SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Kimberly's Whiteboard Drawings and Annotations
From IEPTO1: week of October 7, 2019

Kimberly L. Tripp

Kimberly@SQLskills.com

SQLskills
immerse yourself in sql server

Plus drawings from other classes
(that were better) or just had info
we may not have covered!
Have fun!
-k

Session Settings and Client Connectivity

From Pluralsight course, Optimizing Stored Procedures – Part 2

SET Options	Required for Perf Features	Default Server Value	SSMS	SQLCMD	SQL Server Agent	Default OLE DB and ODBC Value	Default DB-Library Value	.NET / Your App
ANSI_DEFAULTS [2]	NO	OFF	OFF	OFF	OFF	OFF	OFF	?
ANSI_NULL_DFLT_ON [2]	NO	**OFF**	ON	ON	ON	ON	**OFF**	?
ANSI_NULLS [1, 3]	YES = ON	**OFF**	ON	ON	ON	ON	**OFF**	?
ANSI_PADDING [2, 3]	YES = ON	ON	ON	ON	ON	ON	**OFF**	?
ANSI_WARNINGS [2]	YES = ON	**OFF**	ON	ON	ON	ON	**OFF**	?
ARITHABORT [2]	YES = ON	ON	ON	**OFF**	**OFF**	**OFF**	**OFF**	?
CONCAT_NULL_YIELDS_NULL [2, 3]	YES = ON	**OFF**	ON	ON	ON	ON	**OFF**	?
NUMERIC_ROUNDABORT [2]	YES = OFF	OFF	OFF	OFF	OFF	OFF	OFF	?
QUOTED_IDENTIFIER [1]	YES = ON	**OFF**	ON	**OFF**	**OFF**	ON	**OFF**	?

(1) The only state that's important is how it's set when the stored procedure is CREATED; the runtime setting is irrelevant.

(2) If different than existing plan in cache, will be recompiled and added to the plan cache; performance may vary at execution.

(3) In a future version of SQL Server, these options will always be ON and any applications that explicitly set the option to OFF will produce an error. Avoid using this feature in new development work and plan to modify applications that currently use this feature.

SQLskills Immersion Event

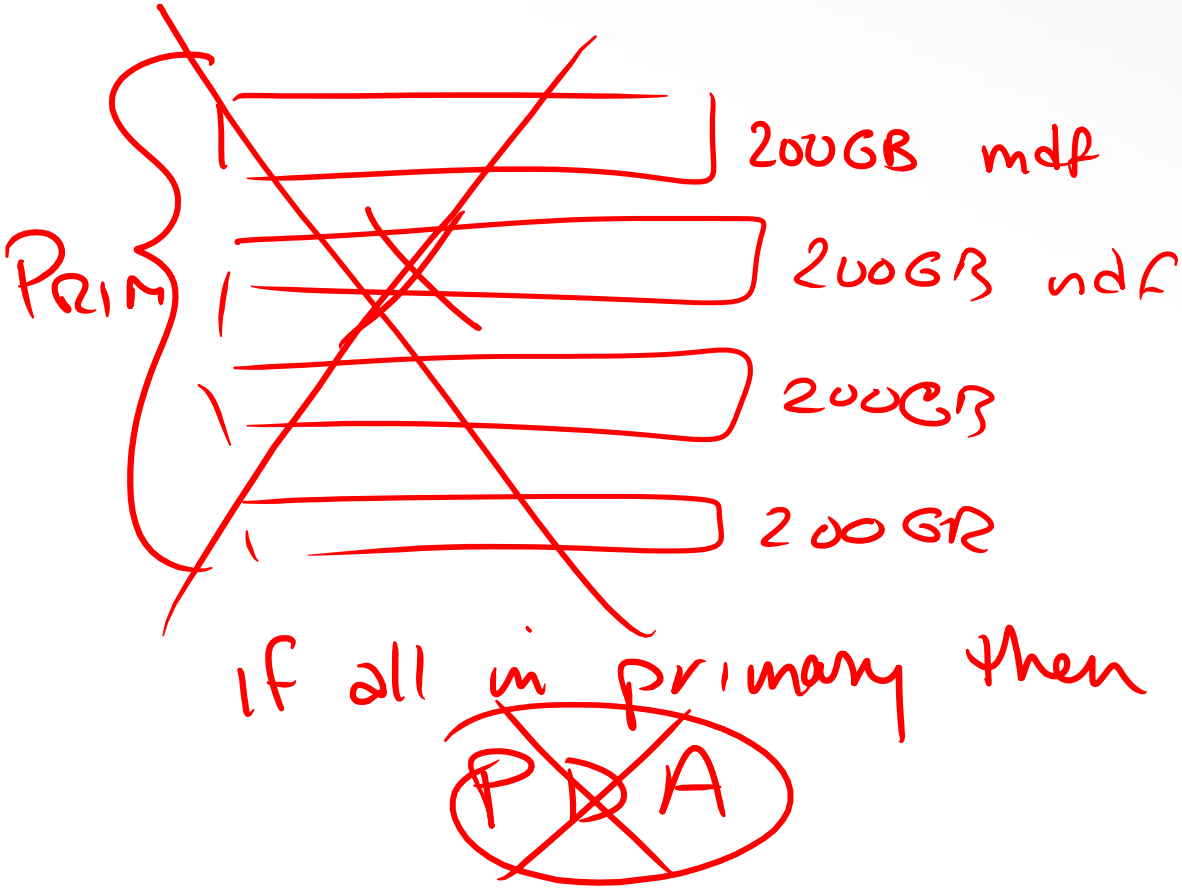
IEPTO1: Performance Tuning and Optimization

General Internals

Kimberly L. Tripp

Kimberly@SQLskills.com

NOTE: On some tangents, I often discuss internals; this first section is from these drawings.

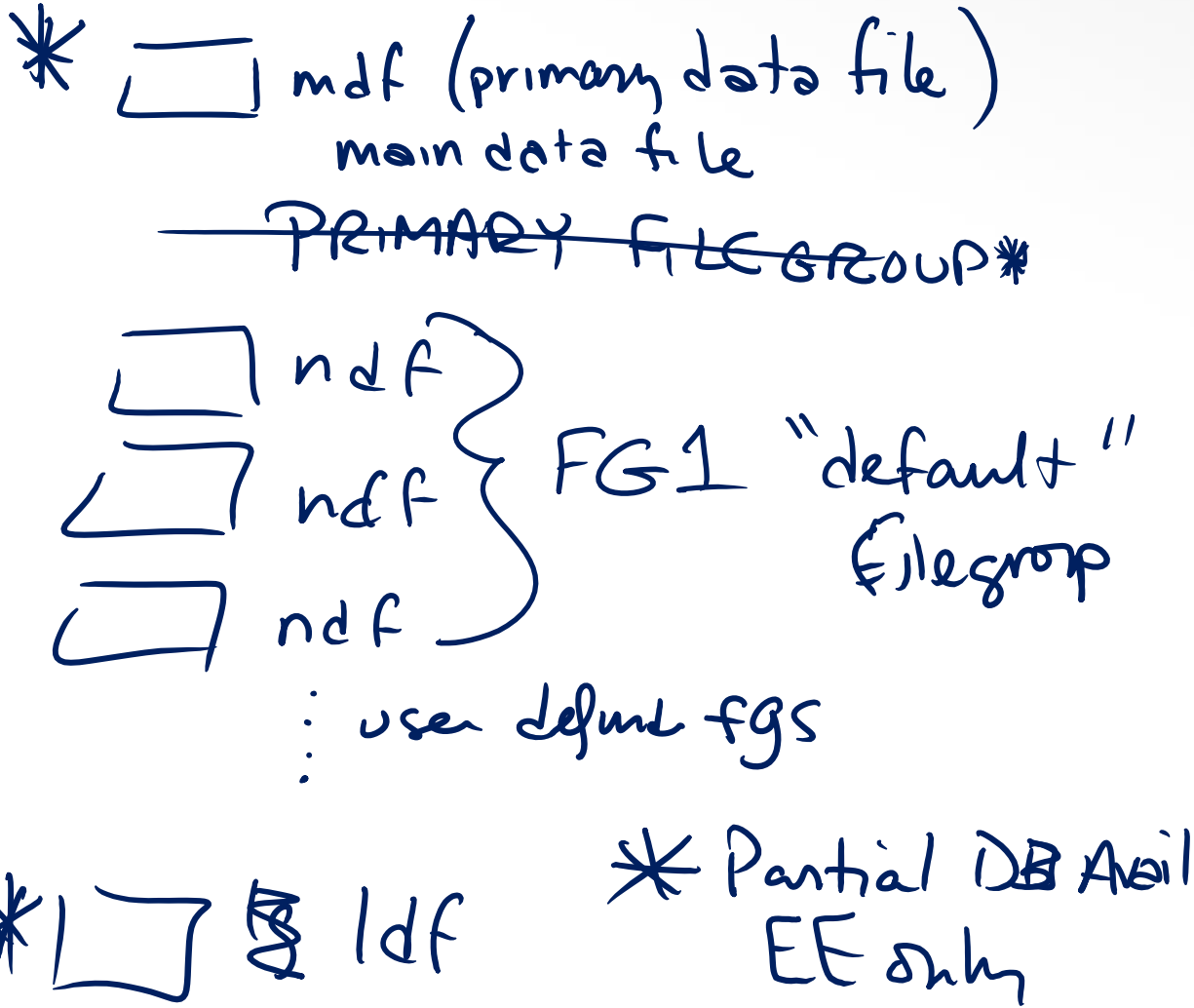


One of the problems in allowing files to be added to the primary filegroup is that is makes it more vulnerable to downtime. SQL Server's Partial Database Availability feature ONLY works when both the primary filegroup AND the log are available. If the primary filegroup has multiple files and one of those files fails – the entire database will be offline until restored.

For VLDBs, you want to isolate the primary filegroup and dedicate it only to system tables. Then, place it on redundant disks.

Secondary, non-primary filegroups are where you'll store data.

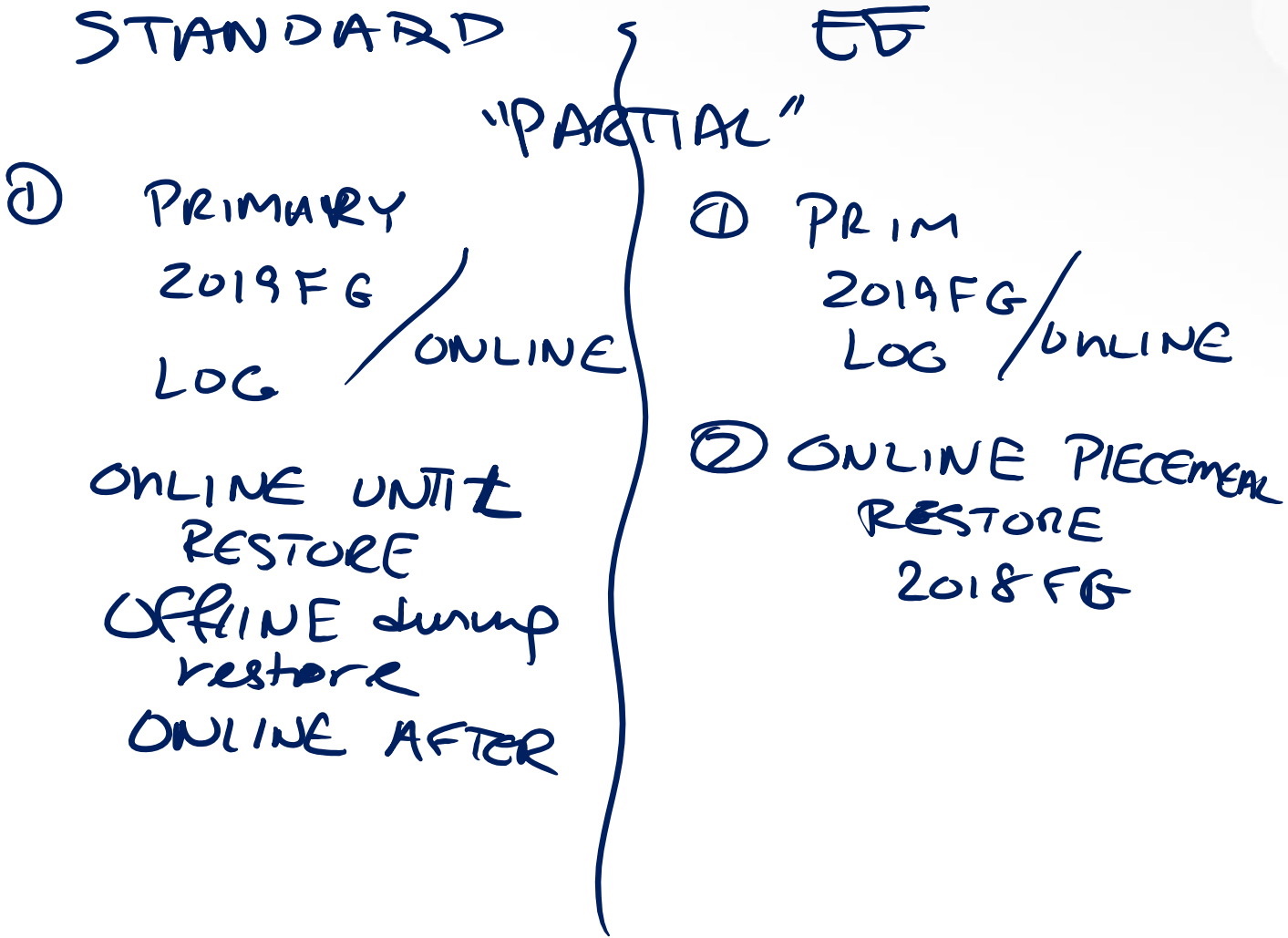
(see the next drawing)



By dedicating only the system objects to the primary filegroup (and limiting this to only one, relatively-small mdf, you reduce the potential for downtime.

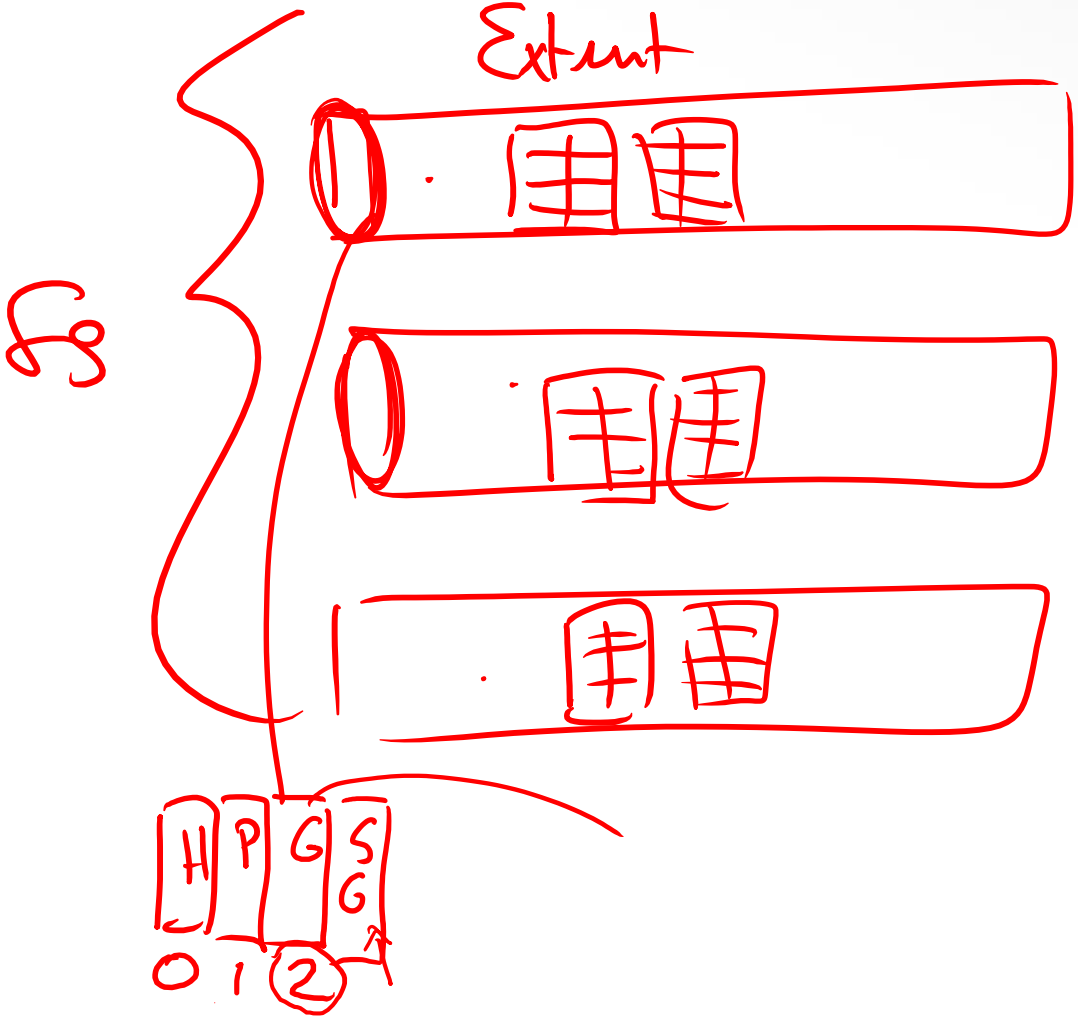
If a file does become damaged then the database can remain online and available (PDA). And, then, using Online Piecemeal Restores, you can keep your database online for most of the recovery process.

NOTE: There is one caveat. To take a file offline (in order to begin the restore process), you must take the filegroup of which it's a member offline. This is NOT an online operation (because SQL Server does not track transactions at the filegroup level). So, all connections to the database are terminated when you take a filegroup offline.



Even Standard Edition supports partial restores – however, once restored and brought online partially – subsequent restores must be done with the database offline!

You can continue to restore partially and keep bringing the database online partially… but, remember that the log always has to be restored to bring the database to "current"



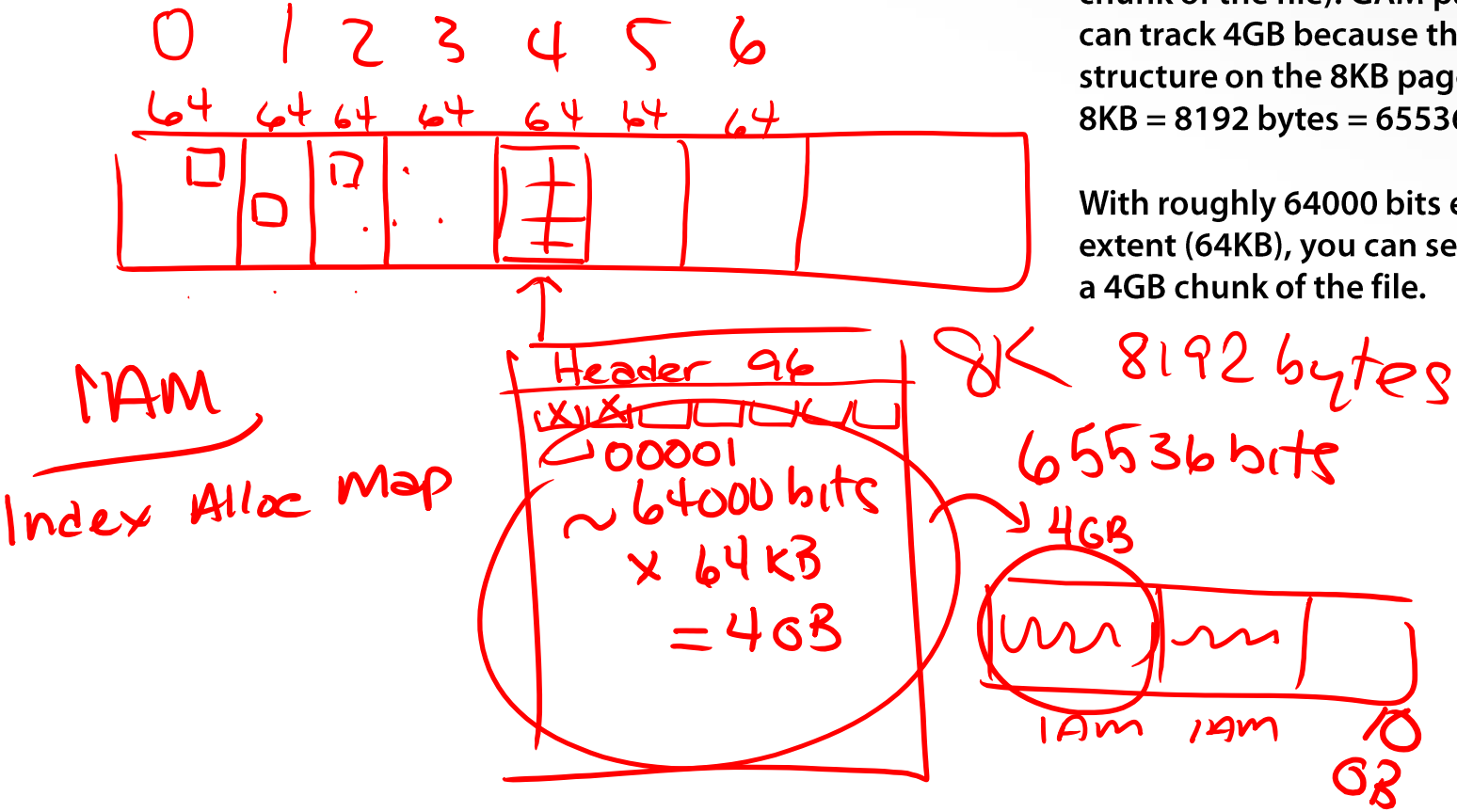
allocations?

After eight 8KB pages have been allocated to a table, SQL Server does extent-based allocations. These allocations are done file by file. Or, in other words, SQL Server round-robins extent allocations from the files in the filegroup.

This helps to reduce IO contention and also contention on the system allocation pages (GAM pages are used for extent allocations) themselves.

To the left – the first 4 pages in a file are:
Header (0)
PFS: Page Free Space (1)
GAM: Global Allocation Map (2)
SGAM: Shared Global Allocation Map (3)

IAM Pages



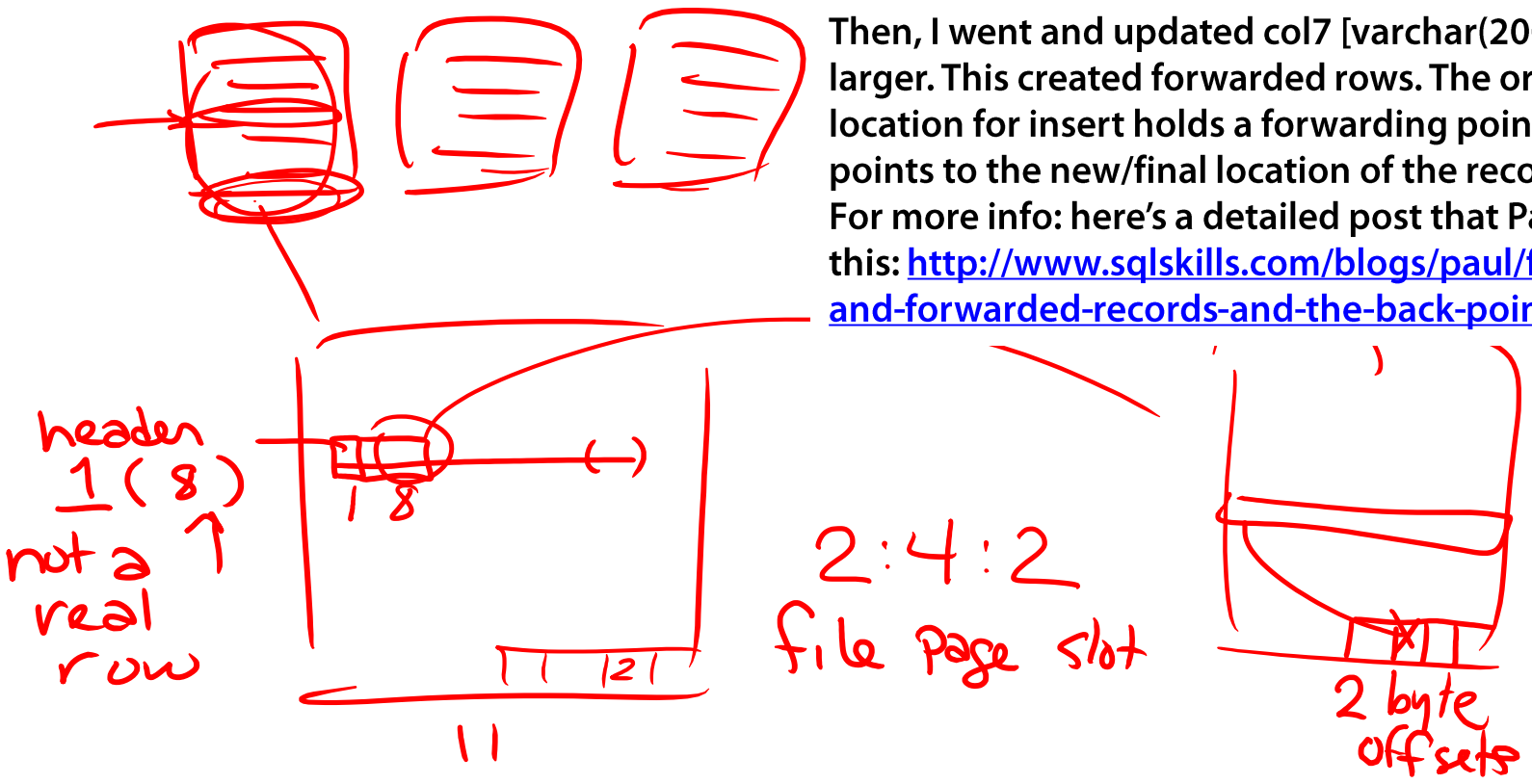
While GAM pages track extent usage within the database, IAM pages track a table's allocated extents within a GAM interval (a 4GB chunk of the file). GAM page and IAM pages can track 4GB because they use a bitmap structure on the 8KB page.
8KB = 8192 bytes = 65536 bits

With roughly 64000 bits each tracking an extent (64KB), you can see how they can track a 4GB chunk of the file.

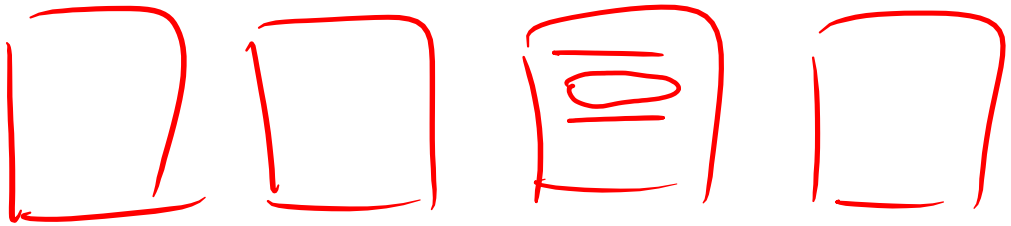
From the Heap DEMO

In my heap demo, we added a bunch of rows to the table. All of the rows were fixed-width and column 7 had only 11 bytes per row. SQL Server filled the pages as fast as they could…

Then, I went and updated col7 [varchar(200)] to be larger. This created forwarded rows. The original location for insert holds a forwarding pointer and that points to the new/final location of the record.
For more info: here's a detailed post that Paul did on this: http://www.sqlskills.com/blogs/paul/forwarding-and-forwarded-records-and-the-back-pointer-size/



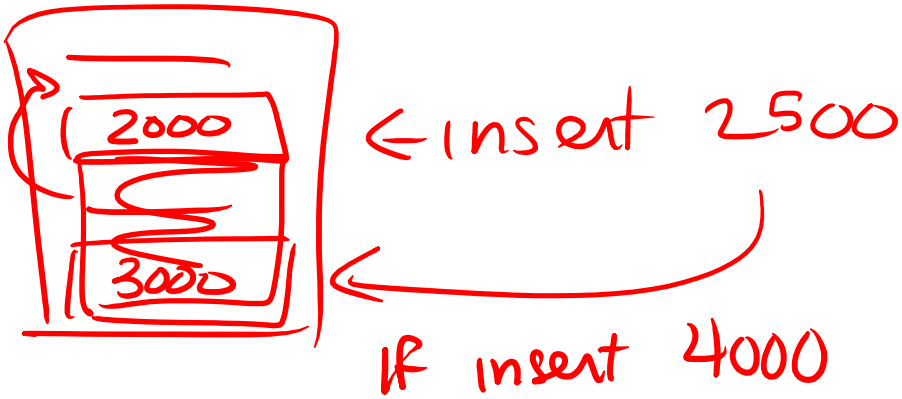
Inserts on a page (1 of 2)



SQL Server does not rearrange the records on a page – unless it needs to.

If there are two records on a page but there's a gap large enough to fit the record being inserted, then SQL Server will put the record wherever it fits.

On a heap, it will receive the next slot available. On an indexed page, the slots will be ordered by the index key.

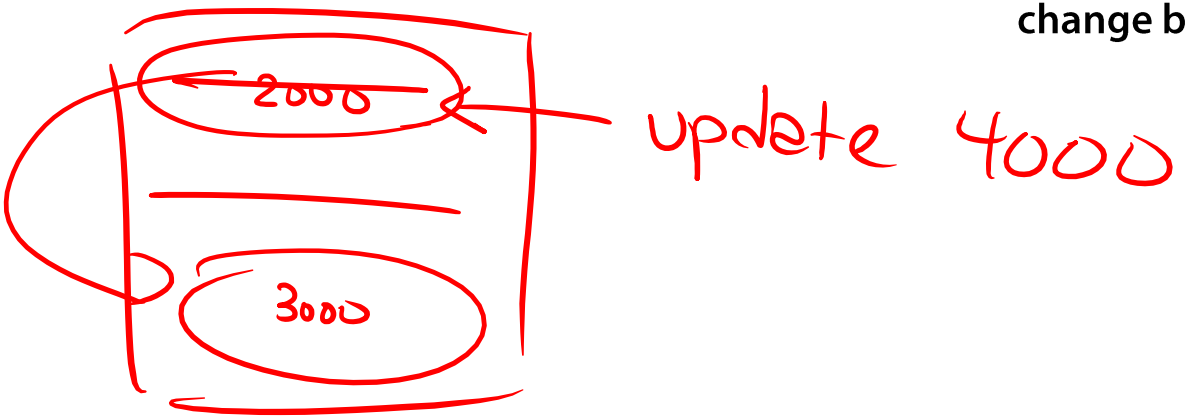


Inserts on a page (2 of 2)

SQL Server does not rearrange the records on a page – unless it needs to.

The same holds for an update. If the first record is being updated and there's not enough room without rearranging the page then SQL Server will rearrange.

If the record moves then the slot doesn't change but the offset will get updated.



Record Structures (1 of 2)

In_row fixed-width
All structures if ≤ 8060
note: never span pages
always be cont

Overflow limited VC cols
Short LOBs
if row size > 8060 then
1+ cols w/overflow

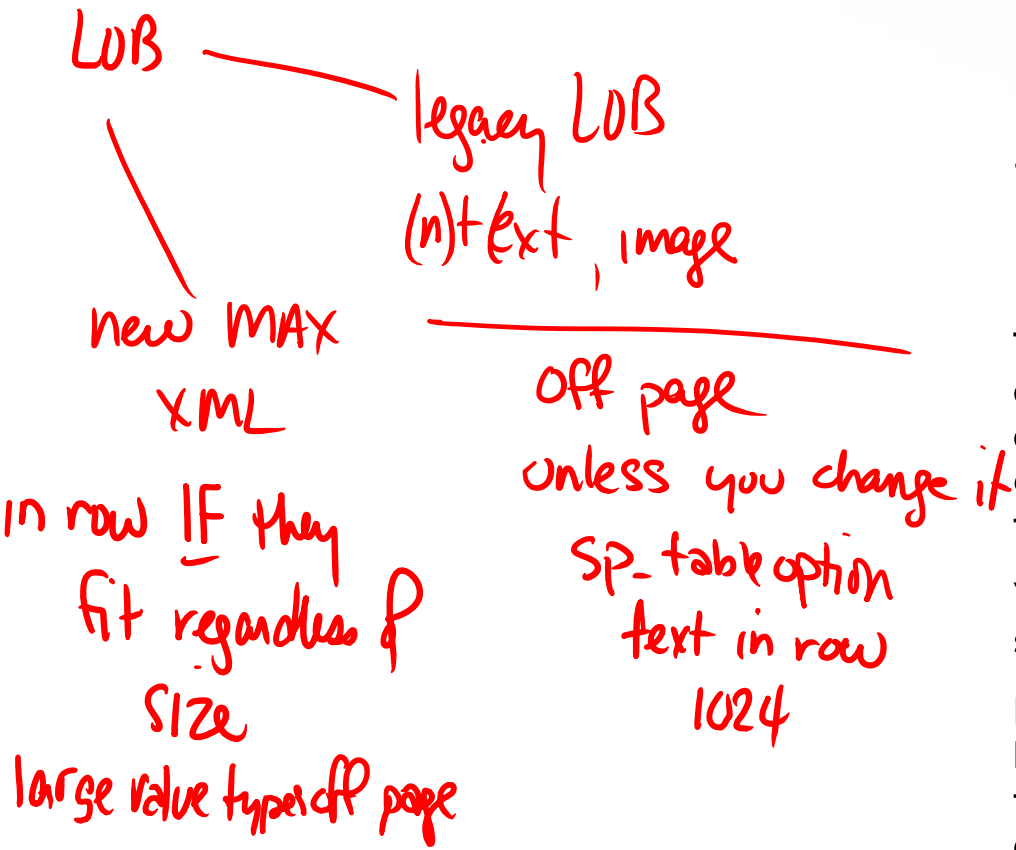
Every table/index can have up to 3 data structures associated with it:

* In_row – every row has an in_row_data structure. This cannot exceed 8060 bytes and must exist as a single, contiguous structure on – at most – one page. If a row has limited variable columns [varchar(n), nvarchar(n) or varbinary(n)] or LOB types [new LOB types: varchar(max), nvarchar(max) or varbinary(max); Legacy LOB types: text, ntext, image; XML] then you *might* have additional structures.

* Overflow – Overflow is only for limited variable columns (often called SLOBs for short LOBs). A row will ONLY overflow to the overflow structure IF the row is over 8060 bytes AND only these limited LOBs will overflow to the overflow structure. Fixed-width columns will NOT overflow. An overflowed column will always overflow as a unit and the column(s) that overflows is not easily predictable.

* LOB – explained on next page.

Record Structures (2 of 2)



* LOB – There are 8 LOB types in SQL Server:
- New LOB types
  - varchar(max)
  - nvarchar(max)
  - varbinary(max)
  - XML
  - User-defined SQLCLR types
- Legacy LOB types
  - text
  - ntext
  - Image

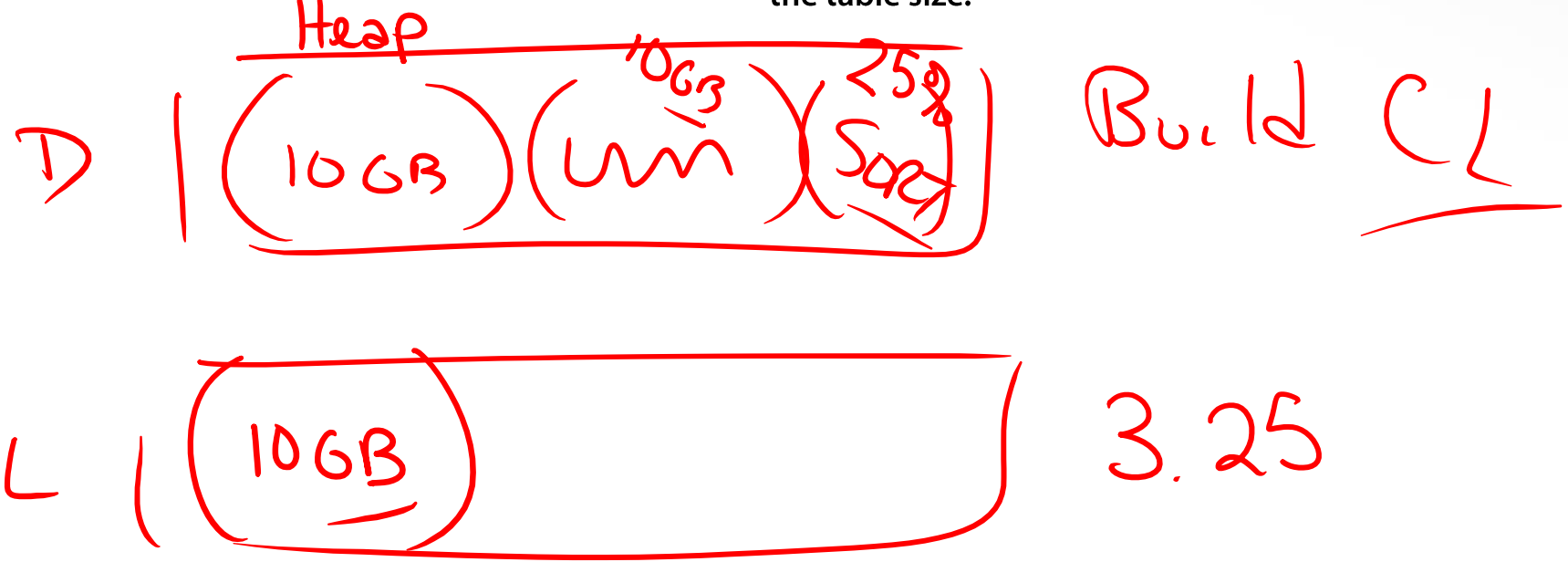
The initial location of the LOB (on insert) will depend on the LOB data type you've used. The default behavior to legacy LOB was to push them off row – regardless of size. For the new LOB types they will put them in_row only if they fit.

You can control these options by using sp_tableoption for every table change.

For legacy LOB you can control the maximum byte length for in_row LOBs. For new LOB types they either go in_row if they fit (default) OR you can say they go off-page no matter what.

The cost of creating a clustered index

To create a clustered index, SQL Server needs to copy the data (in the filegroup that is the destination) as well as use sort space to order the data (also in the destination). For a 10GB table this requires 10GB for the copy and ~25% for the sort space. Then, there's logging – in a fully logged database (the default), this is essentially a "size of data" operation. So, for a 10GB table, this is a 10GB operation in the log. The total size for a create or rebuild in a fully logged database is 3.25x the table size.

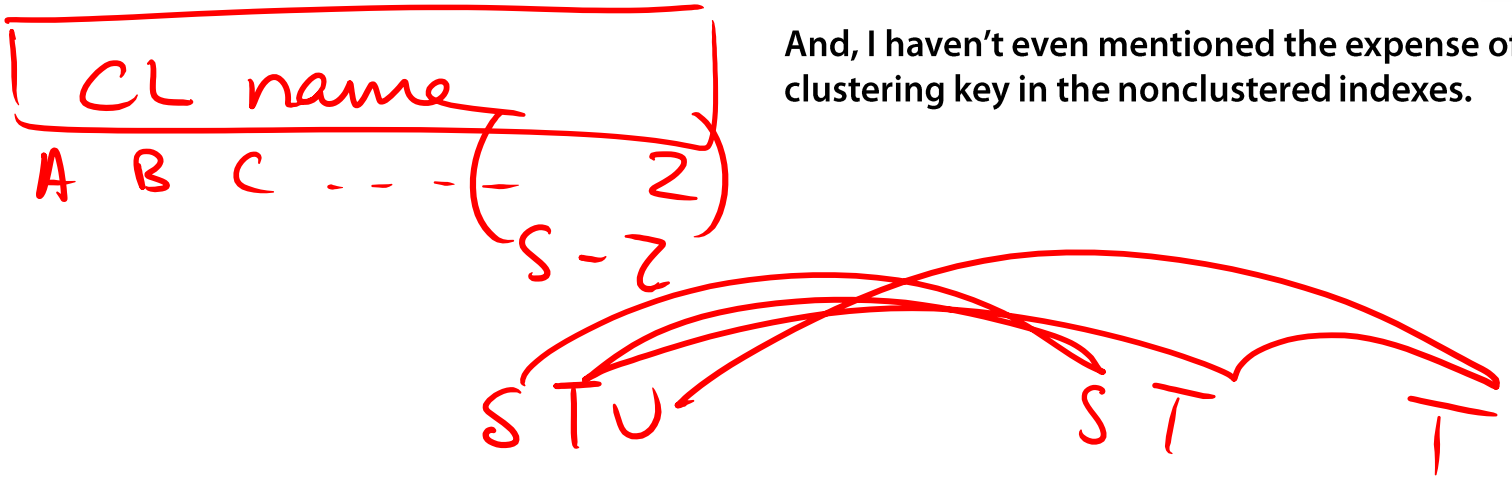


Clustering for ranges
(looks good on paper)

If I show you a clustered structure and talk ONLY about querying for ranges, I can convince you that you should create a clustered index on something like Lastname, Firstname, Middleinitial

And, while this looks good on paper, it doesn't last very long. As soon as inserts/updates occur, this table will become very fragmented. This fragmentation will offset (and begin to overshadow) the apparent benefits of having chosen that order.

And, I haven't even mentioned the expense of a wide clustering key in the nonclustered indexes.



