

SQLskills Immersion Event

IEPTO2: Performance Tuning and Optimization

Whiteboard Drawings

Kimberly L. Tripp
Kimberly@SQLskills.com



NOTE: Includes whiteboard drawings from prior IE courses
+ drawings done the week of Feb 23, 2015
Sydney, NSW, Australia – IEPTO2 Training

Plan Cache “Buckets / Entries” Limits

- It's UNLIKELY (especially with cache management / clearing) but for some systems, you can run into a second problem
 - When too many concurrent inserts occur in the same hash bucket or **the ad hoc SQL Server plan cache hits its entry limit of 160,036**, severe contention on SOS_CACHESTORE spinlock occurs. In this situation, a high CPU usage occurs in Microsoft SQL Server 2012 or SQL Server 2014.
- **Default number of buckets is 40,009**

```
SELECT [cc].[name], [cc].[type], [cc].[entries_count]
FROM [sys].[dm_os_memory_cache_counters] AS [cc]
WHERE [cc].[name] = 'SQL Plans'
```
- **SQL2012SP1CU12 & SQL2014CU6 -> Trace flags 174 & 8032 required to increase total buckets to 160,036**
- **See FIX: SOS_CACHESTORE spinlock contention on ad hoc SQL Server plan cache causes high CPU usage in SQL Server 2012 or 2014**
 - <http://support.microsoft.com/kb/3026083>

Modularization

- **Be careful of block statements/conditional logic**
 - SQL Server optimizes the process of optimization and this may lead to an less-than-optimal plan (no, really!)
- **Breaking the stored procedure into smaller chunks**
 - Can handle the block of statements on a more granular basis
 - Subprocedures can be optimized/compiled
 - AVOID: CREATE subprocedure WITH RECOMPILE
 - CONSIDER: EXECUTE subprocedure WITH RECOMPILE
 - BEST: Statement-level execution options
 - More options for control
 - Better monitoring / tracking: these **can** be tracked in the DMVs through sys.dm_exec_query_stats

SQLskills Immersion Event

IEPTO2: Performance Tuning and Optimization

Module 9: Statement Execution, Stored Procedures, and the Plan Cache

Kimberly L. Tripp

Kimberly@SQLskills.com



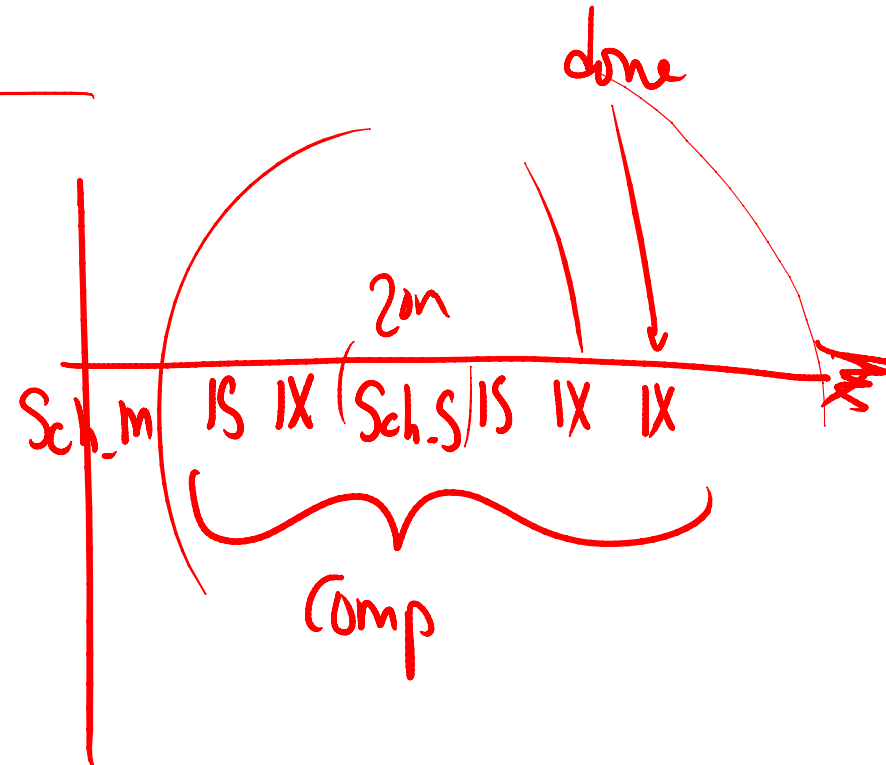
Sch_M on table

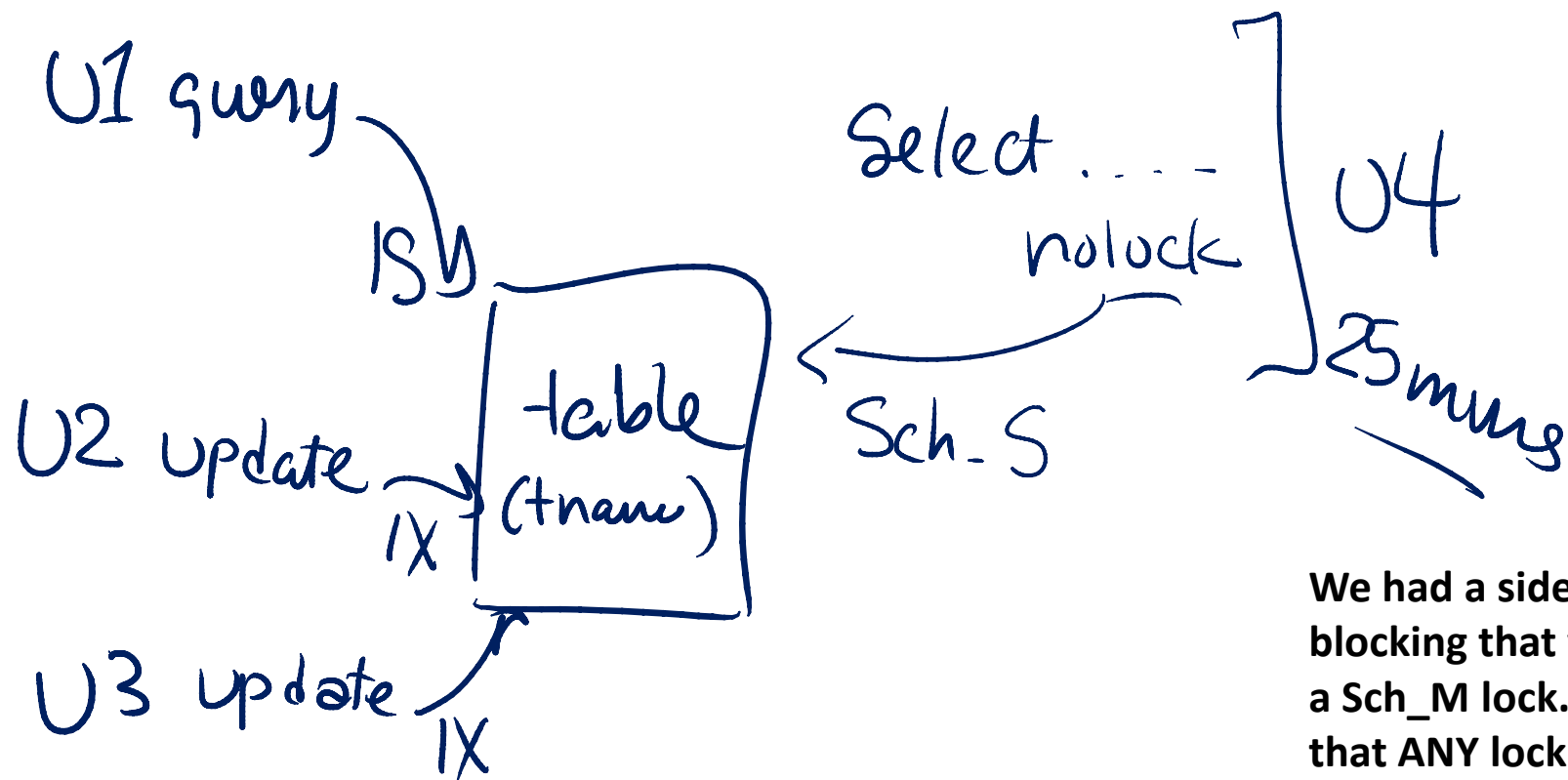
Sp_recompile
tname

NOLOCK longrunning
Sch_S
20 min

We had a side discussion on the blocking that you could have with a Sch_M lock. The key point is that ANY lock blocks a SCH_M lock but a SCH_S held from a NOLOCK query that's extremely long running – will cause nasty secondary blocking while the SCH_M waits for ALL locks to drain off (where the SCH_S lock is probably the longest running)

The next slide is from IE1.



