



Module 9

Plan Cache & Index Analysis

SQL
skills

Reducing Plan Cache Pollution (1 of 3)

- ◆ Server setting: Optimize for adhoc workloads
 - ◆ On first execution, only the query_hash will go into cache. For the prior query this is only 380 bytes (compared to the 24K of the plan)
 - ◆ On second execution (if), the plan will be placed in cache
- ◆ Create a single and more consistent plan with covering indexes – might make the statement safe!
 - ◆ A lot of limitations to SQL Server detecting this (see Appendix A of the “Plan Caching in SQL Server 2008” whitepaper) but if the statements are actually safe...
 - ◆ Consider database setting: forced parameterization
 - ◆ Not as highly recommended but if you’re finding A LOT of single-use plans **with only one query_plan_hash** then this might be great!

Reducing Plan Cache Pollution (2 of 3)

Two primary scenarios to consider

Analyze the plan cache for the number of query plans per query hash (as well as the number of executions)

```
SELECT qs.query_hash
      , COUNT(DISTINCT qs.query_plan_hash) AS [Distinct Plan Count]
      , SUM(qs.EXECUTION_COUNT) AS [Execution Total]
FROM sys.dm_exec_query_stats AS qs
     CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS st
     CROSS APPLY sys.dm_exec_query_plan(plan_handle) AS qp
WHERE st.text LIKE '%member%'
GROUP BY qs.query_hash
ORDER BY [Distinct Plan Count] DESC
```

Scenario 1

query_hash	# Plans	# Executions
0x04BB791B589774AD	1	6456456
0x1706E9EC3049A95B	6	276543
0x5BD9FF487079B335	1	124345
0x6604520C5200ABC0	1	78905
0x77BA5A89C7EBE605	1	14342
0xA078B4BC8768A9A6	1	4567
0xB81E270A58A79D16	1	6

Mostly stable plans (only 1 plan per hash)

Scenario 2

query_hash	# Plans	# Executions
0x04BB791B589774AD	34	6456456
0x1706E9EC3049A95B	6	276543
0x5BD9FF487079B335	8	124345
0x6604520C5200ABC0	3	78905
0x77BA5A89C7EBE605	2	14342
0xA078B4BC8768A9A6	9	4567
0xB81E270A58A79D16	24	6

Mostly UNstable plans (multiple plans per hash)

Scenario 2

*Generally,
more likely...*

Reducing Plan Cache Pollution (3 of 3)

◆ Scenario 2: Default parameterization mode: SIMPLE

- ◆ Use “templatized” plan guides to take the few statements that are safe and make them forced (reduced compilation/CPU)

```
EXEC sp_create_plan_guide  
    Name_of_plan_guide  
    templatized_version_of_safe_query,  
    N'TEMPLATE',  
    NULL,  
    @Parameters,  
    N'OPTION(PARAMETERIZATION FORCED)';
```

◆ Scenario 1: Consider changing parameterization to FORCED

- ◆ Use “templatized” plan guides to take the few statements that are NOT safe and make SIMPLE (recompiled)

```
EXEC sp_create_plan_guide  
    Name_of_plan_guide  
    templatized_version_of_unsafe_query,  
    N'TEMPLATE',  
    NULL,  
    @Parameters,  
    N'OPTION(PARAMETERIZATION SIMPLE)';
```

update stats

↳ plan invalidation?

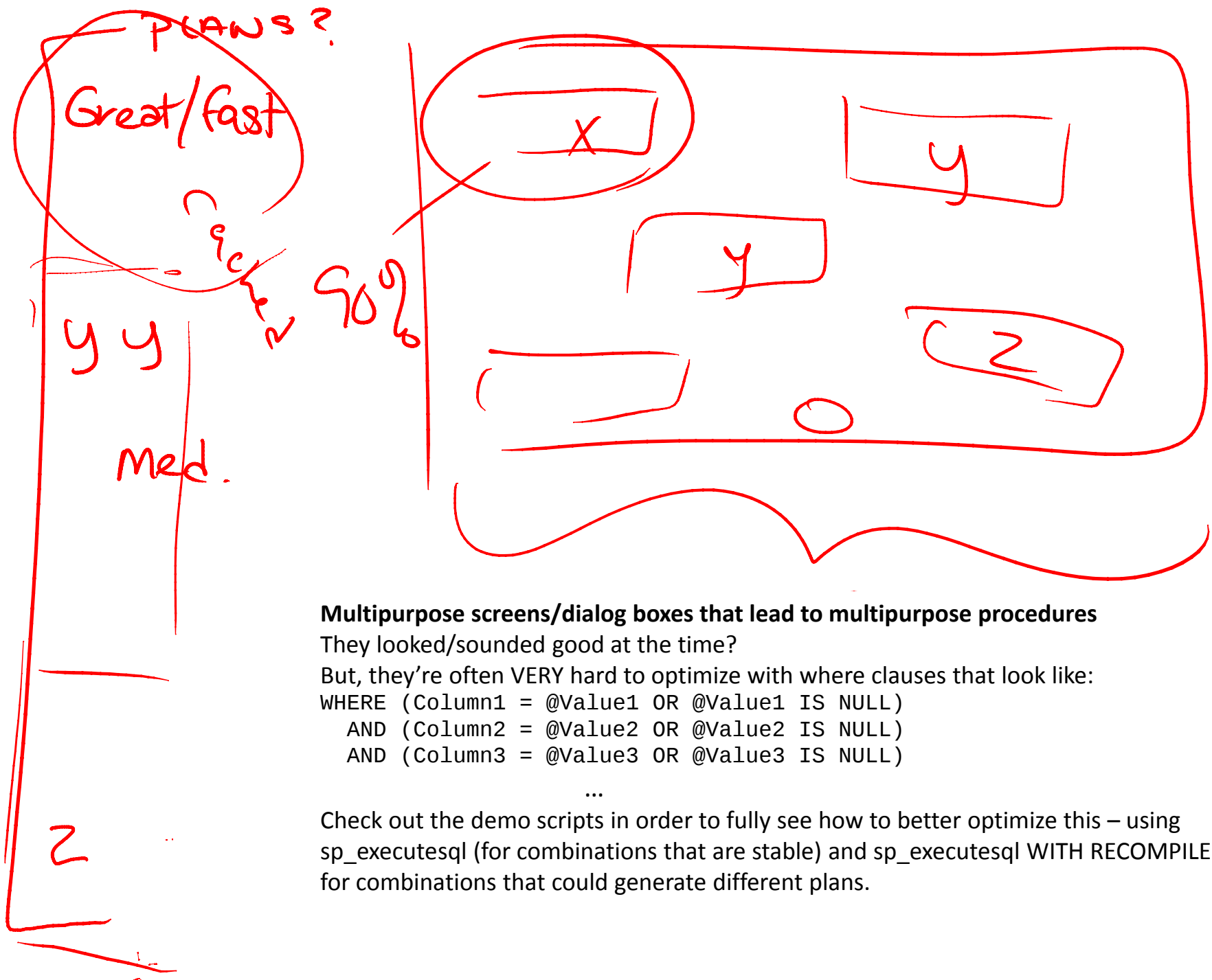
auto update
stats
ON

OFF

This was just a small discussion about the fact that plan invalidation does not actually occur if you don't have auto update statistics (as a database option) turned off. Check out the demo script in the plan cache demos (I just added it); `DoNotTurnOffAutoUpdateStatistics.sql`