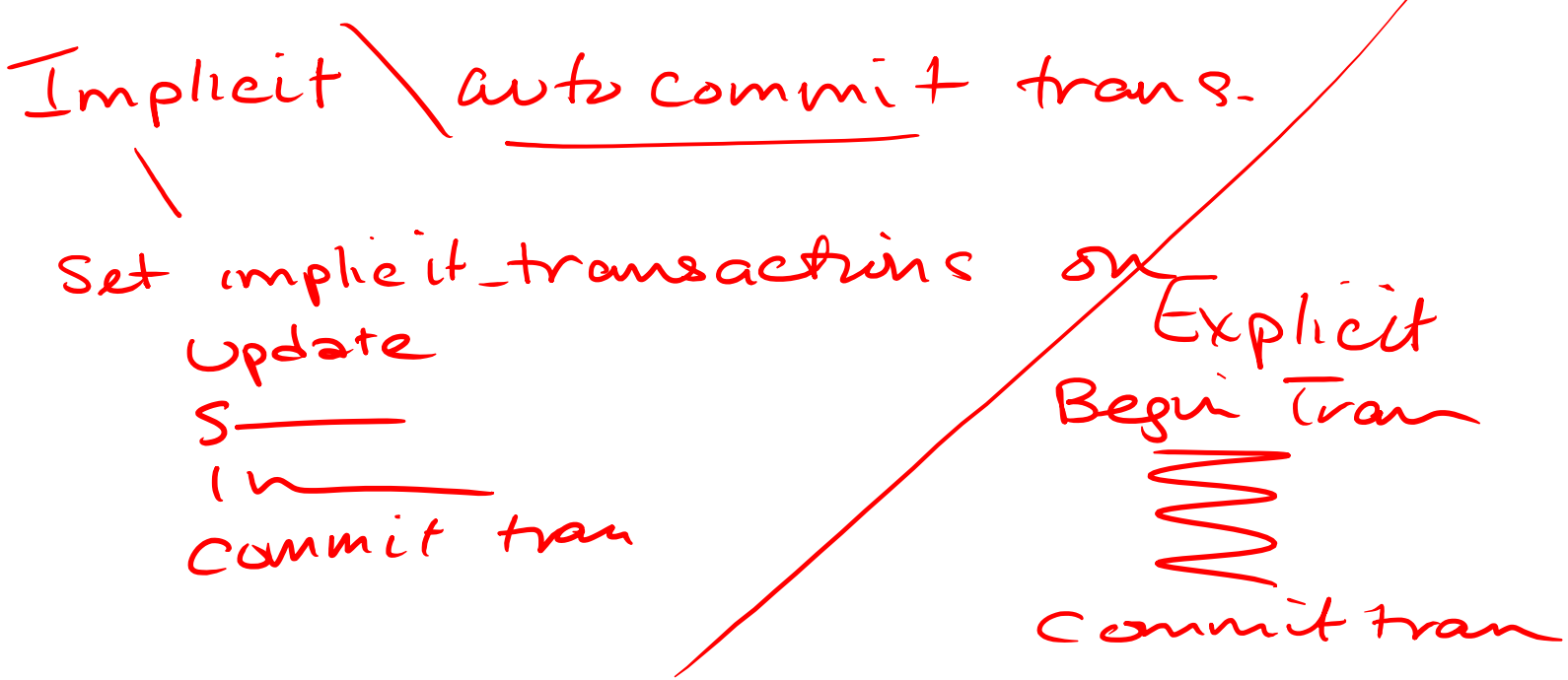
SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 3: Locking and Blocking

Kimberly L. Tripp

Kimberly@SQLskills.com

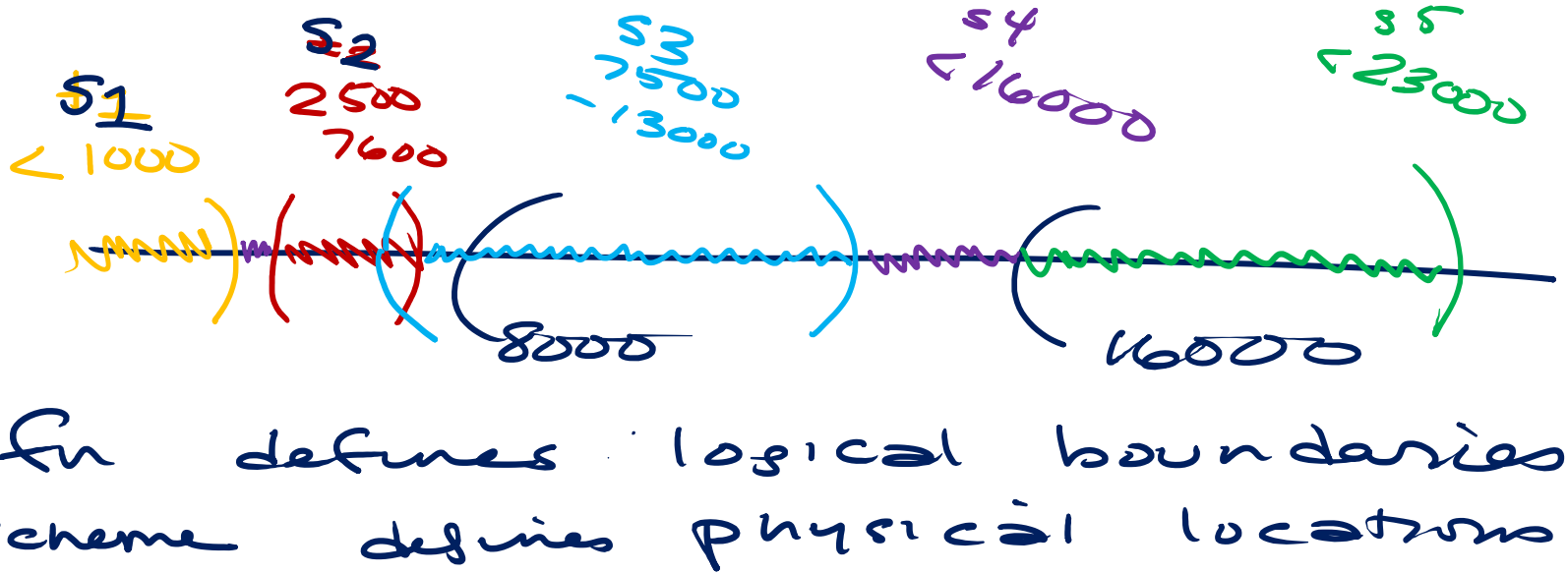


ACID properties and transaction definition

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 mode begins a transaction for you "implicitly" and so these should no be confused. To reduce confusion, use the term auto-commit instead.
- Explicit are user-defined transactions where you've explicitly defined the begin/commit.

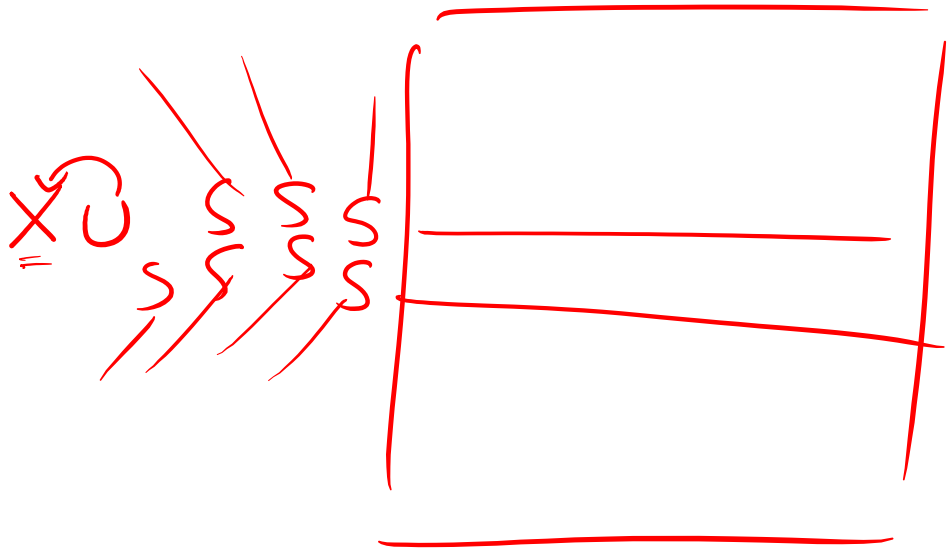


This is from the partition-level lock escalation demo

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:

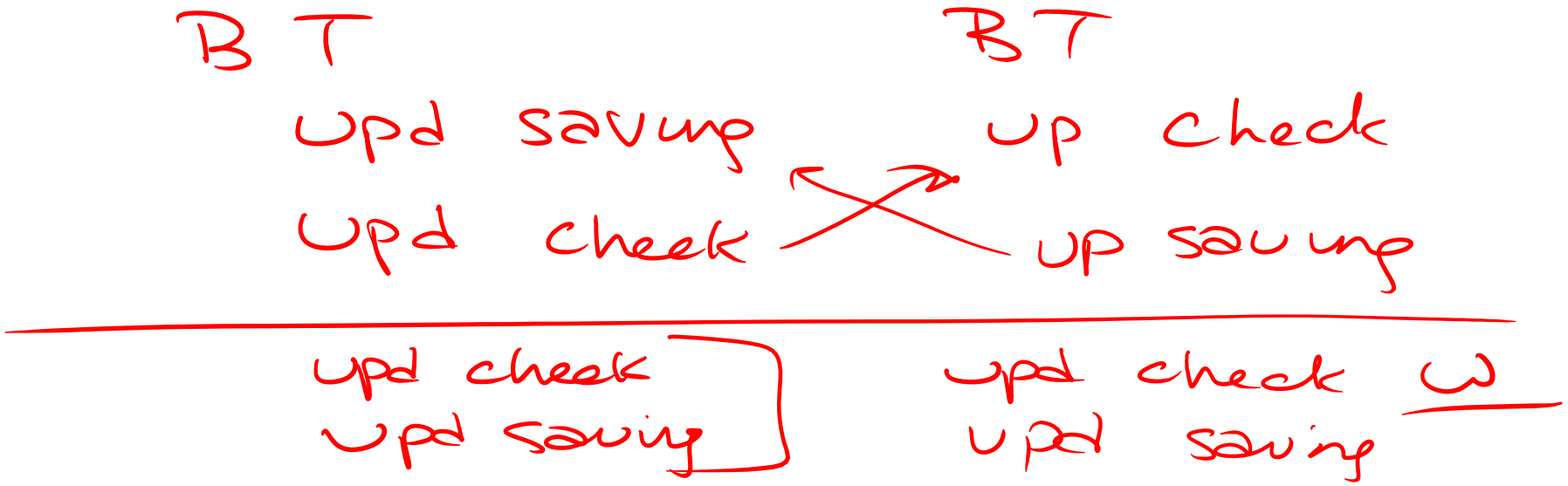
```
-∞  to 7999 | 8000 to 15999  | 16000 to ∞
 Partition 1 |  Partition 2   | Partition 3
```

To highlight what impacts escalation, I used multiple transactions. The 1st affected only rows under 1K. The 2nd affected rows 2500-7600. The 3rd affected 7500-13000. And, the 4th was all under 16K. But, none of them (individually) required ~5000 locks so none escalated.



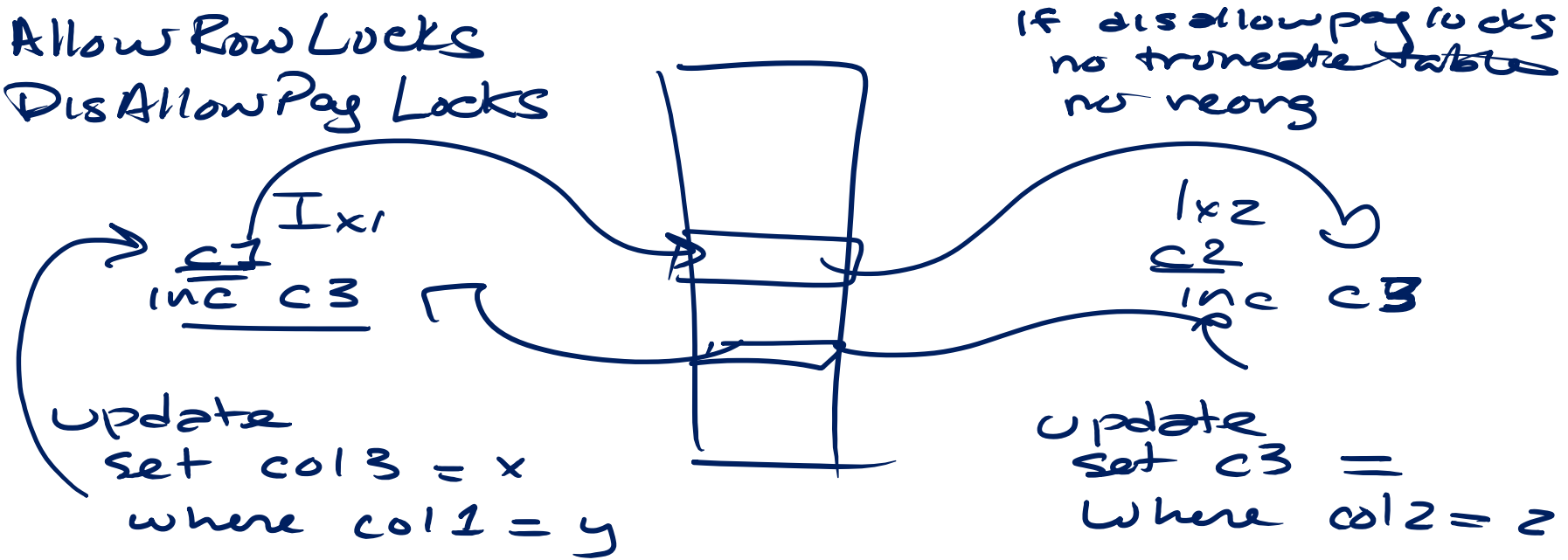
Lock starvation vs. relaxed FIFO locking

This was just a further discussion about the process that occurs when the update lock is converted to an exclusive lock. The discussion really focused back on the anatomy of a data modification. The conversion does not occur until SQL Server has to process the row. So, in a large/longer running transaction you might have many update locks acquired and then as the data is processed, the rows are converted. Until the conversion is requested, shared locks are also allowed on these rows (even while the update lock is waiting) because of relaxed FIFO. However, once the conversion is requested then ALL locks will wait.



Reducing deadlocks programmatically by changing access pattern

Deadlocks often occur when resource patterns are interleaved. Where possible, rewriting code can be really helpful. A simple example is “the banking transaction” where one user is moving money from checking to savings and the other from savings to checking. This “crisscross” of activity will be more prone to deadlocking. One option (because this is a transaction) is to reorder the statements of the transactions to always access the tables in a particular order. If this were the case then these deadlocks would be removed and replaced with blocking (not deadlocks).