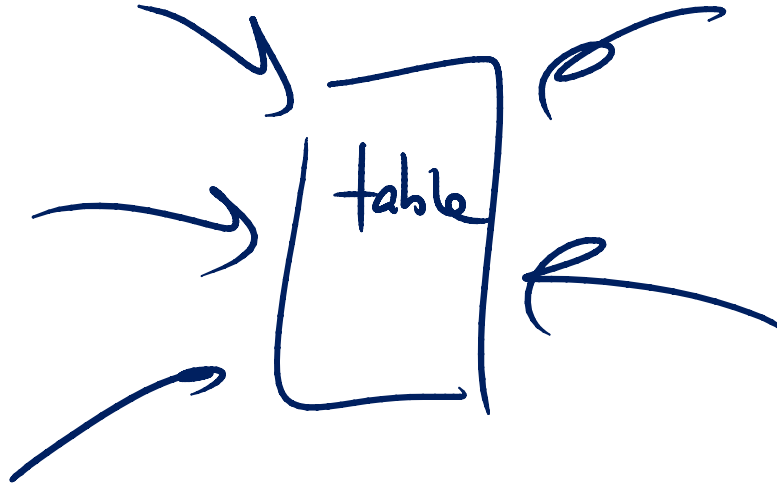


We had a side discussion on the blocking that you could have with a Sch_M lock. The key point is that ANY lock blocks a SCH_M lock but a SCH_S held from a NOLOCK query that's extremely long running – will cause nasty secondary blocking while the SCH_M waits for ALL locks to drain off (where the SCH_S lock is probably the longest running) The next slide talks about my script (PreventLongBlockingChains.sql). The following slide is a graphic from IEPTO1.

sp_recompile tname
WANTS Sch_M

Every other request WANTS

Be sure to check out the script:
PreventLongBlockingChains.sql



WANT sp_recompile frame

any long running statements?
(configurable)
30 sec

if not

then get in queue

if I can't get lock in 5 sec

Get out

Nasty Blocking CHAINS

Relaxed FIFO isn't Always Enough

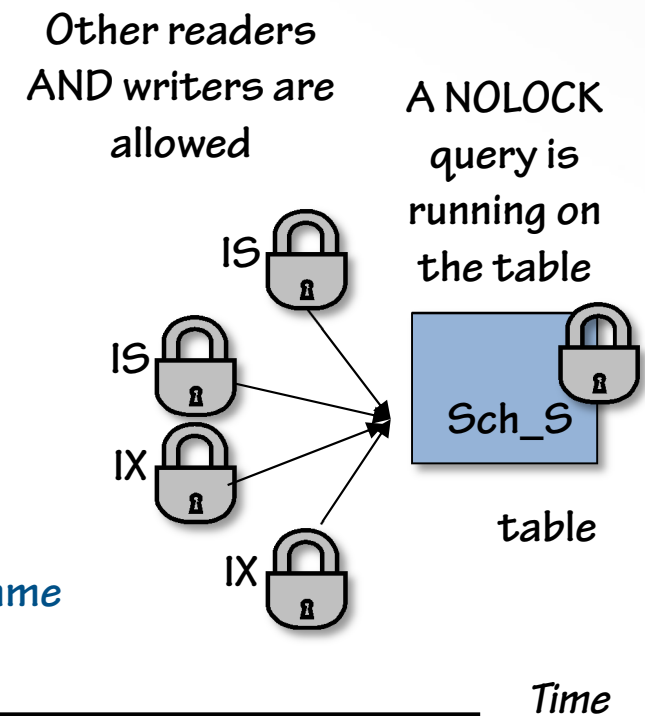
We had a side discussion on the blocking that you could have with a Sch_M lock. This slide is from IEPTO1.

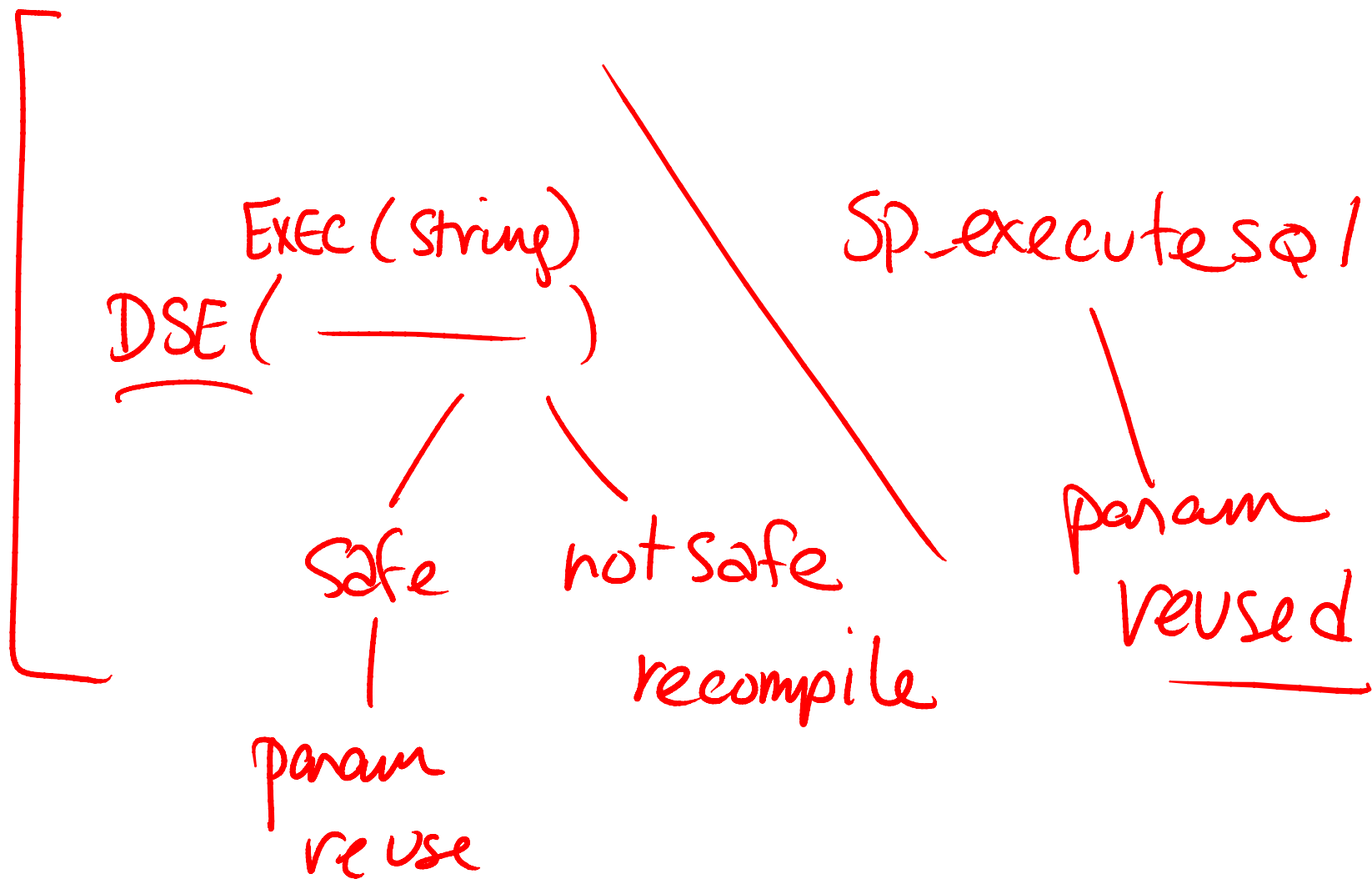
- Imagine someone's running a NOLOCK query against a very large table (the query takes 4 minutes to run [ok, not that nasty])
- This query's running off-hours but the standard is to use NOLOCK
- An `sp_recompile tname` is requested

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.