Also, be sure to review Erin Stellato's links about auto update stats and plan invalidation:

- Statistics and Recompilations: <http://erinstellato.com/2012/01/statistics-recompilations/>
- Statistics and Recompilations, Part II: <http://erinstellato.com/2012/02/statistics-recompilations-part-ii/>



Multipurpose screens/dialog boxes that lead to multipurpose procedures

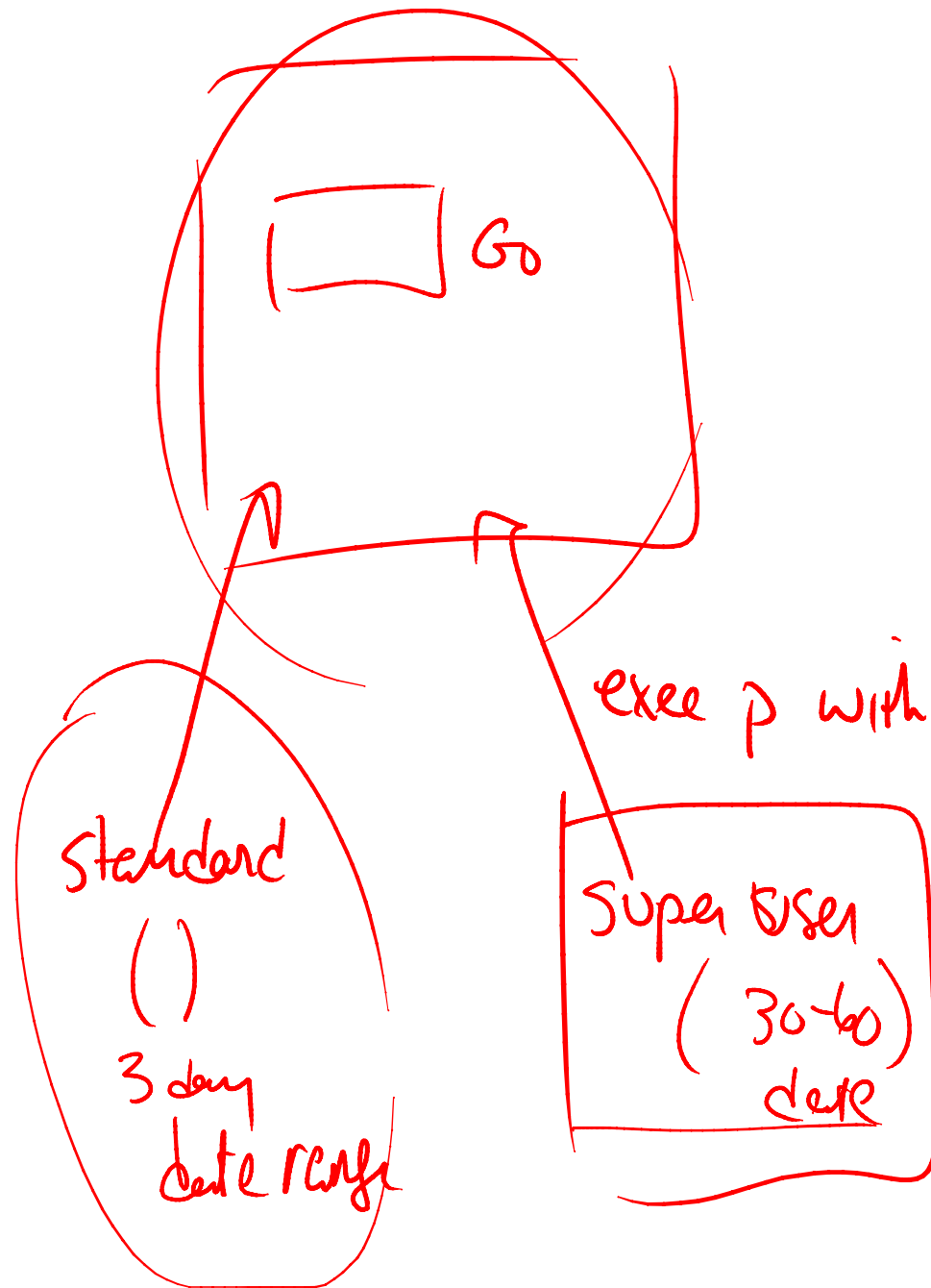
They looked/sounded good at the time?

But, they're often VERY hard to optimize with where clauses that look like:

```
WHERE (Column1 = @Value1 OR @Value1 IS NULL)
      AND (Column2 = @Value2 OR @Value2 IS NULL)
      AND (Column3 = @Value3 OR @Value3 IS NULL)
```

...

