

# Deadlocks – Analysing, Preventing and Mitigating

Erland Sommarskog  
SQL Server MVP



# **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>**





Attollo



B3 Consulting Group



Quest Software



SQL Service



**dbWatch**

dbWatch



Transmokopter SQL



Catman Solution



DB24



**redgate**

Redgate Software

# Agenda

- Introduction to deadlocks.
- How to get information about deadlocks?
- Understanding the deadlock XML.
- Means to prevent deadlocks.
- Ways to mitigate deadlocks.



# 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 process, the *deadlock victim*, which rolls back the 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 get upset if they get errors about being the deadlock victim, not the least if they get them daily.
- Too many deadlocks can hamper 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.



# Azure SQL Database

- For Azure SQL Database, run this query in your master database:

```
; WITH CTE AS (  
    SELECT CAST(event_data AS xml) AS xml, timestamp_utc  
    FROM sys.fn_xe_telemetry_blob_target_read_file(  
        'dl', NULL, NULL, NULL))  
SELECT timestamp_utc, xml.query(  
    '/event/data[@name="xml_report"]/value/deadlock' )  
FROM CTE  
ORDER BY timestamp_utc DESC
```

- See the [script file](#) for how to filter on database.



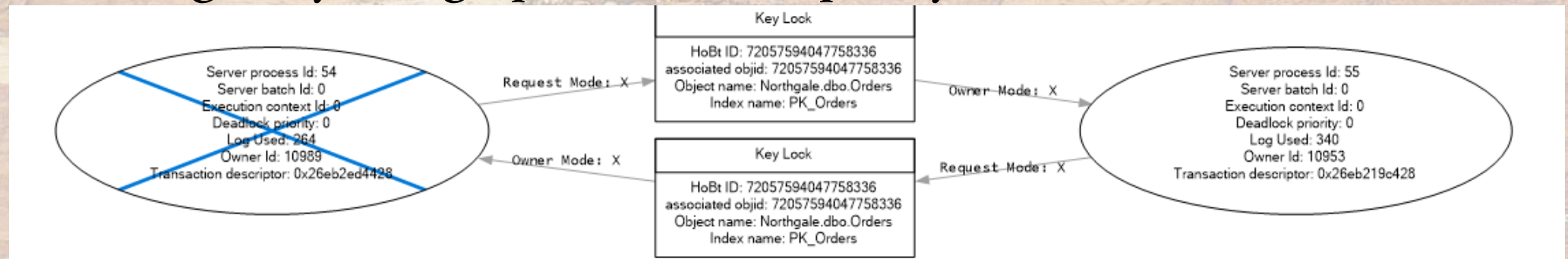
# Defining Your Own XEvent Session

- Include the event `xml_deadlock_report`.
- Useful if:
  - You are on Azure SQL Managed Instance (write to BLOB store).
    - On MI you can only access `system_health` through ring buffer which is unreliable.
  - You find that querying `system_health` session is slow.
  - You want low latency.
- Query it in the same way as the `system_health` session.  
(Don't forget the asterisk!)
- See the [script file](#) for example.



