

Whiteboard
annotations from
Monday, April 8, 2013

SQLintersection

PRECON6

Understanding Statement Execution and Optimizing Stored Procedures

(8:30 AM - 4:30 PM)

Kimberly L. Tripp

Kimberly@SQLskills.com



Parameters vs Variables

Stored Proc

_____ p2 \Rightarrow plan

_____ p1/p3 \Rightarrow plan

_____ variable - no plan
that sniffs

_____ use AVERAGE

_____ p1 - uses histograms

_____ variable (use density vector)

Param List

@p1 value

@p2 value

@p3 value

⋮

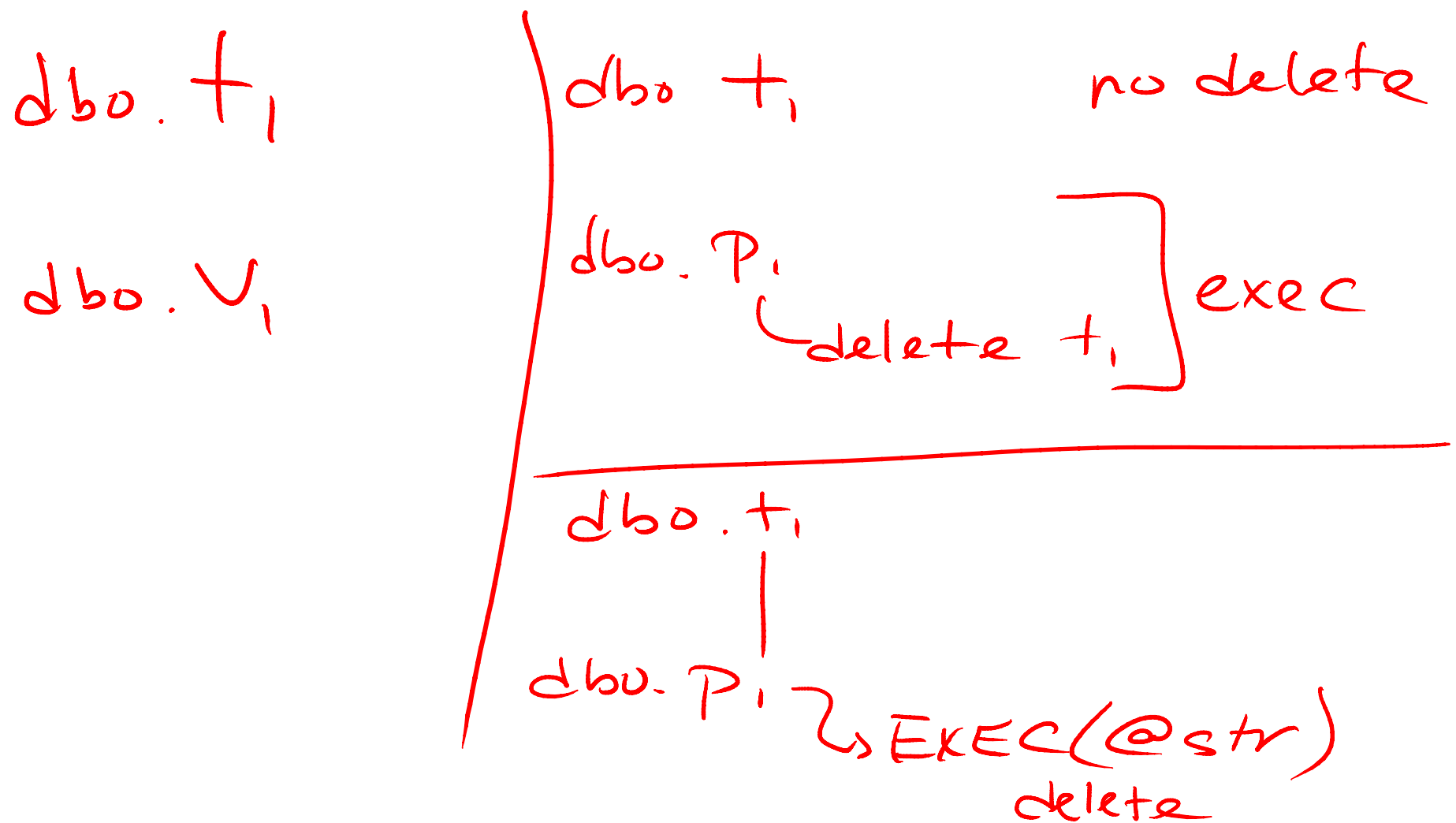
Parameters can be "sniffed" = histogram
Variable cannot = density-vector (average)

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).

