NOTE: Whiteboard drawings from our discussions during the workshop at SQLintersection on Sunday, April 13, 2014.

PS: I also included a few other relevant drawings from other workshops.

SQLintersection

Precon 05

Queries Gone Wild: Real-world Solutions

Whiteboard Drawings and Annotations

Kimberly L. Tripp
President/Founder/Co-owner
Kimberly@SQLskills.com

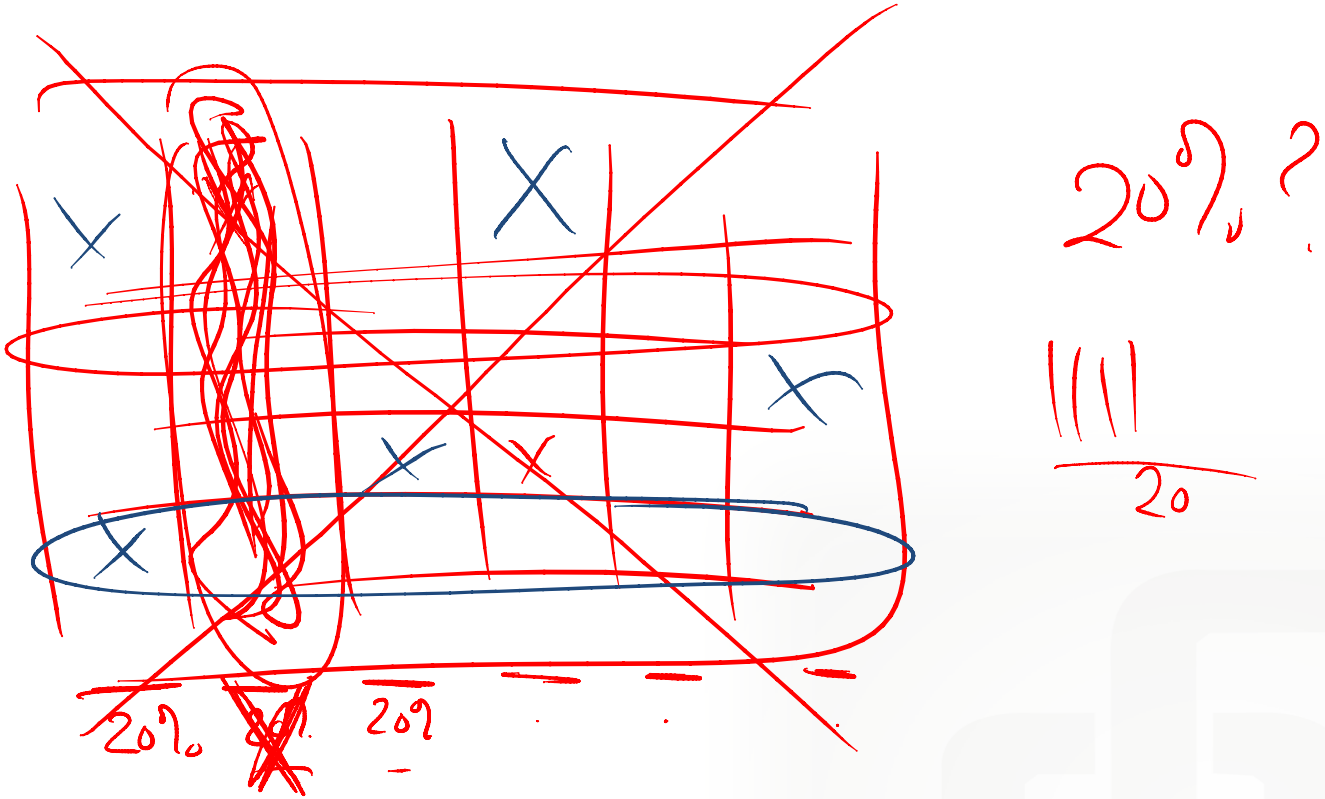
SQLskills
immerse yourself in sql server

SQL intersection

Auto Update Statistics

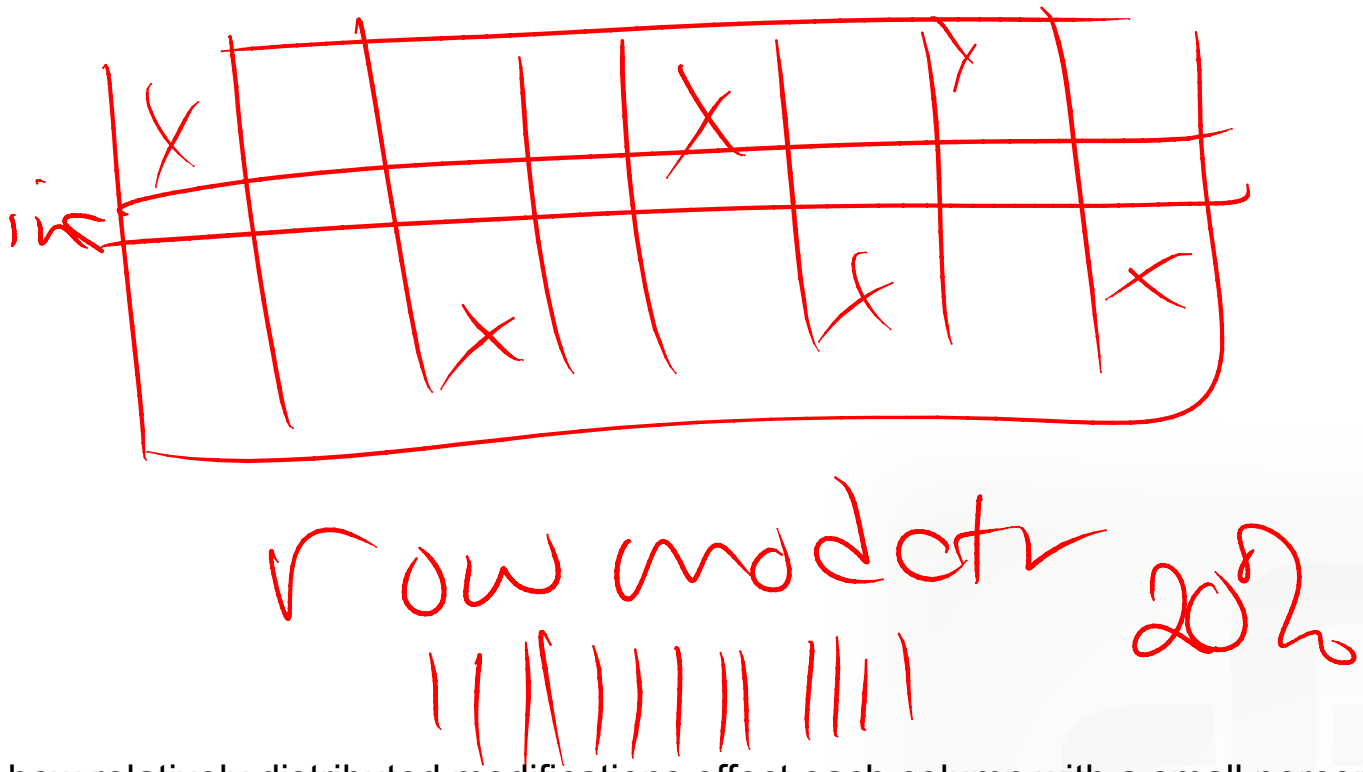
Following drawings apply to slide 21 from our workshop deck