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.

OSFA Dialog (An attempt at *One Size Fits All* Programming)

where

(CustID = @CustID
or custID
is NULL)

CustID
LName FName
SSN
City State

AND (LN = @LN or @LN is NULL)

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)
```

...

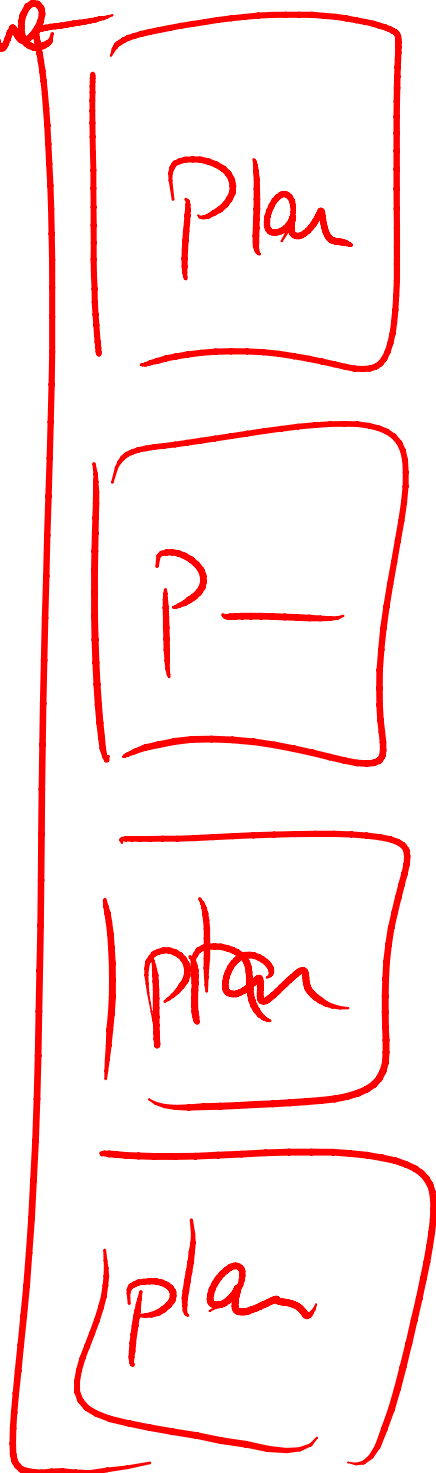
Check out the demo scripts in order to fully see how to better optimize this – using sp_executesql (for combinations that are stable) and sp_executesql WITH RECOMPILE for combinations that could generate different plans.

men

BAD

Horrible

brisque



Opt
ad
hoc

too
much
to
look for
Plan
Stub



limited
at 1^{or}2
GB
Clean

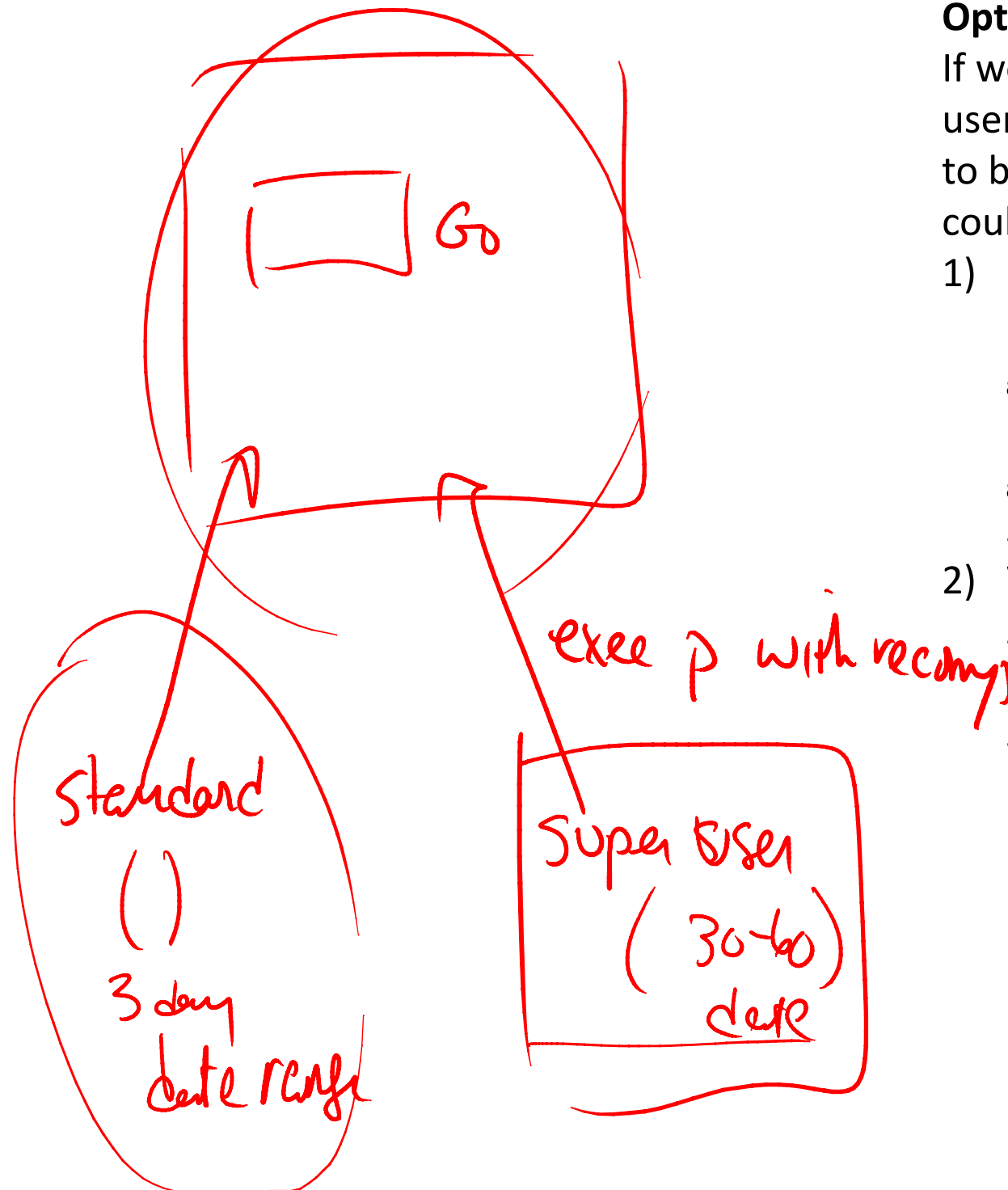
If you turn on "optimize for ad hoc workloads" then the number of "stubs" that can fit in cache is much higher.

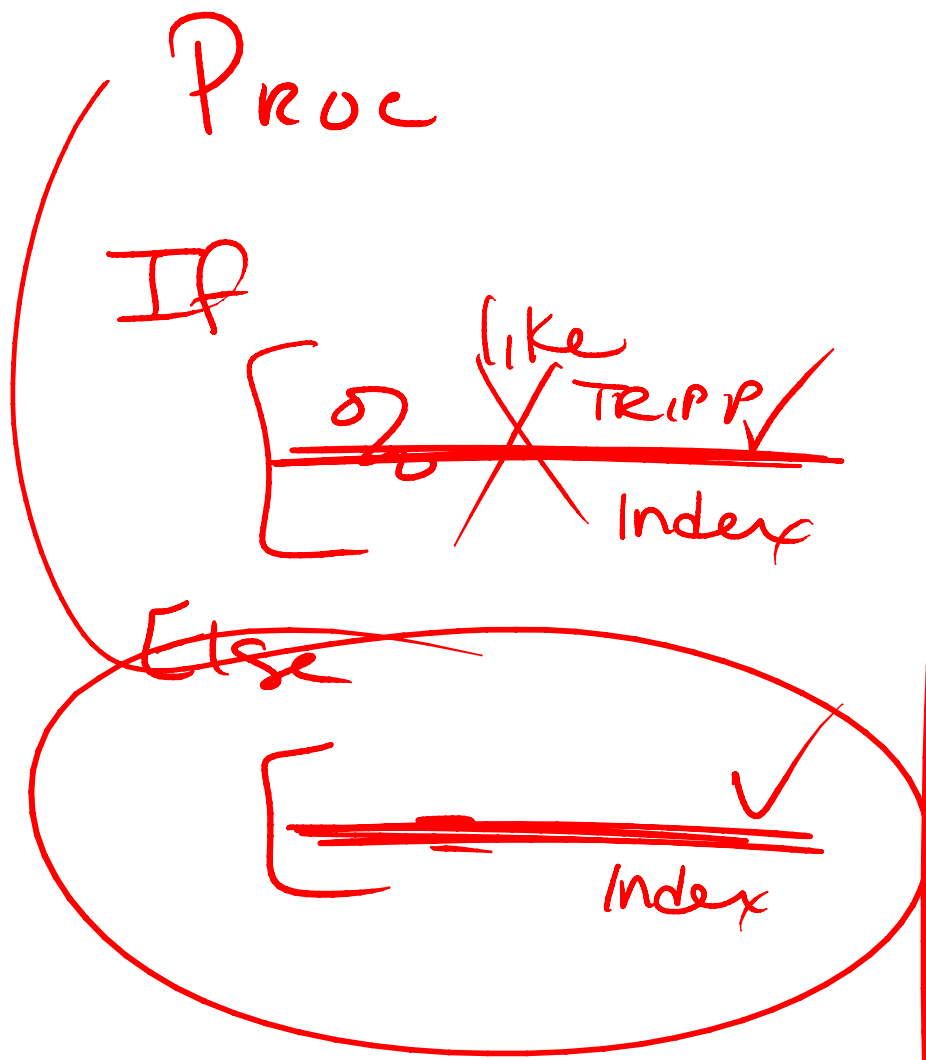
At that point, searching to find whether or not the statement has executed is harder... this is why you want to make sure that you don't let this cache get really large (clearing it at 1 or 2GB).

Options for different executions

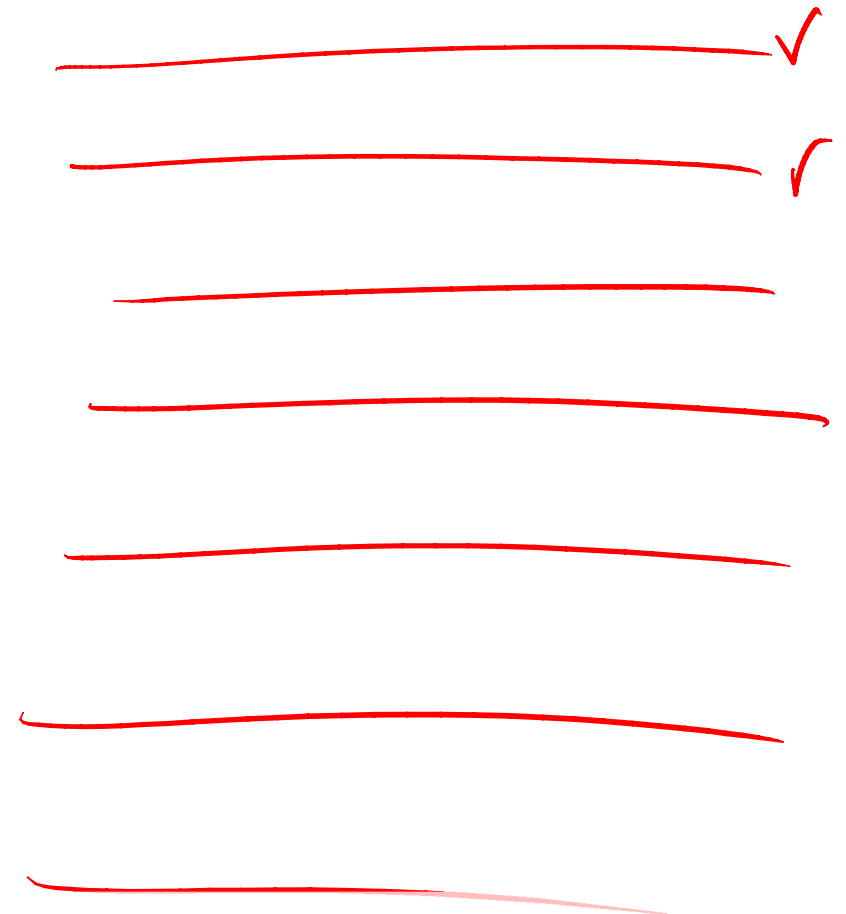
If we largely execute with “standard” users and we absolutely want THAT plan to be in cache and to be re-used then we could:

- 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)



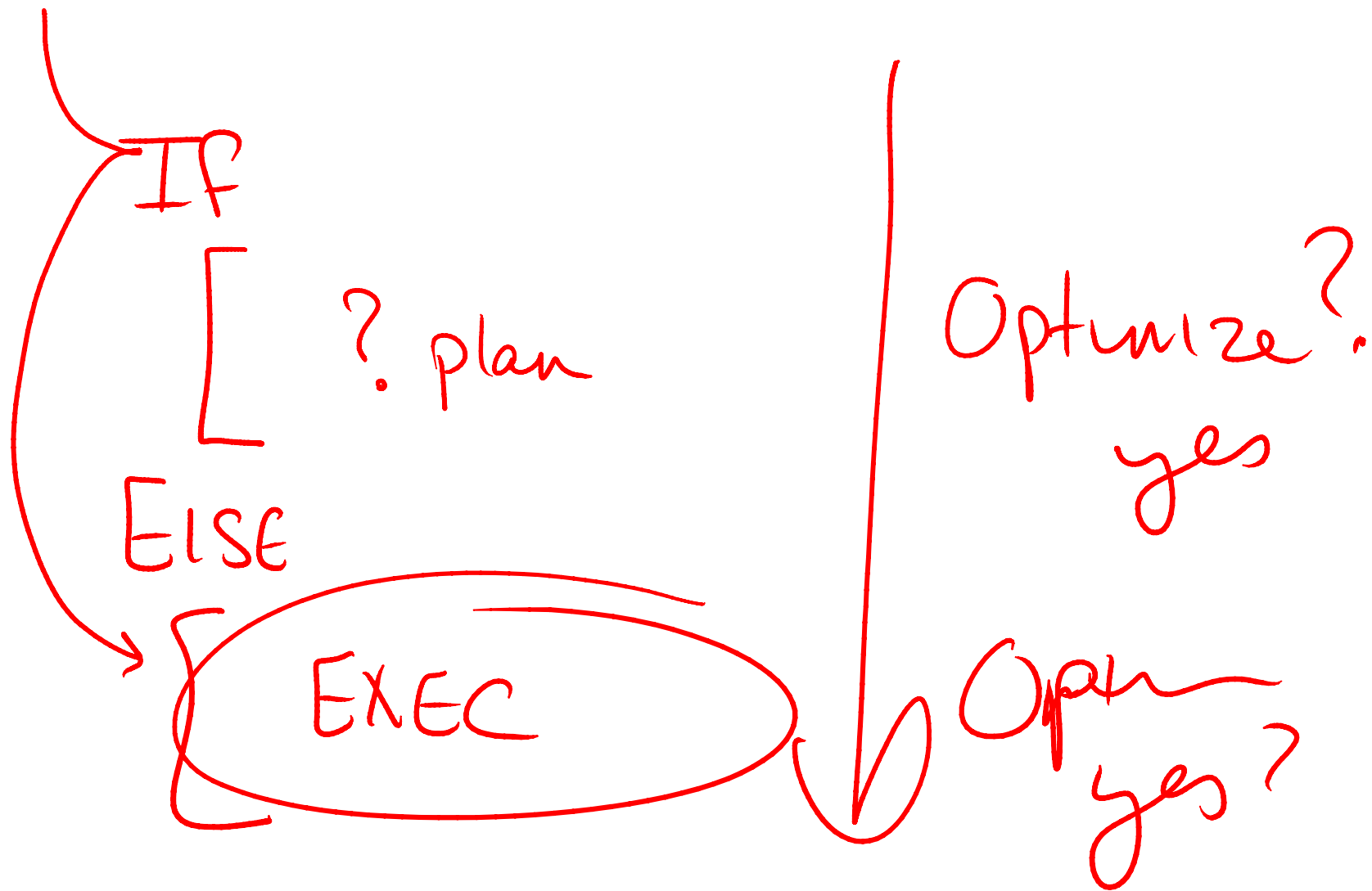


Proc



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.



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.

CR PROC —
(param1
 pa 2
 —)

} Sniffed