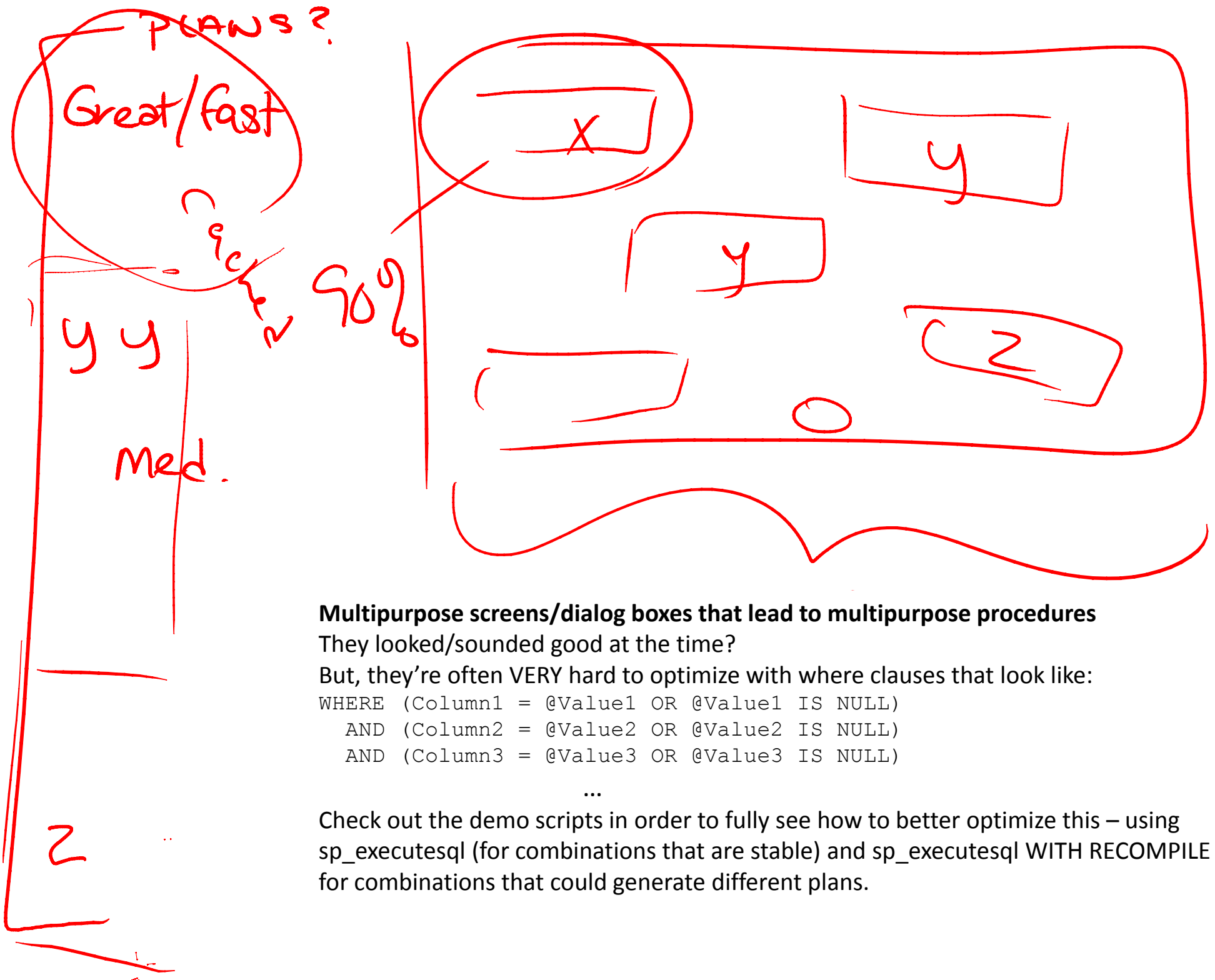
Variables are "unknown"

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.



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.



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.

Demo: Combinations of Parameters Supplied

like LN → recompile

like FN → an

mno → stable

like LN FN → compile

~~LN~~ mno → stable

~~FN~~ mno → stable

~~FN~~ ~~LN~~ mno → stable

Options that better balance 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.