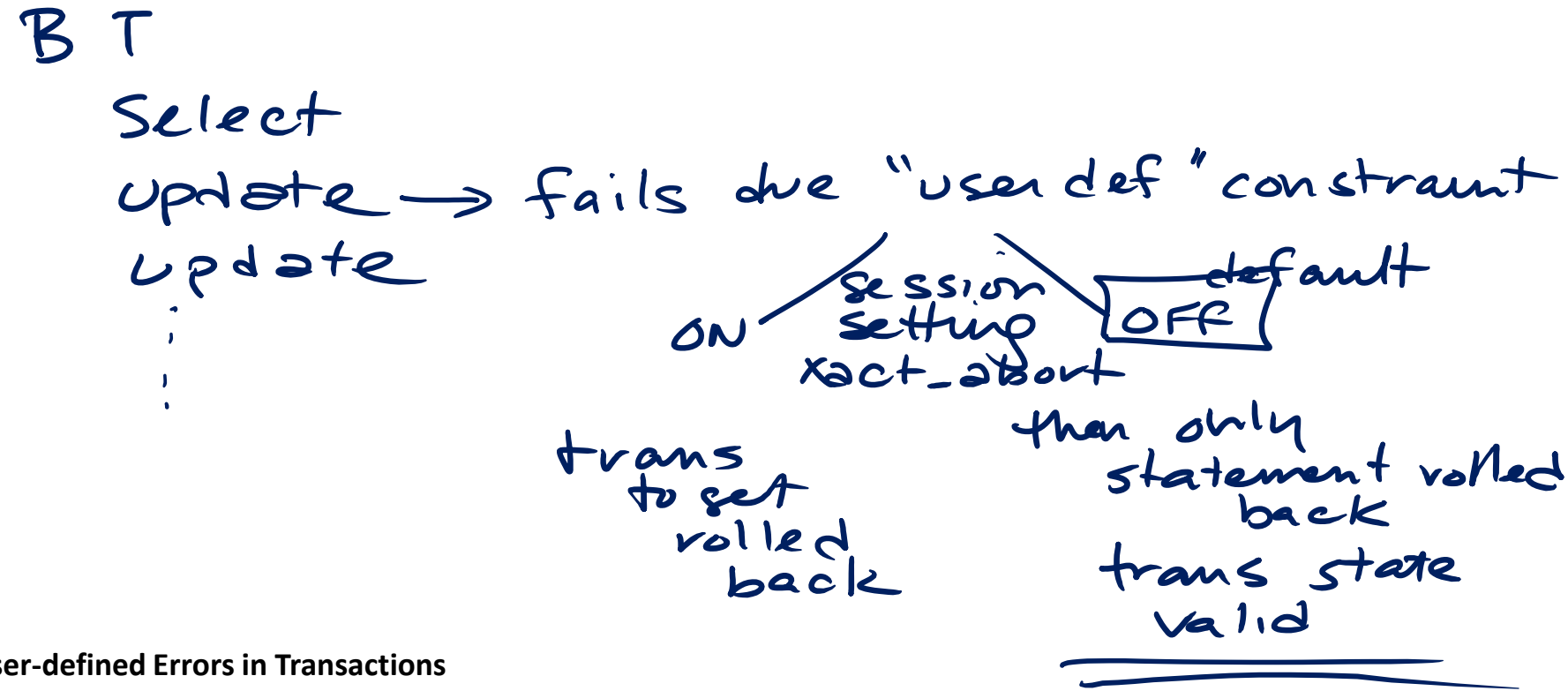
Deadlocks in an Index

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 col1 INCLUDE (col4)
Right side = index on col2 INCLUDE (col4)
```

Imagine an update to col4 that uses WHERE col1 = Y running at the same time as an update to col4 that uses WHERE col2 = X

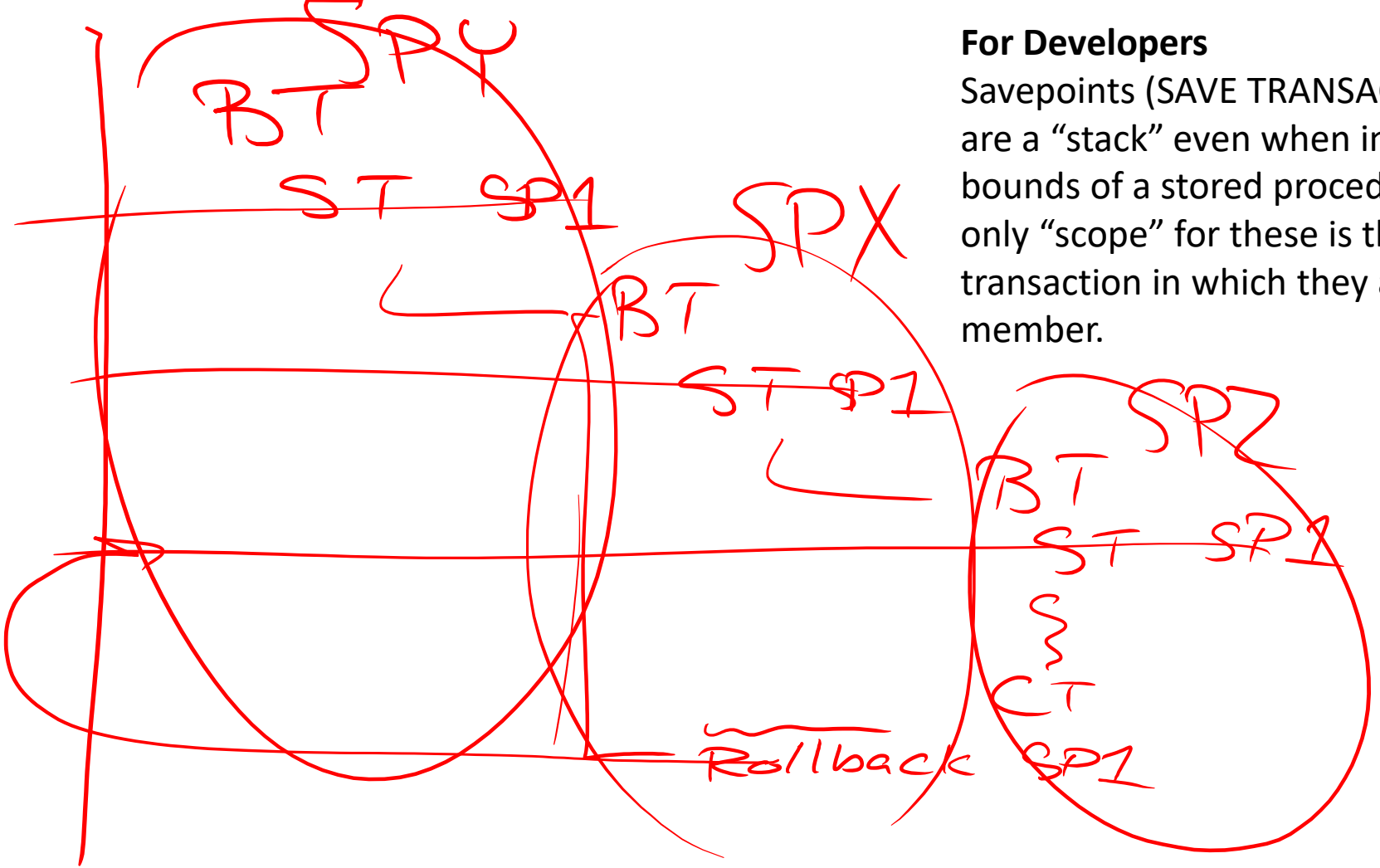
These updates are MUCH more likely to create deadlocks because of how locking works within indexes. To reduce the potential here – I often use DISALLOWPAGLOCKS and ALLOWROWLOCKS.



User-defined Errors in Transactions

If you hit a resource error, SQL Server will definitely rollback. However, if you hit a user-defined error condition (for example a constraint violation or a lock timeout), then the DEFAULT behavior is that YOU have to [programmatically] decide what to do. This requires GOOD error handling.

If you change the setting for XACT_ABORT (off by default) then ALL errors will cause a transaction to rollback.



For Developers

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.

For Developers (Transaction Madness)

The key points are:

BEGIN TRANSACTION

- Increments @@trancount

COMMIT TRANSACTION

- Decrements @@trancount

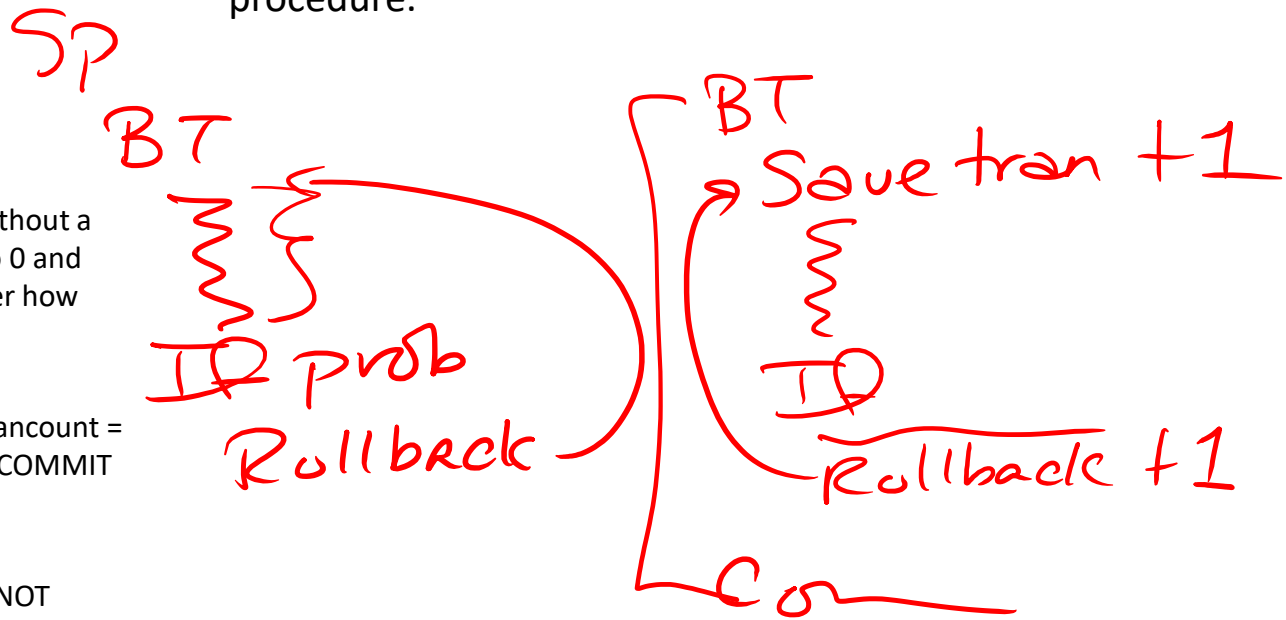
ROLLBACK TRANSACTION

- 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.

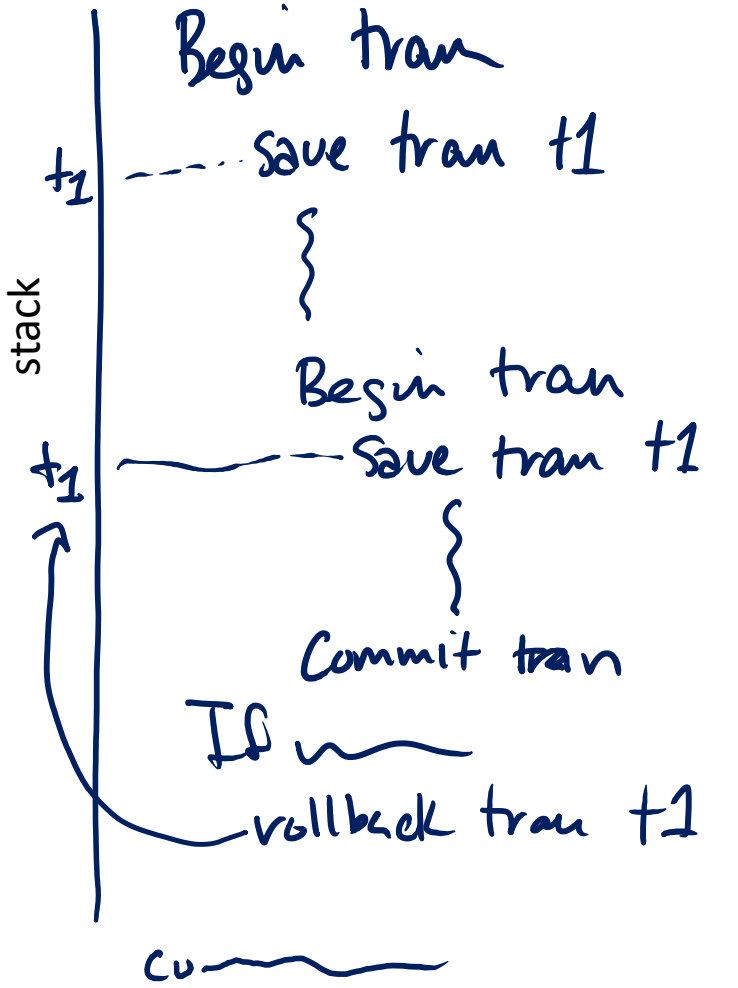
A transaction is NOT committed until @@trancount = 0. When transactions are nested you MUST COMMIT for every BEGIN

Savepoints (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.



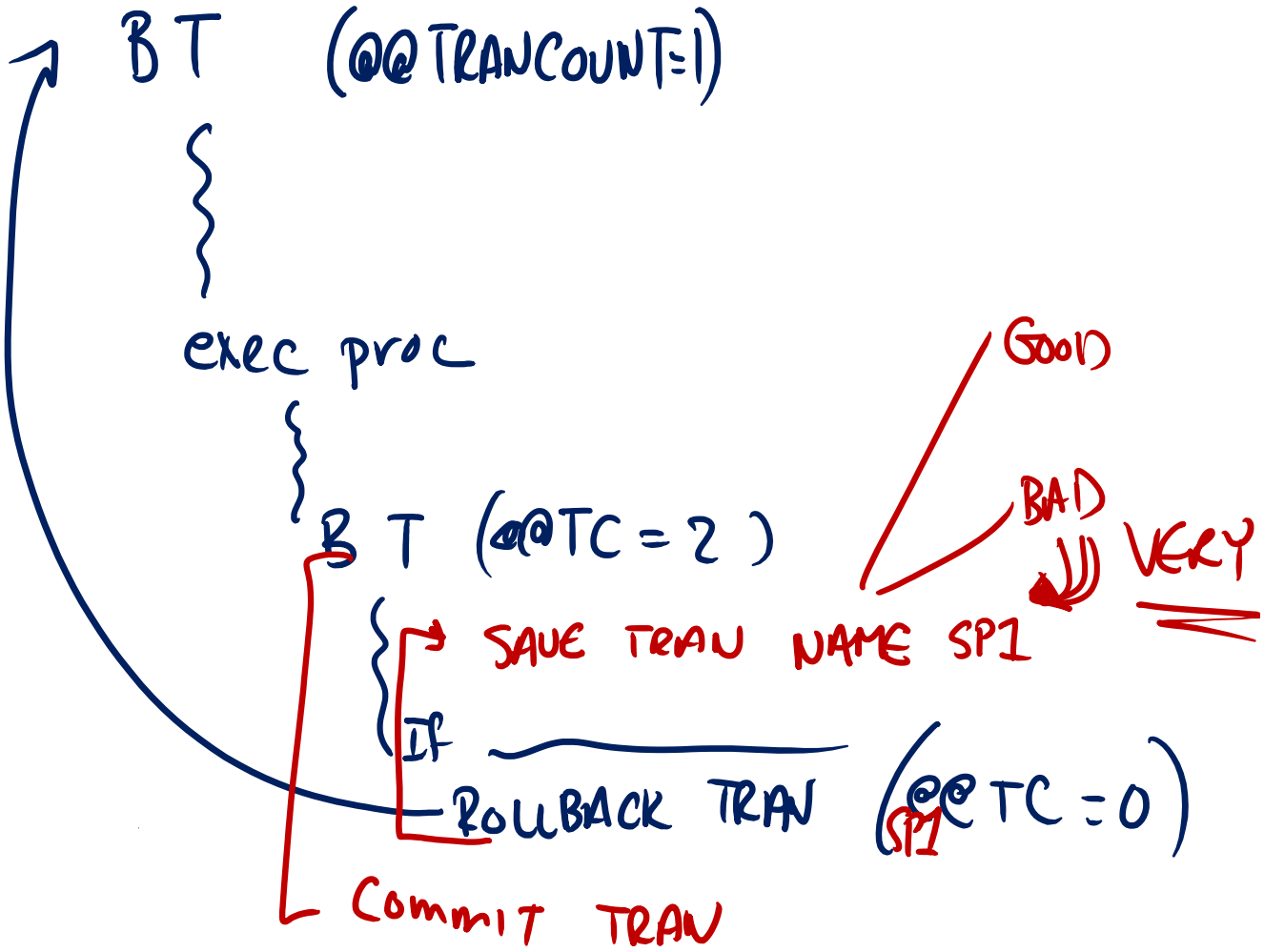
Be sure to go through the demo scripts if you need more insight into transactions!



For Developers

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.

The rollback that’s below actually rolls back to the LAST savepoint (t1) in the stack. You’ll want to make sure that every save point is unique and well-named.



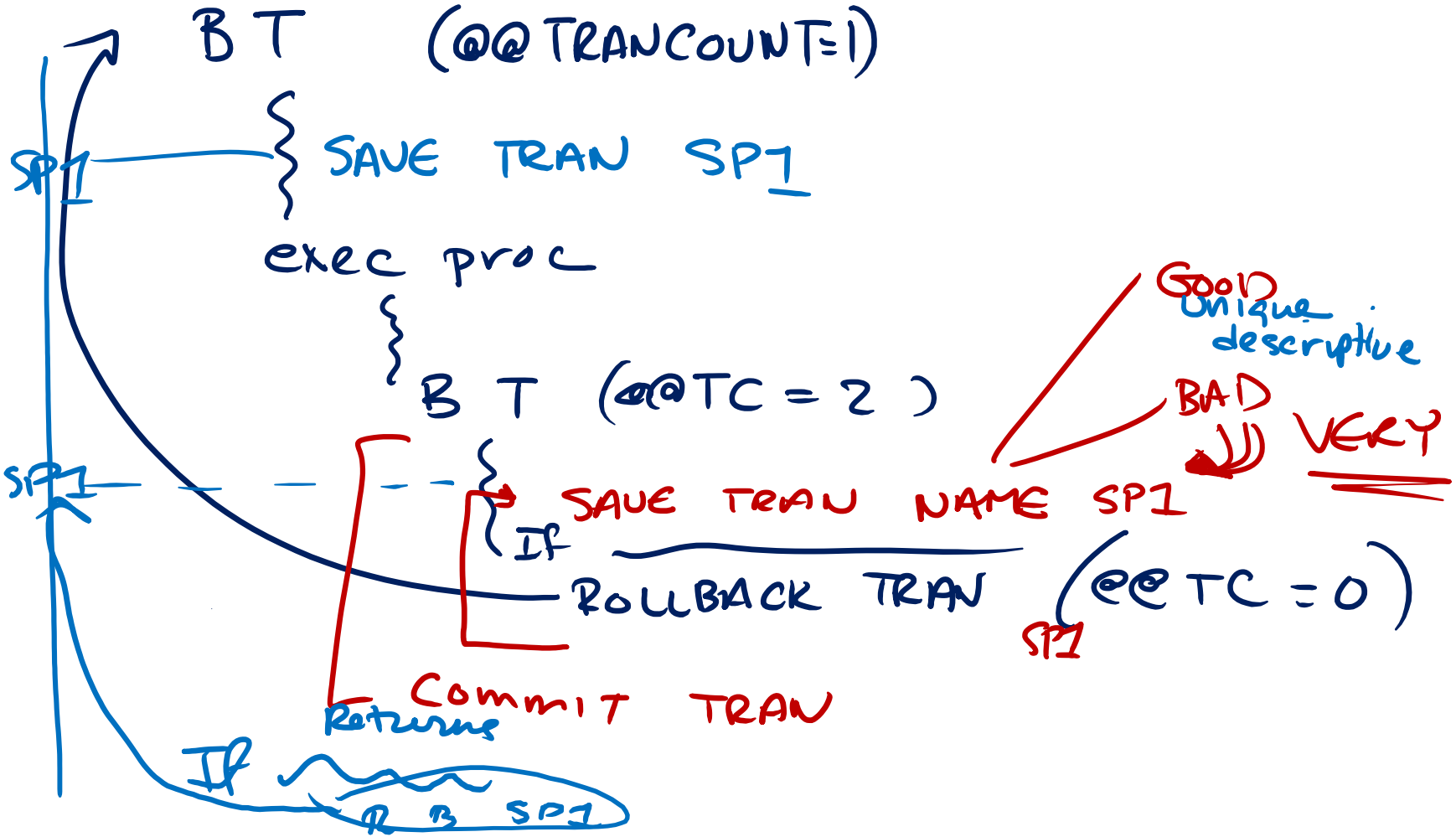
Naming savepoints

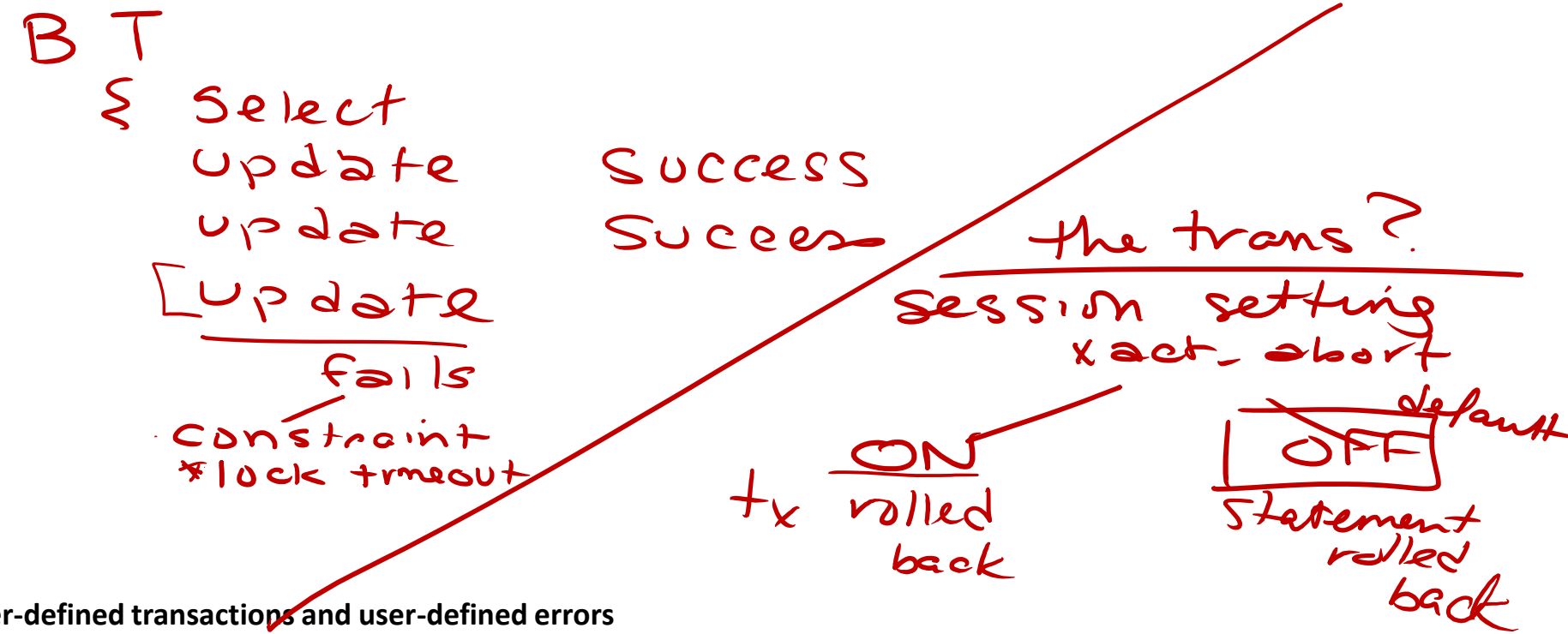
Because "nested transactions" do not exist (instead there is only ONE transaction) – you need to be careful when using (and specifically in naming) savepoints. Since these are just a stack – the "last" save point is used in a rollback – even if it's not contextually within the code that's rolling back to it.

See the next image if you've named your save points with "simple" names

A GOOD save point name is one that reflects the procedure and the state with a bit of detail.

Proc1_StateX_PointY


BT (@@TRANCOUNT=1)
SP1
SAVE TRAN SP1
exec proc
BT (@@TC = 2)
SP1
SAVE TRAN NAME SP1
Good
Unique
descriptive
BAD
VERY
If
ROLLBACK TRAN
(@@TC = 0)
SP1
COMMIT TRAN
Returns
If
R B SP1



User-defined transactions and user-defined errors

If a statement of a transaction hits a USER-DEFINED error – what happens?

It's up to the application to determine the fate of the transaction.

In this case, if the 3rd statement errors with Error 1222 (Lock request time out period exceeded.) then the statement will be rolled back but the transaction will be pending.

This is the default behavior when SET XACT_ABORT is not on.

Having better error handling – writing code within TRY/CATCH blocks is essential for reducing errors and increasing data integrity.

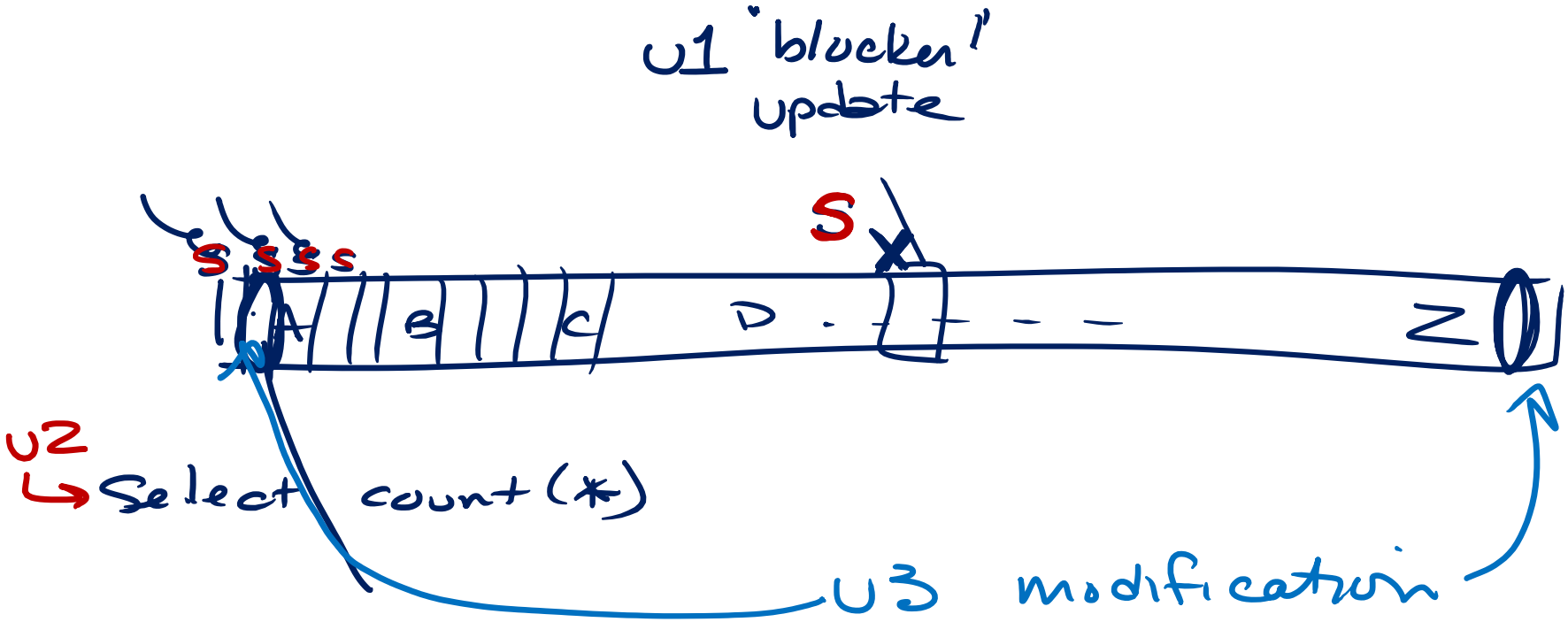
SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 4: Versioning

Kimberly L. Tripp

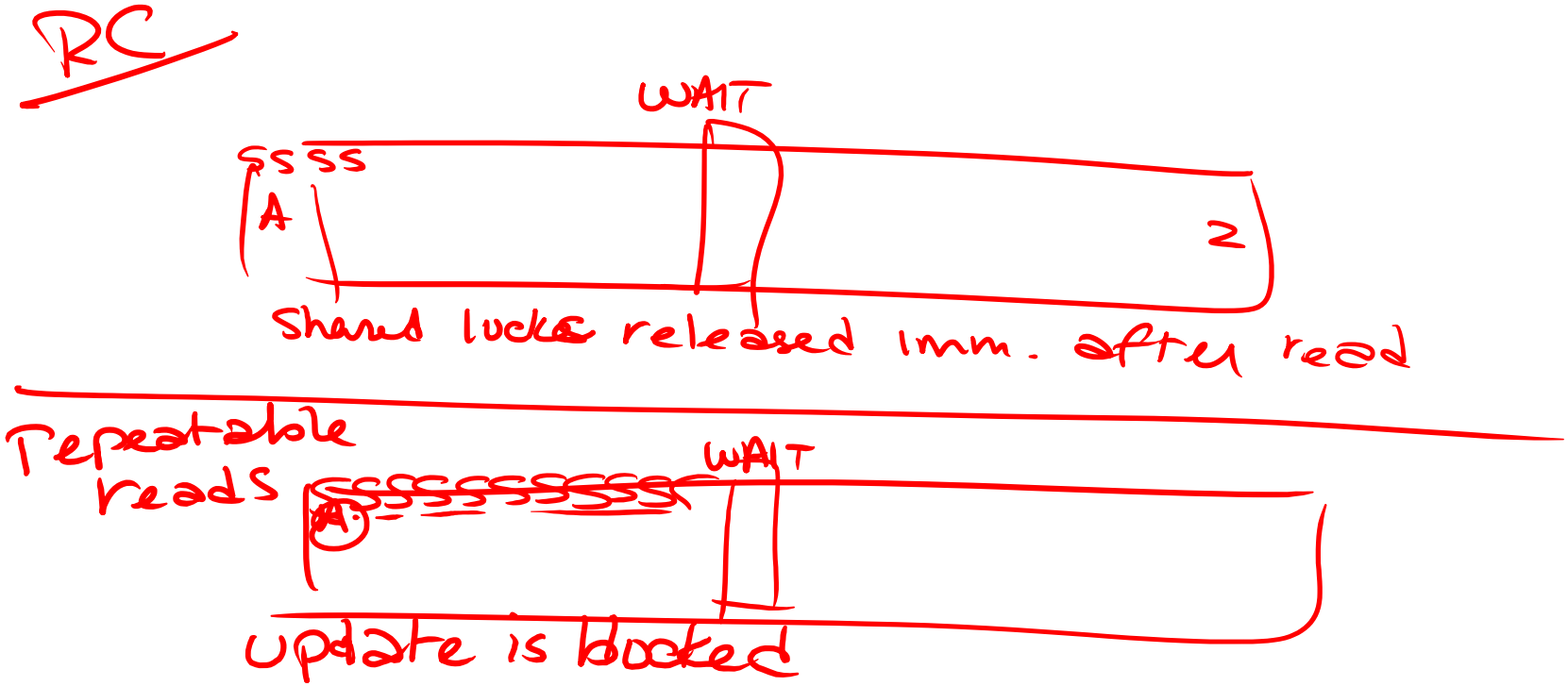
Kimberly@SQLskills.com



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. ☺



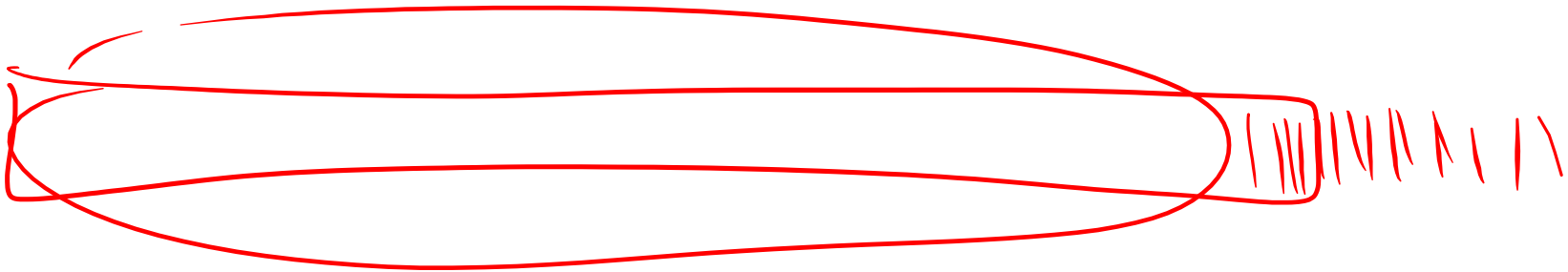
Demo: Why does RC (read committed) have non-repeatable reads?

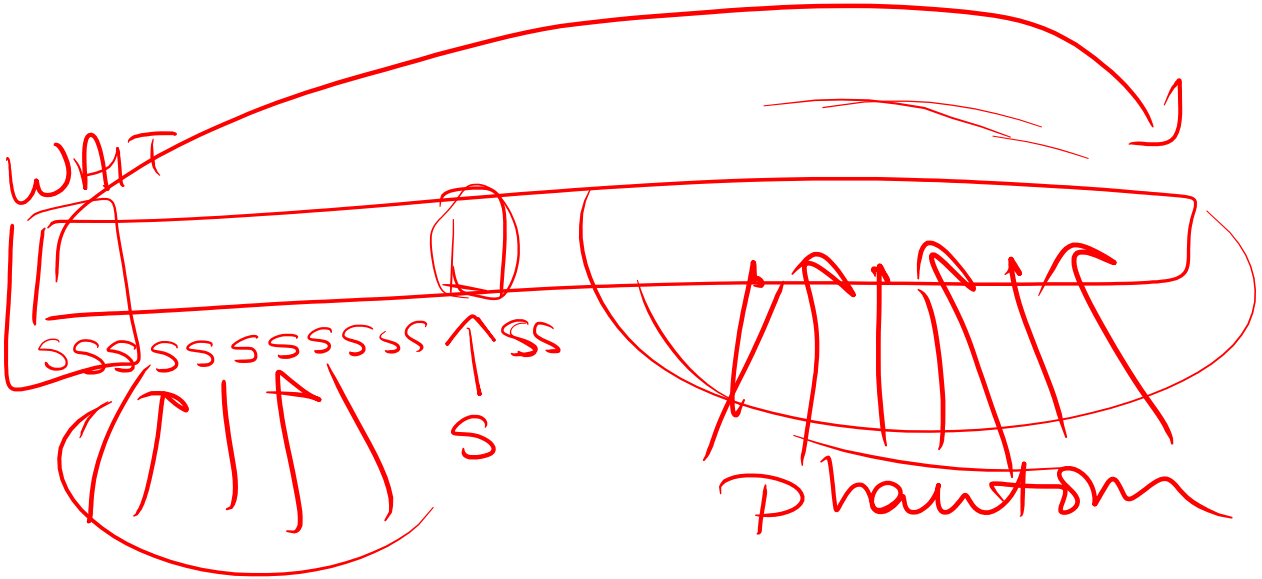
Because the row-level shared locks are released immediately after the row is read.

Increasing your isolation level to repeatable reads gets rid of this problem by LEAVING the row-level shared locks for the life of the transaction. Guaranteeing that once a row has been read – it will not be able to be changed by another transaction/user.

Discussion: Phantoms (Inconsistent Analysis)

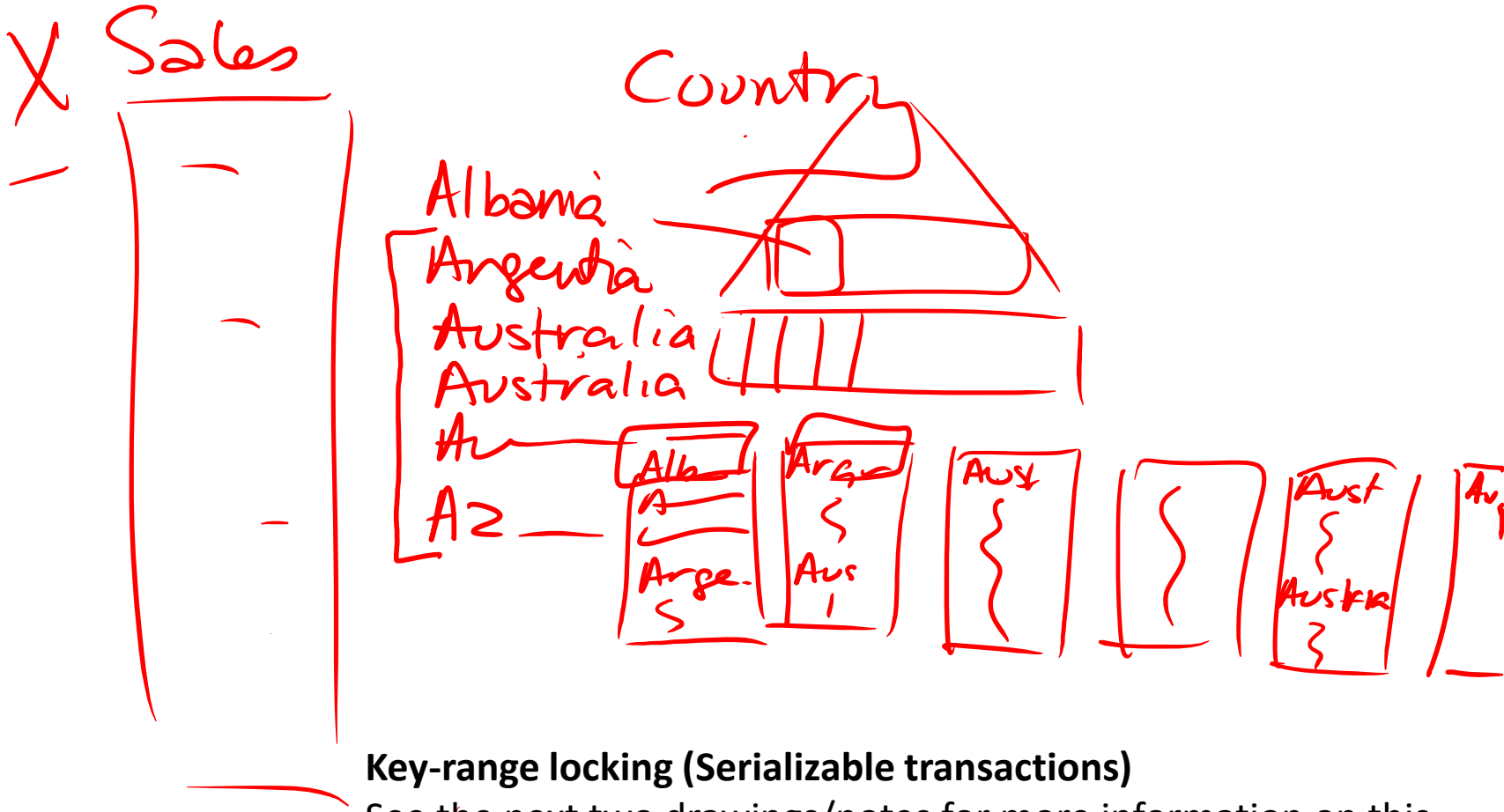
What is correct? Are we talking about row-consistency or statement-level consistency? The standards only define the state of the row at the time that the row is accessed. There is NOTHING that ties together the state of those rows at the time the query is accessed (unless you use versioning). So, here 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. ☺





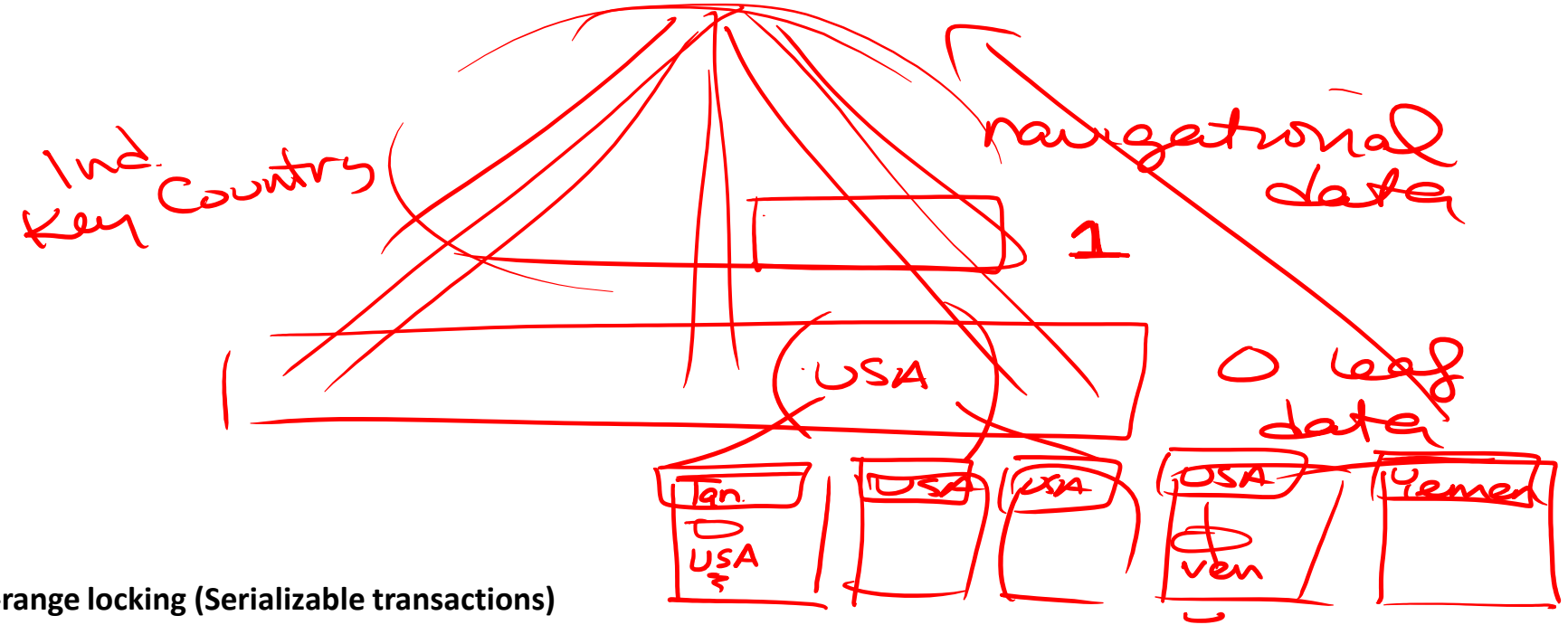
Discussion: How does repeatable read prevent the Anderson/Zembrowsky problem?

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.



Key-range locking (Serializable transactions)

See the next two drawings/notes for more information on this.

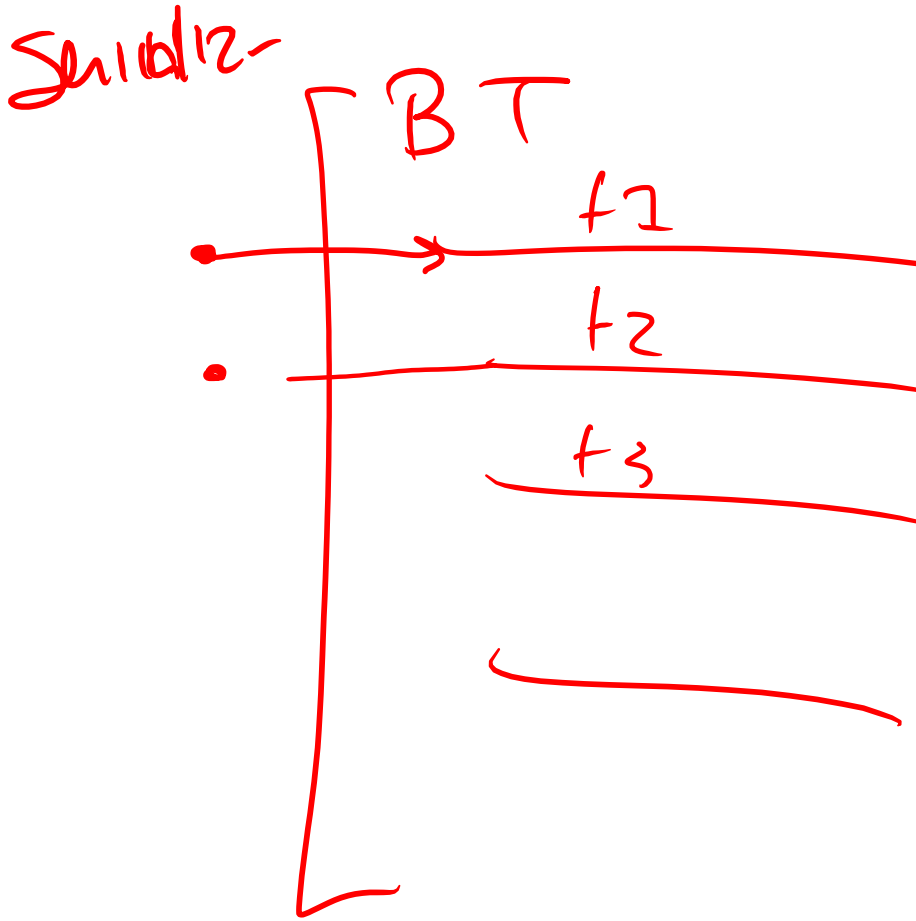


Key-range locking (Serializable transactions)

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.

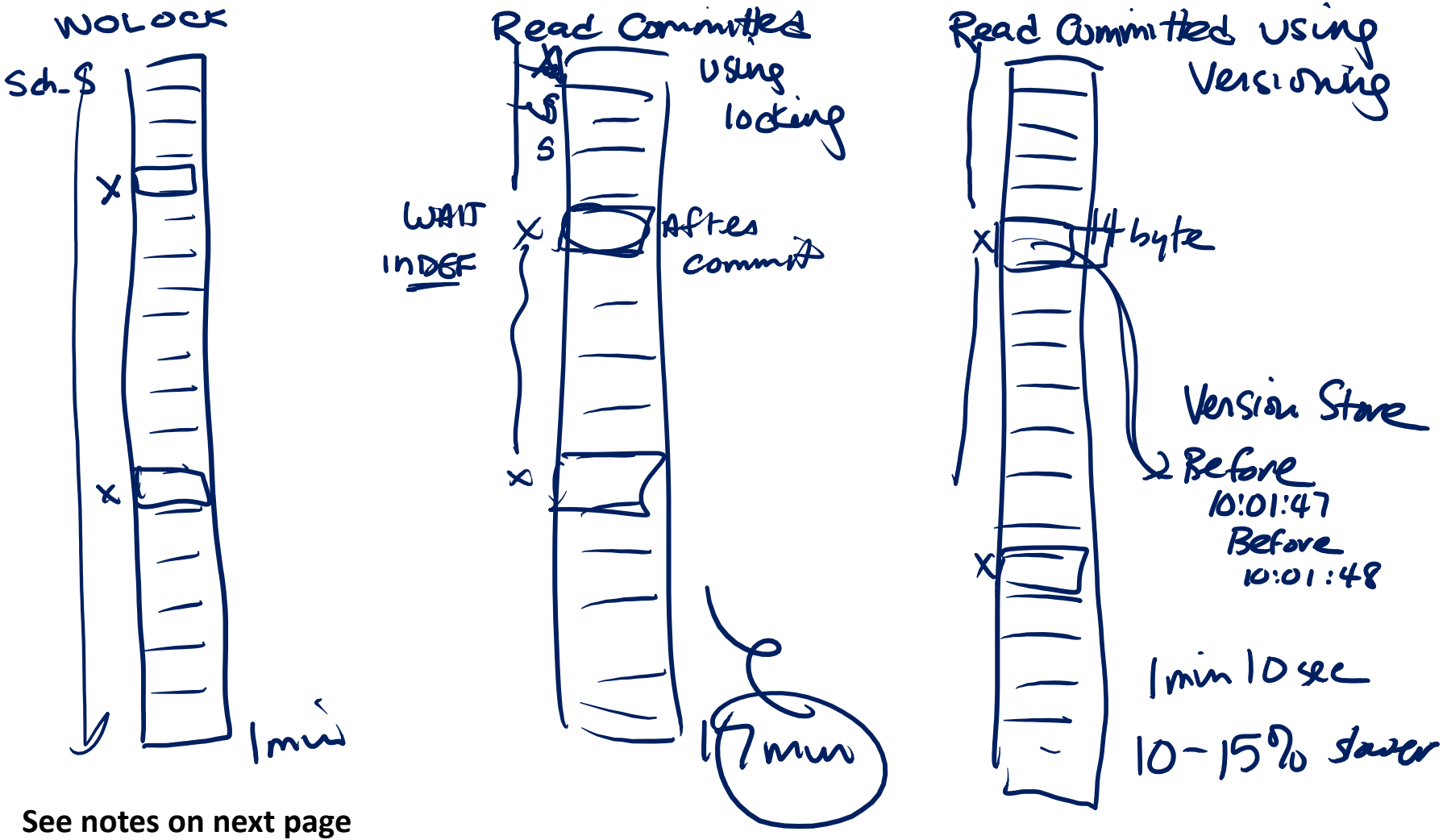
Key-range locking (Serializable transactions)

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.



Snapshot Isolation vs. Serializable transactions

Sometimes I like to say that transaction-level read consistency = Snapshot Isolation (using versioning) is more serializable than the locking based serializable transaction isolation level is. The reason why – S.I. defines the point to which **all statements** reconcile as the BEGINNING of the transaction. All statements in the transaction reconcile to EXACTLY that same point in time (defined by an LSN). However, serializable can only lock resources AS it requests them (statement by statement). In this picture, I drew that distinction – each statement locks AS it executes. So, the 2nd and 3rd statements reconcile to t2 (time 2) and t3 (time 3). Where as a snapshot isolation-based transaction would always reconcile to t1.



See notes on next 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 (using locking)

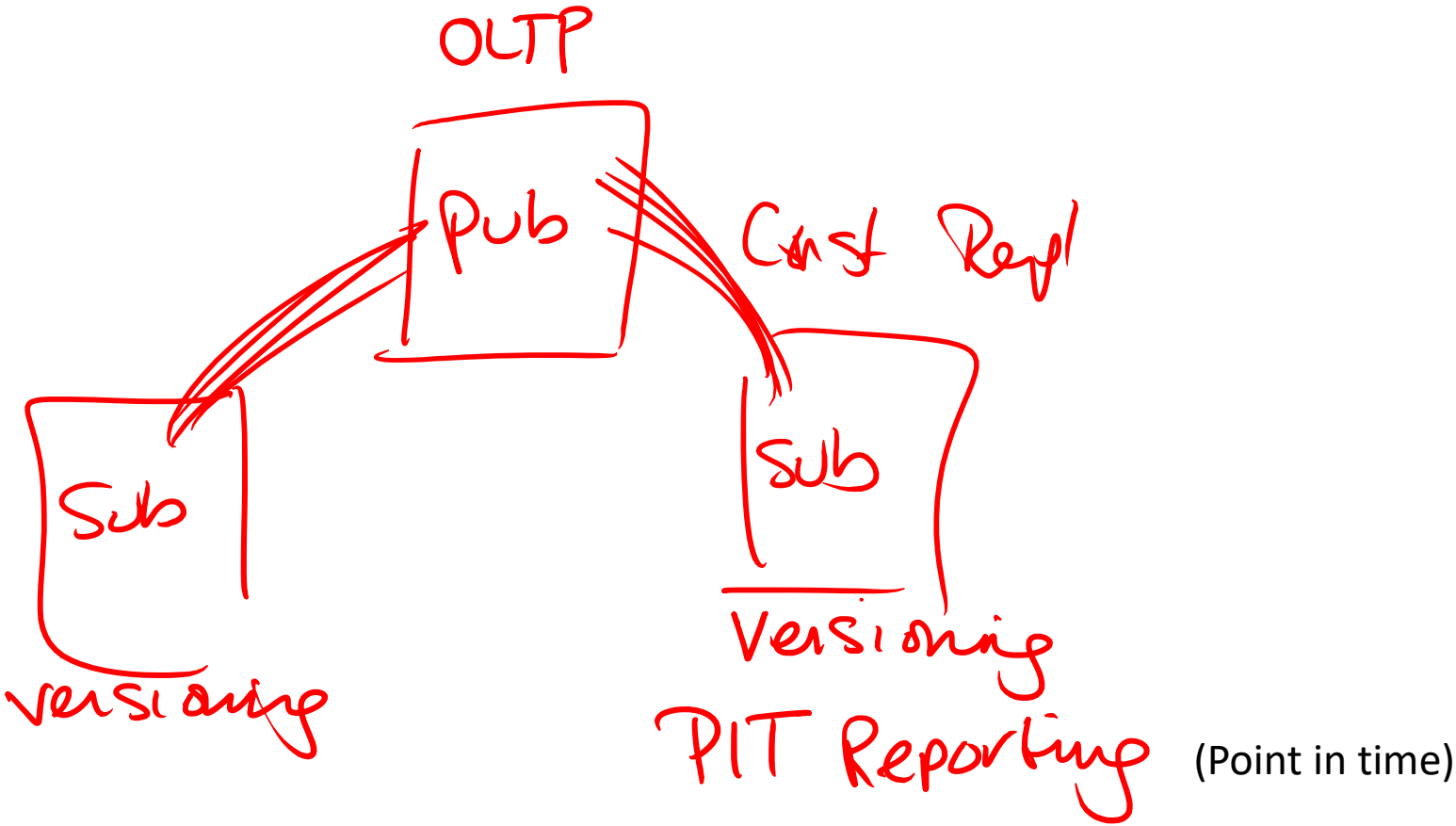
The key things to remember from the picture is that (1) has "hiccups" along the way as they encounter rows that are locked. They read… wait… read… wait. This is *partially* what slows down a statement. However, it's worse that this. It's possible that AFTER a row is read, it will appear again (if the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

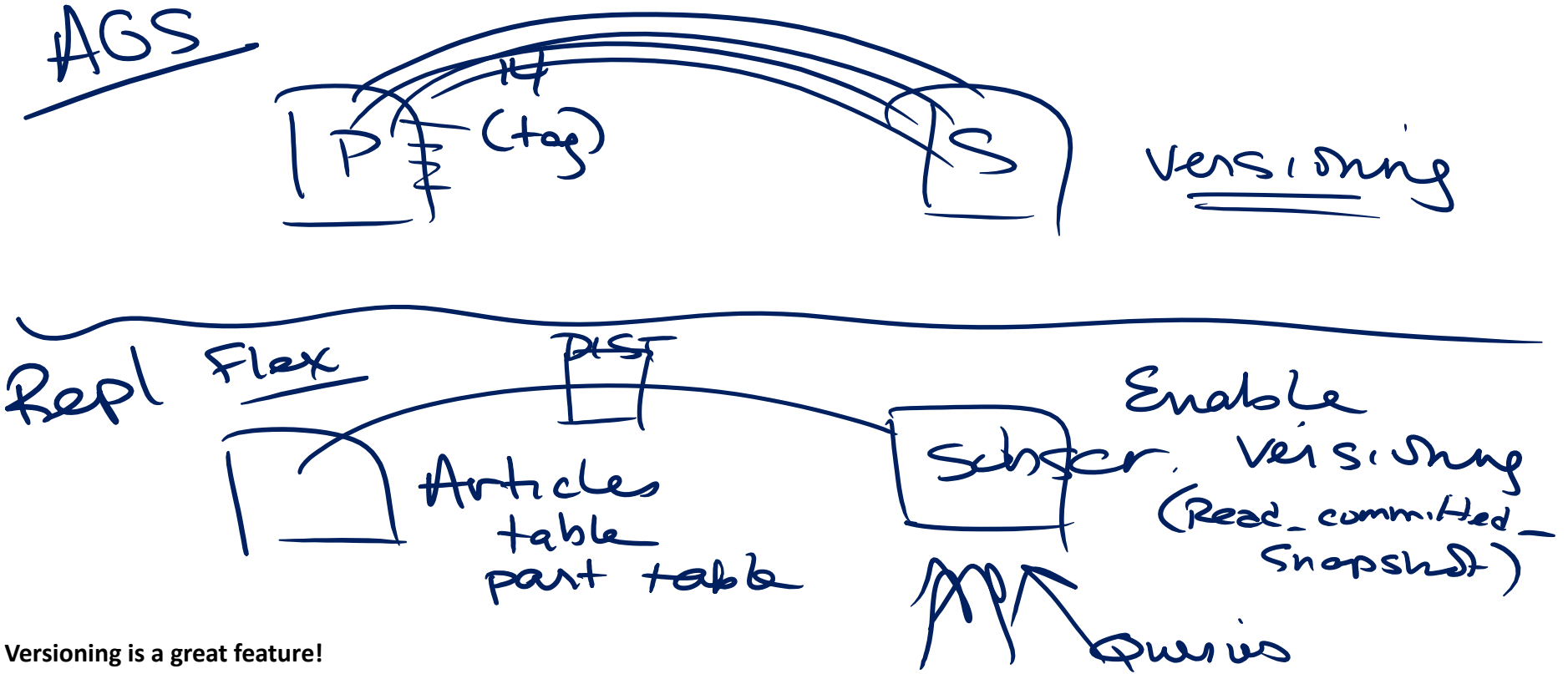
(3) Versioning (and it was implied that it was statement-level)