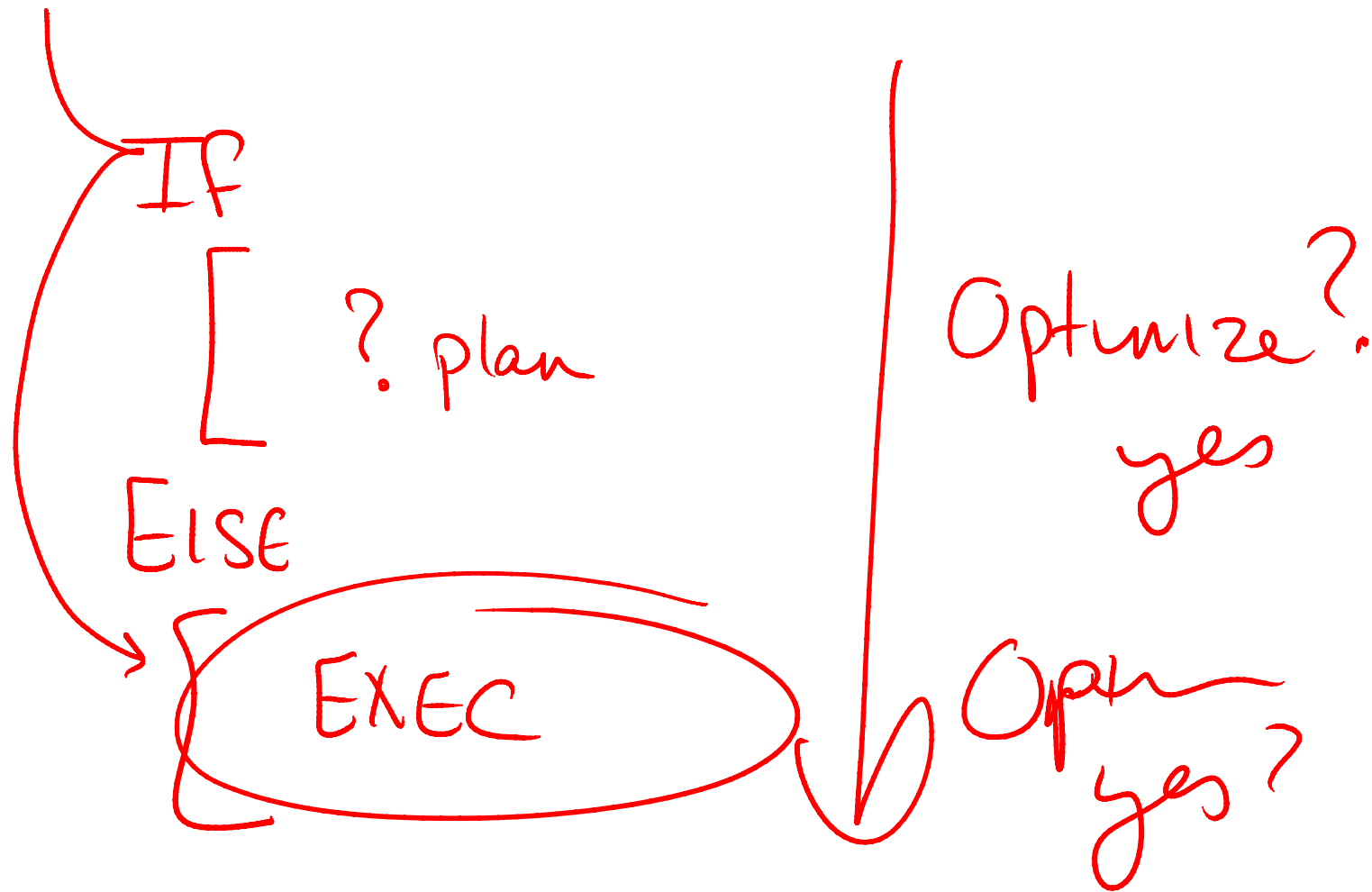
Check out the demo scripts in order to fully see how to better optimize this – using `sp_executesql` (for combinations that are stable) and `sp_executesql WITH RECOMPILE` for combinations that could generate different plans.



Options for different executions

If we largely execute with “standard” users and we absolutely want THAT plan to be in cache and to be re-used then we could:

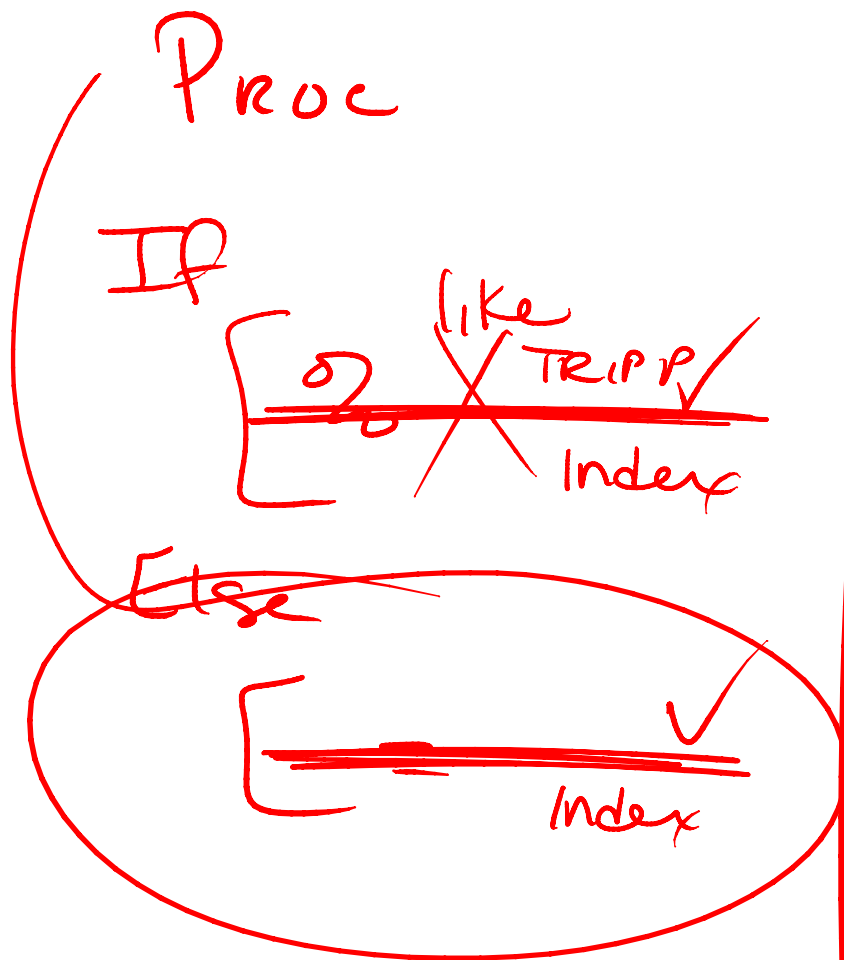
- 1) Use option (optimize FOR...) and use a literal that will generate a plan that’s great for the “typical” scenario. However, the performance won’t be great for the atypical (super user) scenario.
- 2) We could use EXEC for the typical scenario and EXEC proc WITH RECOMPILE for the super user scenario. The benefit of this is:
 - 1) We get a cached plan for the typical user (no CPU/plan stability)
 - 2) We get a specific plan for the atypical user (the super user) as opposed to a possibly inefficient plan (so, we have to compile but we’re only compiling this one type of plan)



