

SQLskills Online Immersion Event

Kimberly's Whiteboard Drawings and Annotations from IETLB
IETLB: Transactions, Locking, Blocking, Isolation, and Versioning
Online Course: October 9-10-11, 2018

Plus drawings from other classes (that were better) or just had info we may not have covered!

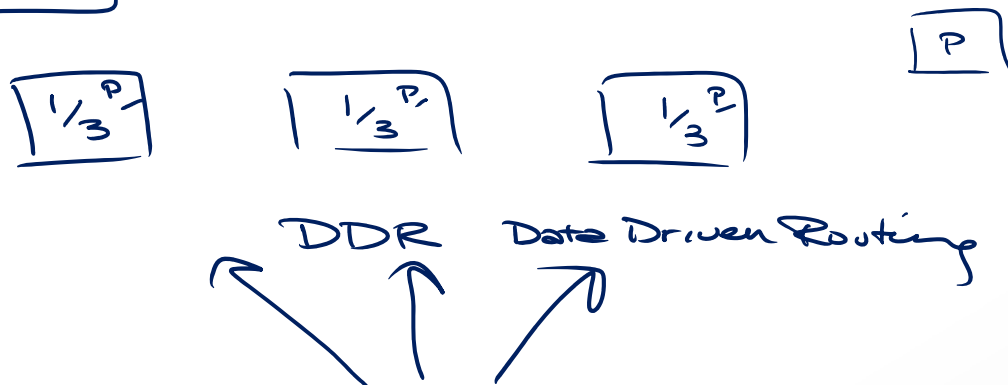
Have fun!

- k

Kimberly L. Tripp
President / Founder, SQLskills.com
Kimberly@SQLskills.com
@KimberlyLTripp



Scale Out



Scale-out / designing for a scalable architecture

We talked about the potential need for distributed transactions in a scale out scenario. However, to really make "scale-out" design truly scale – you need to design it properly. The most important thing is that MOST activity as to stay local and be correctly directed to the proper server through middle-tier DDR (data-driven routing). If every user randomly goes to any of these instances and all requests go through [distributed partitioned] views then performance will likely be worse! What this will likely require is some data redundancy... for example, the products table will be duplicated on all servers. When price changes need to be made – they must be made on all servers, transactionally, using MSDTC.

Check out the whitepapers on SODA (service-oriented database architectures) from our whitepapers page:

<https://www.sqlskills.com/sql-server-resources/sql-server-whitepapers/>

Implicit / auto commit trans.

Set implicit_transactions on

Update

S

in

commit tran

Explicit

Begin Tran

commit tran

ACID properties and transaction definition

While discussing locking basics, we talked about ACID properties.

We also started to discuss transactions:

- *Implicit* is now auto-commit. In most people's minds implicit used to mean that SQL Server implicitly treats your statements as a transaction. However, Oracle's implicit mode begins a transaction for you "implicitly" and so these should not be confused. To reduce confusion, use the term auto-commit instead.
- *Explicit* are user-defined transactions where you've explicitly defined the begin/commit.

Transaction Madness

The key points are:

BEGIN TRANSACTION

- Increments @@trancount

COMMIT TRANSACTION

- Decrements @@trancount

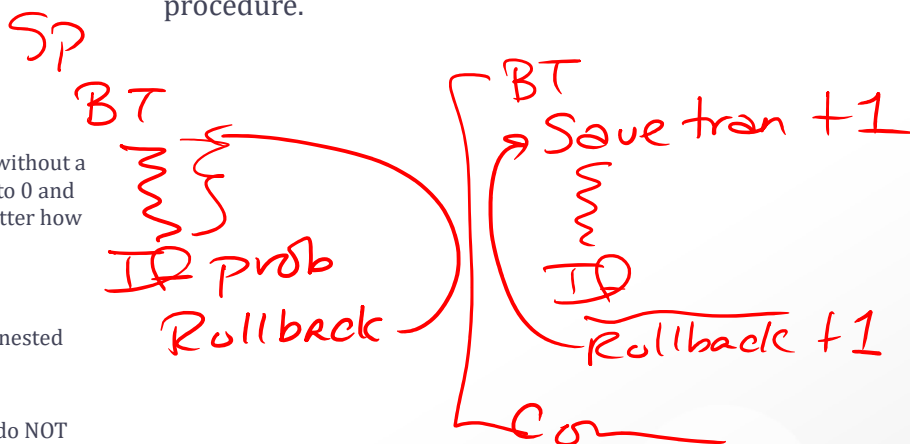
ROLLBACK TRANSACTION

- When used with a transaction name OR without a name at all - always resets @@trancount to 0 and cancels ALL pending transactions - no matter how deeply nested.

A transaction is NOT committed until @@trancount = 0. When transactions are nested you MUST COMMIT for every BEGIN

Savepoints (SAVE TRANSACTION NAME) do NOT affect @@trancount and can be rolled back to safely without affecting @@trancount.

A stored procedure is NOT a transaction in and of itself. You MUST use BEGIN/COMMIT to engage the ACID properties of the statements of a stored procedure.

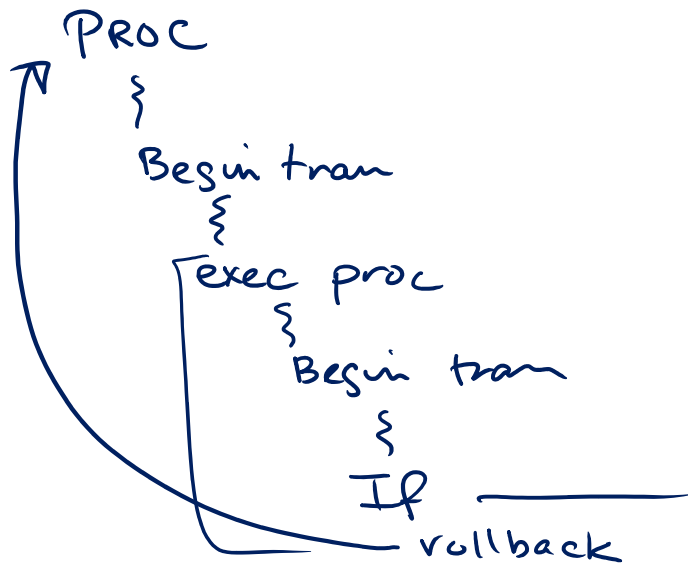


Be sure to go through the demo scripts if you need more insight into transactions!

