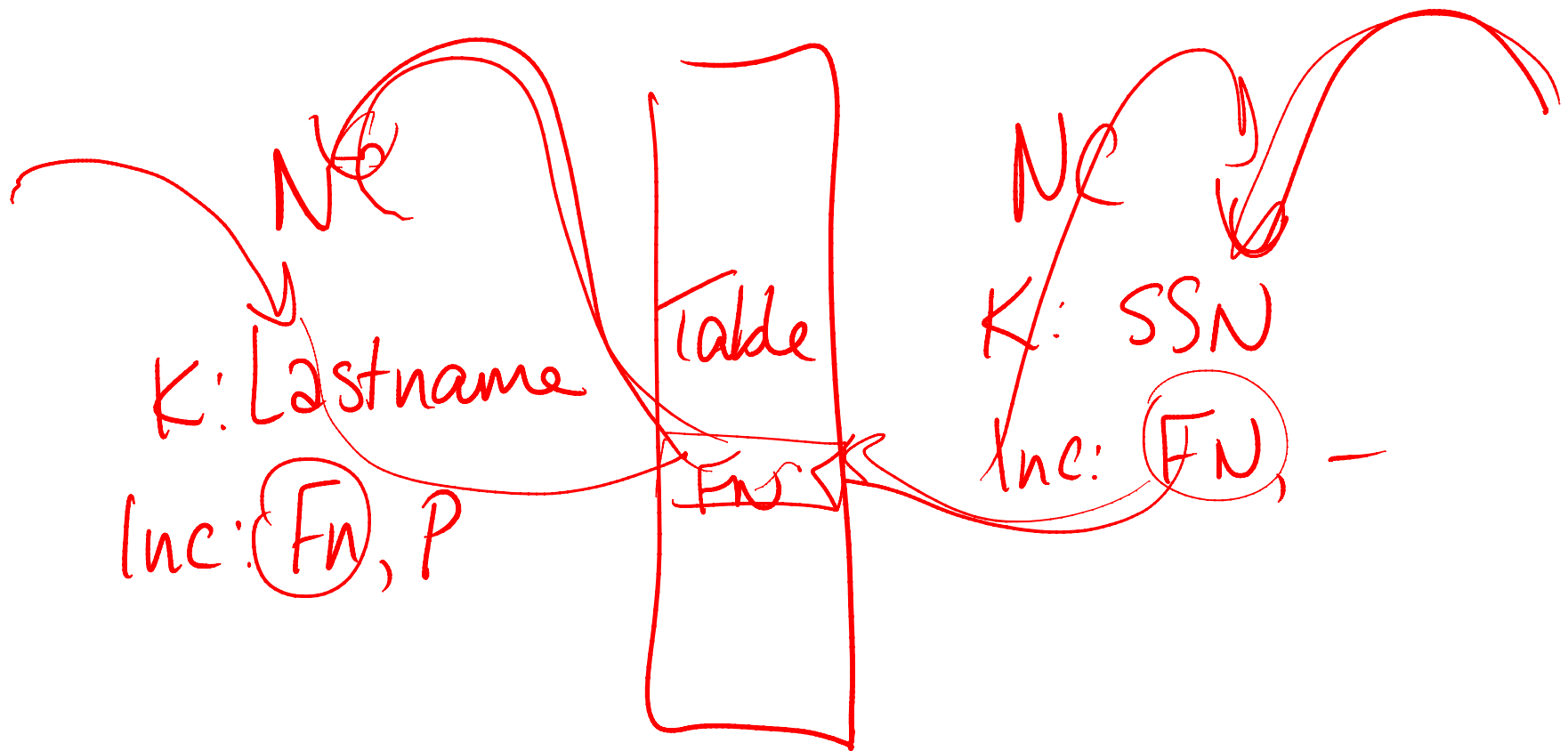