- **SQL Server 7.0**
  - Invalidated when sysindexes.rowmodctr reached
  - Updated when invalidated
- **SQL Server 2000**
  - Invalidated when sysindexes.rowmodctr reached
  - ✓ Updated when needed
- **SQL Server 2005**
  - ✓ Invalidated when sysrowsetcolumns.rcmodified reached
  - Updated when needed
- **SQL Server 2008**
  - Invalidated when sysrscols.rcmodified reached
  - Updated when needed



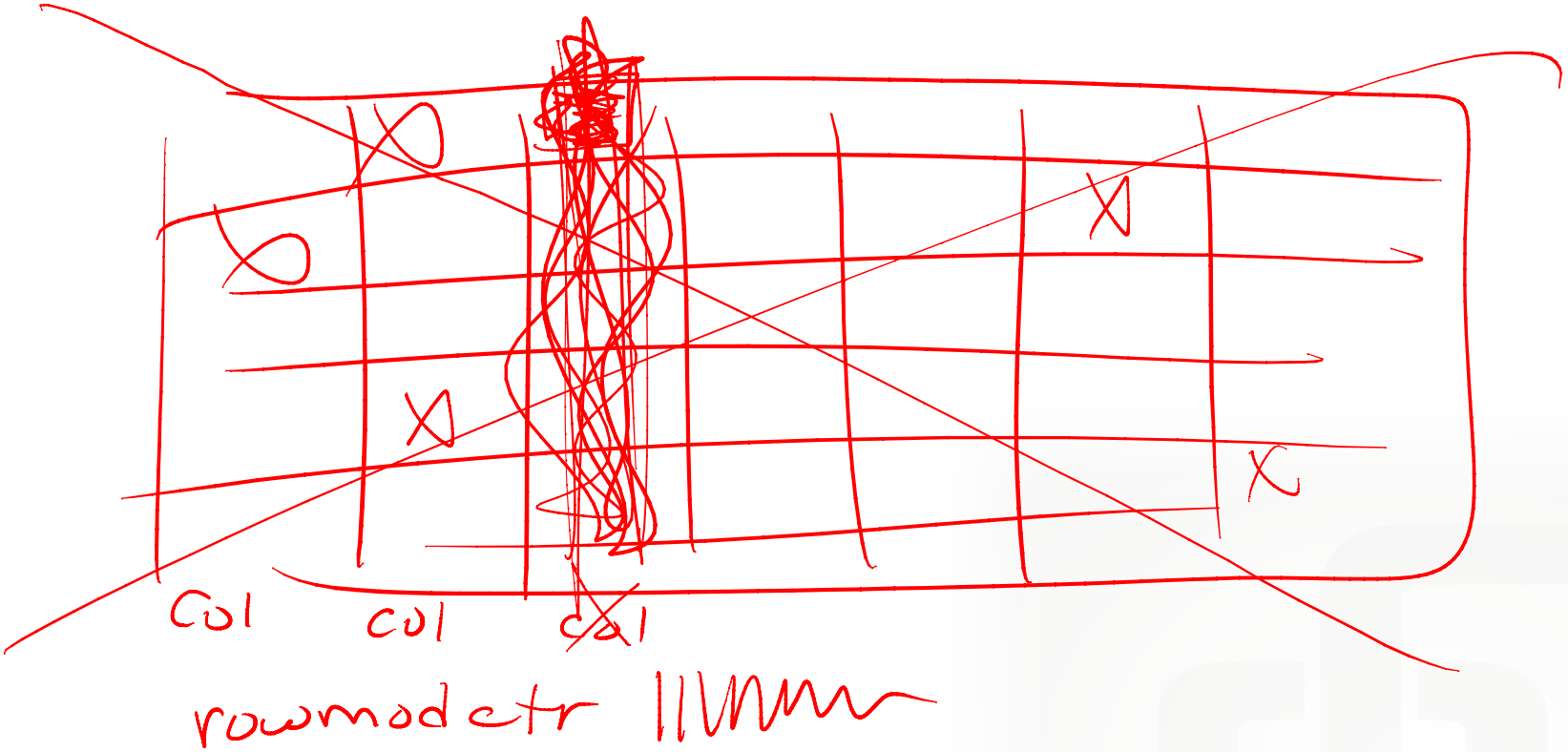
This is a diagram showing the pros/cons of the methods for statistics invalidation. SQL Server 7.0 and 2000 used a rowmodctr (sysindexes.rowmodctr) to do invalidation. Even though SQL Server doesn't use this anymore – you can (through the new sys.dm_db_stats_properties DMV). In 2005+ they moved to an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