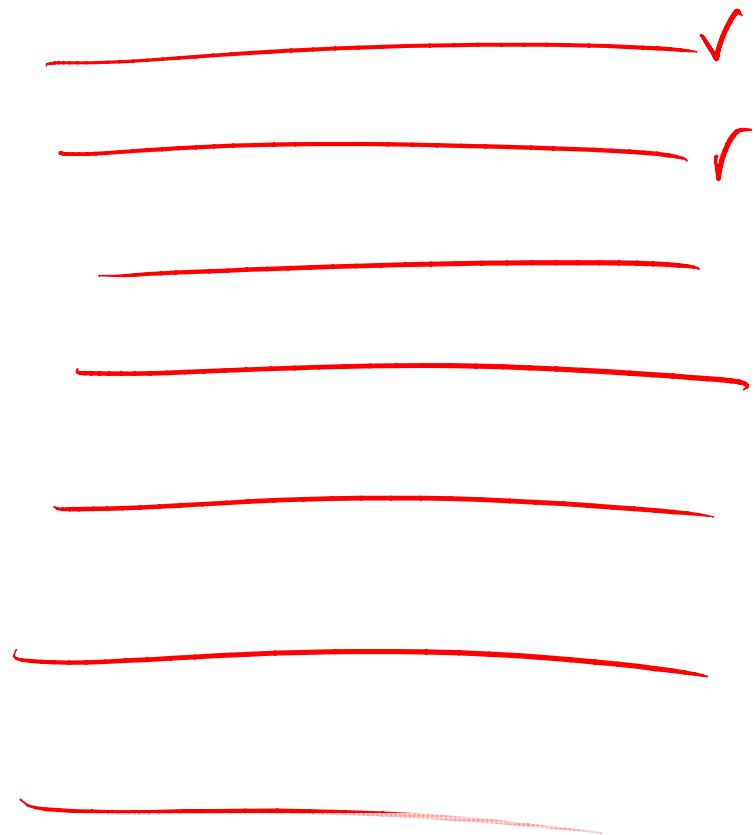
Conditional logic and modularization

Creating branching logic for different types of parameters seems logical but because SQL Server optimizes the process of optimization (where they try to optimize anything that's optimizable), you can often end up with a plan that's not ideal.

You are much better off breaking that code into smaller subprocedures – SQL Server will never step into a subprocedure unless it's executed and in this case it will only be executed with exactly the right parameters.



Proc



Conditional logic and modularization

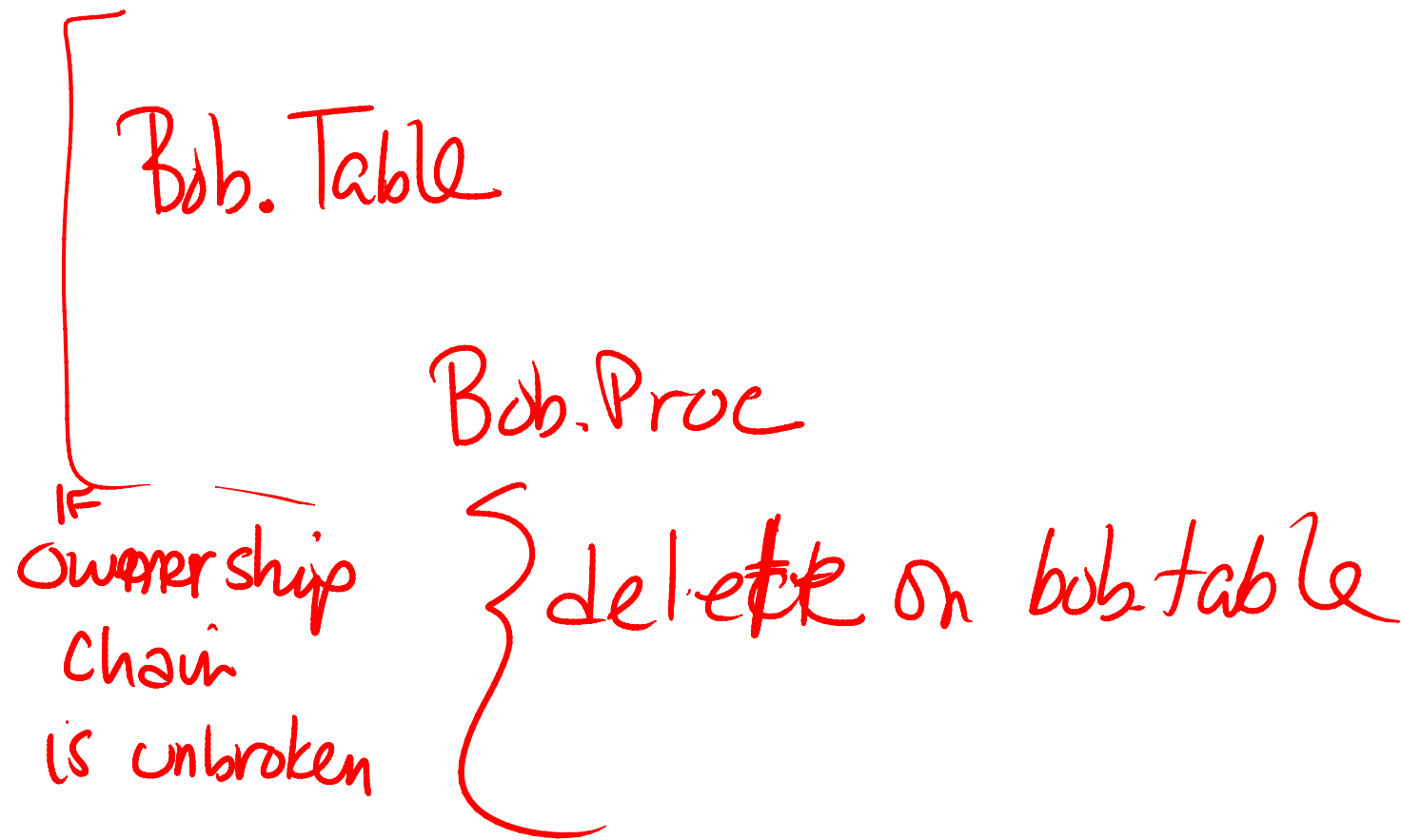
This is another way of looking at the same thing from the prior diagram. Really, don't think of the "logic" in your procedure; think only of the procedure as a list of statements (regardless of conditional logic). SQL Server tries to optimize ANY statement that's "*optimizable*" with the parameters that were supplied (regardless of those specific parameters apply for *that* execution). This is another case where you can end up with an inefficient plan because of parameter sniffing.