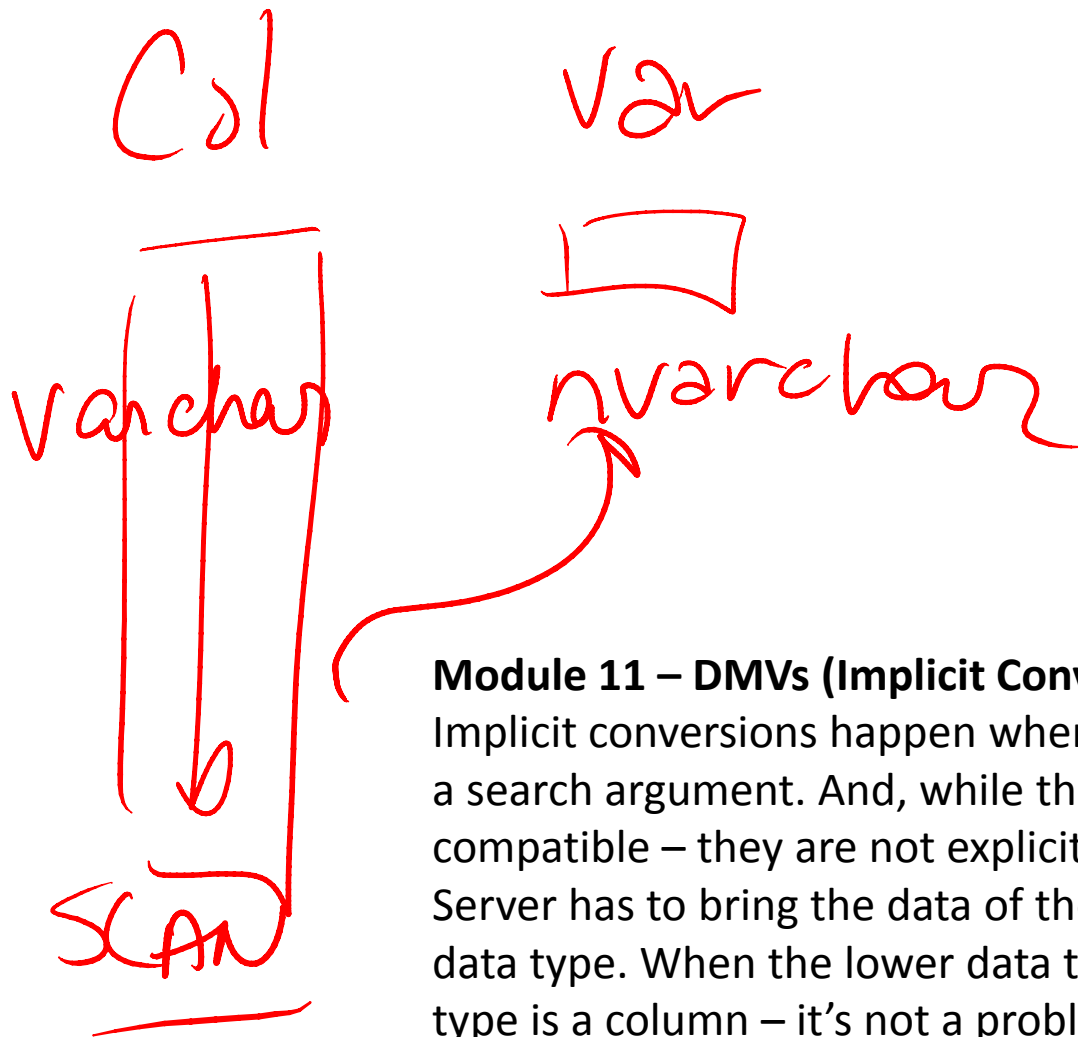




