

# Analysing and Resolving Deadlocks



Erland Sommarskog  
SQL Server MVP

<https://sqlb.it/?10606>



# **Erland Sommarskog**

**Independent consultant based in Stockholm**

**SQL Server MVP since 2001**

**esquel@sommarskog.se**

**<http://www.sommarskog.se>**

Slides and scripts are available on  
**<http://www.sommarskog.se/present>**



# Agenda

- Introduction to deadlocks.
- How to get information about deadlocks?
- Understanding the deadlock XML.
- Quick recap on locking.
- Isolation levels and deadlock prevention.
- More means to prevent deadlocks.
- Deadlock retry.



# What Is a Deadlock?

[01\\_FirstWindow.sql](#)  
[01\\_SecondWindow.sql](#)

- A deadlock is when two or more processes are blocking each other in such a way that neither can continue.
- SQL Server checks for this condition every five seconds.
- If a deadlock is found, SQL Server injects an error in one of the processes, the *deadlock victim*, and rolls back its transaction.
  - The deadlock victim is selected based on the amount of log records.



# Are Deadlocks a Problem?

- Deadlocks occur in the best of families.
- An occasional deadlock is no cause for alarm.
- But many deadlocks per day often are.
- Users don't like errors about being “deadlock victims”.
- Too many deadlocks can reduce throughput, since it takes a few seconds until a deadlock is resolved.



# Getting Information about Deadlocks

- Easiest is to query the `system_health` session which is always running.

```
SELECT CAST(event_data AS xml), timestamp_utc
FROM sys.fn_xe_file_target_read_file(
    N'system_health*.xel',
    DEFAULT, DEFAULT, DEFAULT)
WHERE object_name = 'xml_deadlock_report'
ORDER BY timestamp_utc DESC
```

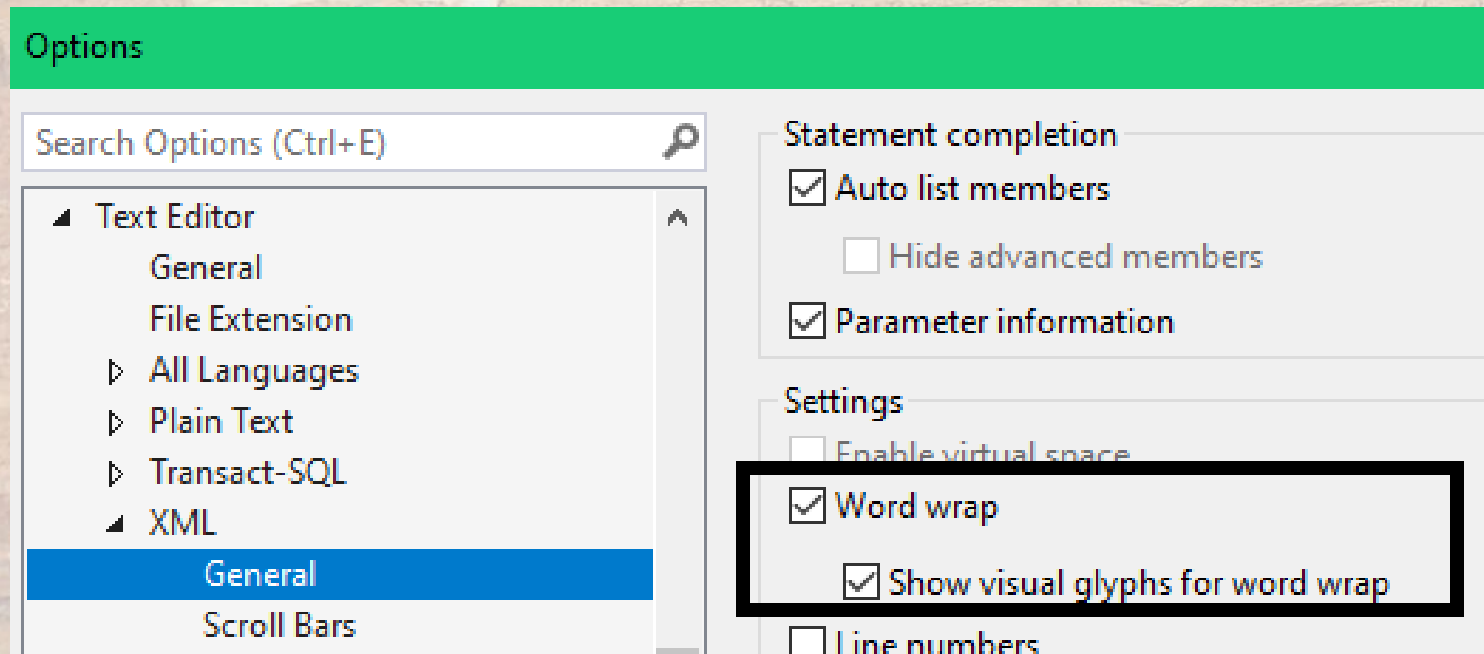
[02\\_query\\_system\\_health.sql](#)

- Observe the asterisk – without it you get no output.
- `timestamp_utc` only available on SQL 2017 and later.