DEFAULT
Proc

DSE — protects against SQLinj
execute under context
CALLER

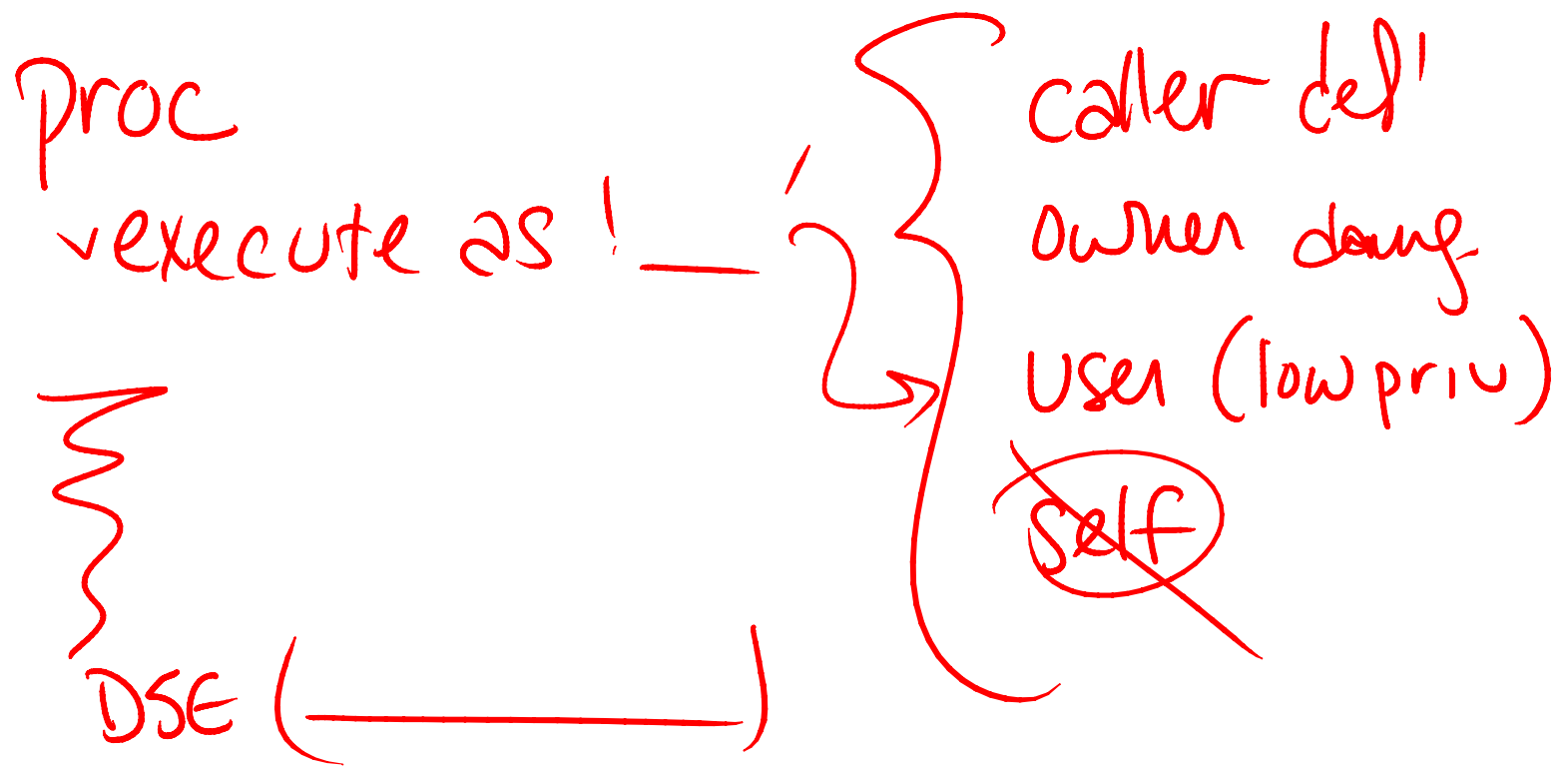
Dynamic String Execution and SQL Injection

TO prevent SQL Injection, SQL Server requires that strings execute under the context of the caller.



Without Dynamic String Execution permissions flow through the ownership chain

The idea is that when the ownership chain is unbroken then direct permissions (for example, to do a delete) are not necessary. As an owner you are giving rights to a process; this procedure can be protected with more business rules/logic and even reduced/isolated to specific rows. The end user does NOT need direct delete permissions to be able to execute the procedure as long as they've been given execute rights on the sproc.



Dynamic String Execution, EXECUTE AS and SQL Injection

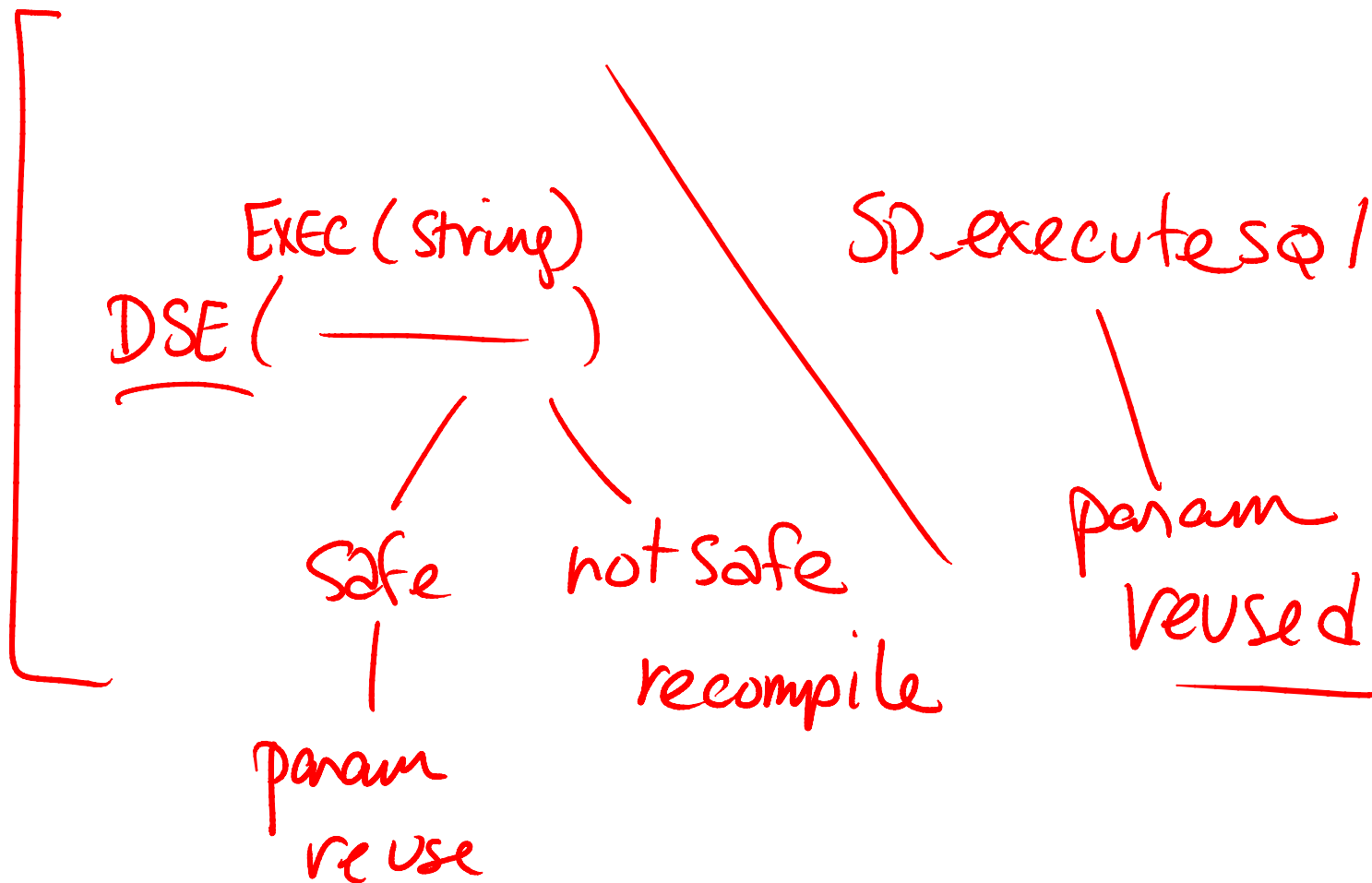
People complained that granting permissions directly to the user was a problem... so, they wanted an alternative. Enter: EXECUTE AS.