Transaction Madness

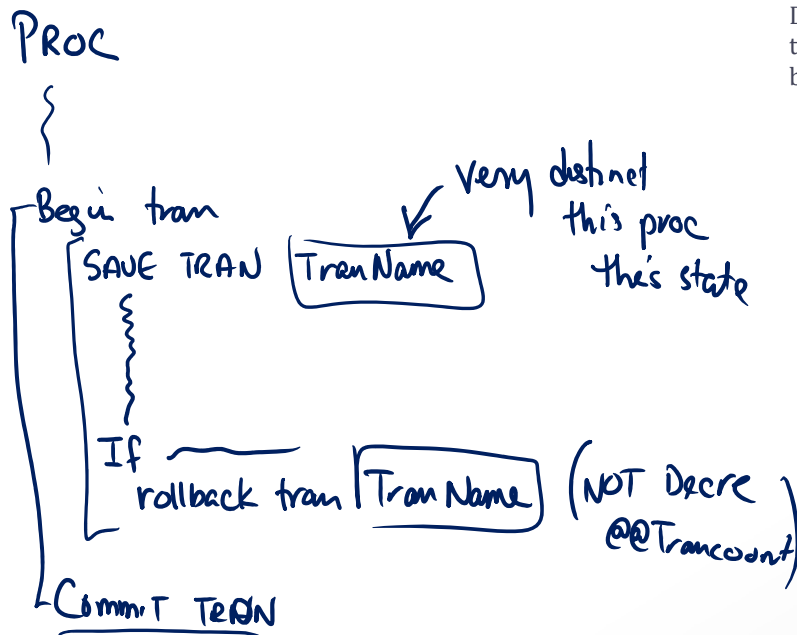
There is ALWAYS only ONE transaction.

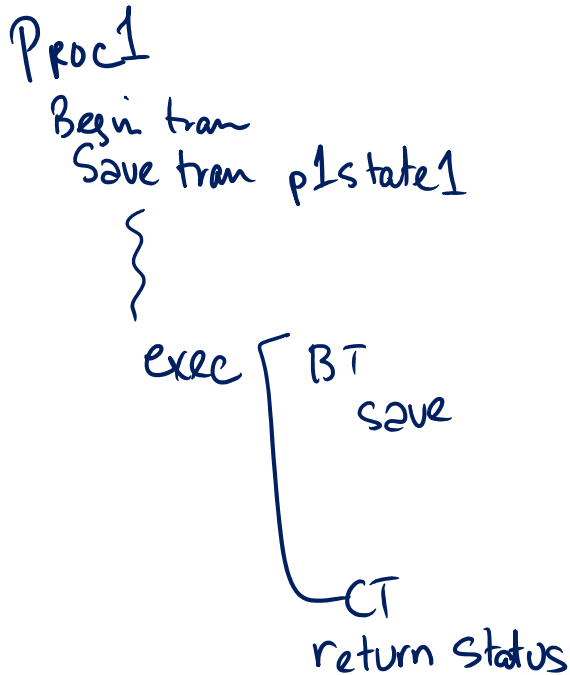
Rollback tran does exactly that – it rolls back the ENTIRE transaction and resets @@TRANCOUNT to 0 (no matter how many levels deep it's nested)



Rollback to a savepoint...

Does not reset @@transaction and therefore requires a commit transaction before returning to the caller!



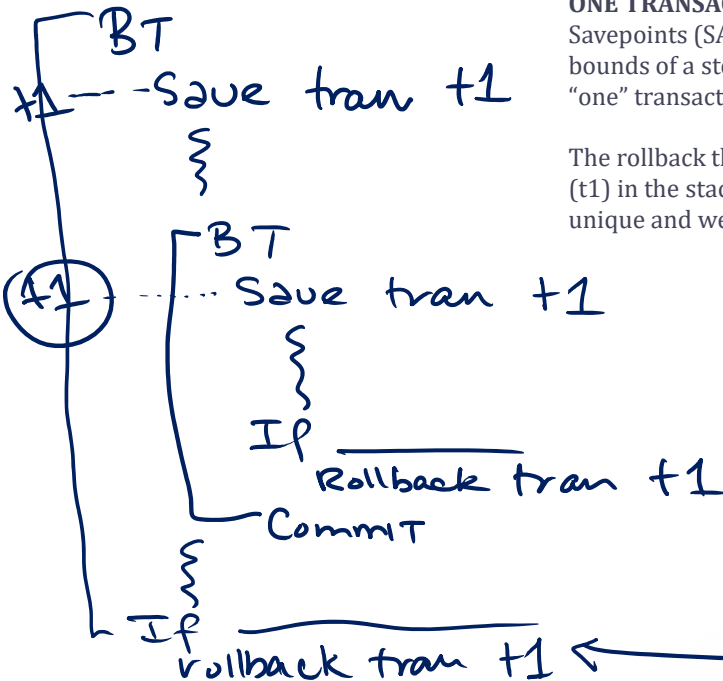


Procedure Standards / Consistency

Consider using savepoints (SAVE TRANSACTION) within procedures to allow a procedure to rollback to ONLY that which its done. However, REMEMBER that if SET XACT_ABORT ON AND using SAVE TRANSACTION you might end up in an uncommittable transaction and the only option will be to ROLLBACK the entire thing.

So, you need to either pass back return statuses and/or check @@trancount at every level to see where you're at. Using XACT_STATE will tell you the state of a transaction and whether or not it's committable.