# Tip for SSMS

- To avoid having to scroll sideways, turn on word wrap for XML:





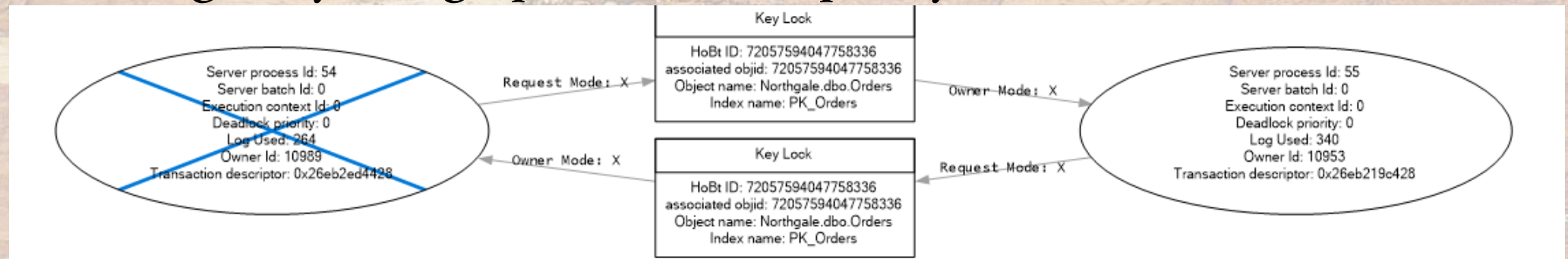
# Defining Your Own XEvent Session

- You find that querying system\_health session is slow.
- On Azure SQL Managed Instance the system\_health session is [currently broken](#).
- In Azure SQL Database there is no system\_health at all.
- In Azure write to BLOB store or ring buffer.
  - Script file has an example query for ring\_buffer.
- Use the event **xml\_deadlock\_report**.
  - In Azure SQL DB, use **database\_xml\_deadlock\_report**.



# Other Means to Get Information

- Turn on TF 1222 to get XML in the SQL Server errorlog.
  - Simple, but litters the errorlog, and information is difficult to read.
- Trace/Profiler. Mainly an option on SQL 2005/2008.
  - Profiler gives you a graph that looks pretty:



- But it's confusing and important information is missing. Discard!
- The TextData column has the actual XML.



# The Deadlock XML

```
<deadlock>
  <victim-list>
    <victimProcess id="process18519e59088" />
  </victim-list>
  <process-list>...</process-list>
  <resource-list>...</resource-list>
</deadlock>
```

- **victim-list** – The process(es) that were rolled back.
- **process-list** – Information about the processes.
- **resource-list** – The locks involved in the deadlock.



# The Process List

```
<process-list>
  <process id="process2b79fb44ca8" ... >
    <stackFrames>...</stackFrames>
    <executionStack>...</executionStack>
    <inputbuf>...</inputbuf>
  </process>
  <process id="process2b792576">...</process>
</process-list>
```

- One tag for each process in the deadlock.
- Many attributes about the processes.



# Nested Elements in the Process Tag

```
<stackFrames>...</stackFrames>  
<executionStack>...</executionStack>  
<inputbuf>...</inputbuf>
```

- **stackFrames** – SQL Server call stack, only of interest to support engineers. (New in SQL 2022).
- **executionStack** and **inputbuf** – information about the submitted command and current statement.



# The inputbuf Tag, take one

```
<inputbuf>  
UPDATE dbo.Orders  
SET     Freight = 17.23  
WHERE   OrderID = 11000  
</inputbuf>
```

- For ad-hoc commands, the `inputbuf` tag often suffices.
- For multi-statement batches, you may have use for the line numbers in the `executionStack` tag.



# The inputbuf Tag, Take Two

- You may be looking at:

```
<inputbuf>  
Proc [Database Id = 2 Object Id = 757577737] </inputbuf>
```

- This is a stored procedure called through RPC.
- You need to look at the **executionStack** tag for more information.



# The executionStack Tag

```
<executionStack>
  <frame procname="tempdb.dbo.inner_sp" line="10" stmtstart="502"
    stmtend="612" sqlhandle="0x03000200ee8...">
INSERT RollYourOwn(id, value) VALUES(@id, @value    </frame>
  <frame procname="tempdb.dbo.outer_sp" line="3" stmtstart="150"
    stmtend="212" sqlhandle="0x0300020027b...">
EXEC inner_sp @value, @id OUTPU    </frame>
</executionStack>
```

- You see the statement of the deadlock, and the call stack.
- In this example: the application called outer\_sp.
- Last character in statement is stripped off on SQL 2014+.



# All the Process Attributes

```
<process id="process26eab72f848" taskpriority="0" logused="244"  
waitresource="KEY: 6:72057594047889408 (fadcdcb5e33c)"  
waittime="2728" ownerId="4260688" transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="0x270d78b8428"  
lockMode="X" schedulerid="1" kpid="9744" status="suspended"  
spid="53" sbid="0" ecid="0" priority="0" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783" lastattention="1900-  
01-01T00:00:00.783" clientapp="slask.pl" hostname="SOMMERWALD"  
hostpid="6632" loginname="PRESENT10\sommar" isolationlevel="read  
committed (2)" xactid="4260688" currentdb="6"  
currentdbname="Northgale" lockTimeout="4294967295"  
clientoption1="671088672" clientoption2="128056">
```



# The Relevant Ones

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerId=""  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="" lockMode=""  
schedulerid="" kpid="" status="" spid="53" sbid="0" ecid="0"  
priority="" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```



# Process Identification

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerId=""  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="" lockMode=""  
schedulerid="" kpid="" status="" spid="53" sbid="0" ecid="0"  
priority="" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```



# Process Identification Details

- `id="process26eab72f848"` Mapping between `process-list`, `resource-list` and `victim-list`.
- `spid="53"` The one we know and love.
- `sbid="0"` Non-zero if MARS is in use.
- `ecid="0"` Non-zero when there is parallelism.
- `currentdbname` and `loginname`. Self-explanatory.
- `clientapp` and `hostname`. Recall that they are set in the connection string.