Read Uncommitted = no lock
DIRTY READS

Read Committed (default) / RC

uses locking

Inconsistent Analysis

using
Versioning

remove

non repeatable
phantoms

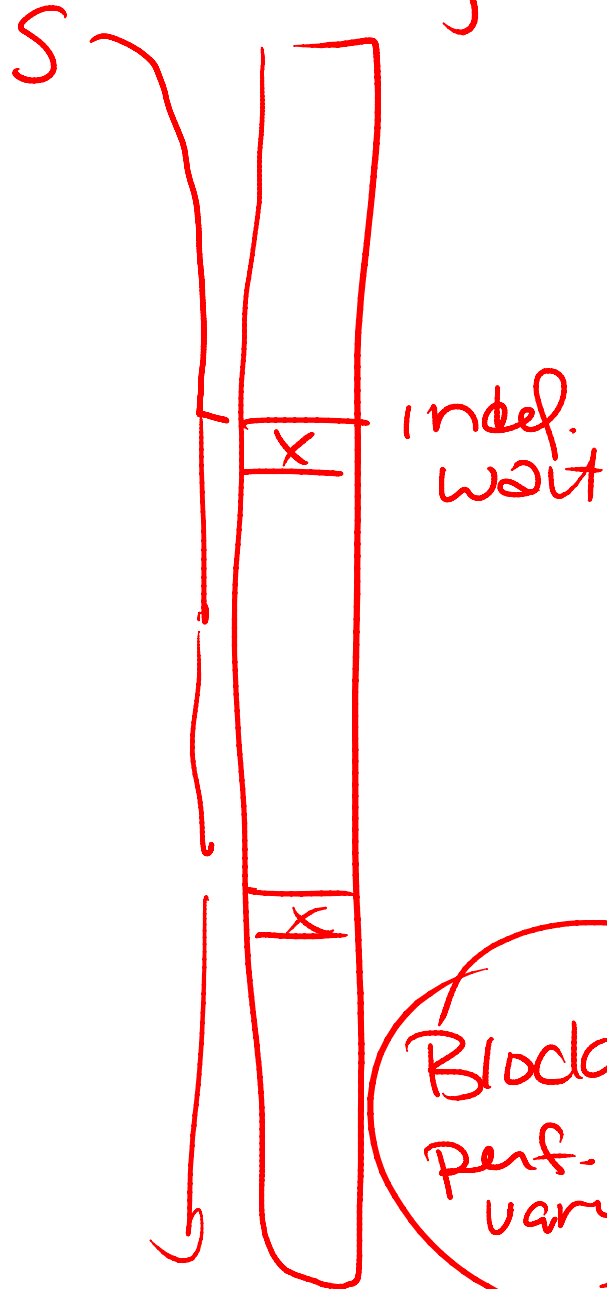
Repeatable Reads

Serializable

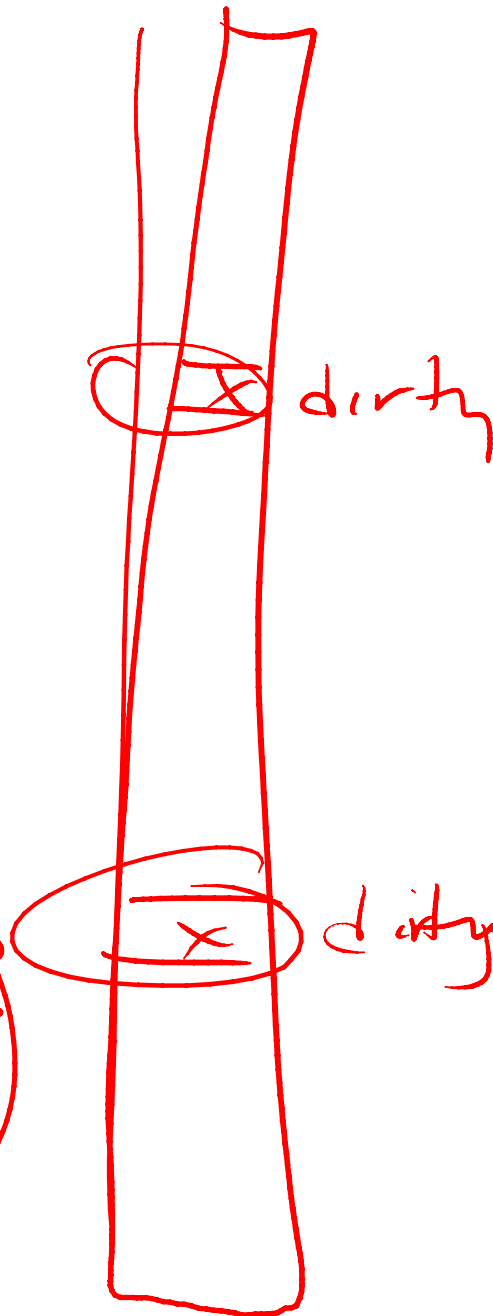
read-committed-
snapshot

- **ACID transaction requirements**
 - Atomicity Consistency **Isolation** Durability
- **Isolation levels (session setting shown in sys.dm_exec_sessions)**
 - Read uncommitted (1)
 - Read committed (2)
 - Repeatable reads (3)
 - Serializable (4)
 - Snapshot (5)
- **Default isolation level in every release is ANSI/ISO read committed**
- **SQL Server implementation uses locking for all levels**