as

Declare var — unknown why?
 we don't know
 until EXECUTION

Select -- -- param?

Select -- -- param?

Parameters are “sniffed”

SQL Server uses parameters to optimize. Sniffing implies that SQL Server uses the histogram to evaluate the data selectivity. For the FIRST execution, parameter sniffing is great. It's the subsequent executions that can suffer from parameter sniffing (and therefore end up with PSP = parameter sniffing problems).

Variables are “unknown”

Variables are only known at runtime as the statements execute and the variables are assigned. As a result, they are unknown during optimization. Without an actual value, SQL Server cannot use the histogram. Instead SQL Server uses the density vector for the “average” numbers of rows that meet that criteria.

PROC

param list

@p1 value

@p2 value

SQL → uses a variable unknown
D.V.

SQL → uses a literal Known → special features

SQL → uses a parameter — histogram sniffing
(no special features)

If ~~~~~

[SQL → uses param — hist. sniffing

Else

[SQL → uses param — hist sniffing

SQL Server optimizes everything that's optimizable (sp?) ☺

Again, parameters are known. Parameters can be sniffed. SQL Server will optimize every statement that it can – regardless of what actually ends up executing. At the time of optimization they do not know what will or will not execute and they do not know the value of a variable.

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.

Dynamic String Execution and Execute As