# transactionname Attribute

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerId=""  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="" lockMode=""  
schedulerid="" kpid="" status="" spid="53" sbid="0" ecid="0"  
priority="" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```



# Multi-Statement Transaction?

- In a multi-statement transaction, locks in the deadlock may come from previous statements in the transaction!
- If **transactionname** reads *user\_transaction*, you have a multi-statement transaction.
- Same if it is *MyTran* or some other name. (Someone said **BEGIN TRANSACTION MyTran**).
- *implicit\_transaction* is also a multi-statement transaction. **SET IMPLICIT\_TRANSACTIONS ON** is in effect.



# Multi-Statement Transaction? cont'd

- If **transactionname** reads SELECT, INSERT, UPDATE, DELETE etc, likely to be an auto-committed statement.
- But check **executionStack** – the deadlock statement could be in a trigger with multiple statements.
  - A trigger always executes in the context of a transaction defined by the statement that fired it.
- **transactionname** can read INSERT EXEC. In this case the procedure runs a multi-statement transaction defined by the INSERT statement.



# Three Important Timestamps

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerId=""  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="" lockMode=""  
schedulerid="" kpid="" status="" spid="53" sbid="0" ecid="0"  
priority="" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```



# Transaction Started by Application?

- Is **lasttranstarted** before **lastbatchstarted**?  
Typically, it's a multi-batch transaction started by the application.
  - But it could be a runaway transaction without a COMMIT.
- Are **lastbatchcompleted** and **lastbatchstarted** far apart in a multi-batch transaction? Investigate!
  - Is it doing work elsewhere which increases the length of the transaction – and thus makes the window for a deadlock wider?



# The lastattention Attribute

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerId=""  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="" lockMode=""  
schedulerid="" kpid="" status="" spid="53" sbid="0" ecid="0"  
priority="" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```



# Attention Signals

- The application can send an *attention signal* to SQL Server.
- On this signal, SQL Server will:
  - Roll back the current statement.
  - Leave any user transaction open (unless XACT\_ABORT is on).
- Common reason is the error *Timeout Expired*. (Most APIs have a default timeout of 30 s.)
- The red button in SSMS also produces an attention signal.



# Unhandled Timeout Errors

- **lastattention** reflects the last time an attention signal was received.
- Is **lastattention** later than **lasttranstarted**? This is a red flag for a runaway transaction!
- Most likely the application has experienced a timeout without rolling back.
- Stop analysing the deadlock. Instead, review the error-handling code in the application.



# Transaction Count

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerId=""  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="" lockMode=""  
schedulerid="" kpid="" status="" spid="53" sbid="0" ecid="0"  
priority="" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```



# The Deceivable trancount Attribute

- For an INSERT, UPDATE, DELETE or MERGE statement, it will always be at least 2, never one or zero.
- And it's 2, regardless if it is a single-statement transaction or a multi-statement transaction.
- For a nested transaction it is 3 or higher.
- A value that does not match the stack depth could indicate a runaway transaction.



# The isolationlevel Attribute

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerT  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.783" XDES="" lockMode=""  
schedulerid="" kpid="" spid="53" sbid="0" ecid="0"  
priority="" tr  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```

Covered later



# The resource-list Element - Overview

```
<resource-list>
  <keylock attr="...">
    <owner-list>...</owner-list>
    <waiter-list>...</waiter-list>
  </keylock>
  <keylock attr="...">
    <owner-list>...</owner-list>
    <waiter-list>...</waiter-list>
  </keylock>
</resource-list>
```

- One element for each lock in the deadlock.
- Name of element depends on type of lock.
- Nested in that element:
  - **owner-list** element listing the owner(s) of the lock.
  - **waiter-list** element listing processes waiting for the lock.



```
<keylock attr="...">
  <owner-list>
    <owner id="process26eabfb68c8" mode="X" />
  </owner-list>
  <waiter-list>
    <waiter id="process26eab72e4e8" mode="S" requestType="wait" />
  </waiter-list>
</keylock>
<keylock attr="...">
  <owner-list>
    <owner id="process26eab72e4e8" mode="S" />
  </owner-list>
  <waiter-list>
    <waiter id="process26eabfb68c8" mode="X" requestType="wait" />
  </waiter-list>
</keylock>
```

A diagram illustrating the mapping of process IDs to XML attributes. A grey rounded rectangle on the right contains the text "Mapping to **id** in the process elements." Four arrows point from this box to the **id** attributes in the XML code: the first arrow points to `process26eabfb68c8` in the first `<owner>` element, the second arrow points to `process26eab72e4e8` in the first `<waiter>` element, the third arrow points to `process26eab72e4e8` in the second `<owner>` element, and the fourth arrow points to `process26eabfb68c8` in the second `<waiter>` element.