Module 10 – Analyzing Trace Data (Deadlocks):

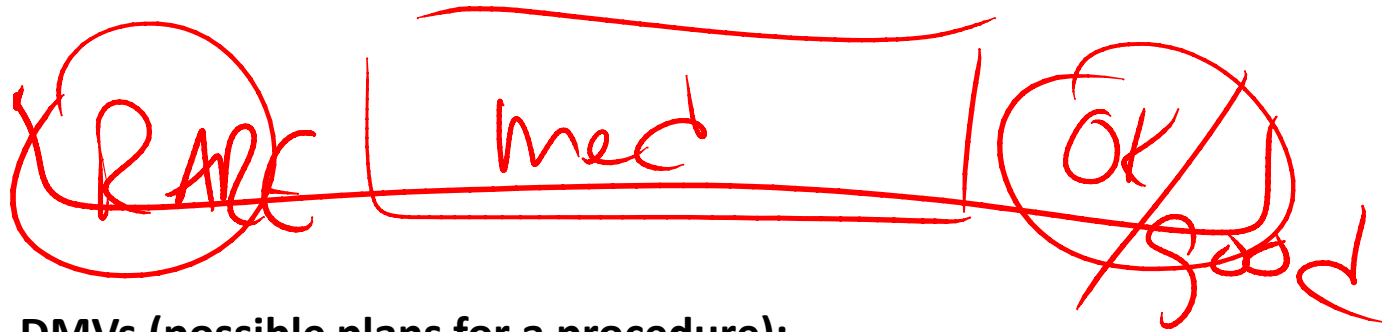
Another form of deadlock that can occur is one that happens because updates use different indexes to access the data (because of the WHERE clause predicate) and then need to modify the other indexes because they INCLUDE columns that are being changed by the update. If you see a lot of deadlocks in indexes (and they're not always on the same row – but, instead on the same page), you can use **sp_indexoption** to tell SQL Server to “use row locks” and “disable page locks.”



Module 11 – DMVs (Implicit Conversions):