I have a love/hate relationship with EXECUTE AS. On the one hand, you can give someone rights to something that you wouldn't otherwise be able to give...

However, be careful giving elevated privs around DSE. Generally, my recommendation around DSE is to only create a login-less user with very low privs.

fname

SPROC

EXEC(~~~~~)
Delete... fname

Executes As?

CALLER

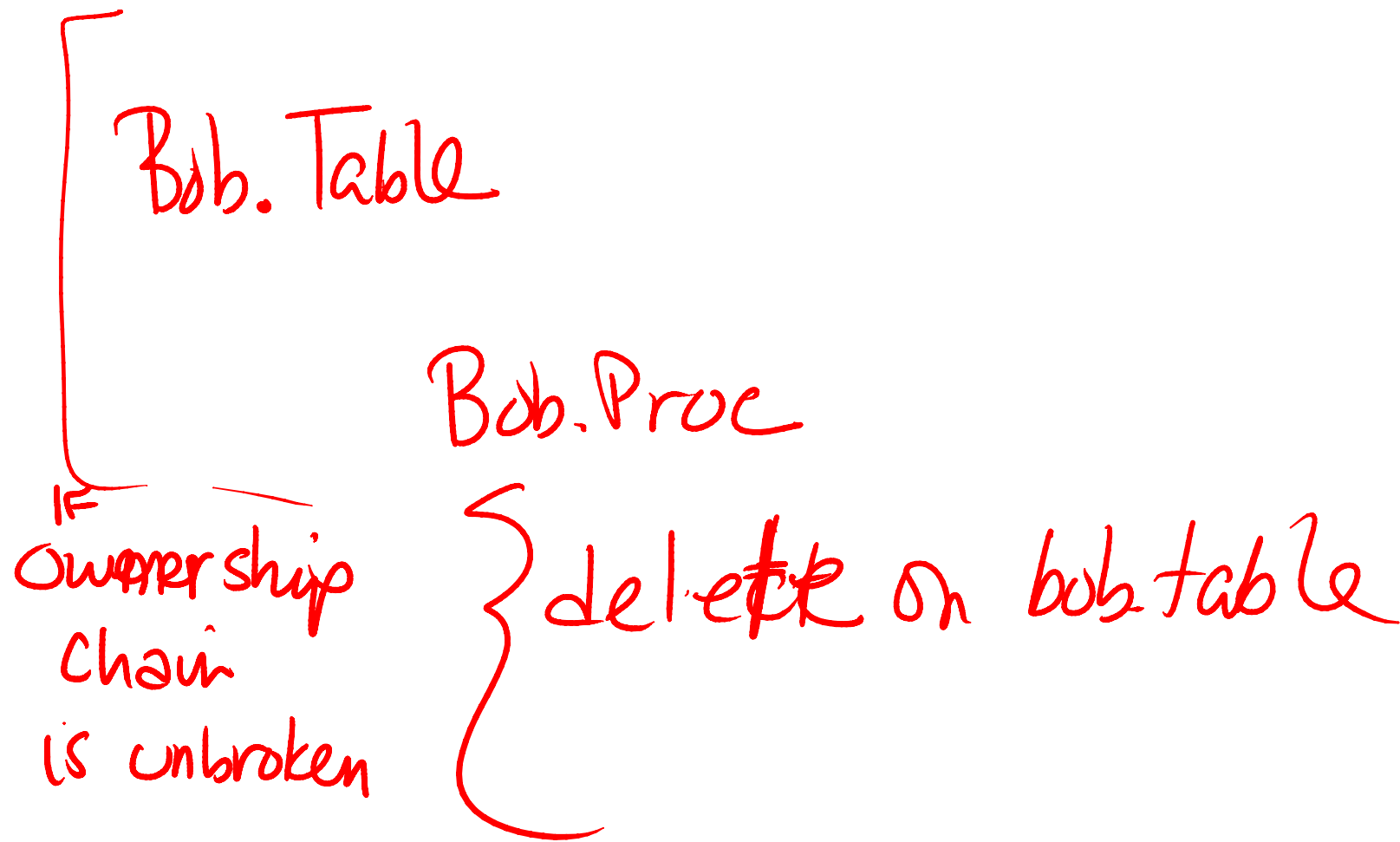
USER

~~Higher~~

low

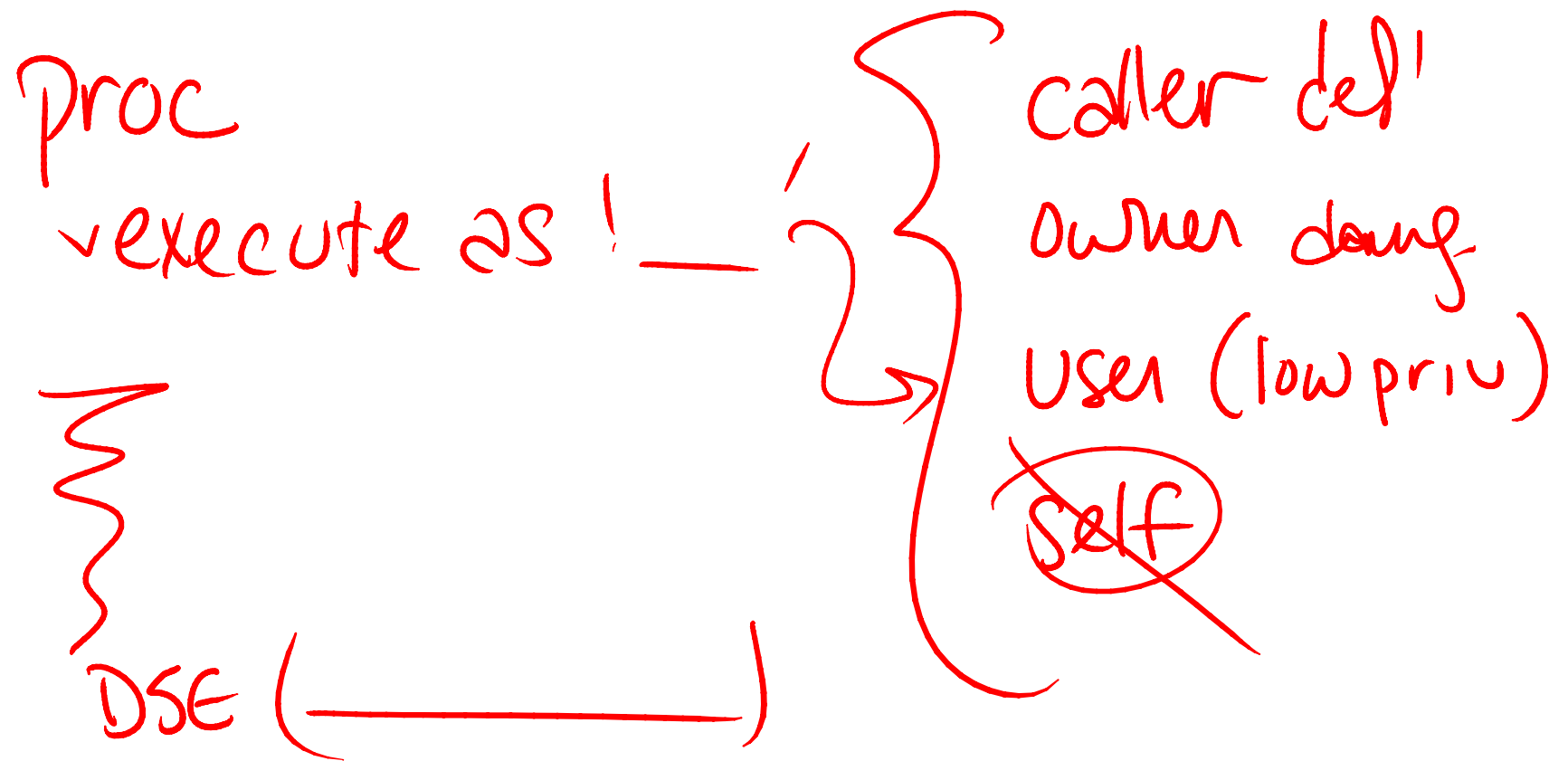
~~owner~~

~~self~~ stupid



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 \(for auditing or prevention\)](#)

✓ fn .
✓ (fn + ln) ?

Stable plans vs. unstable plans – how do you know?

- You could test different combinations using execute with recompile.
- You might know because of selectivity?

* ~~fn + ln + mno~~ ← stable/cons
✓ ln

* ~~ln + mno~~ ← }
* ~~mno~~ ← } ← stable/cons
* ~~mno + fn~~ ← }

Options that better balancing recompiling too much vs. not recompiling enough

This is from the multipurpose procedure demo. There are 3 parameters and 7 possible combinations that can be submitted. Instead of adding `OPTION (RECOMPILE)` to the statement (which doesn't always work [remember, it has a very checkered past]) you can build the statement dynamically and then programmatically (by setting a `RECOMPILE` flag to 0 or 1) decide whether or not `OPTION(RECOMPILE)` should be added to the dynamically constructed statement. Then, you can use `sp_executesql` to place ALL seven statements in cache. Those without the `OPTION(RECOMPILE)` clause will be cached. Those with `OPTION (RECOMPILE)` will get a plan on each execution.