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

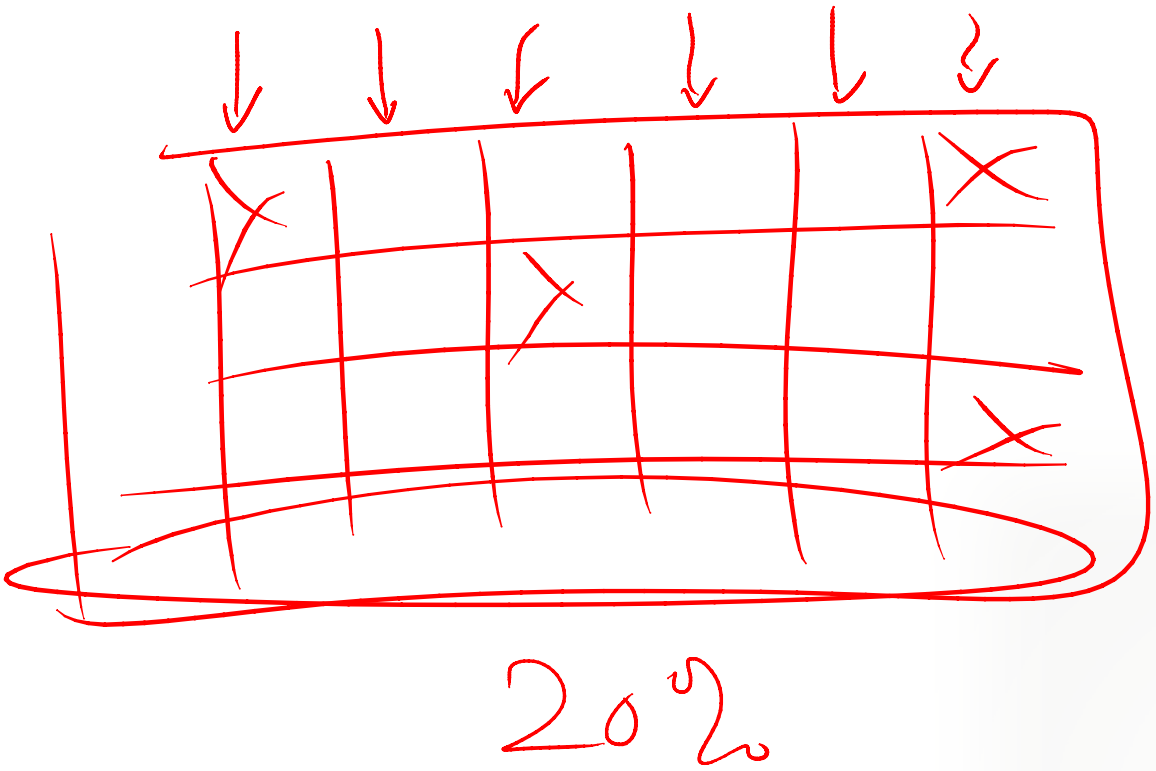


This shows how relatively distributed modifications effect each column with a small percentage of the modifications but when they add up to 20% ALL columns become invalidated (this was the PRE-2005 way of doing it):

- 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).



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



In 2005+ they use an internally defined colmodctr:

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

Even Distribution: Always Even??

Following drawing applies to slide 53 from our workshop deck

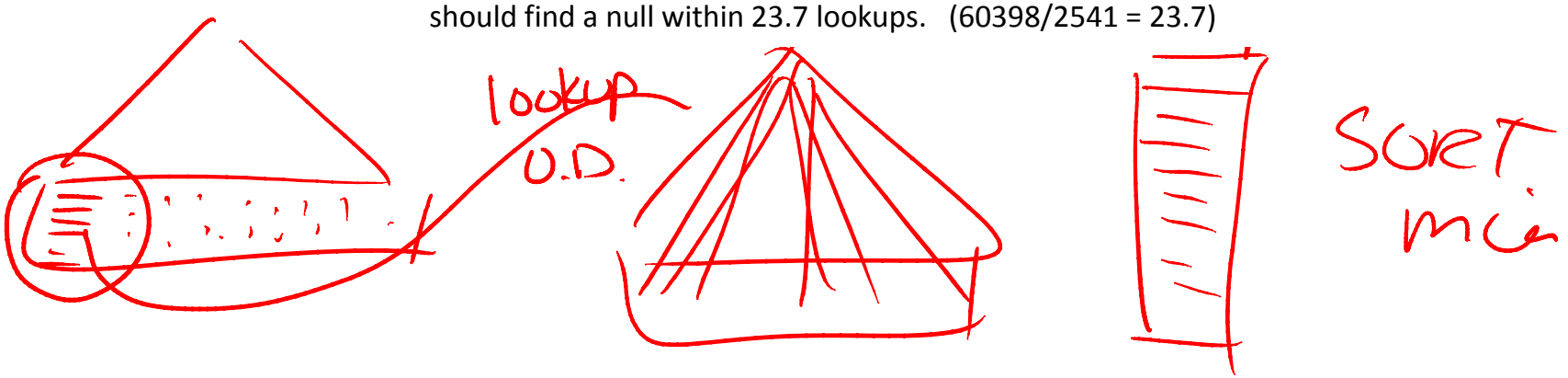
- **Table scan is always an option**
- **Use an index on ShipDateKey to look up the actual date for all orders where ShipDateKey is NULL**
  - This means a bookmark lookup must be run for EVERY NULL so that we can get the OrderDateKey
  - The worktable then needs to be sorted to find the lowest order date
- **Use an index on OrderDateKey as only 1 on 23.7 sales have a NULL for ShipDateKey**
  - SQL Server estimates that they'll find a NULL within 23.7 rows and they won't need a worktable
  - This sounds better…
- **But the rows are not evenly distributed!**

DEMO

(1) TS **FactInternetSales2**

This was the demo on uneven distribution.
(1) A table scan is always an option.
NOTE: With narrow indexes, SQL Server does not understand the correlation between these columns. As a result, estimates are going to expect even distribution.
(2) Index on ShipDateKey – requires looking up the OrderDateKey values and placing them in a temptable. Then, to find min – we have to sort.
(3) Index on OrderDateKey – requires looking up the shipdatekey until we find the first NULL (because that would be the min). How many lookups? If 2541 are null (out of 60398) then we should find a null within 23.7 lookups. (60398/2541 = 23.7)