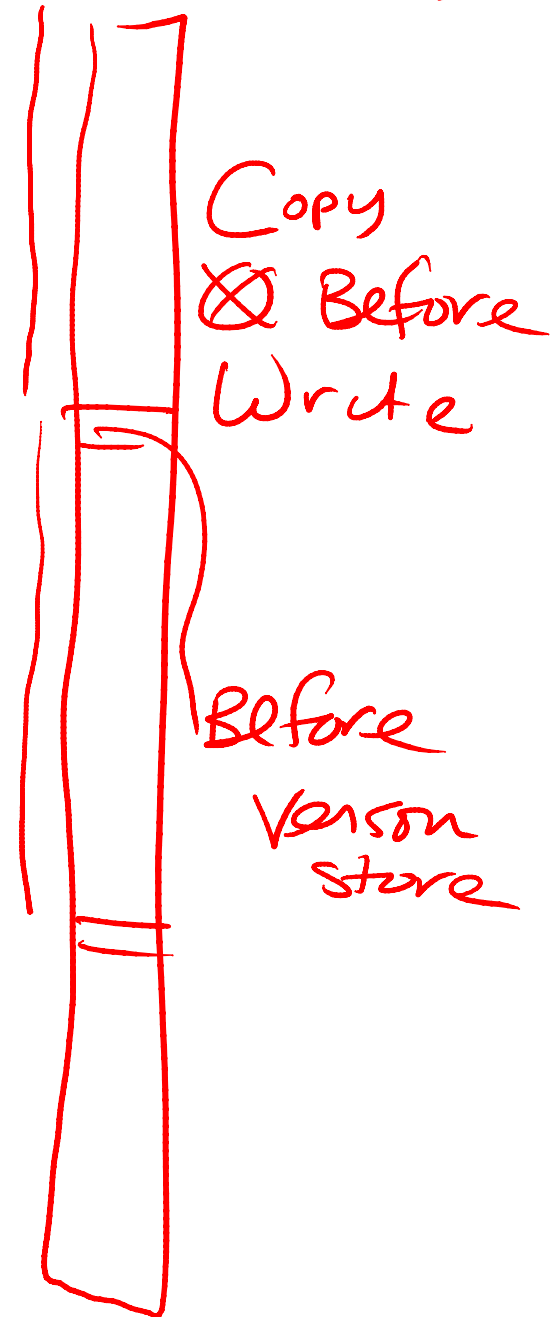
RC using
locking



RC-Read
Uncommitted



RC using
Versioning



Blocking?
perf.
vary

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 than 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.

Recovery Models

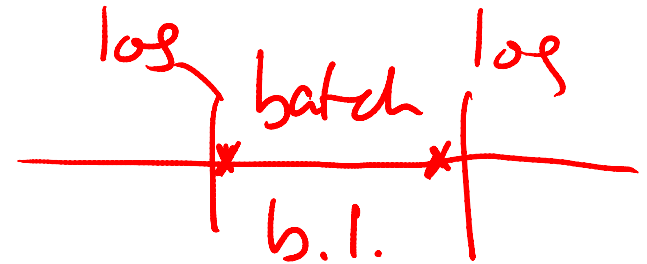
min
loss
mode

Simple - Clears log when checkpoint

Full backups min

BULK_LOGGED -

Controlled batch window
w/o users



FULL until 1ST backup
(pseudo-simple)

DB+LOG min backups

up-to-the-
minute

The prior slide showed the 3 recovery models

(1) The default – FULL Recovery Model

Until you've done a backup, you're in the pseudo simple recovery model. This means that the log will clear on checkpoint (which is what the SIMPLE recovery model does). But, the benefits of FULL when you are doing backups is that you have the "full" set of recovery options available to you. You have:

- * Up-to-the-minute recovery – this is the ability to backup the log even when the database is inaccessible/damaged. And, if you have a good backup strategy then you should be able to recovery without any data loss at all.

- * Point-in-time recovery – this is the ability to recover to a definable point in time.

The negative is that operations must be fully logged to support these features.

(2) BULK_LOGGED Recovery Model

To allow some operations to run minimally logged (for speed) you can switch to bulk_logged without breaking the continuity of the log chain. However, a log backup is only possible when the data portion of the database is available. And, the log backup will be as large as if you had been running in full but the operation runs faster and uses less active log space with minimal logging. However, only a small number of things can be run in a minimally logged manner.

(3) SIMPLE Recovery Model

There are lots of benefits to the "simple" recovery model. The biggest of which is that your backup strategy is simplified. You will not be required to do log backups and in fact, you can't. The log will be cleared on checkpoint. Your recovery options are to recover from a full (and possibly a differential) and you have the largest amount of potential data loss. This should not be used for critical databases.

Resources

[Backup Resources – Where, oh where, can they be?](#) (Tripp, Blog post)

[TechNet Magazine: feature article on understanding logging and recovery](#) (Randal, Blog post)