# The xxxlock Element

```
<keylock hobtid="72057594047758336" dbid="6"  
objectname="Northgale.dbo.Orders" indexname="PK_Orders"  
id="lock2709eb35780" mode="X"  
associatedObjectId="72057594047758336">
```

- You need to know table and index name. Ignore the rest.
- **keylock** = Row lock in index, clustered or non-clustered.
- **ridlock** = Row lock in heap when accessing data page.
- **pagelock** = What it says, lock on page level.
- **objectlock** = Lock on table level.



# Index Name in Page Locks

- In **pagelock** elements, the index name is missing.
- Work from the **associatedObjectId** attribute with this query:

```
SELECT object_name(p.object_id) AS table_name,  
       i.name AS index_name  
FROM   sys.partitions p  
JOIN   sys.indexes i ON p.object_id = i.object_id  
                        AND p.index_id = i.index_id  
WHERE  p.hobt_id = @associatedObjectId
```



# Deadlocks with Parallelism

- If a deadlock occurs with parallel execution, there is one **process** per thread with same **spid** and different **ecid**.
- The **resource-list** is likely to have **exchangeEvent** elements, because threads are waiting for each other.
- Try to ignore these, and focus on the locks.
- If all processes in the **process-list** have the same **spid**, you have an intra-query deadlock.
  - They are bugs in SQL Server.



# Quick Recap on Locking

- Granularity.
- The most important lock types.
- Intent locks.



# Recap on Locking - Granularity

- For DML operations, locks can be taken on Row, Page or Table level.
  - (And partition, if enabled for the table.)
- The higher the level, the bigger the risk for deadlocks.
- Normally, SQL Server takes row locks, but if it thinks that a query can touch many rows, it may start with page or table locks to preserve memory.



# Lock Escalation

- If a *query* takes more than ~5000 locks on a table, SQL Server attempts to escalate to table level.
- If escalation is blocked by other locks, SQL Server jogs along with the row locks and tries again later.
- Lock escalation is never to page level.
- Thus, if you see page locks in a deadlock, they were there when the query started.



# The Basic Lock Modes

From strongest to weakest:

- Sch-M – Schema-modification locks.
- X – Exclusive locks.
- U – Update locks.
- S – Shared locks.
- Sch-S – Schema-stability locks.



# Schema-Modification Locks (Sch-M)

- Always on table level.
- Taken by DDL statements (ALTER TABLE, CREATE INDEX etc.)
- Incompatible with all other locks.
- Held to the end of transaction.



# Exclusive Locks (X)

- Can be on any level.
- Taken by INSERT, UPDATE, DELETE and MERGE.
- Incompatible with all other locks but Sch-S.
- Held to the end of transaction.



# Schema-Stability Locks (Sch-S)

- Always on table level.
- Taken by SELECT and other read operations in:
  - Snapshot isolation.
  - Read-committed snapshot.
  - Read uncommitted, a.k.a. NOLOCK.
- Compatible with all locks, but Schema-modification.
  - Prevents schema from changing while query is running.
- Held to the end of the query.



# Shared Locks (S)

- Can be on any level.
- Taken by SELECT and other read operations in these isolation levels:
  - Read Committed without snapshot.
  - Repeatable Read.
  - Serializable.
- Incompatible with Sch-M and Exclusive locks.
- Compatible with Sch-S, Update locks and other Shared locks.



# Update Locks (U)

- Can be taken on any level.
- Taken by UPDATE, DELETE and MERGE.
- Compatible with Sch-S and Shared locks.
- Incompatible with Sch-M, Exclusive *and with other* Update locks.
- Purpose: reserve a row for update without blocking readers.
- Converted to Exclusive or released when not needed.



# Intent Locks

- A.k.a “I am here” locks.
- A process wants an exclusive lock on table level. This should be blocked if there is a shared lock on any row.
- Check all rows to see if any is locked?
- That would be highly inefficient!
- Intent locks to the rescue!



# Intent Locks at Work

- Consider: `SELECT Col1, Col2 FROM tbl WHERE keycol = 987.`
- First take an Intent Shared (IS) lock on table level.
- This will be blocked if another process holds a schema-modification (Sch-M) or exclusive (X) lock on table level.
- Once the IS lock is taken, any attempt to take an exclusive table lock will be blocked.
- Repeat on page level.
- Finally, take a Shared (S) lock the row with keycol = 987.



# Intent Locks and Compatibility

- There are three types of Intent Locks:
  - Intent Shared (IS).
  - Intent Update (IU).
  - Intent Exclusive (IX).
- All intent locks are compatible with each other.
- IS is incompatible with X and Sch-M.
- IU is incompatible with U, X and Sch-M.
- IX is incompatible with S, U, X and Sch-M.



# Isolation Levels

- Read Uncommitted.
- Read Committed (the default, has two implementations).
- Snapshot.
- Repeatable Read.
- Serializable.



# Controlling Isolation Levels

- There are two ways.
- By SET command, for instance:  
`SET TRANSACTION ISOLATION LEVEL REPEATABLE READ`
  - This level applies to all queries until there is a new SET TRANSACTION ISOLATION LEVEL command or module exits.
- By hint for a specific table in a query, for instance:  
`SELECT * FROM tbl WITH (SERIALIZABLE)`



# Read Uncommitted

- Also known as “dirty reads”.
- You read from the data rows without taking shared locks.
- You could be reading uncommitted and inconsistent data.
- In query hints often specified as NOLOCK.
- You are still taking Sch-S locks.
  - You can be blocked by index rebuilds and other DDL.