NC (2) **Index on ShipDateKey**

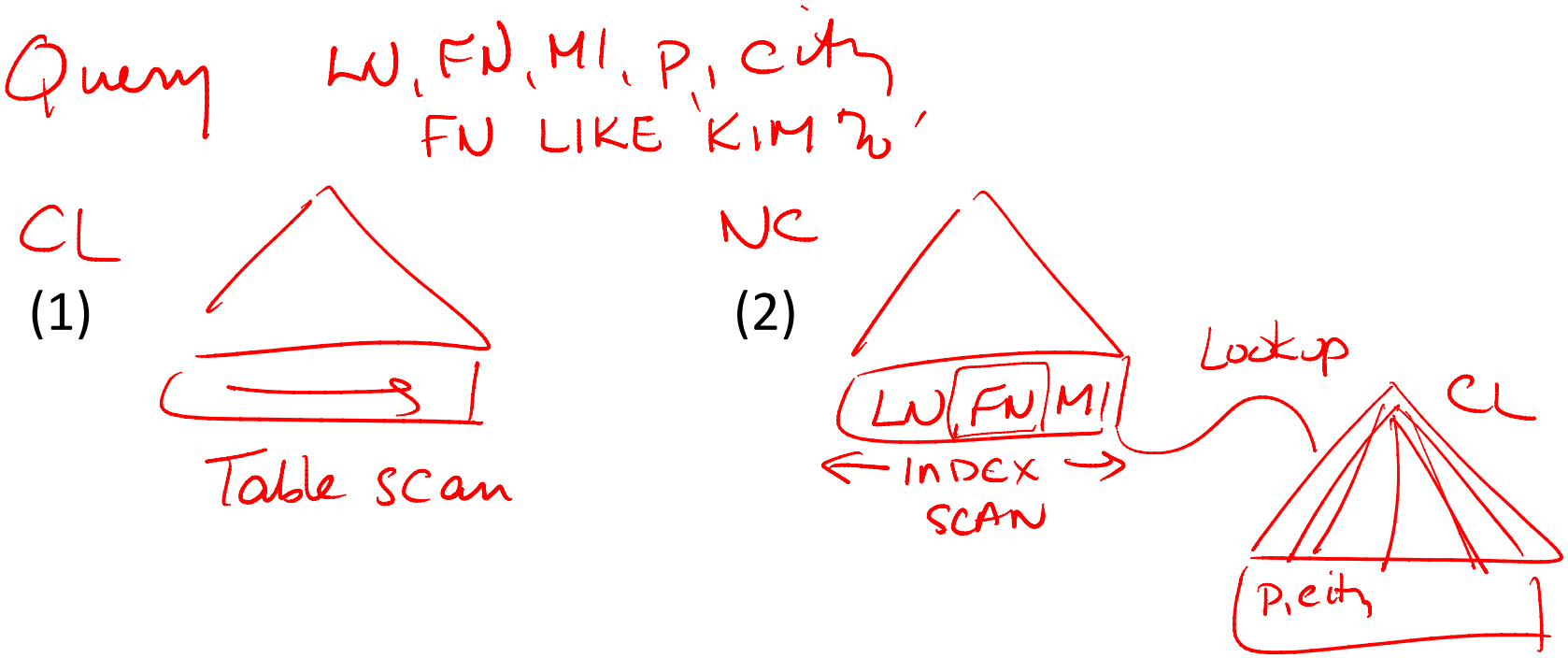


NC. **Index on OrderDateKey**

(3)

when will I hit a NULL

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



DEMO

This was the demo/discussion on how SQL Server uses nonclustered indexes to scan (when there isn't a better index AND the data is selective enough to make it worthwhile).

(1) A table scan is always an option. And, we'll always (easily) know the cost.

NOTE: No index exists for seeking by firstname so – what do we do?

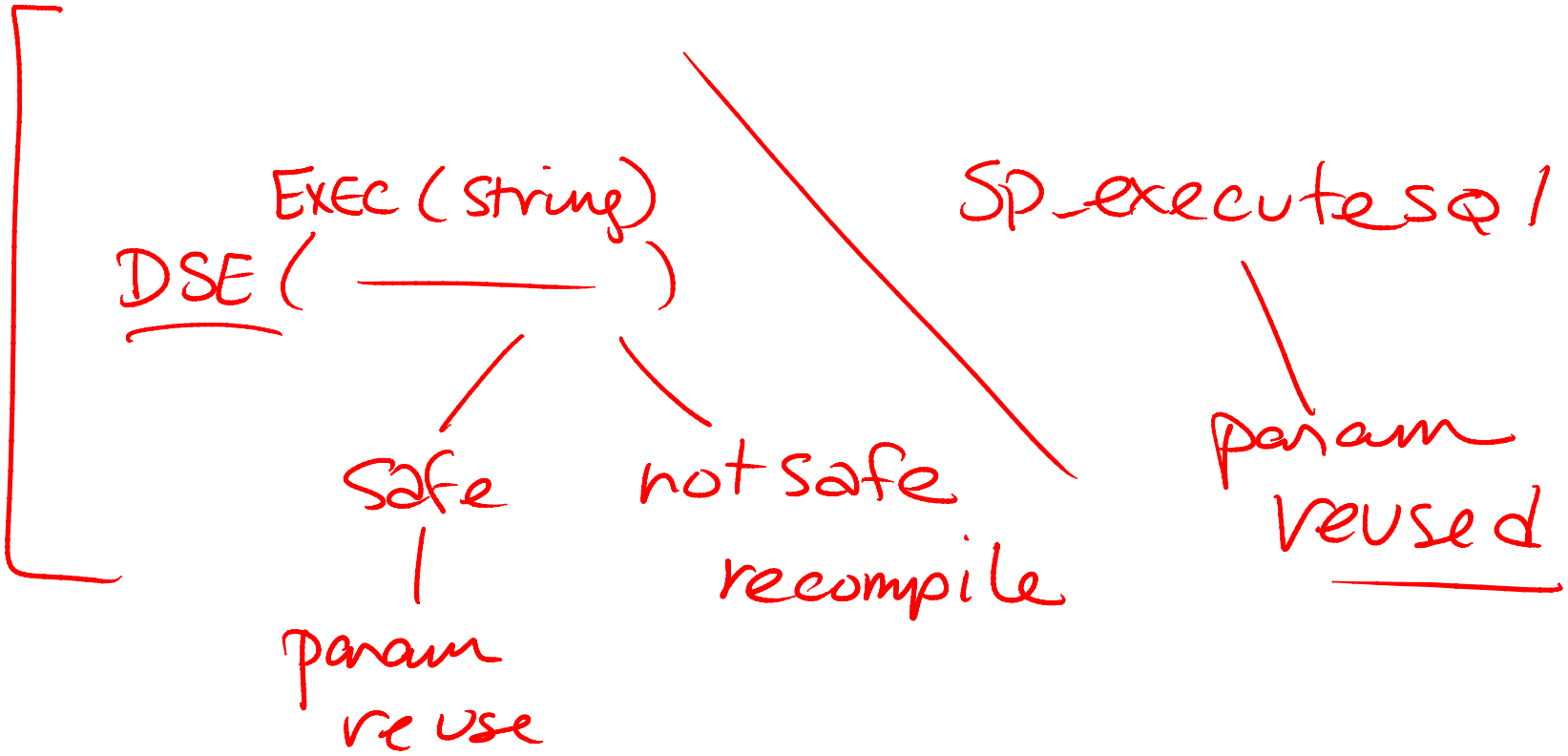
(2) Index on LN, FN, MI can be scanned (fewer I/Os than a table scan) but, we'll still need to do bookmark lookups to get the rest of the data. But, if there are only a few rows then it might be worth it… how would we know how many rows? Statistics on firstname would tell us! So, SQL Server auto creates statistics on firstname, optimizes the query, and executes!

if not too many rows?

