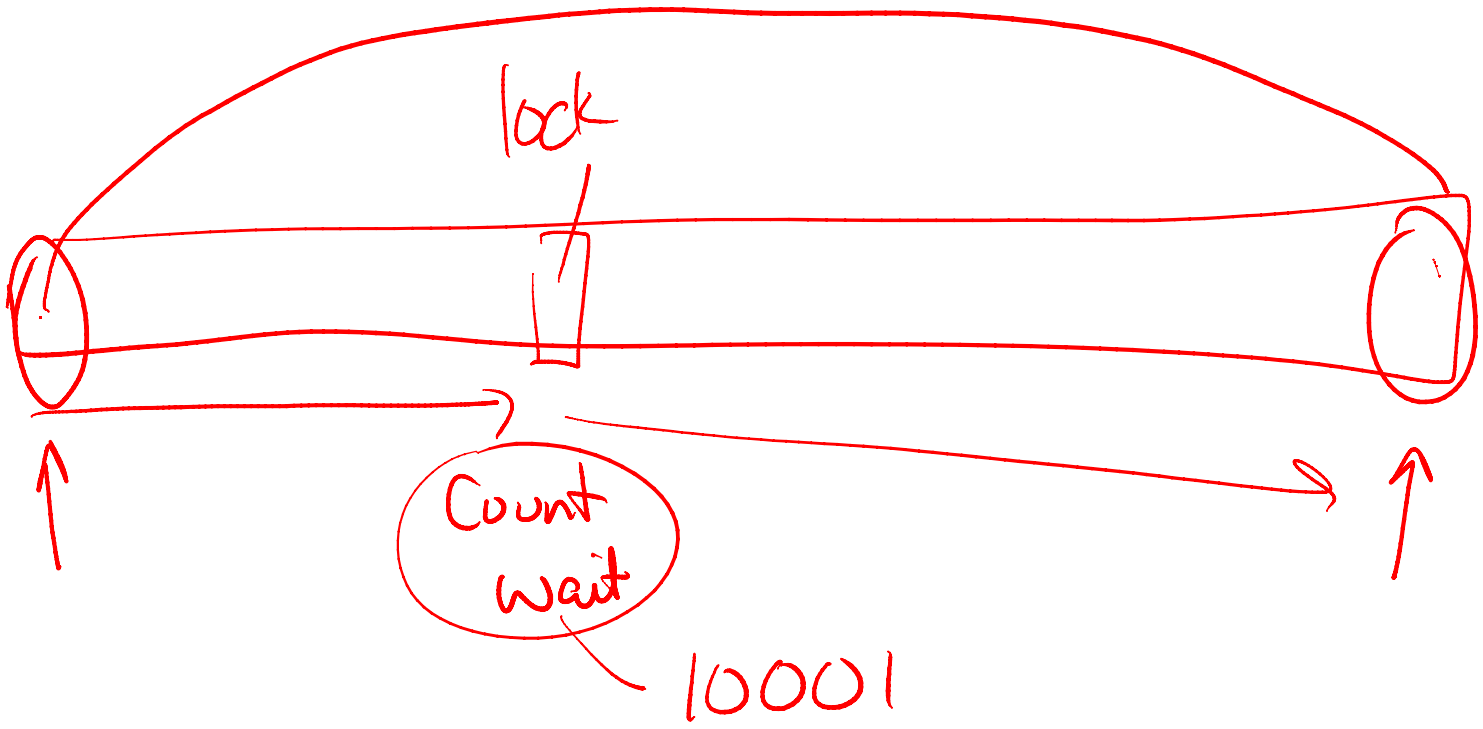
This picture described a deadlock that can occur when multiple indexes exist on a single table and they INCLUDE columns that are being updated.

```
CL = clustered table
Left side = index on SSN INCLUDE (lastname)
Right side = index on firstname INCLUDE (lastname)
```

Imagine an update to firstname that uses WHERE SSN = running at the same time as an update to firstname that uses WHERE Lastname = (let's not debate how bad this latter update is ☺).

These updates are MUCH more likely to create deadlocks because of how locking works within indexes. To reduce the potential here – I often use sp_tableoption.


Module 4
Snapshot Isolation
SQL
skills



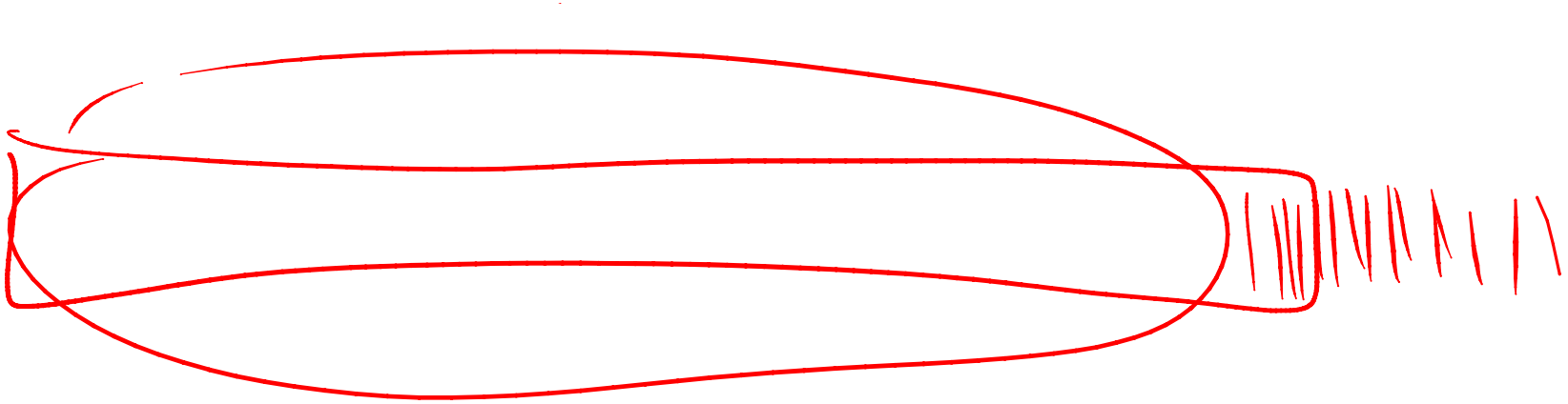
Demo: Non-repeatable reads (Inconsistent Analysis)

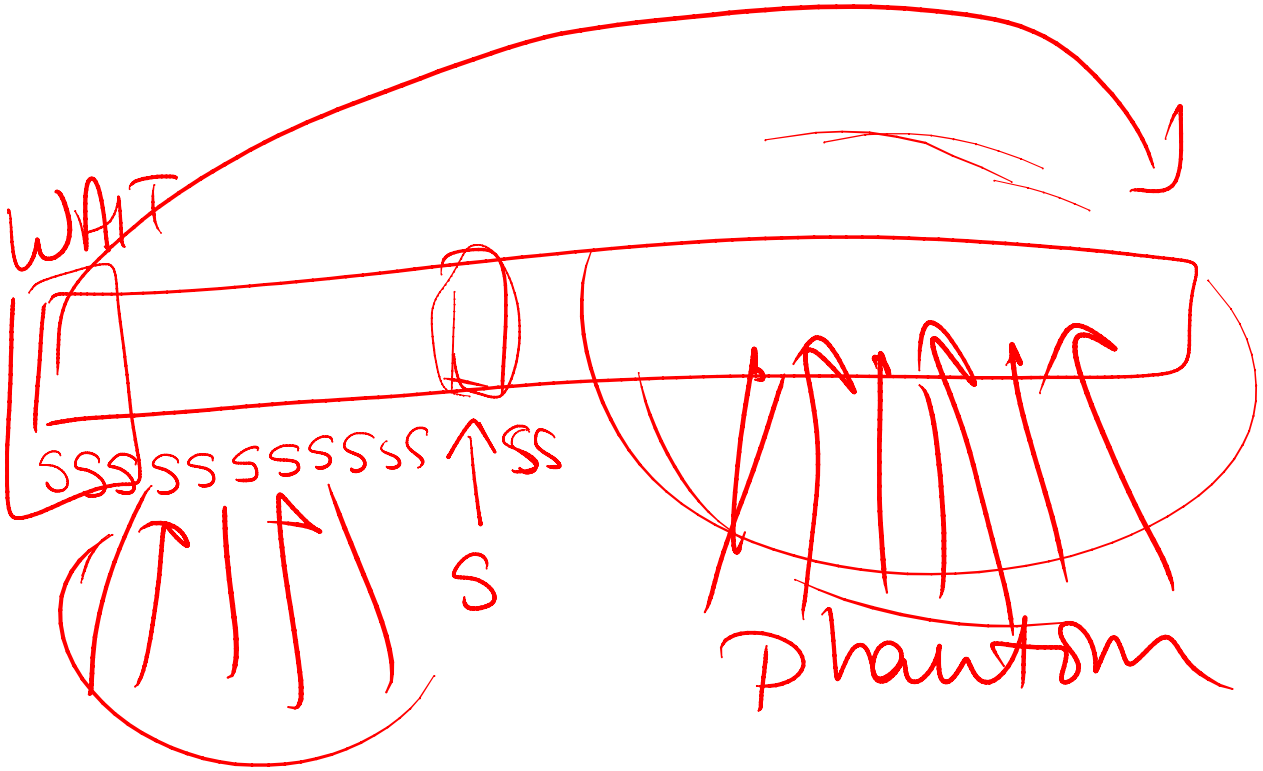
This is showing how an update to a CL key value that causes record relocation (after another query has already read [and released the lock on] the row) allows the reader to read the row TWICE (non-repeatedly).

This is from the demo on non-repeatable reads and shows what I called the “anderson-zembrowsky” problem. ☺

Discussion: Phantoms (Inconsistent Analysis)

What is correct? Are we talking about row-consistency or statement-level consistency? The standards only define the state of the row at the time that the row is accessed. There is NOTHING that ties together the state of those rows at the time the query is accessed (unless you use versioning). So, here I was discussing the fact that NEW rows that are coming into the set will be visible in all isolation levels (except serializable). I always think of this scenario like a dog chasing its tail. ☺



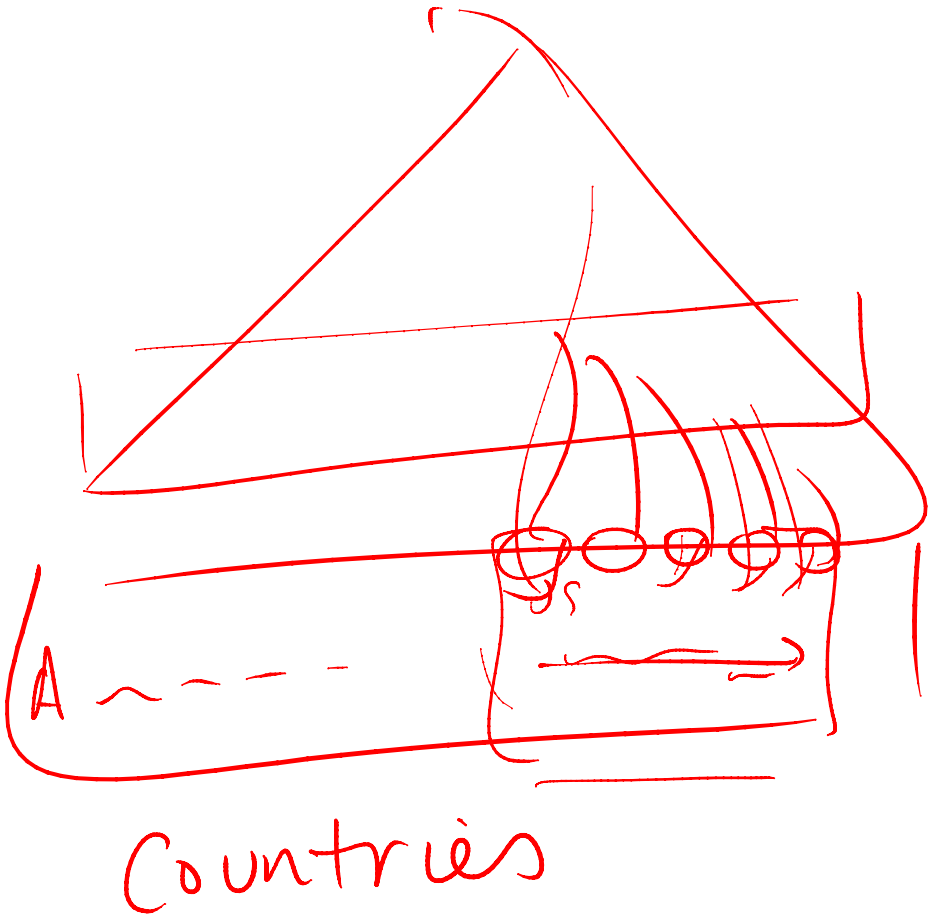


Discussion: How does repeatable read prevent the Anderson/Zembrowsky problem?

As rows are read, the shared locks used to read them are left behind to protect the rows. The rows cannot be read again in any other state because the shared locks will prevent other users from modifying the rows.

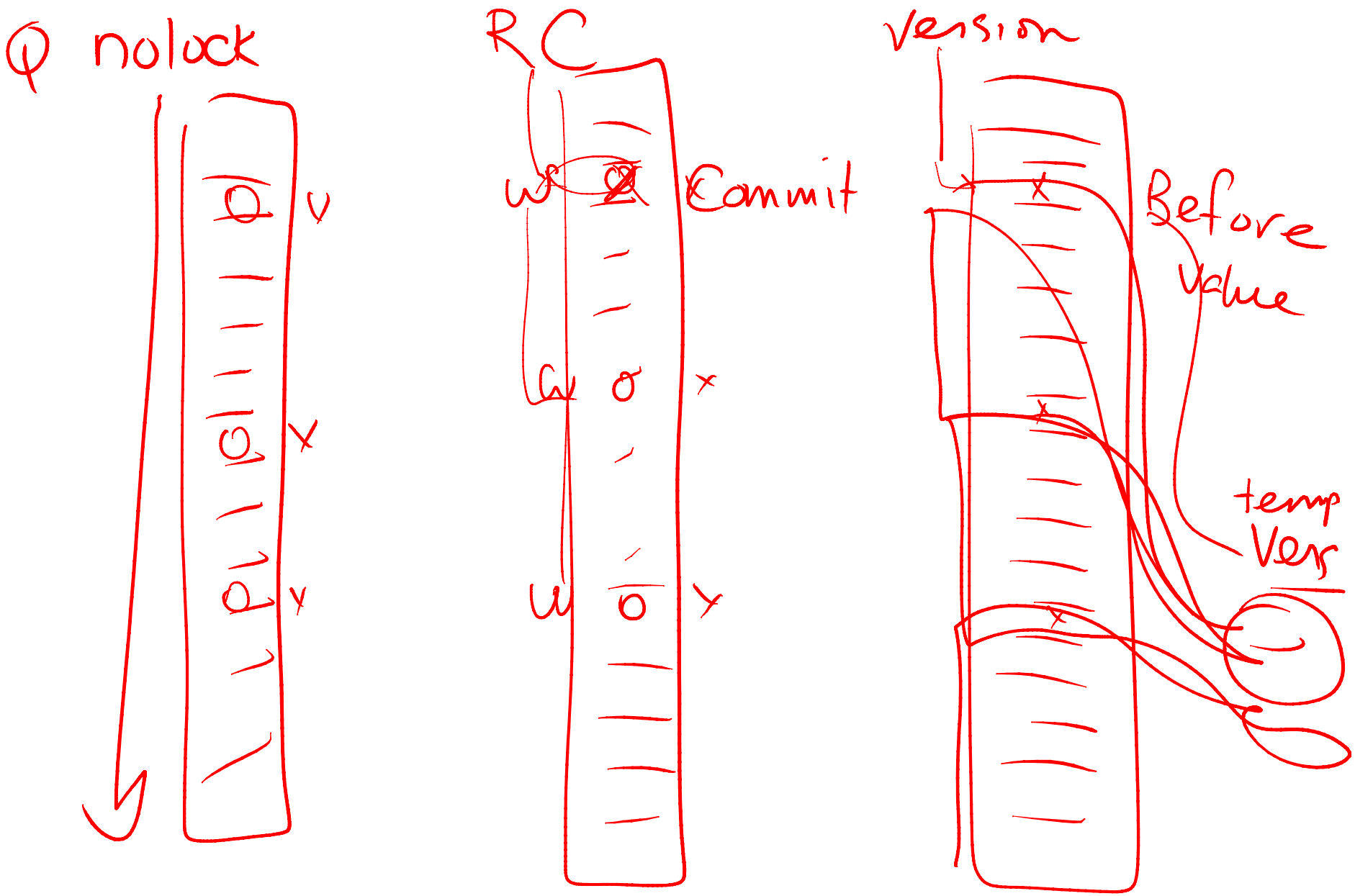
M4s11 – Serializable transactions

This picture describes key range locking in an index. When a query runs, the isolation level dictates the state of the data (even in the bounds of a transaction) that can be seen. In a serializable transaction the data is isolated at the time the statement runs. As a result, data must be locked to prevent anomalies. To prevent new rows from coming into the set, SQL Server will lock the "range" of rows affected by the query. If a good index exists then SQL Server can lock within the index (it's in the first intermediate level – the level above the leaf level). If a good index does not exist then SQL Server must do table-level locking. This example was showing key-range locking in an index for the "country" set. In the end, we might end up locking more data than just that country but only a small amount of data that surrounds the value(s) of interest.



M4s11 – Serializable transactions

This picture described key range locking in an index. When a query runs the isolation level dictates the state of the data (even in the bounds of a transaction) that we should see. In a serializable transaction the data is isolated at the time the statement runs. As a result, data must be locked at that time. To prevent new rows from coming into the set, SQL Server will lock the "range" of rows affected by the query. If a good index exists then SQL Server can lock within the index (it's in the first intermediate level – the level above the leaf level). If a good index does not exist then SQL Server must do table-level locking.


Q nolock
RC
Commit
version
Before
Value
temp
Ver

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 (using locking)

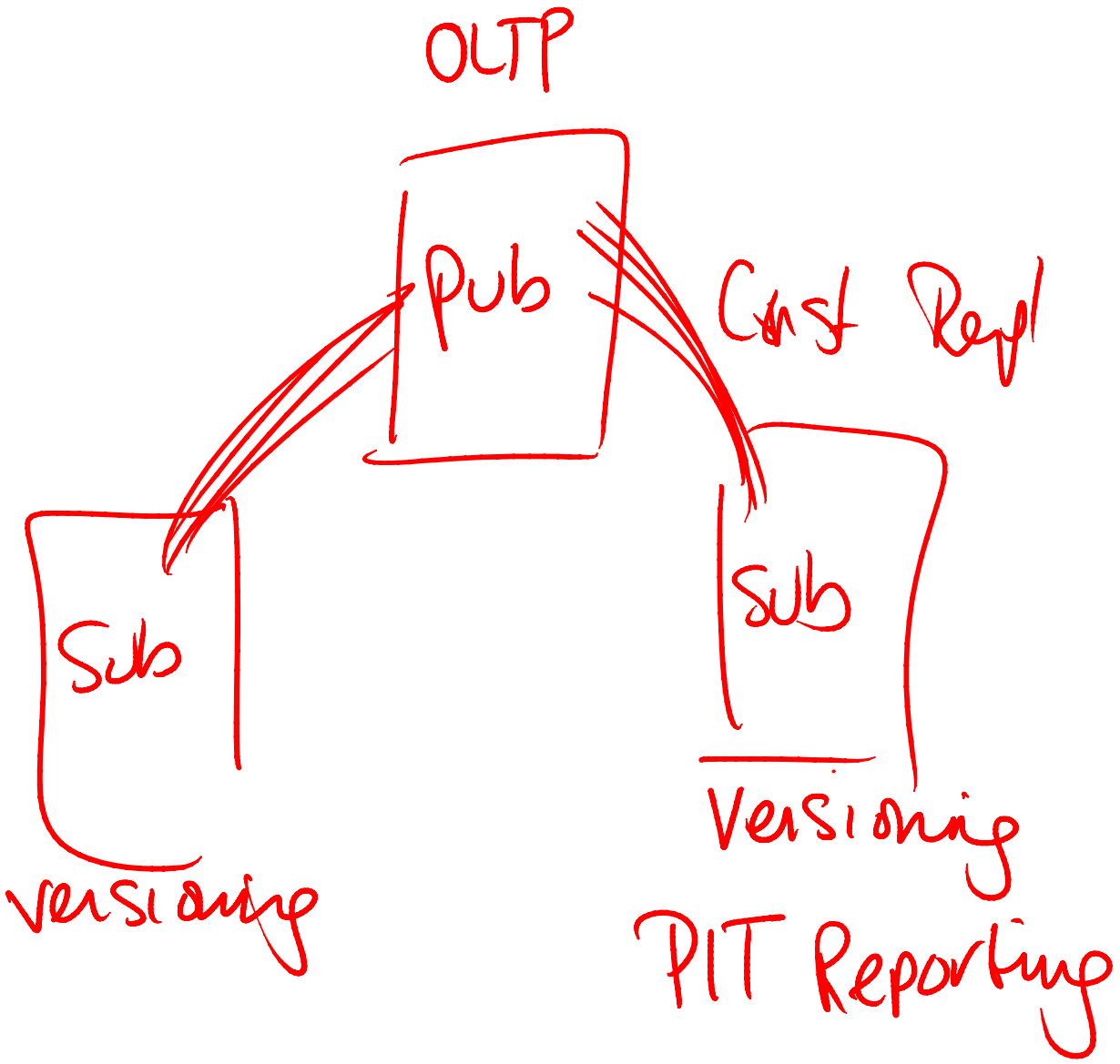
The key things to remember from the picture is that (1) has "hiccups" along the way as they encounter rows that are locked. They read… wait… read… wait. This is *partially* what slows down a statement. However, it's worse that this. It's possible that AFTER a row is read, it will appear again (if the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

(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)

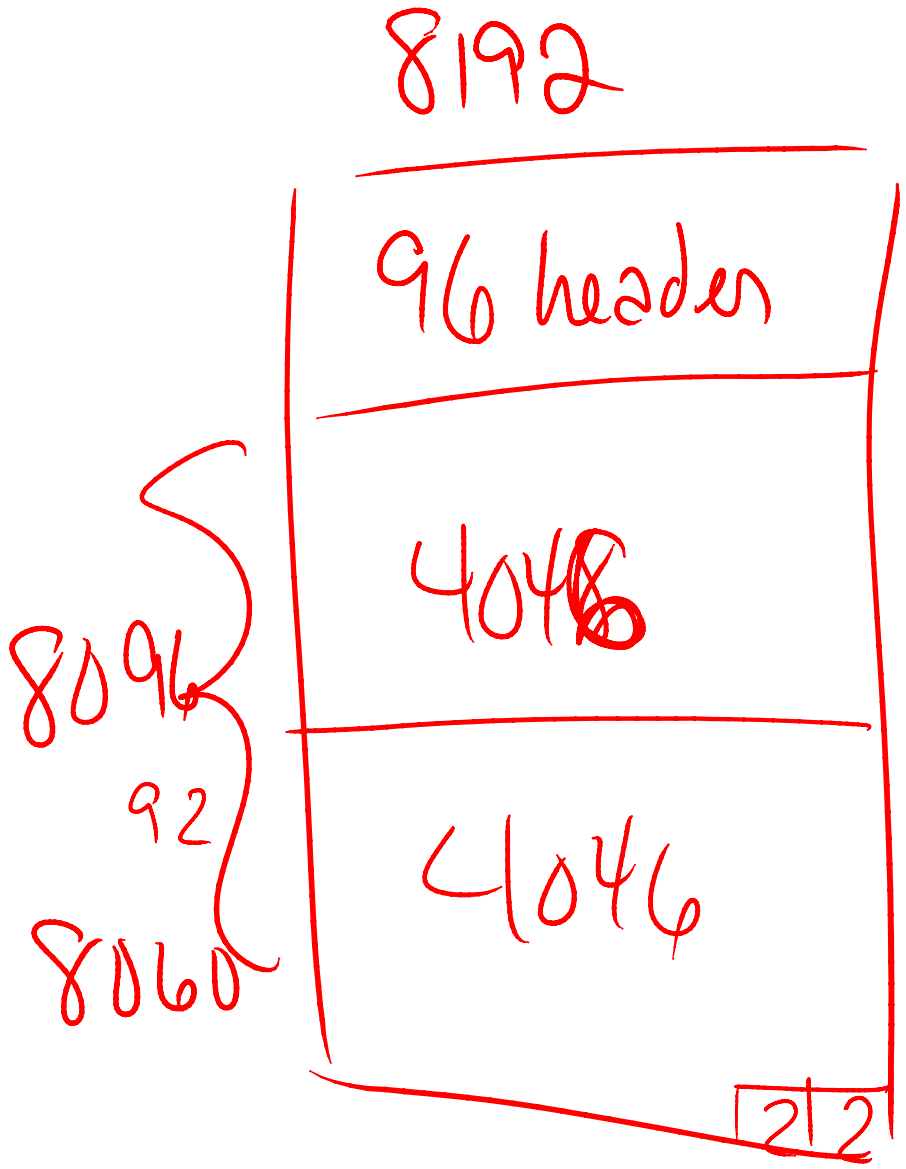
When a versioned query runs the reader will not stop but will have to go to the version store for any row that's in flight. This is a tiny bit slower but guarantees consistency of the ENTIRE read to the point in time when the statement started. This gives you better concurrency as well as a definable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



An ideal use for Snapshot Isolation

The idea is that you don't want to impact your production OLTP environment with READERs or with versioning so instead, you replicate to subscribers and use read committed snapshot there!


Module 6
Table Design Strategies
SQL
skills



M6s4

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:

8192 bytes (page size)

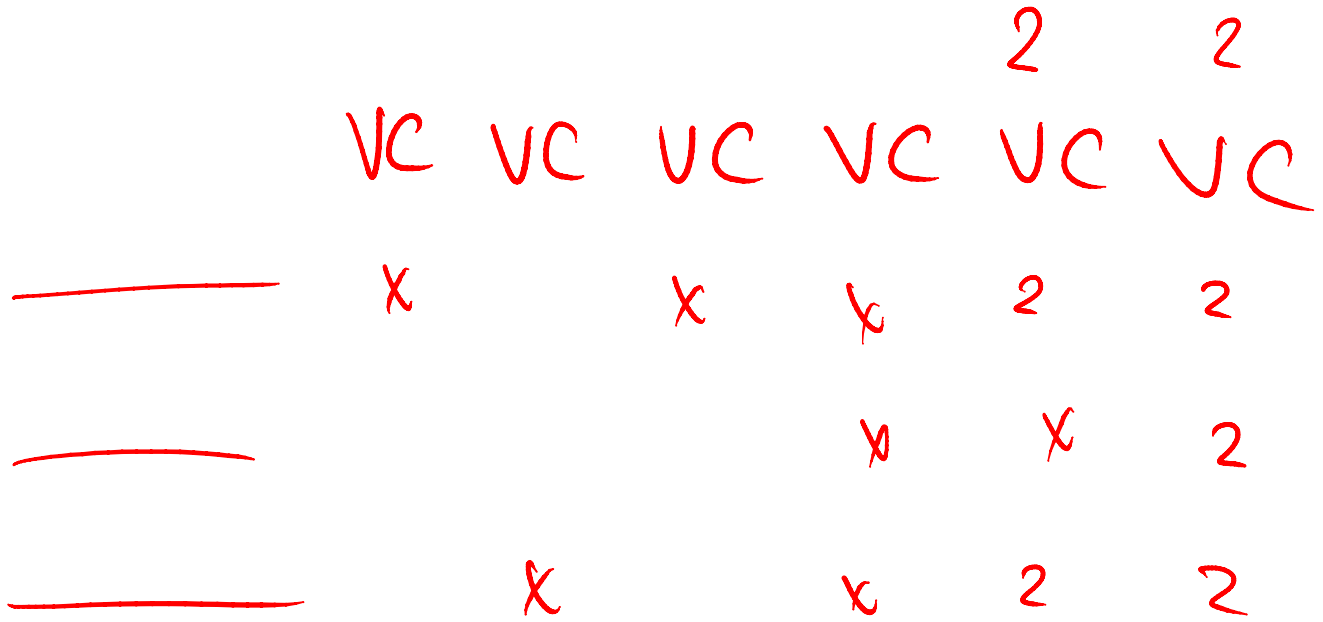
- 96 bytes in the header

= 8096 as the maximum amount of space for data

A row must have a header of 4 bytes

= 8092 and the SQL team took "32 bytes" for future growth.

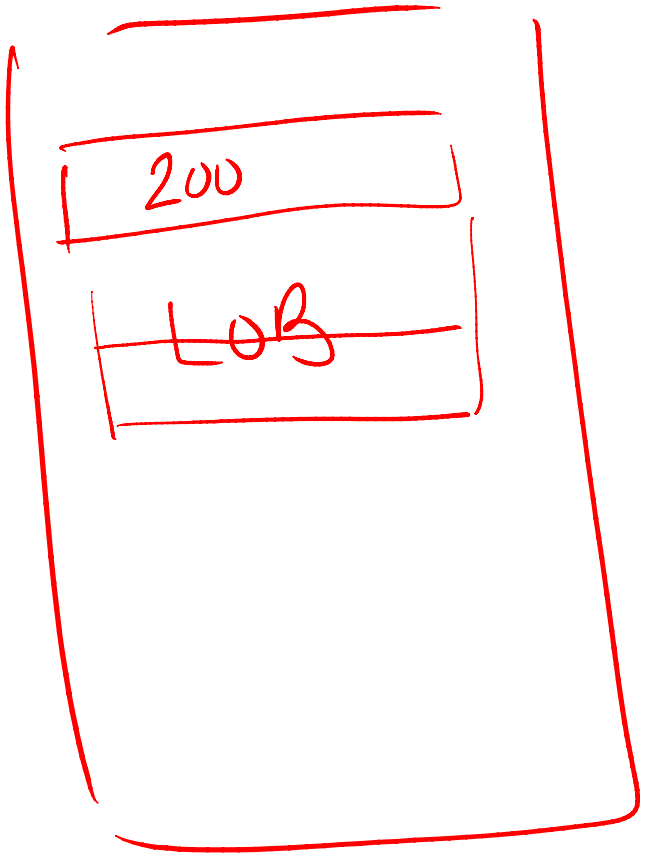
Some of which they're already using in SQL Server 2005 and higher. For row versioning, each versioned row requires a 14 byte offset. If they had allowed a row to be 8096 – where would they have put the offset?



Column order does not matter… it depends!

OK, this diagram should remind you that leaving the columns that are most likely to be null – at the end of the row definition *COULD* save some space.