# NOLOCK and Deadlocks

- Sure, NOLOCK can remove plenty of deadlocks.
- ...and your queries will start to produce spurious incorrect results that are difficult to reproduce.
- With NOLOCK you may:
  - Read uncommitted data.
  - Fail to read committed data that was moved due to page splits.
  - Read the same data twice.
- NOLOCK is not a solution to deadlocks, period!
- We will look at a better alternative later.



# Read Committed

- The default isolation level in SQL Server.
- You only read committed data.
- If you read the same row twice in a transaction you can get different results due to intervening updates.
- There are two implementations: Locks and Snapshot.
- Locks is the default in the box product and in Azure MI.
- Snapshot is the default in Azure SQL Database.



# Configuring Read Committed

- To enable/disable Read Committed Snapshot:

```
ALTER DATABASE db SET READ_COMMITTED_SNAPSHOT ON / OFF
```

- Requires exclusive access to the database.
- When disabled, Read Committed is always by Locks.
- When enabled, Read Committed defaults to Snapshot.
- Can be overridden by READCOMMITTEDLOCK hint.
  - There is no SET TRANSACTION ISOLATION LEVEL command.



# Read Committed with Locks

- You read directly from the data rows, taking shared locks. (On row, page or table.)
- Thus, you are blocked by uncommitted updates.
- Locks are released when not needed anymore, often before the end of the query.



# Read Committed Snapshot (RCSI)

- SQL Server reads data from a version store, not from the data rows.
  - Except for target tables for UPDATE, DELETE and MERGE.
- A statement sees a snapshot of the database as it was when the statement started. (...ish)
  - User-defined functions have their own starting points.
- If a row has an uncommitted change, you get the old data.



# RCSI, the Deadlock Killer

- With RCSI no shared locks are taken on rows, only Sch-S locks on table level.
- Thus, with RCSI readers cannot block writers and writers cannot block readers.
- Enabling Read Committed Snapshot is great for removing deadlocks between readers and writers with no code changes!



# RCSI, Technical Caveats

- There is a price: Incurs an overhead to update and delete operations to maintain the version store.
- Version store is in tempdb (unless you have Accelerated Database Recovery turned on).
  - Thus, you may have increase size of tempdb.
- Adds a 14-byte pointer to updated rows, leading to page splits.
  - These pointers are removed on index rebuild!
  - Rebuild indexes with 90% fill factor?



# RCSI, Application Caveats

- You are reading stale data. Does that matter?
  - Example: Cannot inactivate a customer with an active order. An inactivated customer cannot add a new order. This is checked by triggers.
  - Transaction A: `INSERT Orders(CustomerID, ...) VALUES(77, ...)`
  - Transaction B: `UPDATE Customers SET Active = 0 WHERE ID = 77`
  - Trigger on Orders reads from snapshot and sees that customer is active.
  - Trigger on Customers reads from snapshot and sees no active order.
  - Business-rule violation!
- Generally, post/pre-update validations do best in using READCOMMITTEDLOCK hint.
  - Foreign-key validations is always by locks.



# Snapshot Isolation Level

- What you can use if you RCSI makes you nervous.
- In true snapshot isolation, you see the database as it was when the transaction started.
  - And no ...ish this time!
- All reads are from the version store.
  - Including target tables for UPDATE, DELETE and MERGE.
- No shared locks taken, only Sch-S locks on tables.



# Snapshot Isolation, cont'd

- By default, snapshot isolation is not available.
- You need to enable it with  
`ALTER DATABASE db SET ALLOW_SNAPSHOT_ISOLATION ON`
  - Does not require exclusive database access.
- To run a snapshot transaction, you need to say  
`SET TRANSACTION ISOLATION LEVEL SNAPSHOT`
- Because you need to buy in on Snapshot Isolation, you can control where it is used.



# INTERMISSION!

## Analysing and Resolving Deadlocks

<https://sqlb.it/?10606>

Slides and scripts:

<http://www.sommarskog.se/present>

See you again at 14:50!





# Repeatable Read and Serializable

- Both guarantee that once you have read a row, it will not change until you commit.
- Serializable also guarantees that now new rows are added to a range you have read.
- With both you read rows from the data pages, taking shared locks held until the end of the transaction.
- With Serializable you get range locks, e.g., RangeS-S.
- Since locks are held longer, risk for deadlocks increases.



# Reading a Value to Update it Later, I

```
SELECT @val = col FROM tbl WHERE id = @id  
SELECT @val += @incr  
UPDATE tbl SET col = @val WHERE id = @id
```

Transaction A, @id=56, @incr=5.

- Reads current value of col, 10.
- Releases shared lock.
- Computes @val = 15.
- Updates row to 15 and commits.

Transaction B, @id=56, @incr=7.

- Reads current value of col, 10.
- Releases shared lock.
- Computes @val = 17.
- Updates row to 17 and commits.

**WRONG RESULT!**



# Reading a Value to Update it Later, II

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

```
SELECT @val = col FROM tbl WHERE id = @id
```

```
SELECT @val += @incr
```

```
UPDATE tbl SET col = @val WHERE id = @id
```

Transaction A, @id=56, @incr=5.

- Reads current value and retains lock.
- Computes @val = 15.
- Tries to update row, but is blocked by B.

Transaction B, @id=56, @incr=7.

- Reads current value of col, 10 and retains lock.
- Computes @val = 17.
- Tries to update row, but is blocked by A.