15 executions?

15 plans?

2 plans

n 6/5

no good /
Single plan

OPTION
(RECOMPILE)

DSE ?

8/7

IF Subproc
ELSE Subproc

14

1 Sep proc
recompile only work if
you did not
OPTION opt for

How many plans COULD be generated?

When you're testing your procedures you might have 15 test cases. Executing each of the procedures (with the 15 different combinations of parameters) WITH RECOMPILE. Then, review the number of different plans

If there are 15 different plans for 15 different executions – Use OPTION (RECOMPILE)

If there are 2 primary plans that are fairly split – see if you can create subprocedures and then control the behavior that way (especially if they either don't need to recompile OR only one needs to recompile).

If there are only 2 plans where one is 14 of the executions (and, is the norm) then you might use OPTION (OPTIMIZE FOR ...) but it depends on how bad the performance is the for 15th. Can you control that through [other] parameters or programmatically?

① recompile always

- more CPU

+ Plan spec. to those param

Default

PSP

+ less CPU

- possibly horrible plans

- not consistent (who gets there first)

UNKNOWN (Avg)

- Bad plan? for atypical values

How many plans COULD be generated?

This is another way to describe what was on the prior page...

These are the pros/cons of the different procedure (and statements within) strategies:

The default

- Prone to parameter sniffing problems