Workshop Overview

Following drawings apply to our last section of the workshop – Optimizing Procedural Code (starting on slide 66)

- Query Optimization
- Statistics
- Data Distribution
- Statement Execution and Plan Caching
- **Stored Procedures**
  - ☐ Creation
  - ☐ Optimization
  - ☐ Plan invalidation
  - ☐ Recompilation
  - ☐ Optimization strategies

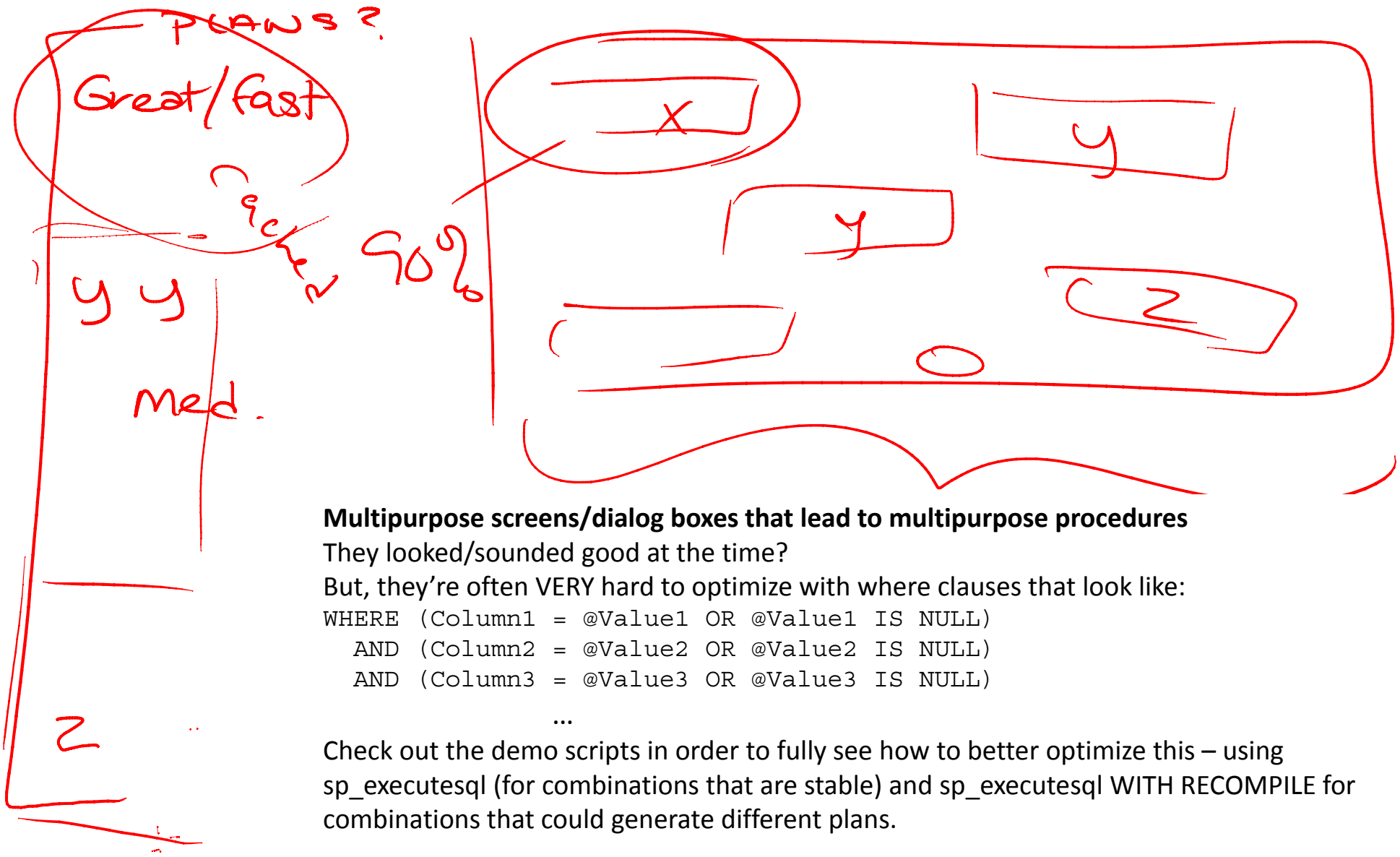


Using EXEC (string) [DSE] and/or sp_ExecuteSql [ES] inside of stored procedures

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.



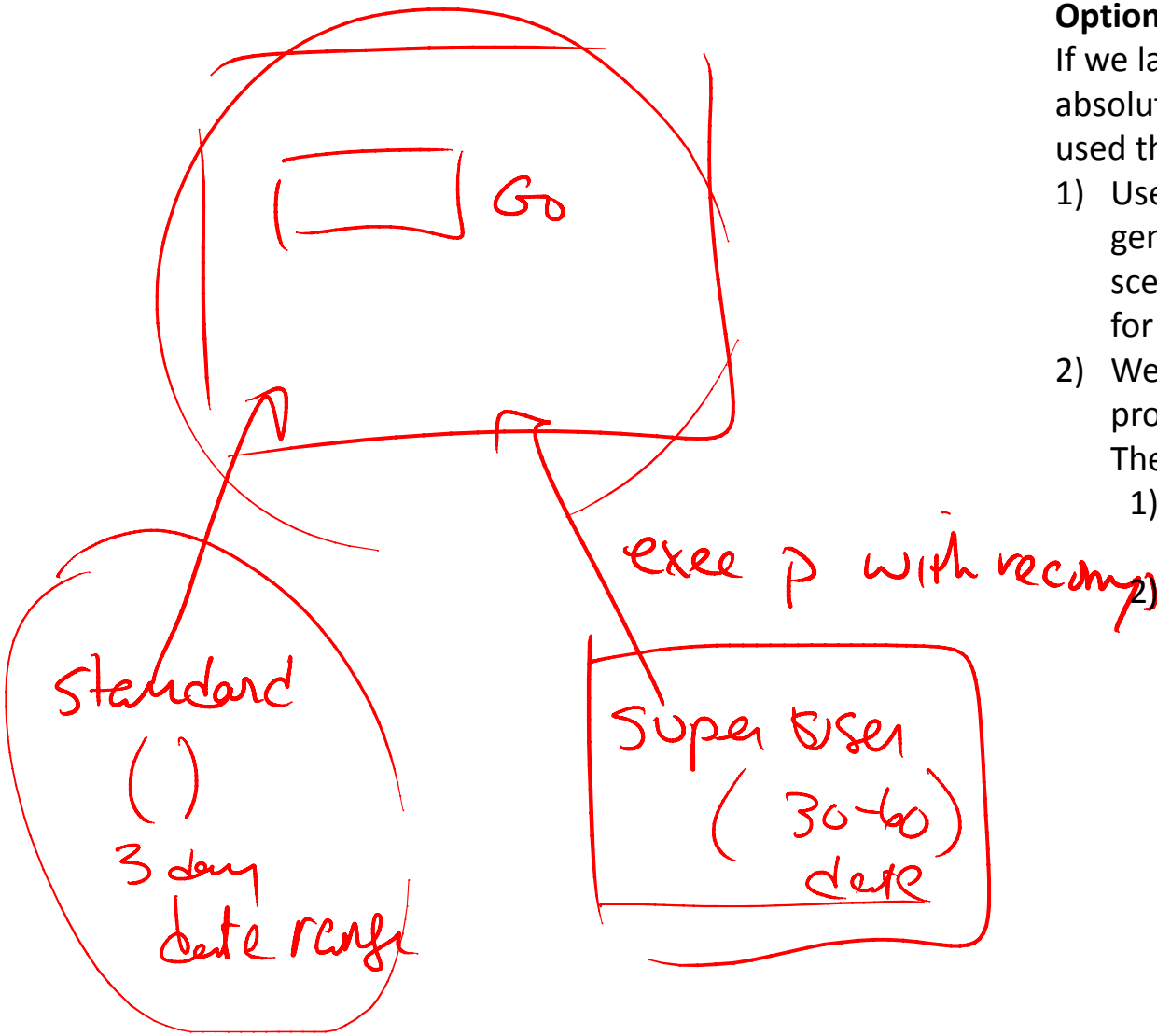
Multipurpose screens/dialog boxes that lead to multipurpose procedures