**DEADLOCK!**



# Reading a Value to Update Later, III

```
SELECT @val = col FROM tbl WITH (UPDLOCK) WHERE id = @id  
SELECT @val += @incr  
UPDATE tbl SET col = @val WHERE id = @id
```

## Transaction A

- Reads current value of col, 10, retains Update lock
- Computes @val = 15.
- Updates row to 15.
- Commits and releases lock.

## Transaction B

- Tries to read row, but is blocked.
- Reads 15.
- Computes @val = 22.
- Updates row to 22, commits.

**CORRECT!**



# The Power of UPDLOCK

- When reading a value to update it later, use UPDLOCK to
  1. Prevent that another process changes the row.
  2. Prevent deadlocks.
- UPDLOCK implies Repeatable Read.
- When you read a range that you will update, you need to say WITH (UPDLOCK, SERIALIZABLE).



# The isolationlevel Attribute

```
<process id="process26eab72f848" taskpriority="" logused=""  
waitresource="" waittime="" ownerId=""  
transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="" lockMode=""  
schedulerid="" kpid="" status="" spid="53" sbid="0" ecid="0"  
priority="" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783"  
lastattention="1900-01-01T00:00:00.783" clientapp="slask.pl"  
hostname="SOMMERWALD" hostpid="" loginname="PRESENT10\sommar"  
isolationlevel="read committed (2)" xactid="" currentdb=""  
currentdbname="Northgale" lockTimeout="" clientoption1=""  
clientoption2="">
```



# The isolationlevel Attribute

- Should always say Read Committed (or Snapshot).
- That is, don't use SET TRANSACTION ISOLATION LEVEL for Repeatable Read and Serializable.
- Force higher isolation levels with hints for the tables where you transaction semantics calls for it.
- For other tables you will release locks earlier – or use the snapshot.



# Preventing Deadlocks

- READ\_COMMITTED\_SNAPSHOT. (Already covered)
- The UPDLOCK hint. (Already covered)
- Always access resources in the same order.
- Query tuning and indexing.
- Review application behaviour.
- Serialise with application locks.
- Deadlock priority.



# Access Resources in the Same Order

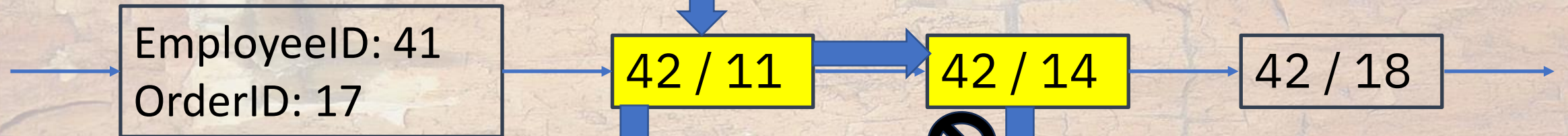
- Standard recommendation. Sounds simple and obvious.
- Can be very difficult to implement in practice.
- Business rules may mandate different access order.
- SQL Server itself does not obey to this rule when building query plans.

[Second\\_window.sql](#)  
[differentaccessorder.xml](#)

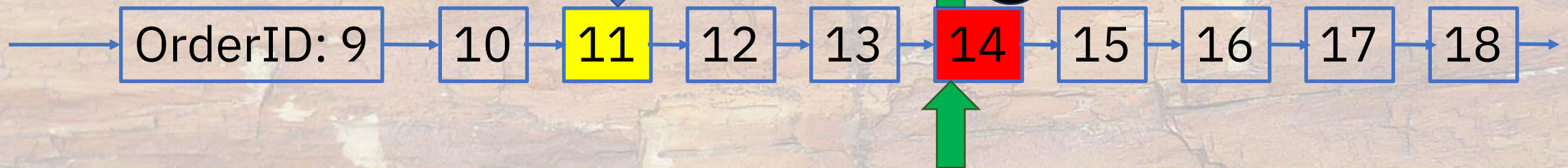


```
SELECT SUM(Freight) FROM Orders WHERE EmployeeID = @empid
```

empid\_ix:



Clustered index:



```
UPDATE dbo.Orders SET EmployeeID = @newempid WHERE OrderID = @orderid
```



# Preventing Deadlocks

- READ\_COMMITTED\_SNAPSHOT.
- The UPDLOCK hint.
- Always access resources in the same order.
- Query tuning and indexing.
- Review application behaviour.
- Serialise with application locks.
- Deadlock priority.



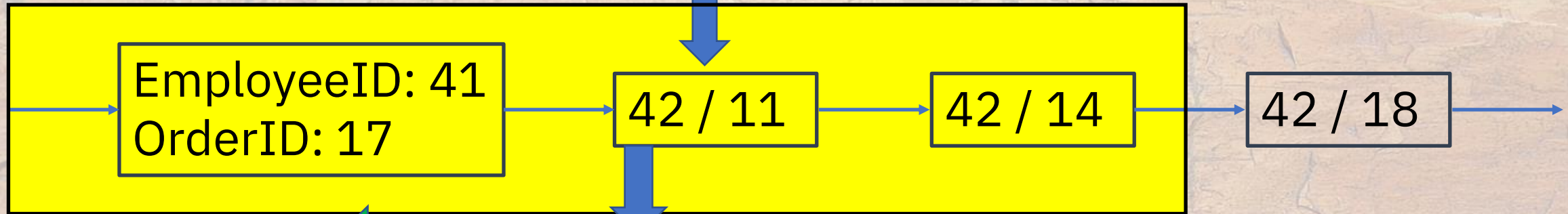
# The Impact of Page and Table Locks

- In the deadlock we just looked at, the processes were fighting about the very same row.
- But imagine that there would only be page locks...