While it solves the problem of permissions, it adds more potential for SQLInjection.

So..... Check out these posts:

[Little Bobby Tables, SQL Injection and EXECUTE AS](#)

["EXECUTE AS" and an important update your DDL Triggers \(for auditing or prevention\)](#)



Using EXEC (string) [DSE] and/or sp_ExecuteSql [ES] inside of stored procedures

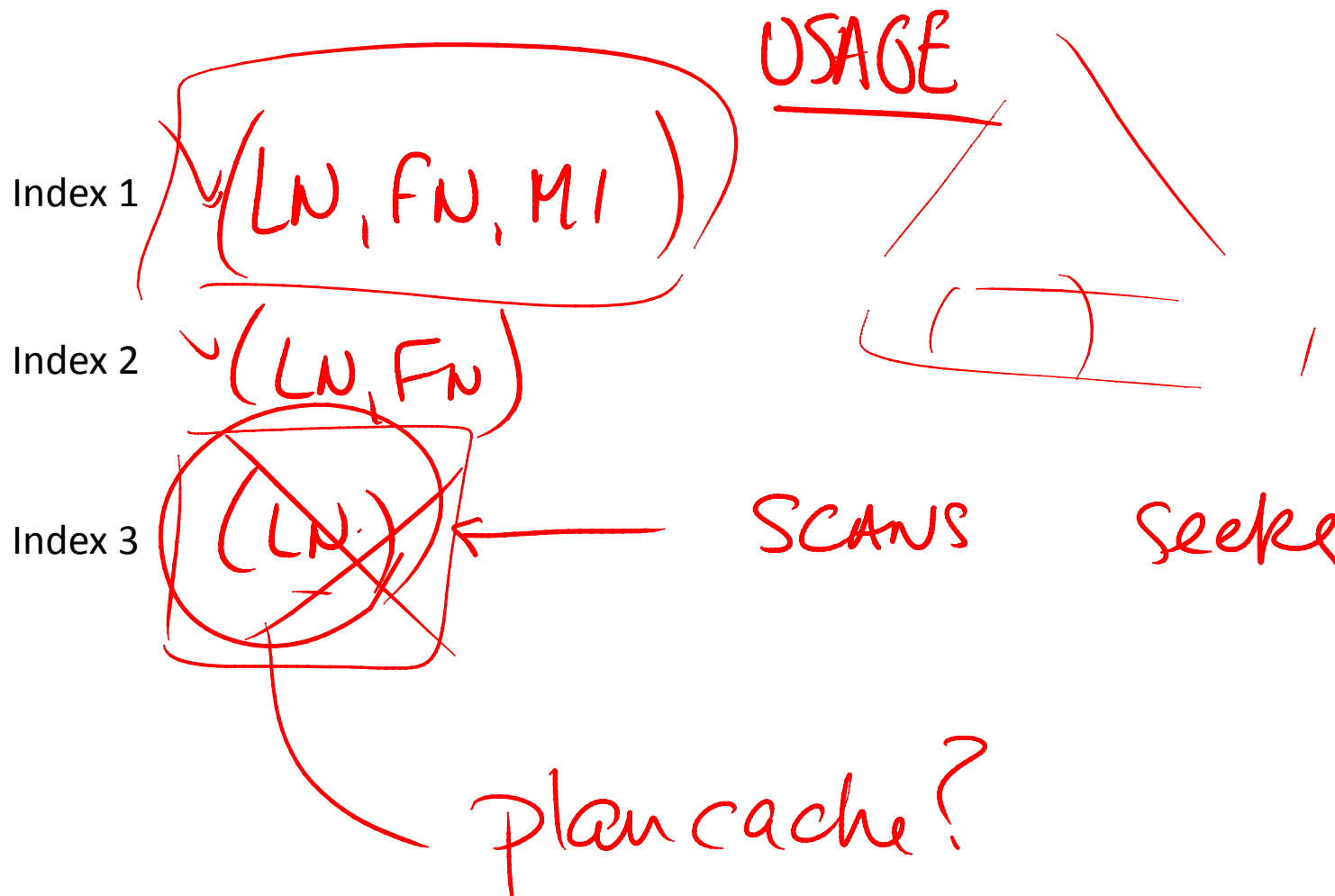
DSE and ES are not the same.

DSE is unknown until runtime. At that point the statement is treated as though it's adhoc. Because most statements are **not** going to be safe, the statement will end up being recompiled (and optimized) for each execution.

ES is forced parameterization. When a statement is stable you can use this to reduce compilation/CPU costs.