See the error handling and xact_abort script for help with best practices in this area!



ONE TRANSACTION

Savepoints (SAVE TRANSACTION) are a "stack" even when in the bounds of a stored procedure. The only "scope" for these is the "one" transaction in which they are a member.

The rollback that's below actually rolls back to the LAST savepoint (t1) in the stack. You'll want to make sure that every save point is unique and well-named.

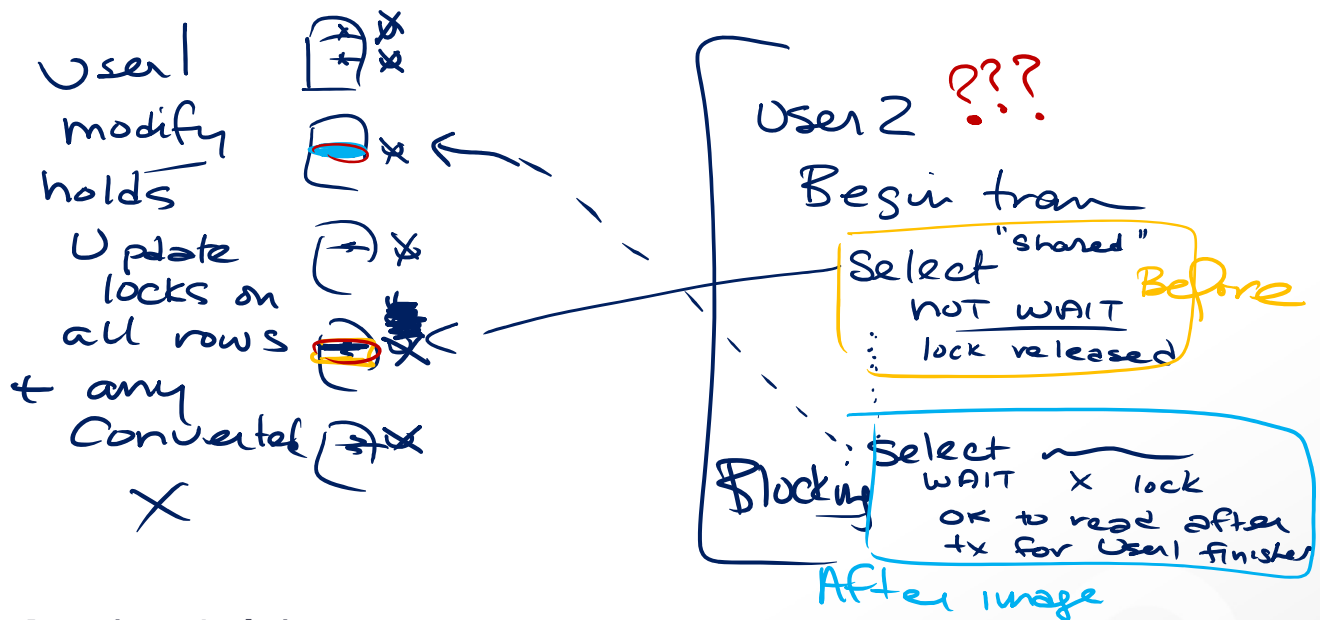
Naming savepoints

Because "nested transactions" do not exist (instead there is only ONE transaction) – you need to be careful when using (and specifically in naming) savepoints. Since these are just a stack – the "last" save point is used in a rollback – even if it's not contextually within the code that's rolling back to it.

See the next image if you've named your save points with "simple" names

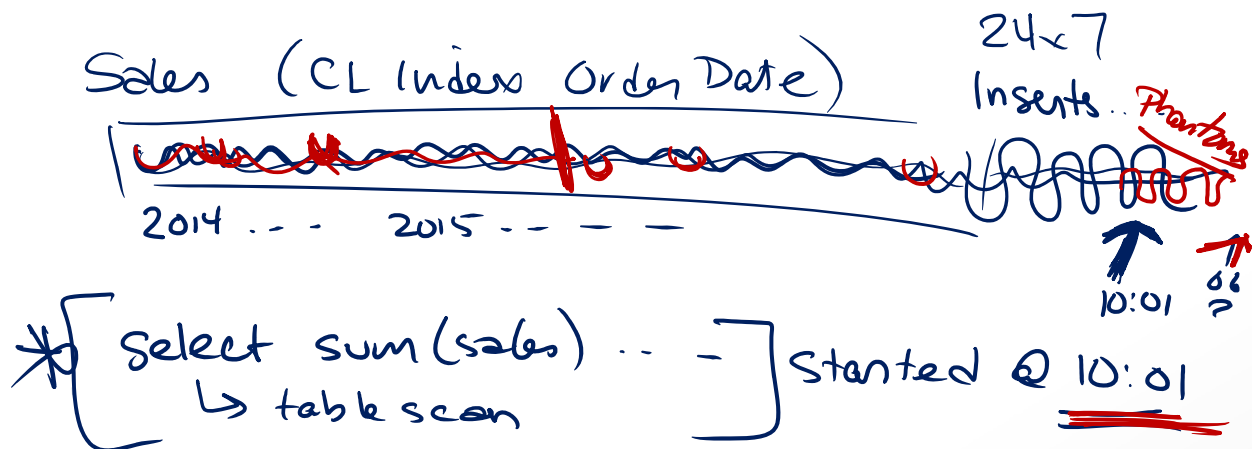
A GOOD save point name is one that reflects the procedure and the state with a bit of detail.