Implicit conversions happen when two things are being compared in a search argument. And, while these two things might be “implicitly” compatible – they are not explicitly defined as such. In this case, SQL Server has to bring the data of the lower data type UP to the higher data type. When the lower data type is a variable and the higher data type is a column – it’s not a problem; it’s a simple conversion.

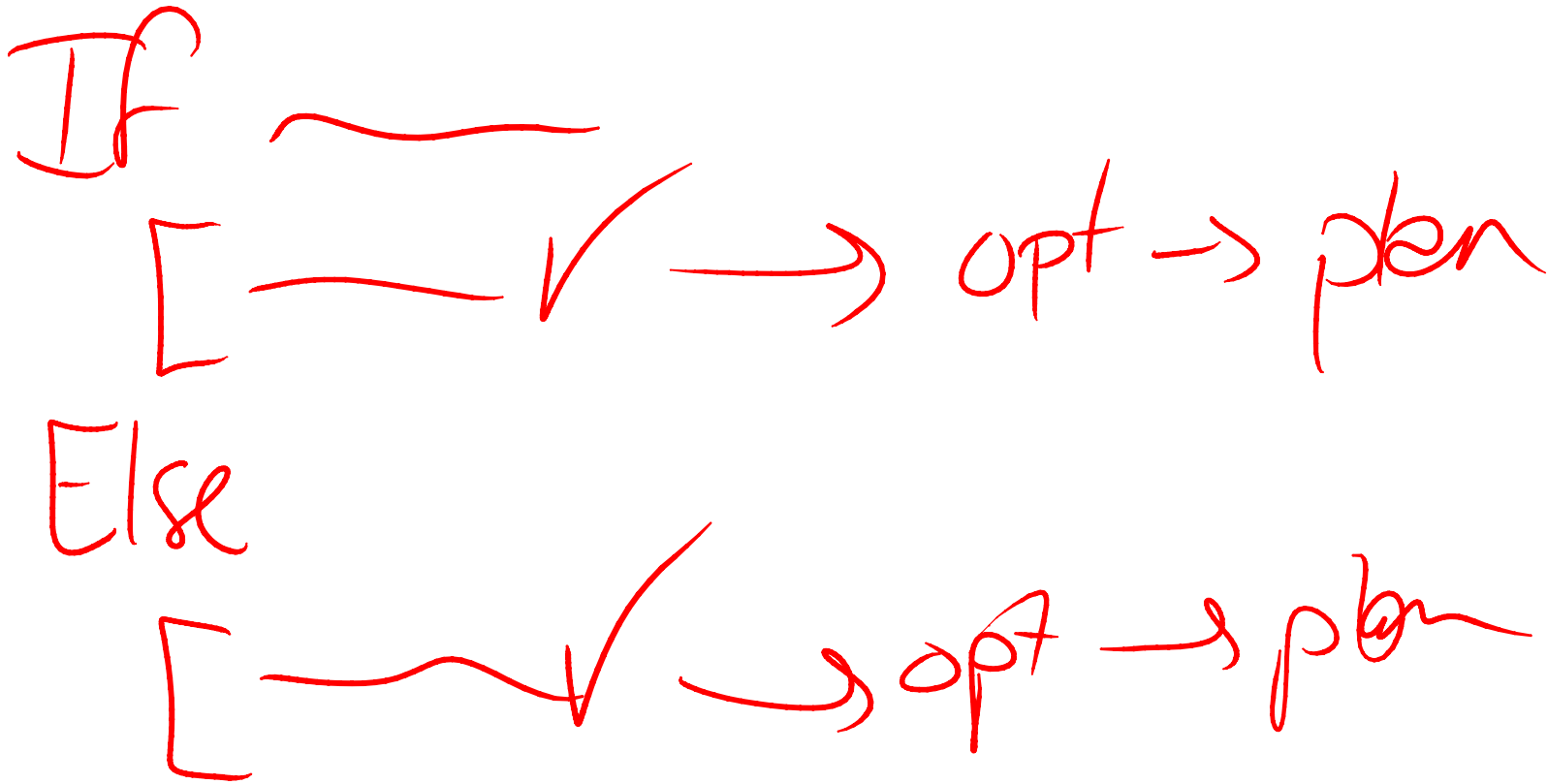
However, when the lower data type is a column and the higher data type is a variable – it’s a non-seekable expression and therefore requires a scan (not necessarily a table scan but if the right index doesn’t exist, then a table scan is often performed). You can use the code in the demo to find all implicit conversions in cache right now.