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

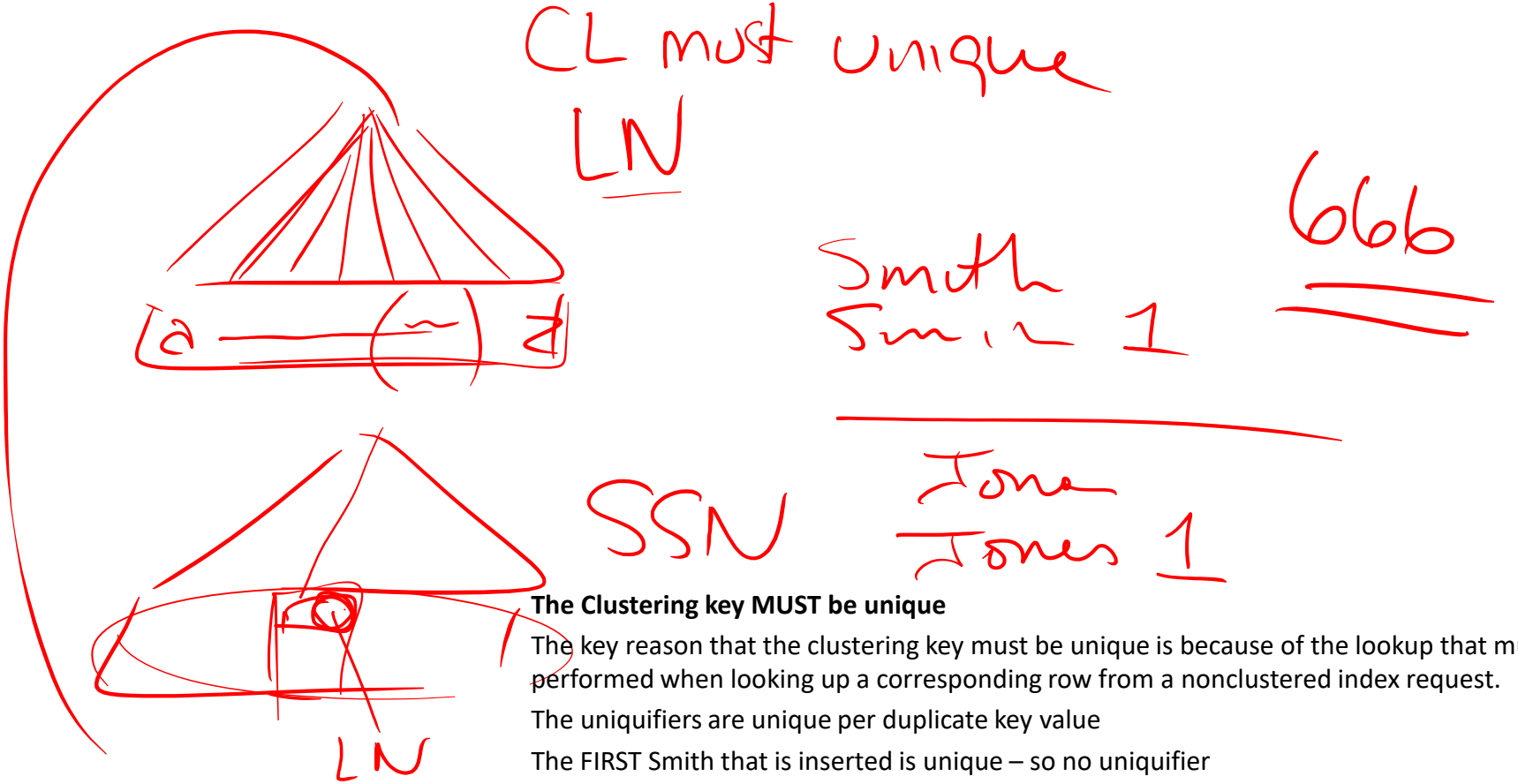
The idea is that you don't want to impact your production OLTP environment with READERs or with versioning so instead, you replicate to subscribers and use read committed snapshot there!



Versioning is a great feature!
AGs use part of versioning on the primary so that the read-only secondaries can support versioning as changing are sent.

If you use replication, definitely consider setting up versioning (read_committed_snapshot) so that reporting users don't block replication AND replication doesn't block the reporting users!

Check out my whitepaper on versioning: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2005/administrator/ms345124(v=sql.90)



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

CL DATE

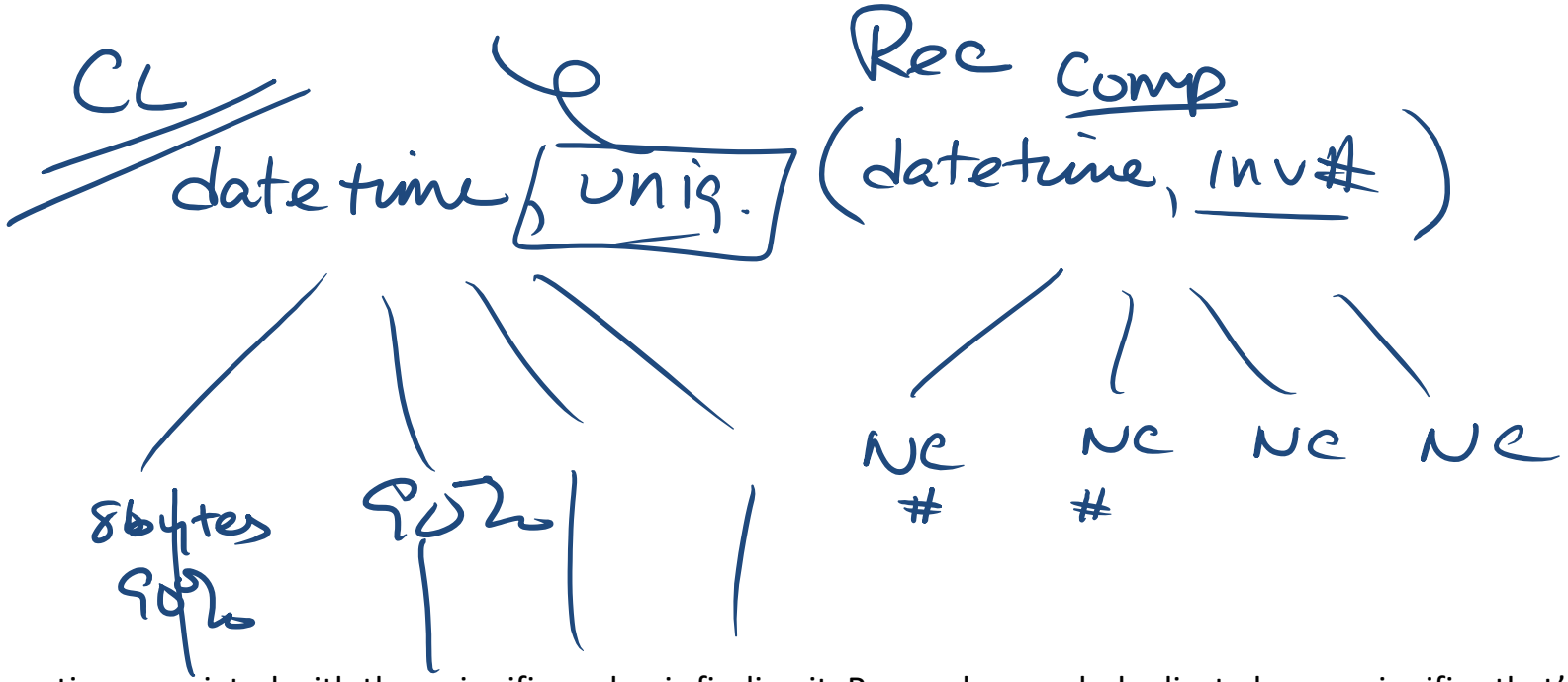
Int 2.147 bill

Error 666

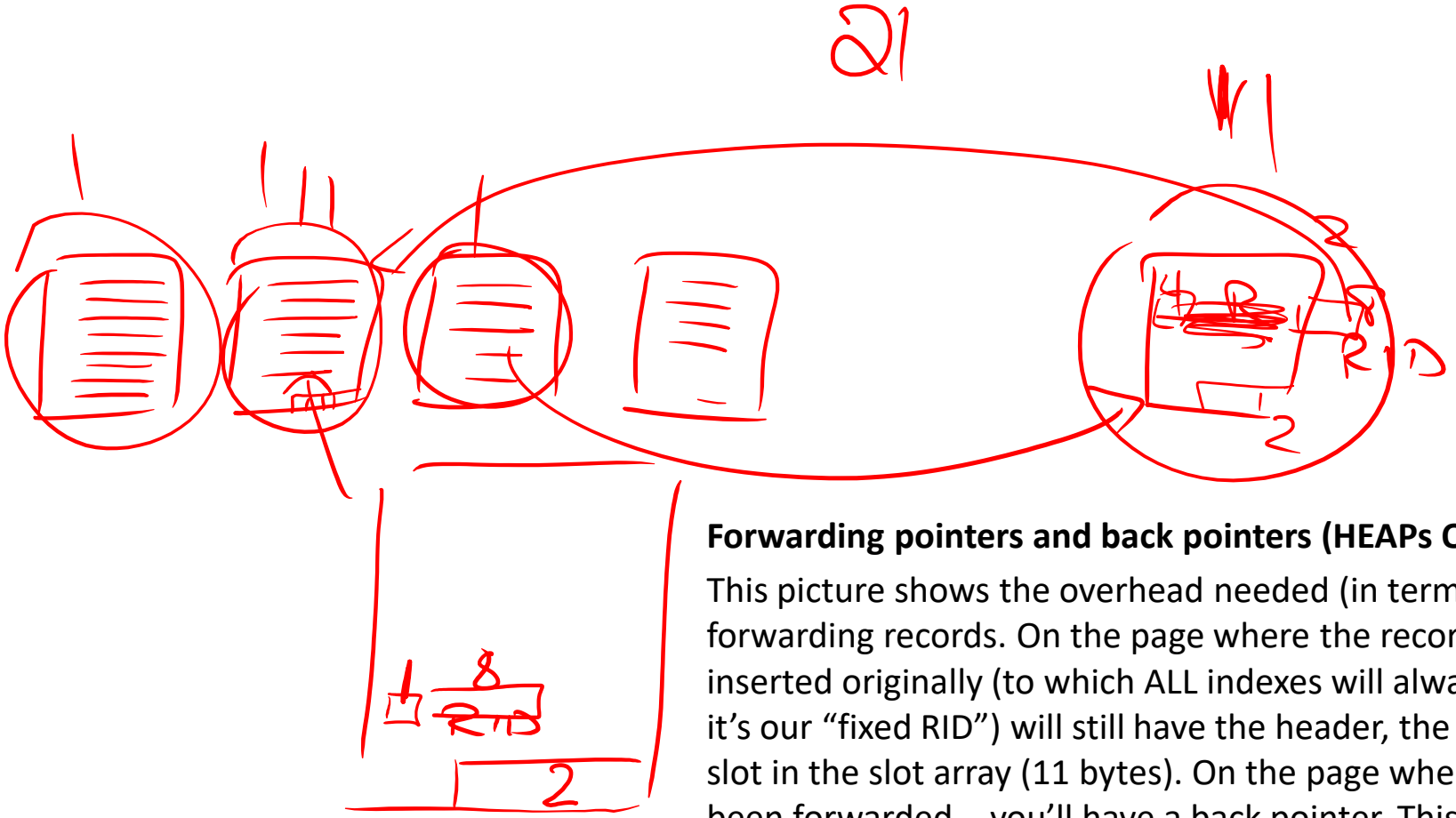
A clustering key should be unique by definition. However, if the CL key is NOT unique then SQL Server will uniquify it. And, this process of uniquification can be expensive because the next uniquifier value is unknown (they must scan the last page of values in that duplicate range).

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.

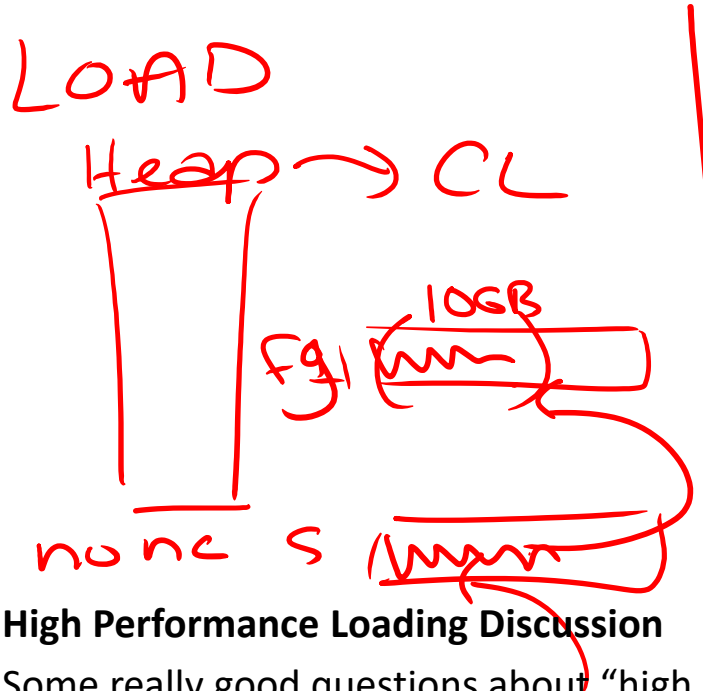


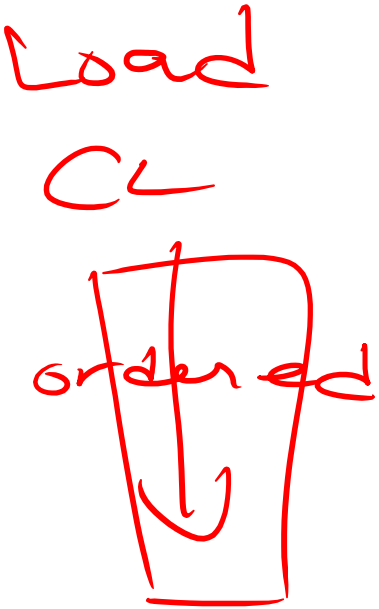
Another negative associated with the uniquifier value is finding it. Remember, each duplicate has a uniquifier that's tied only to it. On insert, SQL Server will have to scan the last page to find the maximum *current* value for the uniquifier for that "group." If you have something like a DATE (not 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.



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

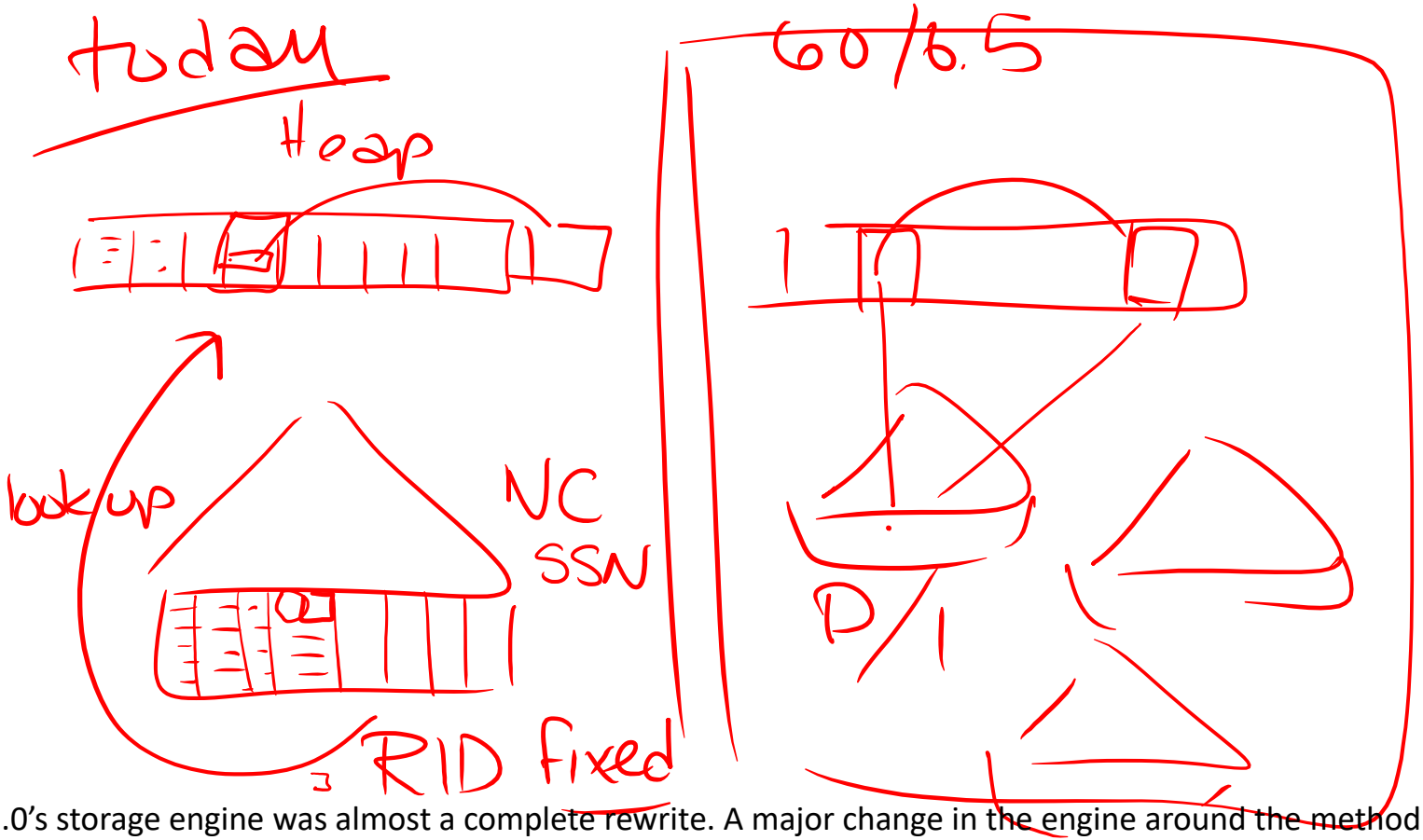
Some really good questions about "high performance loading" that lead to a discussion around these key things:

- **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.

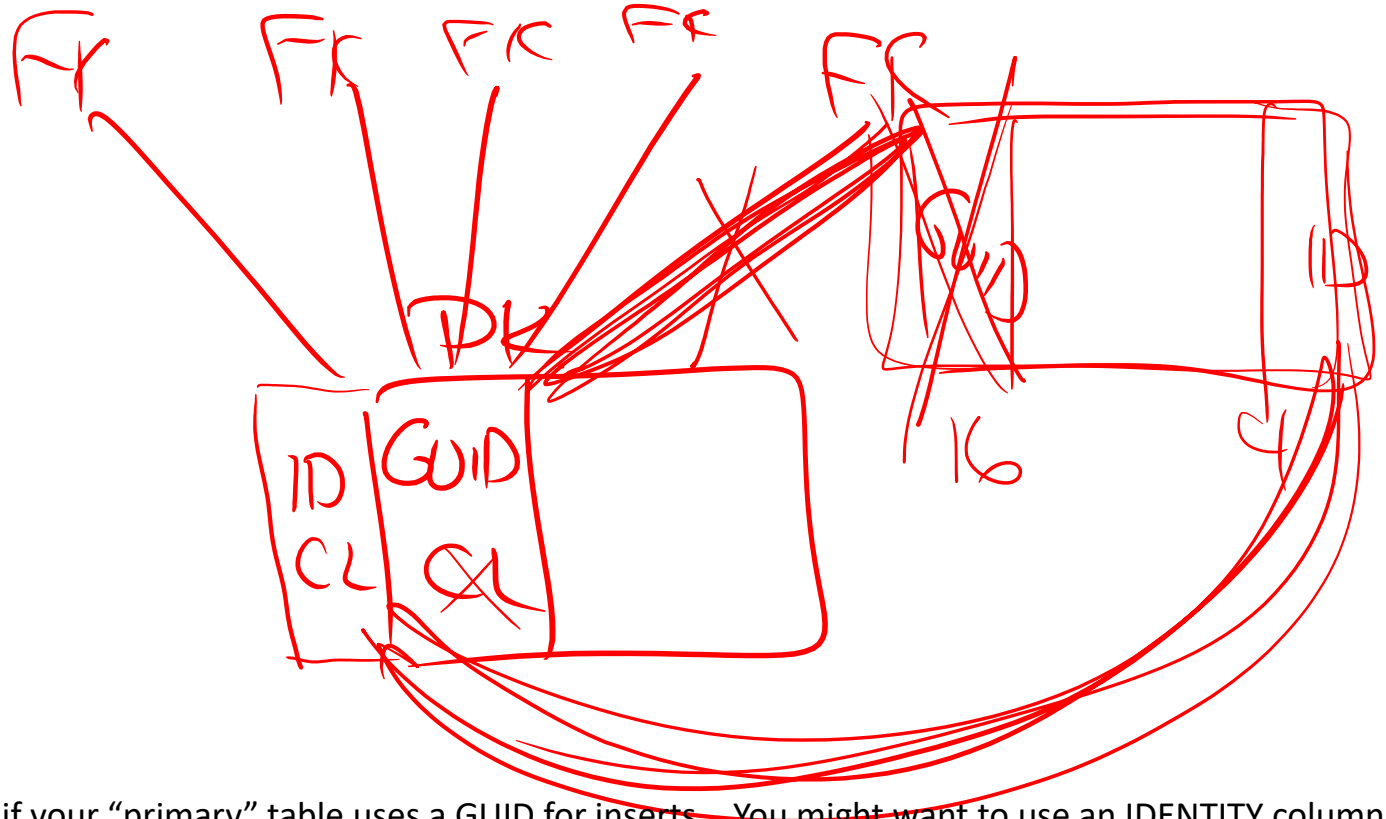
Clustering for range queries looks GREAT on paper… until modifications occur!

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…

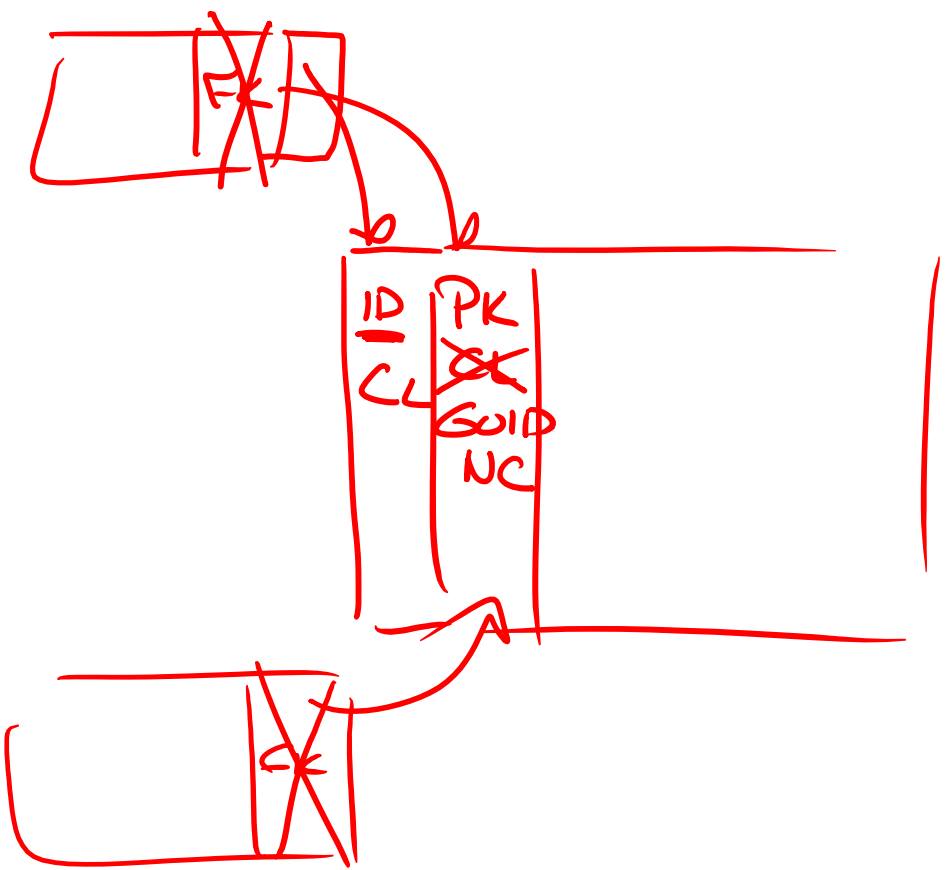
- INSERTs/UPDATEs are slower due to the splits
- This wastes disk space AND memory because of the fragmentation caused.
- And the range query is no longer left-right ordered but instead very out of order (fragmentation)



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).



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 free of GUIDs as clustering keys and GUIDs for joins! You might still have a GUID on that initial table but only as a nonclustered PK. ☺

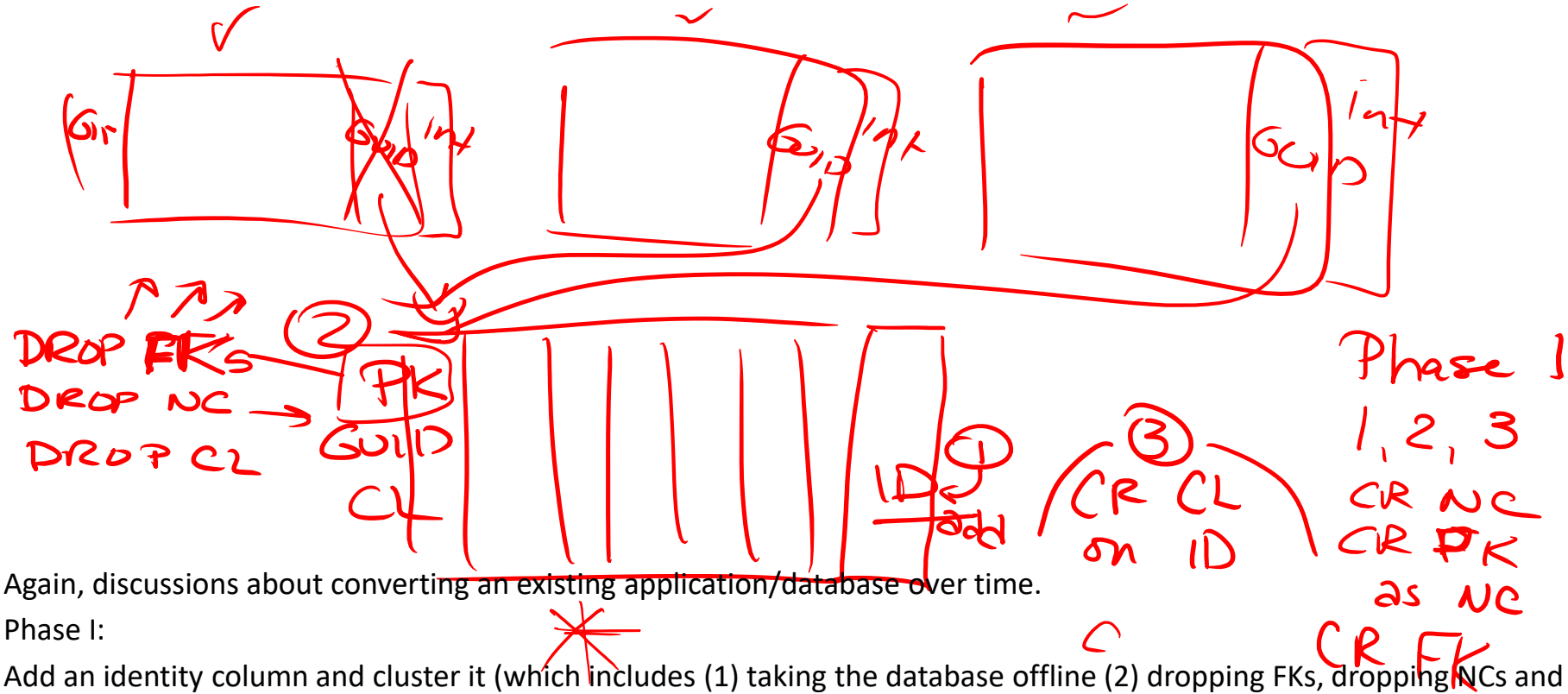


Take offline
DROP FKs
DROP NCs
DROP CL\PK
Add new column
CR CL
CR PK NC
CR NCs
CR FKs
Bring Online

What does it take to change a CL PK?

Unfortunately, A LOT. And, because of the lack of indexes and foreign keys – the entire process is OFFLINE.

This is why this is such an important decision to make early!



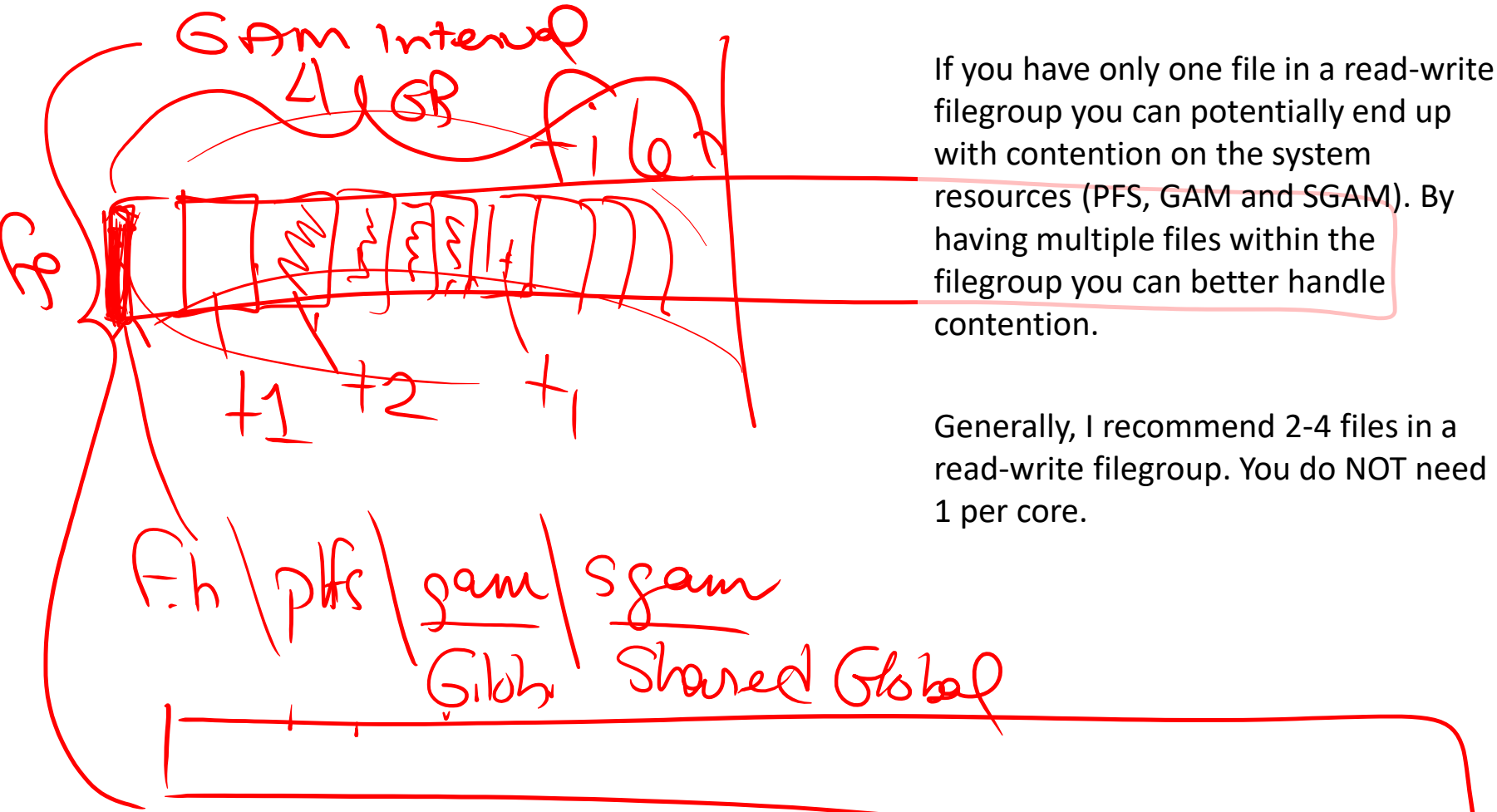
Again, discussions about converting an existing application/database over time.

Phase I:

Add an identity column and cluster it (which includes (1) taking the database offline (2) dropping FKs, dropping NCs and finally, dropping the CL. (3) Add the identity column and cluster it. (4) Create the PK as NC. (5) Add back the other NCs. (6) Add back the FKs.

Phase II (you can do this for a subset of tables at a time, repeat until all are gone):

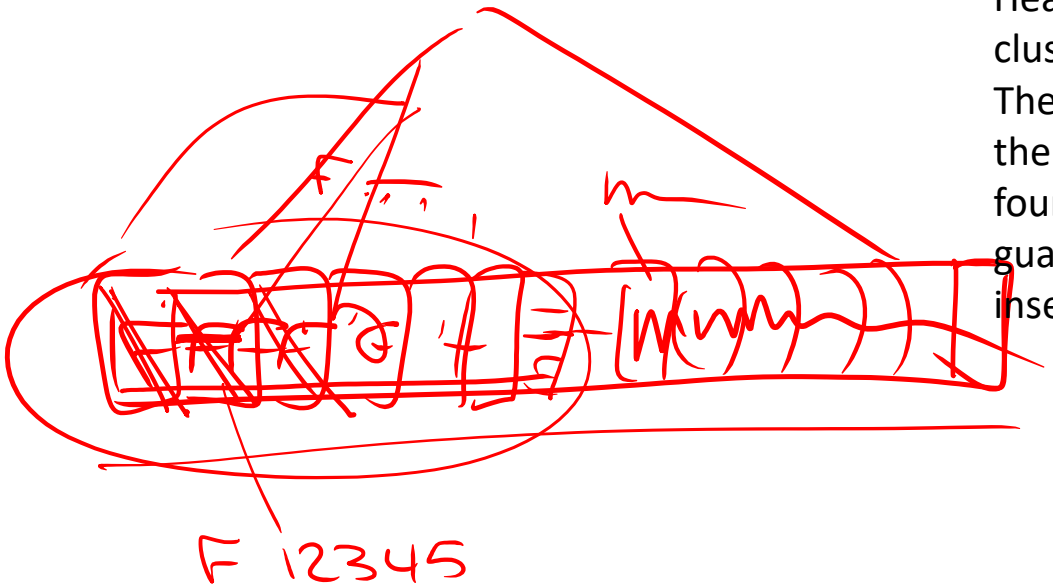
Migrate the related tables to use the Identity column for joins instead of the GUID-based FK. Slowly change code and eventually drop the GUID-based FK leaving ONLY the int-based FKs.



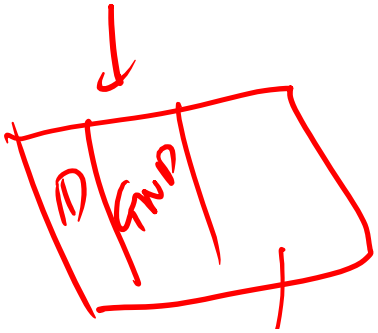
If you have only one file in a read-write filegroup you can potentially end up with contention on the system resources (PFS, GAM and SGAM). By having multiple files within the filegroup you can better handle contention.

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

NC Ind Gender



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.

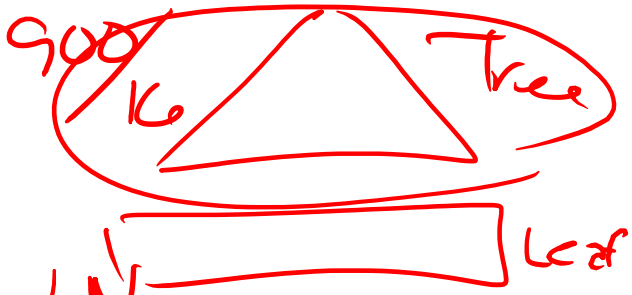
Table CL on EmpID

Key SSN

Inc LN

Key SSN

Leaf SSN, EmpID, LN



---

non-unique index on phone

Key phone

Inc LN

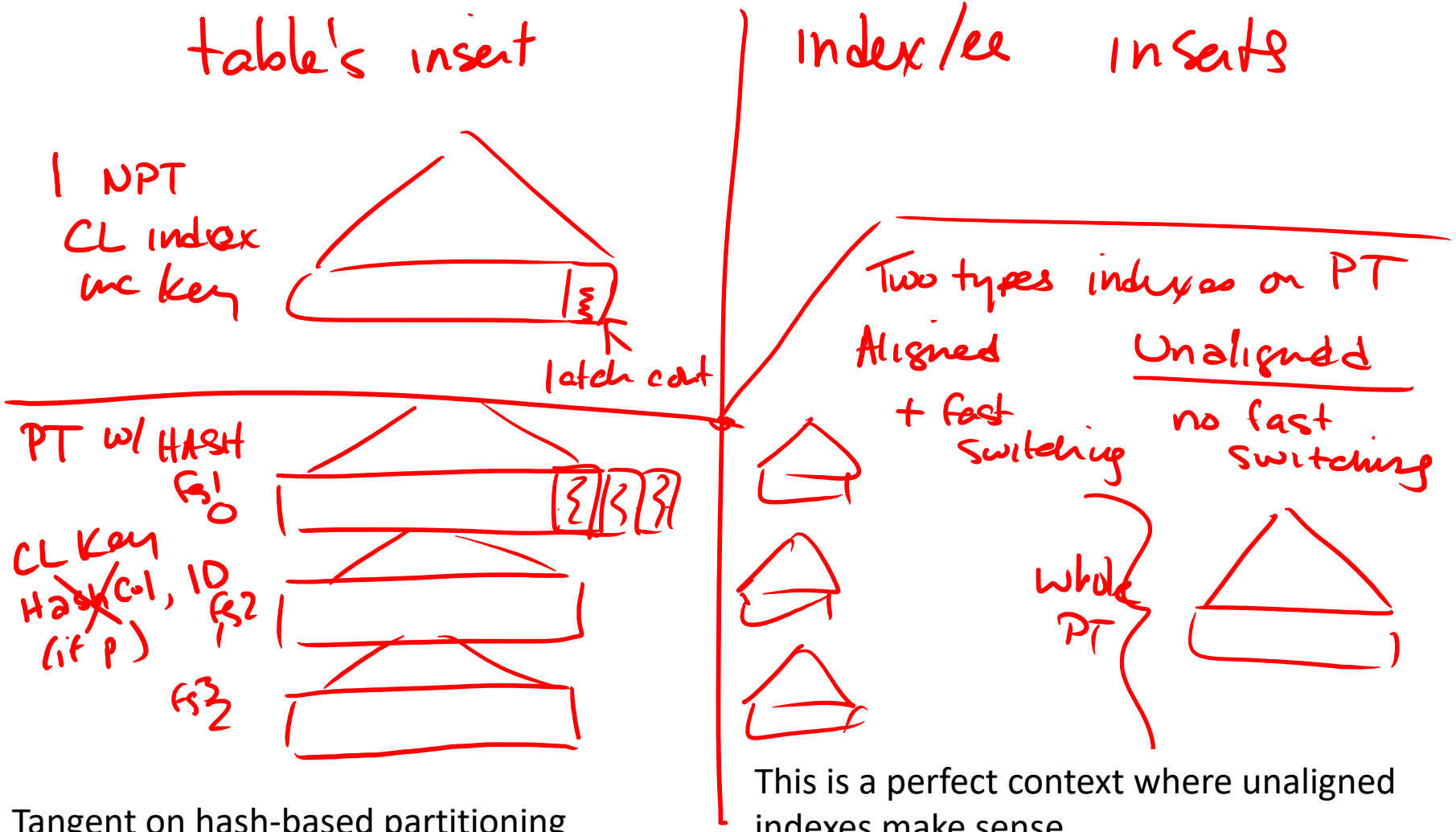
Key phone, EmpID

Leaf phone, EmpID, LN

Where does the CL key go?

If the index is unique – then the CL is added only to the leaf-level.

If the nonclustered is non-unique – then the CL is added to leaf level as well as up the tree. Specifically, it's added immediately after the key; included columns are added after that.



Tangent on hash-based partitioning

This is a perfect context where unaligned indexes make sense.

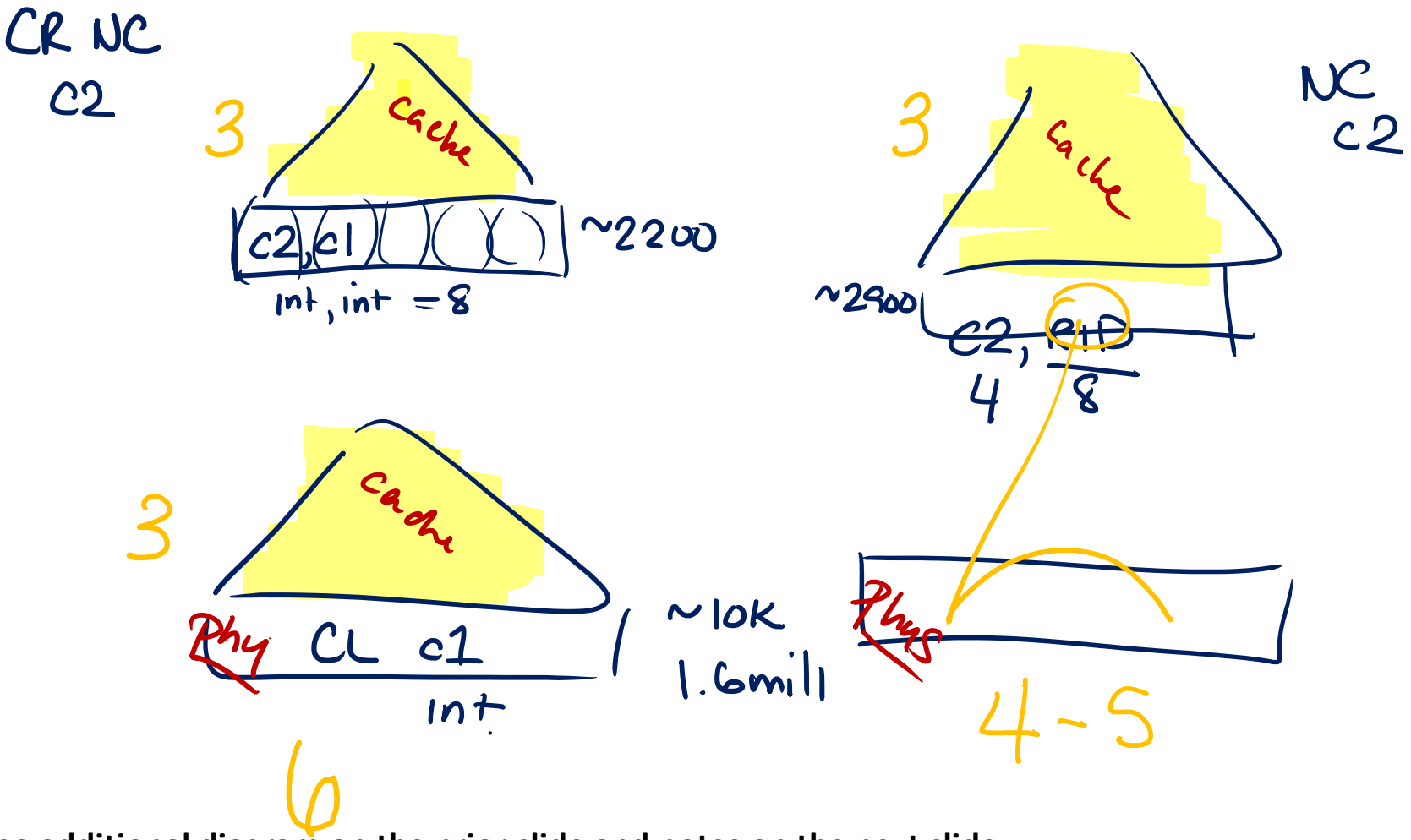
SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 8: Internals and Data Access

Kimberly L. Tripp

Kimberly@SQLskills.com



See additional diagram on the prior slide and notes on the next slide

Question/Discussion

The two previous diagrams were discussing the costs (of IOs) for lookups – between a heap and clustered table.

It *looks* like the CL index is worse:

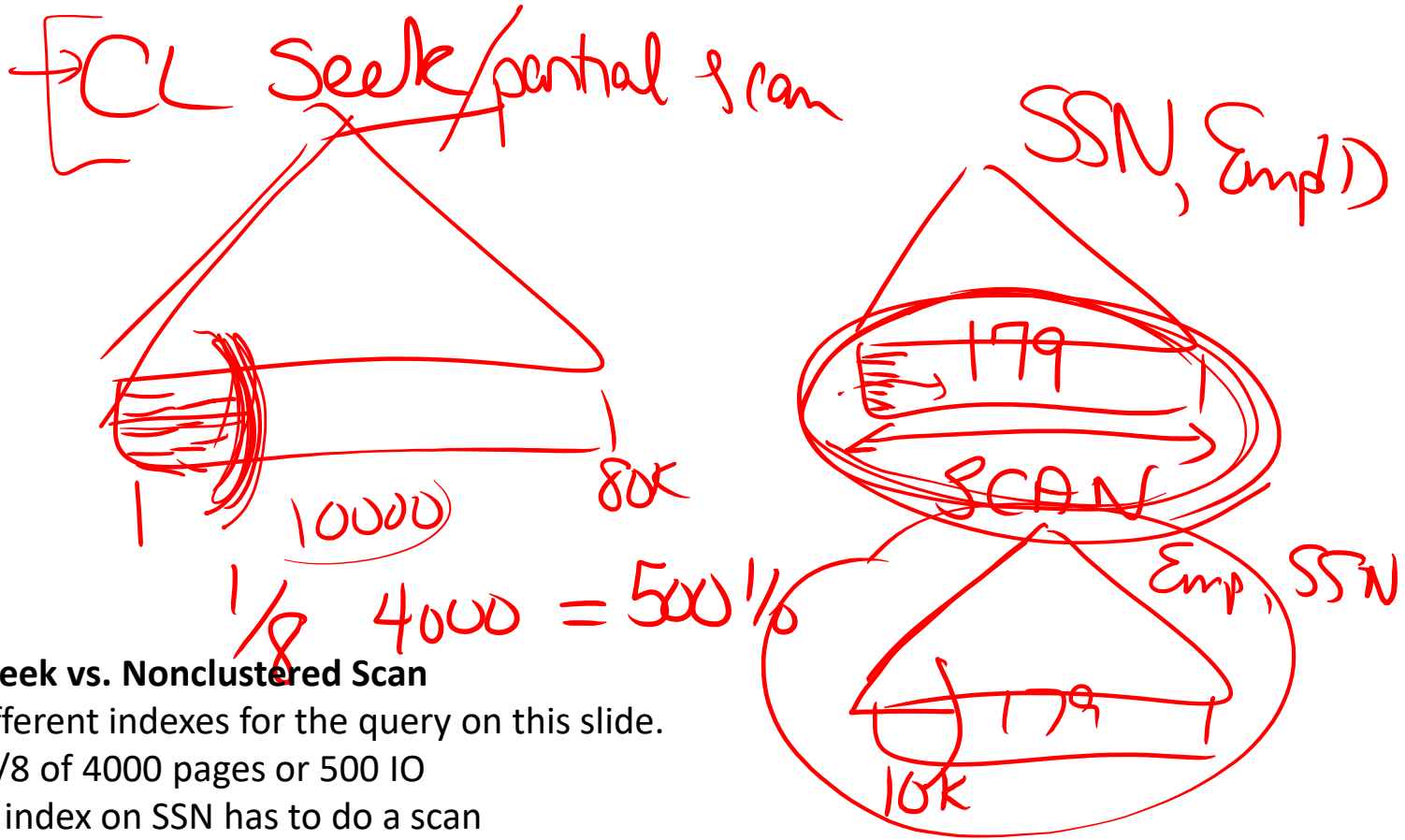
> Using a nonclustered to do a lookup (= 3 IOs), then – using the CL to lookup the data row = 3 more IOs for a total of 6 IOs

Whereas the heap 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 / forwarding) for a total of 4-5 IOs.

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

As a result, a bookmark lookup from a NC to a clustered has 2 potentially physical IOs.
The lookup from a NC to a heap has potentially 2-3 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. ☺



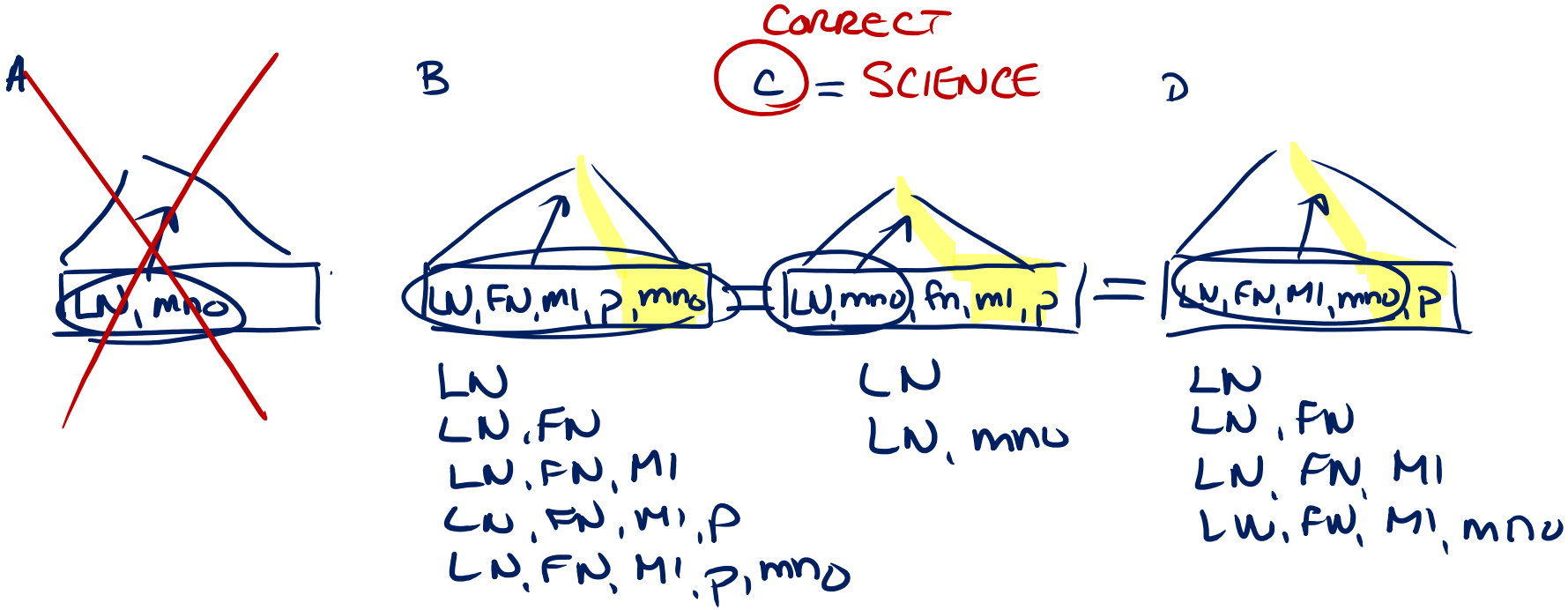
Clustered Index Seek vs. Nonclustered Scan

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.



Module 8, Slide 29

For the member table (in the Credit sample database), the clustered index key is on member_no. To save space, I'm listing that as mno.

If this were a question on a test – the answer to "which index is best?" would be "c"

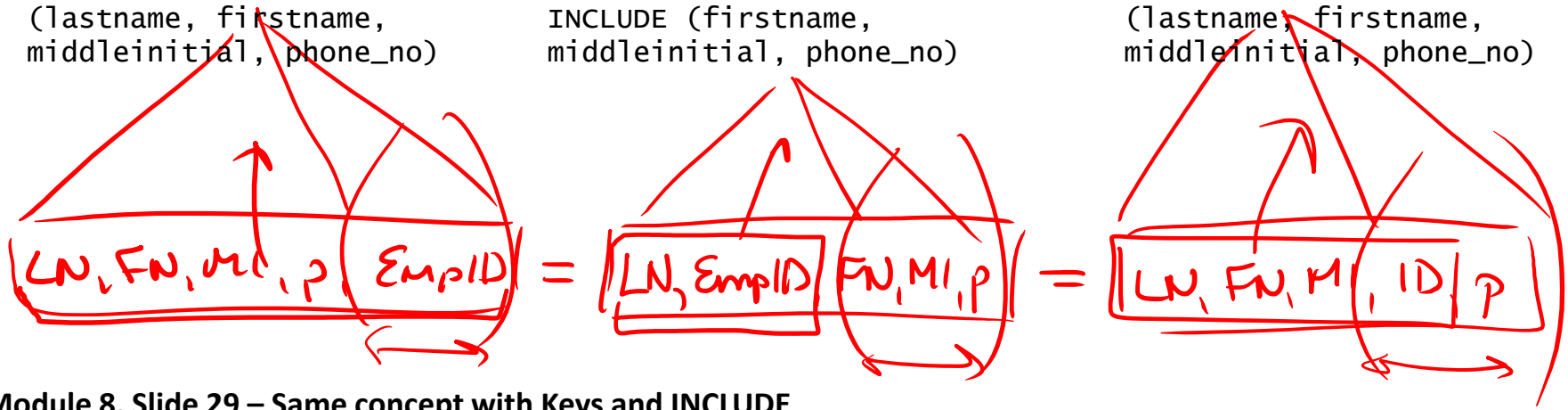
A is useless (it's NOT selective enough)

B = C = D for THIS query and in terms of I/O but C is best as it's ALL that's required. You do NOT need more in the key for THIS query. D is the "art" of indexing as it's likely the result of consolidating this index with an existing index.