- Possibly HORRIBLE plans for subsequent executions
- No guarantee of what plan actually gets into cache
- + Less CPU (not always though – you do save compilation time but if the execution is really expensive then you might lose all of the gains of having compiled/saved a plan)

Forcing a recompile for EVERY execution

- More CPU
- + Plan specific to those parameters

Optimize for UNKNOWN

- Guarantees an AVERAGE plan
- + Works well when the data is evenly distributed (but, the problems are less likely as well – but, if plans were getting into cache from atypical values then this might solve the problem)

SQLskills Immersion Event

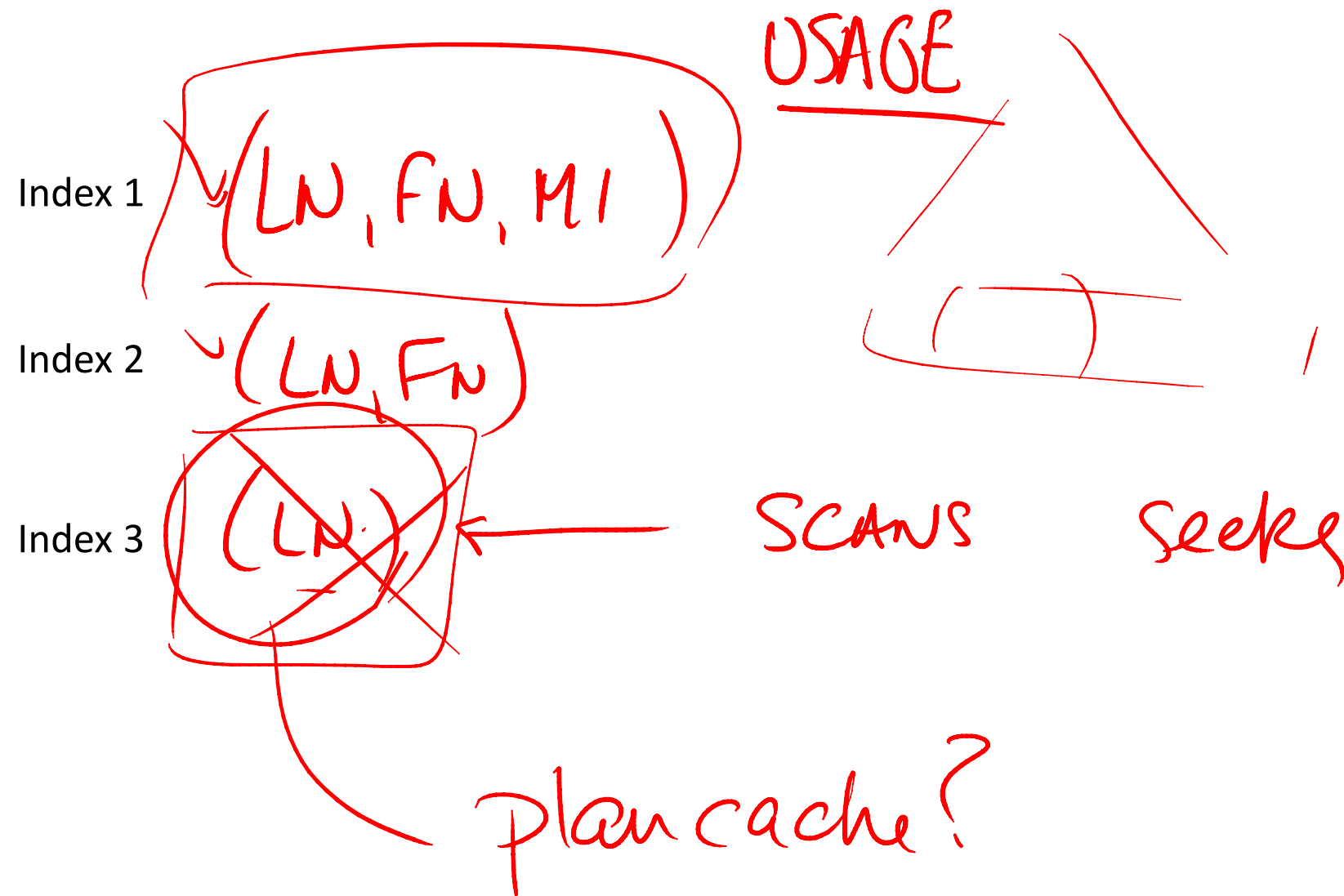
IEPTO2: Performance Tuning and Optimization

Module 10: Index Analysis

Kimberly L. Tripp

Kimberly@SQLskills.com

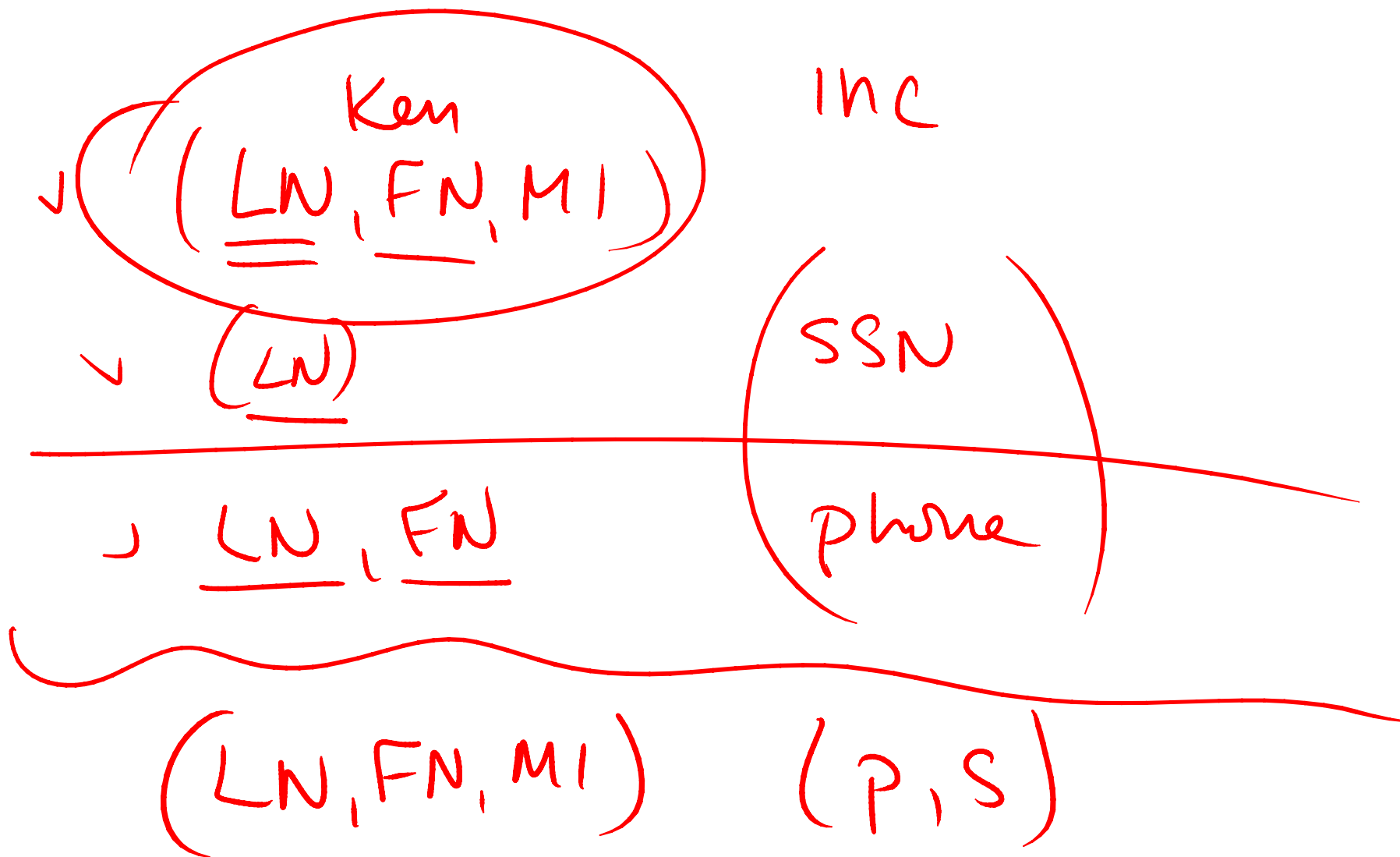




Should you drop redundant indexes? It depends. 😊

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 consolidation

No production environment can support THE BEST index for every query, you'd have too many indexes. To be really successful with indexing – you need to find a balance. This includes consolidating some of your index recommendations to have one index handle multiple purposes!

Key Inc

$C1 = C2$ ~~$C3$~~

$C1, C2, C3$ no inc

Key = navigation

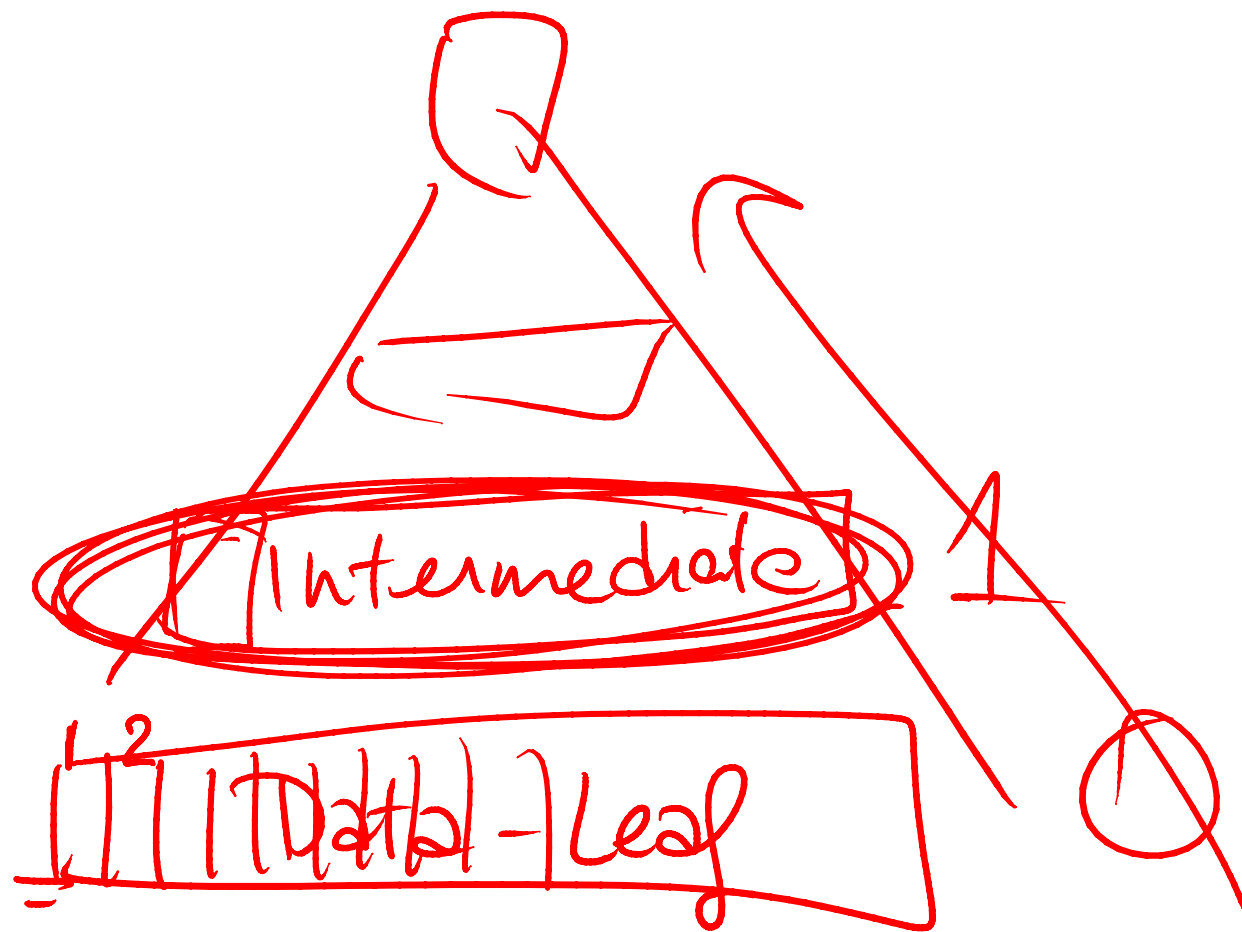
Key must be pres.

Key Inc

<u>LN, FN</u>	
<u>LN</u>	
<u>LN, FN, MI</u>	
LN	SSN
//	
<u>LN, FN, MI</u>	(^{inc} SSN)

Consolidation?

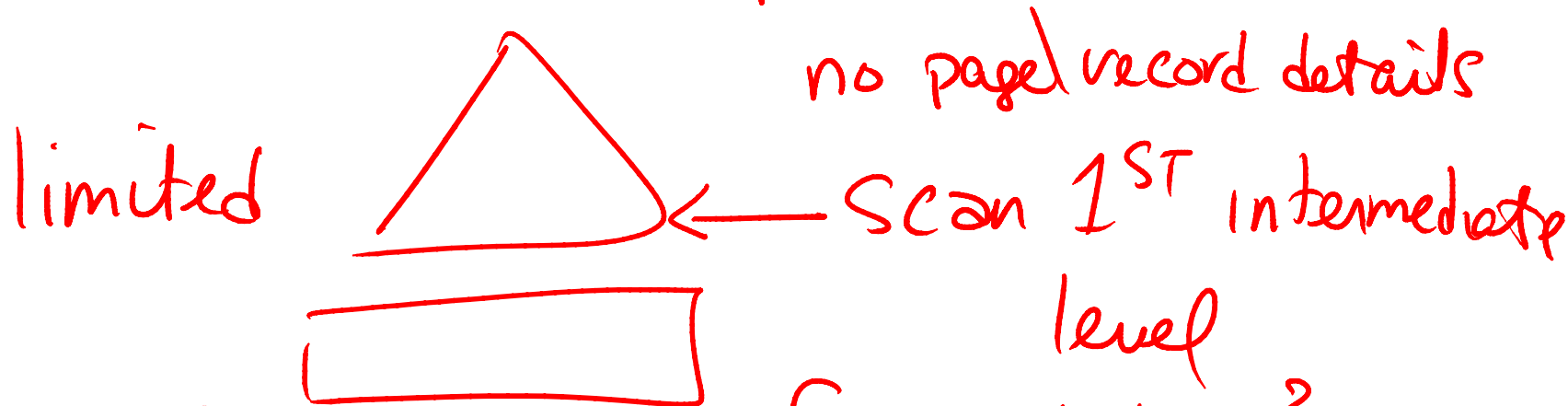