Proc1_StateX_PointY



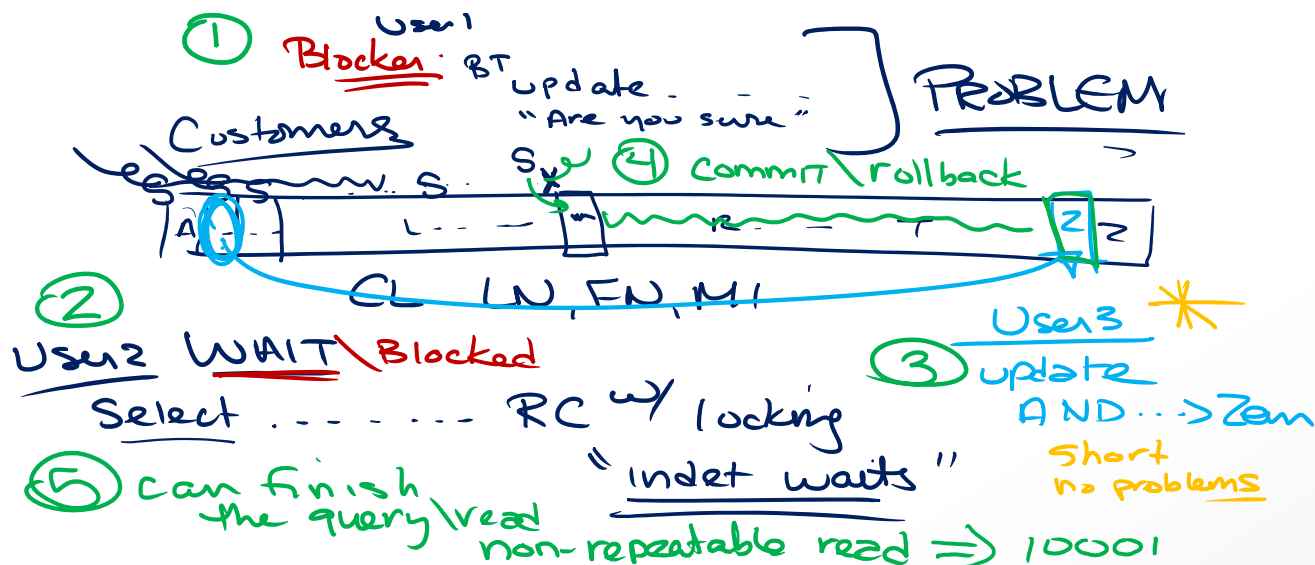
Inconsistent Analysis

There are many forms of inconsistencies caused by read committed's lack of statement / transaction consistency. Because SQL Server only cares about the state when a row is read – there is no accuracy with respect to the other rows in the statement / transaction (unless you use a more strict level of isolation OR versioning).



Inconsistent Analysis

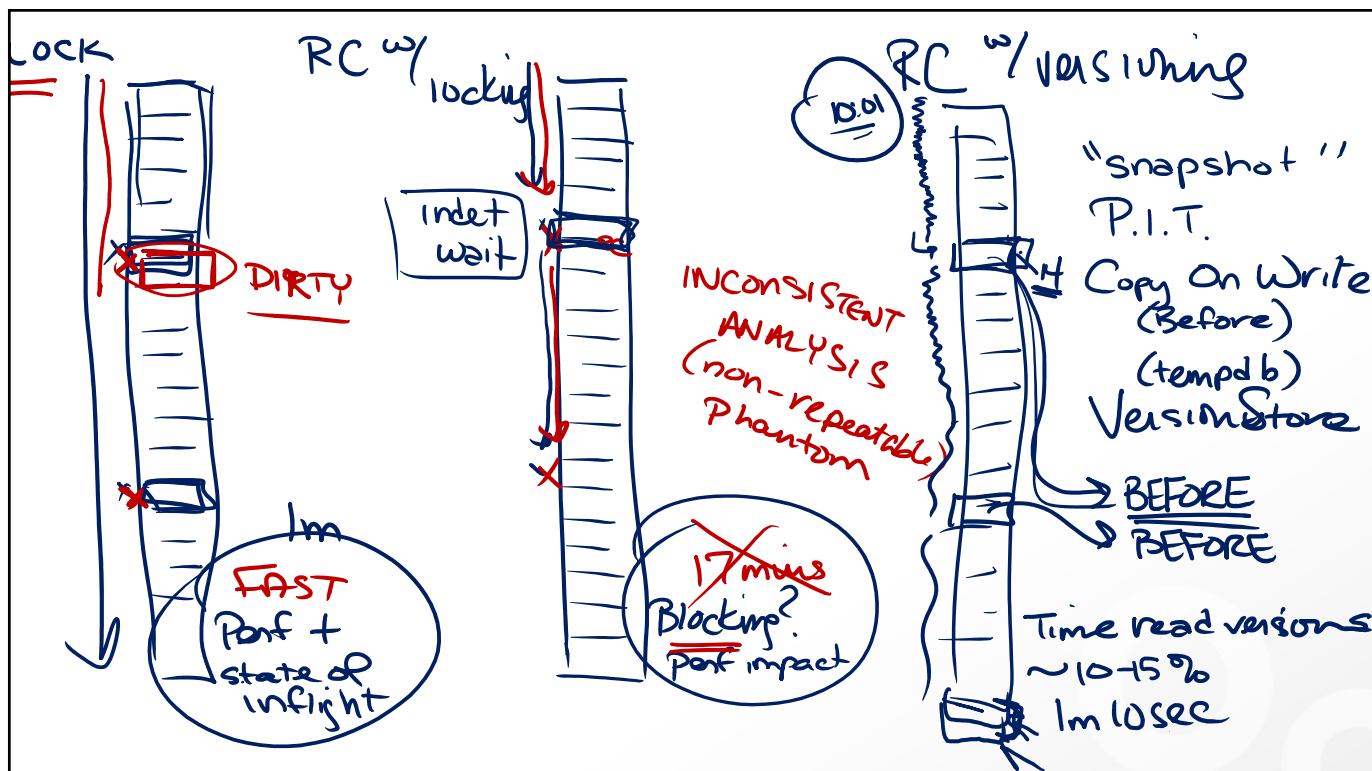
There are many forms of inconsistencies caused by read committed's lack of statement / transaction consistency. In this case, I started by highlighting both the pros and cons of phantoms in a result set. However, when you really look at what RC using locking can end up with (in terms of accuracy) you start to see that the inconsistencies CAN be widespread depending on the volatility, the indexes, and the transaction length.