# 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:

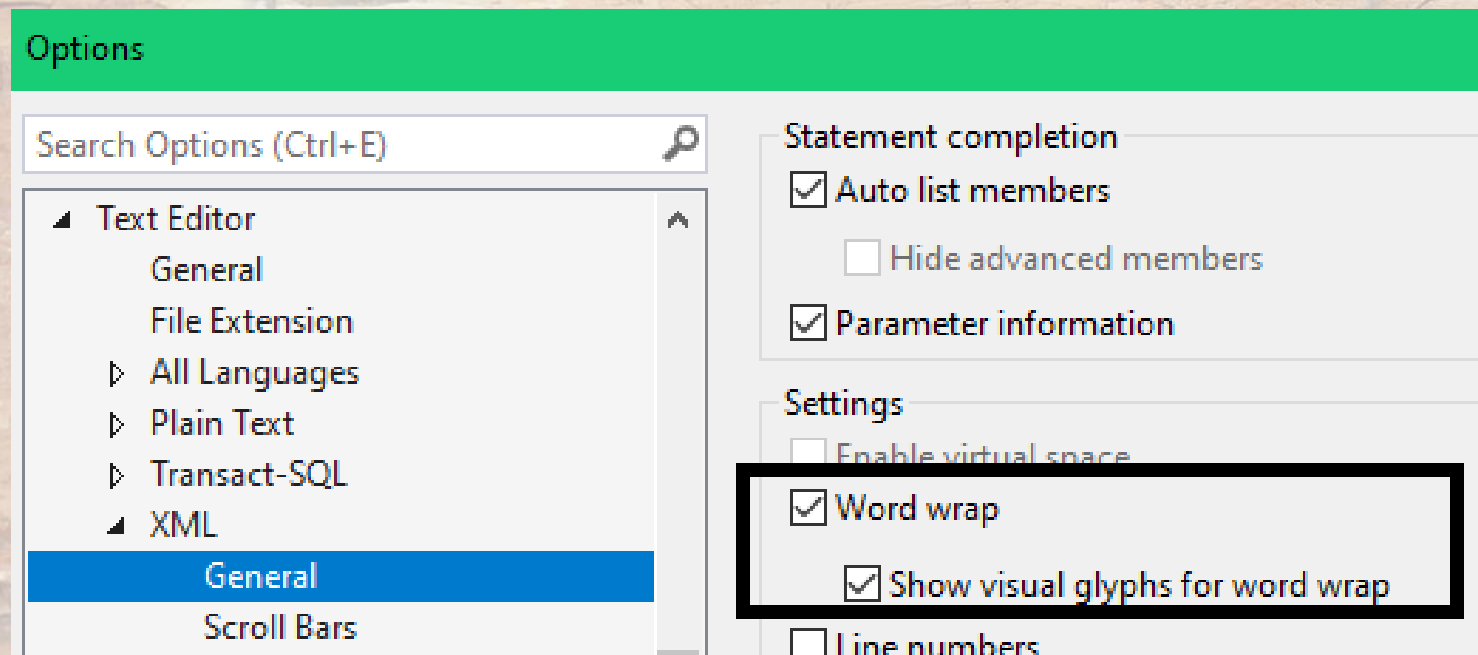


- But it's confusing and important information is missing. Discard!
- The TextData column has the actual XML.



# Tip for SSMS

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





# The Deadlock XML

```
<deadlock>
  <victim-list>
    <victimProcess id="process227300e1088" />
  </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="process227300e1088" taskpriority="0" ... >
    <executionStack>...</executionStack>
    <inputbuf>...</inputbuf>
  </process>
  <process id="process227398ae" taskpriority="0">...</process>
</process-list>
```

- One tag for each process in the deadlock.
- Many attributes about the process.
- **executionStack** and **inputbuf** give 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 is often enough.
- For multi-statement batches, you may have to look at the **executionStack** tag. (Look at line numbers.)



# The inputbuf Tag, Take Two

- You may be looking at:

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

- In this case, it's 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="9" stmtstart="444"
    stmtend="540" sqlhandle="0x03000200d...">
INSERT rollyourown(id, value) VALUES(@id, @value    </frame>
  <frame procname="tempdb.dbo.outer_sp" line="5" stmtstart="254"
    stmtend="316" sqlhandle="0x0300020009...">
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?

- With 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**).
- Also *implicit\_transaction* is 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 making 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="">
```



# Unhandled Timeout Errors

- Is **lastattention** is later than **lasttranstarted**? This is a red flag for a runaway transaction!
- Most likely the application has experienced the error *Timeout expired* without rolling back.
  - On this error, the API sends an attention signal to SQL Server which rolls back current statement and then stops executing. Any open transaction is *not* rolled back (unless XACT\_ABORT is on).
- If you see this, 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.
- An high value could indicate a runaway transaction.
  - High value = not really matching the stack depth.
  - Missing commit or unhandled timeout.



# 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="">
```



# Isolation Levels and Shared Locks

- With default Read Committed, shared locks are released when not needed any more.
- If you see Repeatable Read or Serializable, shared locks are held to the end of the transaction.
- Locks being held longer → higher risk for deadlock.
- Some transaction APIs turn on Serializable by default.
- Keep in mind that the setting may change during the transaction or be overridden by hints.



# 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>
```

Mapping to **id** in the process elements.



# The xxxlock Element

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

- All you need to know is 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 query with a parallel plan is the deadlock, 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.



# Preventing Deadlocks

- Always access resources in the same order.
- Query tuning and indexing.
- READ\_COMMITTED\_SNAPSHOT.
- Lock hints.
- Review application behaviour.
- Serialise with application locks.



# 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](#)



# Access-Order Deadlock

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

- Reader locks row in index on EmployeeID.
- Writer updates employee in data page, exclusive lock.
- Reader needs to read Freight from data page, is blocked.
- Writer needs to update EmployeeID in index, is blocked.
- DEADLOCK!



# Preventing Deadlocks

- Always access resources in the same order.
- Query tuning and indexing.
- READ\_COMMITTED\_SNAPSHOT.
- Lock hints.
- Review application behaviour.
- Serialise with application locks.



# Page Locks and Table Locks

- If a deadlock only has row locks, the processes are fighting about the very same row.
- With page or table locks, processes may be working on unrelated rows that are on the same page or table.
- The lower granularity increases the risk for deadlocks.
- Thus, removing such locks helps to prevent deadlocks.
- Intent locks (IX, IS etc) on table and page level are normal. It's S, X, U locks etc 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.
  - Index Scan 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....



# 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?

- If the deadlock only has row locks, there can still be need for query tuning.
- 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. Q needs to read row in T that P has updated and is blocked.
6. Process P attempts to update row in S that Q has a lock on.
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

- Always access resources in the same order.
- Query tuning and indexing.
- **READ\_COMMITTED\_SNAPSHOT.**
- Lock hints.
- Review application behaviour.
- Serialise with application locks.



# READ\_COMMITTED\_SNAPSHOT

- You can eliminate all deadlocks between writers and readers in just three statements:

```
ALTER DATABASE db SET SINGLE_USER WITH ROLLBACK IMMEDIATE  
ALTER DATABASE db SET READ_COMMITTED_SNAPSHOT ON  
ALTER DATABASE db SET MULTI_USER
```

- Readers will read from a version store to get was committed before any new transaction started.
- Therefore readers do not block writers and vice versa.
- Applies to default isolation level READ COMMITTED.



# RCSI, Technical Caveats

- Incurs an overhead to update and delete operations, including adding a 14-byte pointer to updated rows.
- The version store is normally maintained in tempdb, so you may have to increase tempdb size.
- When Accelerated Database Recovery is enabled, version store is in the database itself.



# RCSI, Application Caveats

- You are reading stale data. Does that matter?
  - Example: Cannot have an inactivated securable in a portfolio. Cannot inactivate a securable if it's in at least one portfolio. This is checked by triggers.
  - Transaction A: adds a securable to a portfolio.
  - Transaction B: inactivates the same securable.
  - Triggers read from snapshot and finds that all good.
  - Business-rule violation!
- Can be avoided by READCOMMITTEDLOCK hint.
  - FK Validations always use locks.



# RCSI - Alternative

- If you are nervous about RCSI, you can do

```
ALTER DATABASE db SET ALLOW_SNAPSHOT_ISOLATION ON
```

- And in places where it is safe to read stale data, you add

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```



# Preventing Deadlocks

- Always access resources in the same order.
- Query tuning and indexing.
- READ\_COMMITTED\_SNAPSHOT.
- Lock hints.
- Review application behaviour.
- Serialise with application locks.



# Lock Hints

- UPDLOCK hint is useful to prevent conversion deadlocks.
  - That is, if you read a row to update it later, apply UPDLOCK rather than REPEATABLE\_READ or SERIALIZABLE.
- ROWLOCK – can be a temporary measure to deal with page/table locks.
  - Locks consume memory, so forcing row locks can threaten server stability.
- NOLOCK. Prevents deadlocks – and instead you get nastier concurrency problems like incorrect results.



# Review Application Behaviour

- 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 processes clash, can one be rescheduled to a different time?
- Generally, be open-minded and think about what you can do differently.



# 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.
- 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 Mitigation

- Deadlock priority.
- Retry on deadlock.
- Lock timeouts.



# 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="" 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 retries 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.



# Where to Implement Deadlock Retry

Inside a stored procedure?

- Code can become very cluttered with business logic hidden behind all the retry code.

Generic retry in the data layer?

- How do you know that the procedure you retry does not run multiple transactions?

Business layer?

- Hm, maybe. Clear everything and start over. Can still be code clutter, though.



# Further Reading

- For further discussion and example on how to implement deadlock retry in a stored procedure, see [section 4.4](#) in Part Three in my series *Error and Transaction Handling in SQL Server*.



# Why Lock Timeouts?

- Background processes can lower their deadlock priority and implement good retry.
- This saves more important processes from being deadlock victims.
- Still, they can be held up for five seconds – that may be bad enough. (Reduced throughput, irritated users etc.)
- What if the background process could step out of the clash at an early stage?



# SET LOCK\_TIMEOUT

```
SET LOCK_TIMEOUT 100
```

- If blocked by a lock, wait 100 ms to get the lock.
- When timeout expires, error 1222 is raised.

Msg 1222, Level 16, State 51, Line 3  
Lock request time out period exceeded.

- On this error, rollback and retry (after a short wait).
- SET LOCK\_TIMEOUT 0 = don't wait. -1 = wait forever.
- There is a **lockTimeout** attribute in the deadlock XML.



# Summary I

- An occasional deadlock is no cause for alarm.

When analysing the deadlock XML:

- Is any process in a multi-statement transaction?
- Is any process running a multi-batch transaction? That is, is **lasttranstarted** before **lastbatchstarted**?
- Ignore **exchangeEvent** elements in the **resource-list**.
- Remember: The faster your operations are, the smaller the window where a deadlock can occur.



# Summary II

- Understanding a deadlock in full detail can be hard.
- The good news: you only need to understand as much to make a decision on prevention or mitigation.
  - Is **lastattention** after **lasttranstarted**? Troubleshoot the unhandled timeout.
  - When you see page and table locks, apply query and index tuning before anything else.
  - READ\_COMMITTED\_SNAPSHOT to isolate readers and writers.
  - Use deadlock priority + deadlock retry for background processes.
  - When things should not run in parallel, serialise or separate.



# Summary III

- When deciding on how to prevent or mitigate a deadlock: 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)

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