Blog post: Column order doesn't matter... generally, but - IT DEPENDS!
http://www.sqlskills.com/BLOGS/KIMBERLY/post/Column-order-doesnt-matter-generally-but-IT-DEPENDS!.aspx



LOB

100 - 200 bytes

majority

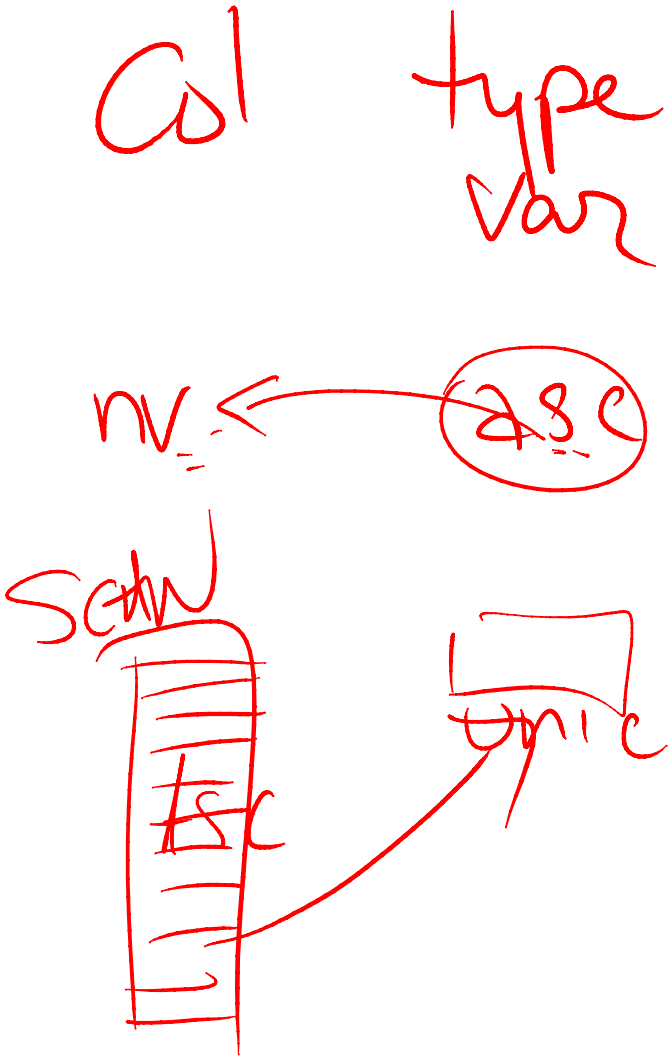
< 2000 bytes

M6s7

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!



M6s8 Implicit Conversions

When a predicate is being evaluated there are two sides to the equation:
The column expression & The variable expression

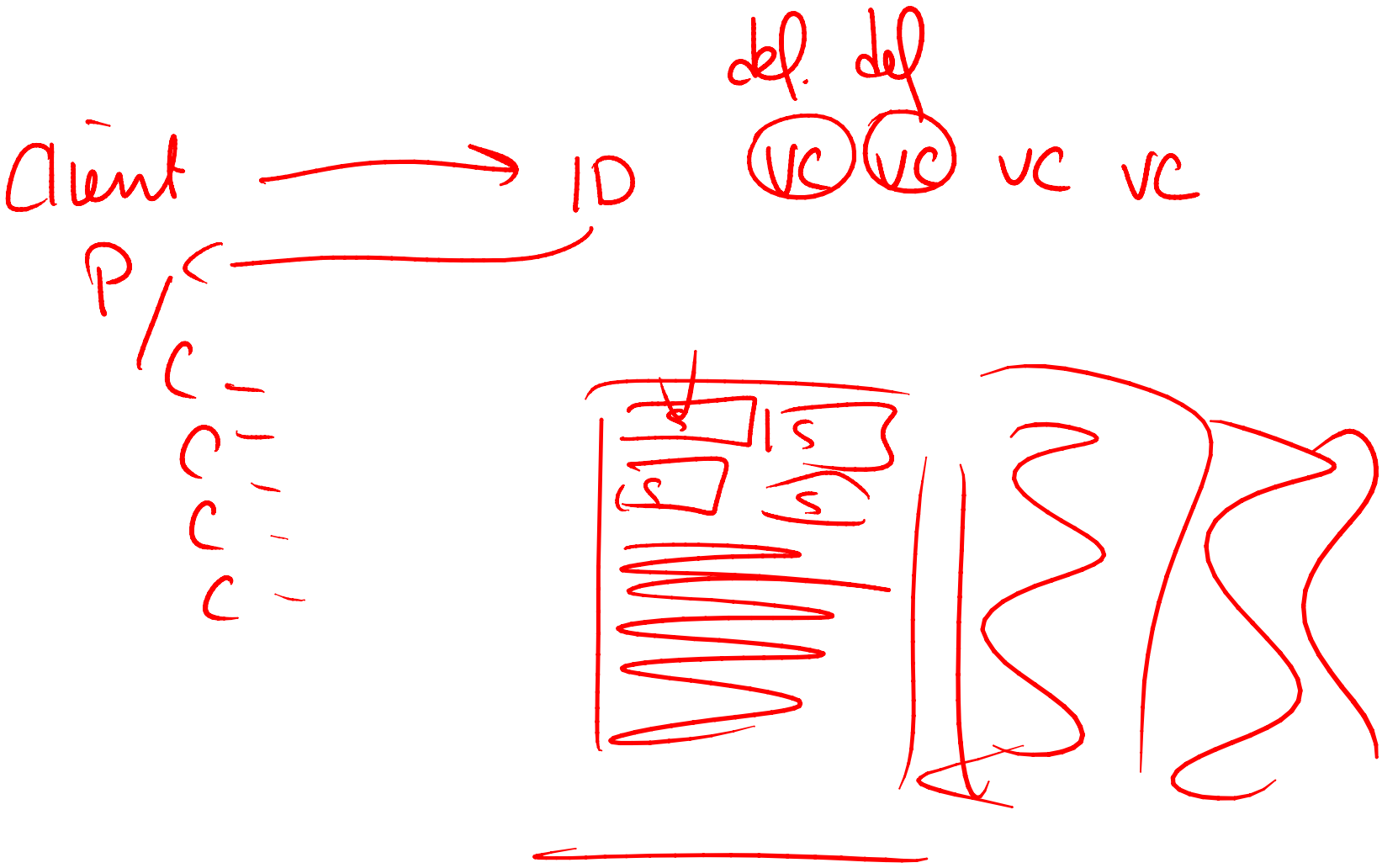
When one of the expressions has a higher data type AND they're implicitly compatible then SQL Server will need to bring the other expression UP to the same type.

If the variable expression is the LOWER type then it's easy, only that variable has to be converted.

The problem starts when the column expression is the lower type – then the column has to be converted. This results in a scan of that column (could be an index scan but if a good index doesn't exist then it might be a table scan).

You can see implicit conversions in cache – check out Jonathan Kehayias' blog post on this:
Finding Implicit Column Conversions in the Plan Cache
http://sqlblog.com/blogs/jonathan_kehayias/archive/2010/01/08/finding-implicit-column-conversions-in-the-plan-cache.aspx

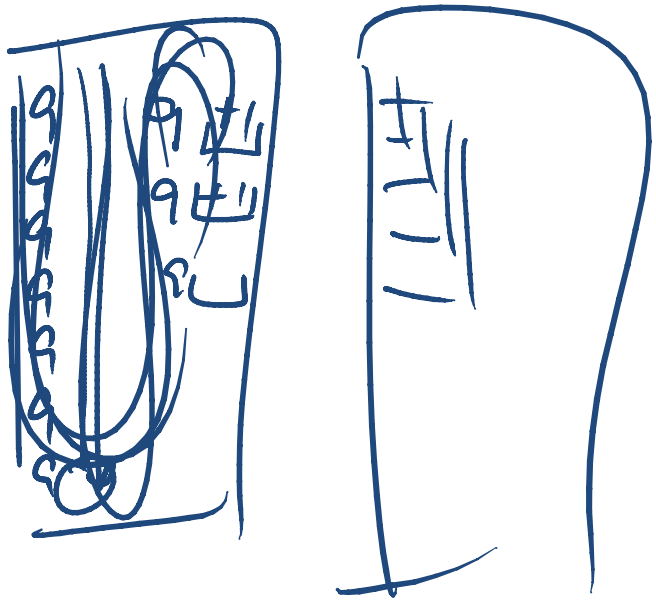
Placeholder Rows




Module 7
Index Internals
SQL
skills

CL DATE

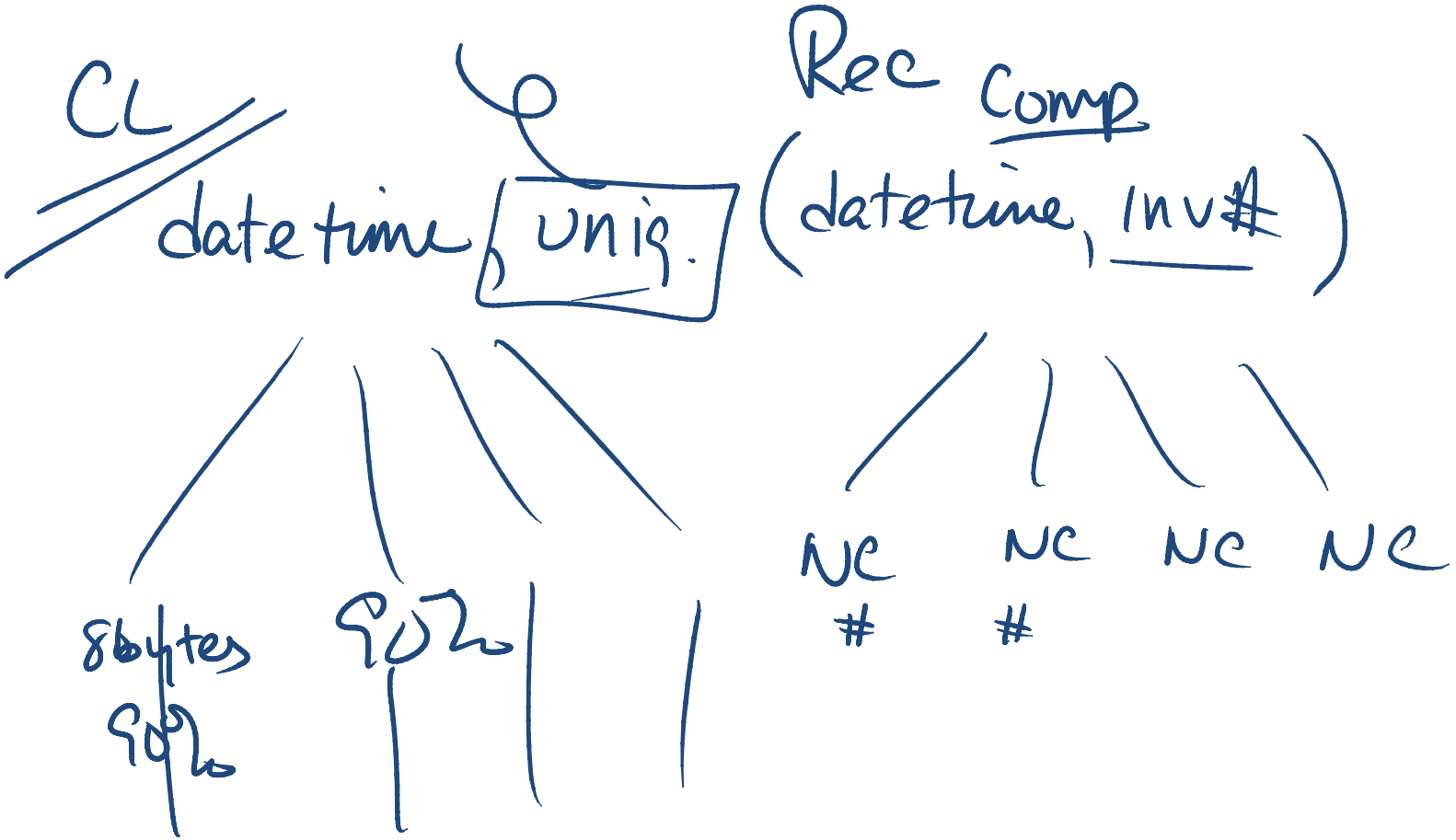
Int 2.147 bill



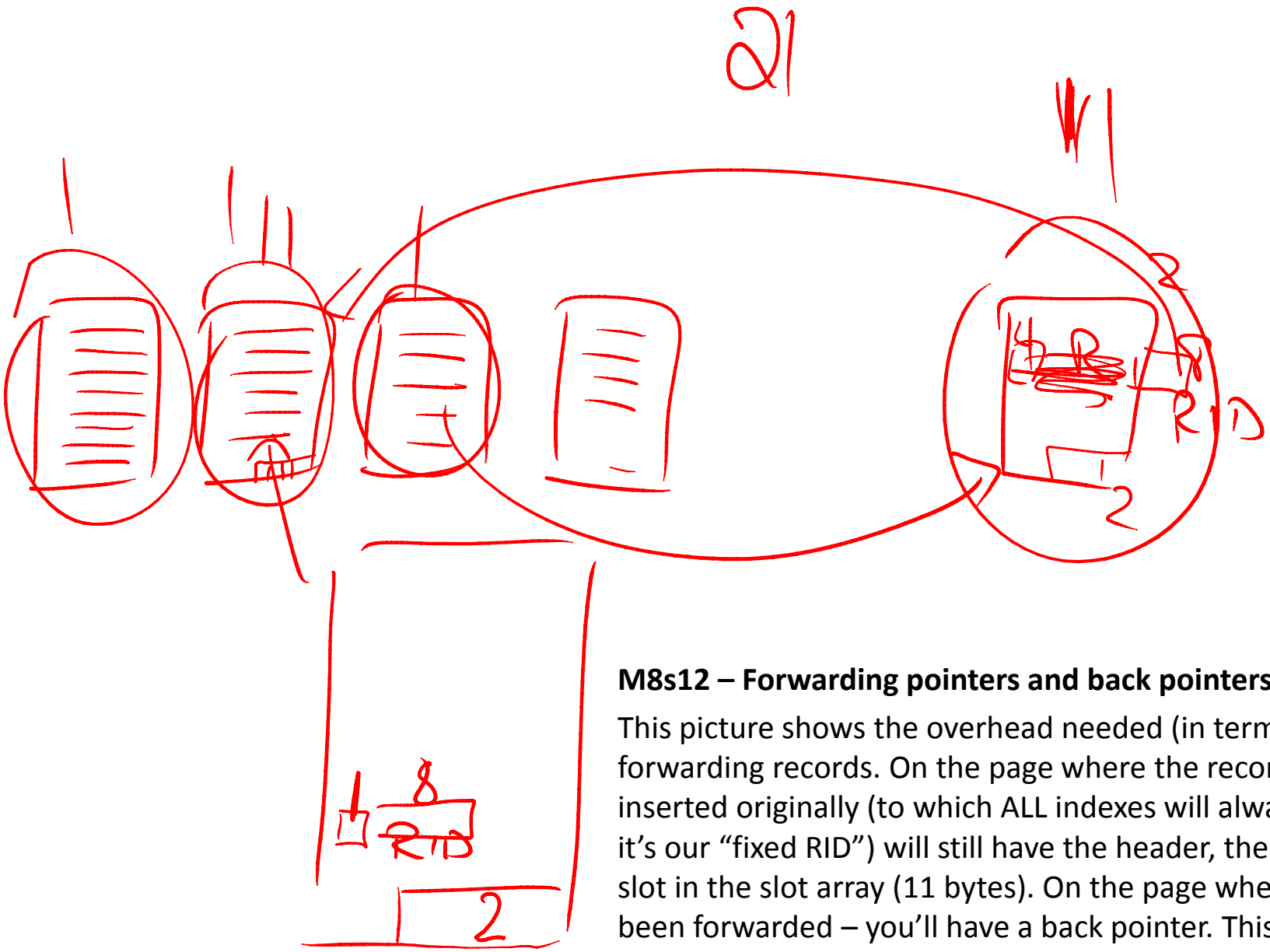
Error 666

A clustering key should be unique by definition. However, if the CL key is NOT unique then SQL Server will uniquify it. Each duplicate group can support 2^{31}-1 duplicates. What happens if you put a CL index on Lastname and then have 2 billion rows with a value of Smith… error 666.

Error 666: The maximum system-generated unique value for a duplicate group was exceeded for index with partition ID %I64d. Dropping and re-creating the index may resolve this; otherwise, use another clustering key.



Another negative associated with the uniquifier value is finding it. Remember, each duplicate has a uniquifier that's tied only to it. On insert, SQL Server will have to scan the last page to find the maximum *current* value for the uniquifier for that "group." If you have something like a DATE (not datetime/datetime2) then you're likely to have an impact from the generation of the uniquifier. It's often better to use something like an invoice number to uniquify the rows. And, secondarily this should also save time. Because the uniquifier has to live in the variable block it requires at least 6 bytes and possibly 8. These additional bytes are in the data rows and in ALL indexes. If 90% of your data rows include the uniquifier AND many of your nonclustered indexes include the invoice number then you're going to be require less storage with an int and probably even still with a bigint given that the table won't need the uniquifier and many indexes [probably] already have the invoice number.

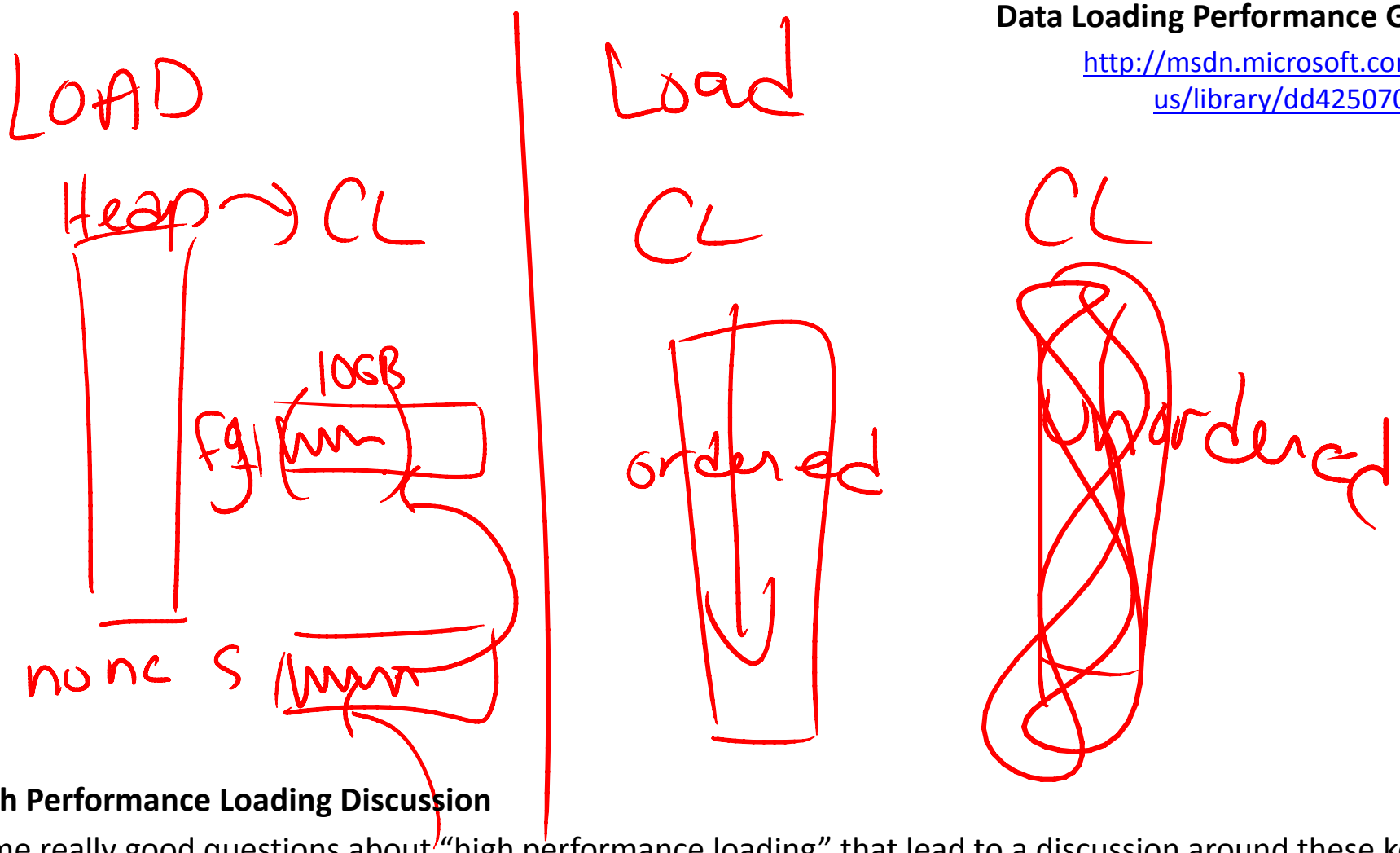


M8s12 – Forwarding pointers and back pointers (HEAPs ONLY)

This picture shows the overhead needed (in terms of bytes) for forwarding records. On the page where the record was inserted originally (to which ALL indexes will always point – as it's our "fixed RID") will still have the header, the RID and the slot in the slot array (11 bytes). On the page where the row has been forwarded – you'll have a back pointer. This back pointer is 8 bytes but it lives in the variable block in the row so it requires an additional 2 bytes. Total space wasted is 21 bytes.

Data Loading Performance Guide

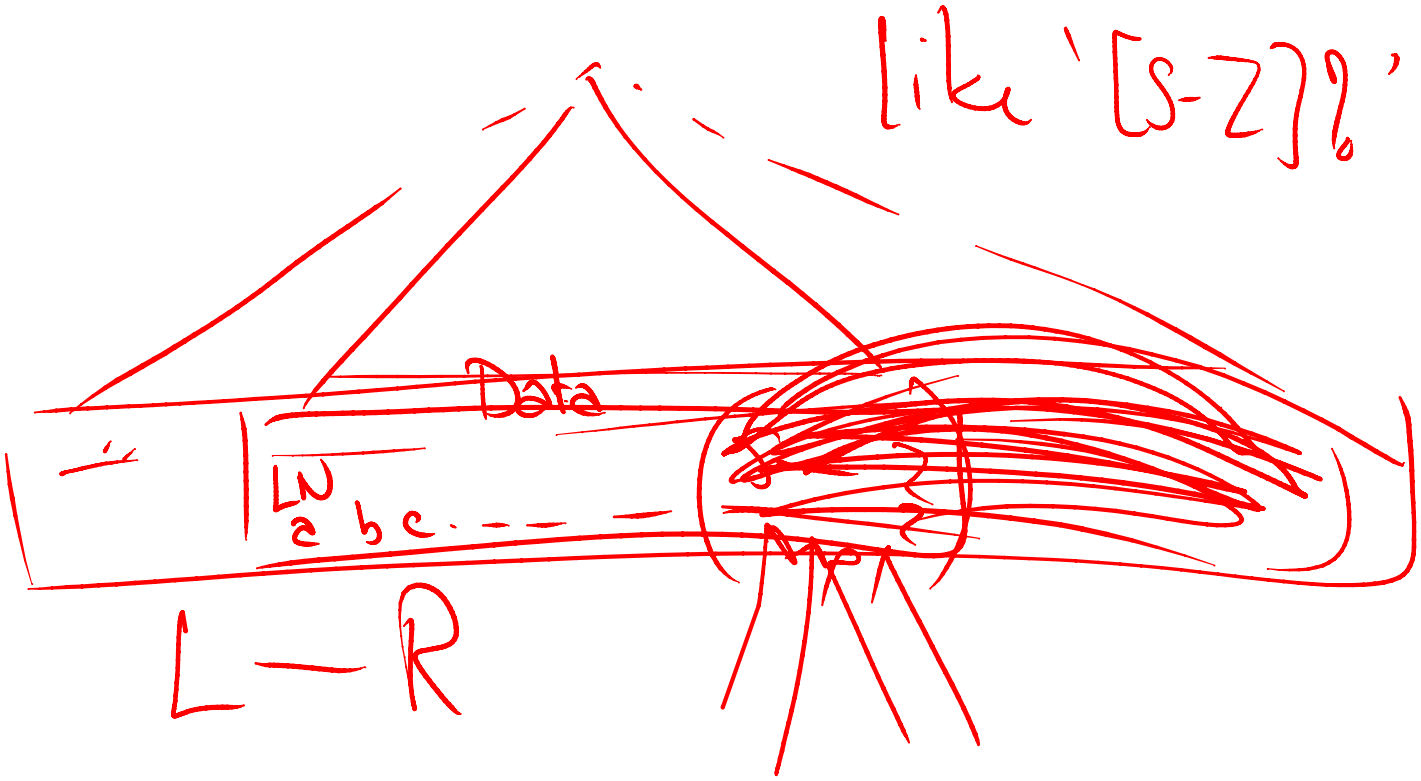
http://msdn.microsoft.com/en-us/library/dd425070.aspx



High Performance Loading Discussion

Some really good questions about "high performance loading" that lead to a discussion around these key things:

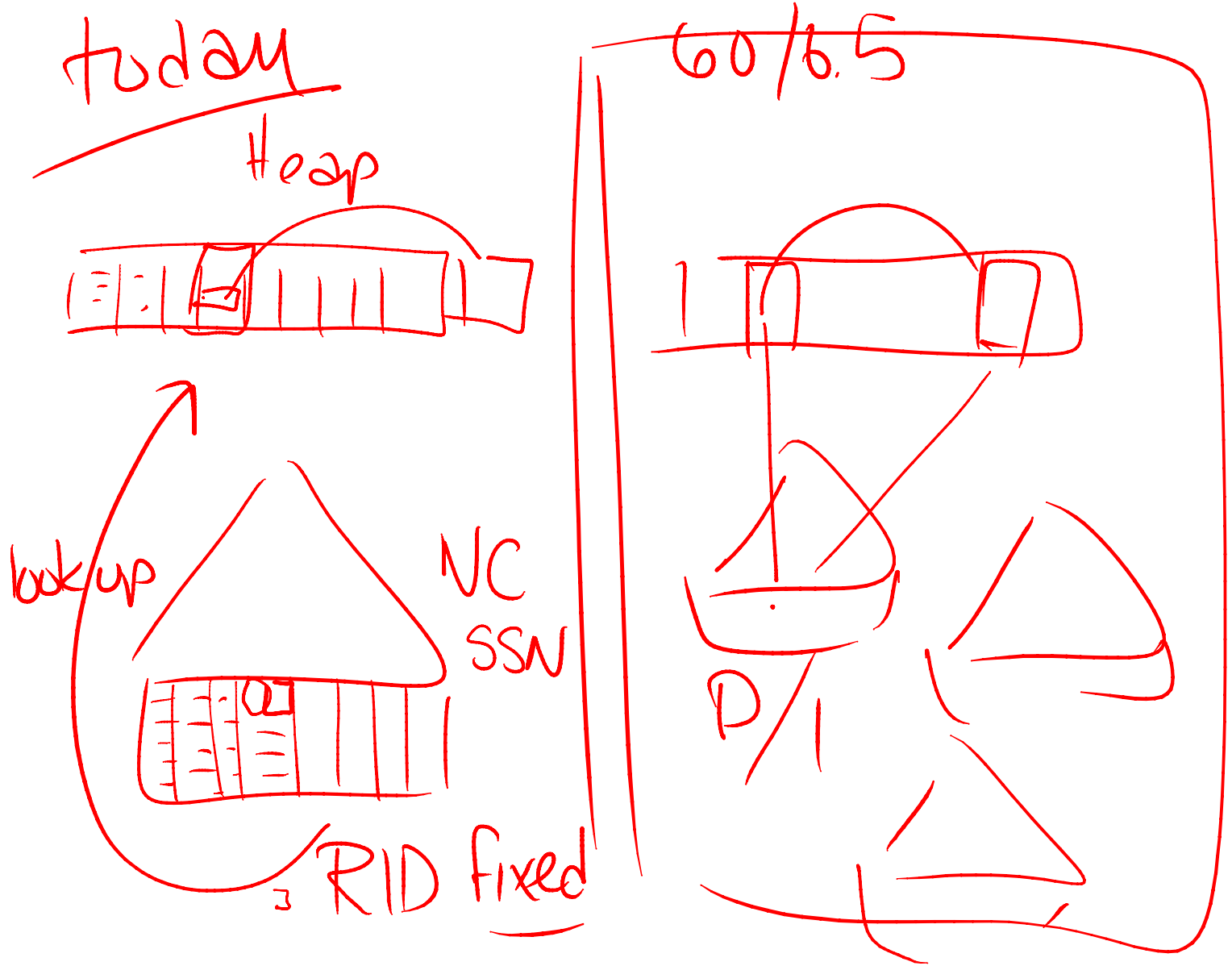
- **Loading into a HEAP** – typically this is best when you don't have an ordered file *and* when you can performance parallel load and parallel index creation.
- **Loading into a Clustered Table with an Ordered Load** – typically this is best when you have an ordered file and you cannot perform a parallel load and/or parallel index creation.
- **Loading into a Clustered Table with all nonclustered indexes pre-created** – this is almost NEVER a good idea unless you're loading only a small amount of data into an existing table. Typically less than 10% new data.



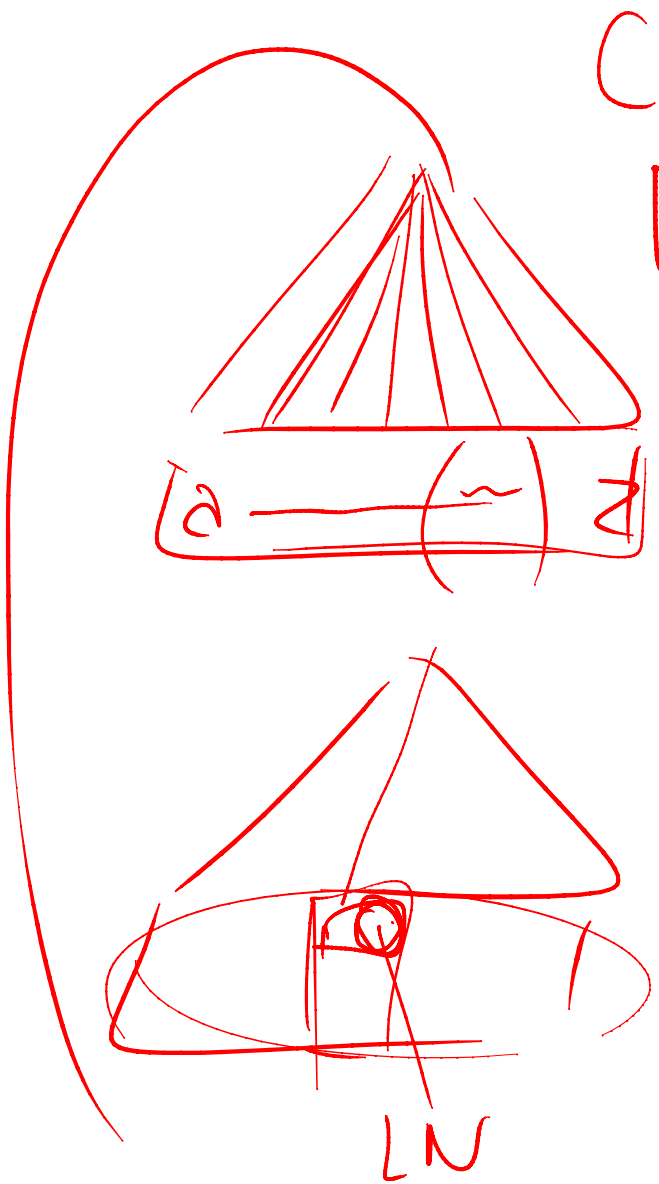
Clustering for range queries looks GREAT on paper… until modifications occur!