Demo: Non-repeatable reads (Inconsistent Analysis)

This is showing how an update to a CL key value that causes record relocation (after another query has already read [and released the lock on] the row) allows the reader to read the row TWICE (non-repeatedly).

This is from the demo on non-repeatable reads and shows what I called the "anderson-zembrowsky" problem. ☺



The prior slide showed a table vertically as a set of pages. The idea was to compare/contrast the behaviors between:

(1) A forced NOLOCK

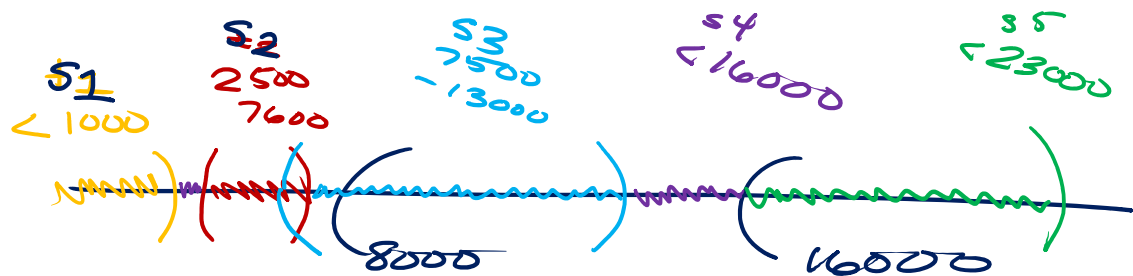
The key thing to remember here is that nologlock allows the reader to read quickly – not stopping for locked rows. However, these rows may be “in-flight” and their modified data might end up getting rolled back. If you’re looking for only an estimate – this might be fine.

(2) The default – READ COMMITTED using locking (RCL)

The key things to remember from the picture is that RCL 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 RCL is prone to a few inconsistencies (aka. Inconsistent analysis). Only way to solve – increase isolation (or [as of 2005] consider using versioning).

(3) READ COMMITTED using versioning (RCSI)

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.



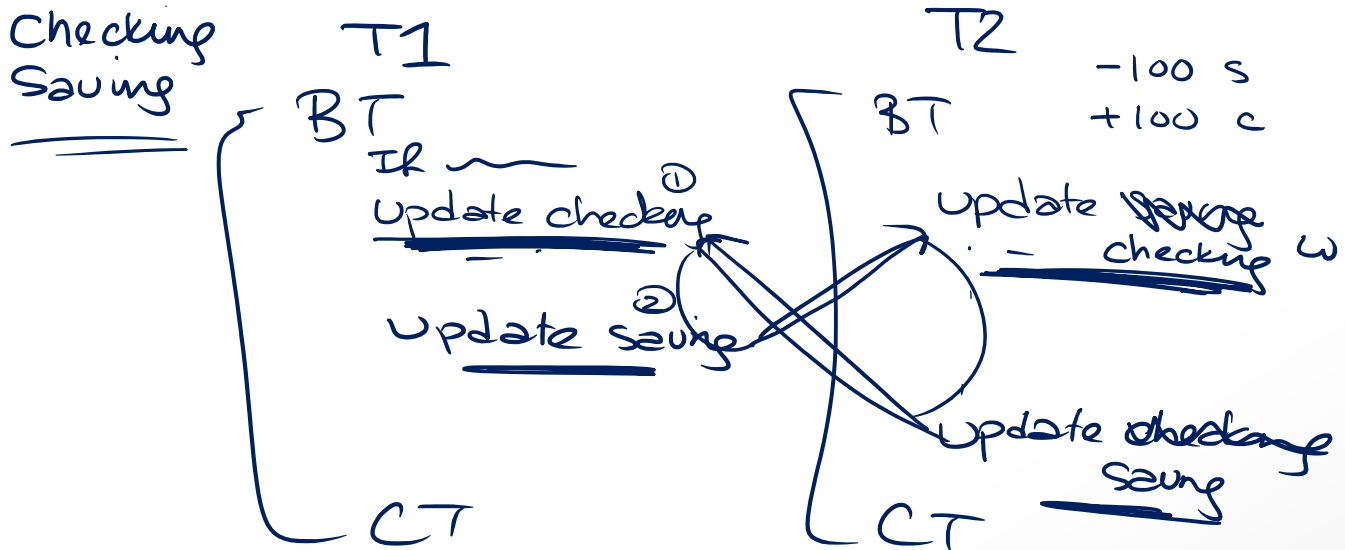
part fn defines logical boundaries
part scheme defines physical locations

This is from the partition-level lock escalation demo

This is from the lock escalation demo. This picture shows how the boundaries define the logical placement of data within the 3 partitions. All this pic is describing (briefly) is that the two RIGHT boundary points of 8000 and 16000 define the breakdown as follows:

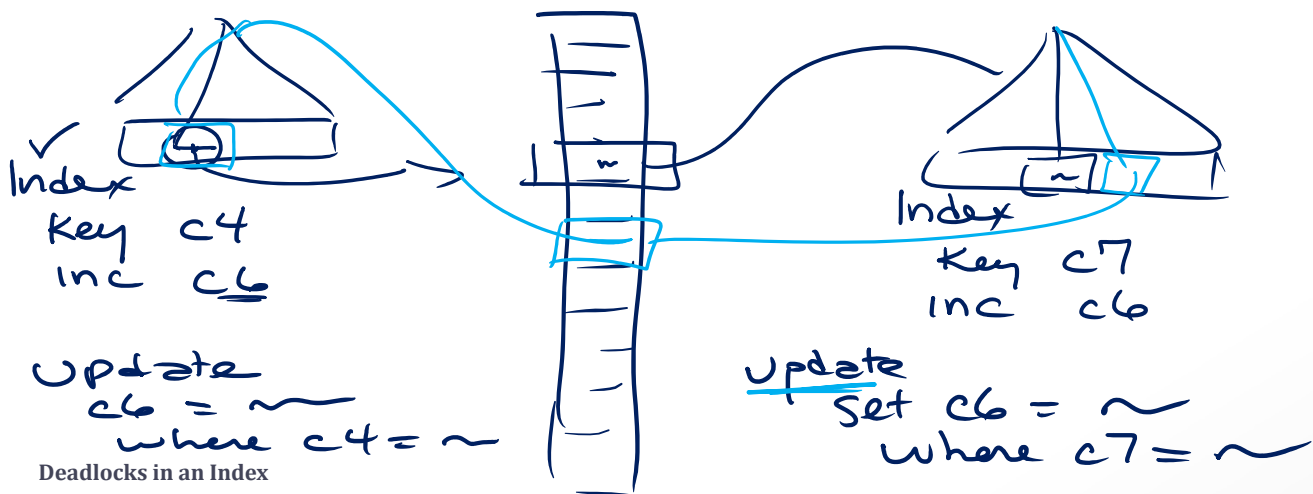
$-\infty$ to 7999	8000 to 15999	16000 to ∞
Partition 1	Partition 2	Partition 3

To highlight what impacts escalation, I used multiple transactions. The 1st affected only rows under 1K. The 2nd affected rows 2500-7600. The 3rd affected 7500-13000. And, the 4th was all under 16K. But, none of them (individually) required ~5000 locks so none escalated.



Reducing deadlocks programmatically by changing access pattern

Deadlocks often occur when resource patterns are interleaved. Where possible, rewriting code can be really helpful. A simple example is "the banking transaction" where one user is moving money from checking to savings and the other from savings to checking. This "crisscross" of activity will be more prone to deadlocking. One option (because this is a transaction) is to reorder the statements of the transactions to always access the tables in a particular order. If this were the case then these deadlocks would be removed and replaced with blocking (not deadlocks).



Deadlocks in an Index

This picture described a deadlock that can occur when multiple indexes exist on a single table and they INCLUDE columns that are being updated.

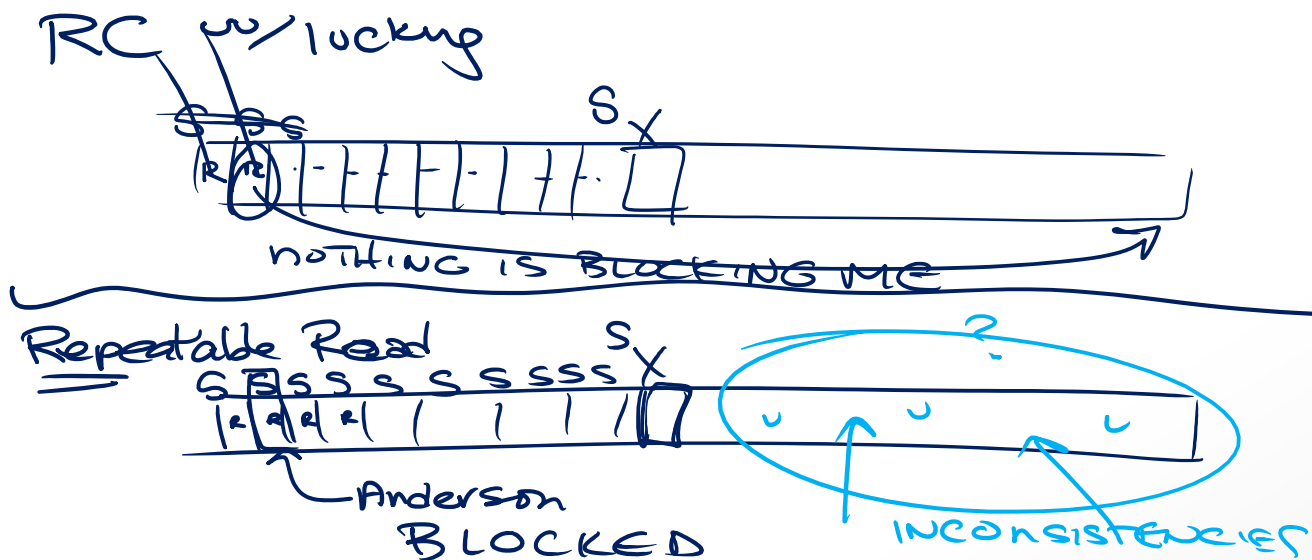
CL = clustered table

Left side = index on col4 INCLUDE (col6)

Right side = index on col7 INCLUDE (col6)

Imagine an update to col6 that uses WHERE col4 = Y running at the same time as an update to col6 that uses WHERE col7 = X

These updates are MUCH more likely to create deadlocks because of how locking works within indexes. To reduce the potential here - I often use DISALLOWPAGELOCKS and ALLOWROWLOCKS. But, you'll need to make sure these are back on for minimally logged operations, truncate table (IF), and index REORG.



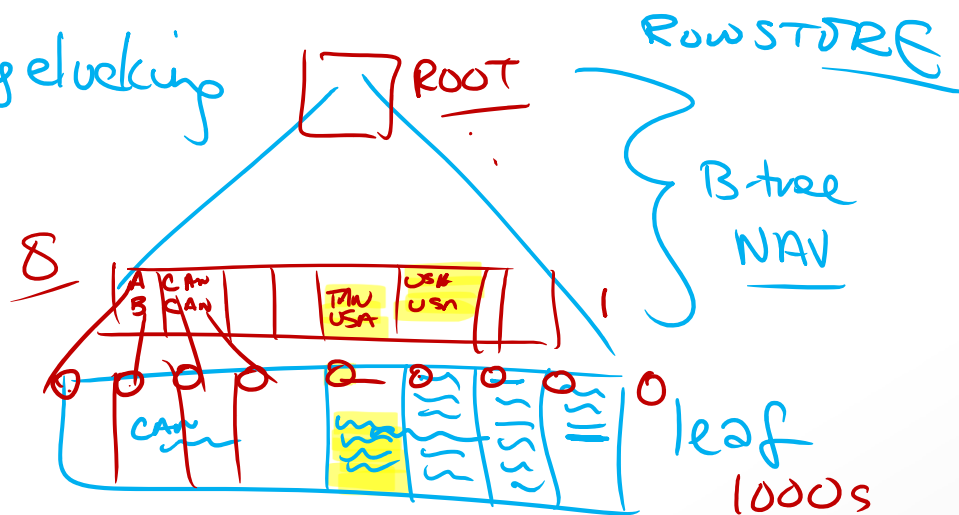
Demo: Why does RC (read committed) have non-repeatable reads?