```
CREATE INDEX NCICovers4Cols
ON dbo.member
(lastname, firstname,
middleinitial, phone_no)
```

```
CREATE INDEX NCILNinKeyInc3Cols
ON dbo.member(lastname)
INCLUDE (firstname,
middleinitial, phone_no)
```

```
CREATE INDEX NCICovers4Cols
ON dbo.member
(lastname, firstname,
middleinitial, phone_no)
```



Module 8, Slide 29 – Same concept with Keys and INCLUDE

In the first example all of the columns are in the key. The clustering key is added to the leaf-level and then all 5 columns go up the tree. (NOTE: The CL Key is an ID… member_no is the actual column but this pic says EmpID as a generic “key”)

In the second example only lastname is in the key. The clustering key is added immediately after that (in the leaf-level) and then the included columns are added to the leaf-level. Then, only lastname and EmpID go up the tree. This is the best index for this query (if we were to tune query-by-query, which is completely unrealistic).

In the final example phone is the only include. The clustering key is added immediately after LN, FN, MI (in the leaf-level) and then phone is added. LN, FN, MI, EmpID go up the tree. This last example is the most realistic index but only after consolidation warrants it.

```
SELECT … FROM member
WHERE Lastname = 'Tripp' AND Firstname LIKE 'K%'
      AND MI = 'L'
```

Index 1

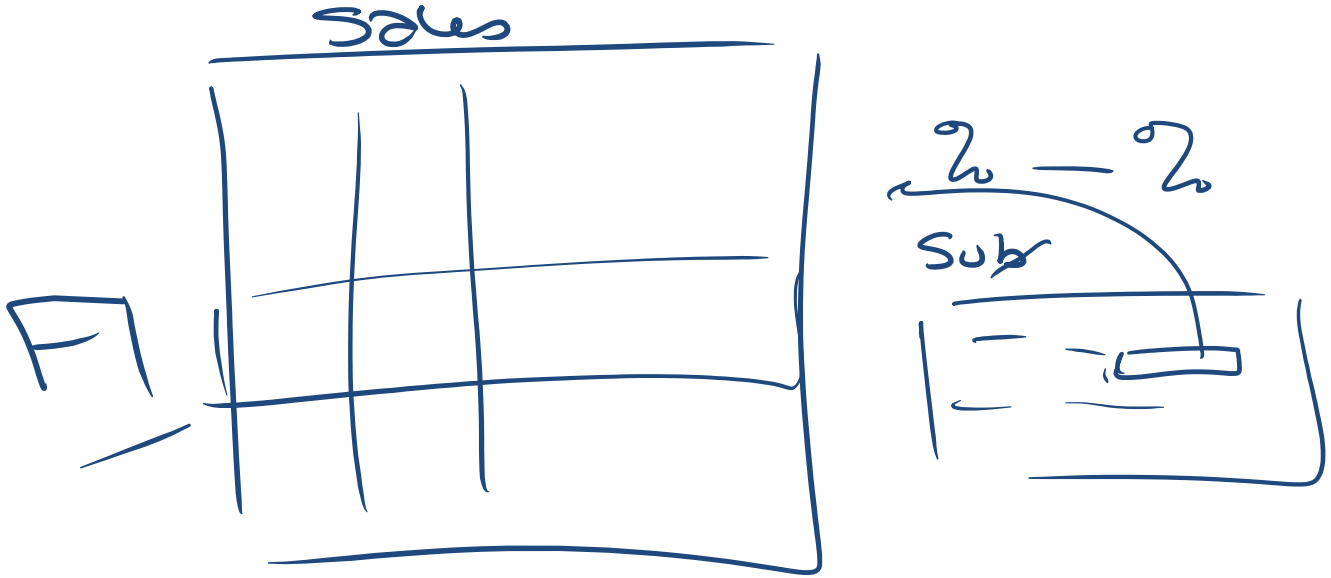
Lastname	Firstname	MI
Tripp	Andrea	L
Tripp	Catherine	S
Tripp	Kimberly	L
Tripp	Lisa	A
Tripp	Zachary	L

Index 2

Lastname	MI	Firstname
Tripp	A	Lisa
Tripp	L	Andrea
Tripp	L	Kimberly
Tripp	L	Zachary
Tripp	S	Catherine

Which index order is better for this query?

Definitely the second one (index 2) because the criteria against firstname is “range-based” while the condition against MI is equality-based.

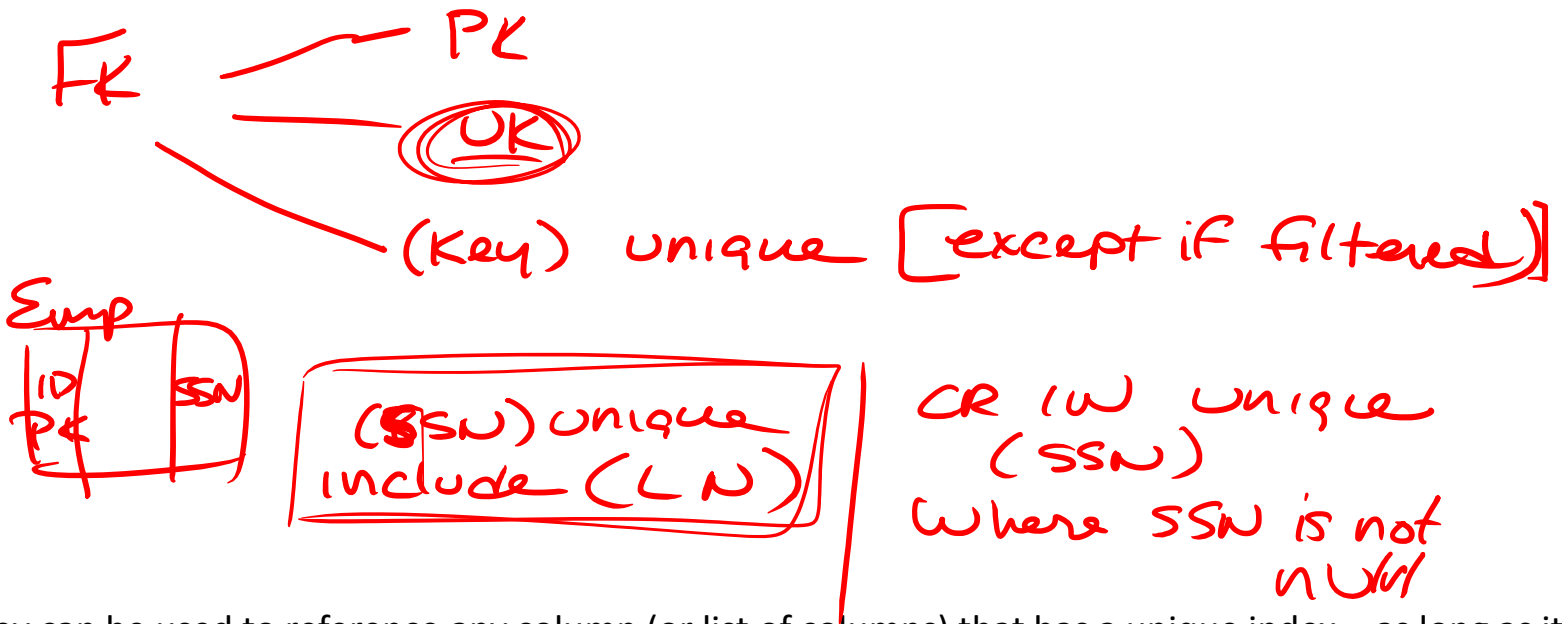


Solving an application problem with a filtered index

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!

The negative – consistent session settings are required AND procedures must do a recompile to ensure that they will pick up the filtered index.



A foreign key can be used to reference any column (or list of columns) that has a unique index – as long as it's not filtered.

In an index consolidation scenario, if you already have a constraint on SSN and you NEED an index on SSN include (LastName) you'd end up with a "redundant" index (if SQL *required* a constraint-based index for foreign keys). Instead, you can drop the constraint on SSN and instead create:

```
CREATE UNIQUE NONCLUSTERED INDEX SSN_Inc_LN ON Member (SSN)
INCLUDE (LastName)
```

Foreign Keys can reference UNIQUE indexes (without constraints)
http://www.sqlskills.com/BLOGS/KIMBERLY/post/Foreign-Keys-can-reference-UNIQUE-indexes-(without-constraints).aspx

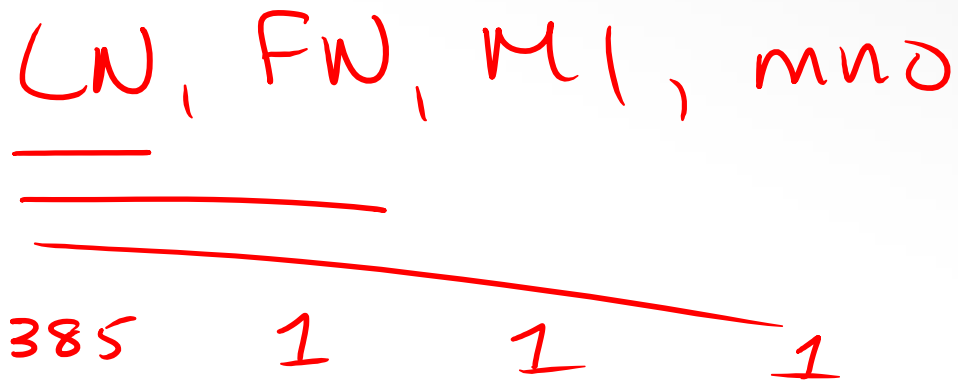
SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 9: Statistics – Internals and Updates

Kimberly L. Tripp

Kimberly@SQLskills.com

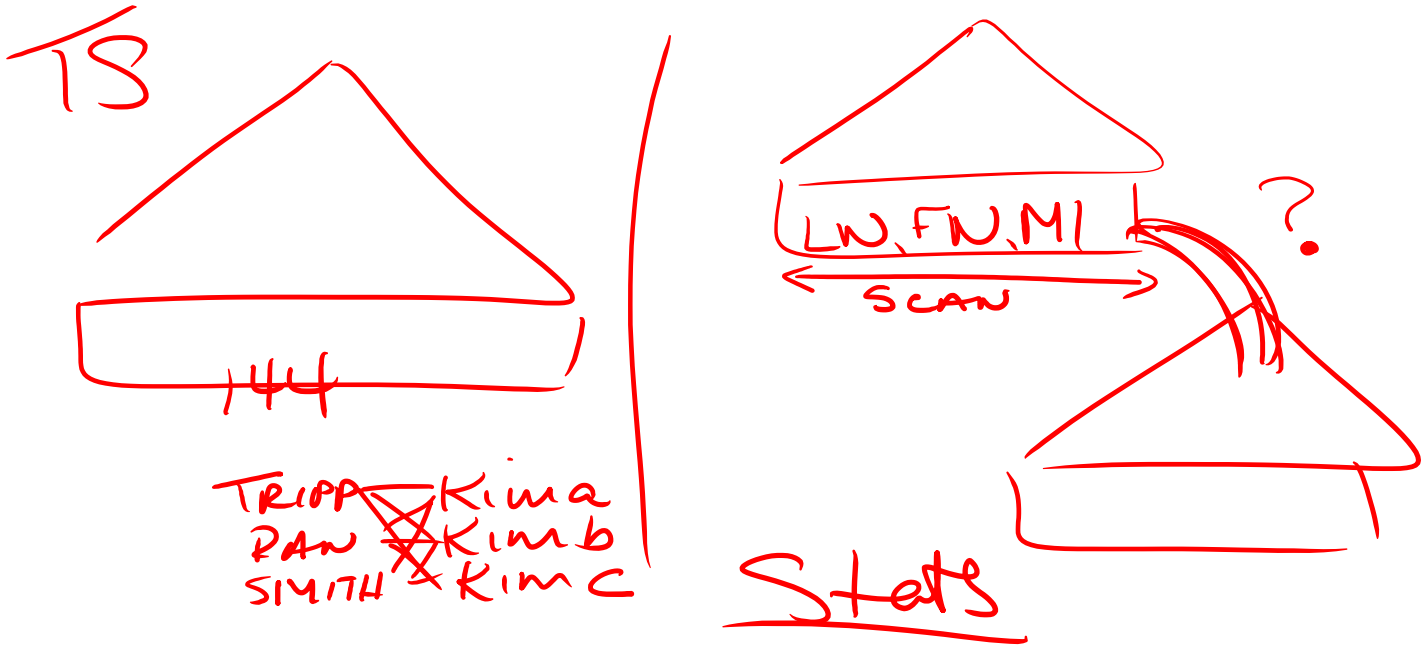


Data selectivity and "need" for additional columns in the key - from left-based density subsets…

If the distribution of the data is unique at the combination of the first and second columns then the third, forth, etc. do not provide any use in terms of seeking (JUST in terms of seeking). However, they might provide use for sorting.

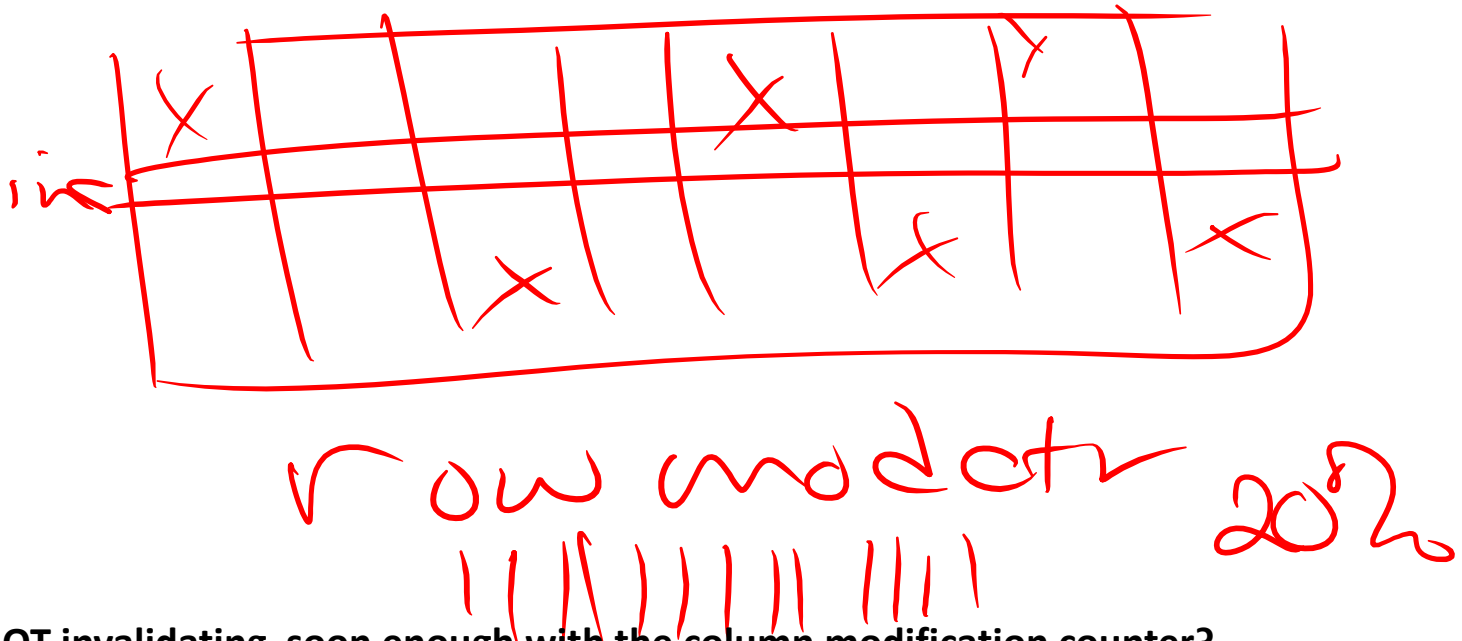
But, it MIGHT be possible to consolidate another index with this one IF you really don't need those extra columns in the key. Something to consider!

Multiple – all density * rows and if unique (= 1 or very close to one) then you can *consider* consolidation with other similar indexes!



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.



We are NOT invalidating soon enough with the column modification counter?

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).

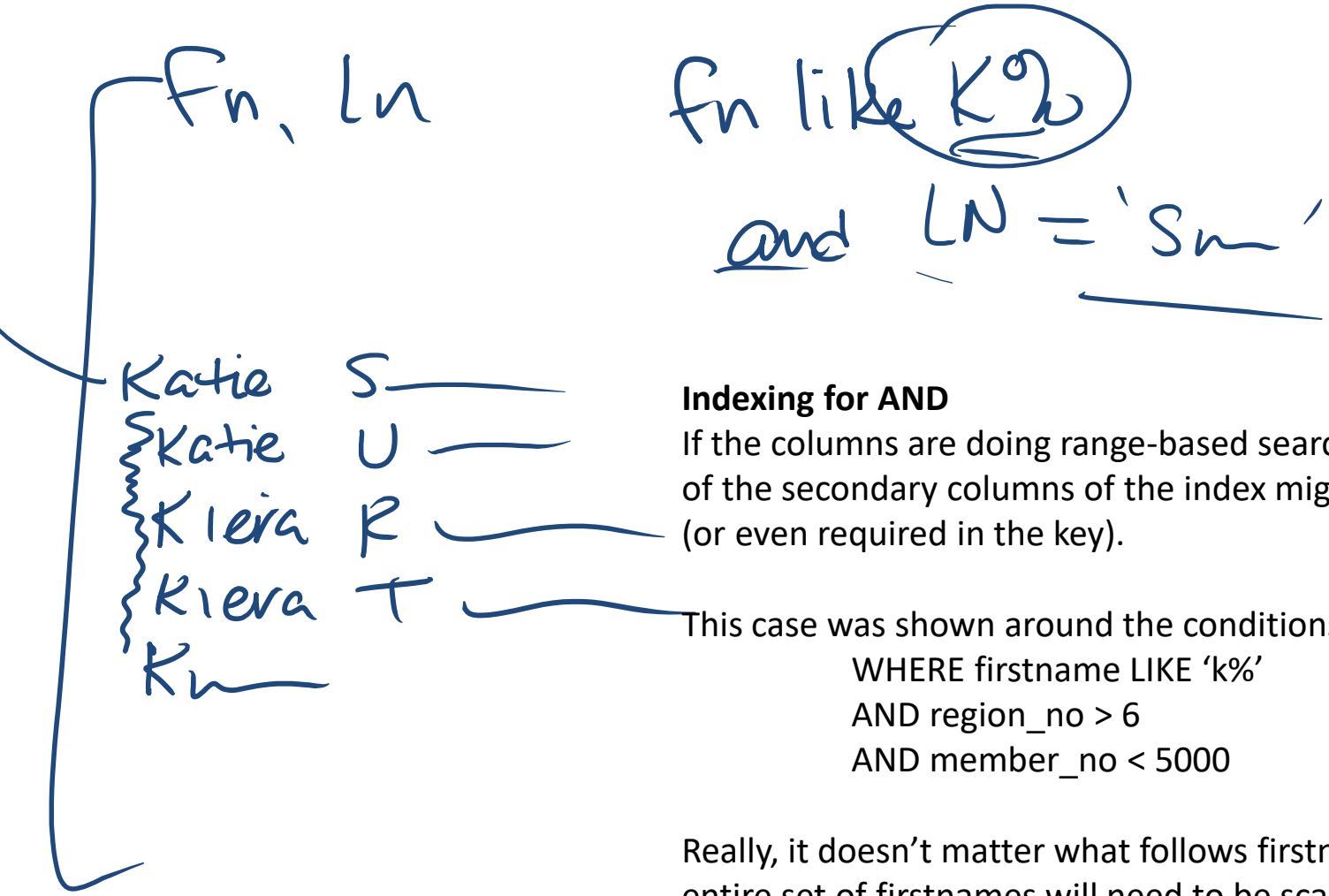
SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 10: Indexing Strategies

Kimberly L. Tripp

Kimberly@SQLskills.com



Indexing for AND

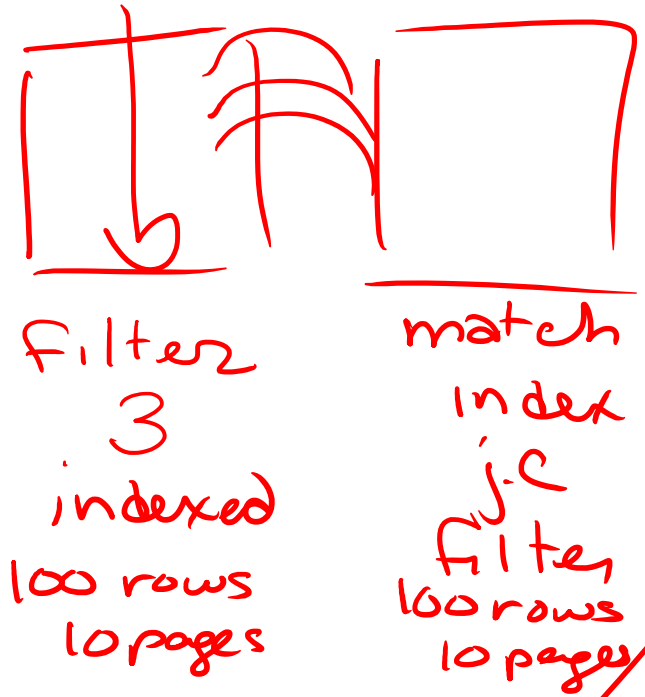
If the columns are doing range-based searching then the order of the secondary columns of the index might not be relevant (or even required in the key).

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.

Loop Joins



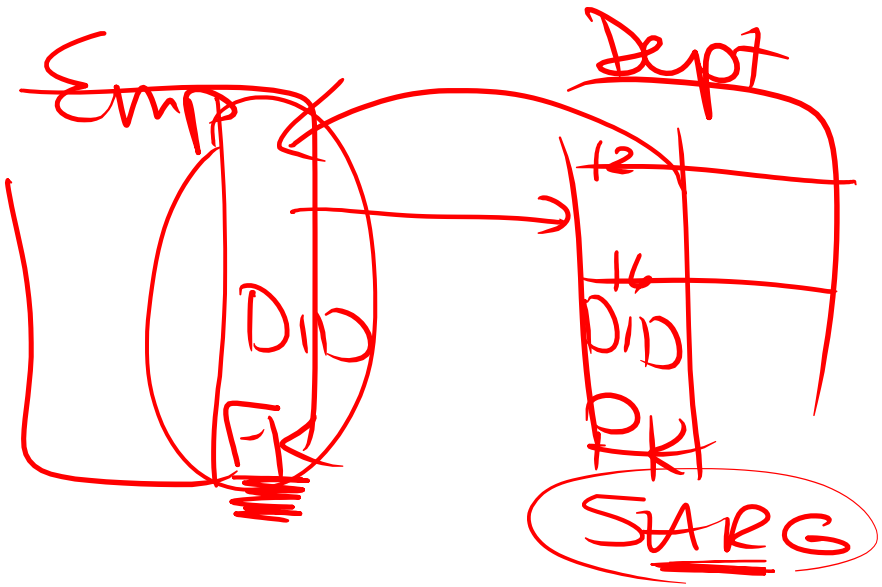
Here we started to talk about how LOOP joins are an iterative process. The driver (the outer/first table) is typically chosen because it has the most selective set. Any of the tables that have a highly selective search argument are more likely to be chosen as the driver. An index that aids in efficiently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

- IF -
no good indexes & SQL HAD to do loop
SCAN
10 + (100 x 10)

In the worst case scenario the costing of this iterative process can be very high/expensive.

Instead of doing this, SQL Server is much more likely to do a loop join.



where d.city = 'Redmond'

Loop Joins continued

If our query was:

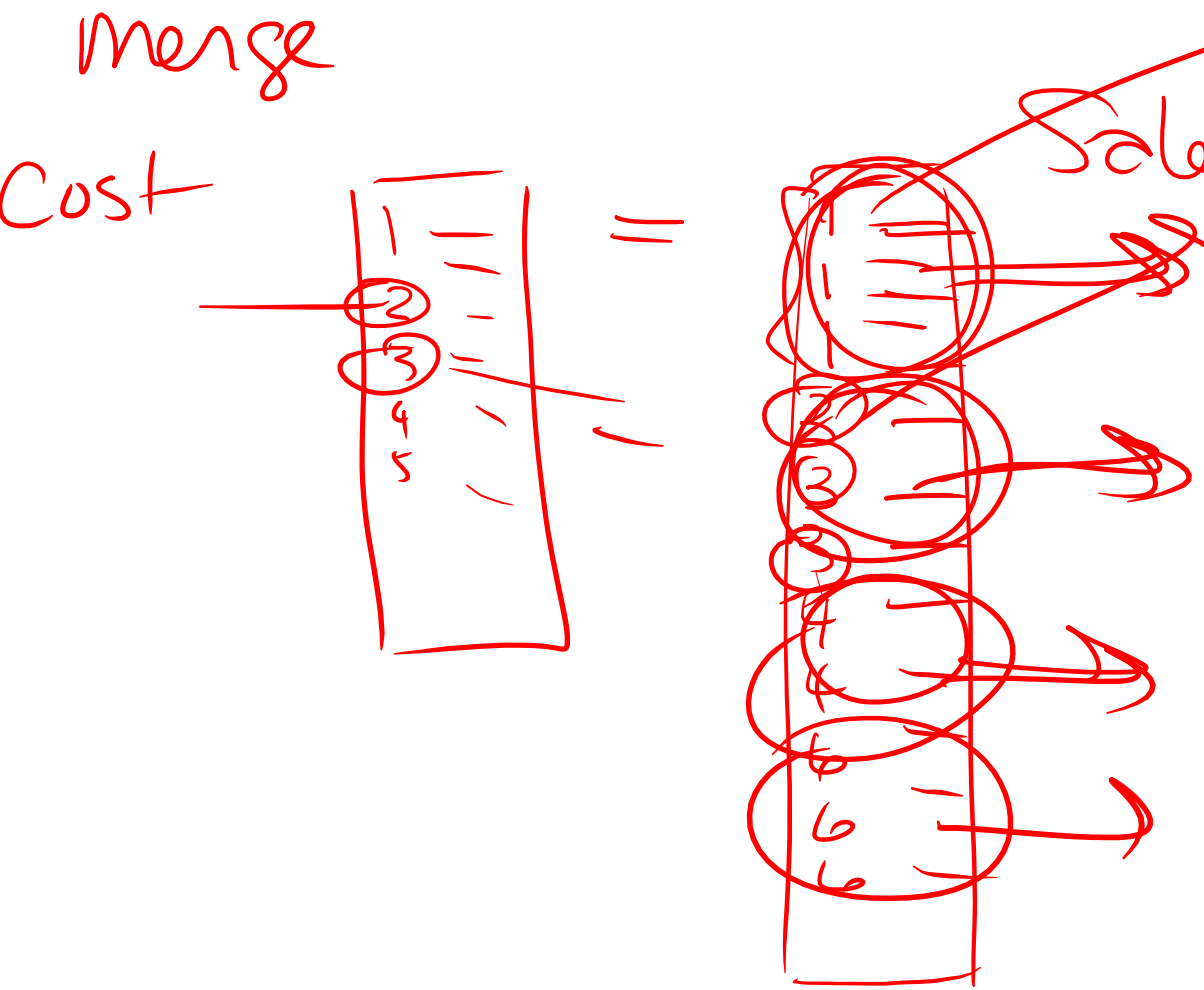
```
SELECT columns (irrelevant here)
FROM Employee AS E
        JOIN Department AS D
                ON E.DID = D.DID
WHERE D.city = 'Redmond'
```

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?



Merge Joins

Merge joins leverage "suitably sorted sets."

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

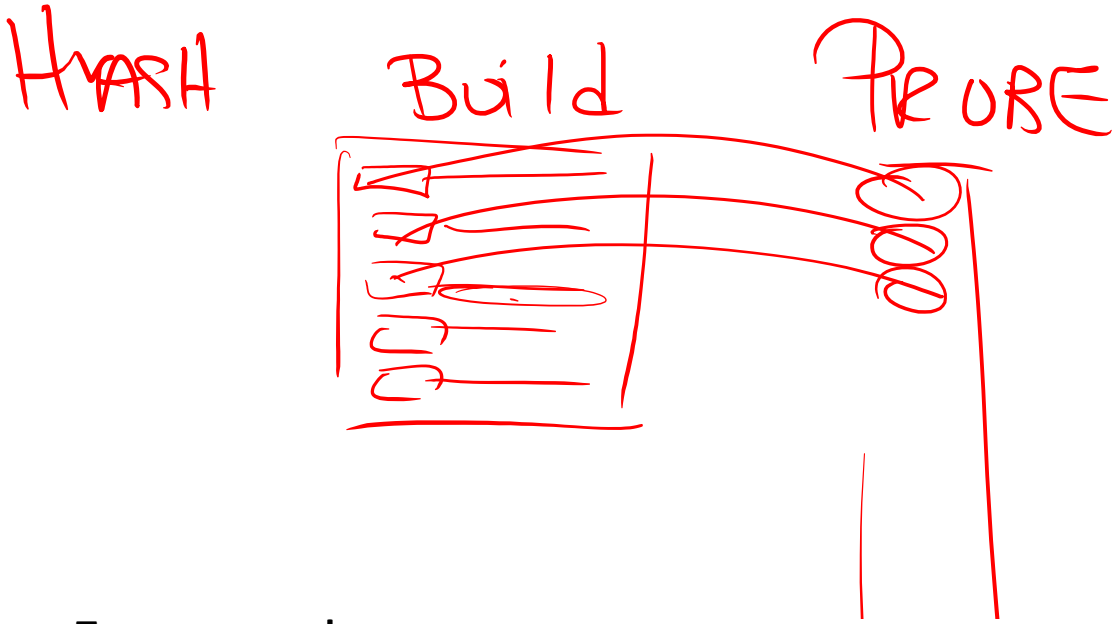
Key question:

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.



Two very good resources:

Hash joins and hash teams in Microsoft SQL Server
by Goetz Graefe, Ross Bunker, Shaun Shaun Cooper
(http://bit.ly/1GVJJnn)

Query Evaluation Techniques for Large Databases
by Goetz Graefe
(http://bit.ly/1bX84f3)

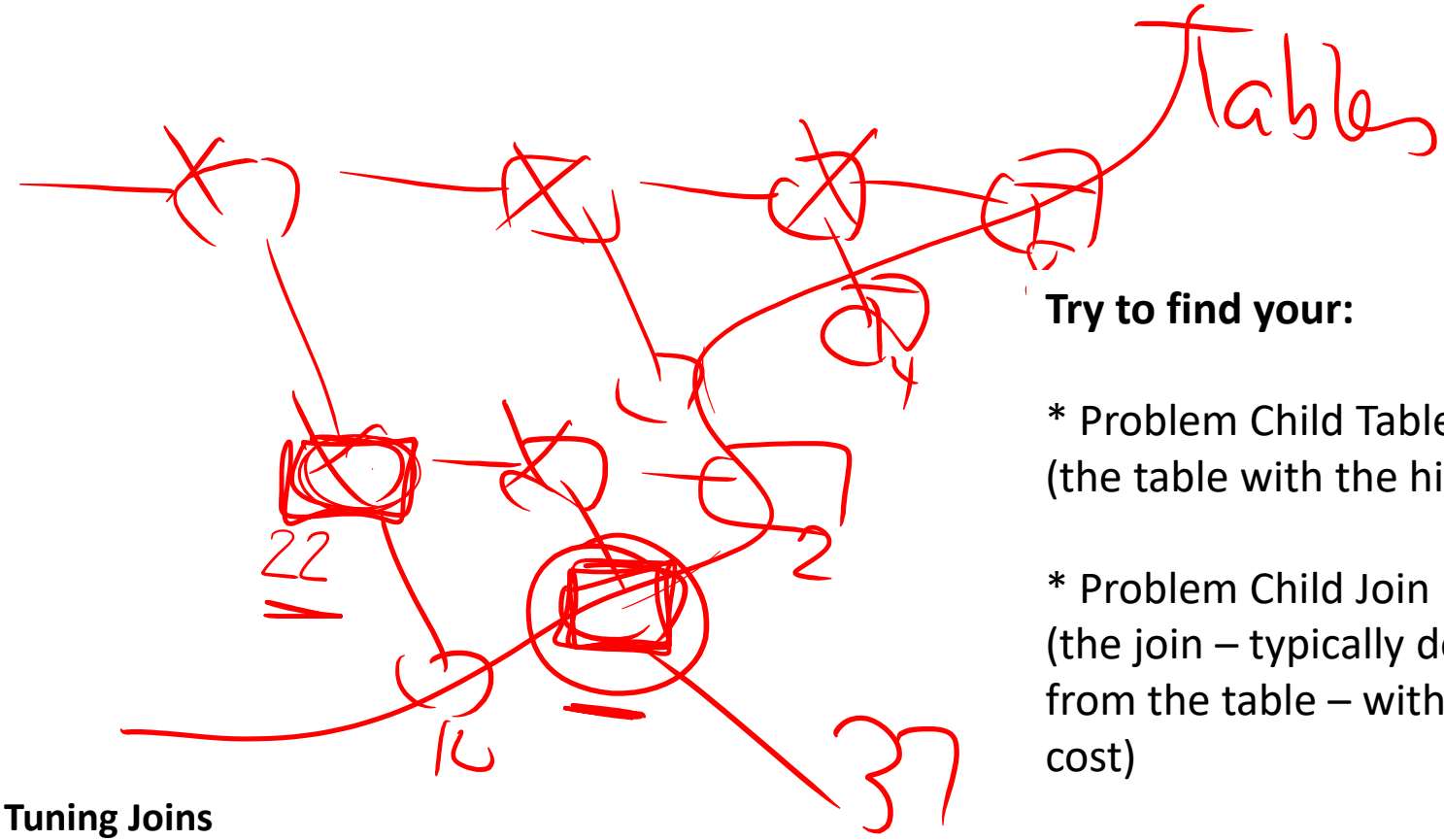
Hash Joins

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.



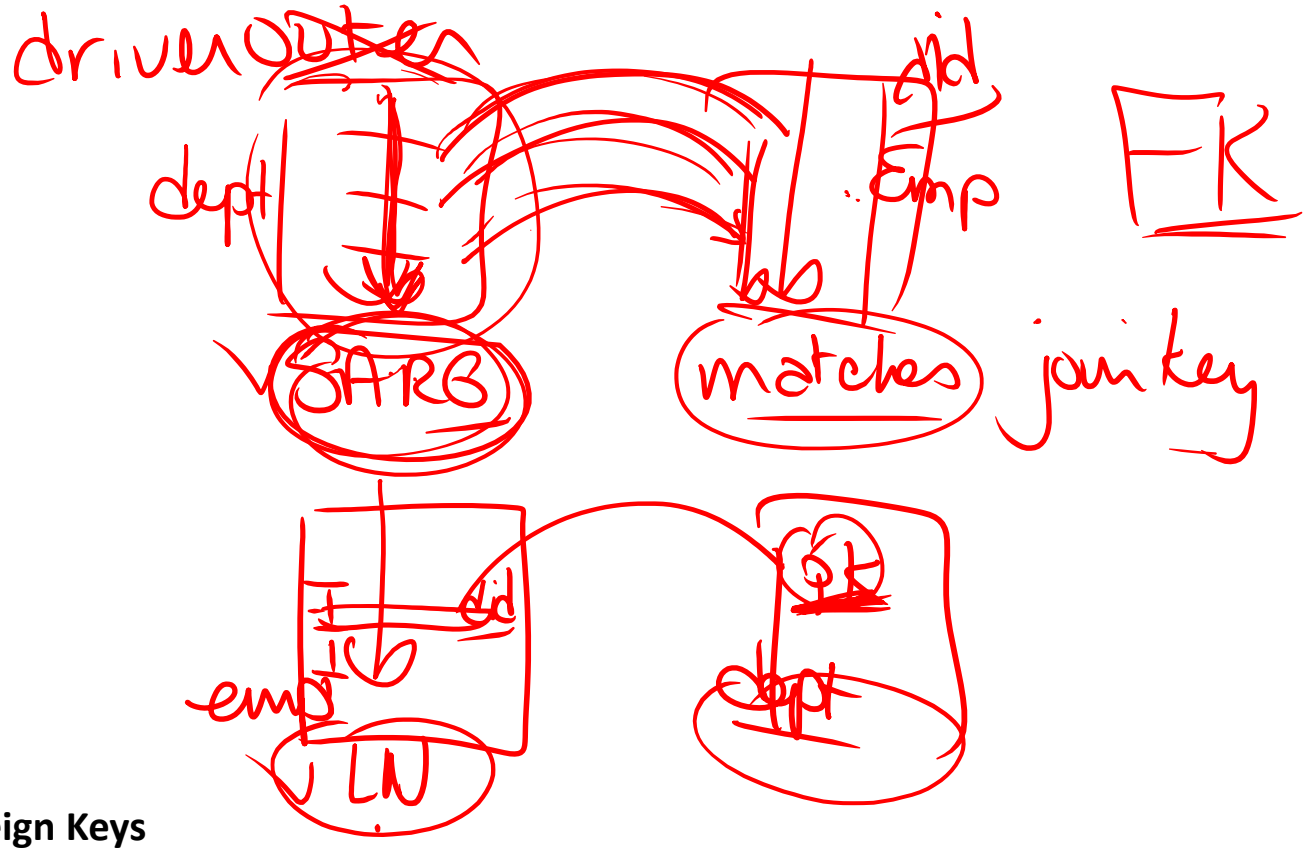
Try to find your:

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

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

Tuning Joins

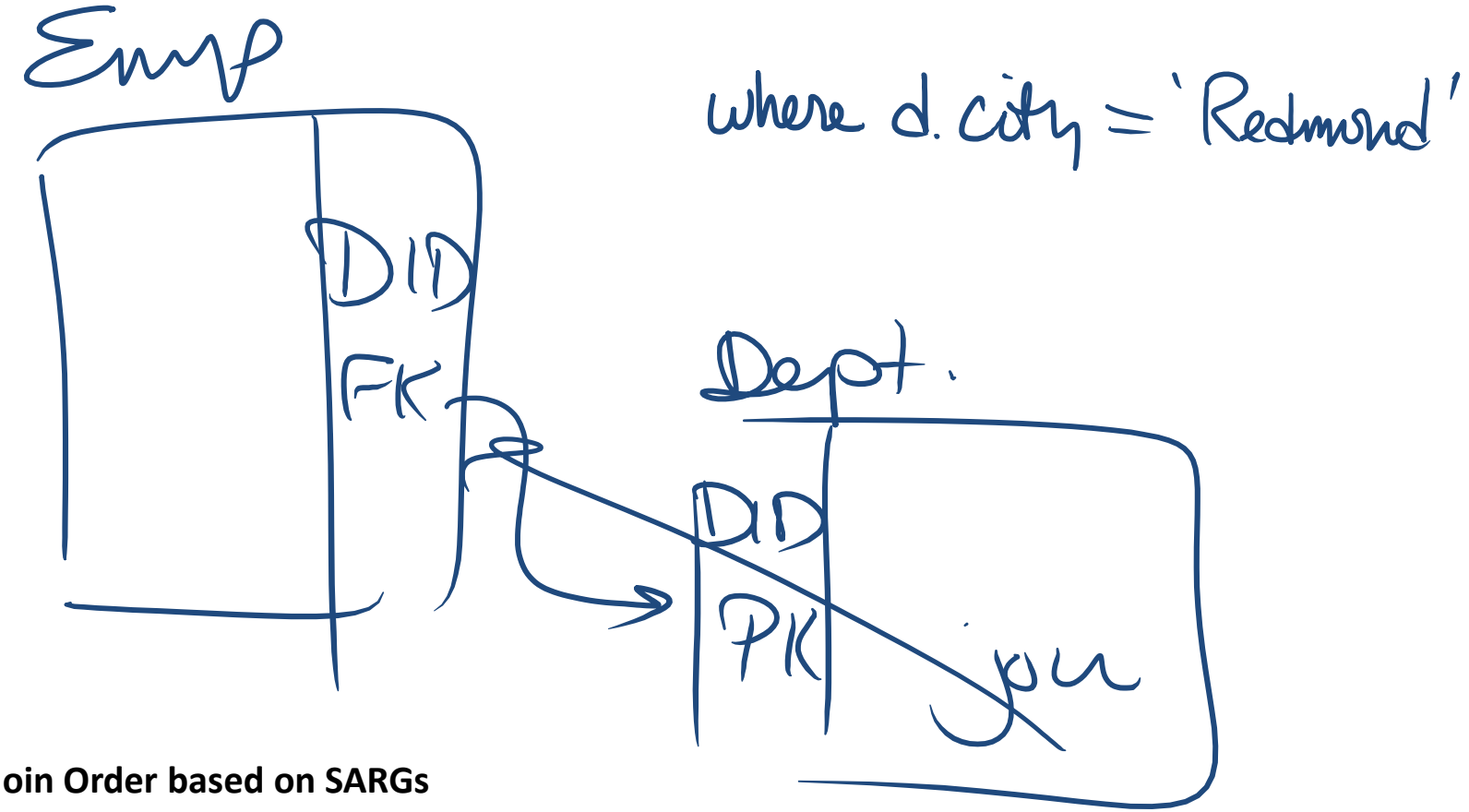
Tuning a large complex join takes breaking it down into smaller chunks. The things that you consider are the costs of the tables (the outer most events) and the costs of the joins. And, typically, the most expensive join is downstream from the most expensive table.