This was just reminding you that when folks tell you that the clustered index should be DESIGNED to support range queries you need to remind them of what the table is going to look like when there are modifications. What sounded good (the data is all together) doesn't stay that way…

- INSERTs/UPDATEs are slower due to the splits
- This wastes disk space AND memory because of the fragmentation caused.
- And the range query is no longer left-right ordered but instead very out of order (fragmentation)



SQL Server 7.0's storage engine was almost a complete rewrite. A major change in the engine around the method used from nonclustered indexes to lookup the corresponding row in the table (heap or clustered). In 7.0 and higher, SQL Server uses a FIXED RID (for heaps) and the clustering key (for tables with a clustered index). In 6.x, SQL Server used a volatile RID for ALL tables (regardless of whether they were clustered or not).



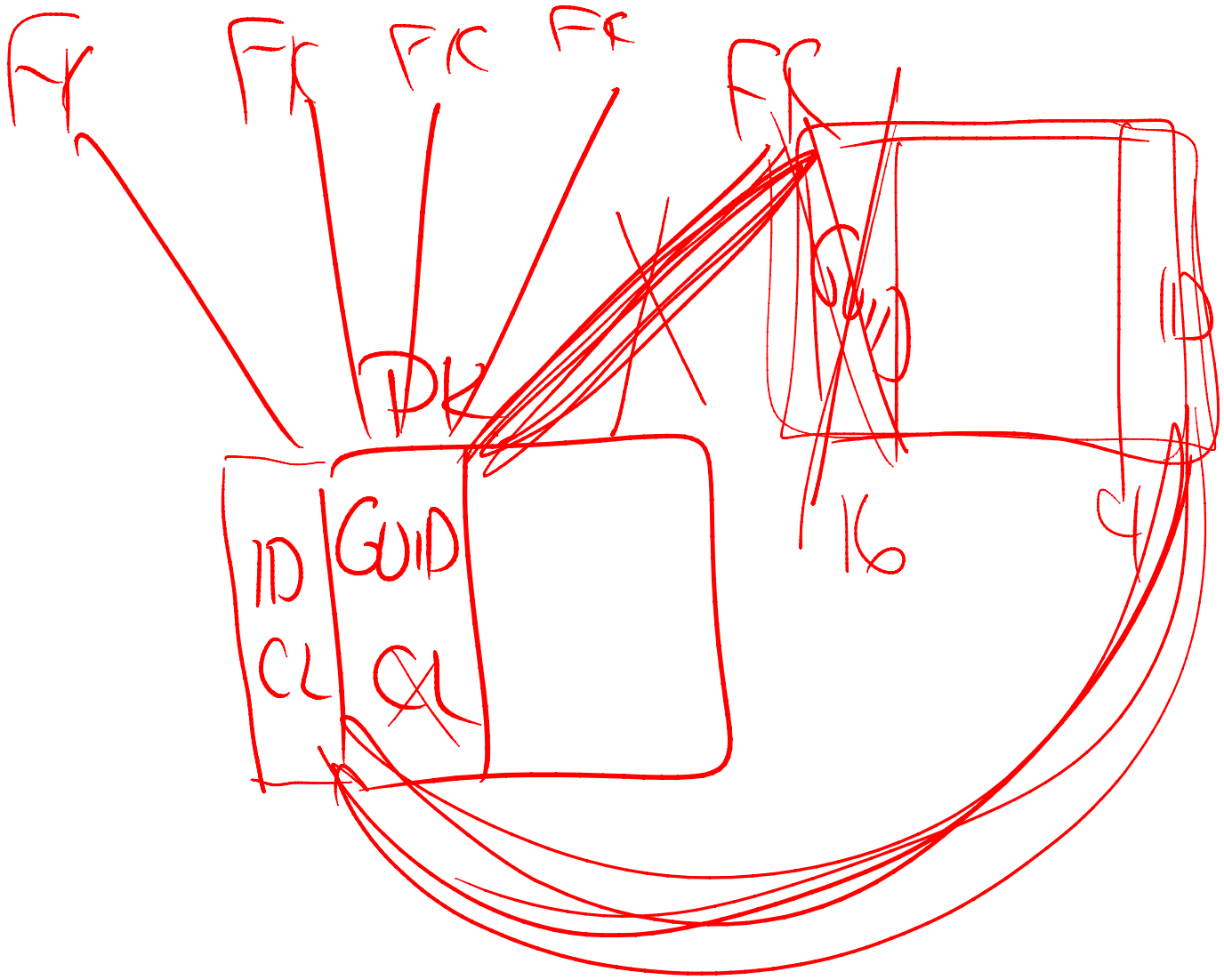
The Clustering key MUST be unique

The key reason that the clustering key must be unique is because of the lookup that must be performed when looking up a corresponding row from a nonclustered index request.

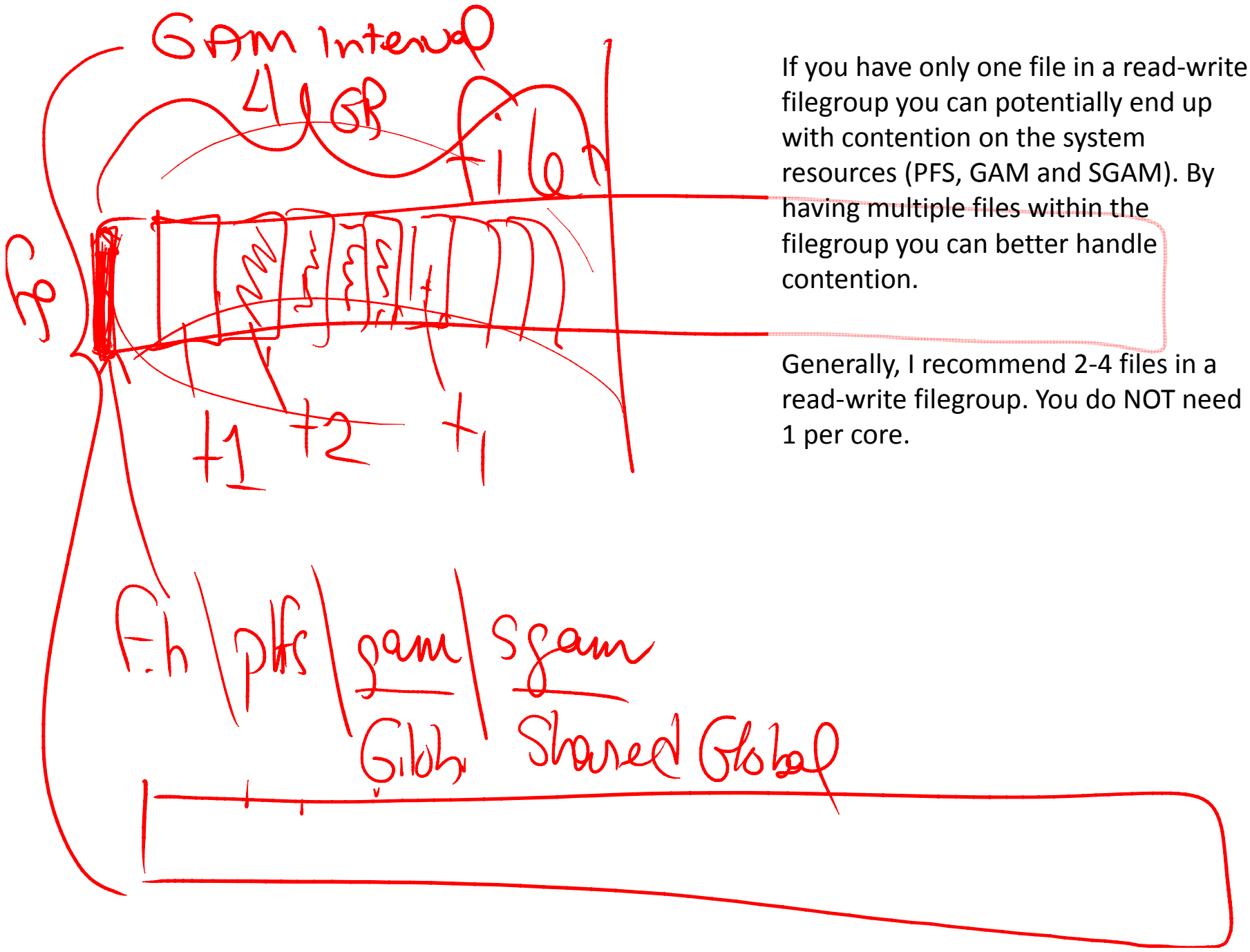
The uniquifiers are unique per duplicate key value

The FIRST Smith that is inserted is unique – so no uniquifier
The second Smith goes in and it becomes Smith 1

The FIRST Jones that is inserted is unique – so no uniquifier
The second Jones goes in and it becomes Jones 1

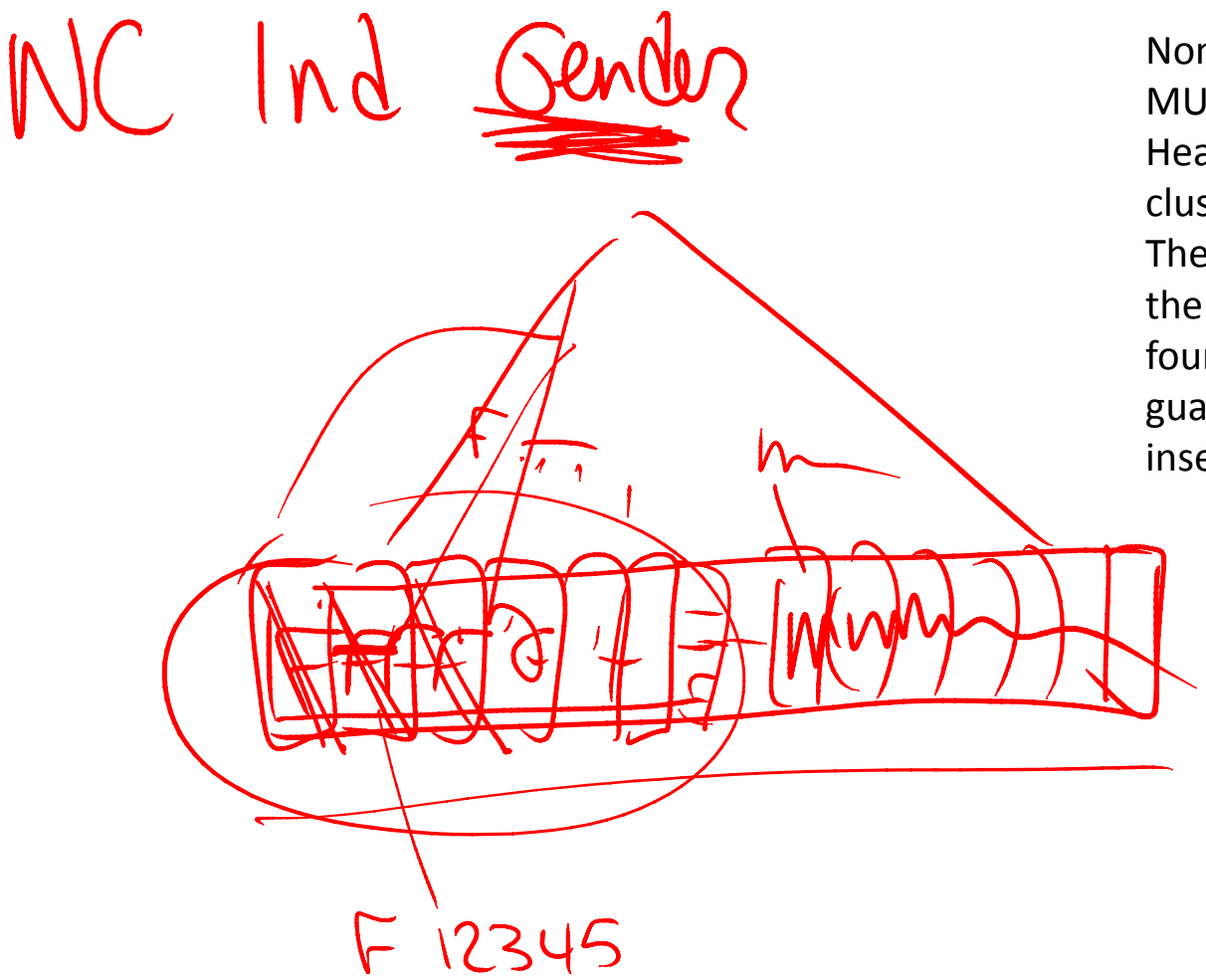


Even if your “primary” table uses a GUID for inserts. You might want to use an IDENTITY column as your foreign key. This takes less data space, less index space and results in a less expensive join. And, this might be an easier “conversion” to make for your application. You can potentially make it over time and through changes to some tables, some procs and then some code. Then, more tables, more procs and other code, etc. Slowly you can remove the old GUID columns and eventually become GUID-free! ☺

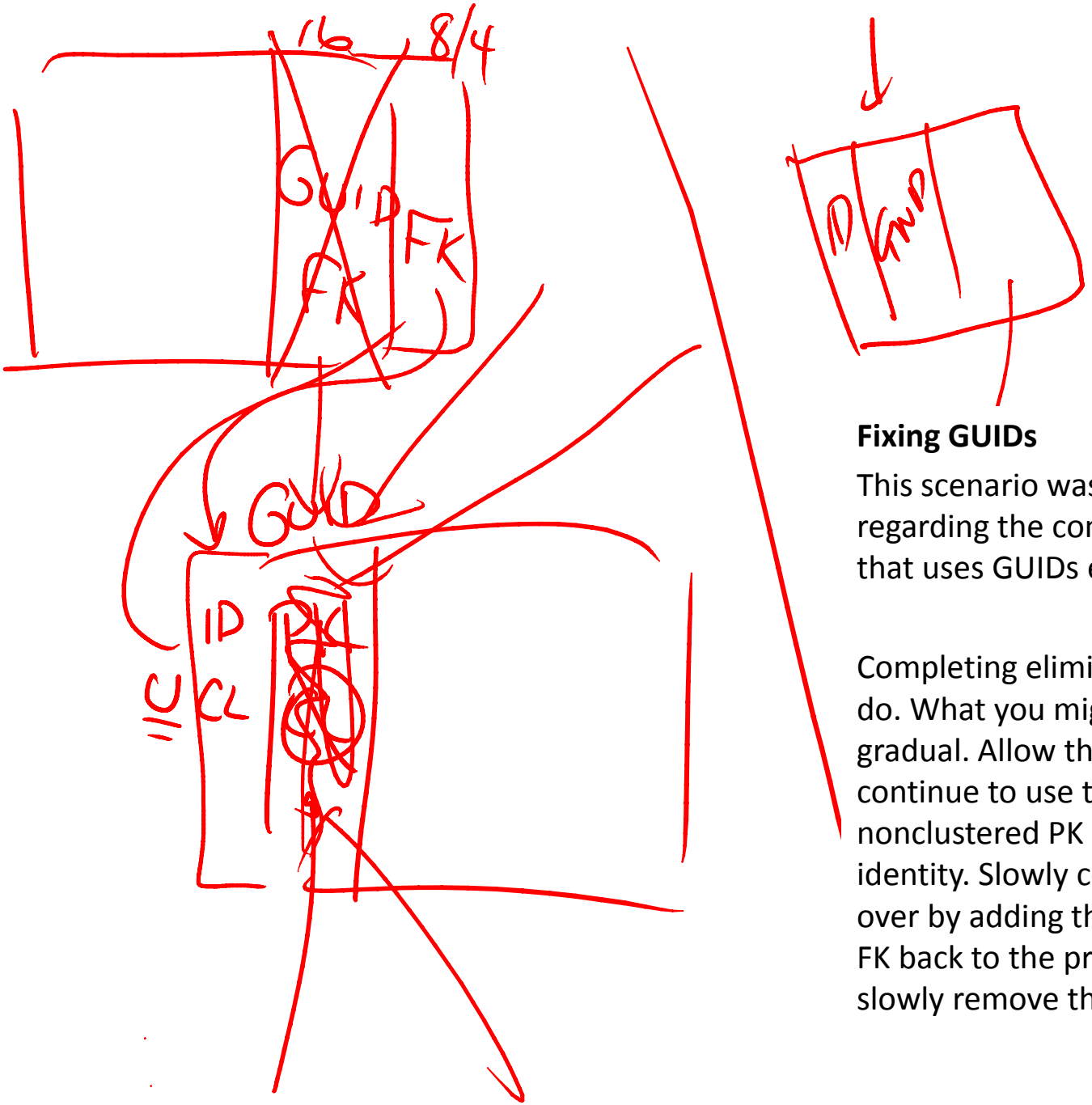


If you have only one file in a read-write filegroup you can potentially end up with contention on the system resources (PFS, GAM and SGAM). By having multiple files within the filegroup you can better handle contention.

Generally, I recommend 2-4 files in a read-write filegroup. You do NOT need 1 per core.



Non-unique nonclustered indexes MUST have the lookup key (the Heap's RID or the clustered table's clustering key) pushed up the tree. The reason stems from the fact that the rows would NOT be able to be found on a delete. And, it guarantees their position on an insert.

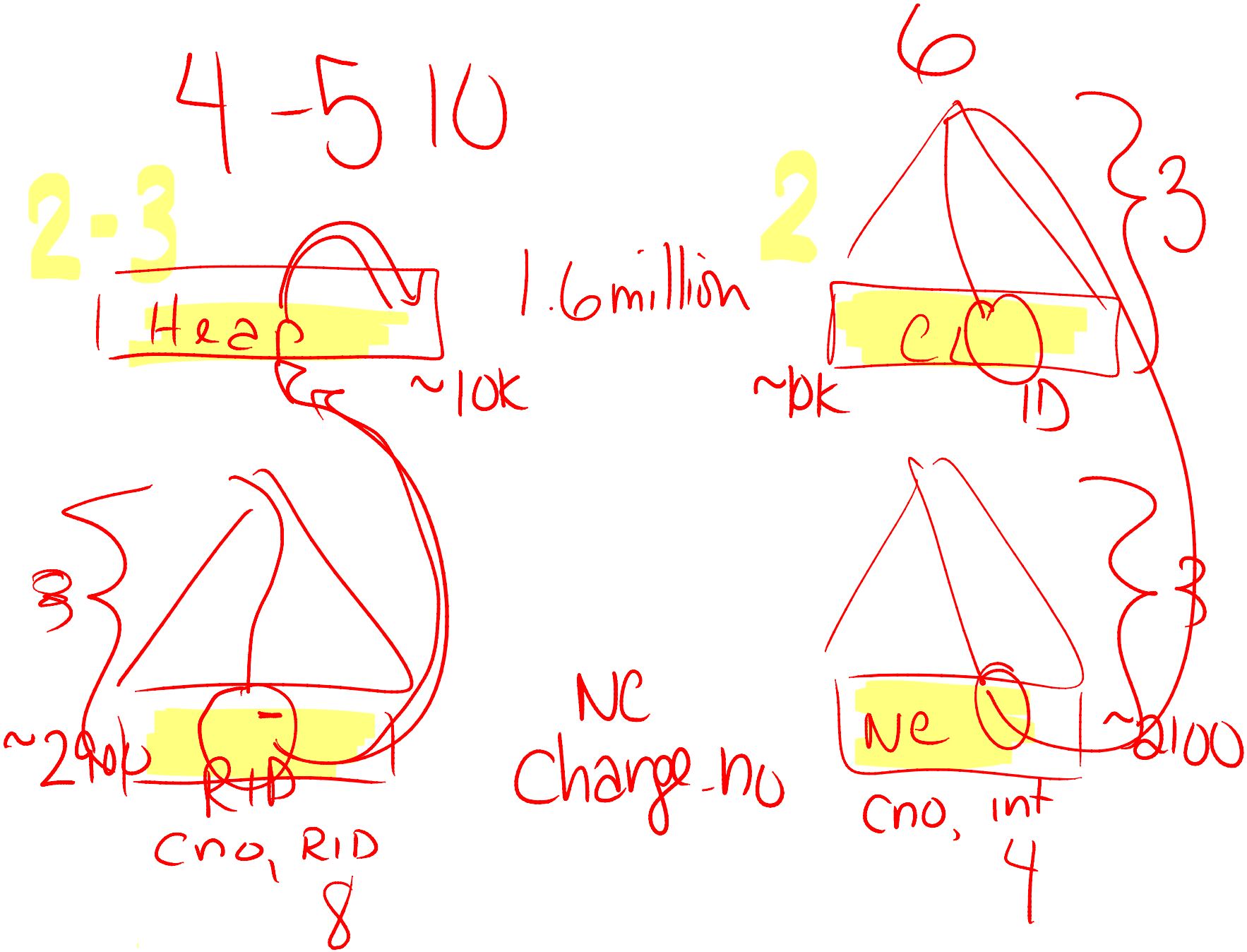


Fixing GUIDs

This scenario was from a question regarding the conversion of a database that uses GUIDs everywhere.

Completing eliminating GUIDs is tough to do. What you might do it something more gradual. Allow the application/user to continue to use the GUID as a nonclustered PK and then add a clustered identity. Slowly convert the related tables over by adding the identity column (as a FK back to the primary table) and then slowly remove this column altogether.


Module 9
Internals
and Data Access
SQL
skills


4-5 10
2-3
1 Heap
~10k
1.6million
6
2
3
~bk
ID
3
~2kb
RID
CNO, RID
8
NC
Charge-no
NE
~8100
CNO, INT
4

M8 Question

Discussing the costing (of 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

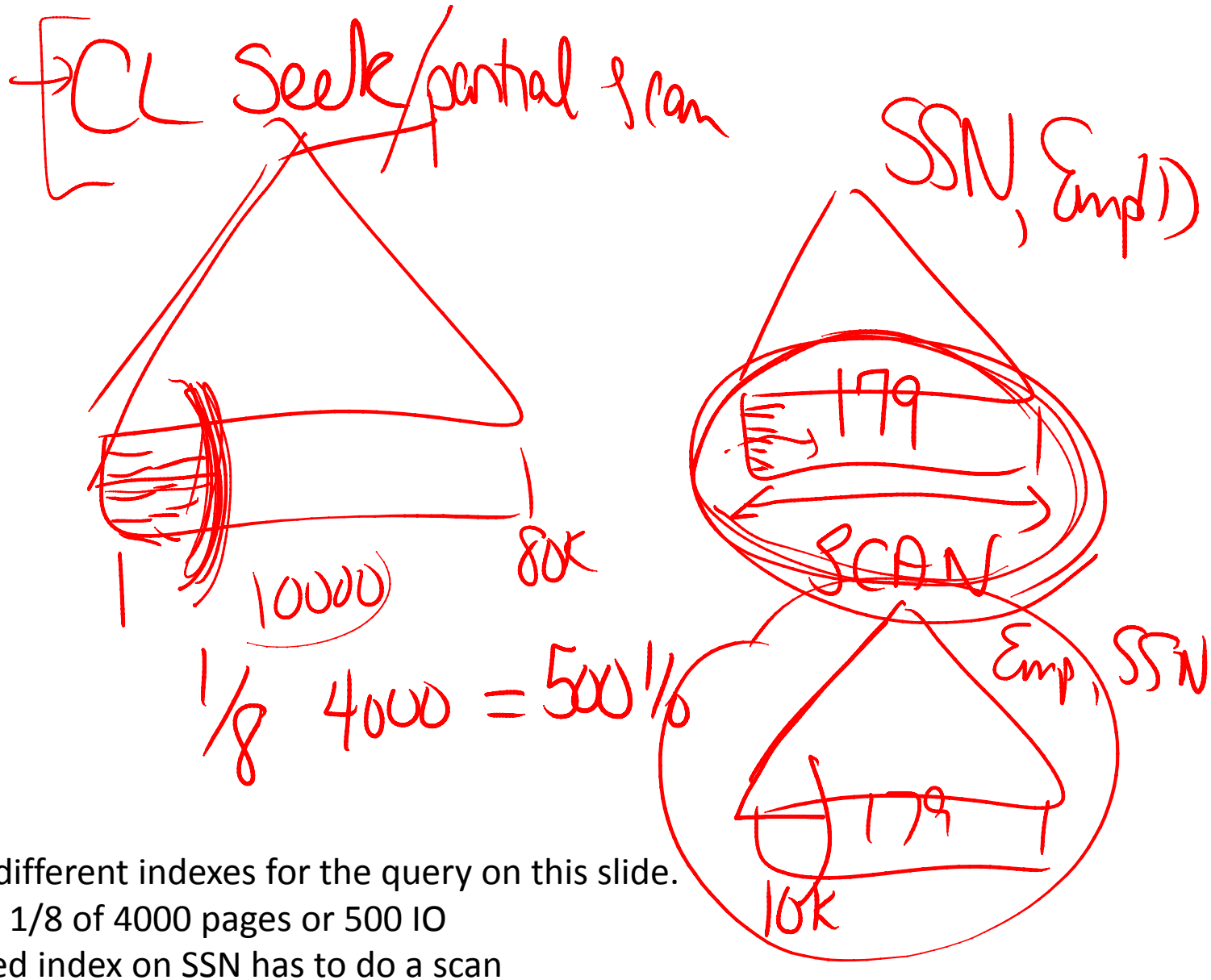
Whereas the head seems to require fewer IOs. Using the nonclustered is about the same (=3 IOs), then – using the Heap to access the data row is 1 (possibly 2 if there's been record relocation) for a total of 4-5 IOs.

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. ☺



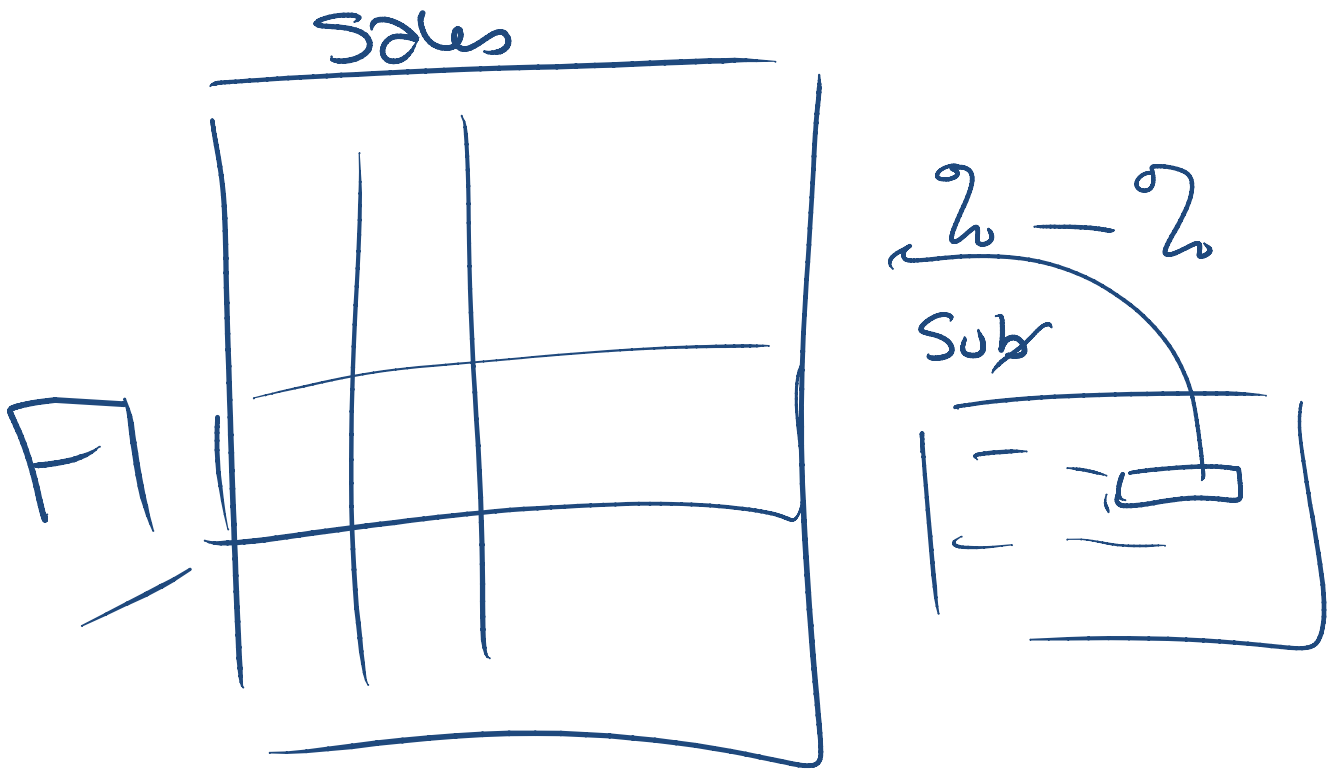
M9s16

Costing IOs for different indexes for the query on this slide.

The clustered is 1/8 of 4000 pages or 500 IO

The nonclustered index on SSN has to do a scan

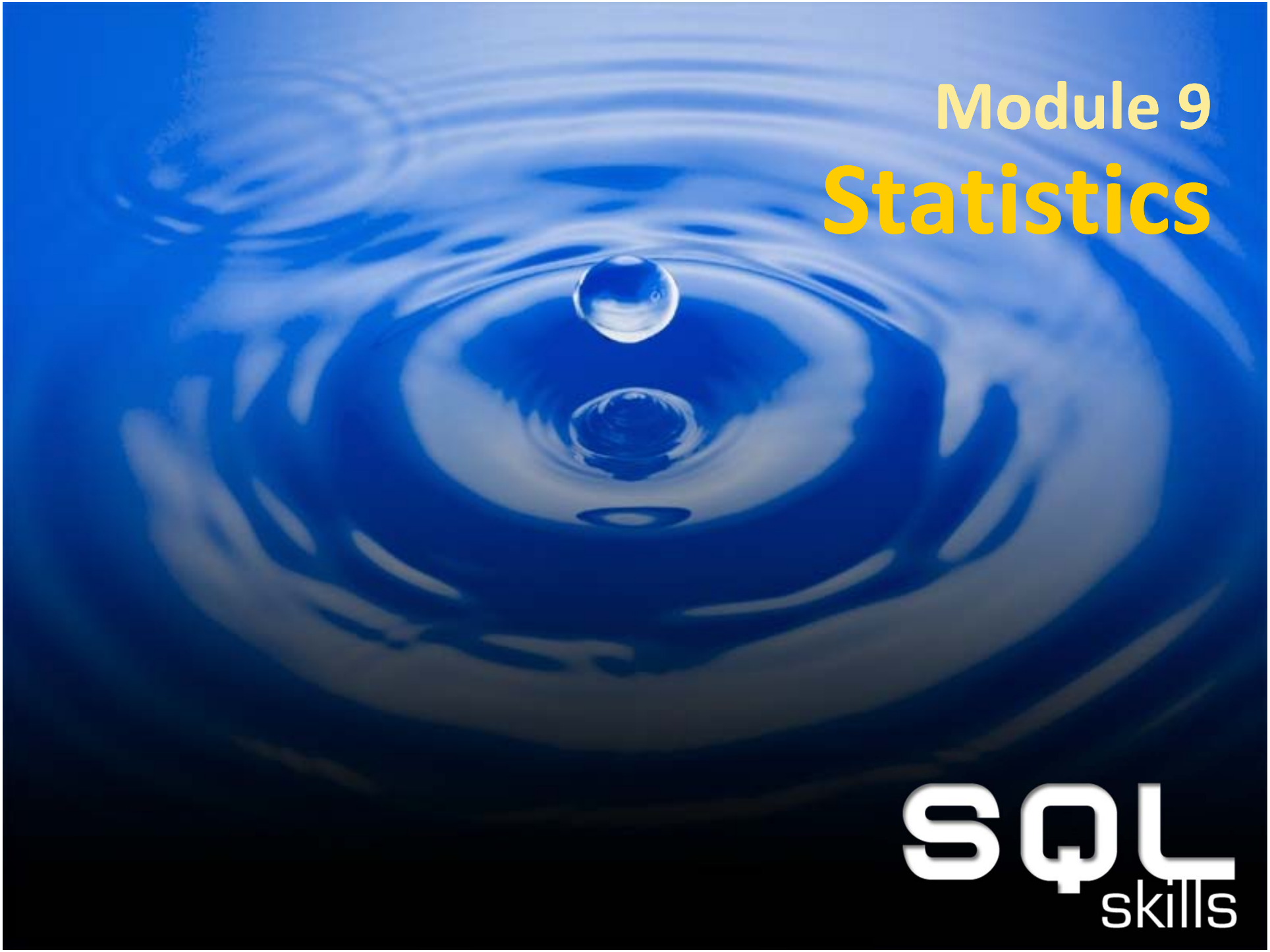
But, a nonclustered covering index that was ordered EmpID then SSN (S17) would be a seekable nonclustered covering index with the fewest IOs as SQL Server would only have to read 1/8 of the 179 pages.