Because the row-level shared locks are released immediately after the row is read.

Increasing your isolation level to repeatable reads gets rid of this problem by LEAVING the row-level shared locks for the life of the transaction. Guaranteeing that once a row has been read – it will not be able to be changed by another transaction/user.

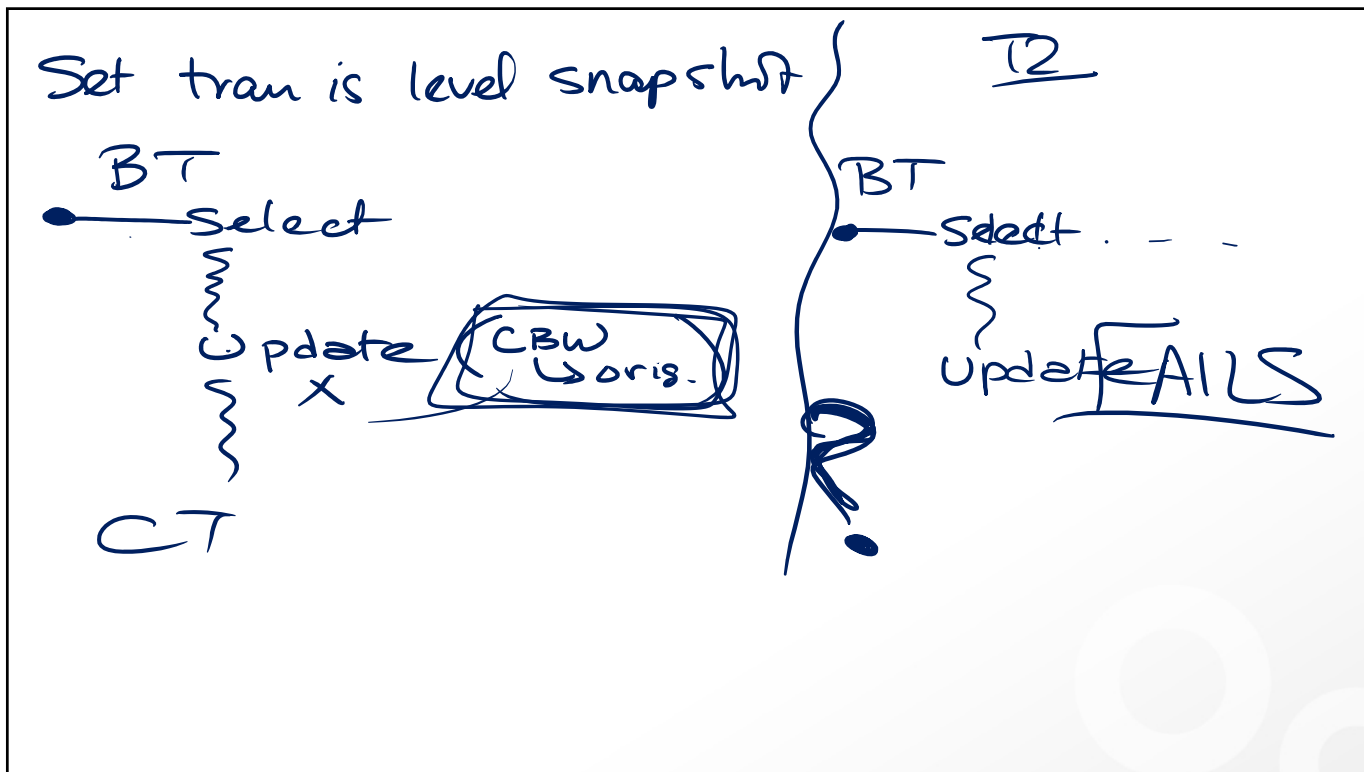
Key-range locking

Simple
Pred



Key-range locking (Serializable transactions)

This picture describes key range locking in an index. When a query runs, the isolation level dictates the state of the data (even in the bounds of a transaction) that can be seen. In a serializable transaction the data is isolated at the time the statement runs. As a result, data must be locked to prevent anomalies. To prevent new rows from coming into the set, SQL Server will lock the “range” of rows affected by the query. If a good index exists then SQL Server can lock within the index (it’s in the first intermediate level – the level above the leaf level). If a good index does not exist then SQL Server must do table-level locking. This example was showing key-range locking in an index for the “country” set. In the end, we might end up locking more data than just that country but only a small amount of data that surrounds the value(s) of interest.