Indexing Foreign Keys

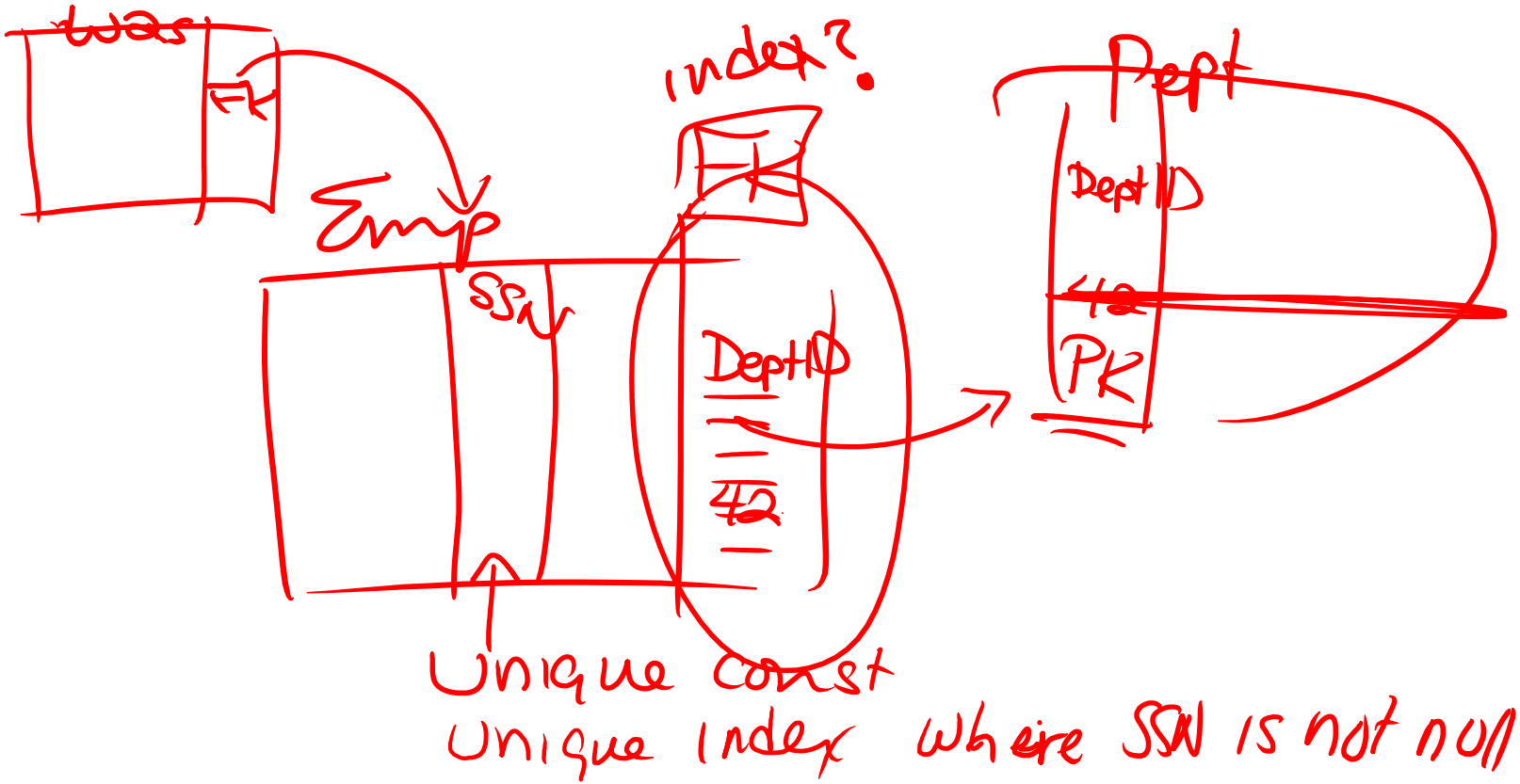
An index on a join column (the foreign key) helps the performance for the referential integrity as well as *some* joins. A narrow index on the join column helps but is often superseded by wider indexes; it's still a good start. (hence the phase – phase **1**)

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



Changing Join Order based on SARGs

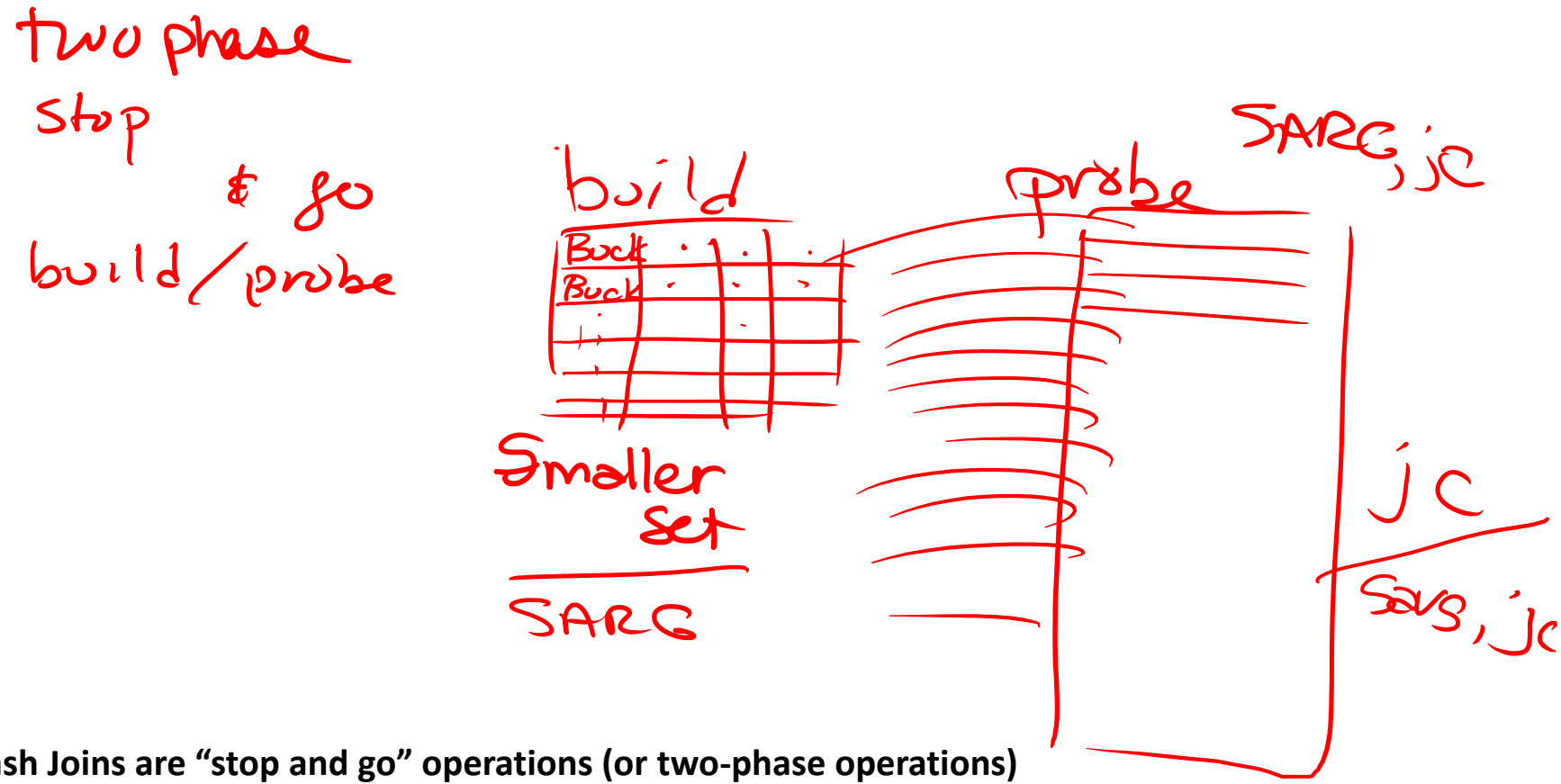
An index on a join column (the foreign key) helps the performance for the referential integrity as well as some joins because your more selective criteria might be on the referencing table rather than the referenced.



A Foreign Key can reference

- Primary Key
- Unique Key
- Any unique index (even with INCLUDE but **not with filters [bummer!]**)

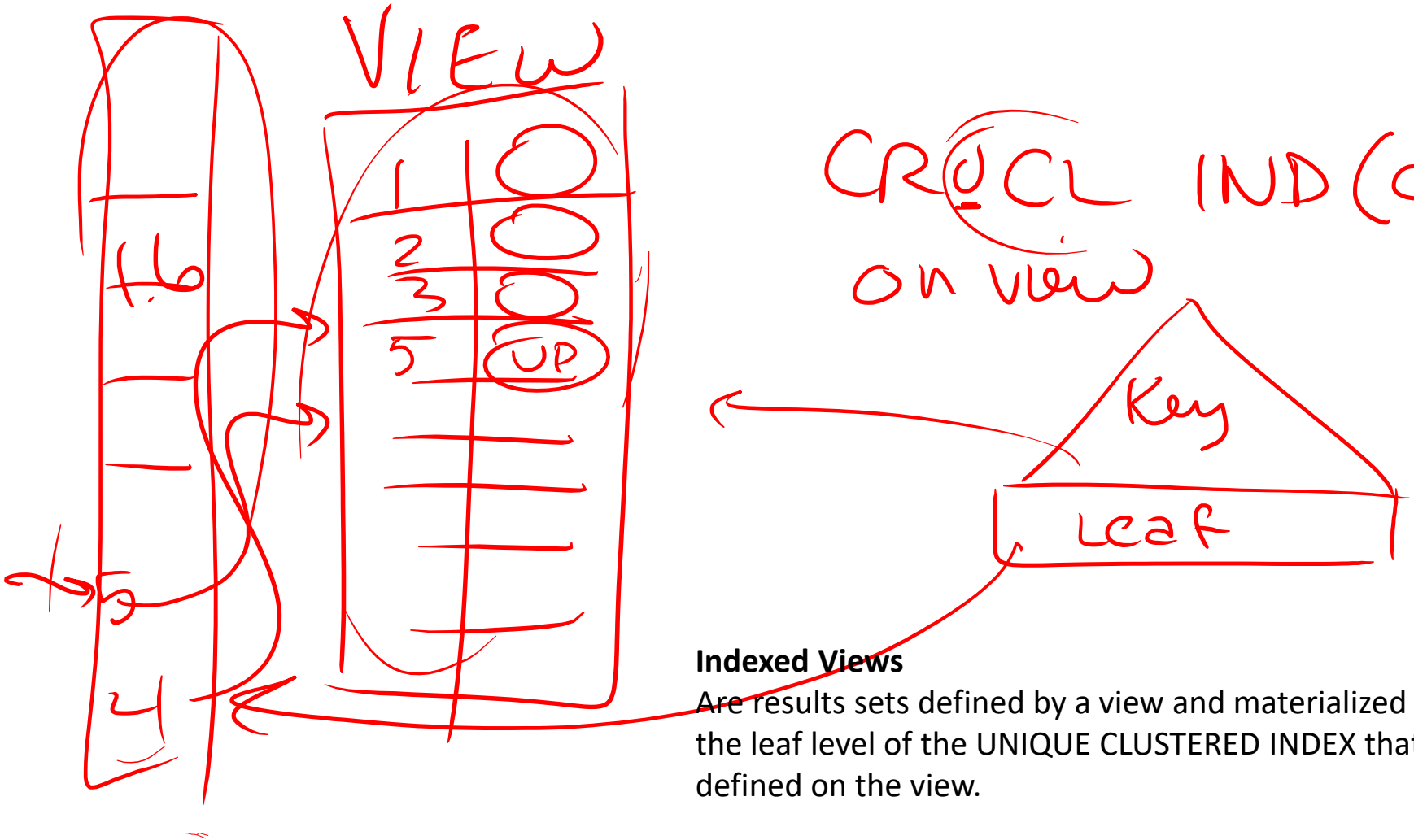
Blog Post: http://www.sqlskills.com/blogs/kimberly/foreign-keys-can-reference-unique-indexes-without-constraints/



Hash Joins are "stop and go" operations (or two-phase operations)

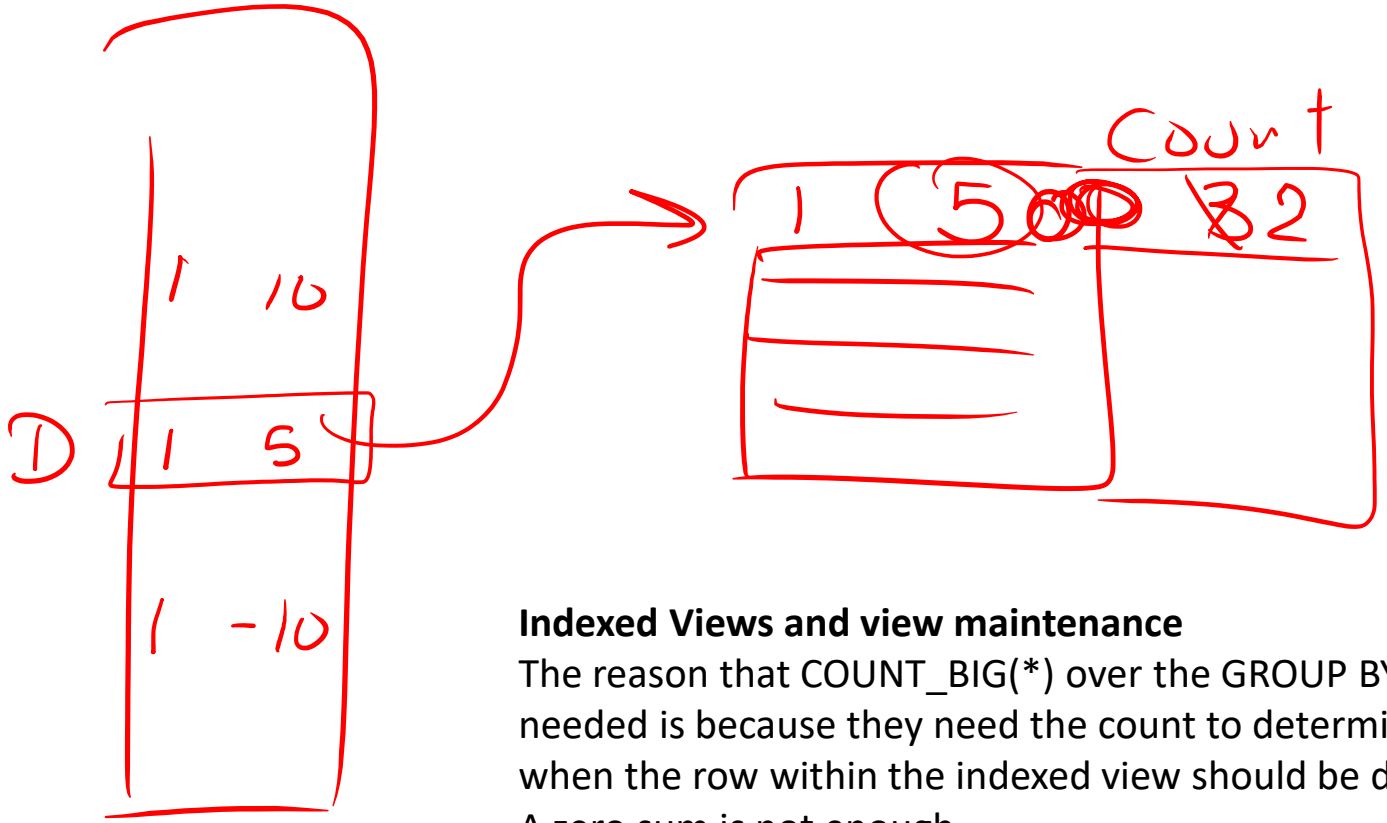
Phase I – Build (this is the phase where they build a table to fit in cache)

Phase II – Probe (this is where the larger set uses the join condition to "probe" into the build/temp table to output matches). Probe cannot "go" until after build completes (hence the term – stop and go).



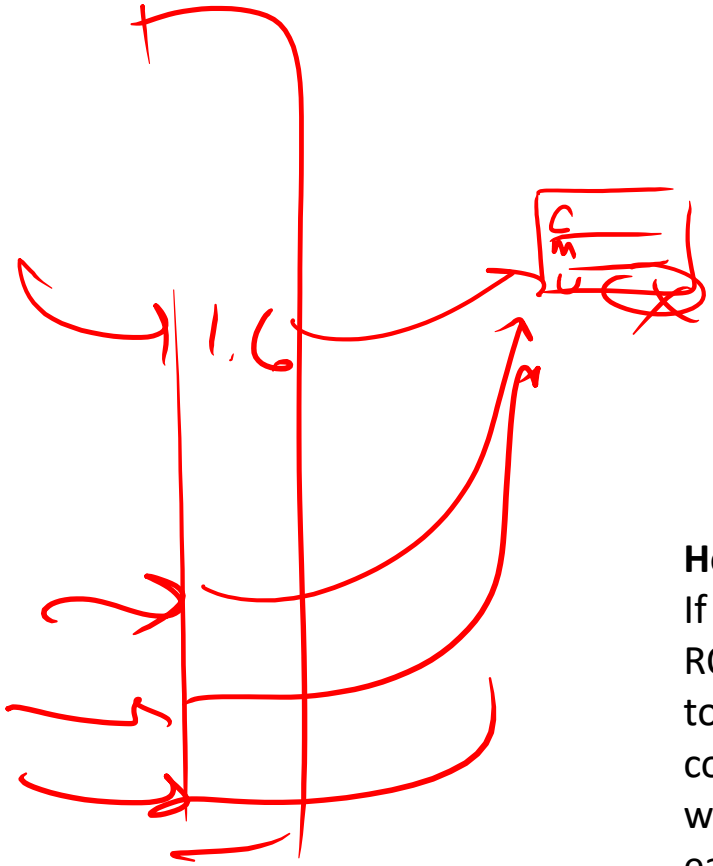
Indexed Views

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



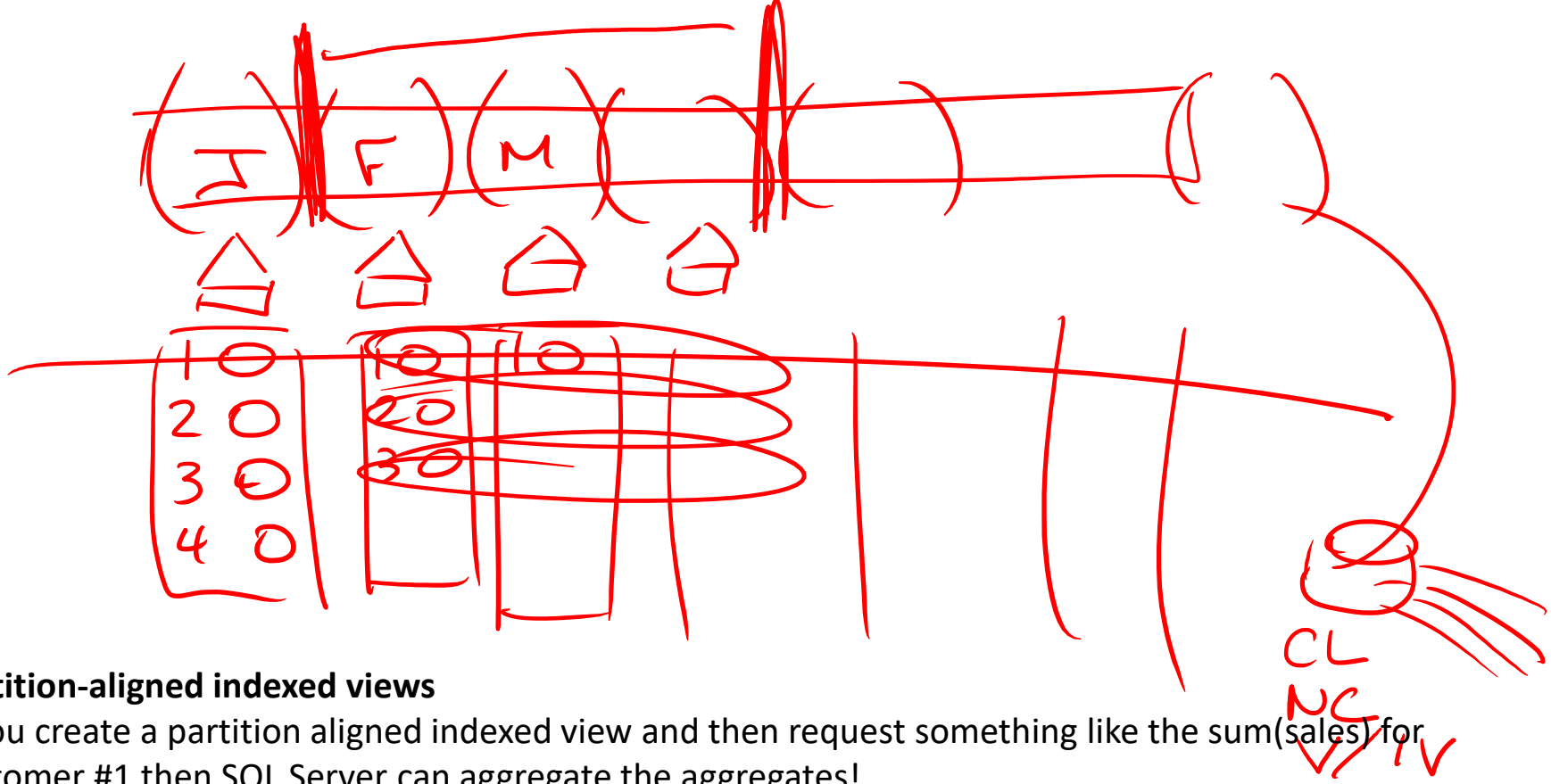
Indexed Views and view maintenance

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.



Hot Row

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 new sale has to update the sum for that country. This will become a terrible bottleneck.



Partition-aligned indexed views

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 2005, indexed views had to be dropped before fast-switching.

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

SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 11: Cardinality Estimation Issues

Kimberly L. Tripp

Kimberly@SQLskills.com

1/3 RED

1/4 ROUND

1/4 min val estimations

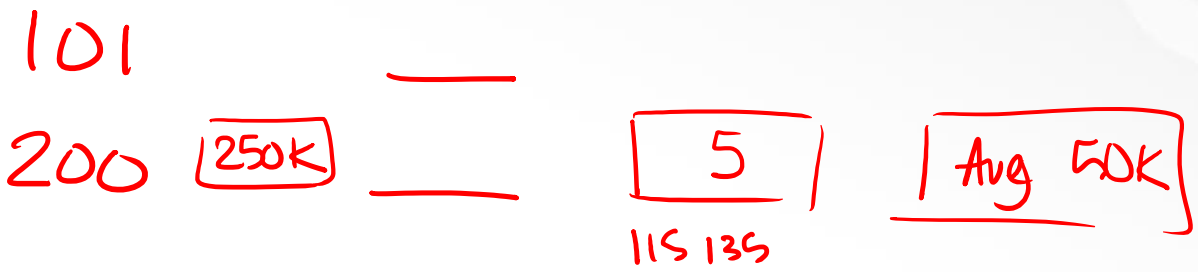
How many are round + red

Legacy CE

$1/4 \times 1/3 = 1/12$

New CE

$1/4 \times 1/3^2 = 1/36$

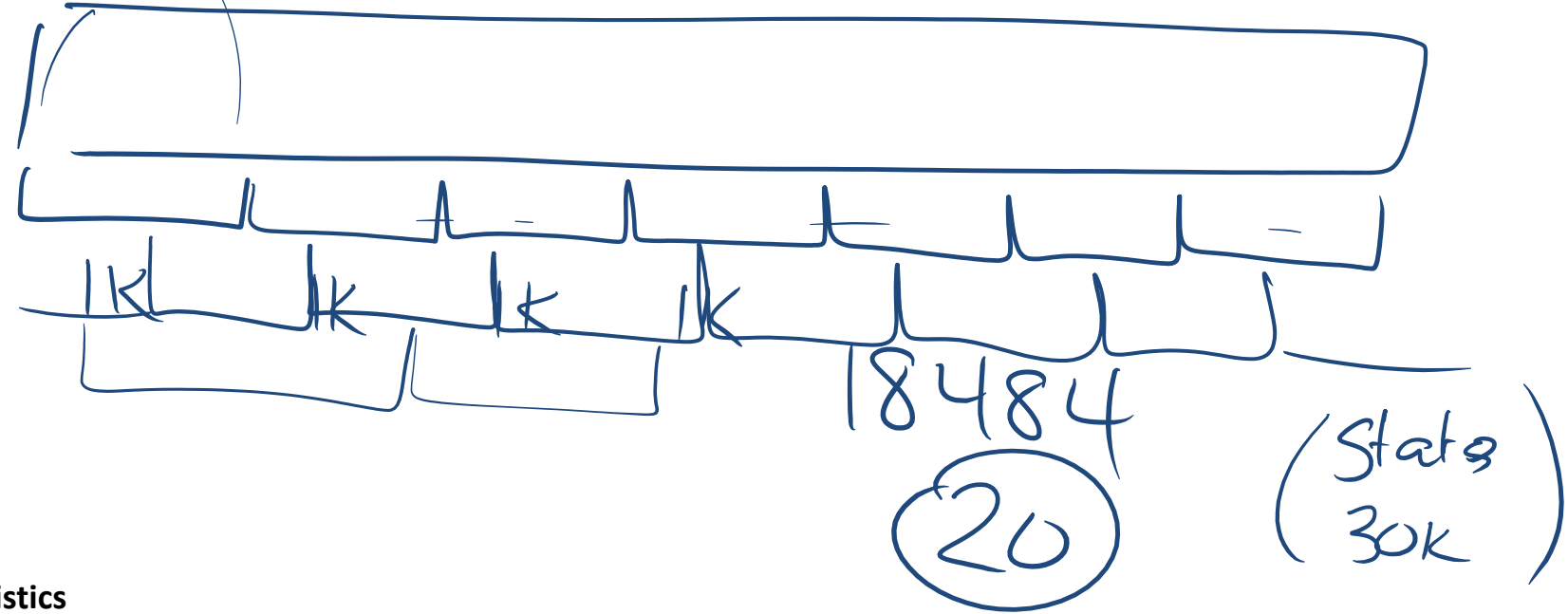


This was around our discussion about the limitation of SQL Server's knowledge of the data with gaps in the step values.

If the step describes the range from 101 to 200 (not including those values)

- There are 250K rows over that range
- There are 5 distinct values

The average will show 50K and EVERY value supplied between 101 and 200 will get an estimate of 50K. There's no way for SQL Server to know WHICH values actually exist.

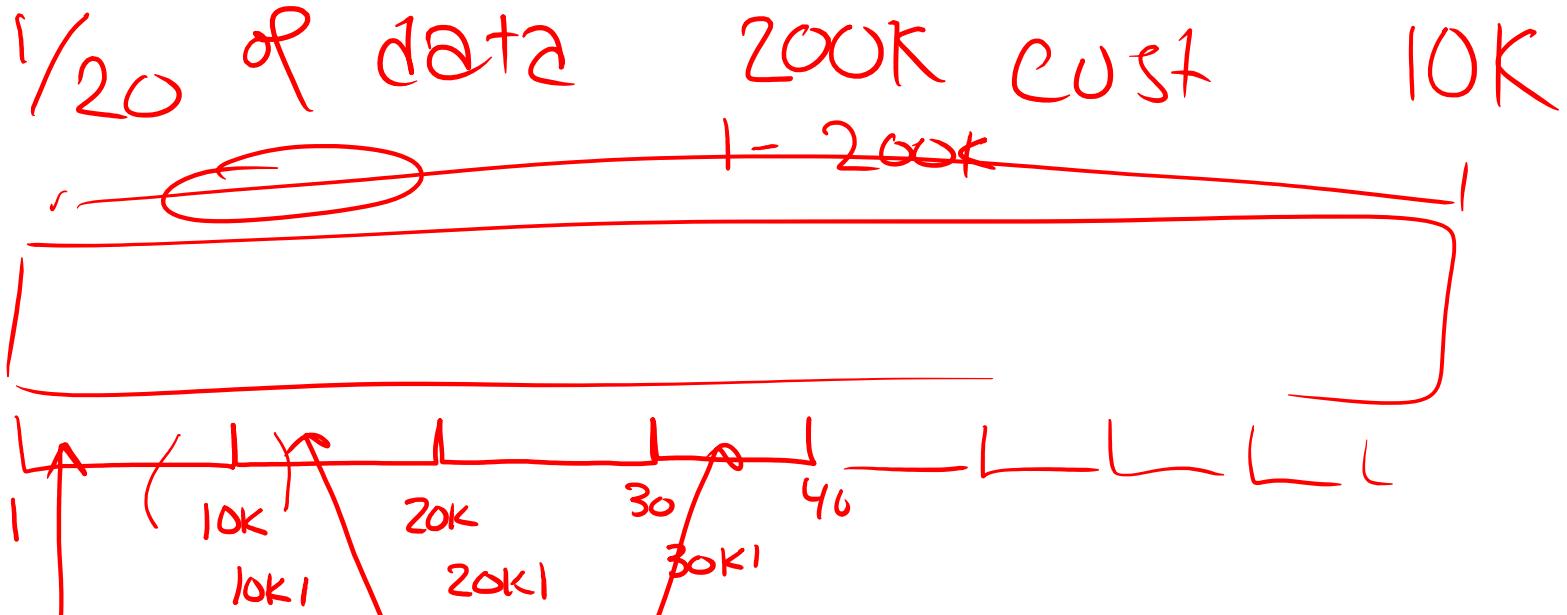


Filtered Statistics

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 adding only 10 filtered stats 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 just average out. So, really, this issue is to significantly help queries that are more targeted (where the stats just weren't good).

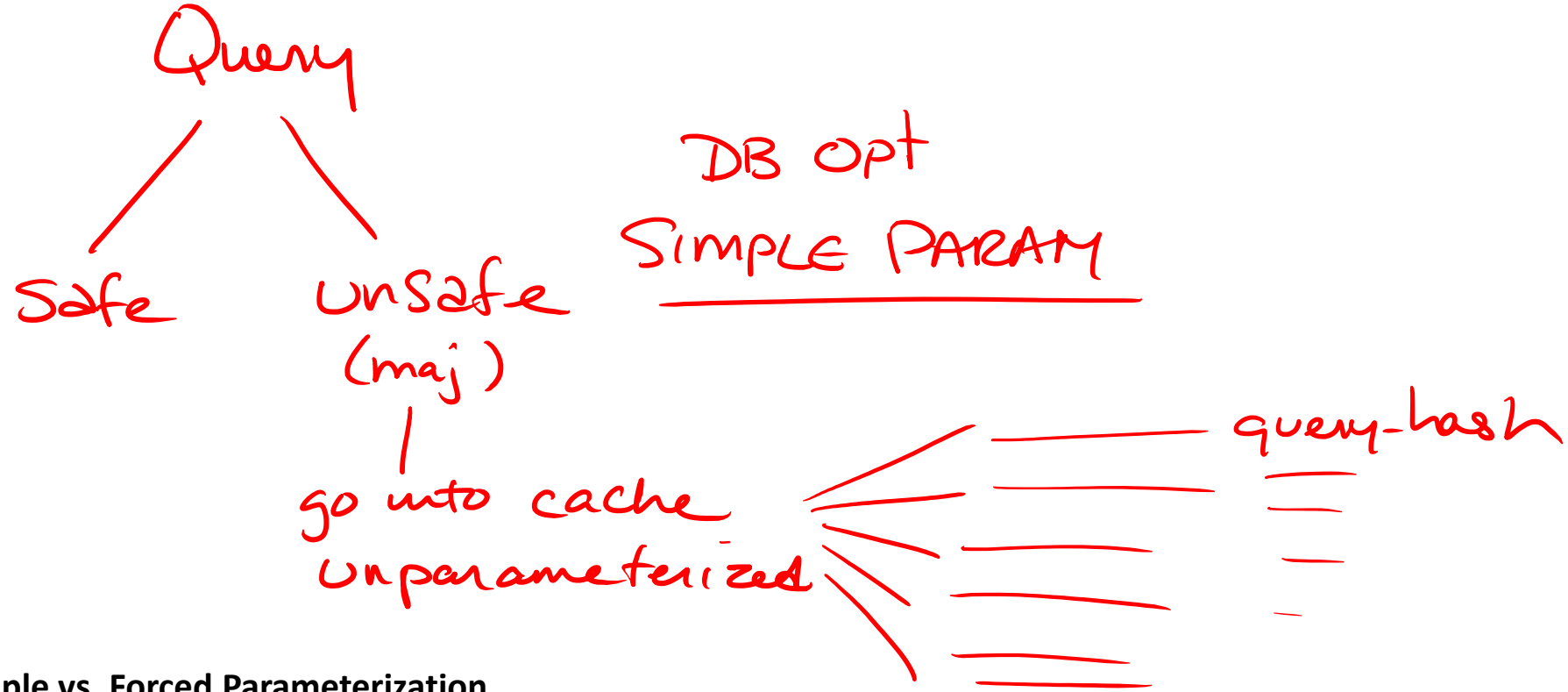
* The optimizer can detect whether interval conditions in a filtered index cover, or "subsume" interval conditions of a query.



Creating Filtered Stats

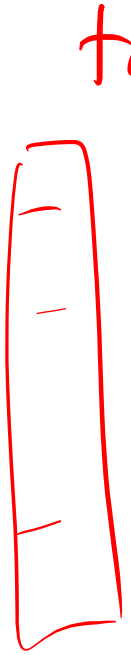
The idea is to just have better statistics than what you have currently. To do this you can divide the range into 10-20 buckets. This will give you 10 times (or 20 times) better information in terms of statistics.

You WILL need to regularly/automatically review the values/ranges and add more or divide them up again to ensure that the statistics are good (and stay good)!



Simple vs. Forced Parameterization

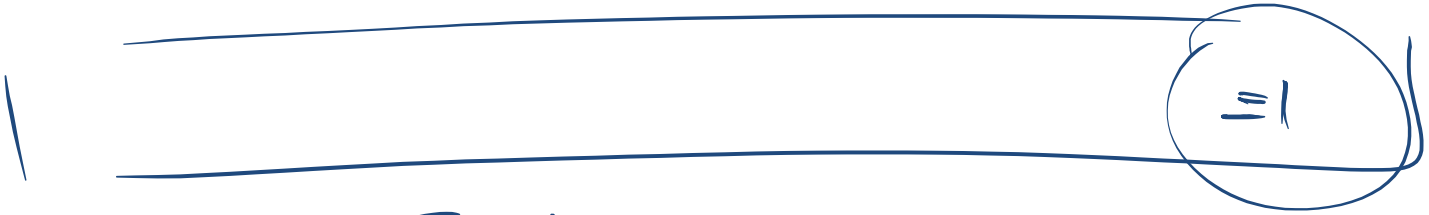
The general process is that SQL Server analyzes a query to determine if it's safe or not (the majority of them will **NOT** be safe). If it's safe then it can get parameterized and reused. If it's unsafe then it goes into memory as an individual query (and it's harder to determine the cumulative effect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



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.

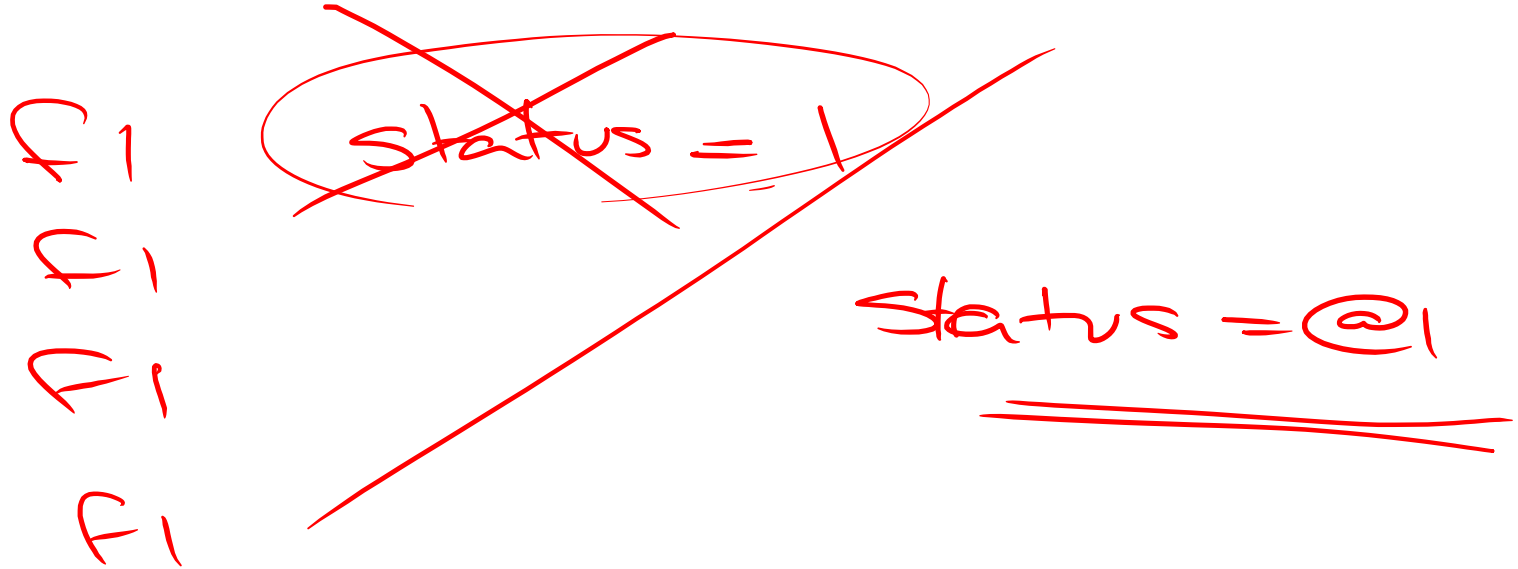


Filtered indexes and filtered stats – Auto update

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



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.

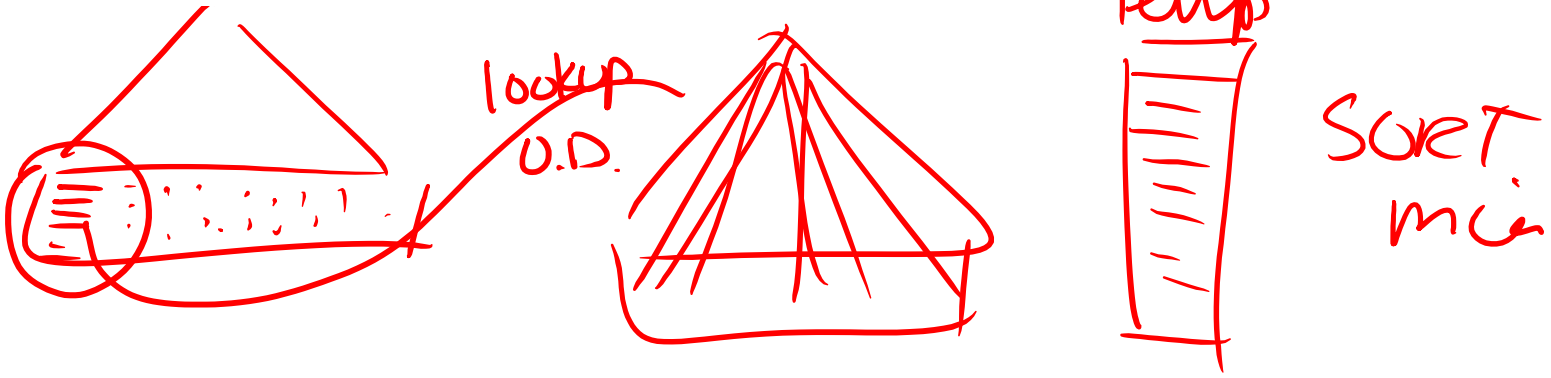
End result. You will NOT have good results with filtered indexes IF you use FORCED parameterization.
The good news: This is NOT on by default and not likely to be on in most environments.

TS ?

FactInternetSales2

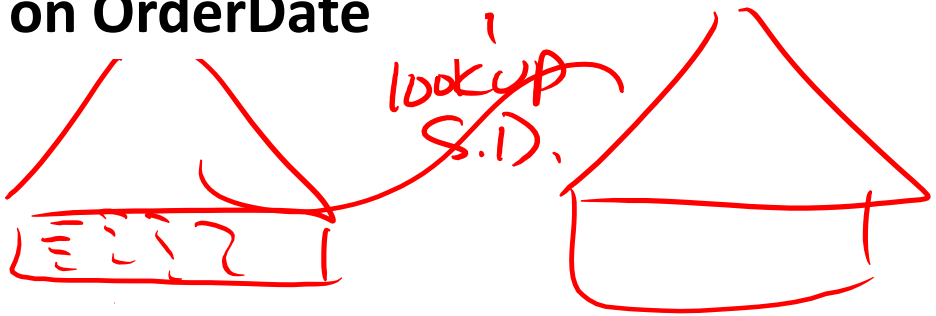
NC

Index on ShippedDate



NC

Index on OrderDate



when will I hit a NULL

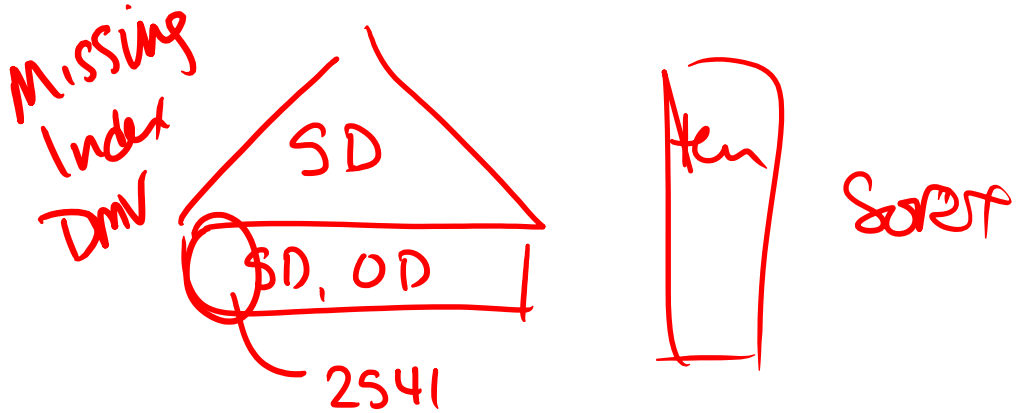
$\frac{2541}{60398} = \frac{1}{23.7}$

DEMO

This was the demo on uneven distribution.

A table scan is always an option.

With narrow indexes, SQL Server does not understand the correlation between these columns. As a result, their estimates are going to assume even distribution.

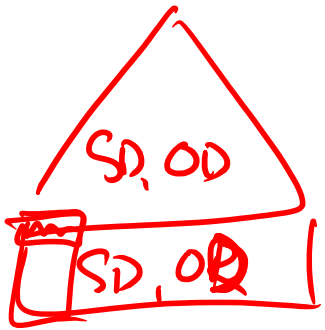


DEMO

This was the suggested index from the green hint in showplan and while it does make this query faster (and with fewer I/Os) it's not the best index that's possible.

Key point, the missing index DMVs (which is where the green hint gets its information from) – gives you good suggestions but not always the best suggestion.

Furthermore, this is the same index that's recommended by DTA. So… sometimes it does take *manually* defining/choosing the index.



DEMO

This was the index that I suggested (putting OrderDateKey in the key) as a combination of:

ShipDateKey, OrderDateKey

The first record on the first page will be the minimum OrderDateKey where ShipDateKey IS NULL.

Poor Man's DW

OLTP → B/R

1. Update stats
2. Rebuild Indexes
   FF = 100
   ~~Update Stats~~
   Sampling?
3. sp_createstats
   indexonly
   fullscan – ?

?

READ ONLY

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.

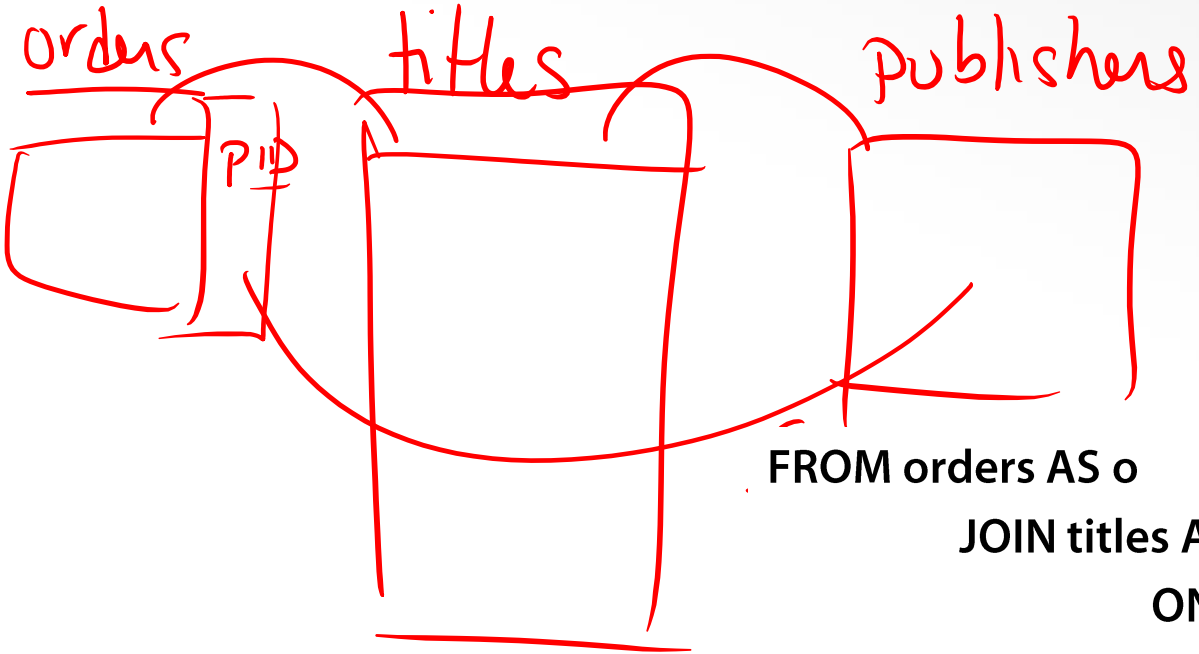
SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Discussion: Table Design Strategies

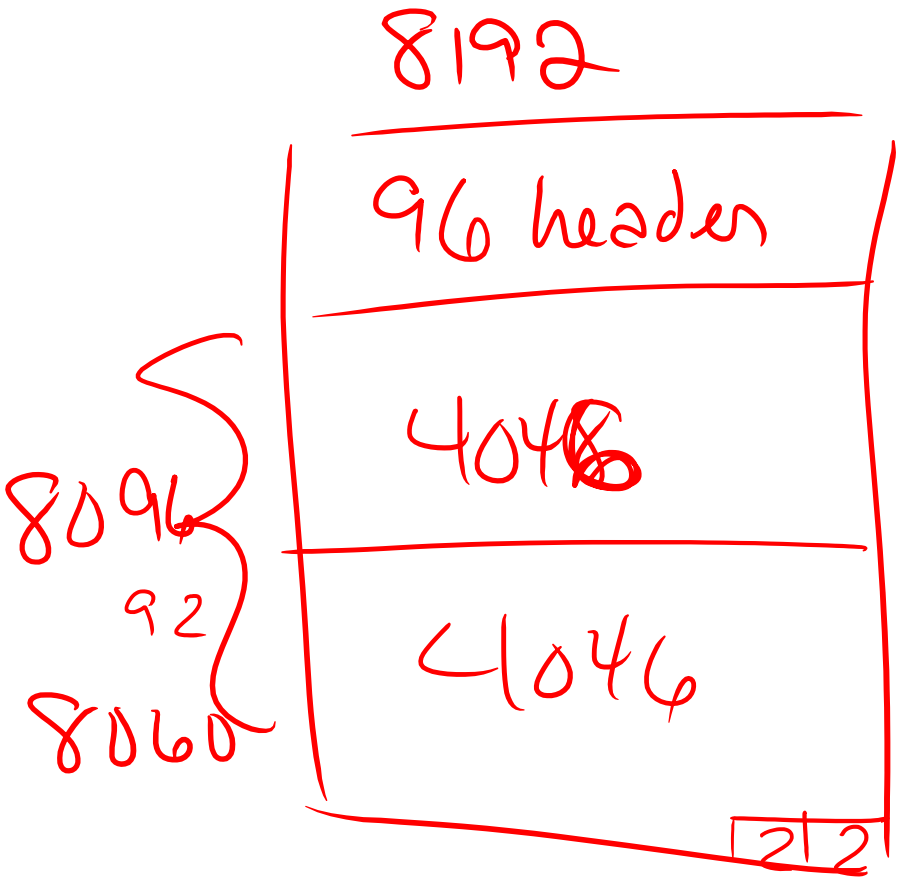
Kimberly L. Tripp

Kimberly@SQLskills.com