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)
                    …
```

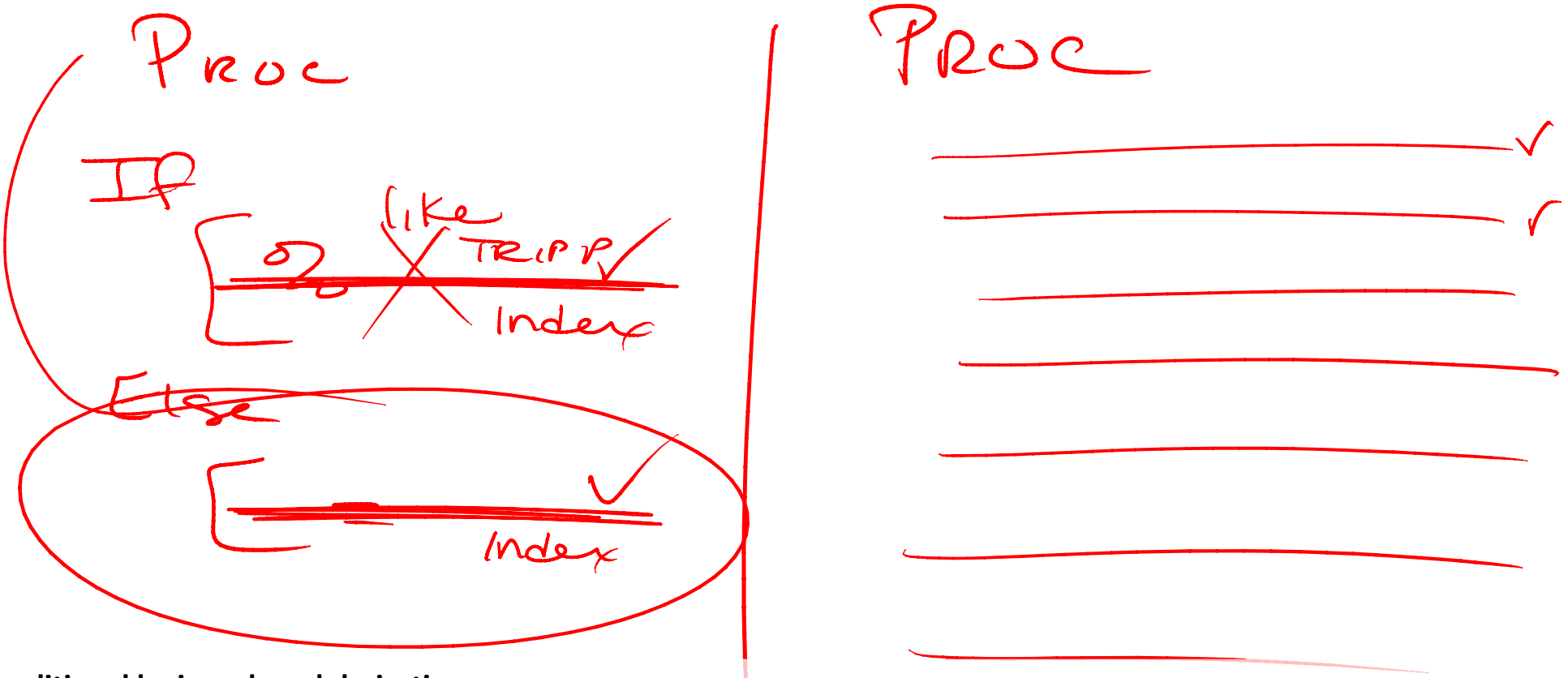
Check out the demo scripts in order to fully see how to better optimize this – using sp_executesql (for combinations that are stable) and sp_executesql WITH RECOMPILE for combinations that could generate different plans.



Options for different executions

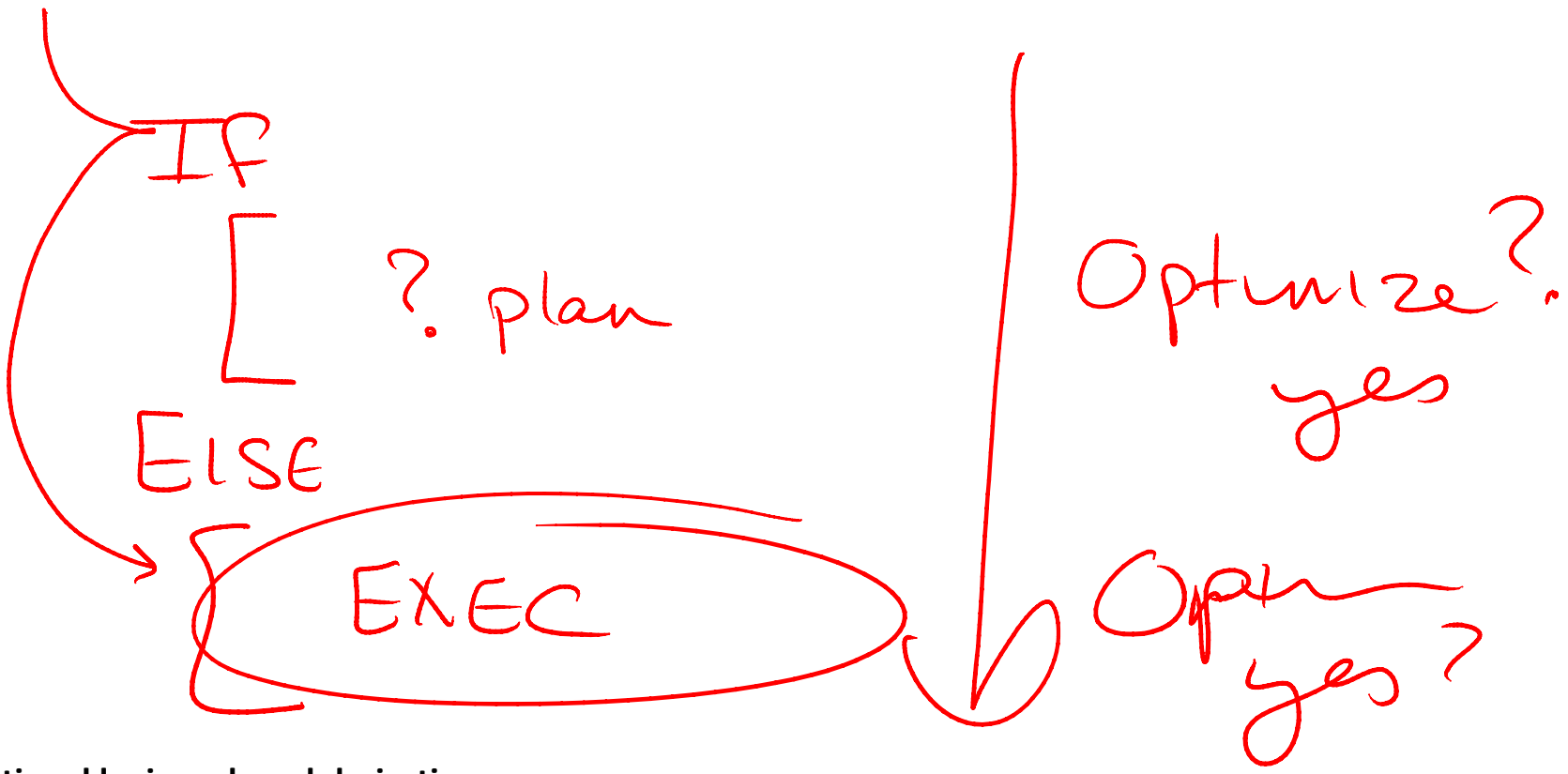
If we largely execute with "standard" users and we absolutely want THAT plan to be in cache and to be re-used then we could:

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)