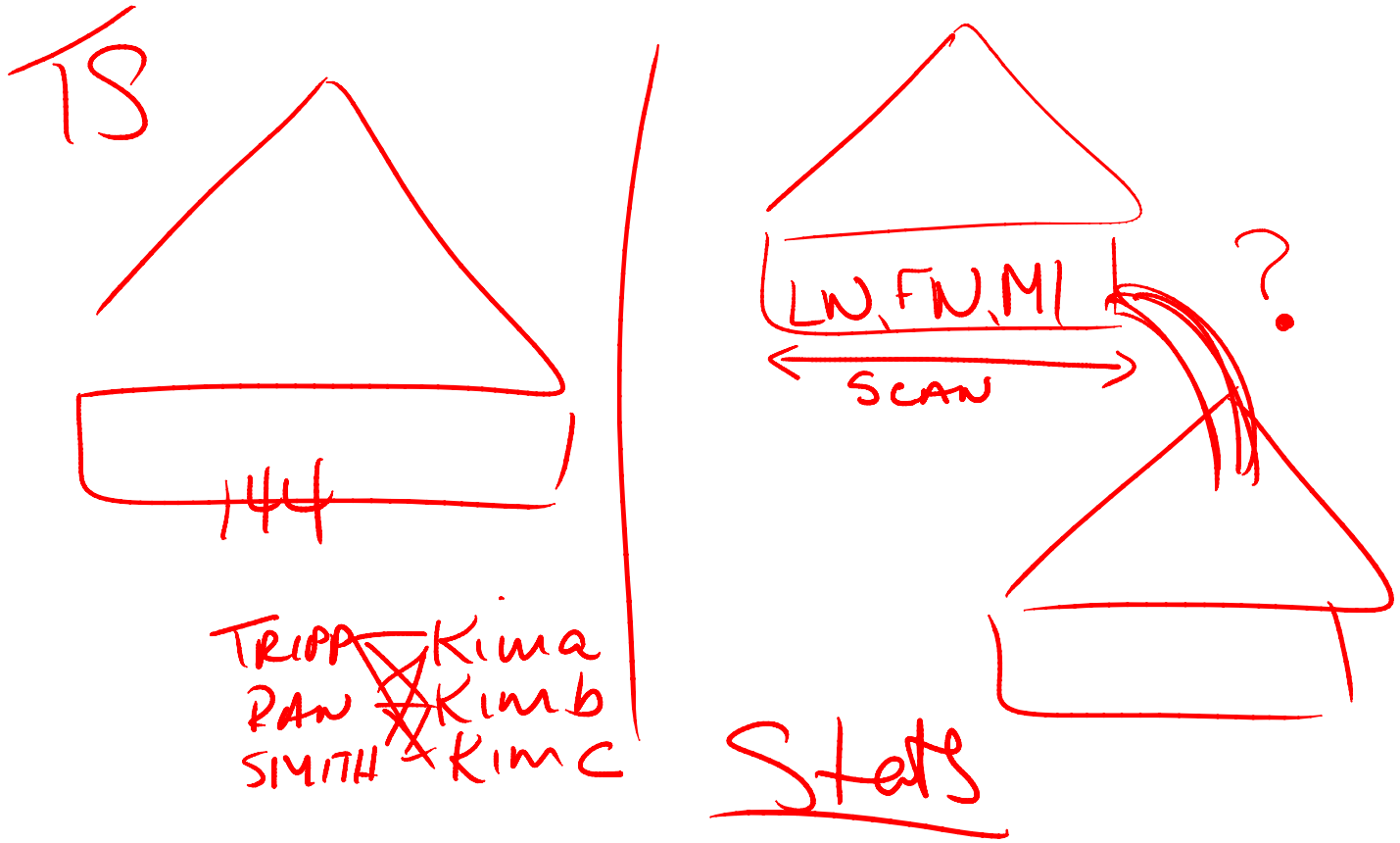


Solving an application problem with a filtered index

Somewhat recently I worked with an application that had a habit of adding % to the start/end of a queried value but only for a subset of data. However, that subset was not selective enough to use an nonclustered index. So, the query would do a complete table scan. This was causing huge performance problems… enter, filtered index!

We could cover the request but only where REGION = ‘subset’ so only that data would be scanned. OK, no, I didn’t like the %value% search either but at least I could reduce the impact of it by using filtered indexes!


Module 9
Statistics
SQL
skills

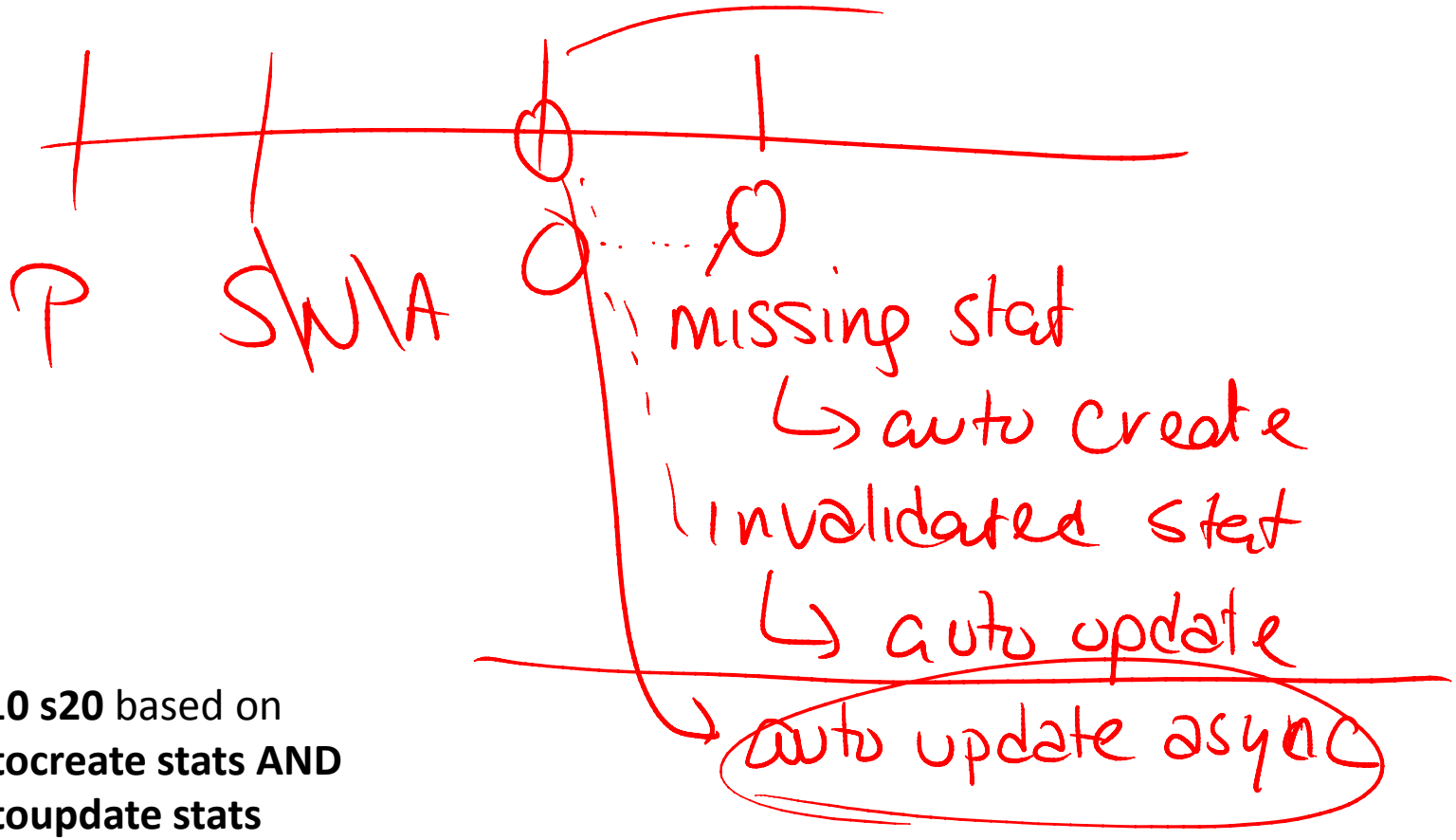


M11s10 – Column-level distribution from left-based density subsets…

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.

M10s11 – 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.



New slide M10 s20 based on
M10s15 – autocreate stats AND
M10s18 – autoupdate stats

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

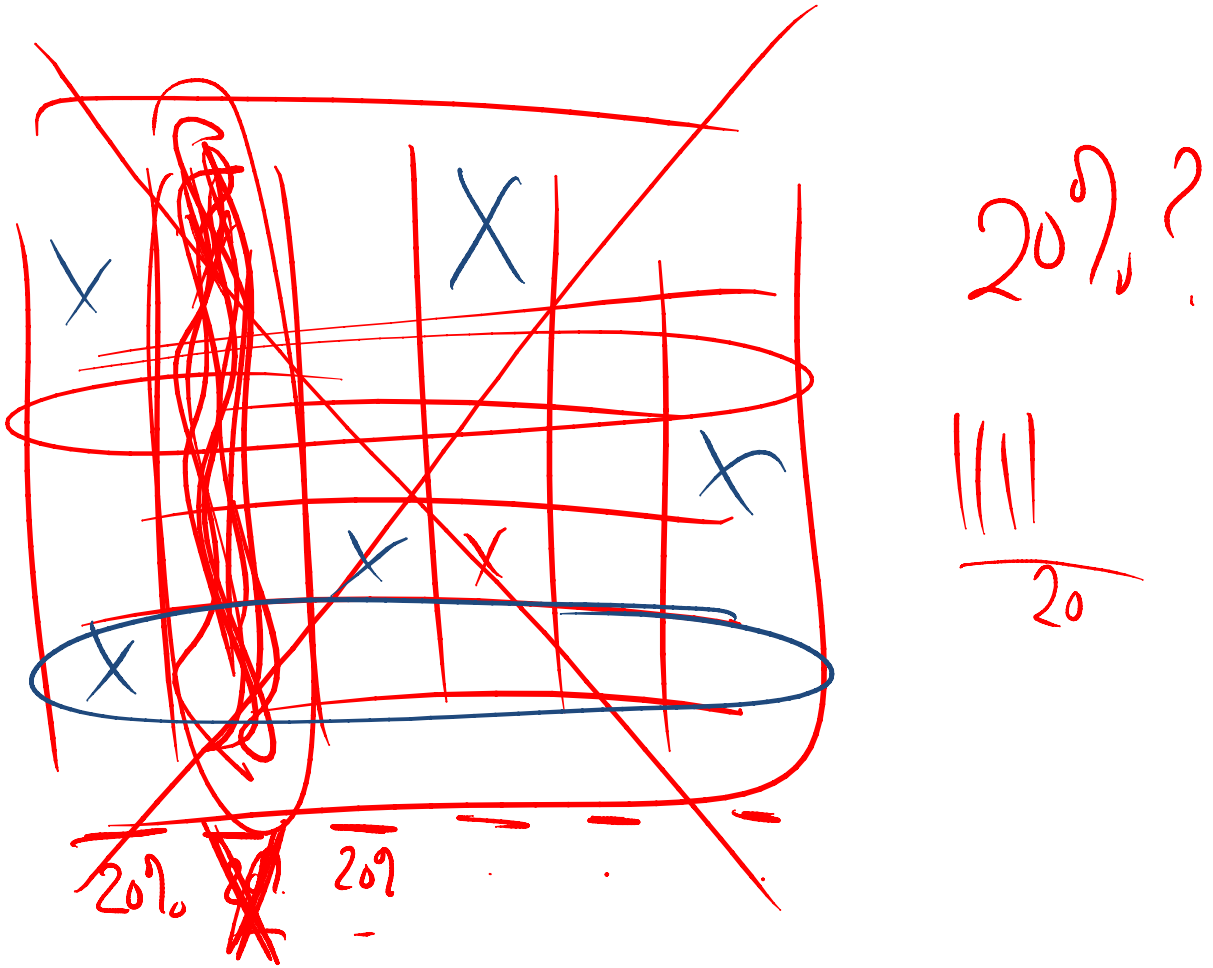
Missing statistics event

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.

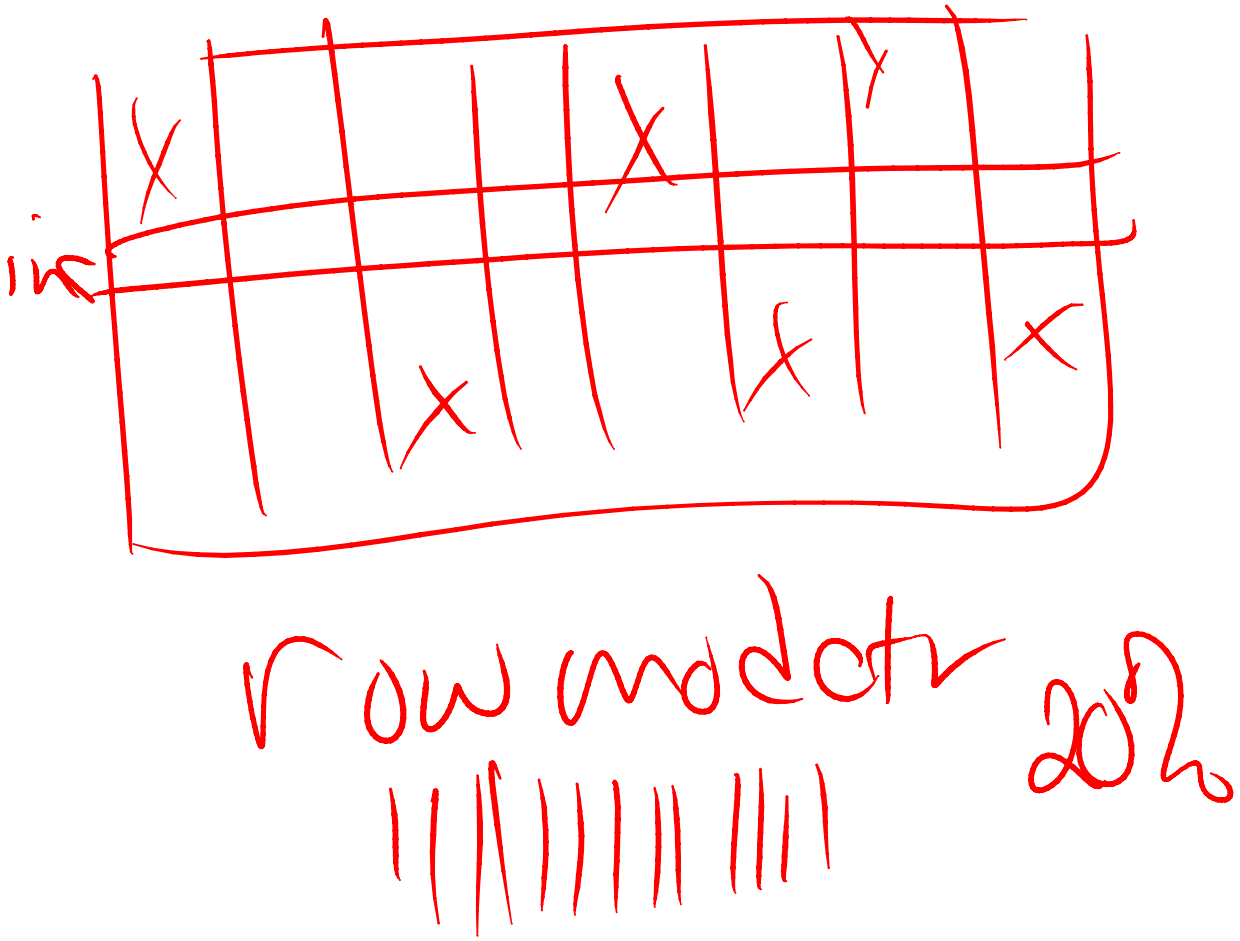
If “auto updates stats async” are on then SQL Server will optimize using the invalidated statistics. Then, asynchronously, SQL Server will update the stats.



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:

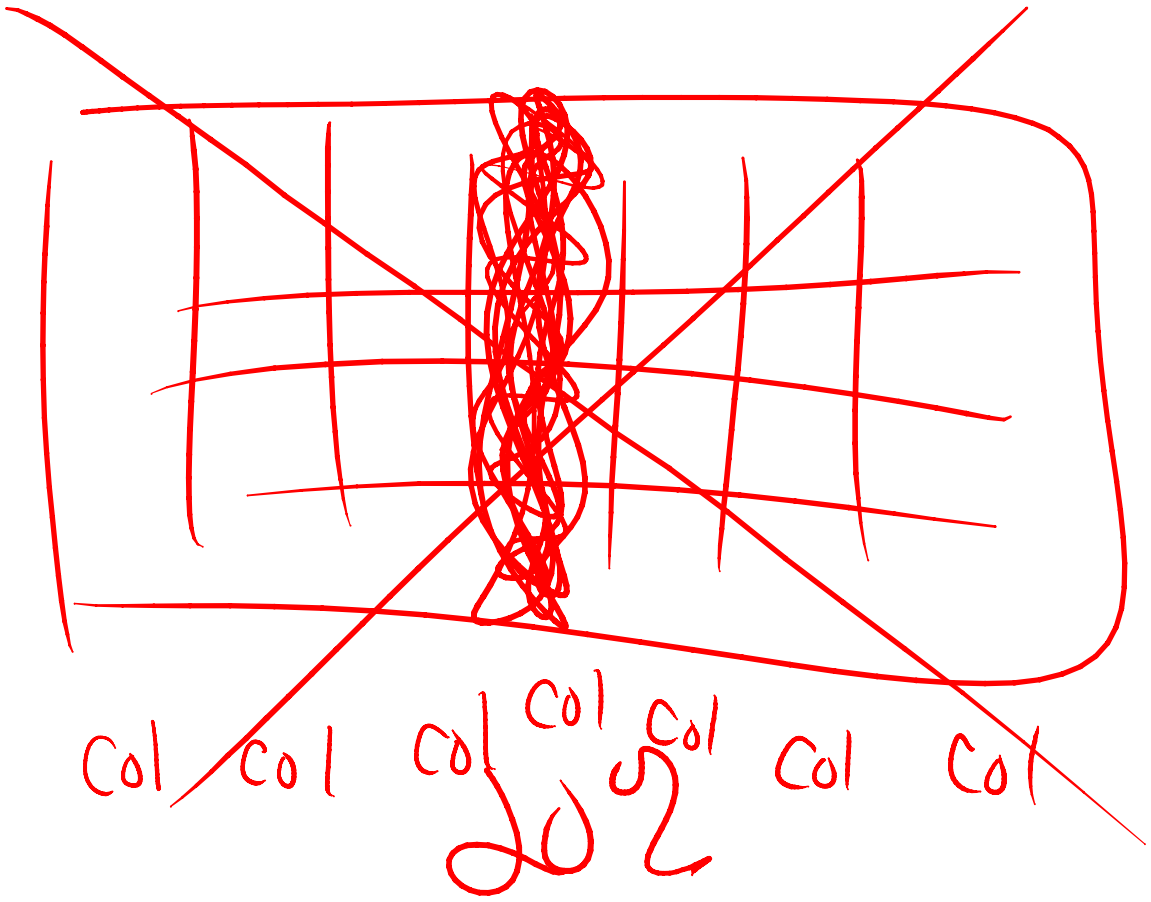
- The pro is that you don't invalidate too soon (when you have a highly volatile column)
- The con is that you might not invalidate soon enough if your modifications are reasonably distributed.



M10s17

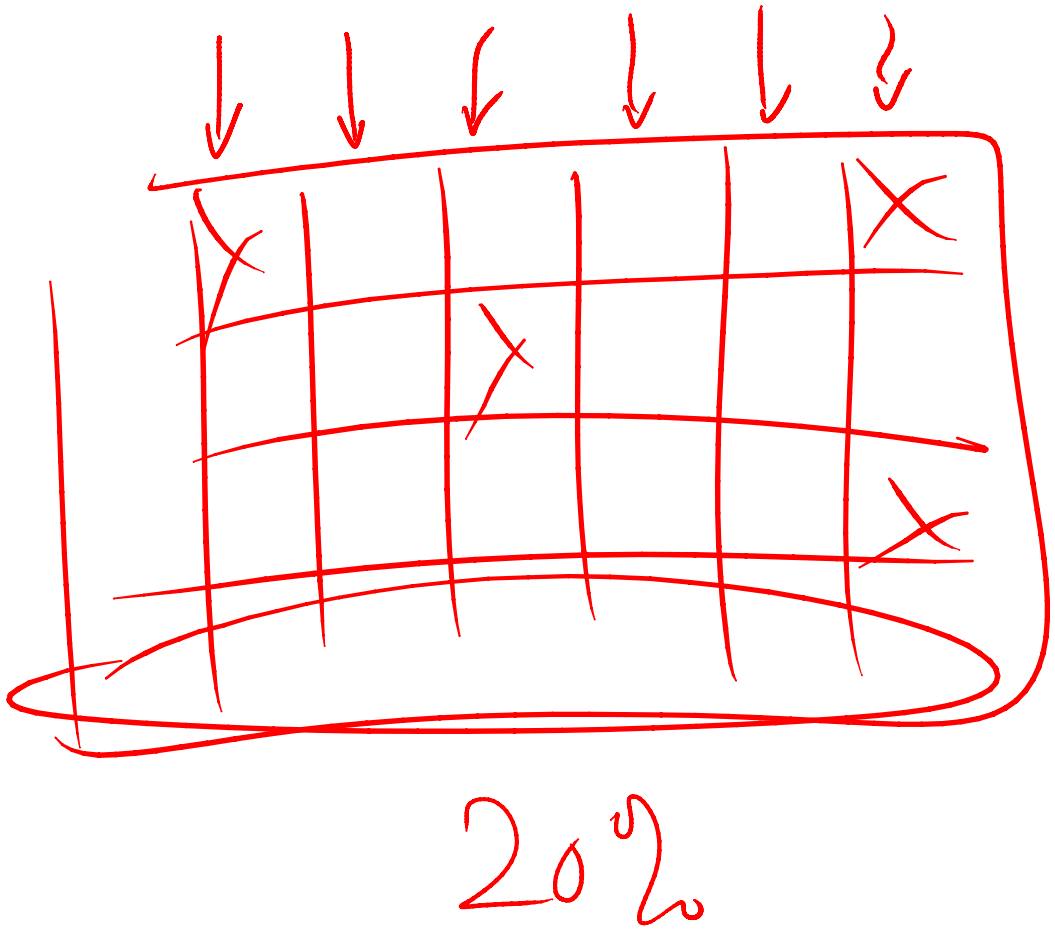
This shows how relatively distributed modifications effect each column with a small percentage of the modifications but when they add up to 20% ALL columns become invalidated (this was the PRE-2005 way of doing it):

- The pro was that each column had a reasonable (but lower) percentage of rows modified and the stats were invalidated
- The con is that a single overly volatile column would cause ALL statistics to be invalidated (which was overkill).



M10s17

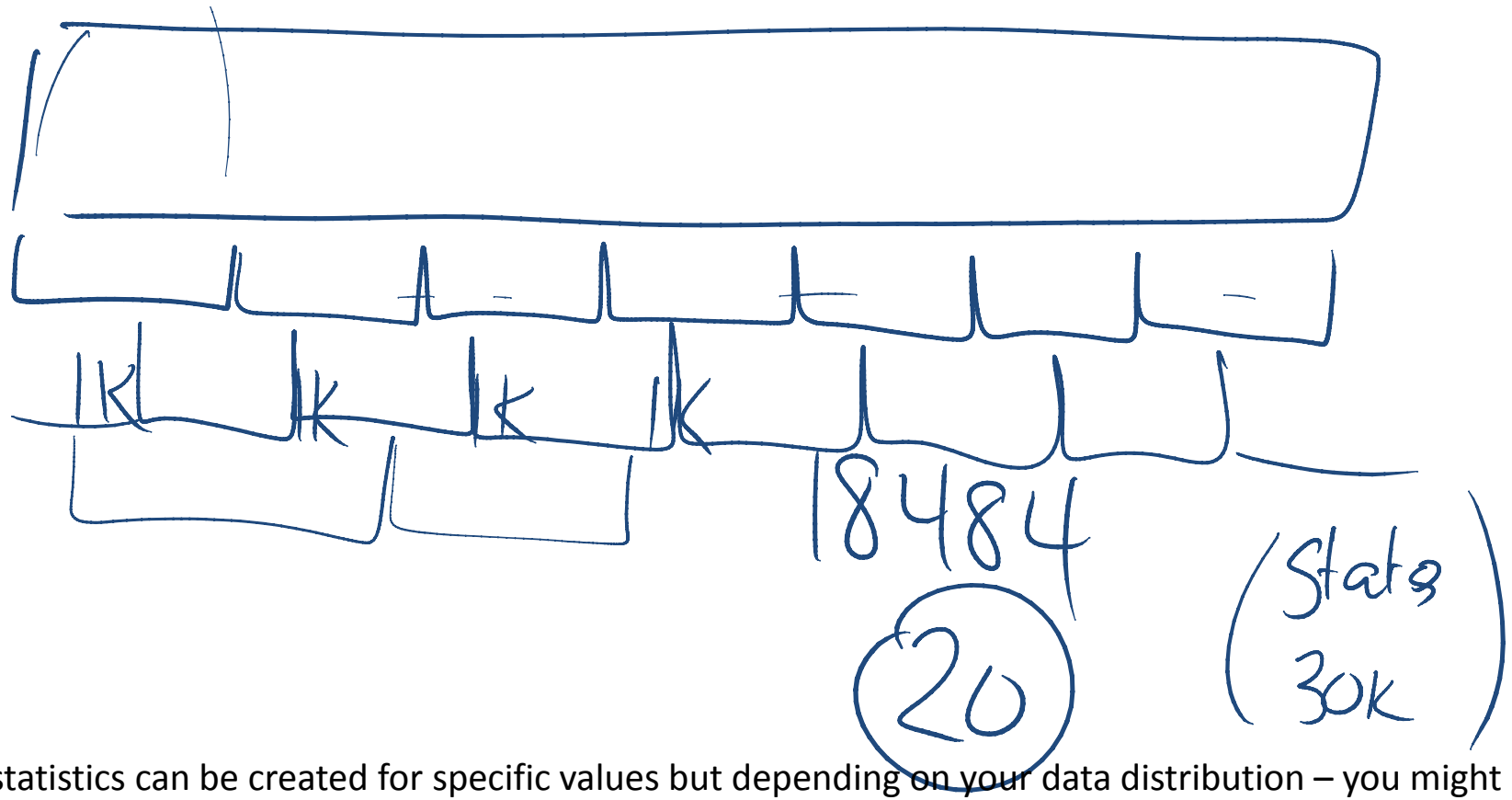
The problem prior to 2005 was that a single overly volatile column would cause ALL statistics to be invalidated.



M10s17

In 2005+ they use an internally defined colmodctr:

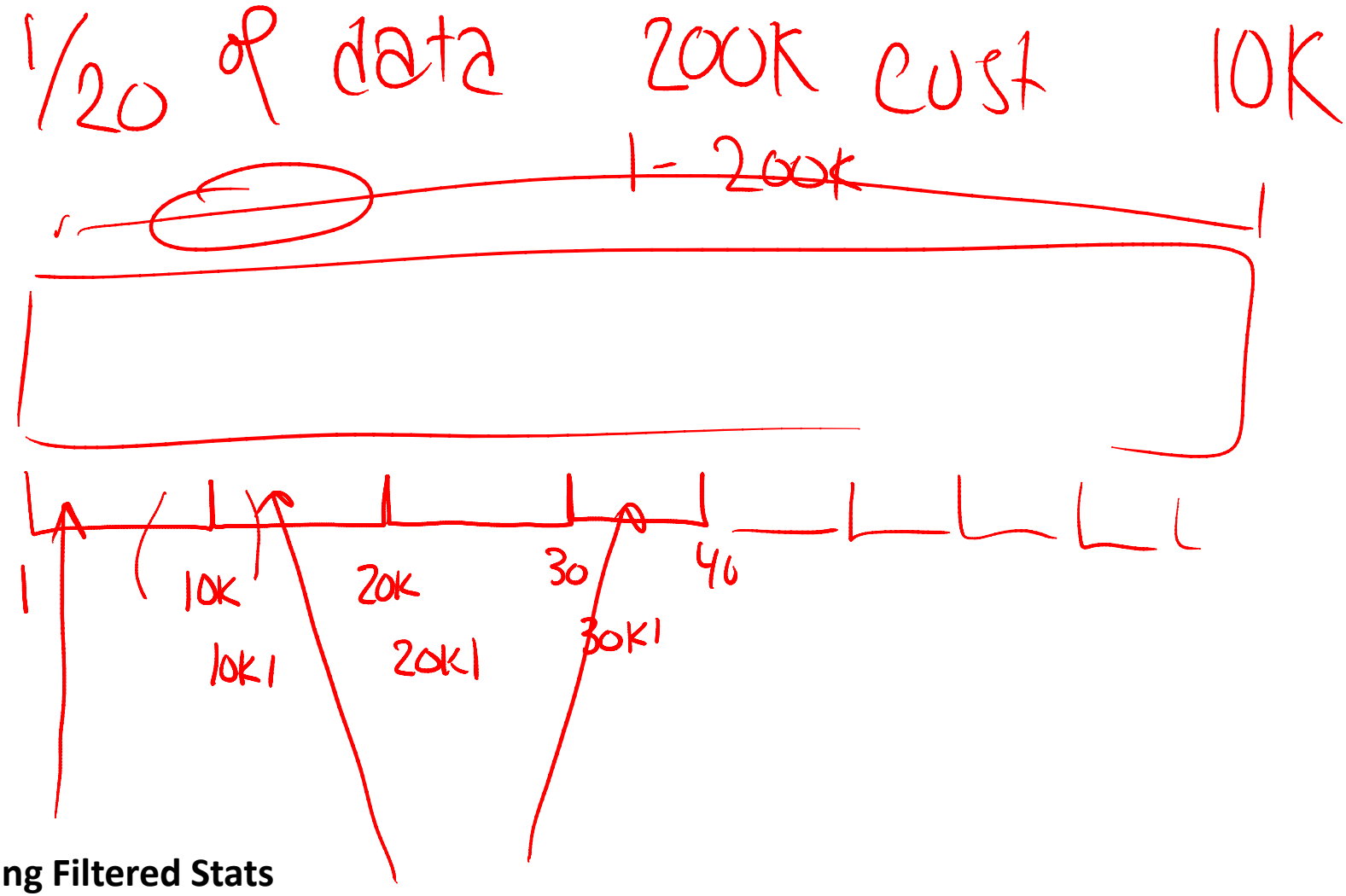
- The pro is that you don't invalidate too soon (when you have a highly volatile column)
- The con is that you might not invalidate soon enough if your modifications are reasonably distributed.