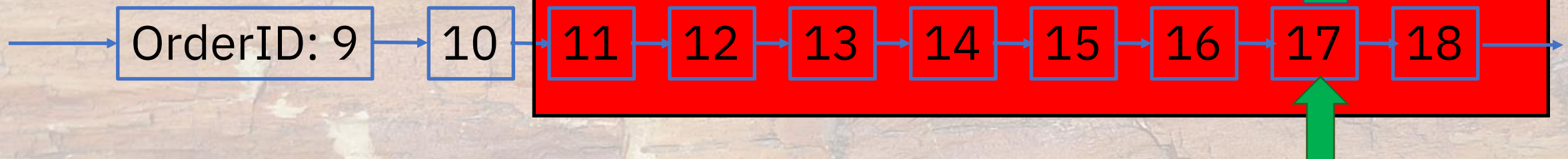


```
SELECT SUM(Freight) FROM Orders WHERE EmployeeID = @empid
```

empid\_ix:



Clustered index:



```
UPDATE dbo.Orders SET EmployeeID = @newempid WHERE OrderID = @orderid
```



# The Impact of Page and Table Locks

- Lower lock granularity increases the risk for deadlocks, as you now can get conflicts on unrelated rows.
- Thus, removing such locks helps to prevent deadlocks.
- Recall that intent locks are always on page/table. They are not a problem.
- It's shared/update/exclusive locks on higher levels you should go after.
- They are often a token of a query that needs tuning.



# Removing Page and Table Locks

- Say that you see this query in a deadlock:

```
SELECT * FROM Orders  
WHERE dateadd(MONTH, 1, OrderDate) > getdate()
```

- Entangling column in an expression, prevents Index Seek.
- This leads to Index Scan which often results in page locks.
- No need to analyse the rest of the deadlock, but rewrite:

```
SELECT * FROM Orders  
WHERE OrderDate > dateadd(MONTH, -1, getdate())
```



# Deadlock with Page Locks

[Second\\_window.sql](#)  
[pagelock-deadlock.xml](#)

```
-- Reader.  
SELECT SUM(Freight) FROM TallOrders WHERE OrderDate = @date  
-- Writer.  
UPDATE TallOrders SET ShipVia = @newshipid WHERE OrderID = @orderid
```

- There is no index with OrderDate as first column, but there is an index on (ShipVia, OrderDate).
- Reader scans this index, taking page locks.
- Parallel plan, thus multiple pages locked simultaneously.



# Deadlock with Page Locks, cont'd

- Writer needs to modify *two* pages in index on ShipVia, one for the old value, one for the new value.
- Thus, writer needs intent locks (IX) on both pages.
- Locks the first page, tries to lock the next but is blocked by a reader thread.
- Another reader thread wants to read rows from first page, but is blocked by writer.
- Possible resolution: add an index on OrderDate.
  - And we could tell this from the start....



# What about the ROWLOCK hint?

- Can be a temporary measure until you get a better solution in place.
- SQL Server chooses page or table locks to preserve memory. Forcing row locks may threaten server stability.
- Therefore, better to review indexes and/or tune queries.
- The ROWLOCK hint does not prevent lock escalation.



# Are Row Locks Permitted?

- If you have table or page locks you cannot understand, run this query:

```
SELECT * FROM sys.indexes
WHERE  object_name(object_id) = 'YourTable'
      AND (allow_row_locks = 0 OR allow_page_locks = 0)
```

- If it returns any rows, enable page and row locks with:

```
CREATE INDEX ix ON tbl(col) WITH
  (ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
   DROP_EXISTING = ON)
```



# Only Row Locks, All Good?

- Even if there are only row locks, query tuning can still be the solution.
- Recall this deadlock:

```
-- Reader
SELECT SUM(Freight) FROM Orders WHERE EmployeeID = @empid
-- Writer
UPDATE Orders SET EmployeeID=@newempid WHERE OrderID=@orderid
```

- Possible resolution: Add Freight as an included column in index on EmployeeID.



# One More Deadlock Case

Consider this:

1. Process P starts a transaction.
2. P updates row(s) in table T.
3. P runs query to populate a temp table for two minutes.
4. Process Q starts SELECT joining S and T, starting from S.
5. The rows in T are updated by P, so Q is blocked.
6. P attempts to update row in S that Q has locked.
7. DEADLOCK!



# The Window for a Deadlock

- What if we can tune P's INSERT query so that it runs 200 ms?
- That does not entirely *prevent* the deadlock.
- But the *window* where the deadlock can occur is drastically reduced.
- Shorter transactions → Less risk for deadlocks.
- ...but that does not mean you should split up a transaction that needs to be atomic in multiple.



# Preventing Deadlocks

- READ\_COMMITTED\_SNAPSHOT.
- The UPDLOCK hint.
- Always access resources in the same order.
- Query tuning and indexing.
- Review application behaviour.
- Serialise with application locks.
- Deadlock priority.



# Think of What You Can Do Differently

- Is application running a multi-batch transaction with a lot of work between the batches?
- Rewrite one-by-one processing into set-based.
- If two operations clash, can one be rescheduled to a different time?
- Disable user updates when you need to run a big import during business hours.
- Split up the transaction into several in a *controlled* way. (Not a step to take lightly!)