SQLskills ONLINE Immersion Event

Backup Strategy and Internals

(We talked about minimally logged operations and how to deal with log backups)

Kimberly L. Tripp

Kimberly@SQLskills.com



FYI: These were originally from a 4x3 format so they're not perfect in widescreen but the info's still good!
;-)

Trace Flags To Stop Log Dumping

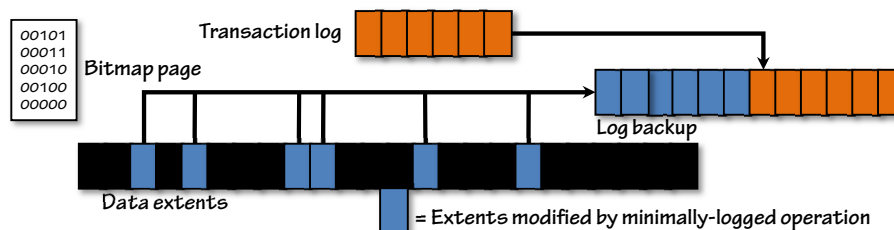
- To change the behavior of BACKUP LOG WITH TRUNCATE_ONLY or NO_LOG:
- In SQL Server 2000
 - Trace flag –T3231
 - In FULL or BULK_LOGGED recovery model they do nothing
 - In SIMPLE recovery model they will clear the log
- In SQL Server 2005
 - Trace flag –T3231 works the same way as in 2000
 - Trace flag –T3031 does a checkpoint
- In SQL Server 2008+
 - Manually clearing the log is no longer possible, but some people may still do it by switching back-and-forth to SIMPLE recovery model

Continuity of the Log Backup Chain

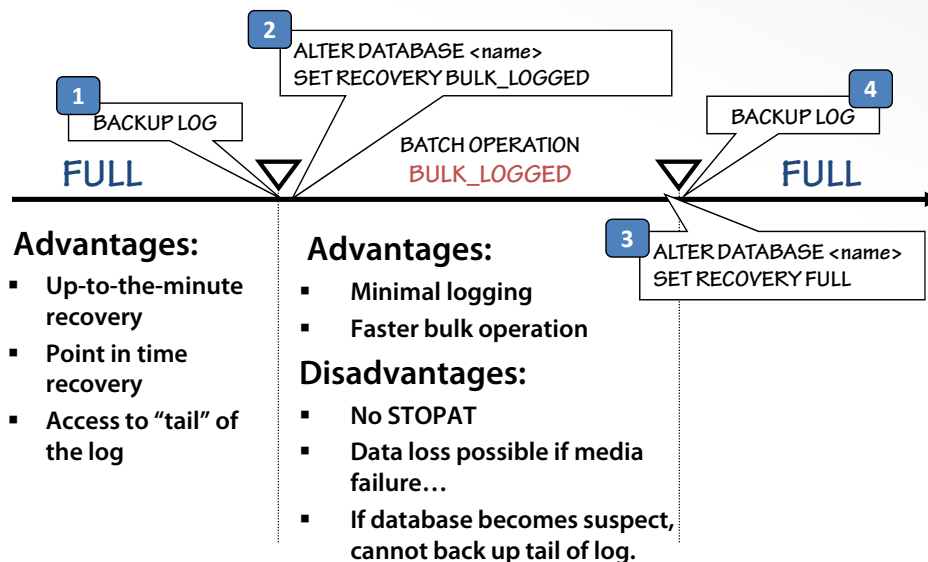
- If the log backup chain is not complete (meaning one or more log backups are damaged or missing) then complete recovery is not possible
 - Must have complete chain of log backups from which to restore
- Consider using mirrored backups of the transaction log and/or storing them on redundant disks
- What breaks the log chain?
 - Clearing the transaction log manually (using WITH TRUNCATE_ONLY or WITH NO_LOG)
 - Changing the database to SIMPLE recovery model (and then changing back)
 - Deleting, overwriting or throwing away a log backup that was not made as COPY_ONLY
 - Reverting from a database snapshot

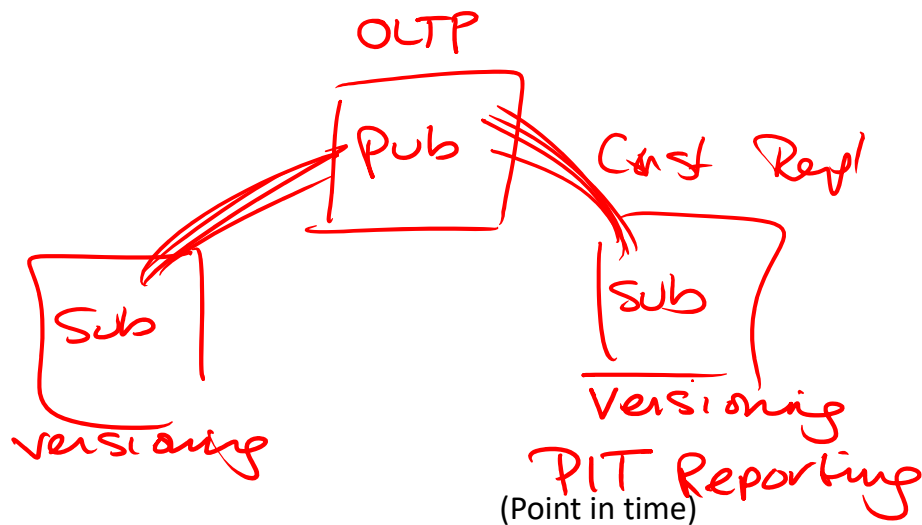
Log Backups After Minimally-Logged Operations

- If there has been a minimally-logged operation since the previous log backup, the next log backup will also back up all data extents modified by the minimally-logged operation
 - This means the minimally-logged operation can be fully reconstituted
- The resulting log backup will be roughly the same size as if the operation was performed in the FULL recovery model
 - A point-in-time restore operation cannot stop at any point between the start and end of the log backup



Switching Recovery Models





An ideal use for versioning – on the SUBSCRIBER (Replication)

The idea is that you don't want to impact your production OLTP environment with READERS or with versioning so instead, you replicate to subscribers and use read committed versioning (read_committed_snapshot) there!

Poor Man's DW

OLTP → B/R

Poor Man's Datawarehousing

Setting a database to RO (read-only) can be a good idea when you plan to use it solely for reads. However, how does SQL Server update statistics or add statistics?
Really, it can't.
So, the main point... if you're going to move a backup to another server for read-only access – you should automate some basic optimizations before setting it to RO.

- ① Update stats ←
 - ② Rebuild Indexes
FF = 100
~~update stats~~ ←
 - ③ sp_createstats
Index only
fullscan - ?
- Sampling?
- READ ONLY