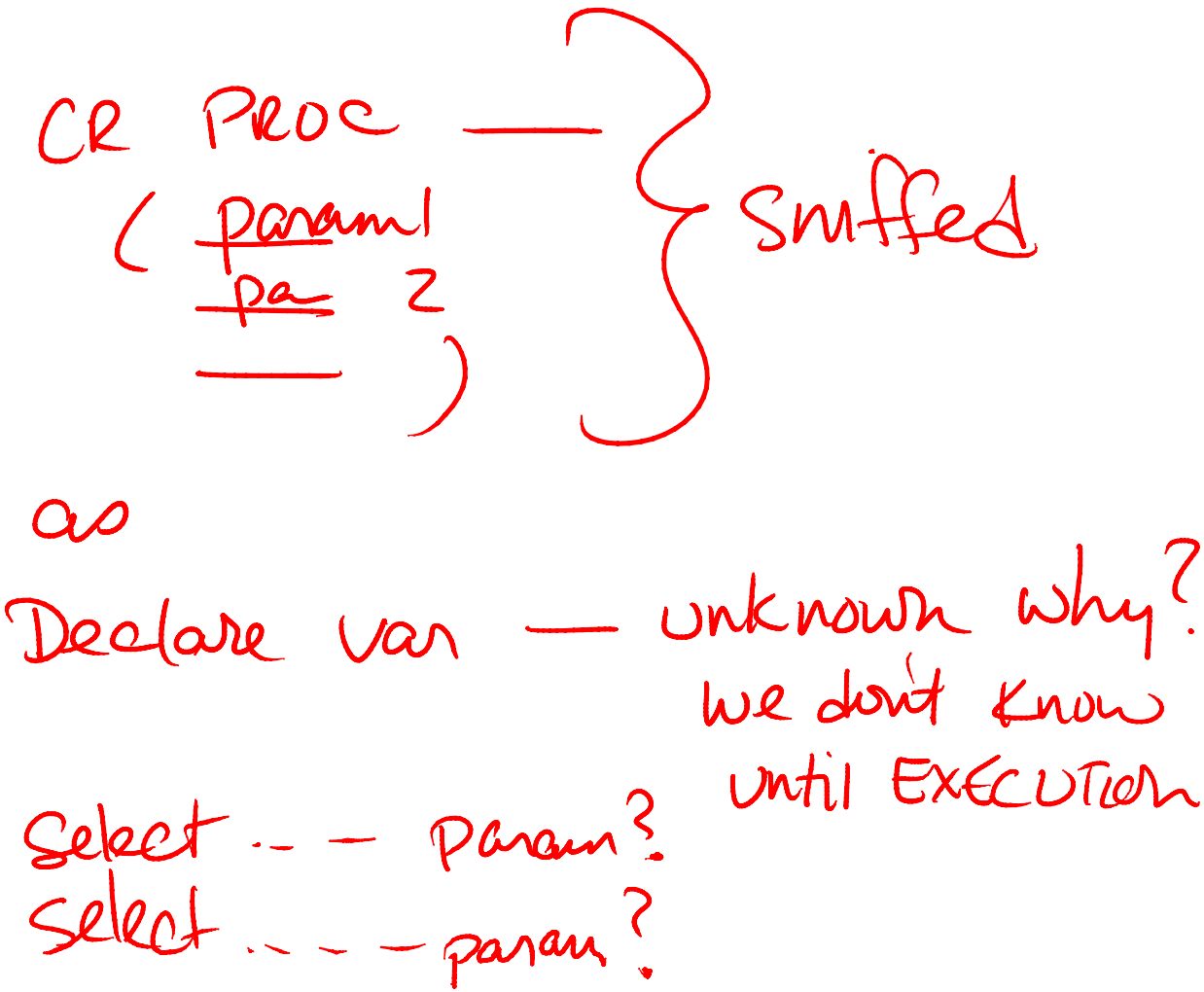
Conditional logic and modularization

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.



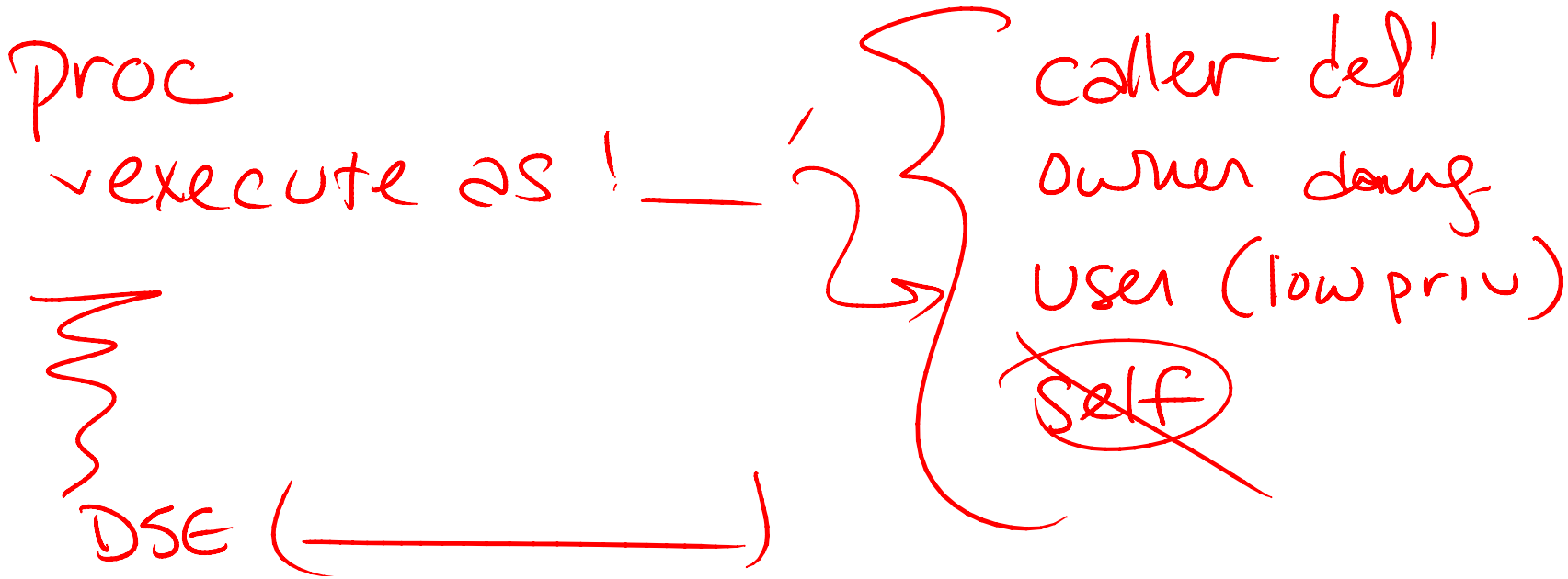
Conditional logic and modularization

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.



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. With 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.



Dynamic String Execution, EXECUTE AS and SQL Injection

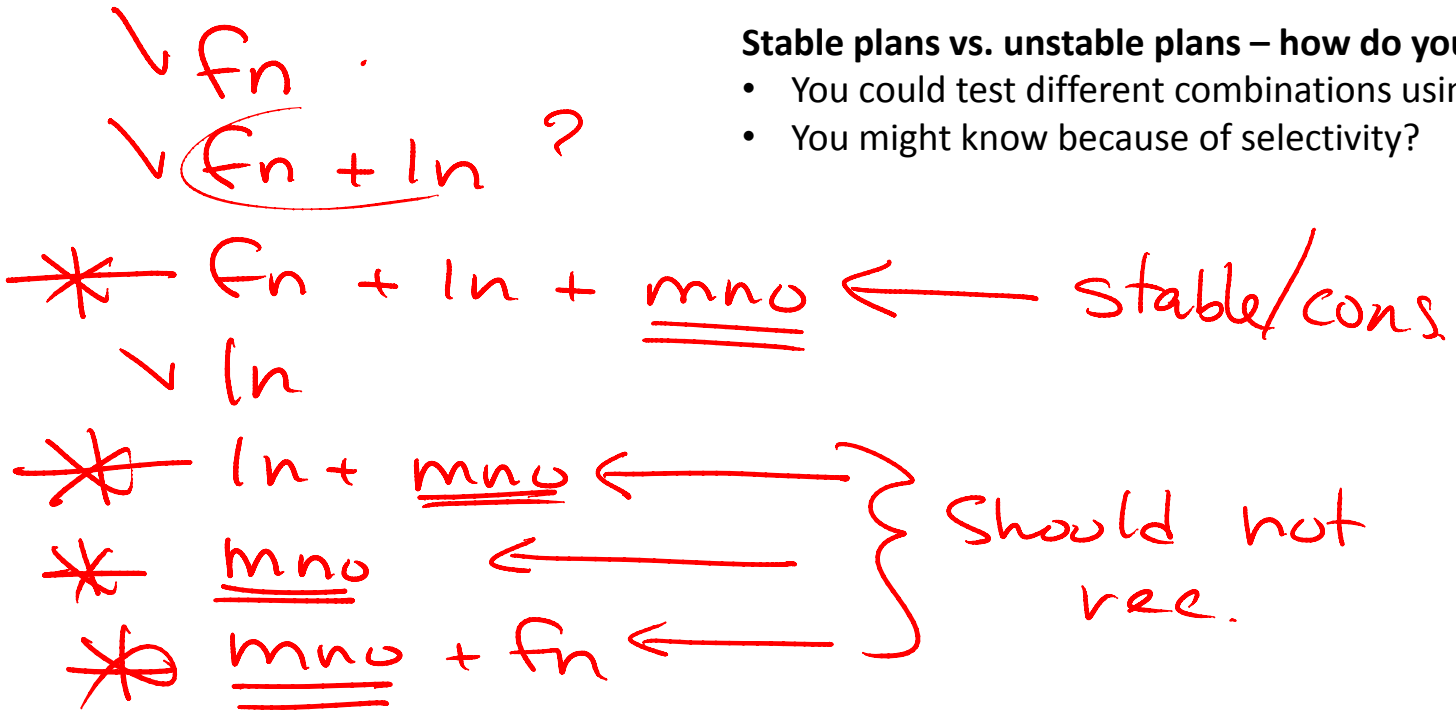
People complained that granting permissions directly to the user was a problem… so, they wanted an alternative. Enter: EXECUTE AS.

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: http://www.sqlskills.com/BLOGS/KIMBERLY/post/Little-Bobby-Tables-SQL-Injection-and-EXECUTE-AS.aspx

"EXECUTE AS" and an important update your DDL Triggers (for auditing or prevention): http://www.sqlskills.com/BLOGS/KIMBERLY/post/EXECUTE-AS-and-an-important-update-your-DDL-Triggers-(for-auditing-or-prevention).aspx



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

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

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 – will get a plan on each execution.