# Application Locks

- When you have an operation that cannot run in parallel without clashes, you can serialise with application locks.

```
EXEC sp_getapplock @Resource = 'MyLock',  
                  @LockMode = 'Exclusive'
```

- *MyLock* = name of your lock. Up to 32 characters.
- Name is local to the database.
- Blocks until lock is available.
- Lock is held until end of the transaction.
- Don't use this for occasional deadlocks – this can hamper concurrency in the system.



# Deadlock Priority

- If a background process clashes with end users, you may prefer the background process to be the deadlock victim.
- Have the background process submit this command:

```
SET DEADLOCK_PRIORITY LOW
```

- Rather than LOW, you can give HIGH, NORMAL or a number from -10 to 10. LOW is the same as -5.
- I've never had use for anything but LOW.



# Deadlock Priority in the XML

```
<process id="process26eab72f848" taskpriority="5" logused="244"  
waitresource="KEY: 6:72057594047889408 (fadcdcb5e33c)"  
waittime="2728" ownerId="4260688" transactionname="UPDATE"  
lasttranstarted="2022-01-30T14:57:47.787" XDES="0x270d78b8428"  
lockMode="X" schedulerid="1" kpid="9744" status="suspended"  
spid="53" sbid="0" ecid="0" priority="-5" trancount="2"  
lastbatchstarted="2022-01-30T14:57:47.783"  
lastbatchcompleted="2022-01-30T14:57:47.783" lastattention="1900-  
01-01T00:00:00.783" clientapp="slask.pl" hostname="SOMMERWALD"  
hostpid="6632" loginname="PRESENT10\sommar" isolationlevel="read  
committed (2)" xactid="4260688" currentdb="6"  
currentdbname="Northgale" lockTimeout="4294967295"  
clientoption1="671088672" clientoption2="128056">
```



# Retry on Deadlock

- Rather than just displaying an error and quit, we can re-attempt the operation that caused the deadlock.
- Some people say: Always retry on deadlock.
- I say: *Only* implement retry in situations where deadlocks are a real problem.
- Incorrectly implemented deadlock retry can lead to data errors that are difficult to explain.
- Testing is difficult – you need a deadlock to test.



# Two Key Rules for Deadlock Retries

- Never redo only part of a transaction.
  - **Never do deadlock retry when called inside a transaction.**
  - On deadlock, the *entire* transaction must be rolled back – you cannot only roll back your own part.
- Never redo a committed transaction. That could lead to data being doubled.
  - E.g.: after the transaction there is a SELECT that deadlocks.
- These sort of errors occur only very occasionally and can be very difficult to troubleshoot.

[o3\\_deadlockdemo\\_main.sql](#)  
[o3\\_deadlockdemo2.sql](#)  
[o3\\_deadlockdemo3.sql](#)



# One More Key Rule

- A deadlock retry requires a loop.
- Local variables must be reset at the top of the loop to the value they had before the loop.
- This includes table variables, they are not affected by the rollback!
- Failure to observe this rule can lead to interesting effects, when a variable retains the value from a previous attempt.



# Code Clutter

- Notice how the deadlock retry clutters the code.
- If you don't want the SELECT inside the transaction, you need a new retry loop for this statement. → Even more clutter.
- Difficult to see the business logic forest behind all those deadlock-retry trees.
- Thus, code is more difficult to maintain and more prone to bugs.



# Generic Retry in Data Layer?

- Avoids the code clutter – but opens trapdoors.
- You must still check for open transactions.
- How do you know that all stored procedures run at most one transaction? What if there is code after COMMIT that can deadlock?
- Strict coding guidelines?
- Retry of read-only operations is OK, if you can discern them.



# Retry in the Business Layer?

- If this is where transaction starts, you have no choice.
- You may be back to the code clutter.
- Then again, if you have a background task that runs once a minute – there is an automatic retry.
- And for that matter, telling the user to try again is also a retry...



# Summary: Key Points of this Session

- An occasional deadlock is no cause for alarm.
- Get the deadlock XML from the system\_health session.
- Is any process in a multi-statement transaction? The transaction name will tell you.
- Is any process running a multi-batch transaction? Is **lasttranstarted** before **lastbatchstarted**?
- Remember: The faster your operations are, the smaller the window where a deadlock can occur.
- Only implement deadlock retry if there is no other way out.



# Summary: Don't Shy the Shortcuts!

- Understanding a deadlock in full detail can be hard.
- The good news: you only need to understand as much to make a decision on how to resolve it.
  - Is **lastattention** after **lasttranstarted**? Troubleshoot the unhandled timeout.
  - When you see page and table locks, apply query and index tuning before anything else.
  - Use RSCI / snapshot isolation to isolate readers and writers.
  - Use deadlock priority + deadlock retry for background processes.
  - Use UPDLOCK to avoid conversion deadlocks.



# Summary: Choose Wisely

- There is often more than one way to resolve a deadlock.
- You need to choose wisely.
- The cure can be worse than the disease.
  - Spurious data errors.
  - Serialisation reducing throughput of the system.
  - Etc.
- Take a holistic view of the situation.



# The Last Slide

Erland Sommarskog,  
[esquel@sommarskog.se](mailto:esquel@sommarskog.se)

<https://sqlb.it/?10606>

Slides and scripts on  
<http://www.sommarskog.se/present>.