M10s26

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).

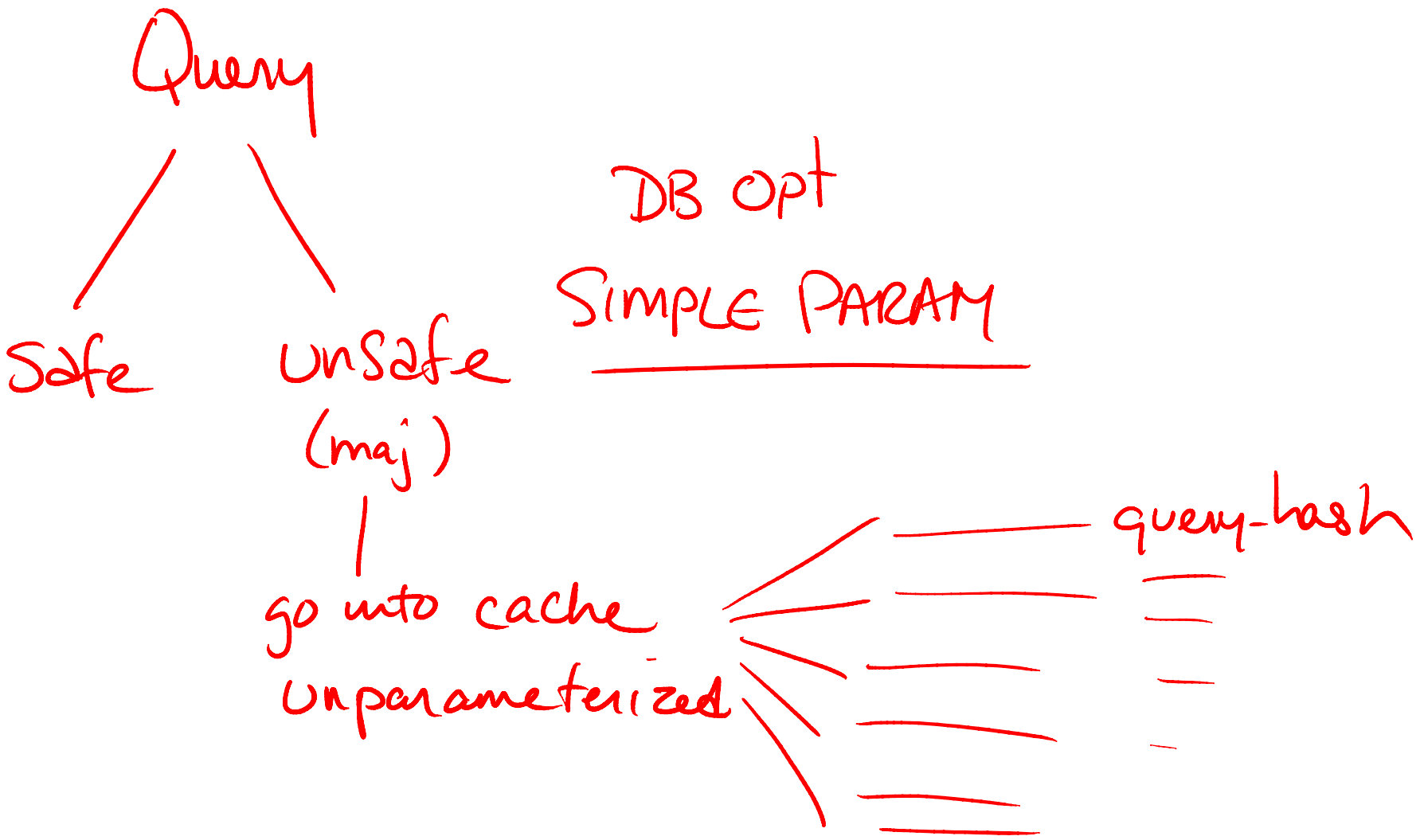
* The optimizer can detect whether interval conditions in a filtered index cover, or "subsume" interval conditions of a query.



Creating Filtered Stats

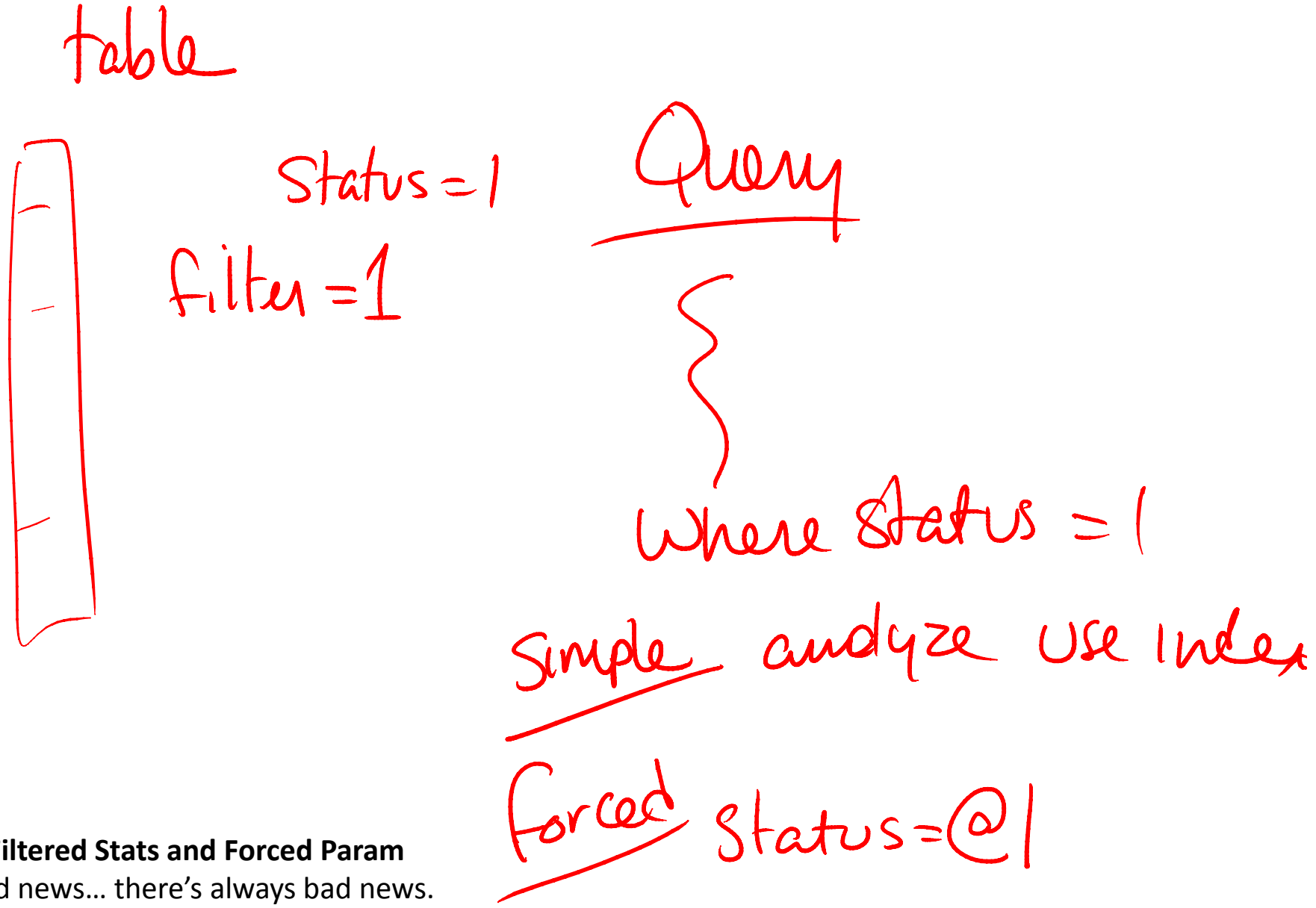
The idea is to just have better statistics than what you have currently. To do this you can divide the range into 10-20 buckets. This will give you 10 times (or 20 times) better information in terms of statistics.

You WILL need to periodically review the values/ranges and add more or divide them up again to ensure that the statistics are good!



Simple vs. Forced Parameterization

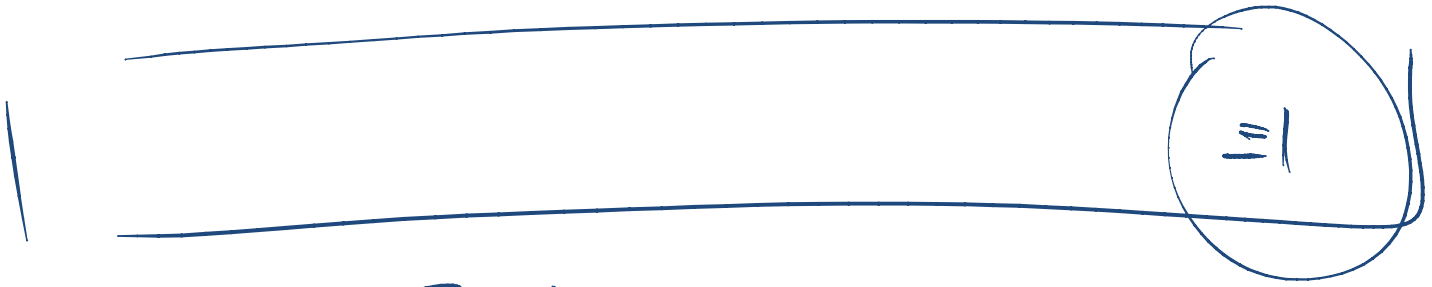
The general process is that SQL Server analyzes a query to determine if it's safe or not (the majority of them will **NOT** be safe). If it's safe then it can get parameterized and reused. If it's unsafe then it goes into memory as an individual query (and it's harder to determine the cumulative effect of these queries). Check out the query_hash and query_plan_hash.



M10s Filtered Stats and Forced Param

The bad news… there's always bad news.

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.



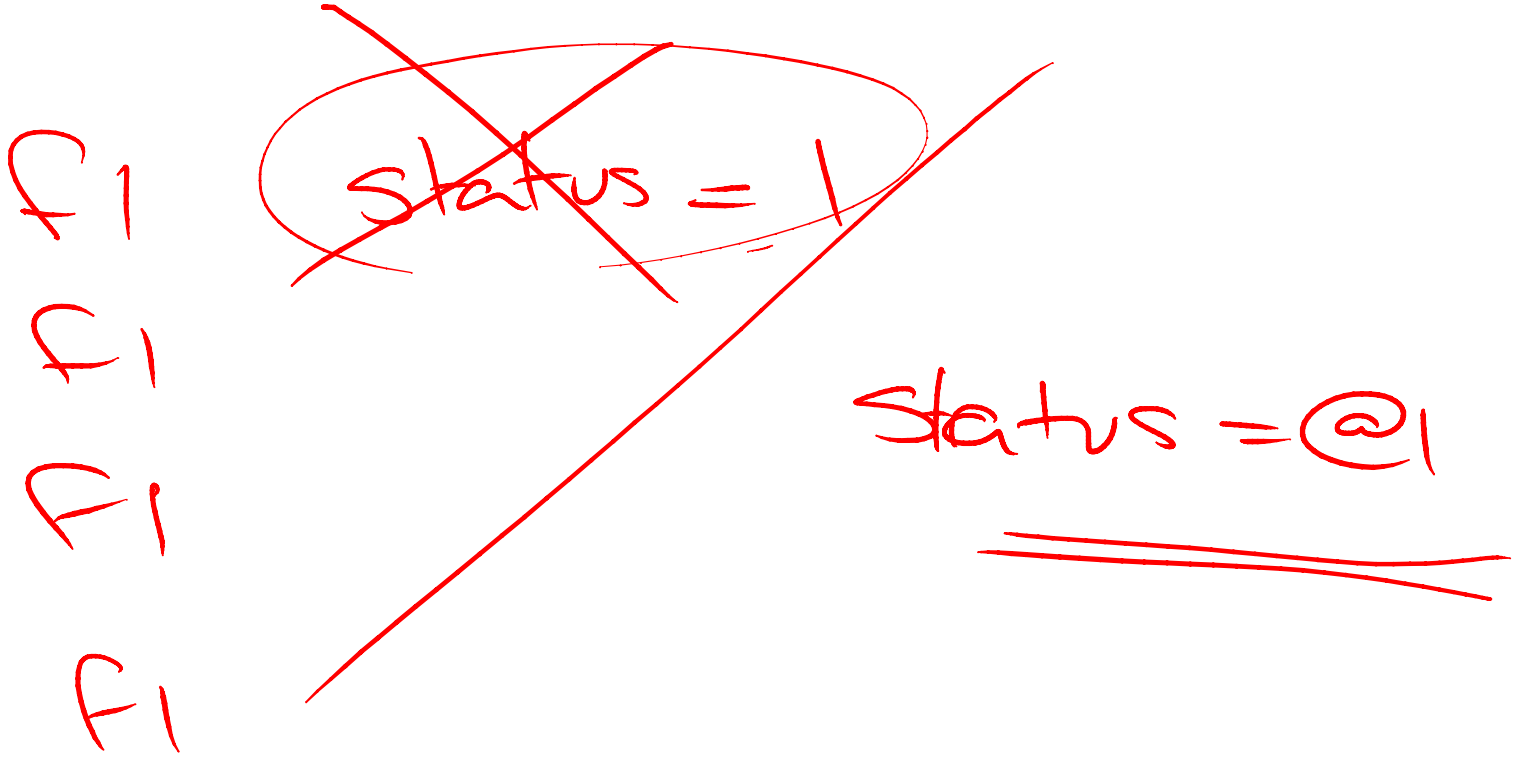
M10s28

The bad news… there's always bad news.

For Updates:

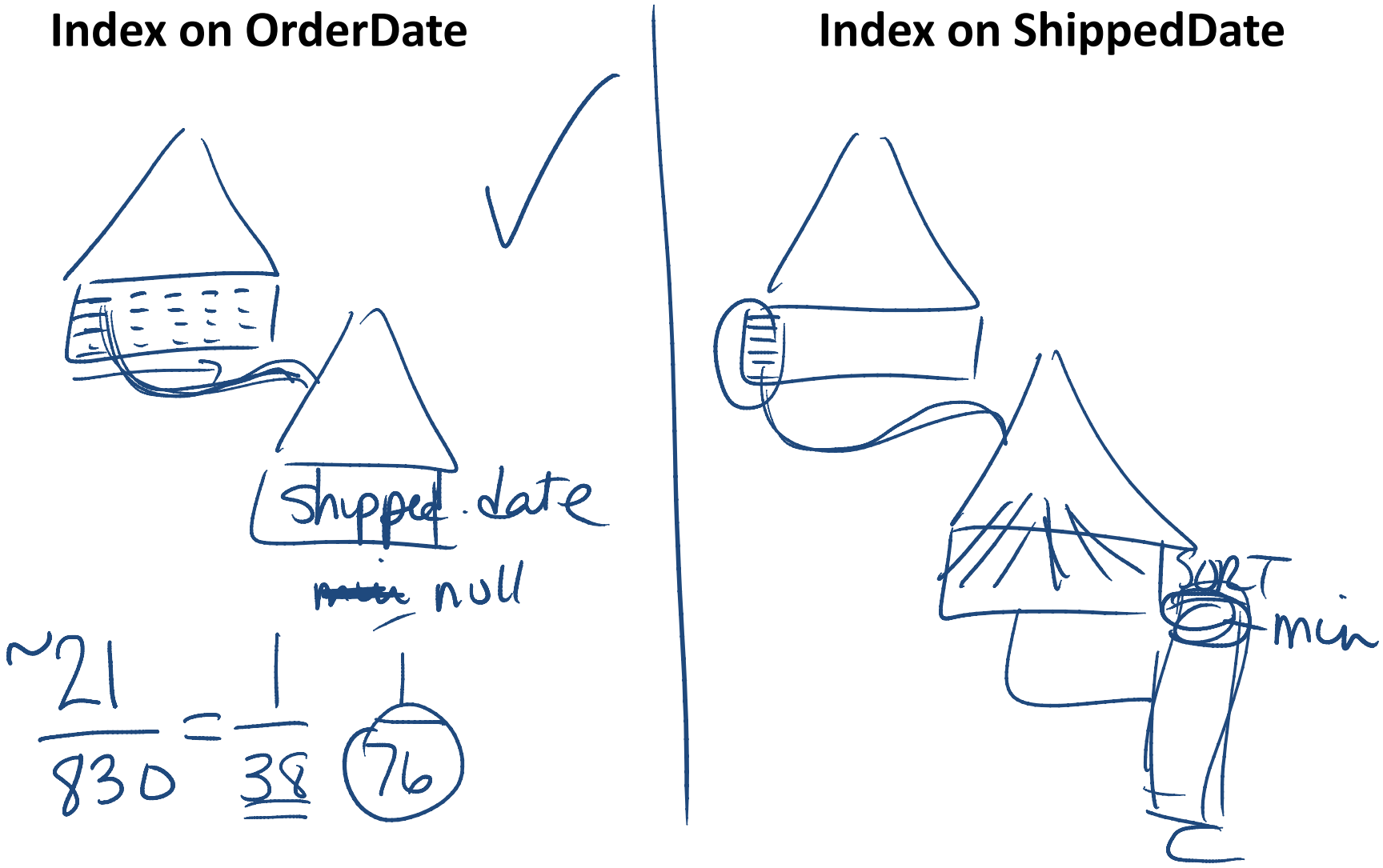
statistics for a filtered index OR a filtered stat – do NOT get updated until that column's statistics get updated (which is when the colmodctr) is reached

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.



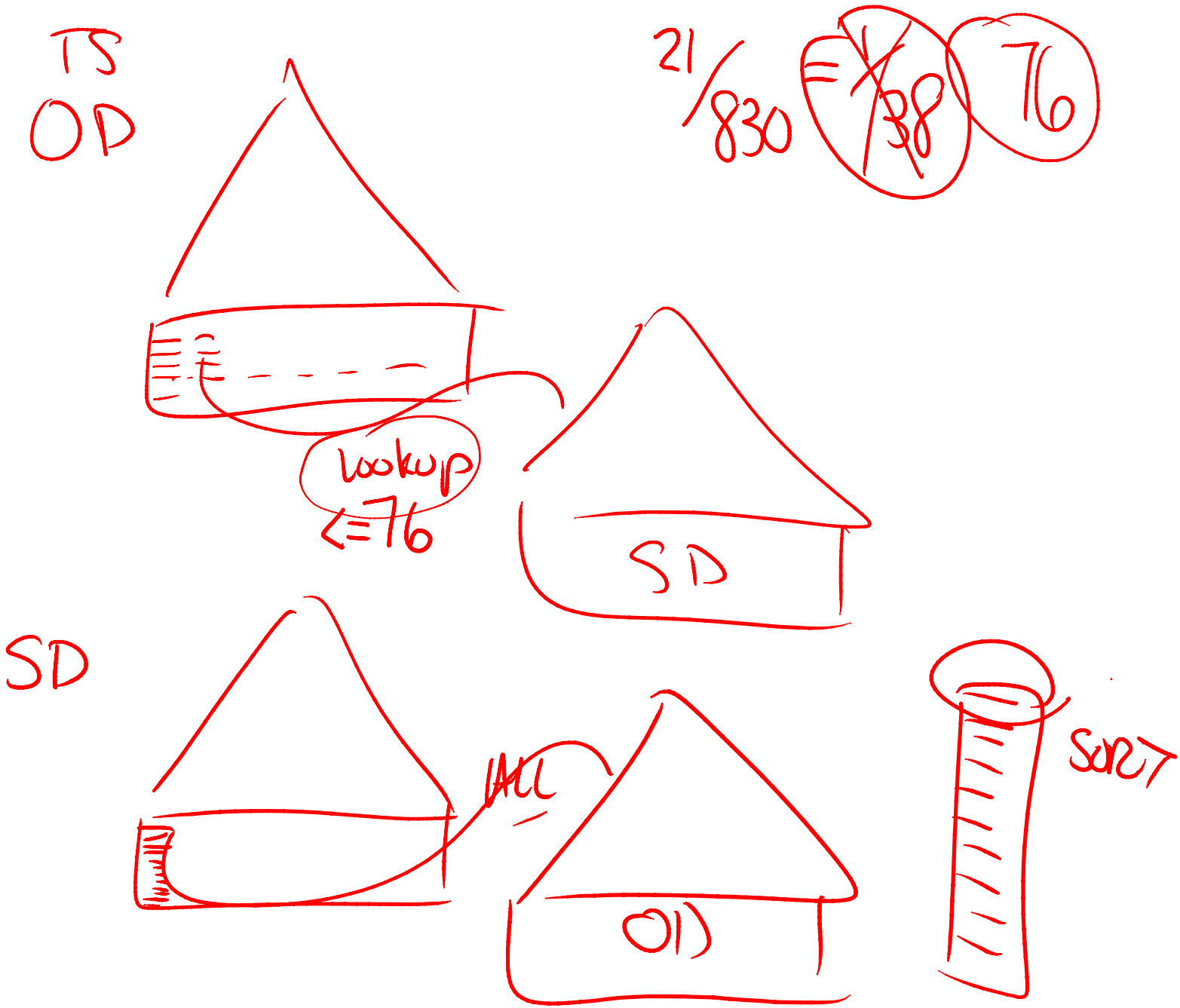
Database option: FORCED parameterization
Because of how forced parameterization changes each variable to a parameter – the value of that parameter is unknown. As a result, a filtered index (for example, WHERE status = 1) cannot be used when a query has been parameterized to status = @1. There's no guarantee that the value their searching on is 1.

End result. You will NOT have good results with filtered indexes IF you use FORCED parameterization. **The good news:** This is NOT on by default and not likely to be on in most environments.

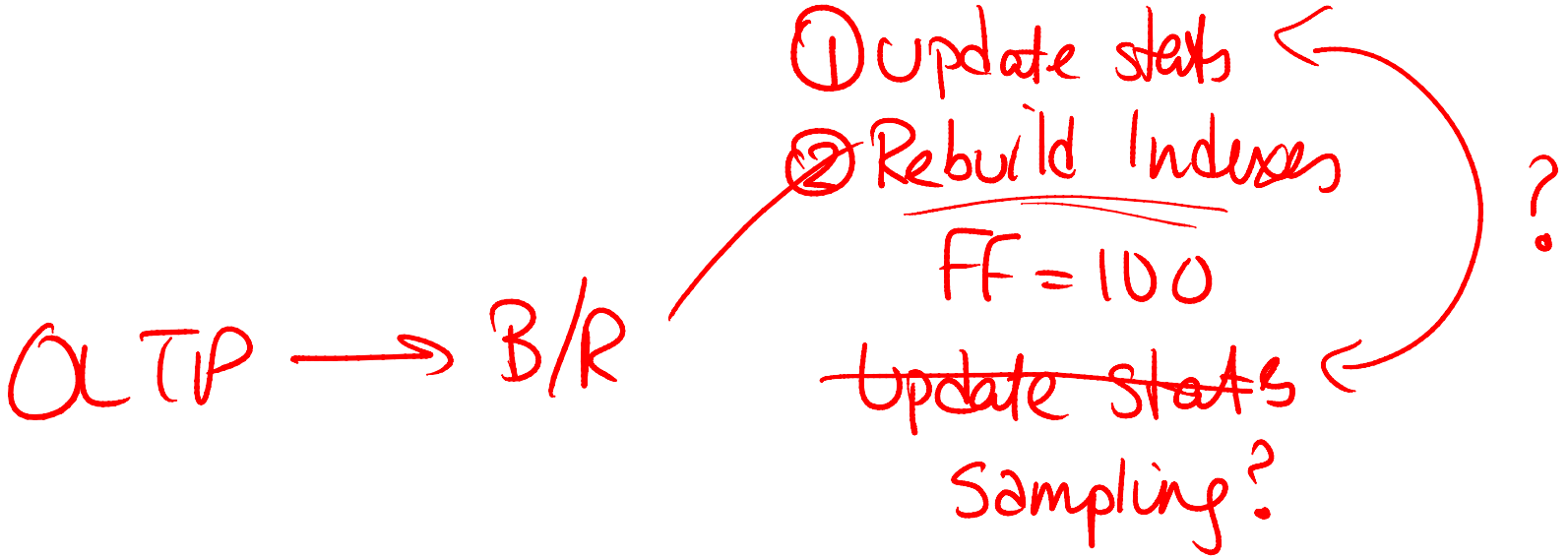


M10s30

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.


TS
OD
21/830
76
lookup
<=76
SD
SD
SD
SD27

Poor Man's DW



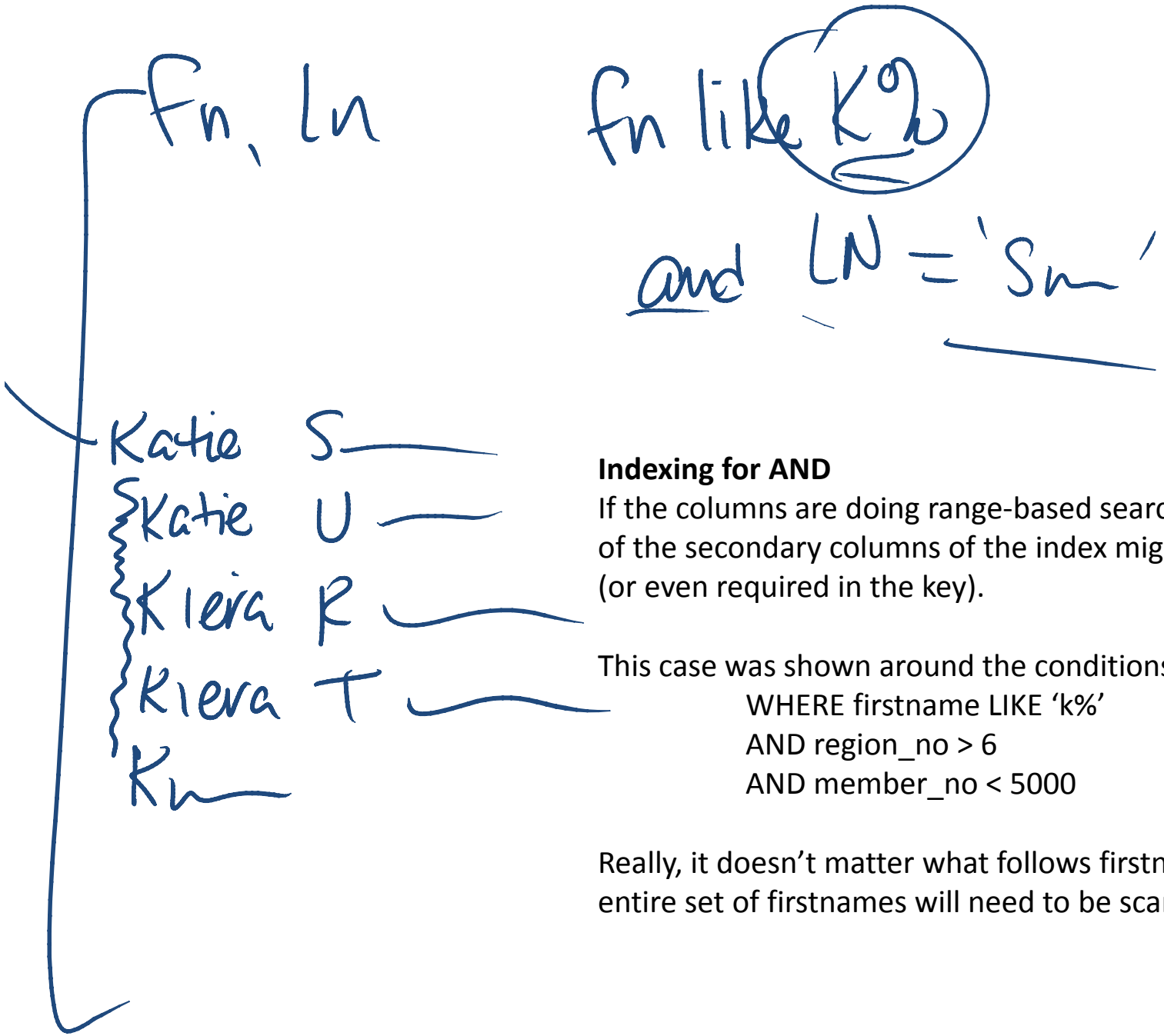
Poor Man's Datawarehousing

Setting a database to RO (read-only) can be a good idea when you plan to use it solely for reads. However, how does SQL Server update statistics or add statistics?

Really, it can't.

So, the main point… if you're going to move a backup to another server for read-only access –you should automate some basic optimizations before setting it to RO.