Showplan and Functions

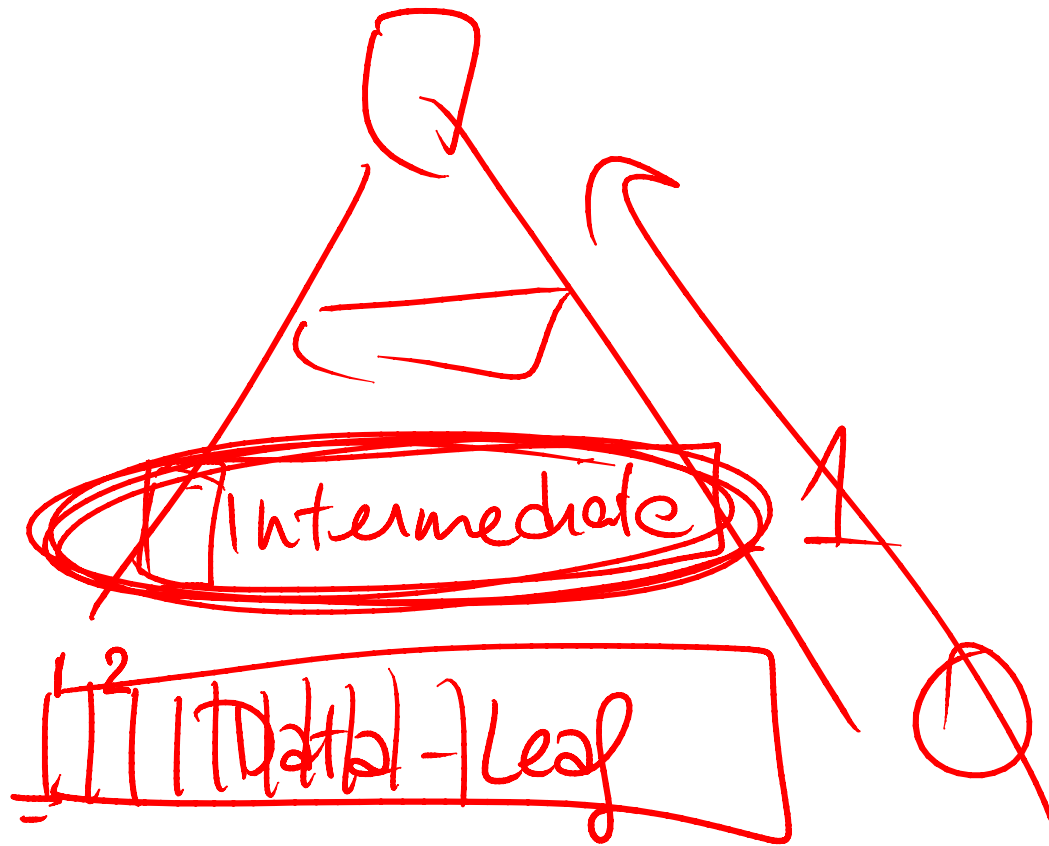
- ◆ TWO concerns
- ◆ Estimates (and therefore the percentages in showplan)
 - ◆ They don't correct cost-estimate the function so any plan that includes one is going to be estimated lower than it likely is (and possibly WAY lower if there's data access involved)
- ◆ Actual showplan
 - ◆ Since "actual" requires execution in a different mode – the actual execution is even more expensive... so, not only is the plan wrong but if you ask for the plan, it takes even longer to execute
 - ◆ 1 min, 8 secs (without showplan)
 - ◆ 18 mins, 25 secs (with showplan)



Should you drop redundant indexes? It depends. 😊

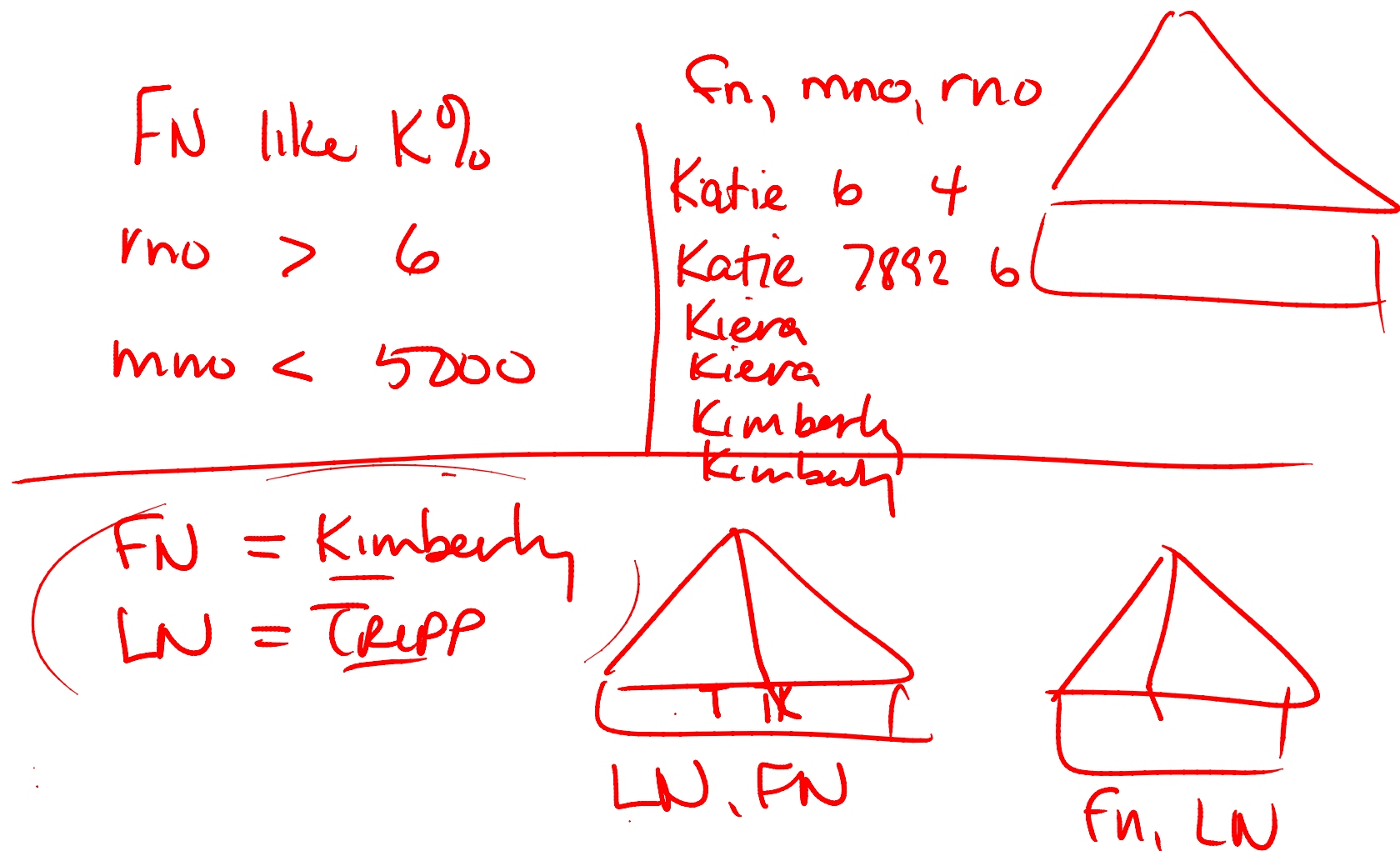
It's TRUE that query that uses a left-based subset of one index (let's say that a query uses index 2) that is COULD use a wider version of the same (left-based) index. It could use Index 1 instead of 2.