Module 11 – DMVs (possible plans for a procedure):

When a stored procedure has numerous input parameters, it's possible that when different combinations of these parameters are supplied – you get different plans. These plans might be OK for that particular execution but they are rarely good for ALL executions. So, what can happen is that patterns start to emerge. Some parameters (if they are used to create the plan) create a reasonable good plan in cache. And, since these are the more common combinations – things are good most of the time. However, there are some cases where other parameters just happen to be used to create the plan and when this happens a mediocre plan gets into cache. You might be able to catch this and force a recompile (directly – or, possibly indirectly through an update stats [having thought that it might be stats that are the problem]) and then the next execution just so happened to get a good plan in cache. So, it seems like the problem has been solved (more of just as Band-Aid really). And, in the worst case scenario – a very rare combination of parameters generates a plan that's not very useful for most other executions and performance tanks.

The scenario that I was describing was a real-world customer where that “rare” execution plan would cause EVERY execution to require 6 million IOs and because this was a frequently executed plan, the entire server would grind to a halt. Their solution (until we worked with them 😊) was to failover their cluster (24-way with 128GB of RAM). After working with them, we changed the way their code was written and moved them away from a lot of conditions like: **(column = @variable OR @variable is NULL)**

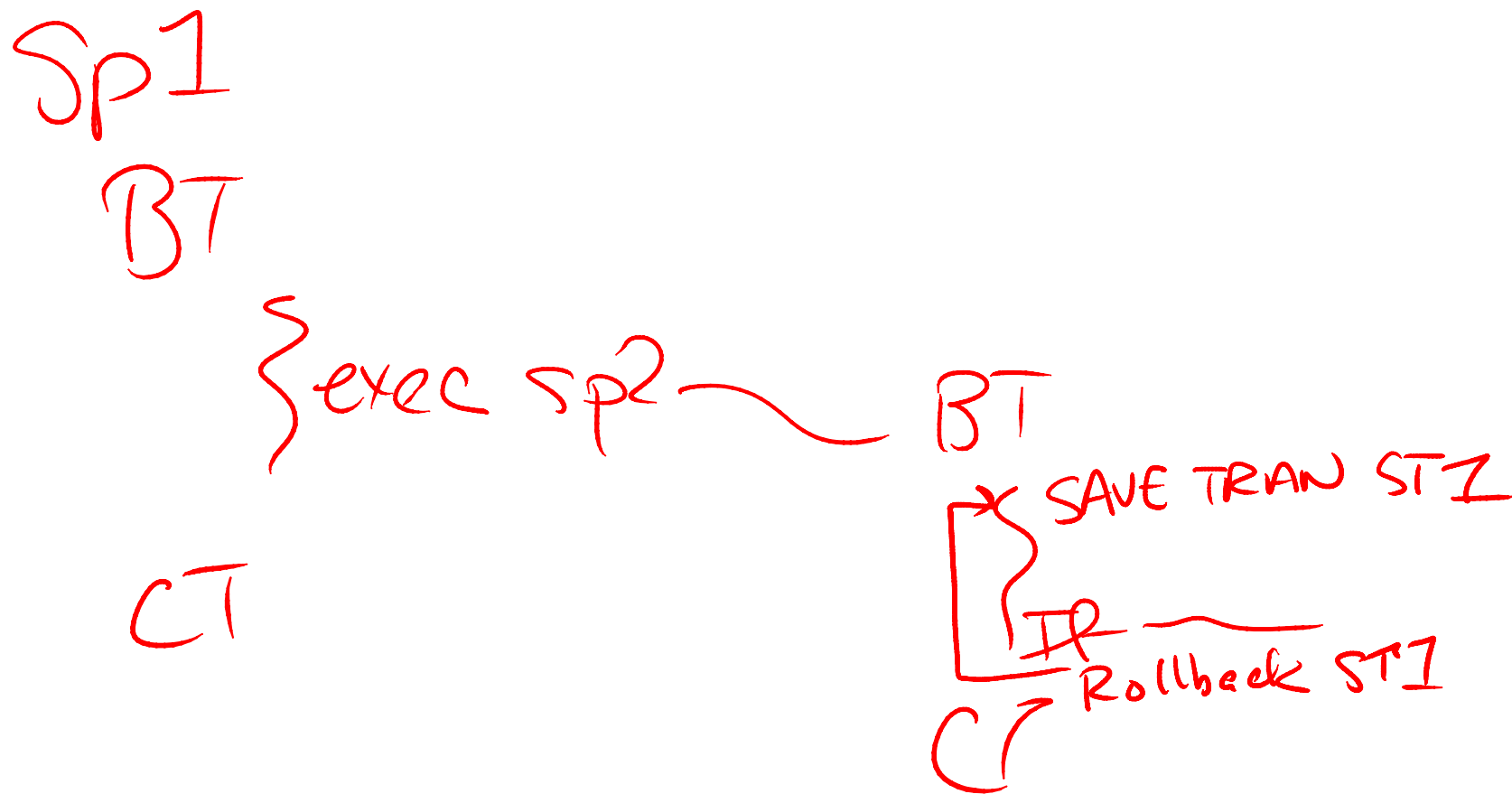


Module 11 – DMVs (Conditional Logic in procedures):

Because SQL Server optimizes the process of optimization – all statements that COULD be optimized WILL, even if the actual execution will not execute that branch of code. This can result in statements having a plan but not one that really applies to the type of parameters passed. When conditional logic branches to similar statements, it might be better to do one of the following:

- Execute on of the statements with **OPTION (RECOMPILE)**

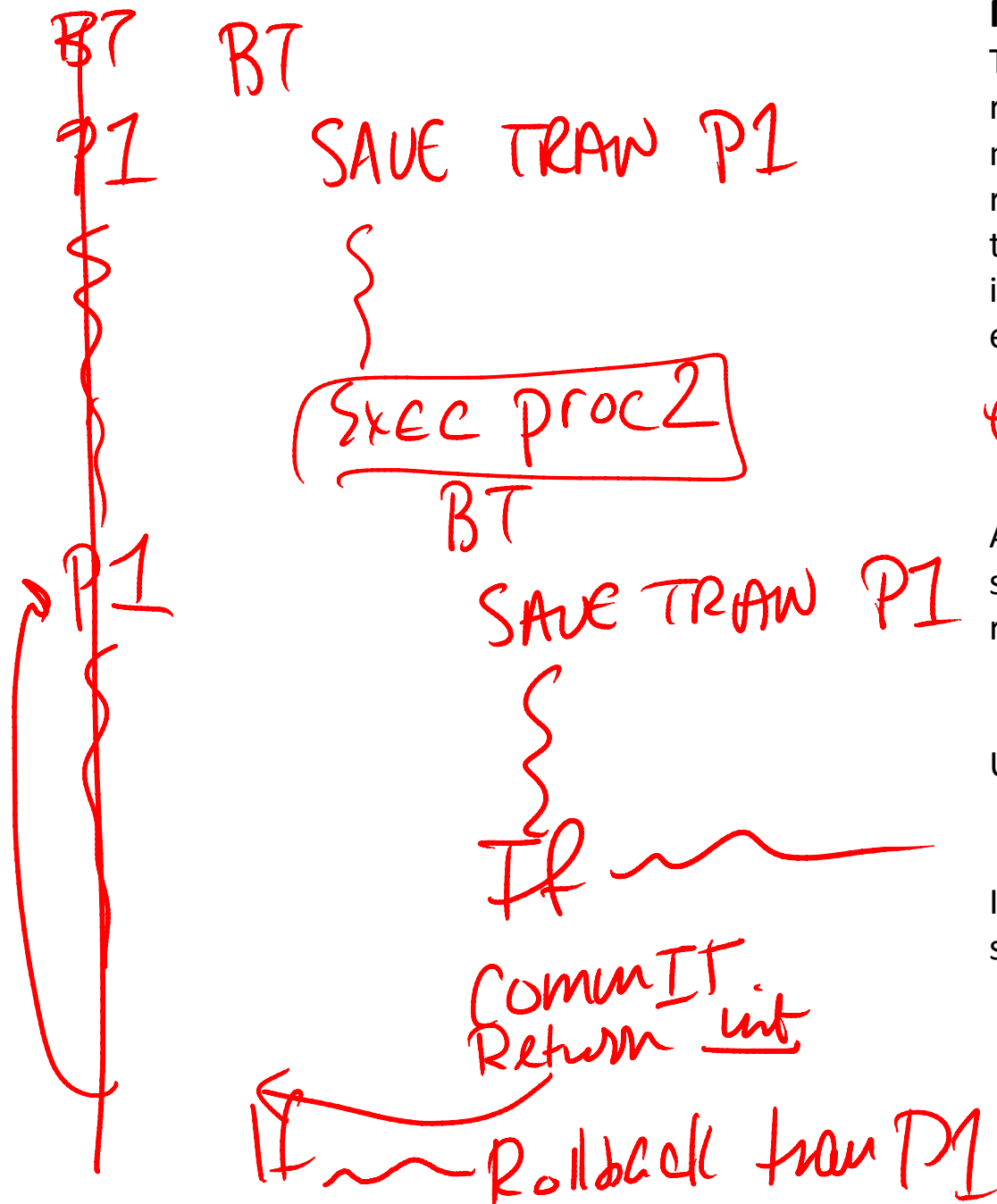
- Modularize your code and call to a subprocedure



Module 11 – “Nested Transactions” don’t really exist:

If a stored procedure calls to a subprocedure and uses a transaction – the best way to handle the transactional logic is to use savepoints (**SAVE TRAN name**). Then, inside the subprocedure you can **ROLLBACK TRAN name** and only rollback that the subprocedure has modified. However, a rollback to a savepoint does NOT decrement @@trancount; you must also use **COMMIT TRAN** before returning to the caller.

Note: be sure to use savepoint names that will be unique. Ideally, name them based on the procedure so that the likelihood of having two with the same name within a series of nested calls is impossible!



Module 11 – “Nested Transactions”

This should remind you that savepoints are really just in a stack – as if there are no nested transactions. If a rollback occurs it rolls back to the most recent savepoint with that name – even if that name was generated in a different procedure than the current executing context.

EXEC @st = _____

Also, another good practice is to use a return status – and, then document the possible return codes to the caller. Instead of

EXEC proc2

Use

EXEC @Status = proc2

Inside of proc2, use a variable to track the status and then return it to the caller:

RETURN @Status