Module 10
Indexing Strategies
SQL
skills



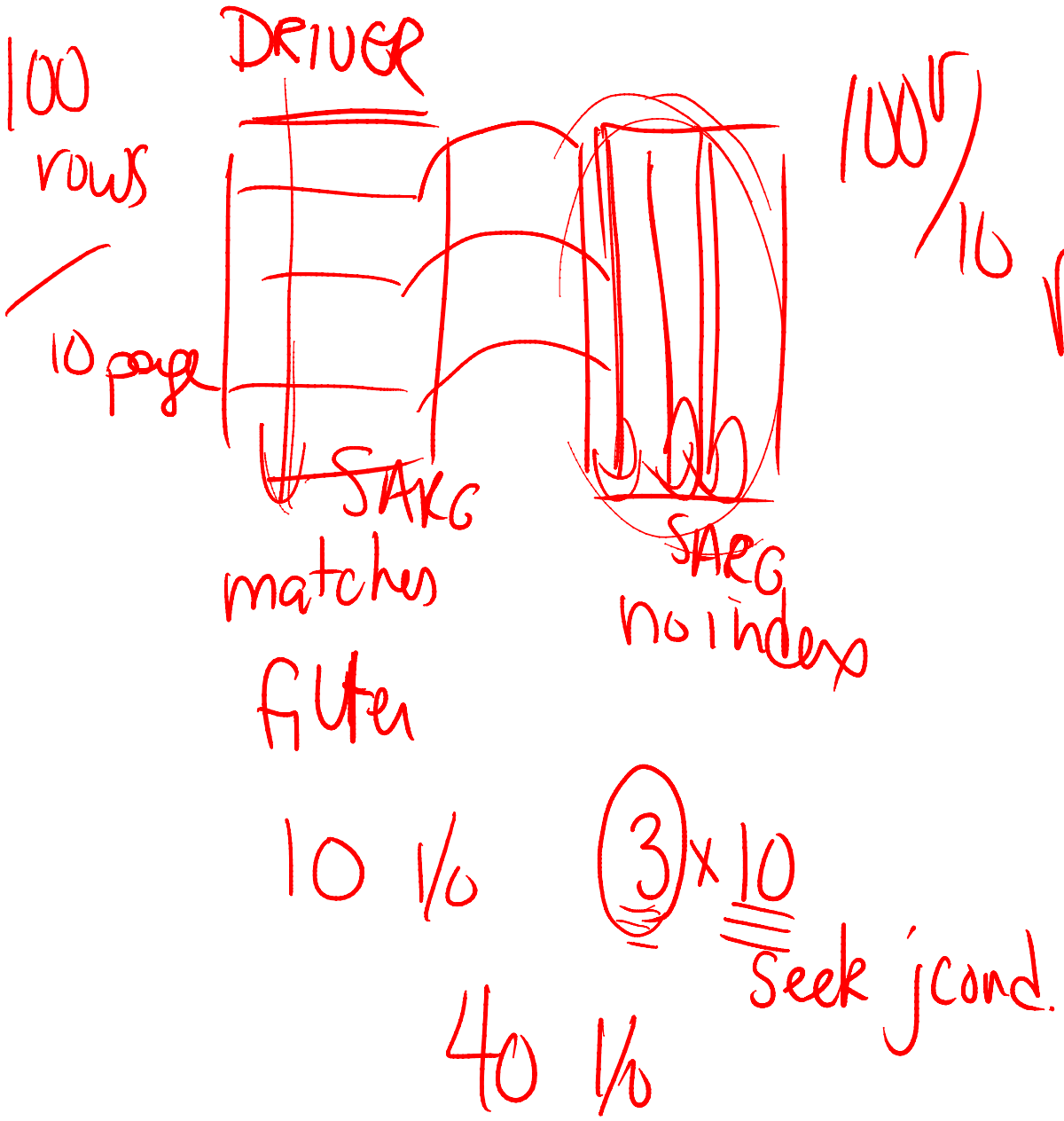
Indexing for AND

If the columns are doing range-based searching then the order of the secondary columns of the index might not be relevant (or even required in the key).

This case was shown around the conditions:

```
WHERE firstname LIKE 'k%'
AND region_no > 6
AND member_no < 5000
```

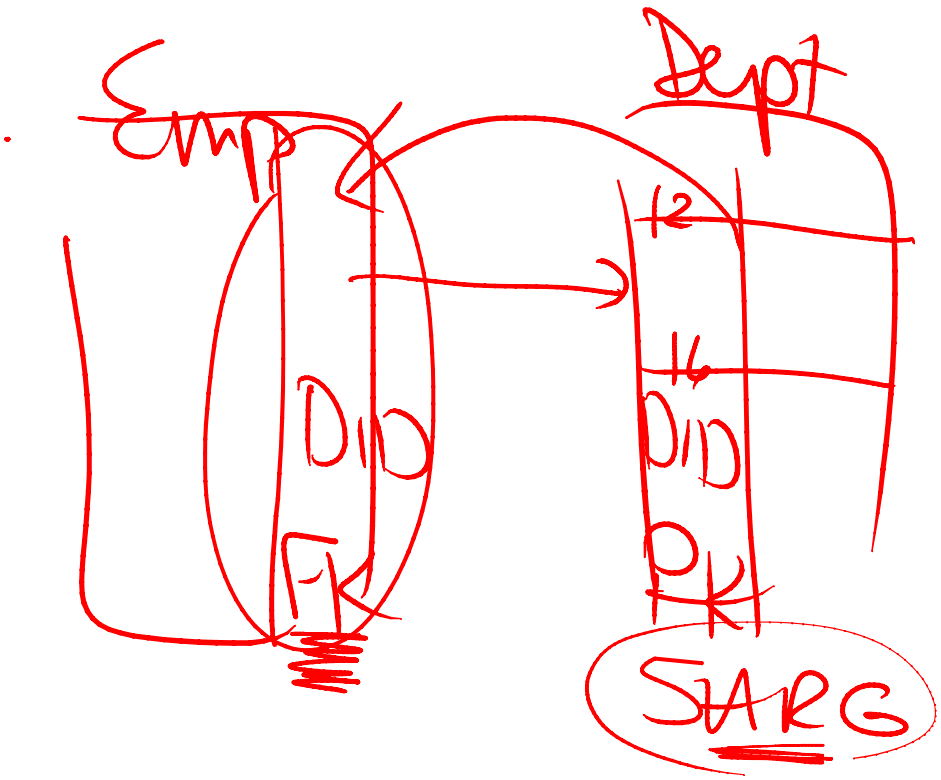
Really, it doesn't matter what follows firstname because the entire set of firstnames will need to be scanned.



M11s20

Here we started to talk about how LOOP joins are an iterative process. The driver (the outer/first table) is typically chosen because it has the most selective set. Any of the tables that have a highly selective search argument are more likely to be chosen as the driver. An index that aids in efficiently finding those rows is REALLY helpful!

Cost can be calculated as:
Number of IOs required for first table +
Number of resulting rows in first table * the cost for each lookup (ideally with an index)



Loop Joins continued

If our query was:

```
SELECT columns (irrelevant here)
FROM Employee AS E
        JOIN Department AS D
                ON E.DID = D.DID
WHERE D.city = 'Redmond'
```

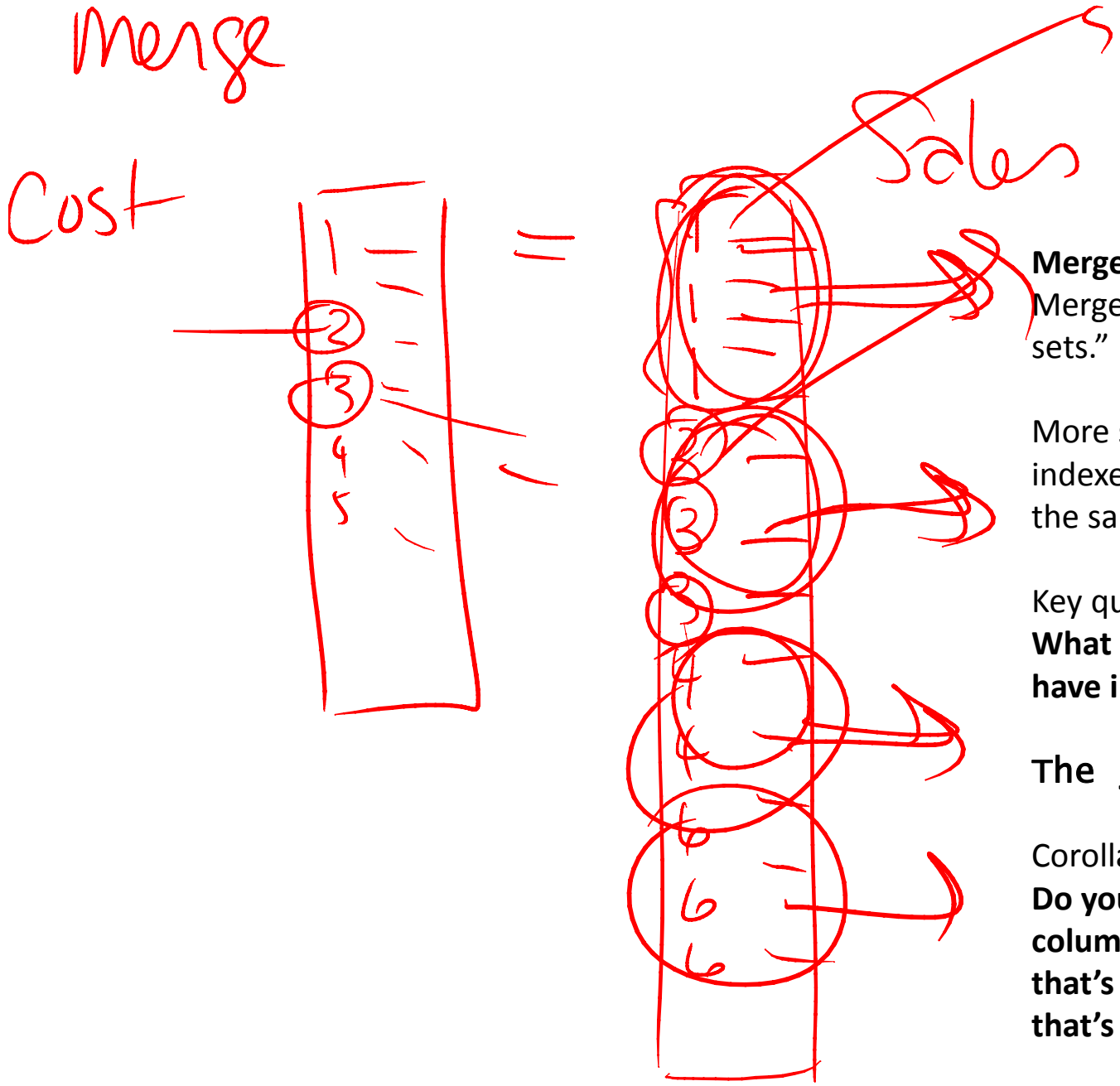
And, there were only 1 or 2 rows for Redmond in the Departments table then that's more likely to be positioned as the driver in the join.

Once we know the DIDs then we're going to need to find all of the employees IN that department. We'll need to search against the DID column in the Employee table.

Key question:

Do you have an index on the foreign key column?

where d.city = 'Redmond'



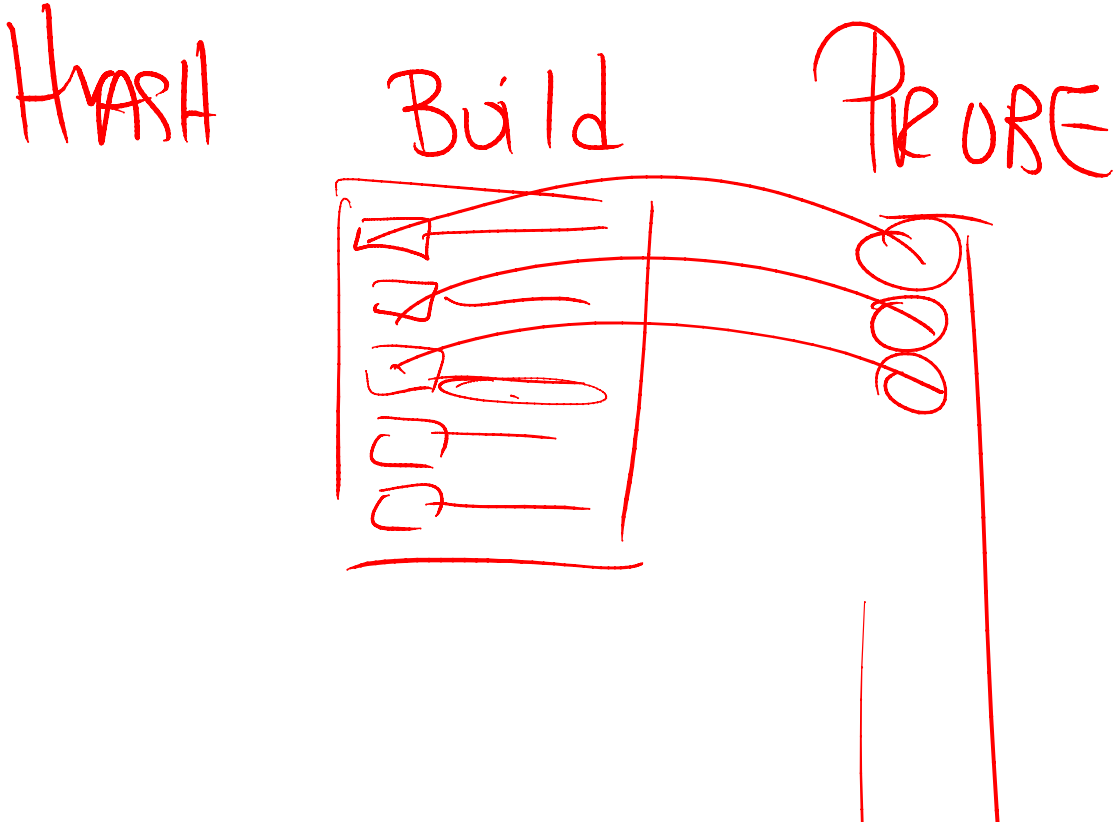
Merge Joins
Merge joins leverage "suitably sorted sets."

More specifically, merge leverages indexes whose leading keys are on the same column.

Key question:
What columns do these two tables have in common?

`The join column…`

Corollary question:
Do you have indexes on EACH of the columns (in each table)? The one that's often missing: the join column that's the foreign key.



Hash Joins

Hash are a little more complicated. There are multiple hash types available in SQL Server and each provide different benefits. The general purpose of a hash join is to significantly reduce the number of rows that have to be processed.

More specifically, there are two phases:

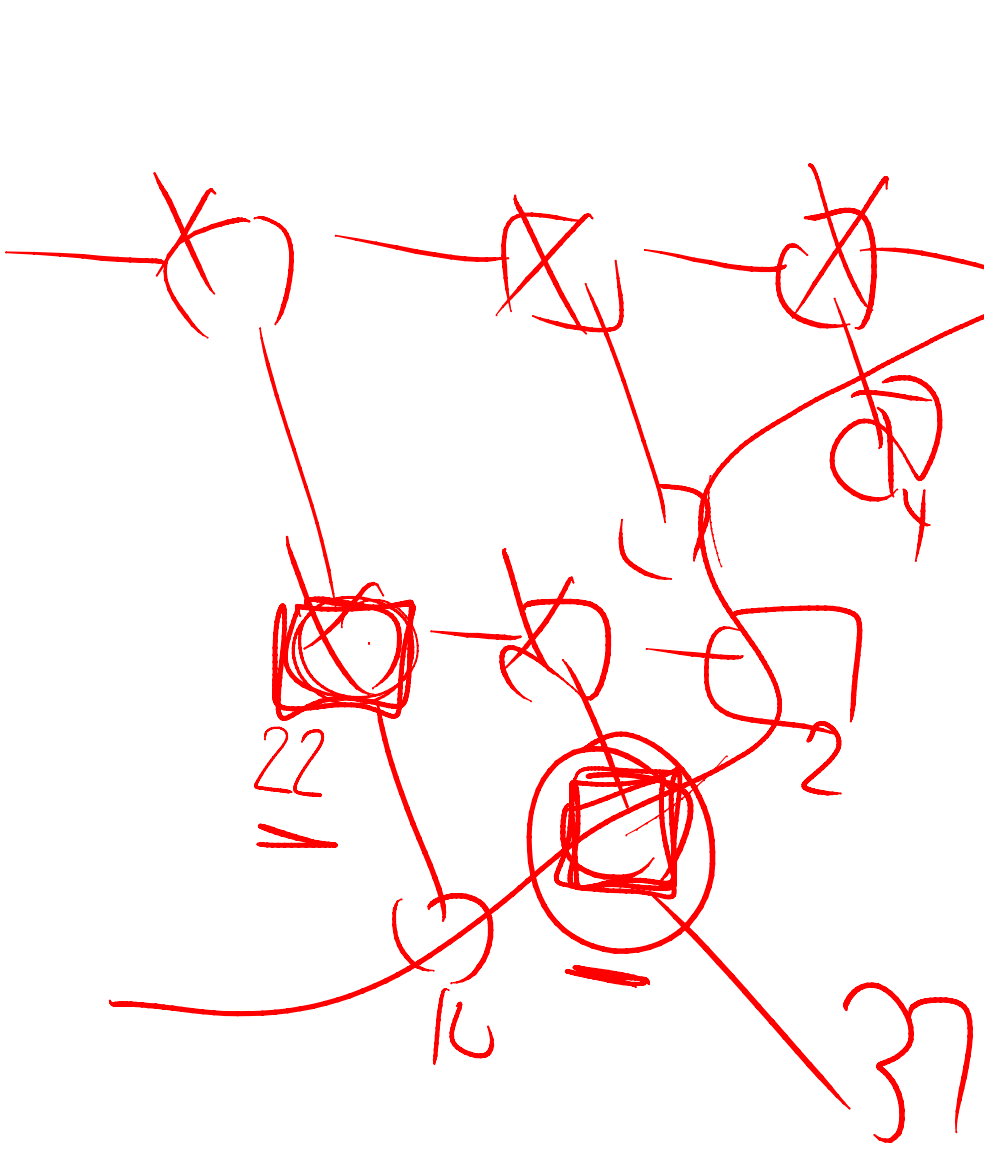
BUILD phase
PROBE phase

The build phase is used to create a small structure into which the larger set can probe to determine if there's the possibility of a matching row.

Two very good resources:

Hash joins and hash teams in Microsoft SQL Server
by Goetz Graefe, Ross Bunker, Shaun Shaun Cooper

Query Evaluation Techniques for Large Databases
by Goetz Graefe



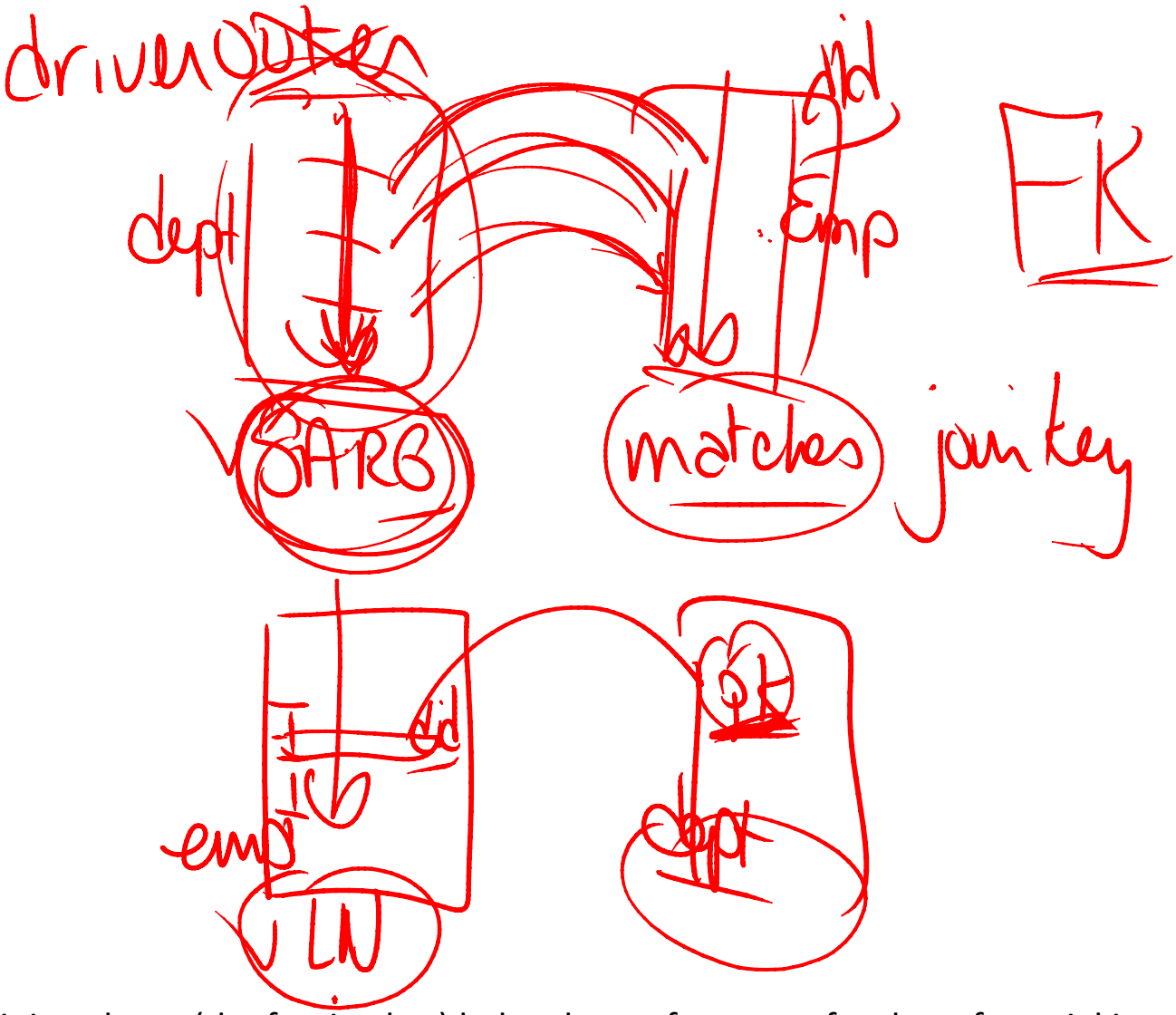
Try to find your:

* Problem Child Table
(the table with the highest cost)

* Problem Child Join
(the join – typically downstream from the table – with the highest cost)

M11s17

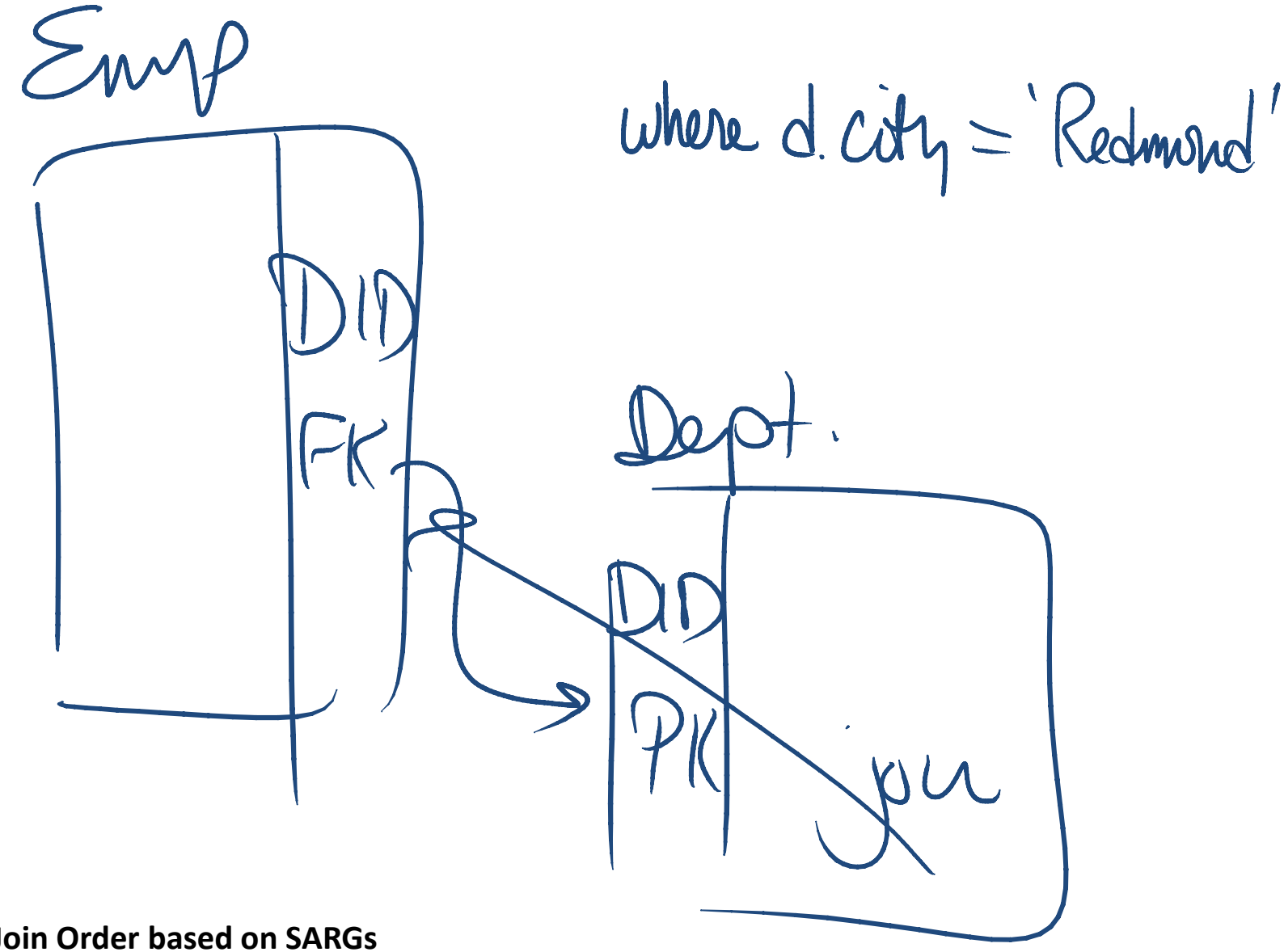
Tuning a large complex join takes breaking it down into smaller chunks. The things that you consider are the costs of the tables (the outer most events) and the costs of the joins. And, typically, the most expensive join is downstream from the most expensive table.



M11s20

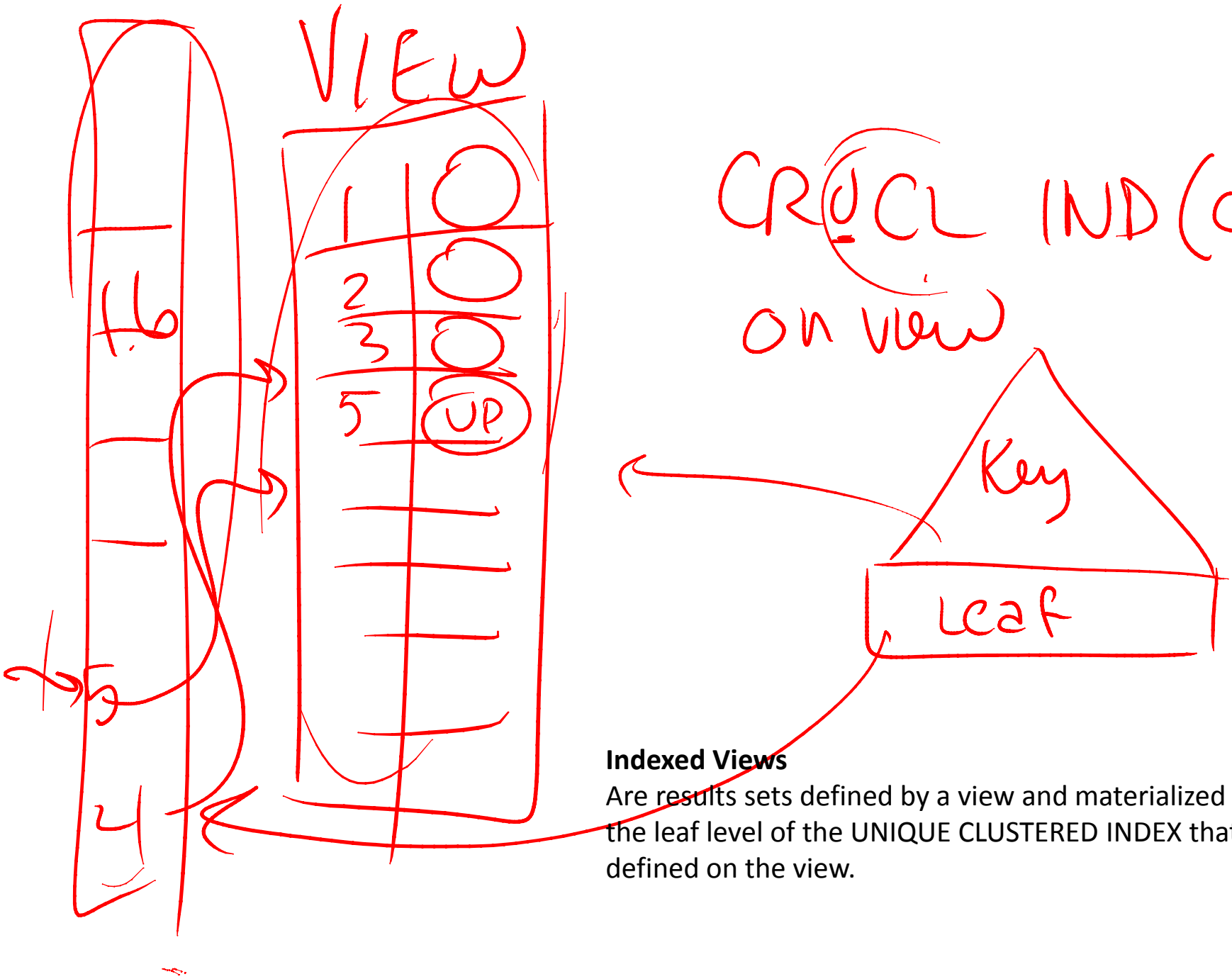
An index on a join column (the foreign key) helps the performance for the referential integrity as well as *some* joins. A narrow index on the join column helps but is often superseded by wider indexes; it's still a good start. (hence the phase – phase **1**)

This is the most ideal for (but not limited to) a loop join.