```
FROM orders AS o
        JOIN titles AS t
                ON o.tid = t.tid
        JOIN publishers AS p
                ON t.pid = p.pid
                AND o.pid = p.pid
```

Considering a schema change to allow joins to be removed from common queries can make a huge difference in join performance. If you do add that column also be sure to always state it in joins. You'll give the optimizer more chances for tuning the join.



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)

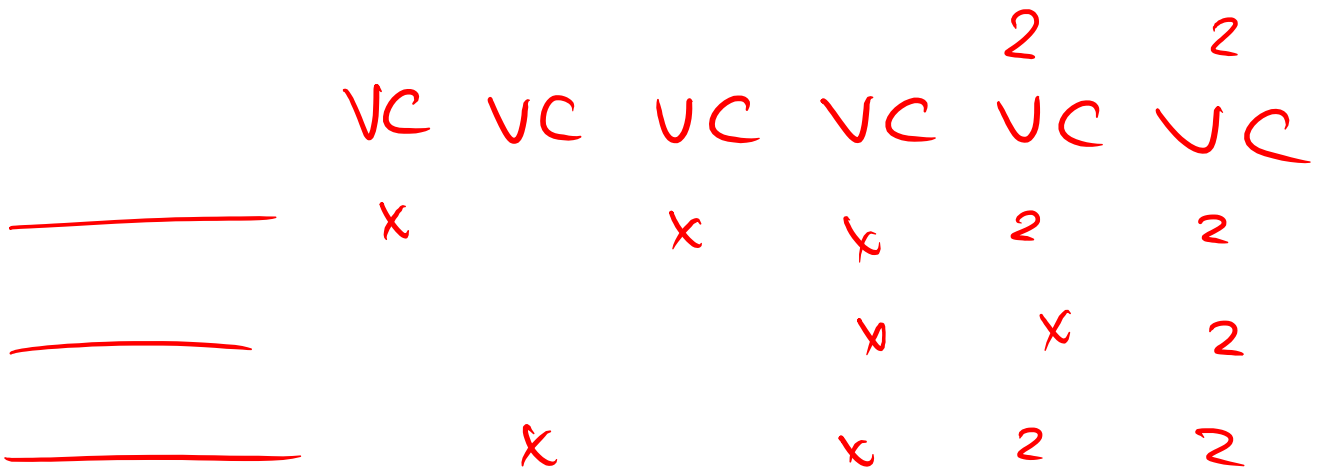
- 96 bytes in the header

= 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.

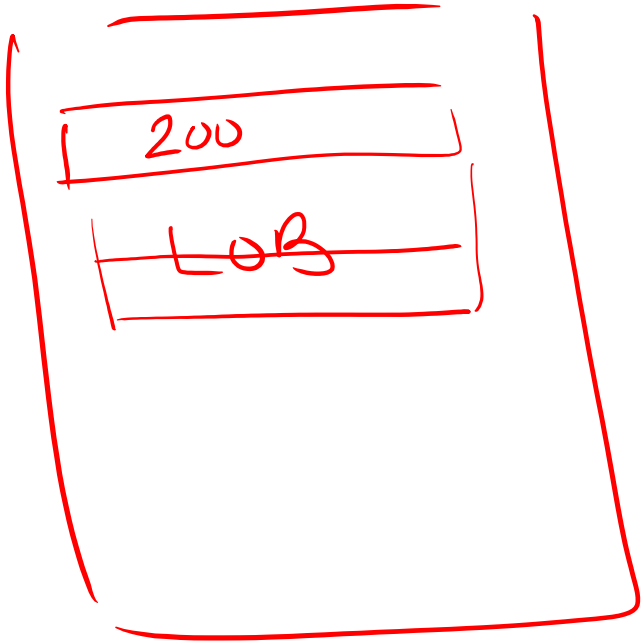
Some of which they're already using in SQL Server 2005 and higher. For row versioning, each versioned row requires a 14 byte offset. If they had allowed a row to be 8096 – where would they have put the offset?



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

MB

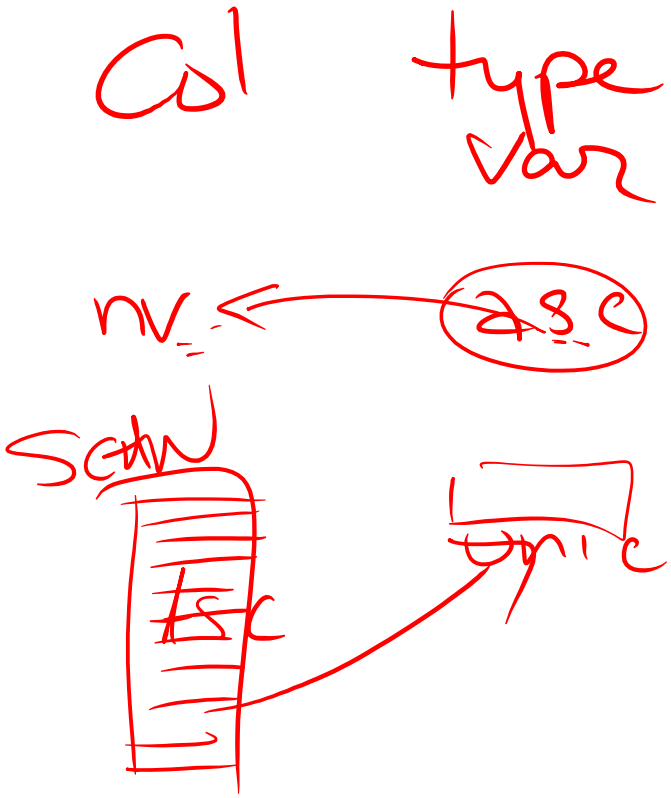
majority

< 2000 bytes

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!



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

client → ID

P/C

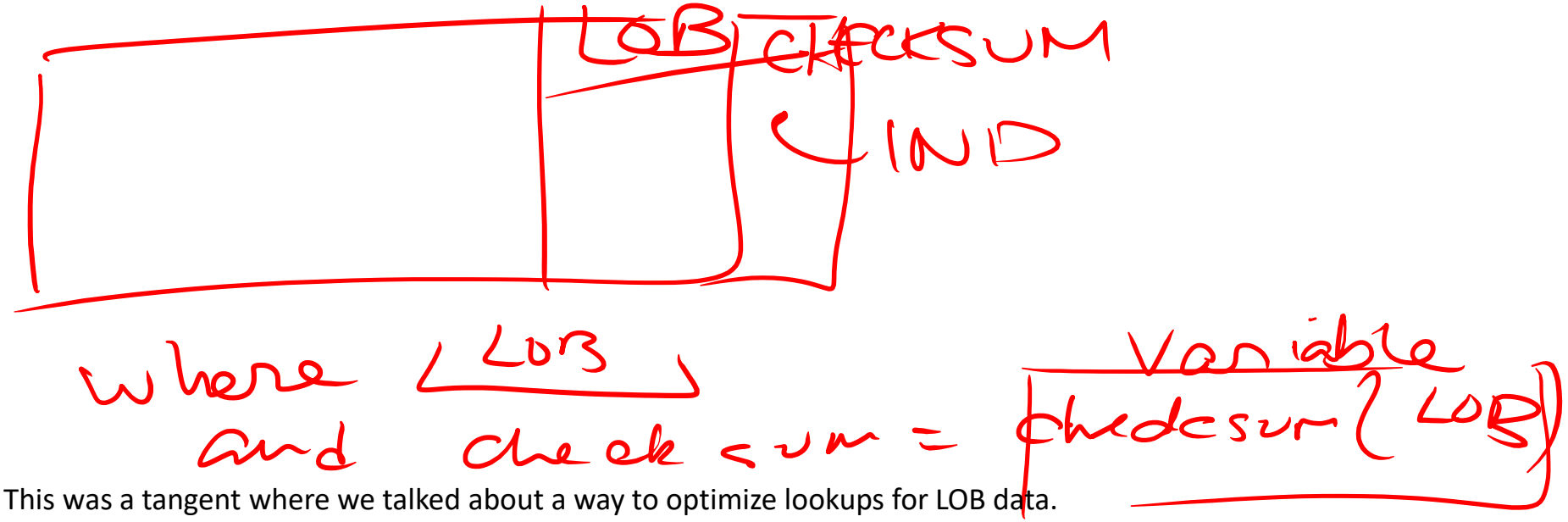
C

C

C

def. def

VC VC VC VC

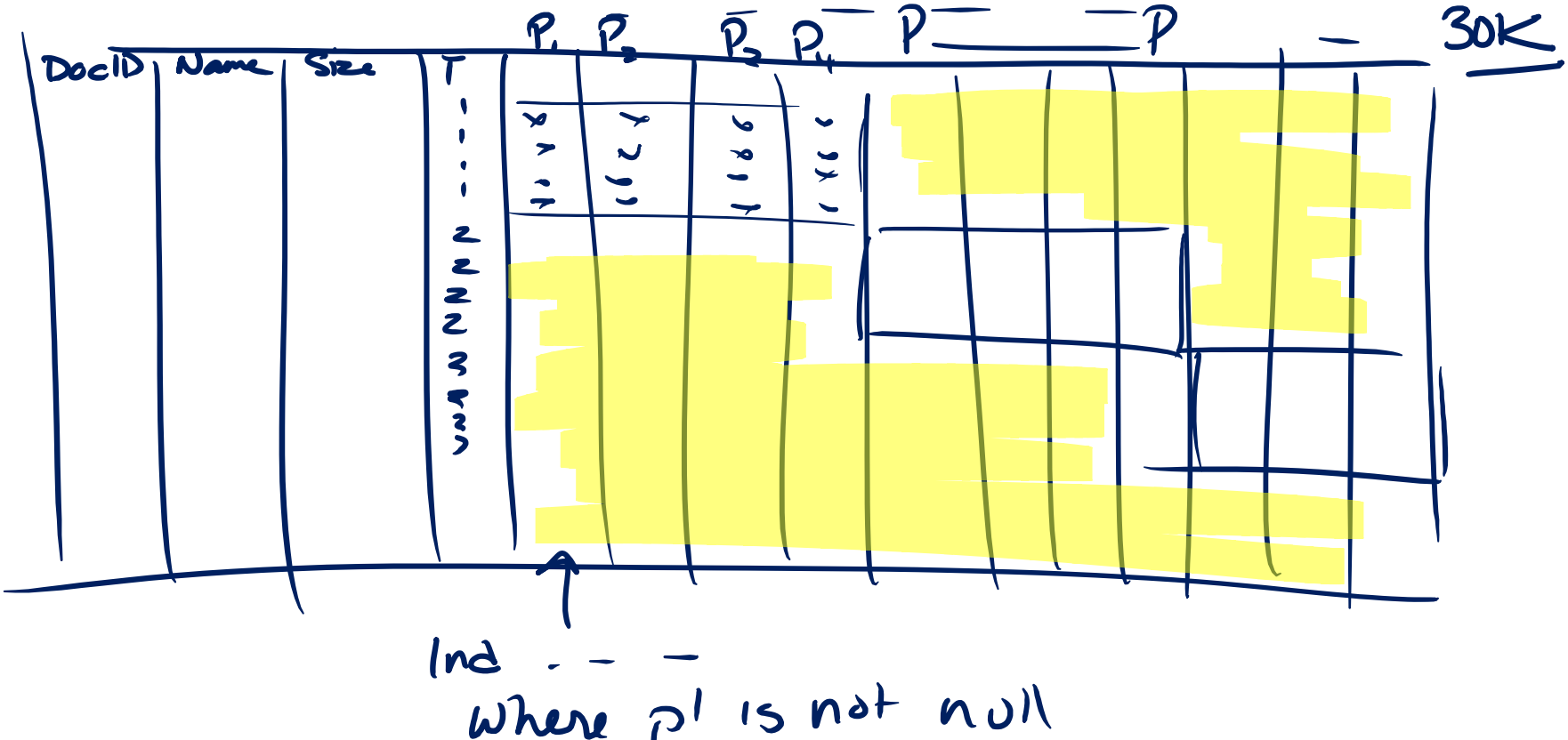


This was a tangent where we talked about a way to optimize lookups for LOB data.
One of the things that always causes people grief is that columns over 900 bytes cannot be indexed. So, if you have a WHERE clause like this:

WHERE lobcolumn = 'lob value……'

SQL Server has to do a table scan to get all of this data.
Another trick is:
Add a CHECKSUM column that checksums a particularly large column (note: legacy LOB column types: (n)text, image AND XML are not supported – see this link: http://msdn.microsoft.com/en-us/library/ms189788(v=sql.105).aspx) and then rewrite your queries to be this:

WHERE lobcolumn = 'lob value……'
AND checksumcol = CHECKSUM('lob value……')



Sparse Columns – It only makes sense to use sparse columns when you have multiple columns that will be sparsely populated. How sparsely populated? That depends on the data type. Check out the BOL for the percentage (per type) that needs to be NULL in order to save 40% space.

LINK: http://technet.microsoft.com/en-us/library/cc280604.aspx (topic: Estimated Space Savings by Data Type)

SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Discussions / drawings around VLDB and "partitioning" large data sets (VLTs)

Kimberly L. Tripp

Kimberly@SQLskills.com

Partitioning: PVs vs. PTs

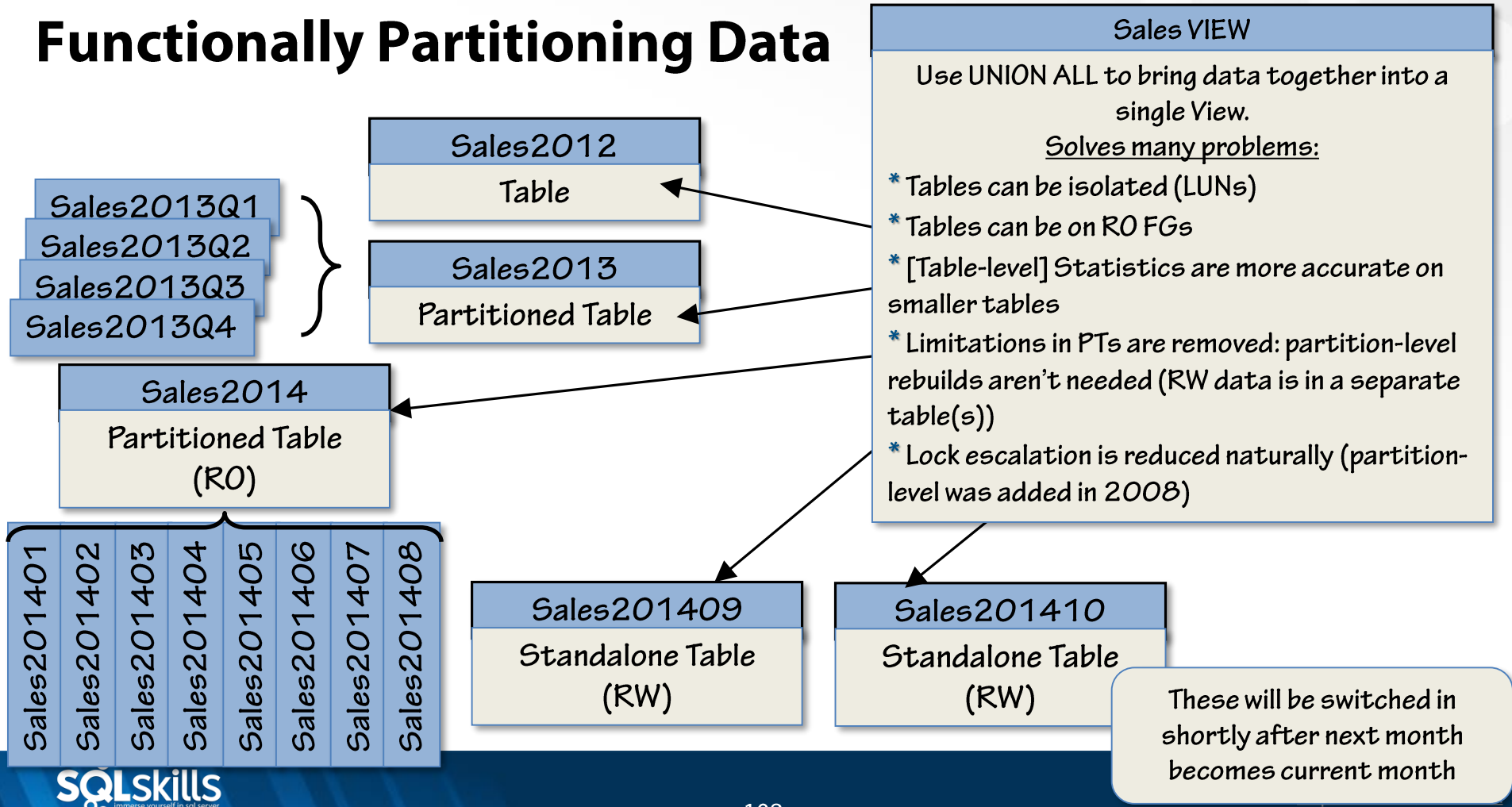
Partitioned views

- Any edition
- Lots of tables to administer
  - Must create/drop indexes on all base tables
    - Can have different indexes
    - Harder for the optimizer to optimize with so many indexes
  - Must verify business logic so that there are no gaps or overlapping values
  - Each table has [potentially] better statistics as the tables are smaller
- Can rebuild any of the tables ONLINE (if using EE)
- Can support multiple constraints on one or more columns

Partitioned tables

- Enterprise Edition only
- Only 1 table to administer
  - Only 1 table to create/drop indexes
    - All partitions have same indexes (which is easier for the optimizer to optimize)
    - Can create different indexes with filtered indexes
  - No possibility of errors (or gaps or overlapping values)
  - Table-level statistics can be less accurate for very large tables when there's a lot of skew to the data
- Partition-level rebuilds are offline but can rebuild the ENTIRE table online (not desirable)
- Can only support partitioning over a single column

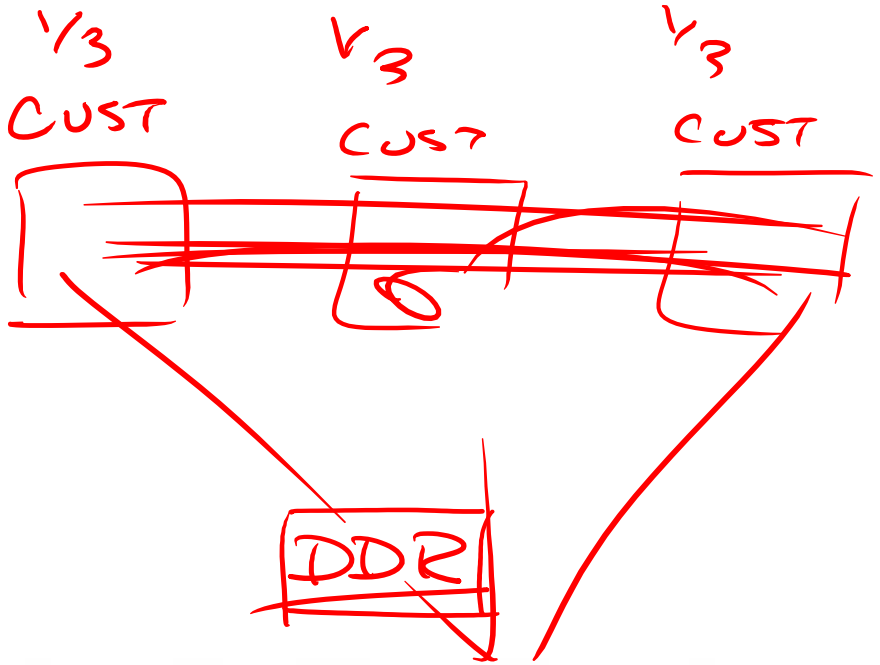
Functionally Partitioning Data



Sharding/Scale-out – also, often called Service-oriented Database Architectures (SODA)

Partitioning in our workshop was all within a single database. However, we had a side/note about sharding and scale-out design. The most important thing that I can highlight is that scaling out is most ideal through middle-tier DDR (data-driven routing) where the applications are directed to the appropriate server. If every user randomly goes to any of these instances and all requests go through [distributed partitioned] views then performance will likely be worse!

Check out the whitepapers on SODA (service-oriented database architectures)

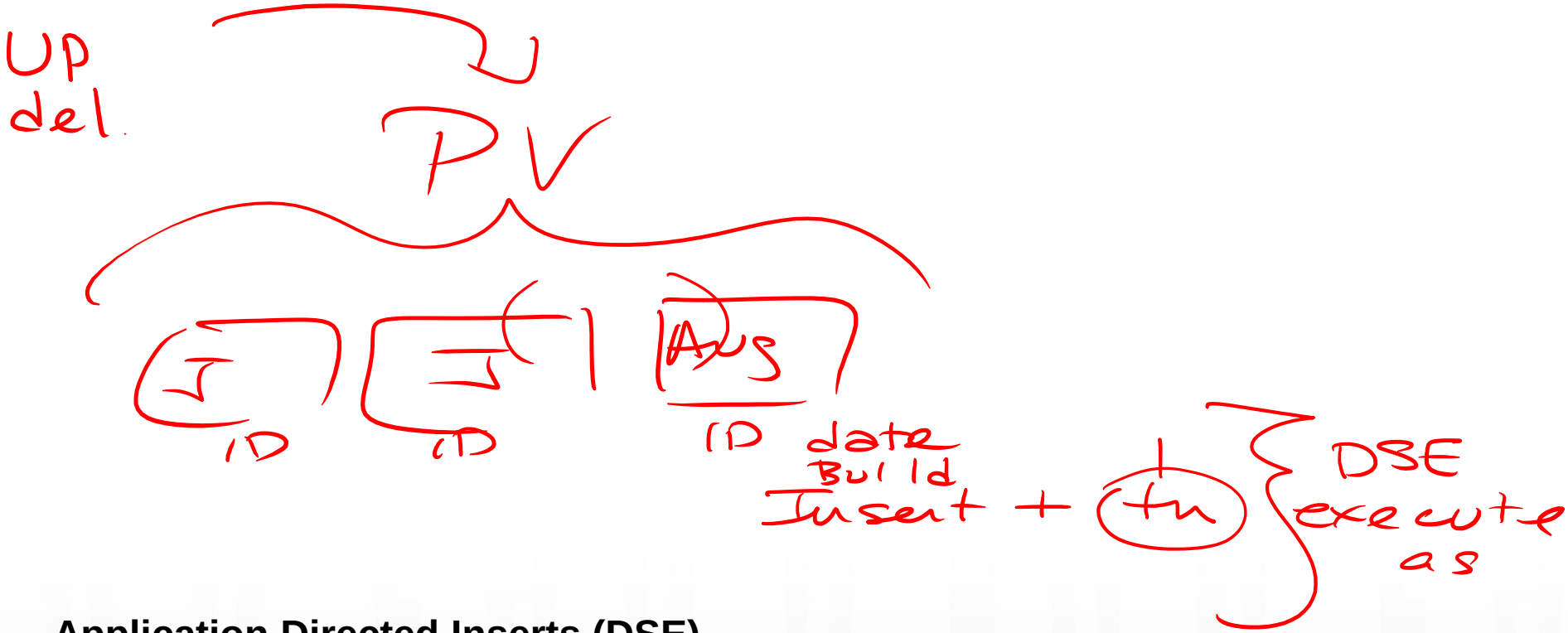


From https://www.sqlskills.com/sql-server-resources/sql-server-whitepapers/

- SODA: Service Oriented Database Architecture: App Server-Lite?
- SODA: How SQL Server 2005 Enables Service Oriented Database Architectures
- Scalability: Planning, Implementing, and Administering Scaleout Solutions with SQL Server 2005
- Scalability: Solutions for Highly Scalable Database Applications: An analysis of architectures and technologies

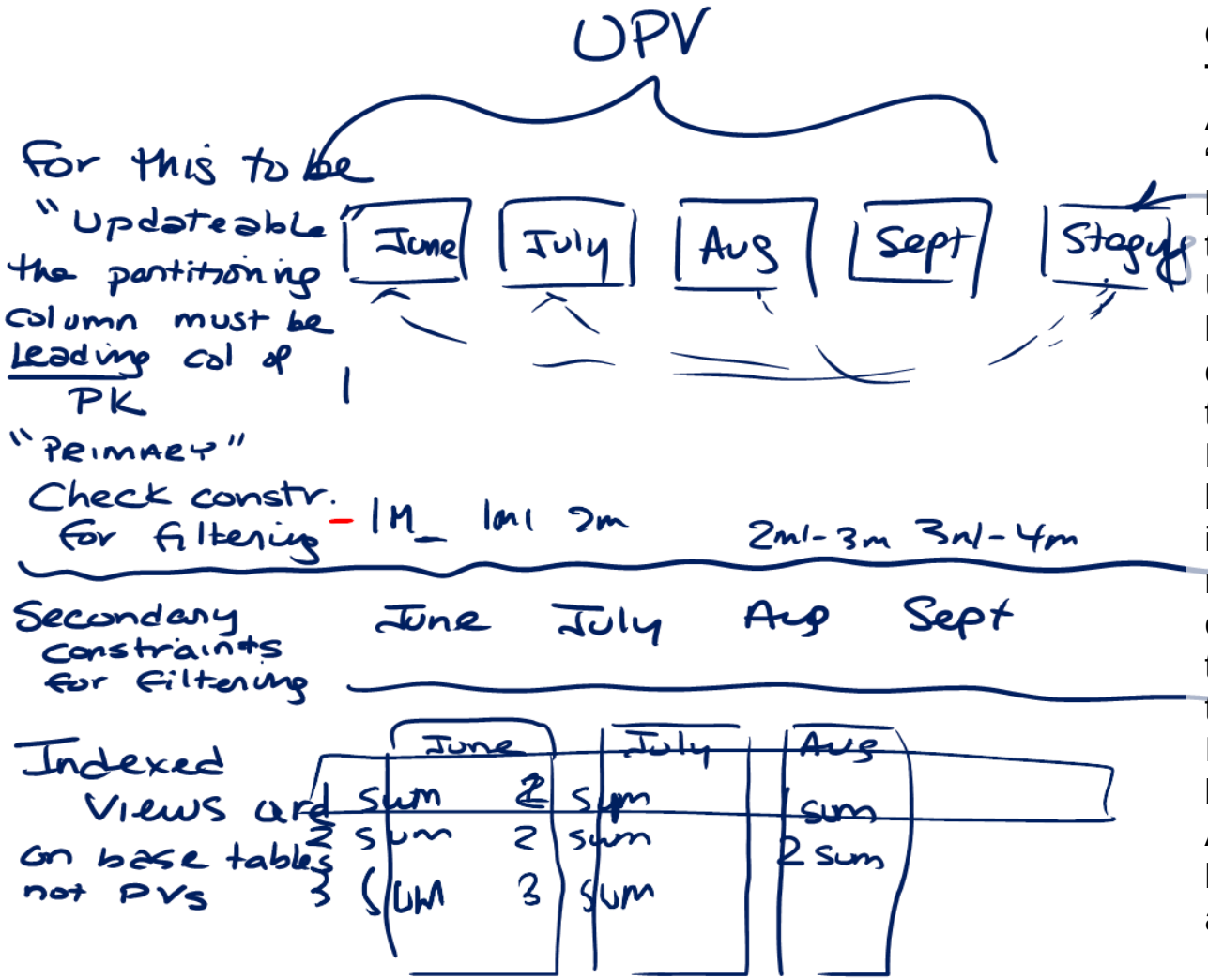
Scalable Shared Databases
Partitioning in our workshop was all within a single database. However, SQL Server (2005 + 2008 + 2008R2) does support Scalable Shared Databases. These are RO databases that are attached to multiple servers and then balanced through WLBS (Windows Load Balancing Services).

Scalable Shared Databases
https://msdn.microsoft.com/en-us/library/ms345392(v=sql.105).aspx



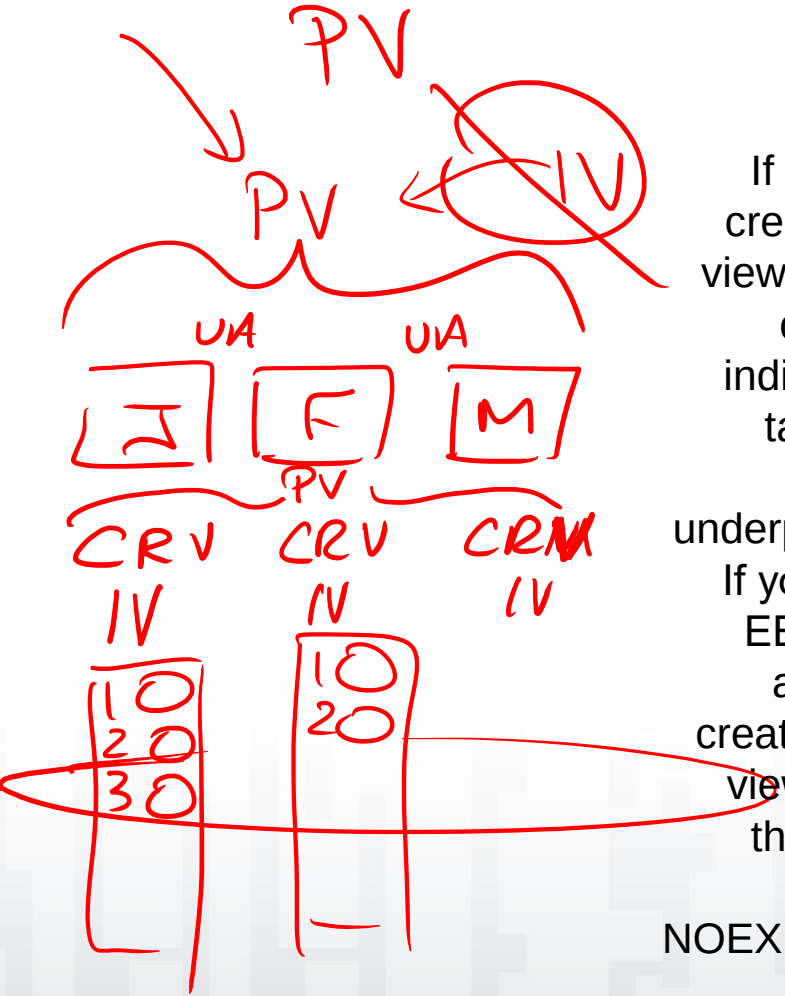
Application Directed Inserts (DSE)

Even with updateable partitioned views, I usually use application directed inserts. If you're concerned about dynamic string execution check out the Little Bobby Tables blog post: http://www.sqlskills.com/BLOGS/KIMBERLY/post/Little-Bobby-Tables-SQL-Injection-and-EXECUTE-AS.aspx



Optimizing VLTs (Very Large Table)

A very large table has many "problems"… to reduce those – don't have just ONE VLT – have smaller tables that are unioned (using UNION ALL) into a view. If you also have restrictive constraints (CHECK constraints) across all of the base tables this combination is called Partitioned Views. You get numerous benefits with this architecture including: better control / manageability, better statistics on each table, online operations, and the reduction of some operations all together (on the historical data). Indexed views are created on the base table (not the partitioned view). And, with partitioned views, you can have secondary filtering criteria (with additionally created constraints).

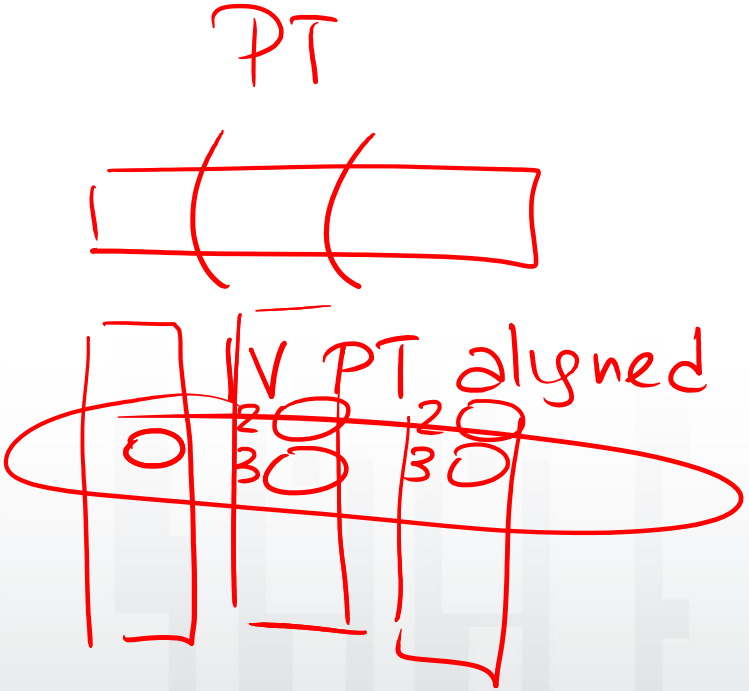


With PVs

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.

With PTs

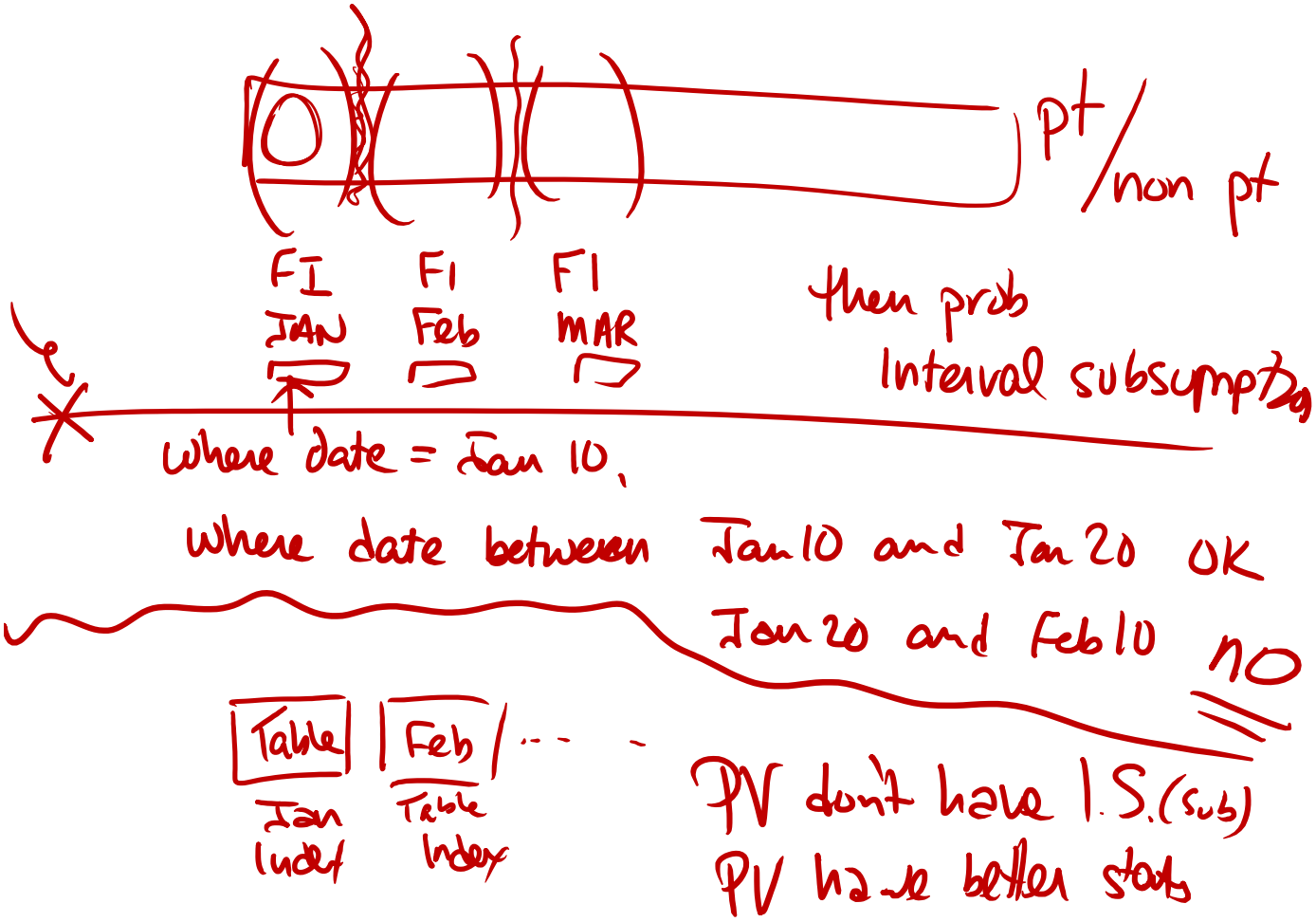
If you want to create indexed views, you must create them as partition-aligned (for fast switching). This is available in SQL Server 2008+.

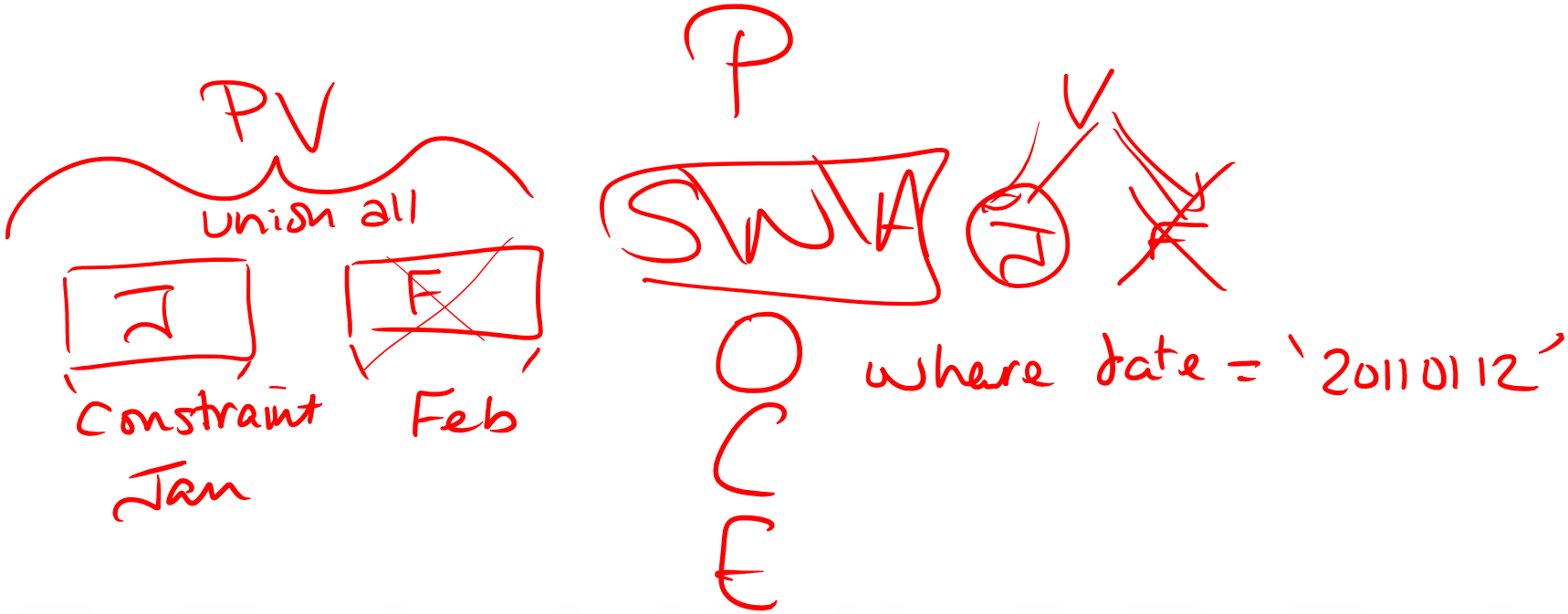


Filtering v. Partitioning

You might think that using JUST a filtered index approach would be better but then there's the interval subsumption problem.

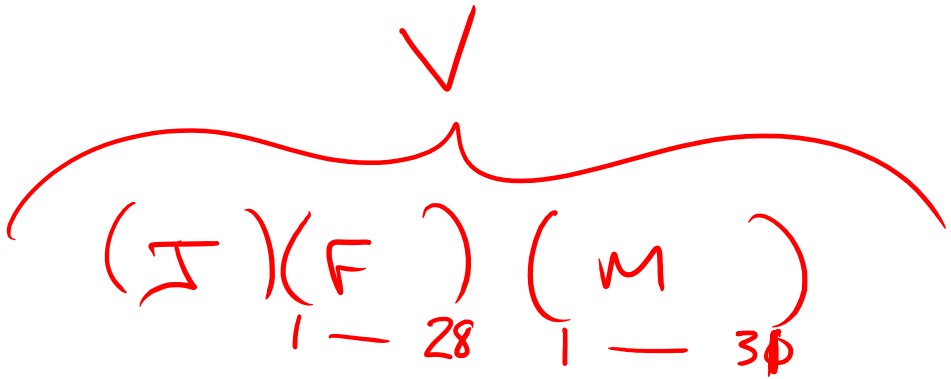
Architecting the RIGHT solution and breaking down a VLT into smaller tables can be ideal. Partitioned Views (PVs) do NOT have interval subsumption problems. And, PVs have better statistics…





Partition Elimination

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. ☺)

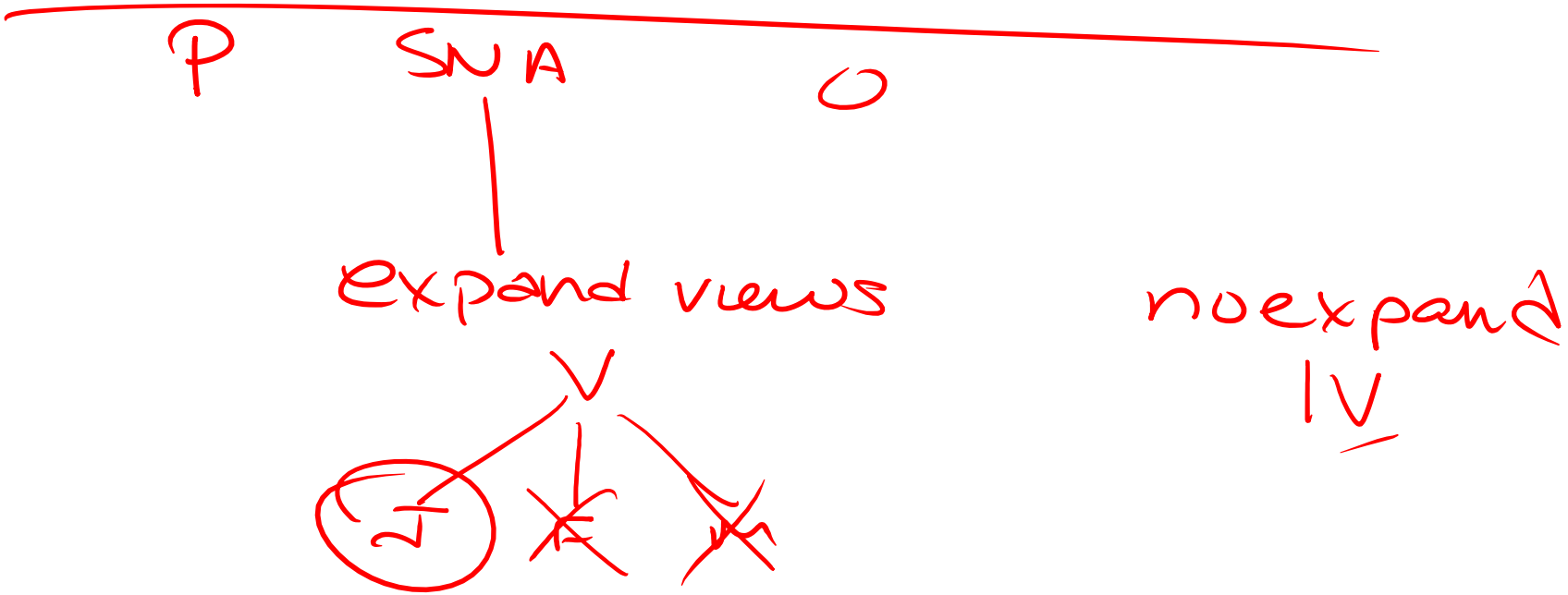


Partitioned Views and business logic

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.

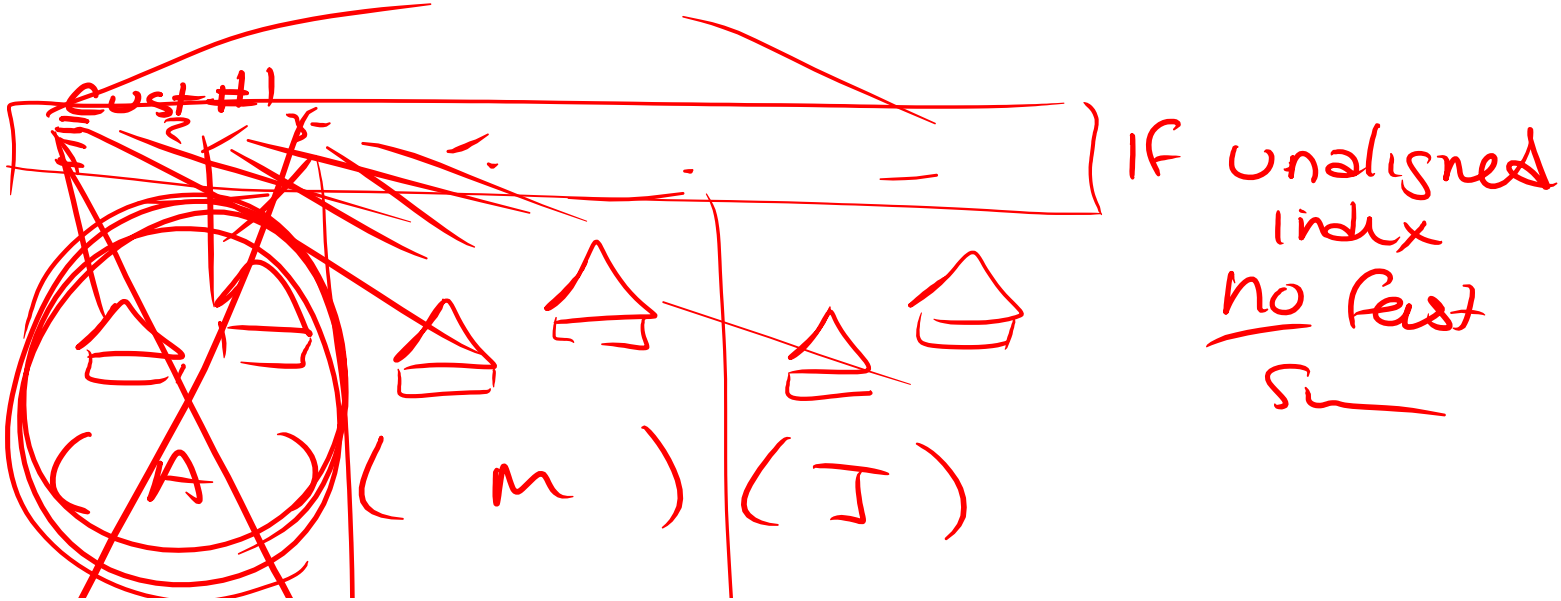
Alternatively, using partitioned tables – there's no way to have a gap or overlapping ranges.

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!



Expand Views