You can CONSIDER removing indexes that are left-based subsets of others (in terms of the key). However, be sure to preserve the key as that's used for navigation/seeking. The columns that are included can be combined in any order.



Index Health – Checking for fragmentation

To get the value: **avg_fragmentation_in_percent** for **sys.dm_db_index_physical_stats**, SQL Server uses a scan of the first intermediate level (index level 1). All of the modes (limited, sampled, and detailed), use this same method. Then, SAMPLED and DETAILED do additional work (on the leaf level of the index) to give you things like average page density, etc. If your analysis/rebuild scripts only use **avg_fragmentation_in_percent** to determine fragmentation then you should only do a LIMITED mode scan. SAMPLED and DETAILED offer more page-specific details (page density, forwarding pointers, etc.) and if you're NOT using those during your automated maintenance process (and most people aren't) don't waste time!

Sys. dm-db-index-physical-stats



Good for avg-fragmentation-in-percent fragmentation?

Sampled } Avg page density, page fullness
detailed } row size

→ 1/100 of leaf / detailed reads all pages
all levels

FN like K%

rno > 6

mno < 5000

fn, mno, rno

Katie 6 4

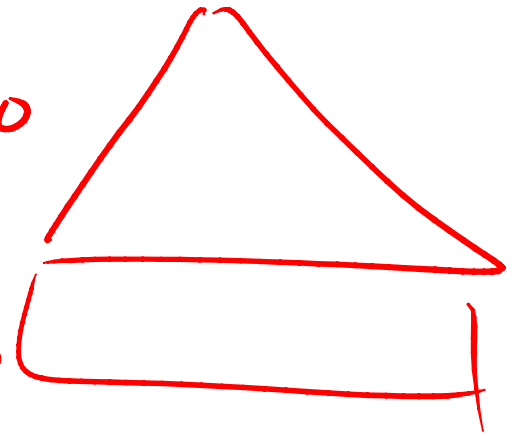
Katie 7892 6

Kiera

Kiera

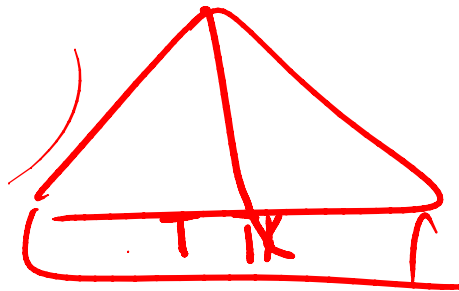
Kimberly

Kimberly

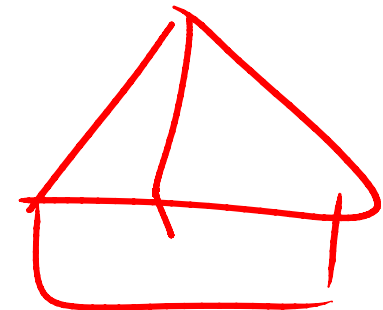


FN = Kimberly

LN = TRIPP



LN, FN



FN, LN

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.