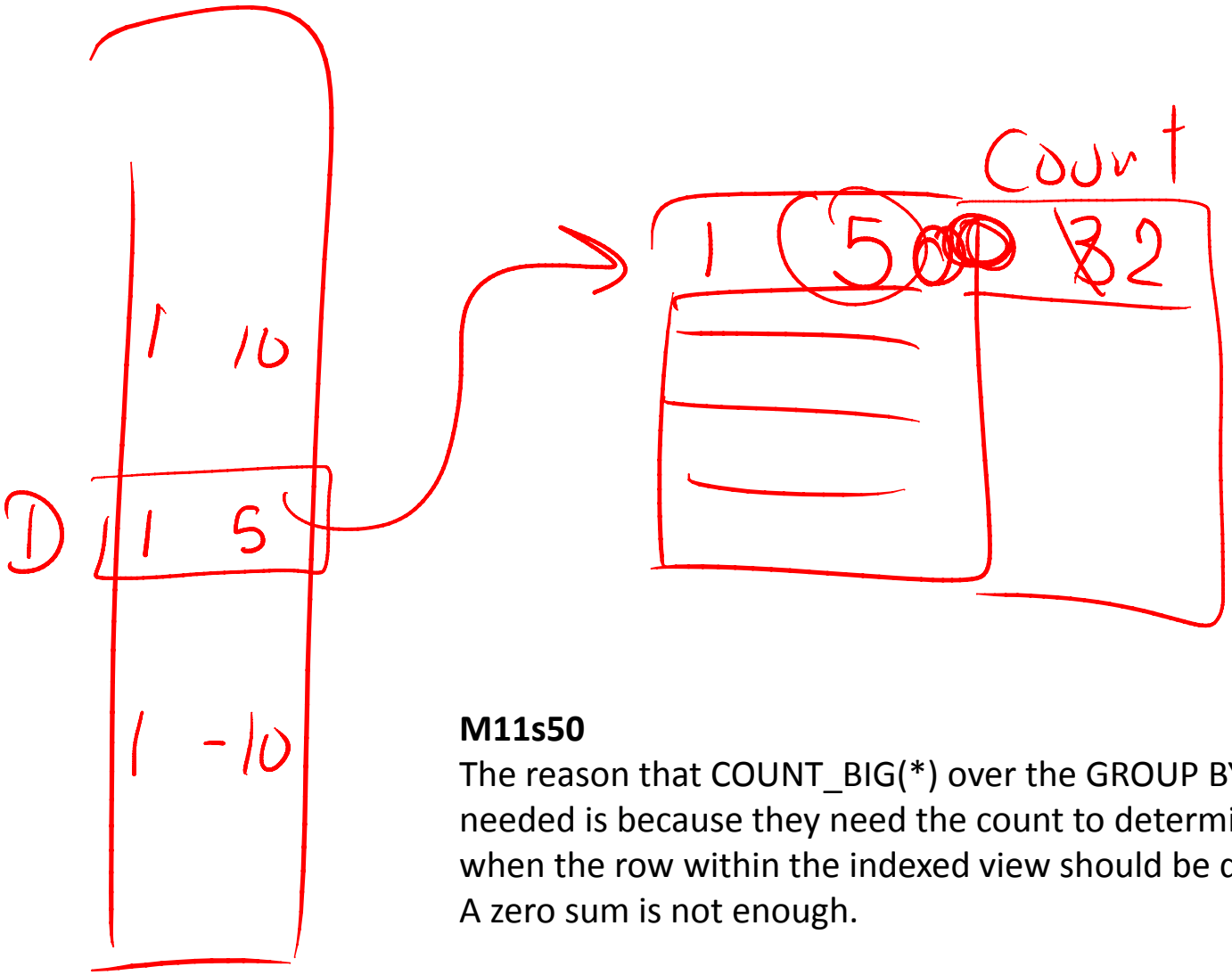
Changing Join Order based on SARGs

An index on a join column (the foreign key) helps the performance for the referential integrity as well as some joins because your more selective criteria might be on the referencing table rather than the referenced.



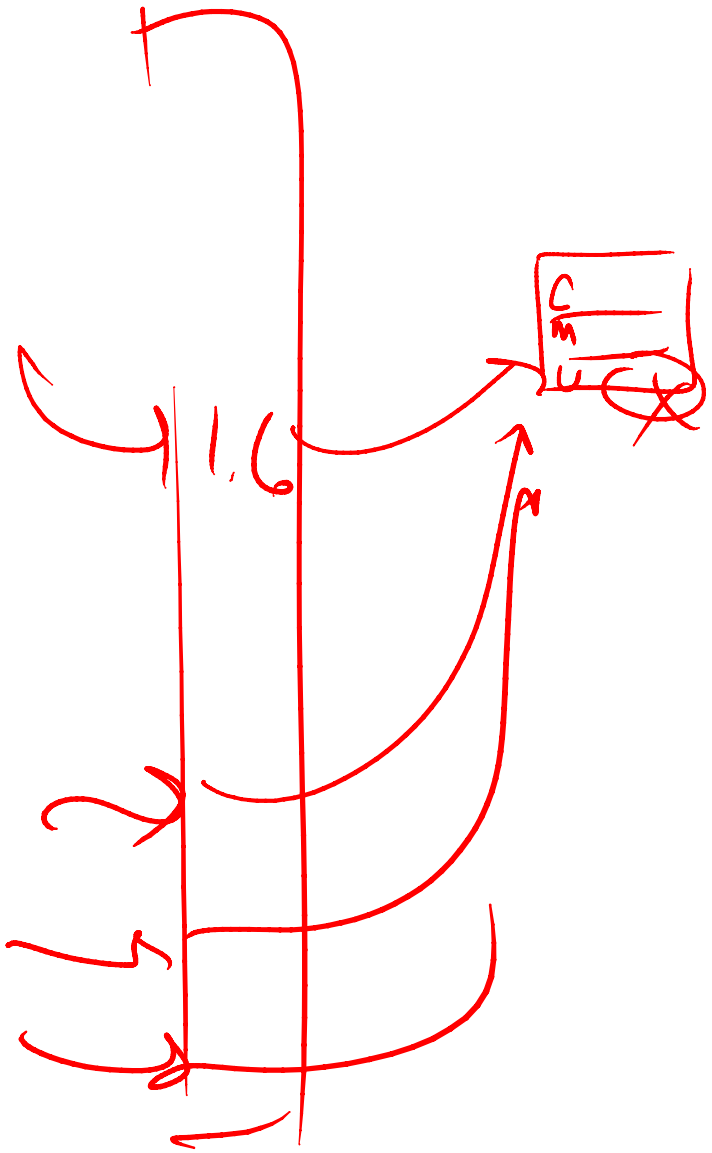
Indexed Views

Are results sets defined by a view and materialized into the leaf level of the UNIQUE CLUSTERED INDEX that's defined on the view.



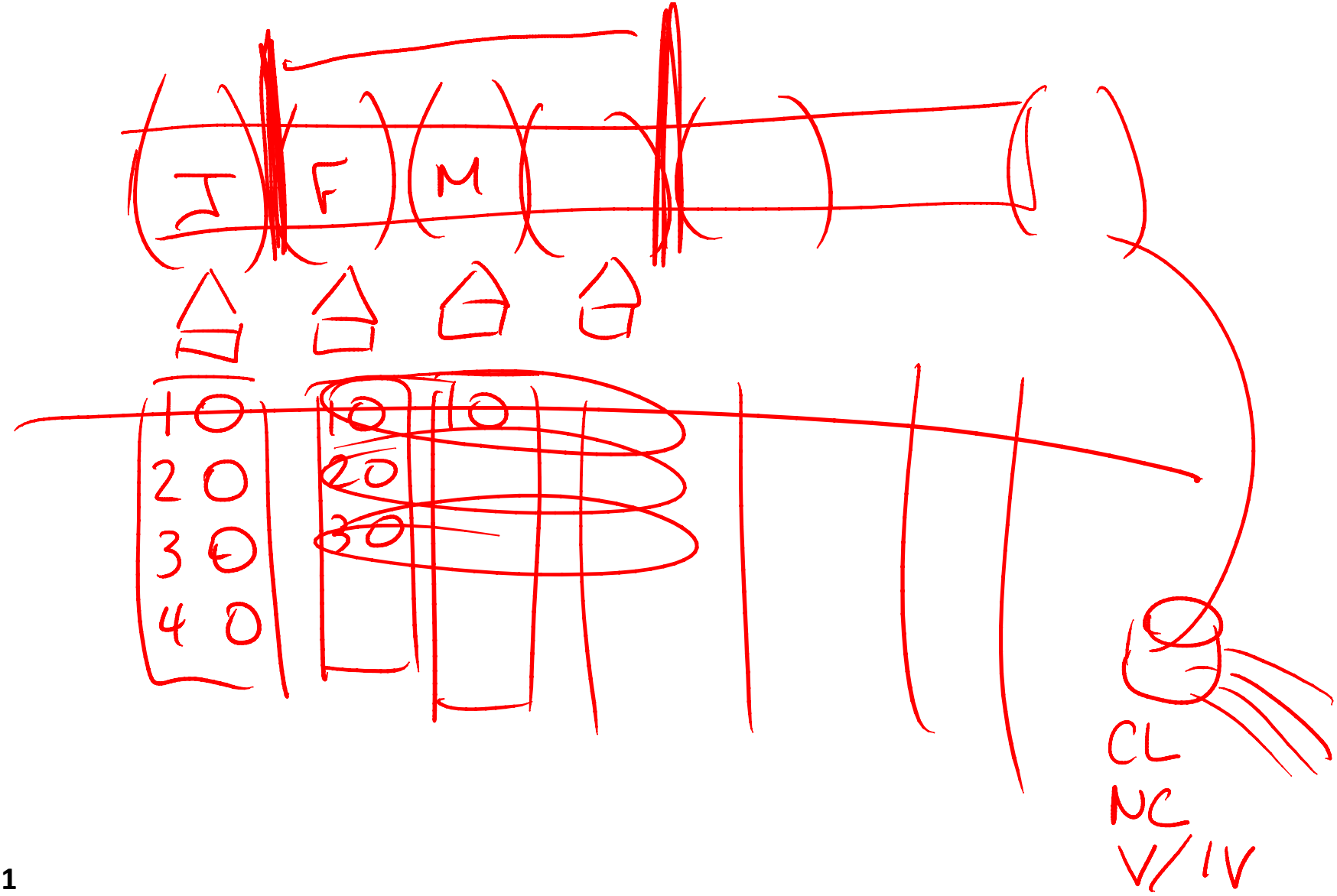
M11s50

The reason that COUNT_BIG(*) over the GROUP BY class is needed is because they need the count to determine when the row within the indexed view should be deleted. A zero sum is not enough.



M11

If your aggregate is too small then you can have a HOT ROW problem where all modifications are blocked trying to write to the aggregate. You'll serialize your inserts by country here… CAN, MEX, USA are the only countries with whom you do business – all US rows will have to wait as each updates the sum. This will become a terrible bottleneck.

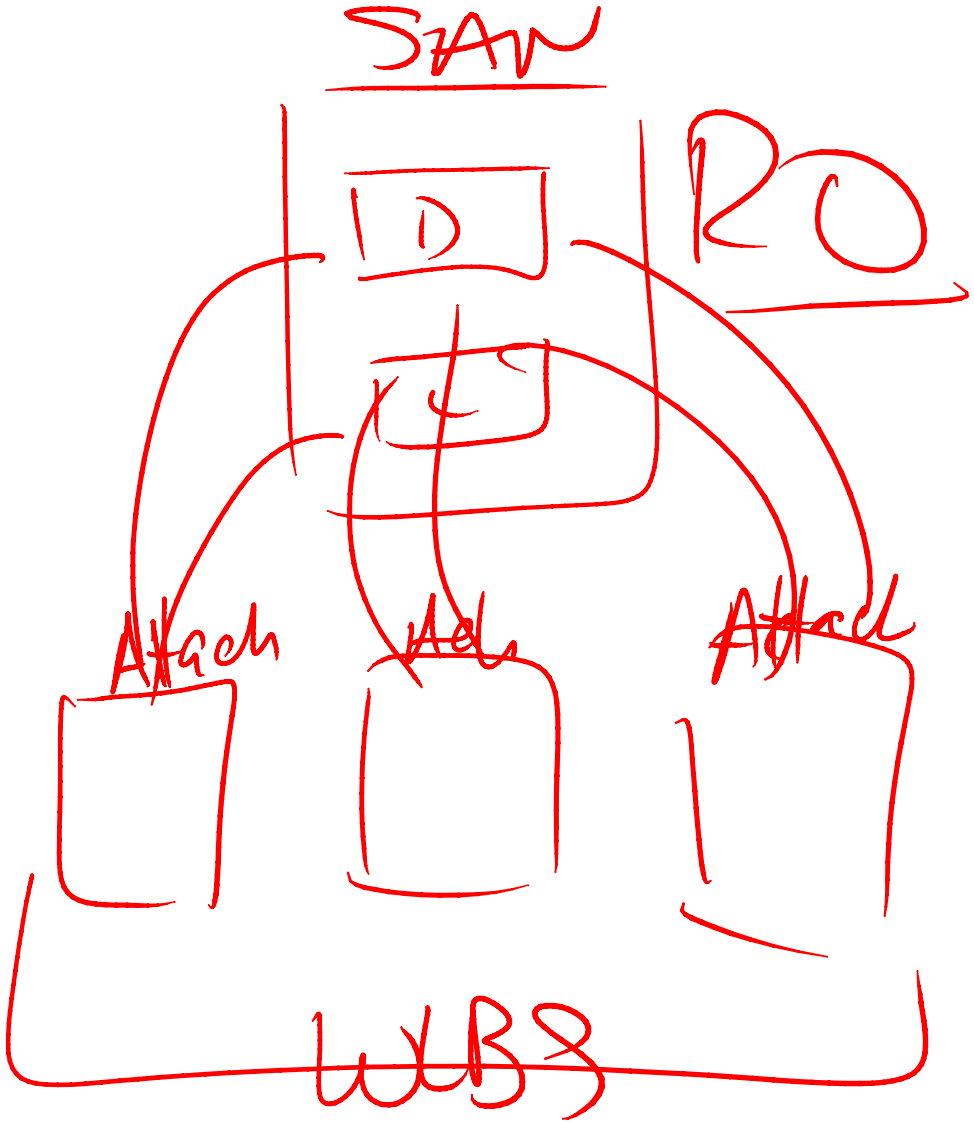


M11

If you create a partition aligned indexed view and then request something like the sum(sales) for customer #1 then SQL Server can aggregate the aggregates!

On SQL Server 2008, when you're switching in – you must defined the Views and IVs in order to successfully switch in.

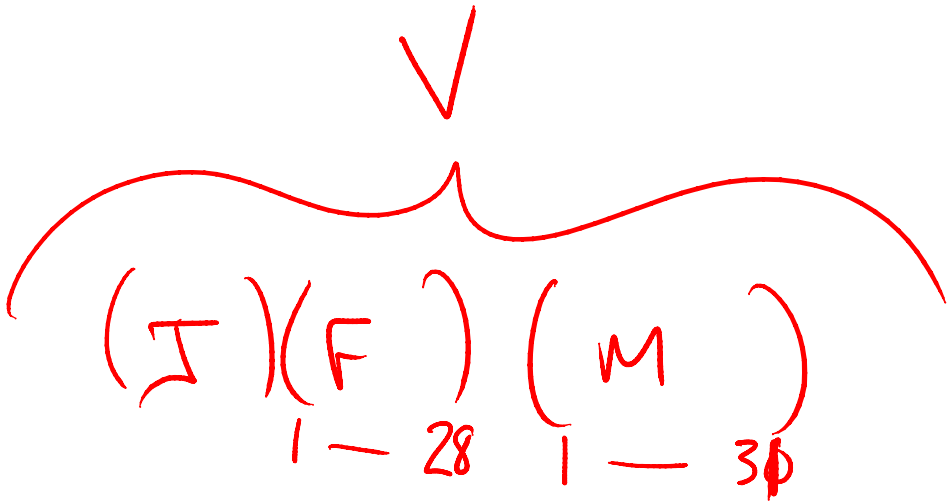

Module 12
Table Partitioning
Partitioned Views & Partitioned Tables
SQL
skills



M12

Partitioning in our module was all within a single database. However, SQL Server does support Scalable Shared Databases. These are RO databases that are attached to multiple servers and then balanced through WLBS.

Scalable Shared Databases

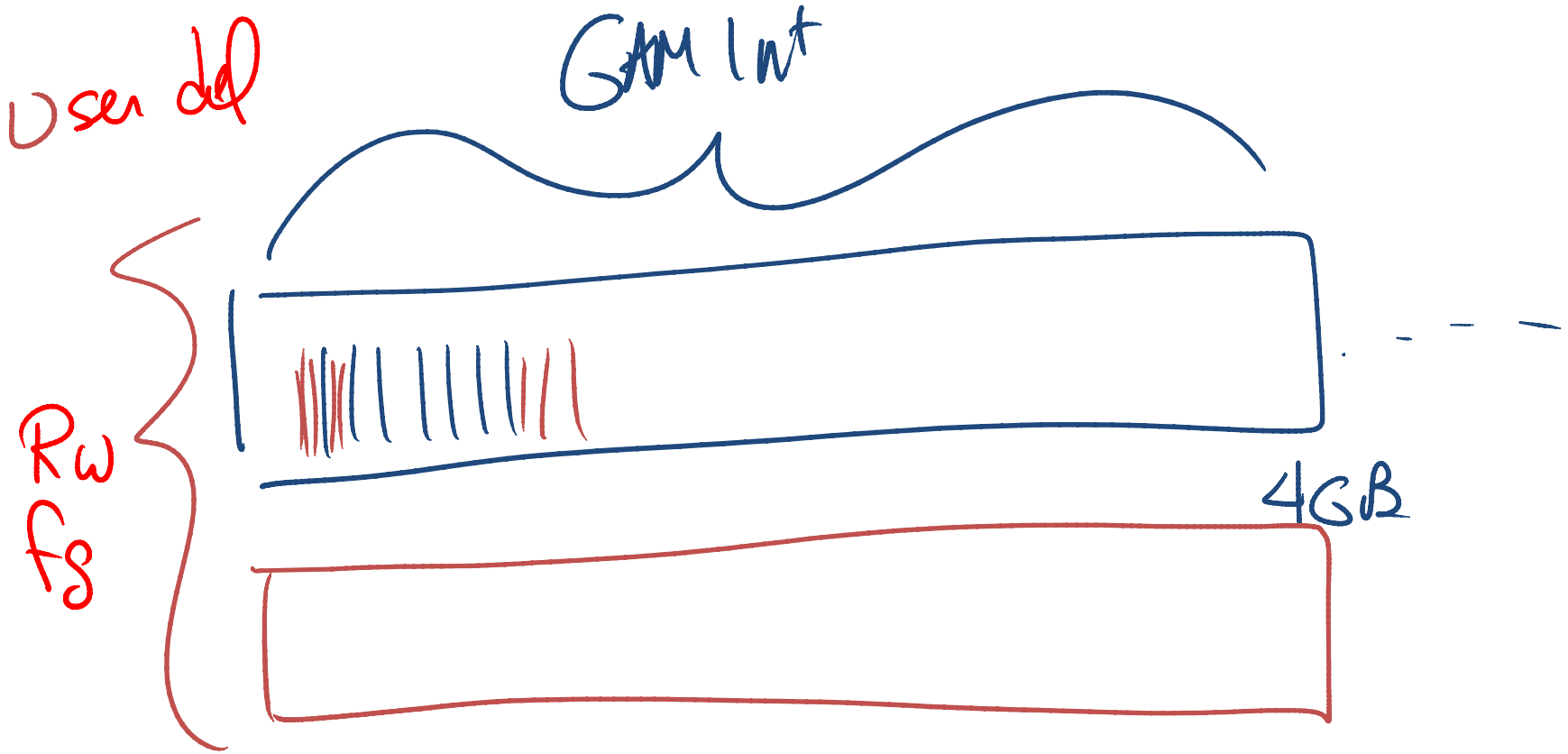


M12

Be careful with partitioned views… there's nothing that's going to test your business logic. As a result, if you miss a day (for example Feb 29, 2008) then inserts into the view will fail because there's no view that can store that date.

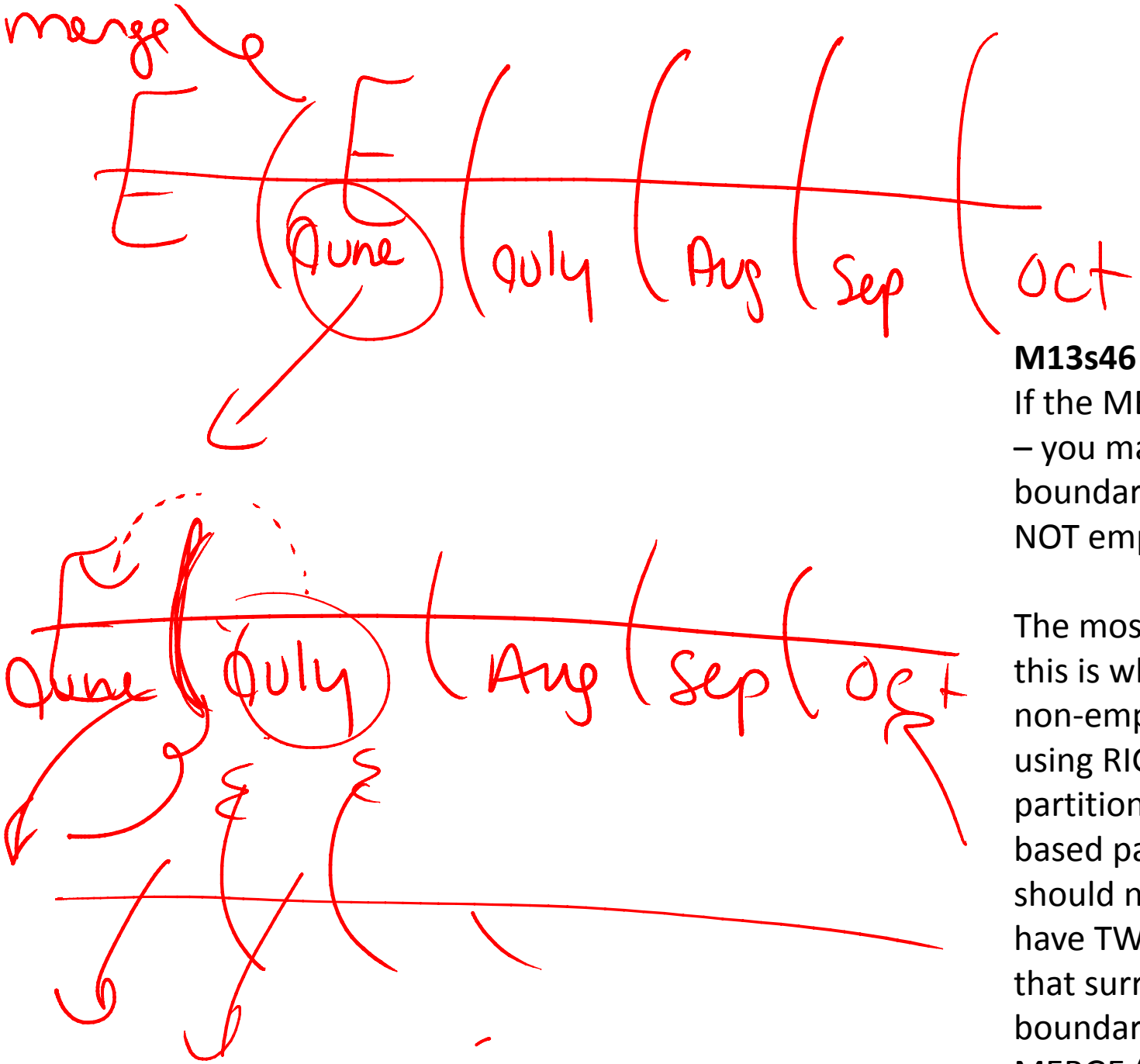
Alternatively, using partitioned tables – there's no way to have a gap or overlapping ranges.

But, there's a lot more to PVs and PTs (this isn't enough to choose one over the other) so this is just a bit more info to add to the list!



M12

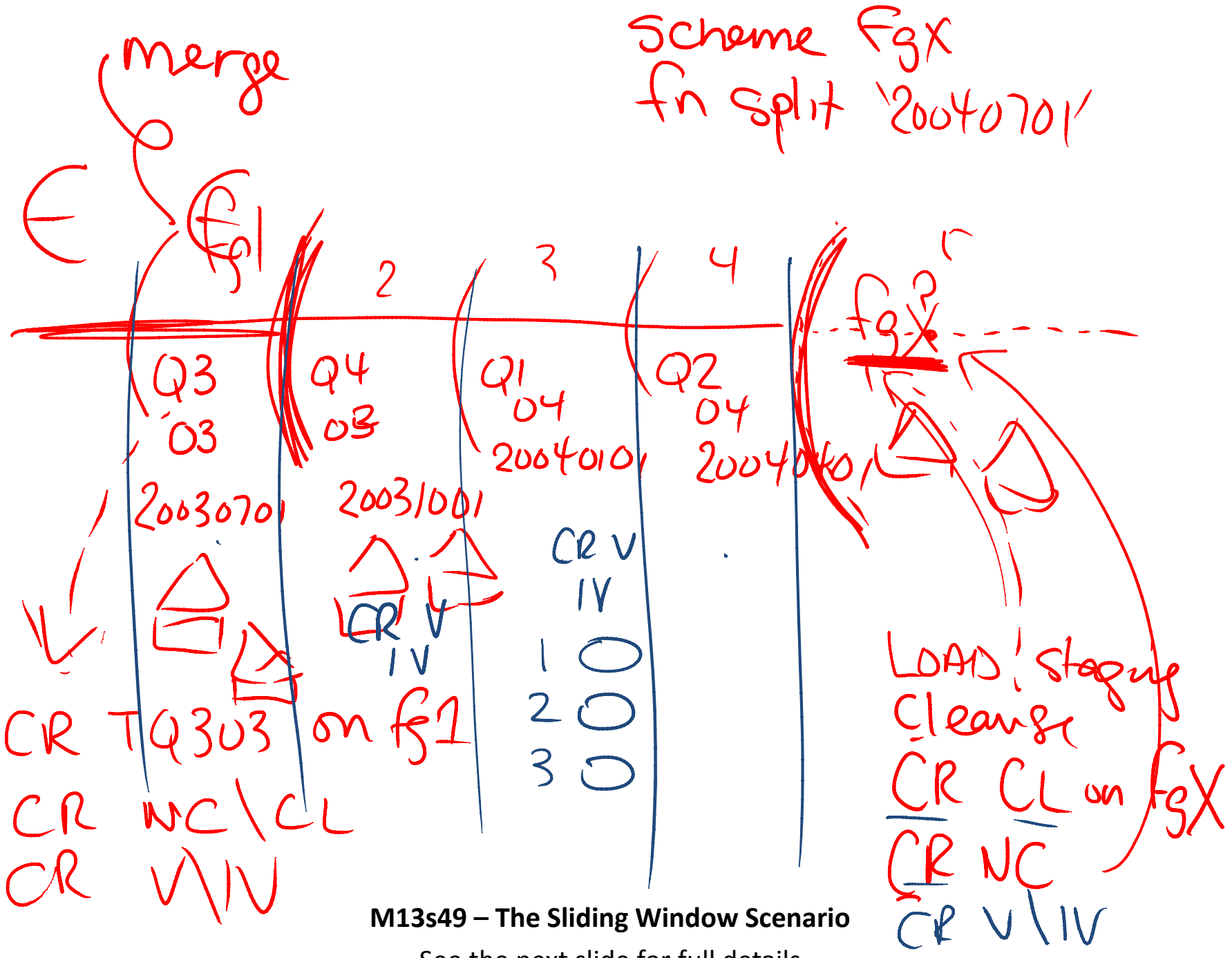
If a RW filegroup has a lot of allocations then you can have contention (much like that in tempdb). One of the best strategies is to make sure that all RW filegroups have at least 2 files and as many as four (as a default practice). You might need more but just moving from 1 is already going to eliminate many problems.



M13s46

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).