Another way to look at this is that there are multiple phases of query processing. The second phase (standardization, normalization, algebrization) is where views are expanded to their base tables (known as EXPANDVIEWS). This might sounds familiar because of the hint NOEXPAND – which is used to force SQL Server to use indexes on views (if for some reason SQL Server isn't using the IV).



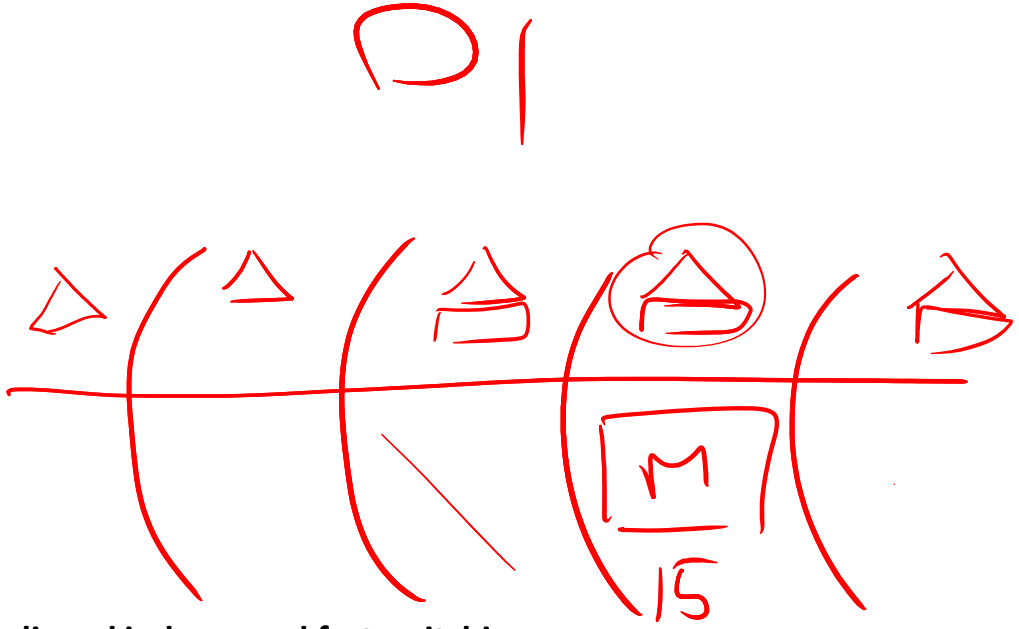
Aligned Indexes

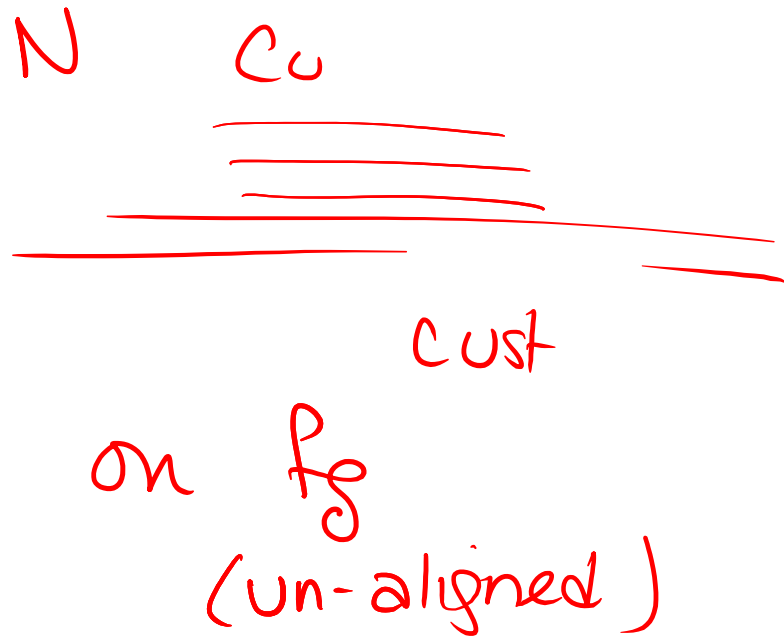
Indexes can be aligned to the same partition scheme:

- Either by creating them ON SCHEME(col) or
- Accepting the default behavior during creation. Nonclustered indexes default to being created on the same scheme as the clustered index – unless they are an UNIQUE index. If the index is unique then the partitioning column must [explicitly] be part of the key.

Indexes can be unaligned

- The index has all of the nonclustered index data for the table in one leaf structure
- Fast-switching is NOT allowed if unaligned indexes exist

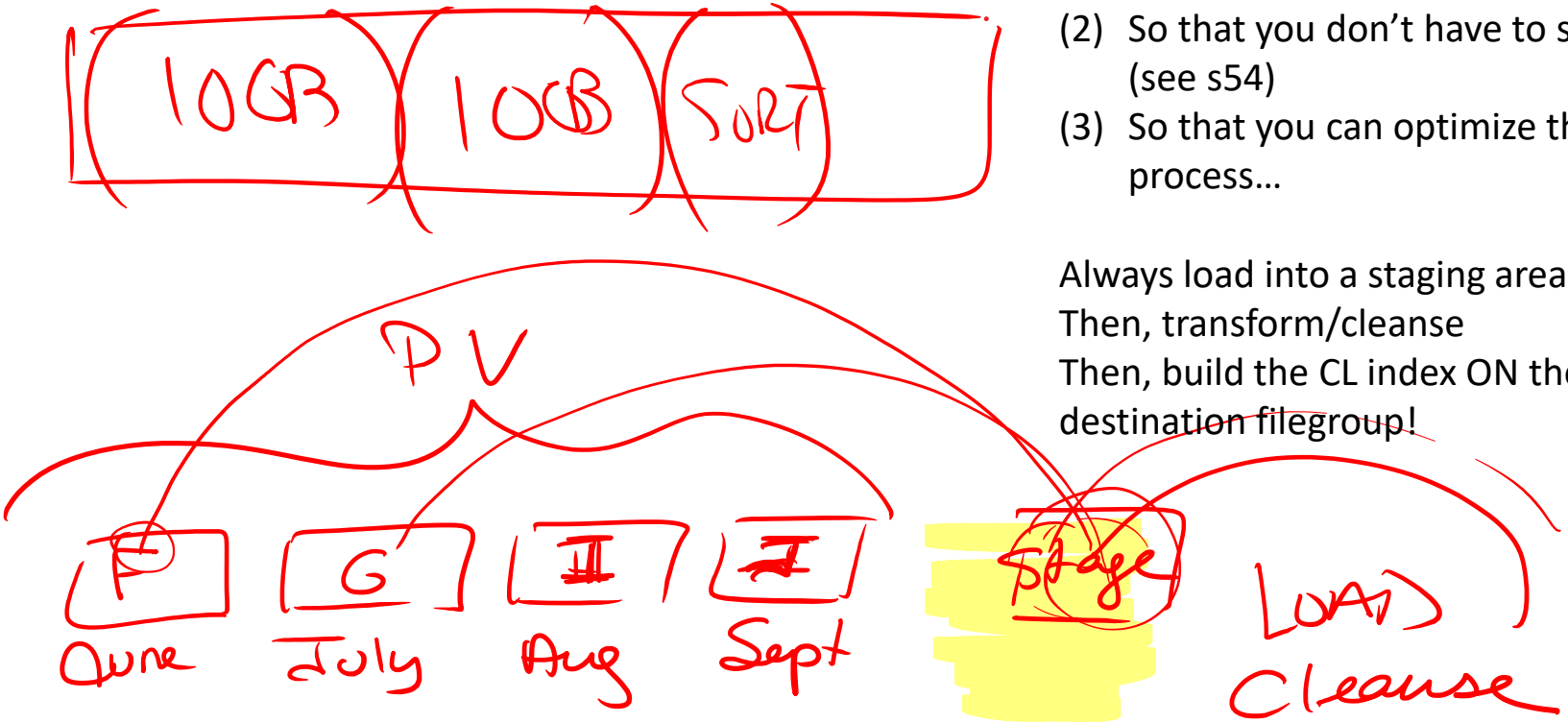




Un-aligned indexes and fast switching

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.



Staging Data

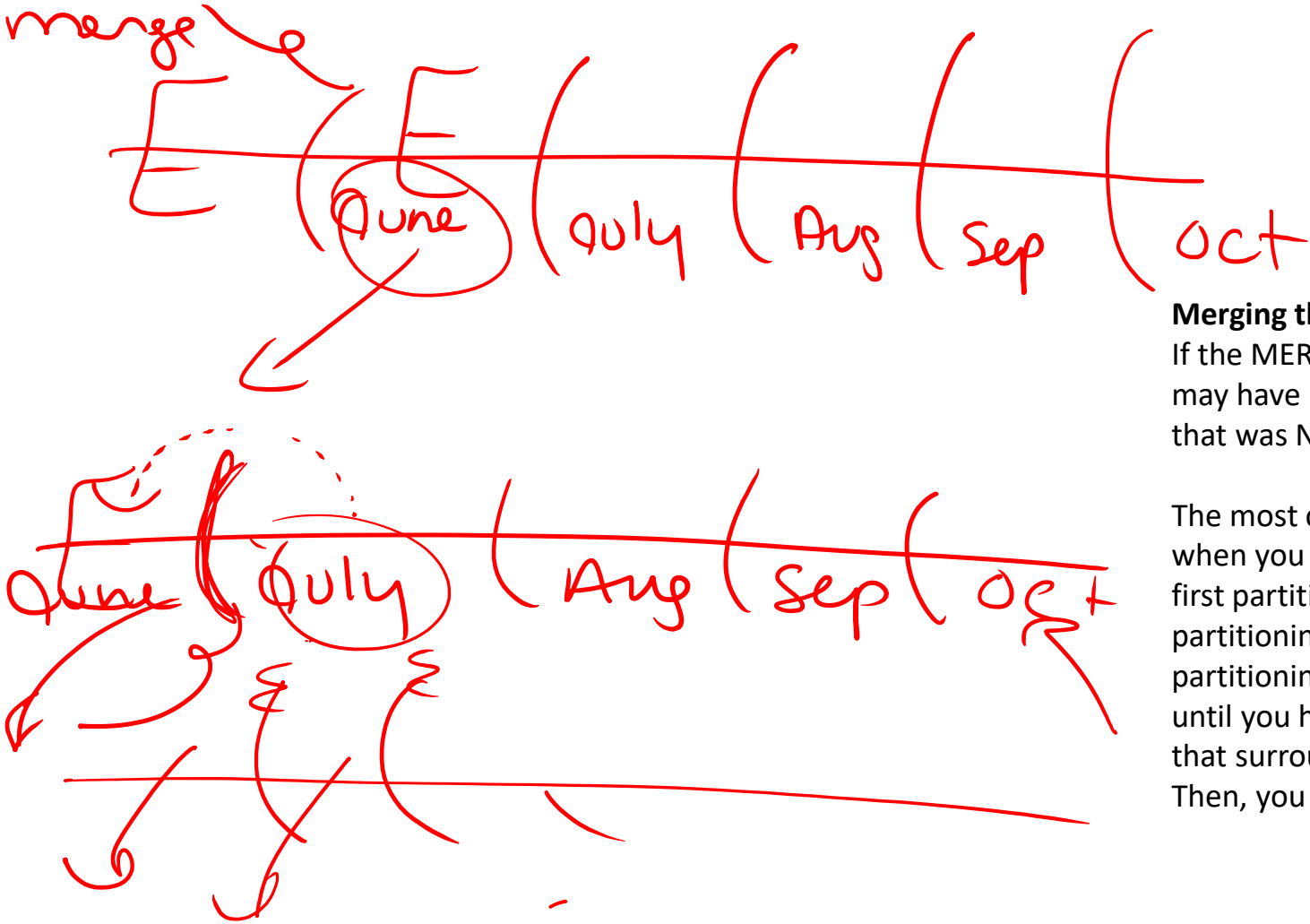
Why do you need a staging area?

(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!

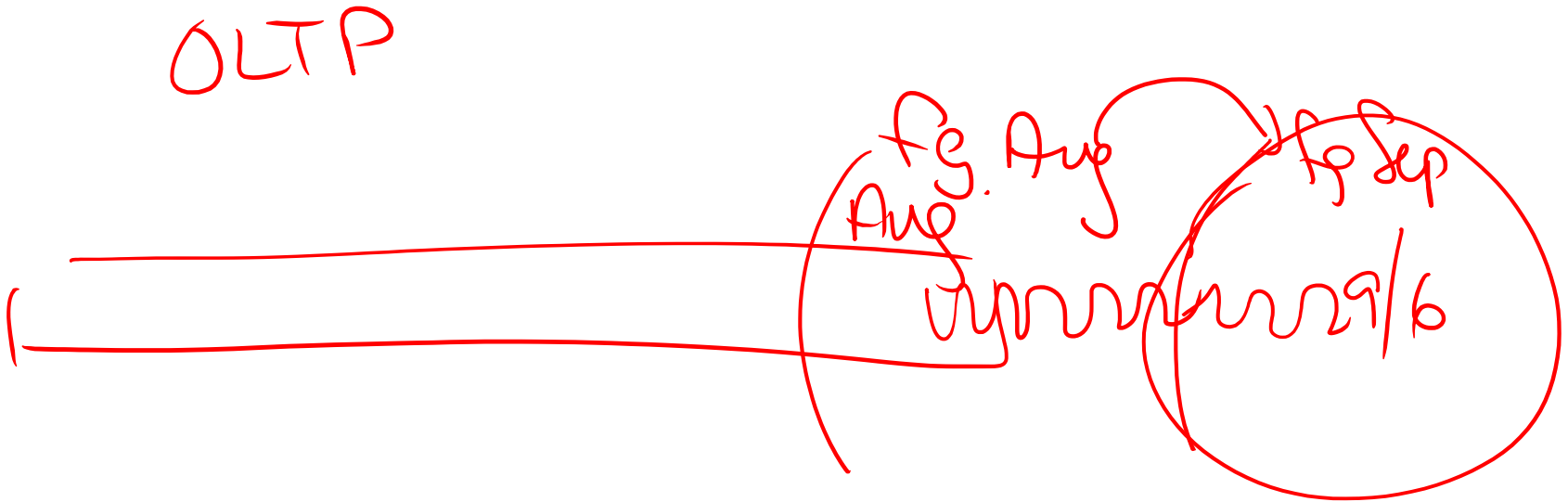


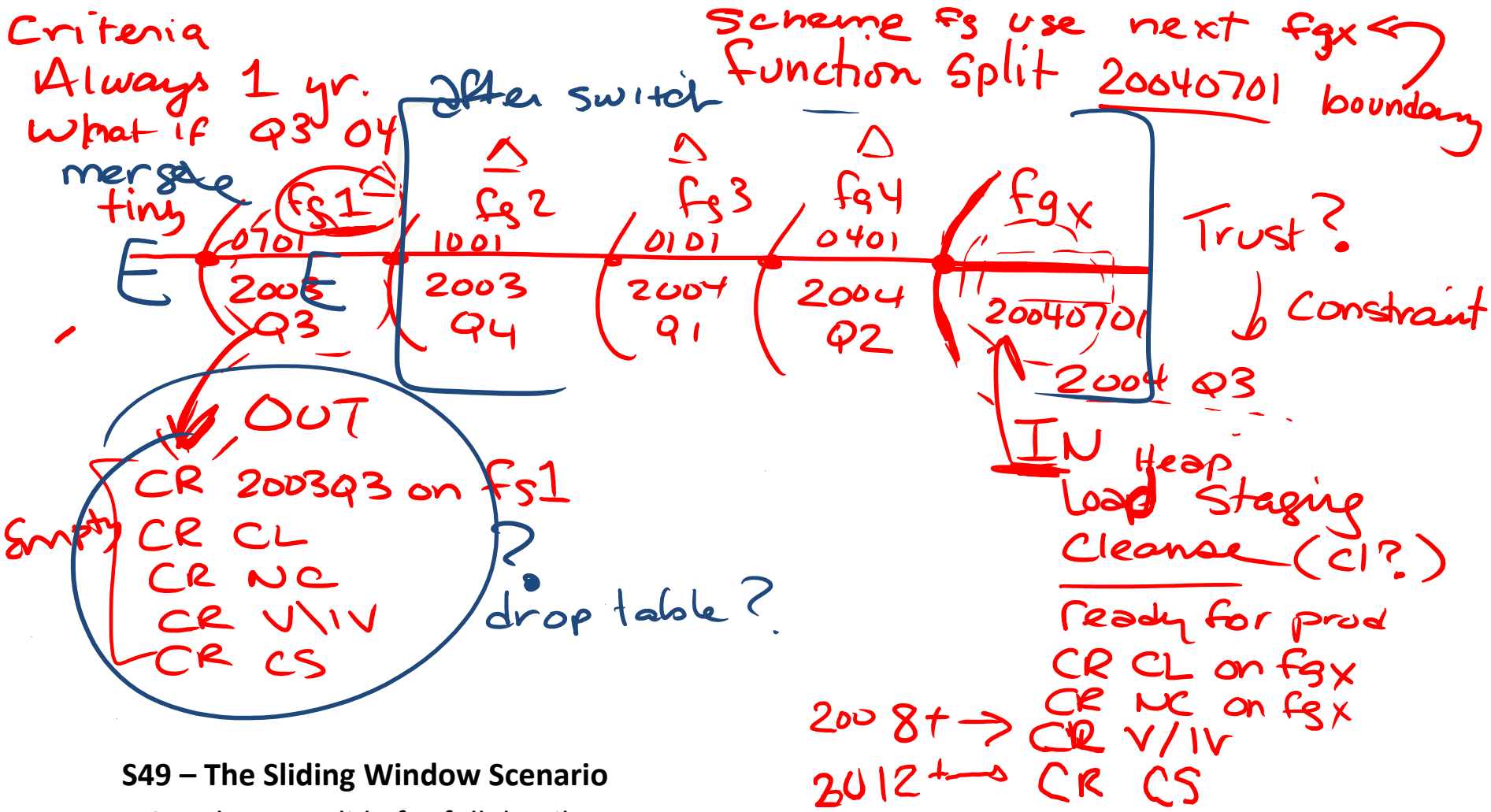
Merging the right boundary
If the MERGE process is slow – you may have merged a boundary point that was NOT empty.

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).

Rebuilding the active partition – OFFLINE ☹

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.





S49 – 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

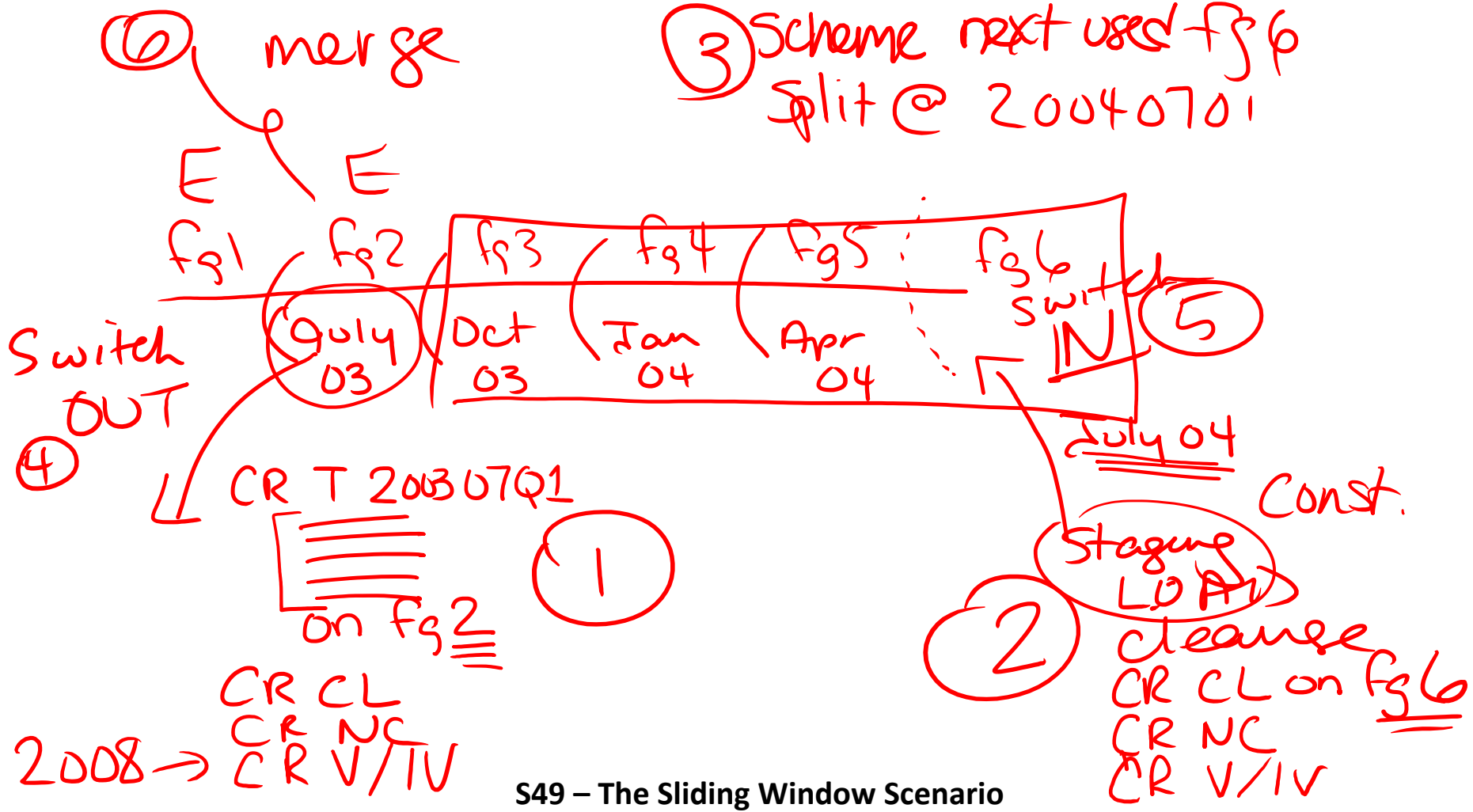
- Now, you're ready

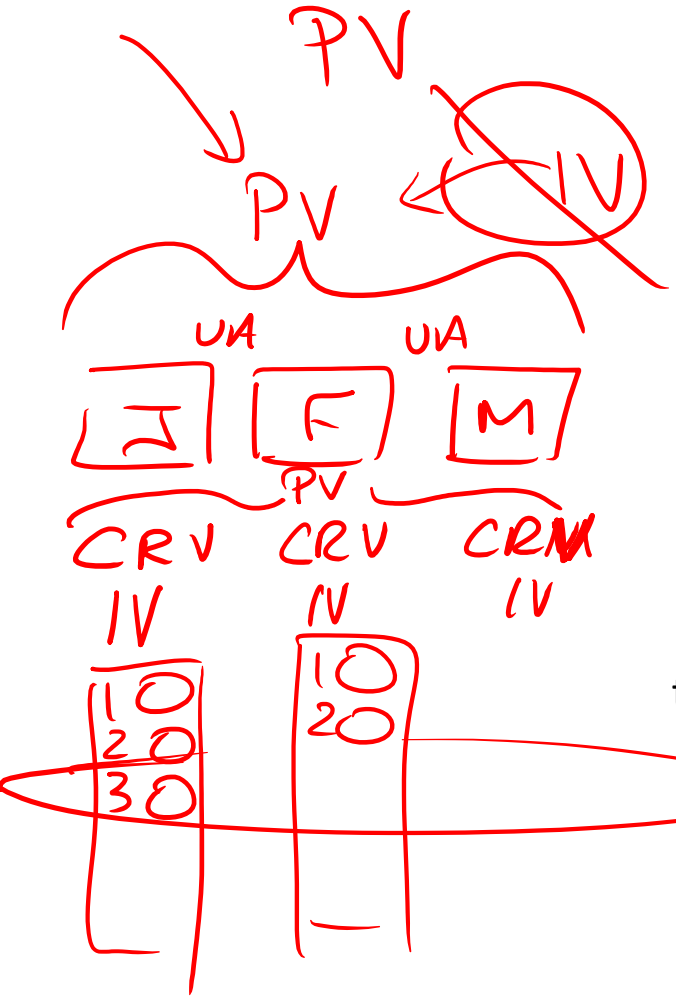
(4) and (5) can be in any order. SWITCH IN the new partition and SWITCH OUT the old.

(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)

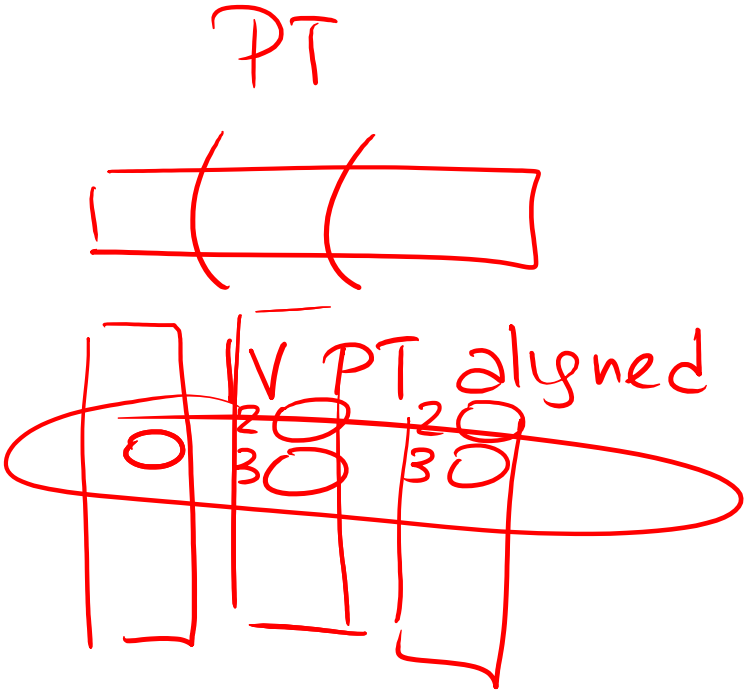
Backup the filegroup where the partition resides that you just switched out. OR, drop the table.

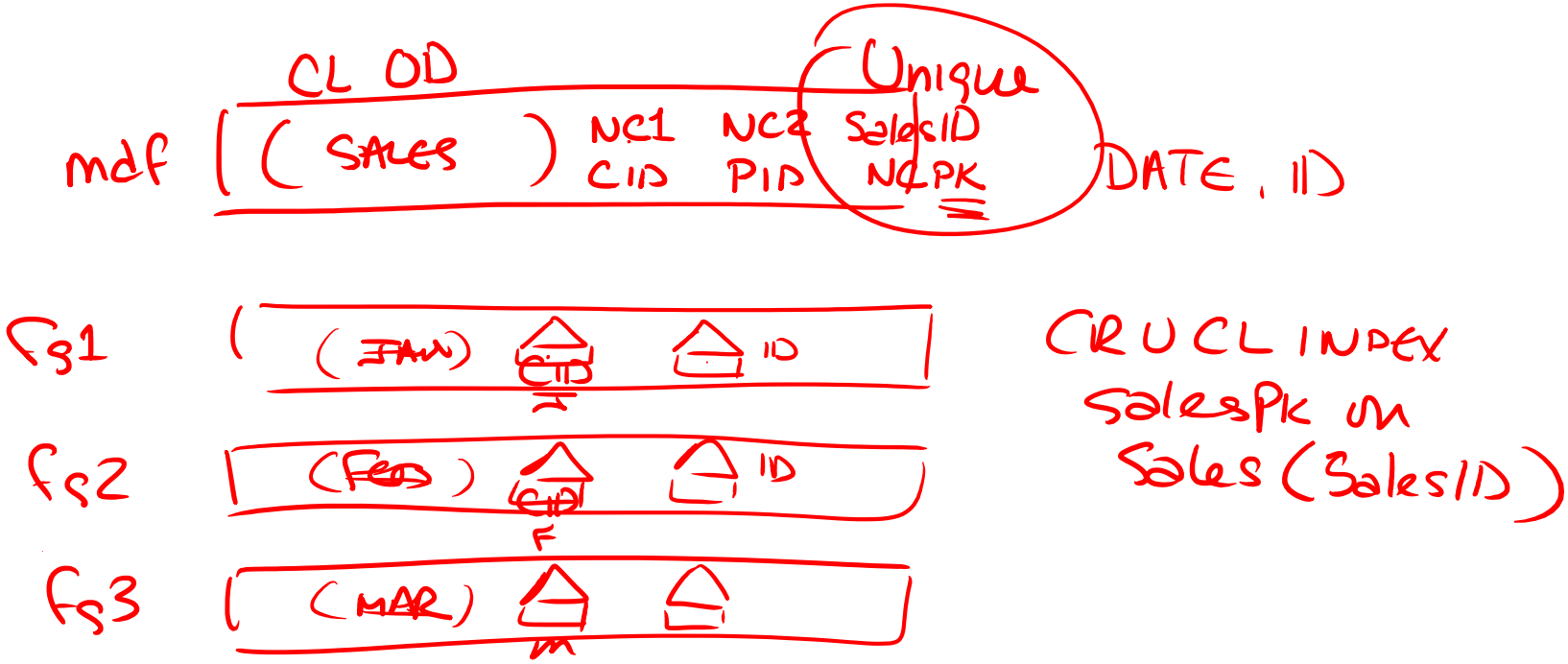

6 merge
3 Scheme next used fg6
Split @ 20040701
E E
fg1 fg2 fg3 fg4 fg5 fg6
July 03
Oct 03
Jan 04
Apr 04
switch IN
5
Switch OUT
4
July 04
CR T 20030701
1
on fg2
Const.
Staging LOAD
2
cleared
CR CL on fg6
CR NC
CR V/IV
CR CL
CR NC
CR V/IV
2008 →



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

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

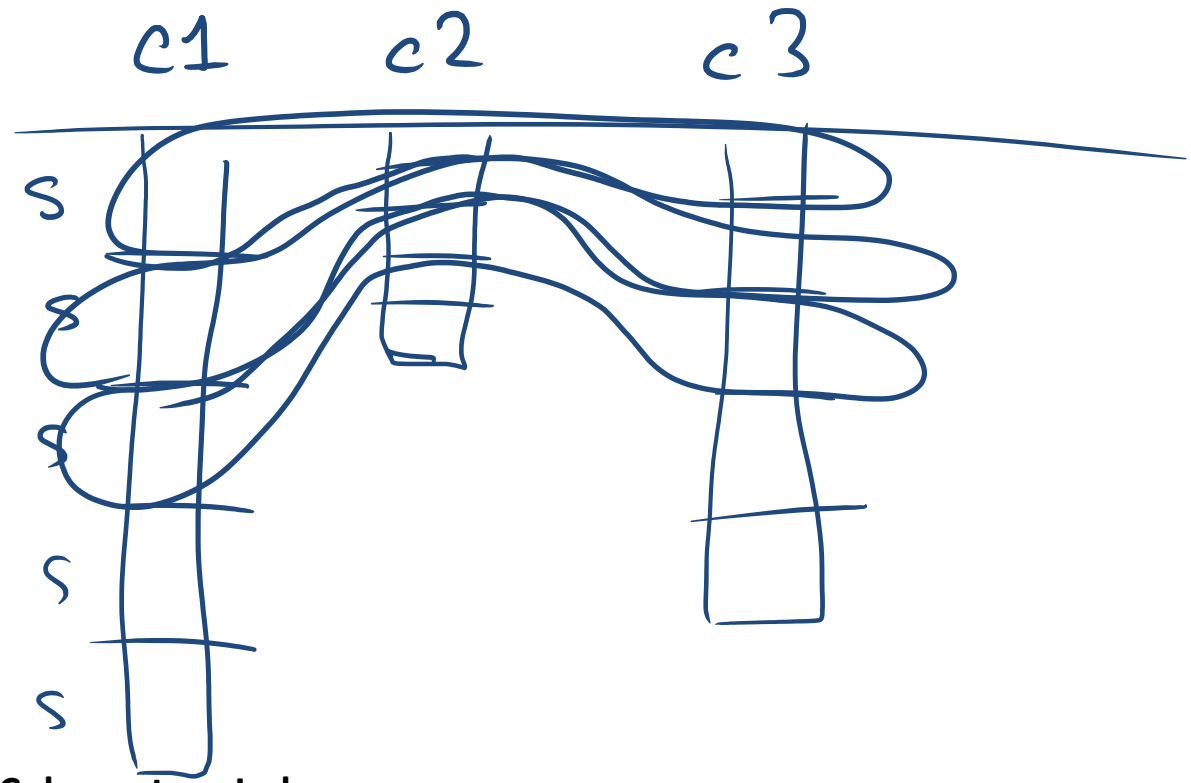




Moving/partitioning the CL index

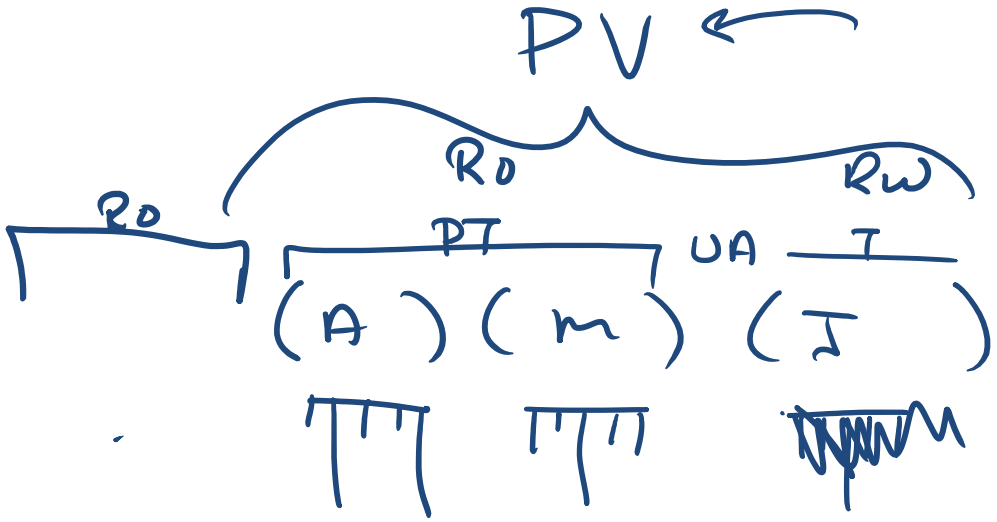
When you rebuild the CL index on a scheme the clustered index (the table) will be partitioned on the new scheme. However, nonclustered indexes will NOT be partitioned. Nonclustered indexes must be built separately/individually. And, they might need to be changed. All unique nonclustered indexes must having the partitioning column defined as part of the key.

If the CL table that you want to partition is clustered by date/id (and is the PK) then rebuilding that on the scheme is fairly easy. But, if it's not the PK and the PK is instead on SalesID then you'll change this PK to have the partitioning key as part of the PK. And, this means all FKs will need to change, etc. This can make the process of converting to a partitioned table become an offline process.

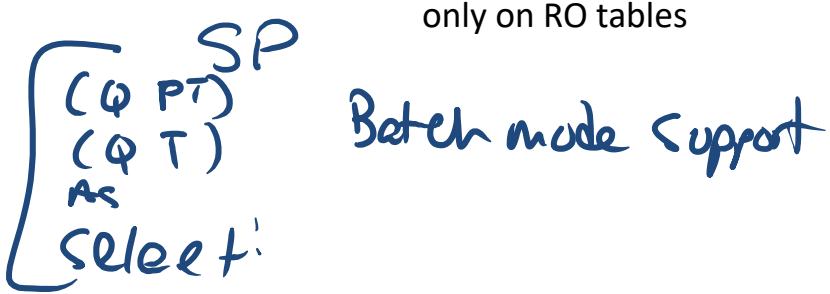


Columnstore Indexes

This was a drawing showing the possible compression of 3 columnstore-based indexes. Then, each columnstore index is broken down into segments. This is the base of batch-mode processing (another core part to the performance gains that can be recognized with columnstore indexes).



Might want to have CS indexes only on RO tables

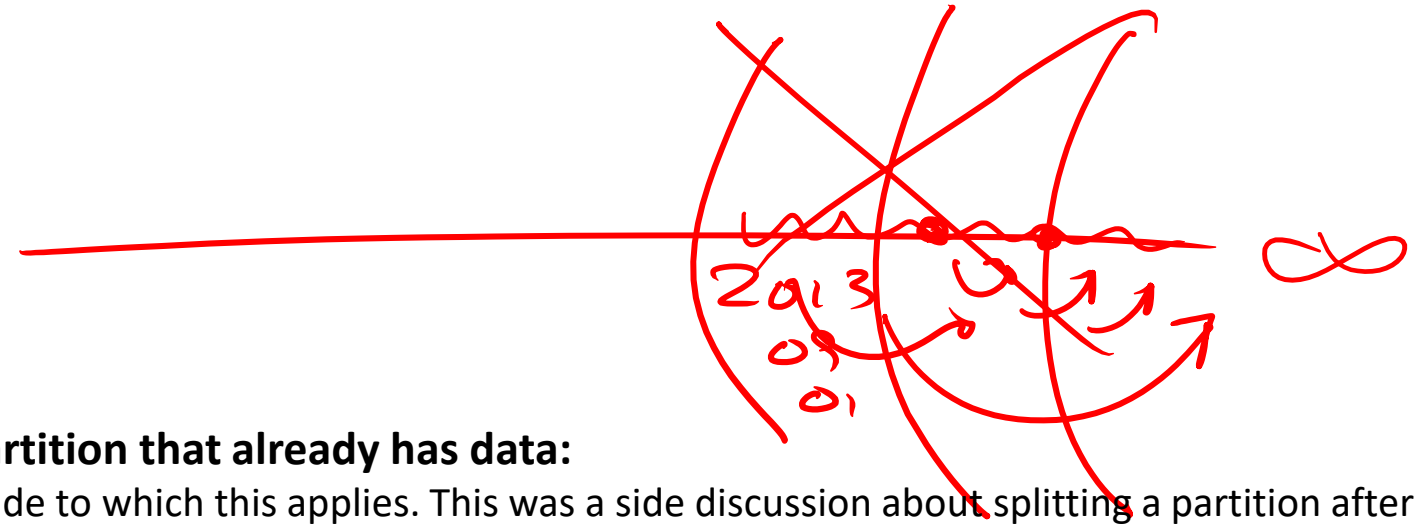


Batch mode support

Since columnstore indexes make the table on which they're created read-only then you might want to use PVs to separate read-write and read-only data.

Having said that, one limitation of the current implementation of nonclustered columnstore indexes is that they do not support batch mode processing across views that include UNION ALL. The workaround is to use CTEs.

Check out the discussion: **Perform UNION ALL and Still Get the Benefit of Batch Processing** on the columnstore wiki here: http://social.technet.microsoft.com/wiki/contents/articles/perform-union-all-and-still-get-the-benefit-of-batch-processing.aspx



Splitting a partition that already has data:

There isn't a slide to which this applies. This was a side discussion about splitting a partition after it already has data.

If you forget to split for October and November and the last split was for Sept 1 – then, instead of splitting for October (which has to move BOTH October and November data) and then splitting for November (which has to move November's data again) – you should always split the last set (November) and then the earlier (October). Then, November's data only moves once and October's only once as well

However, if you're going to make significant changes to a partitioning scheme (like 4 partitions to 8) then instead of splitting 4 times – just create a new function and new scheme and then rebuild (possibly online) the object on the new scheme.

SQLskills Immersion Event

IEHADR: High Availability and Disaster Recovery

Backup Strategy and Internals

Kimberly L. Tripp

Kimberly@SQLskills.com

These were originally from a 4x3 format so they're not perfect in widescreen but the info's good! ;-)

Trace Flags To Stop Log Dumping

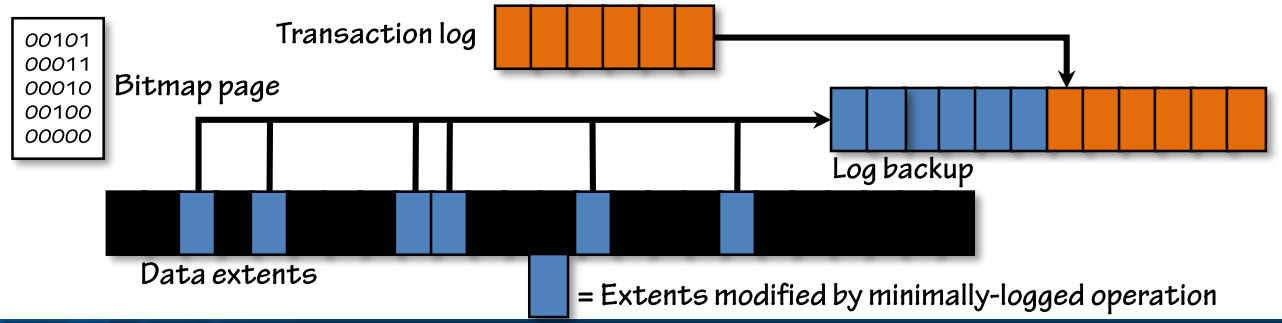
- **To change the behavior of BACKUP LOG WITH TRUNCATE_ONLY or NO_LOG:**
- **In SQL Server 2000**
  - Trace flag –T3231
    - In FULL or BULK_LOGGED recovery model they do nothing
    - In SIMPLE recovery model they will clear the log
- **In SQL Server 2005**
  - Trace flag –T3231 works the same way as in 2000
  - Trace flag –T3031 does a checkpoint
- **In SQL Server 2008+**
  - Manually clearing the log is no longer possible, but some people may still do it by switching back-and-forth to SIMPLE recovery model

Continuity of the Log Backup Chain

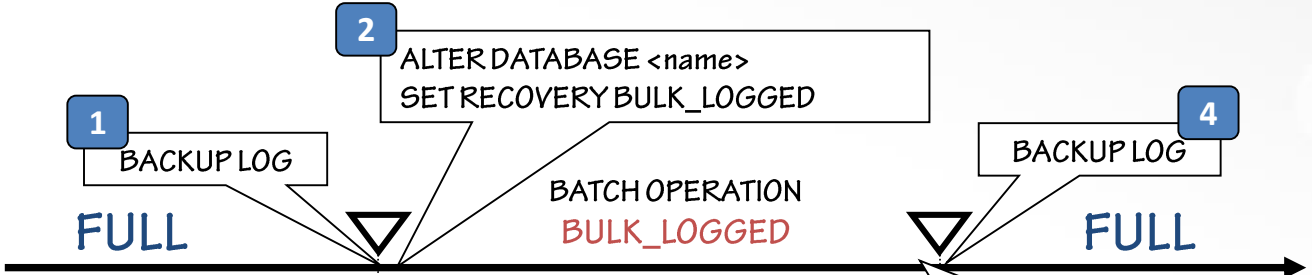
- **If the log backup chain is not complete (meaning one or more log backups are damaged or missing) then complete recovery is not possible**
  - Must have complete chain of log backups from which to restore
- **Consider using mirrored backups of the transaction log and/or storing them on redundant disks**
- **What breaks the log chain?**
  - Clearing the transaction log manually (using WITH TRUNCATE_ONLY or WITH NO_LOG)
  - Changing the database to SIMPLE recovery model (and then changing back)
  - Deleting, overwriting or throwing away a log backup that was not made as COPY_ONLY
  - Reverting from a database snapshot

Log Backups After Minimally-Logged Operations

- **If there has been a minimally-logged operation since the previous log backup, the next log backup will also back up all data extents modified by the minimally-logged operation**
  - This means the minimally-logged operation can be fully reconstituted
- **The resulting log backup will be roughly the same size as if the operation was performed in the FULL recovery model**
  - A point-in-time restore operation cannot stop at any point between the start and end of the log backup



Switching Recovery Models



Advantages:

- Up-to-the-minute recovery
- Point in time recovery
- Access to "tail" of the log

Advantages:

- Minimal logging
- Faster bulk operation

Disadvantages:

- No STOPAT
- Data loss possible if media failure…
- If database becomes suspect, cannot back up tail of log.