The problem is that there could be specific uses that warrant the smaller version. Questions to ask: Is the smaller version used by scans? Are they executed VERY frequently? How important are they?



Index Health – Checking for fragmentation

For a limited mode query against `sys.dm_db_index_physical_stats`, SQL Server uses a scan of the first intermediate level (index level 1) to analyze for the value: `avg_fragmentation_in_percent`. If your analysis/rebuild scripts use this to determine fragmentation than you'd be better off just doing a limited scan. What SAMPLED and DETAILED offer are page-specific details (page density, forwarding pointers, etc.) and if you're NOT using those (most people aren't!) then you're wasting time with a SAMPLED or DETAILED scan during your maintenance window!



Adding more indexes

Here we were discussing the order of columns. If there are range-based predicates then it doesn't usually matter which column is second. You tend to want to put the most selective first (in terms of the PREDICATES supplied). If all of the predicates are equality-based then it doesn't really matter. You're best ordering them by the LEFT-based combination that's used most commonly.

KEY

Density_Vector

c1	c2	c3	c4

_____	_____		
_____	_____	_____	
_____	_____	_____	_____
c2	c4		
c2	c3		
c3	c4		
c2	c3	c4	

Multi-column, column-level statistics

Only DTA recommends multi-column, column-level statistics...

If you are about to trust (and create) a “green hint” recommendation (which comes from the Missing Index DMVs) then you might want to first run the query through DTA. If you end up creating the index that DTA recommends then you should also create the recommended stats (for that same table). These statistics can be helpful to the optimizer to better understand non-left-based subsets of the index for other possible uses.

IE1 Slides (for reference)
Locking & Blocking

SQL
skills

Blocking – Is it Really a Problem?

It depends...

- ◆ Locks guarantee consistency
- ◆ First incompatible lock request waits
- ◆ Once an incompatible lock request is made then pending locks will wait even if they're compatible with locks currently held
 - ◆ Special case WHEN the requested lock IS compatible WITH ALL pending requests...
- ◆ Primary reason for the locks to WAIT
 - ◆ Why? To prevent lock starvation

Even compatible shared locks must WAIT (or queue) behind the incompatible lock

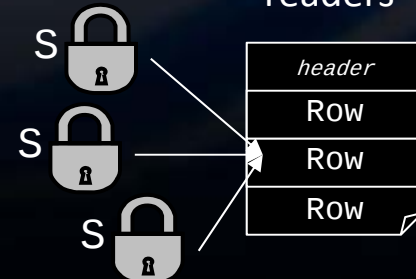


Then, an eXclusive lock is requested



eXclusive lock WAITS

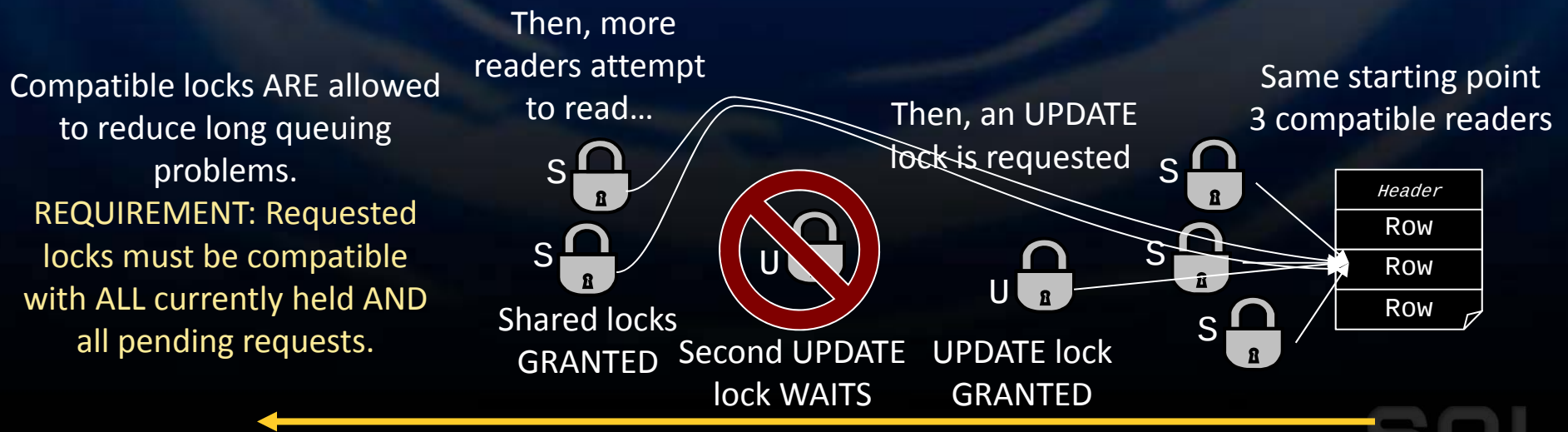
Imagine 3 compatible readers



Relaxed FIFO

To Reduce Really Long Blocking Chains

- ◆ <http://blogs.msdn.com/b/psssql/archive/2009/06/02/sql-server-lock-manager-and-relaxed-fifo.aspx> (<http://bit.ly/NWAgVC>)
- ◆ An update is incompatible with another update
- ◆ The requested (second) update will wait behind the current update...
- ◆ What about shared locks if ONLY update locks?
 - ◆ ALLOWED



Nasty Blocking CHAINS

Relaxed FIFO isn't Always Enough

- Imagine someone's running a NOLOCK query against a very large table (the query takes 4 minutes to run [ok, not *that* nasty])
- This query's running off-hours but the standard is to use NOLOCK
- An `sp_recompile tname` is requested

Unfortunately now...
everybody waits.....

Relaxed FIFO only let's through the
folks that are compatible with ALL
pending locks (not just those that
are currently held).

This can create terribly long blocking
chains.