M13s49 – 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

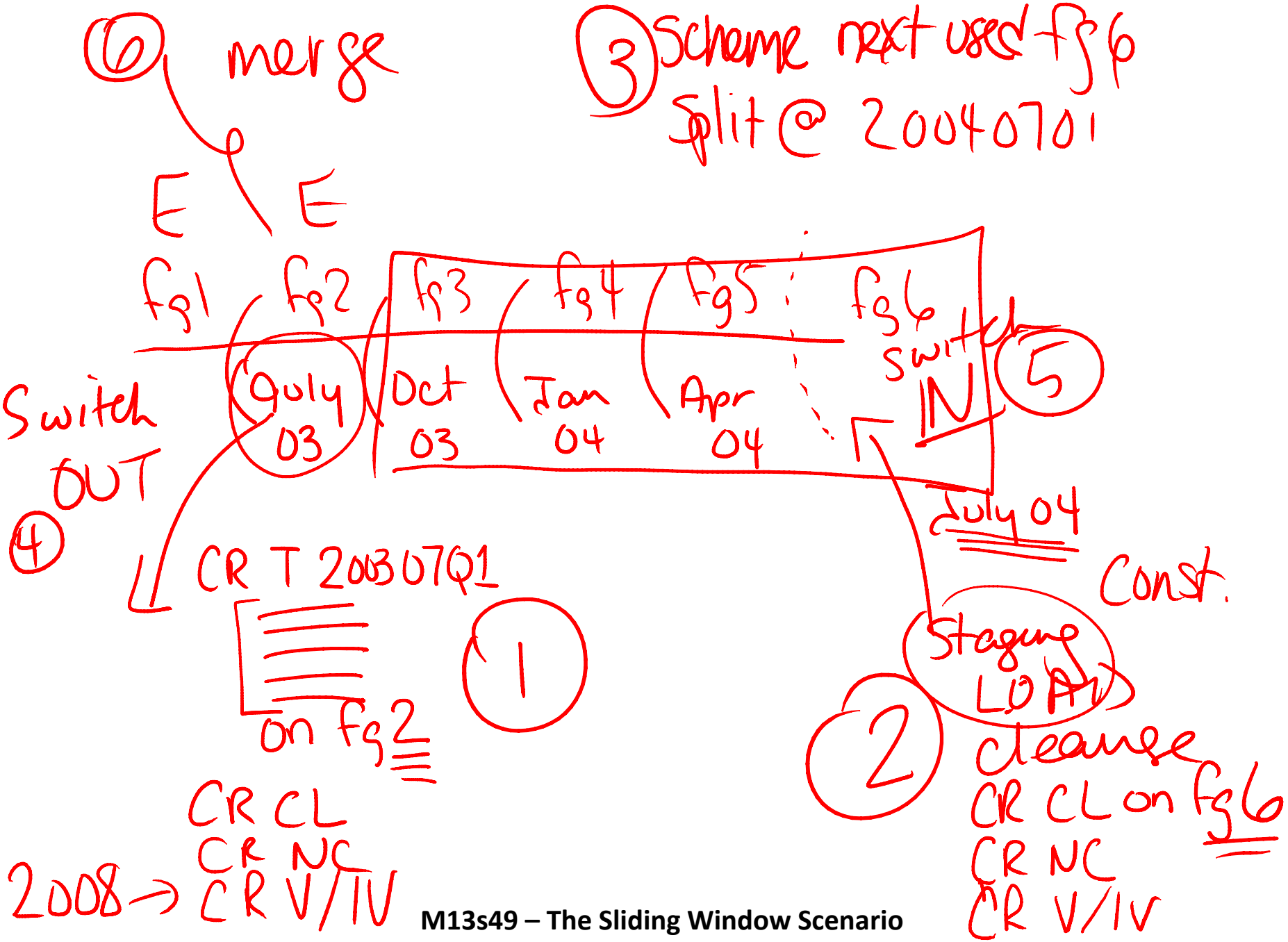
- Now, you're ready

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

(6) Clean up

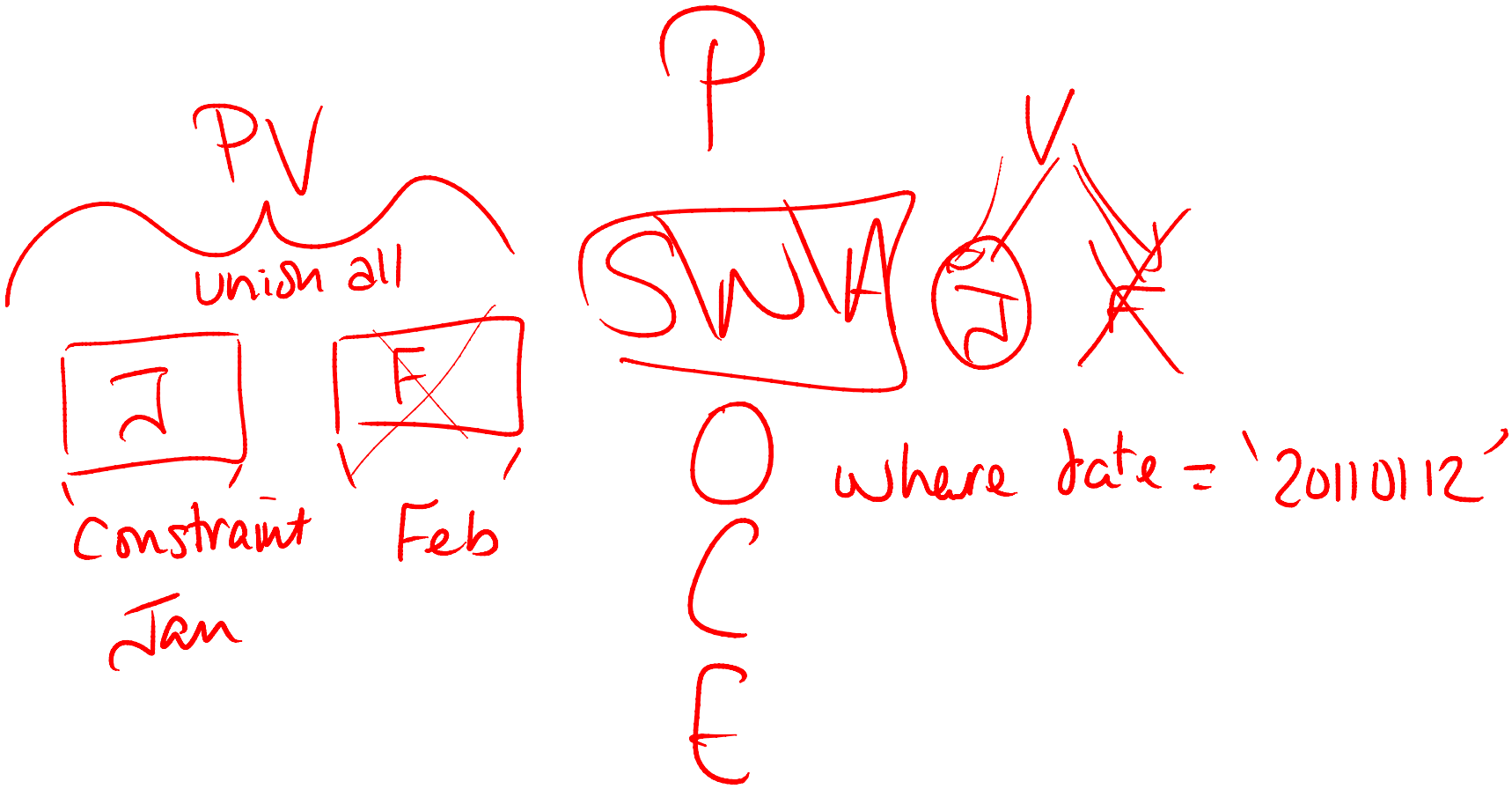
Merge the boundary point but ONLY if it's empty (the next picture will remind you of what happens when you don't MERGE an empty boundary point)

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



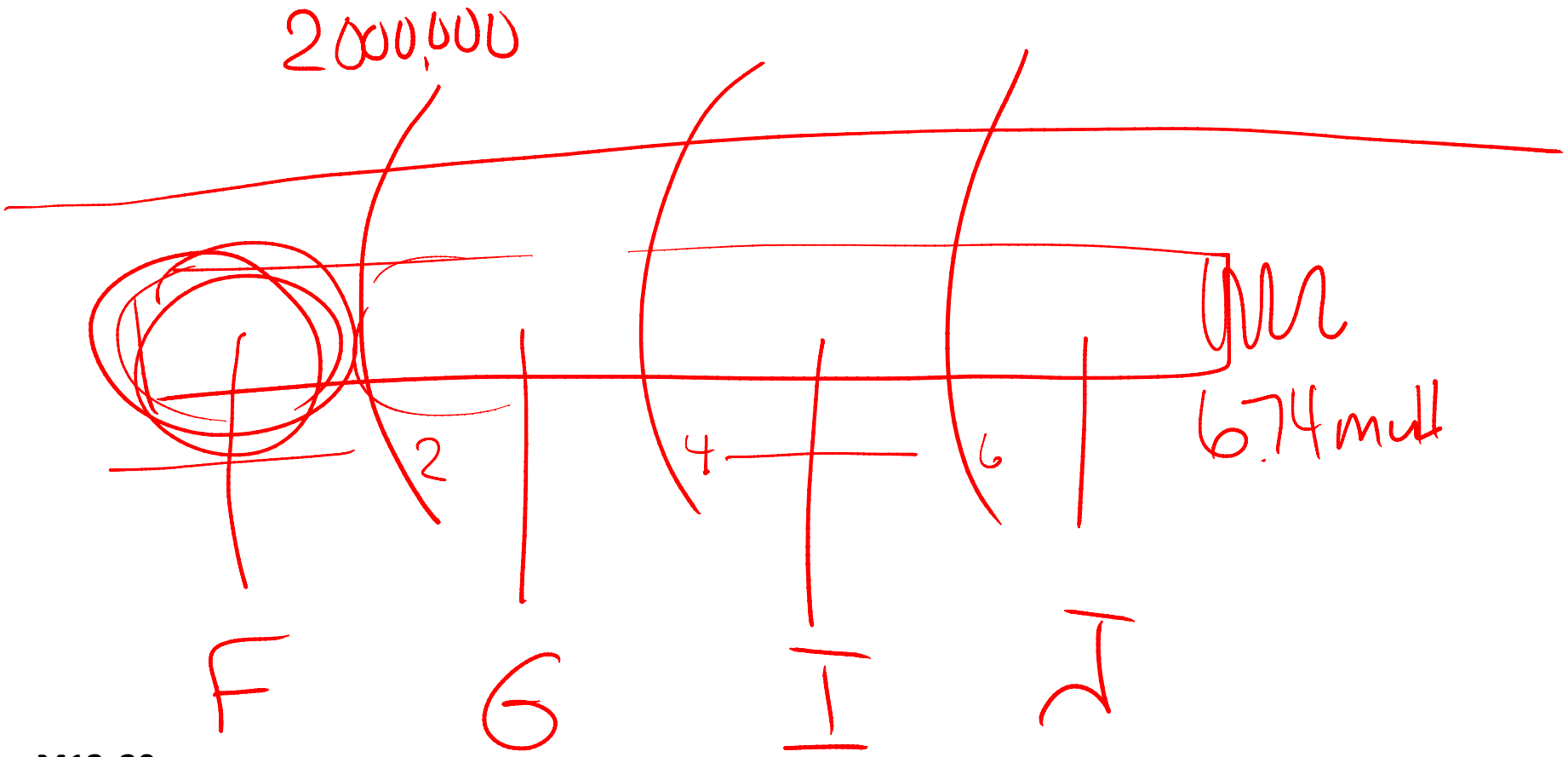
M13s49 – The Sliding Window Scenario

This picture is from another class. I thought I'd include it because it corresponds to the numbers…



M13s17

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. ☺)



M13s20

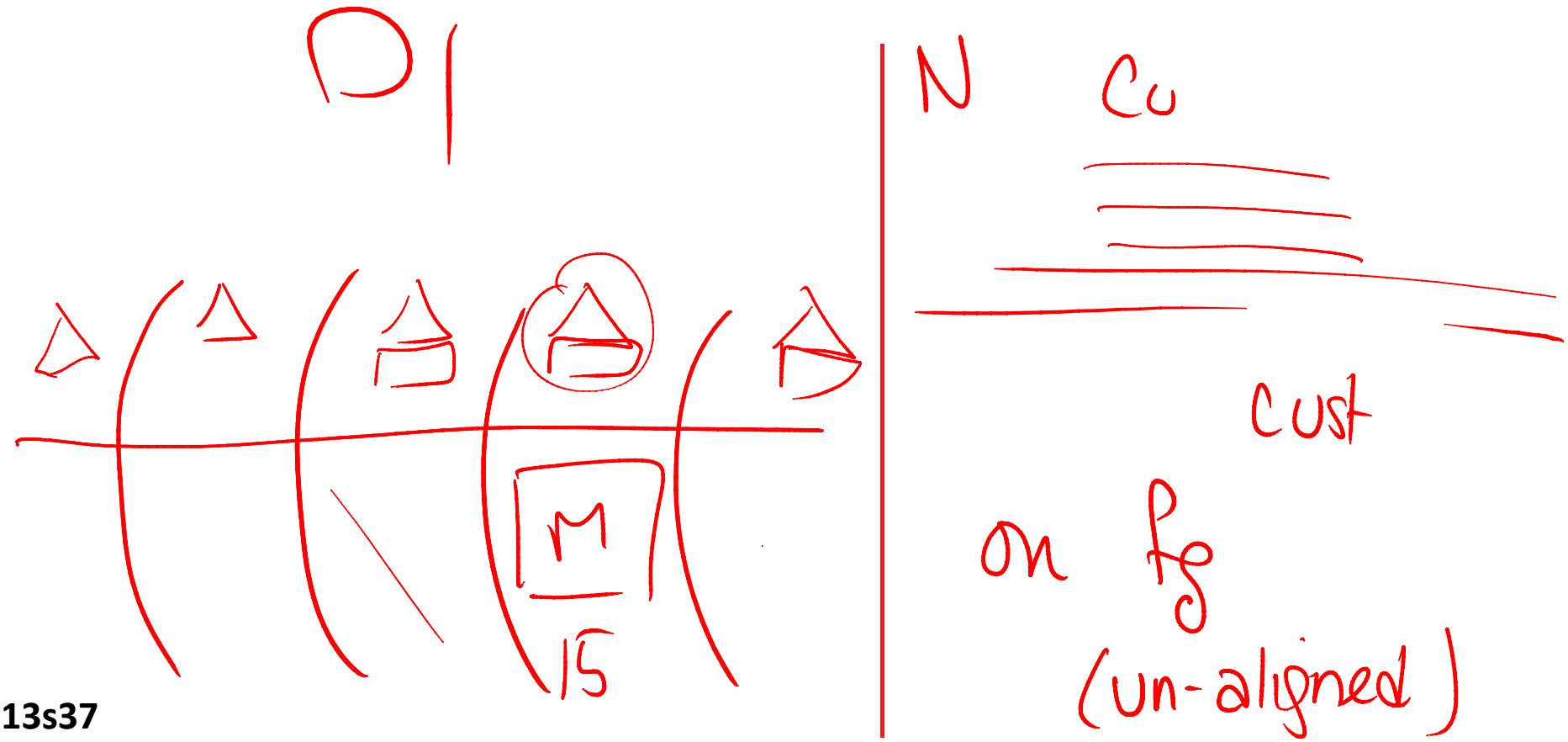
Another example of a partitioned table with 3 boundary points: 2million, 4 million and 6 million. It's a RIGHT-based partition function so the rows that match the boundary point will be to the RIGHT side. Specifically this translates into:

In partition 1: all rows less than 2 million

In partition 2: all rows equal to or greater than 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



M13s37

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, **<u>if you're not interested in fast switching</u>** – 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.

M13s43

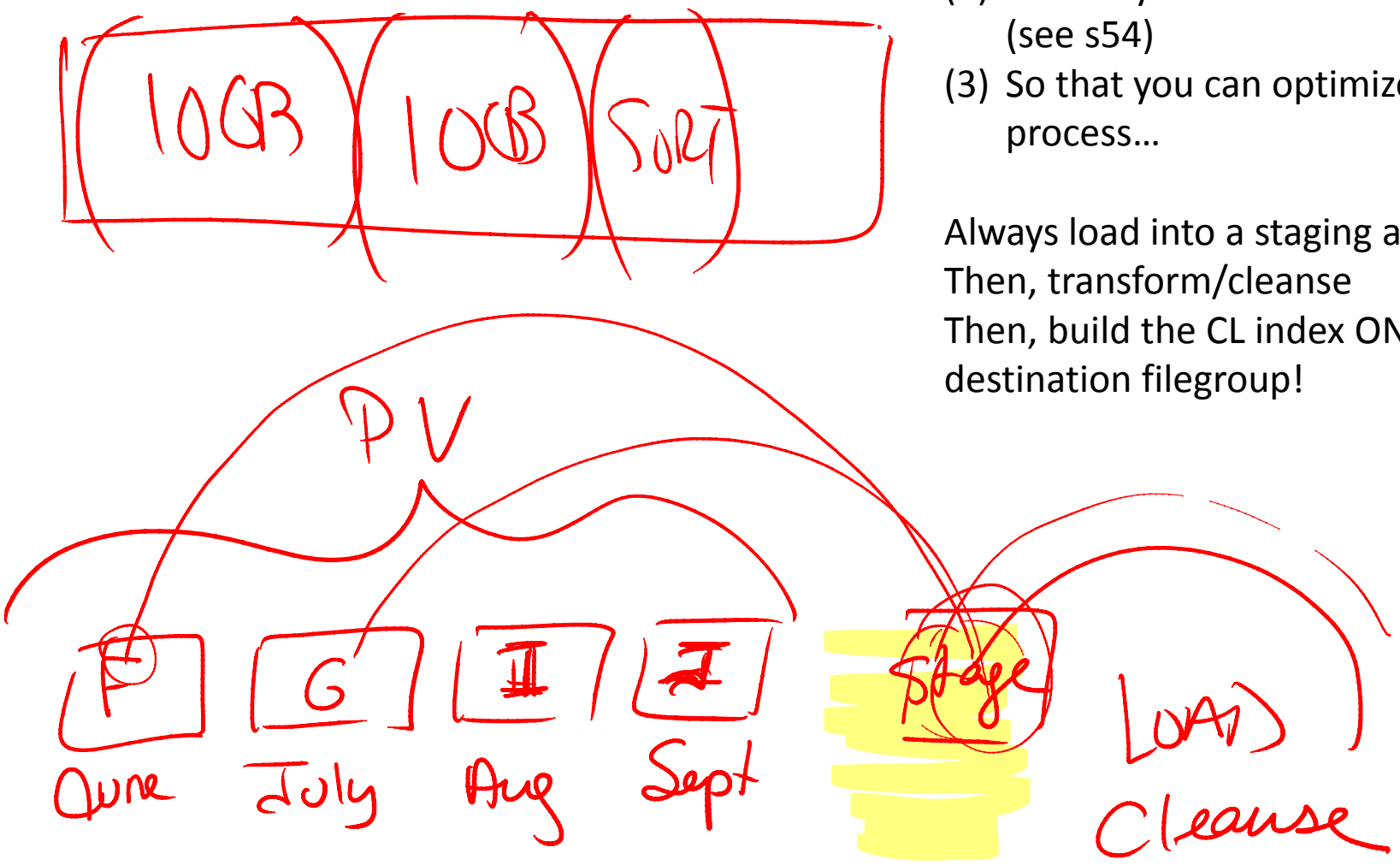
Why do you need a staging area?

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

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

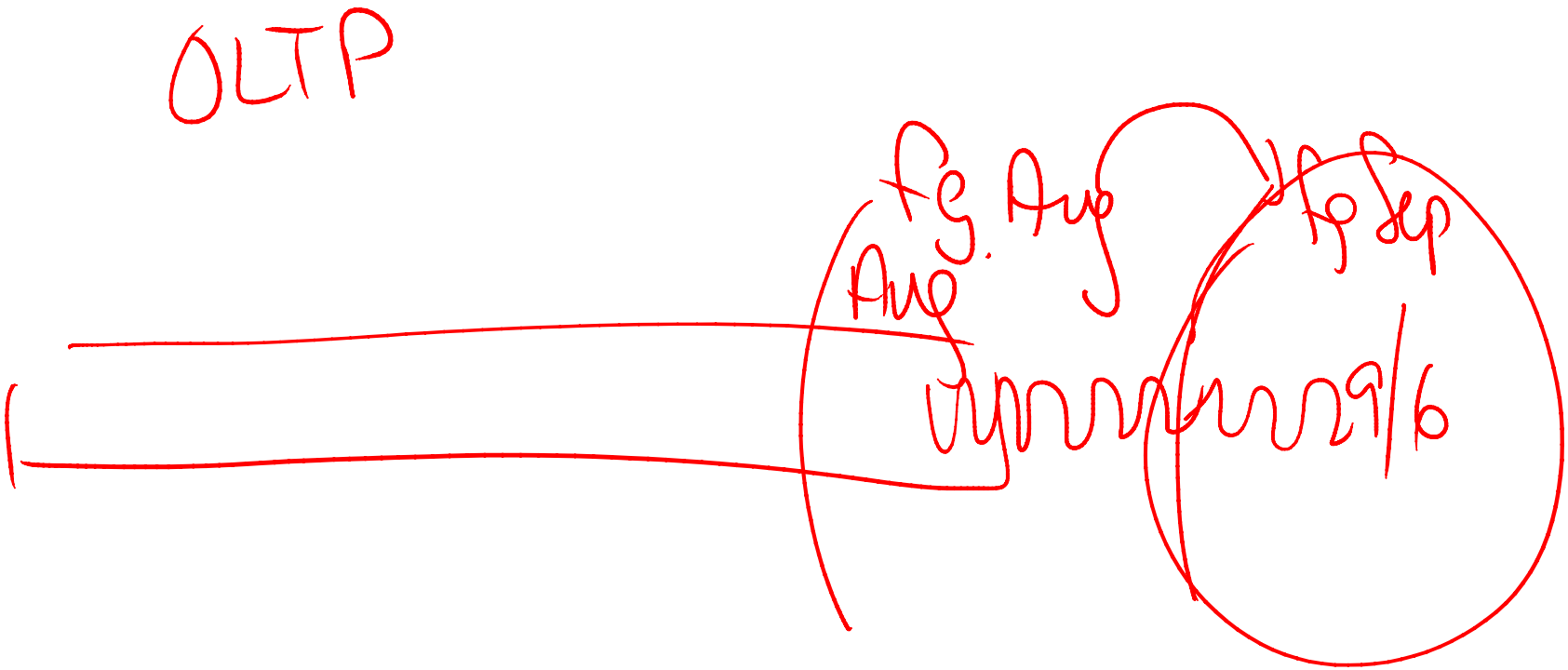
(3) So that you can optimize the process…

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



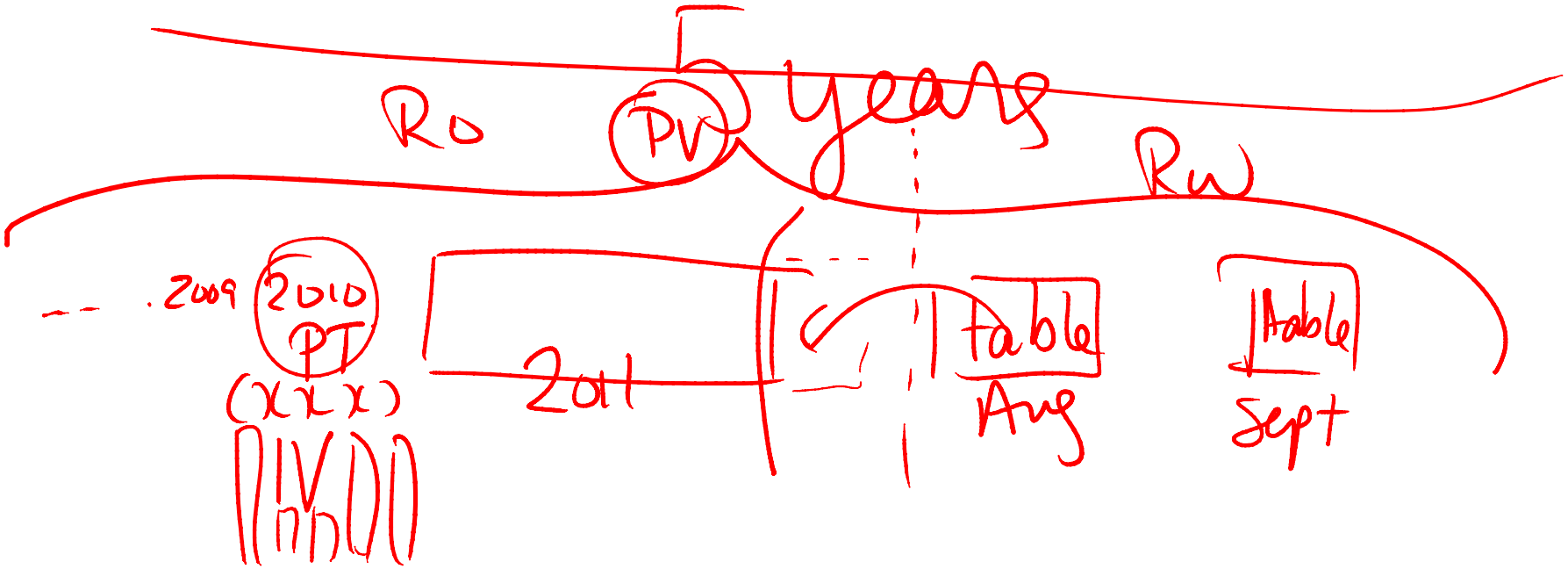
M13s48

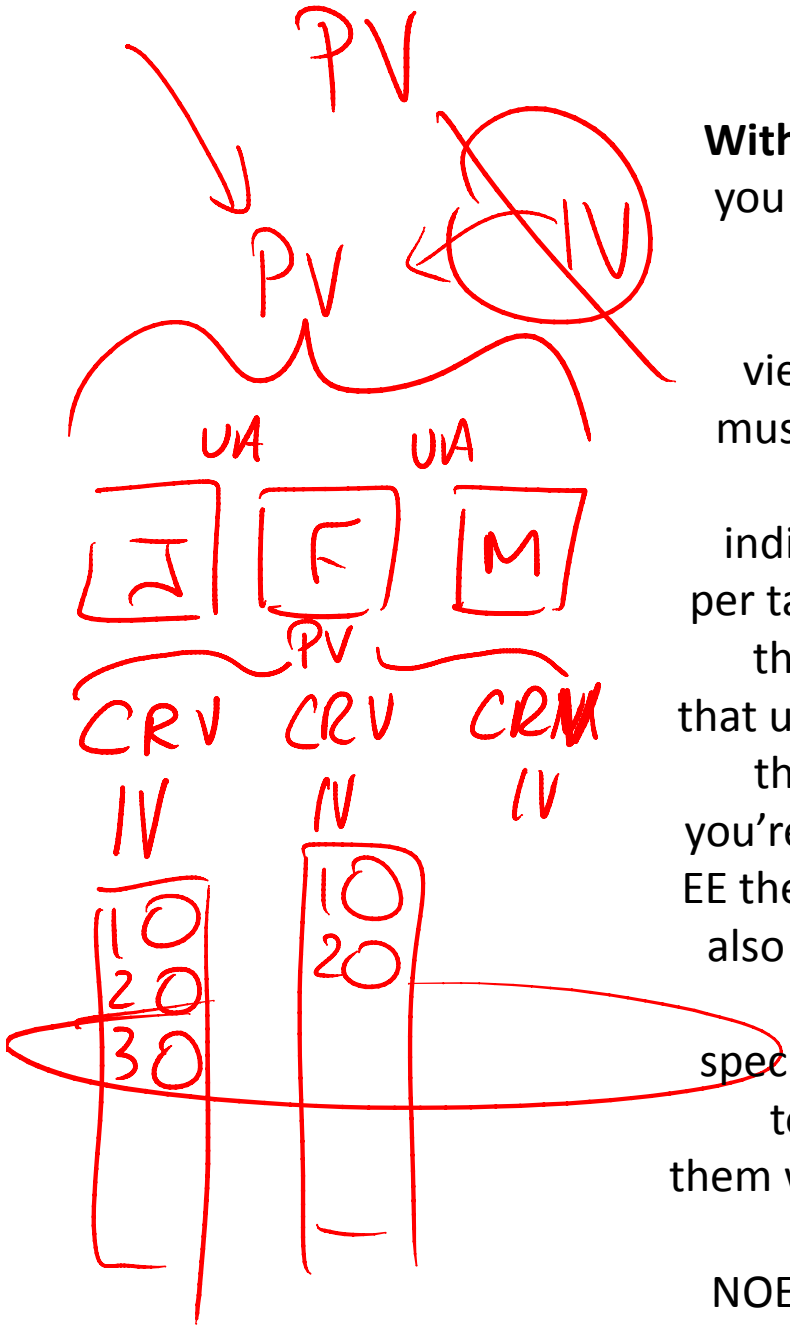
With PTs – it's likely that your last partition (often, the current data) will be active. This is also where it's most likely that you'll have fragmentation. If you want to rebuild ONLY that last partition – you'll need take it offline to do it.



M13s48 (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!

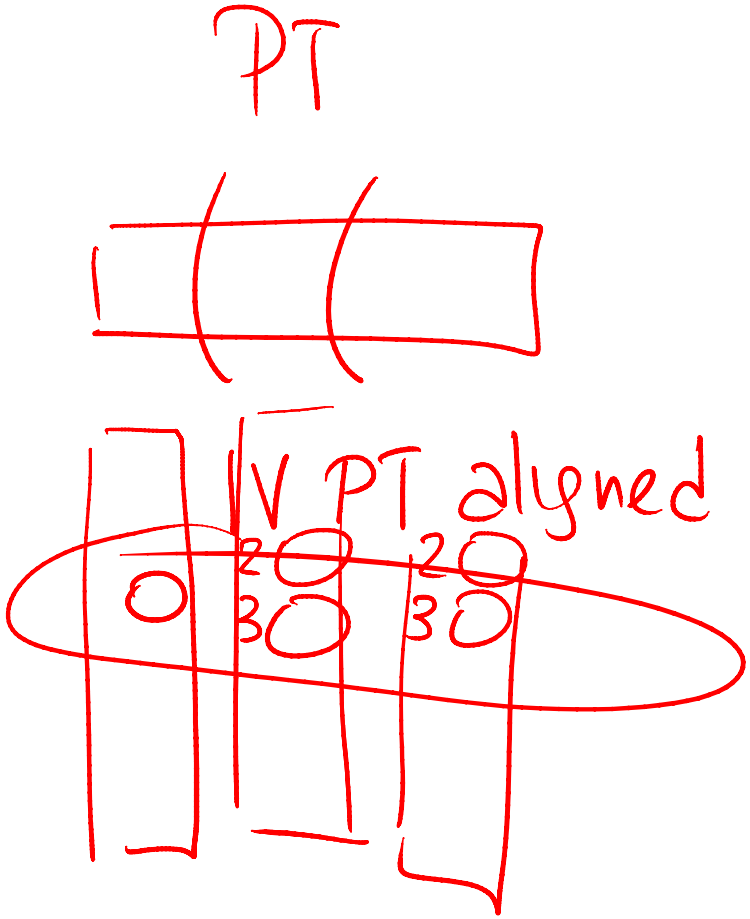


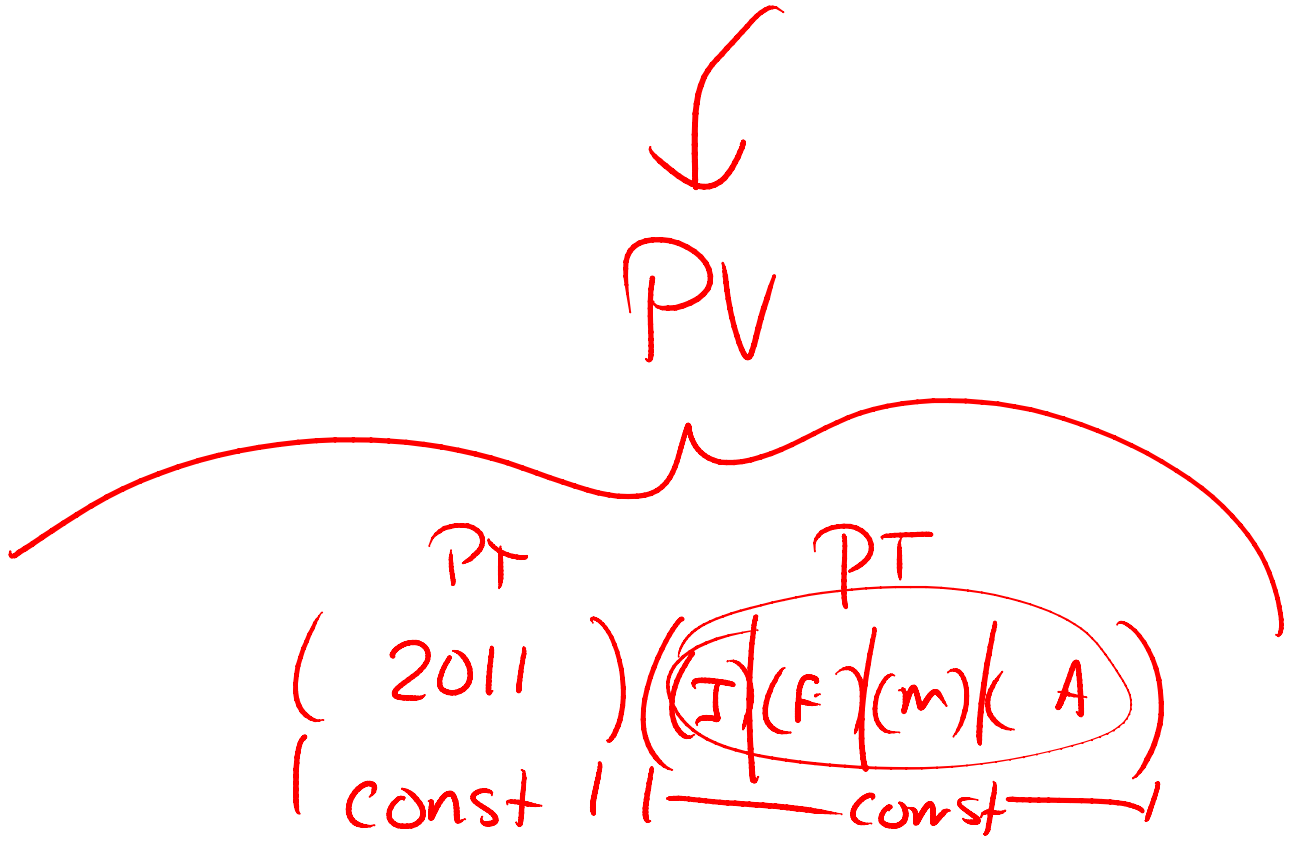


With PVs - If you want to create indexed views, you must create them individually per table (for the tables that underpin the PV). If you're not on EE then you'll also need to create a specific view to access them with the (WITH NOEXPAND) hint.

M13s50

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+





M12

You can have partitioned tables and/or non-partitioned tables that underpin a view. The only requirement – make sure that each of the tables (UNIONed in the view) has a constraint!x