Module 5

Concurrency and transaction handling

Module Overview

- Concurrency and transactions
- Locking internals

Lesson: Concurrency and transactions

- Concurrency problems
- Transaction isolation levels
- Working with row versioning
- Transactions
- Working with transactions

Concurrency problems

- Protected from by default
  - Dirty read
    - Uncommitted data is included in results
  - Double read
    - Data in a range is read twice because the rows moves during the read
  - Missing rows
    - Opposite of above, data is moved during read so that rows are missed
- Not protected from by default
  - Non-repeatable read
    - ”Inconsistent analysis”
    - Data changes between two identical SELECT statements within a transaction
  - Phantom read
    - Data inserted into range after read

Här är en genomgång av de olika *concurrency problems* som nämns i bilden, tillsammans med exempel på varje:

Protected from by default

Dessa problem hanteras automatiskt av SQL Server med standardinställningarna.

1.Dirty Read

1. **Problem:** En transaktion läser data som en annan transaktion har ändrat men ännu inte committat. Om den andra transaktionen sedan rullas tillbaka, har den första transaktionen läst ogiltig data.
2. **Exempel:**
    1. Transaktion A uppdaterar en kunds saldo men gör inte commit.
    2. Transaktion B läser saldot och agerar på det.
    3. Transaktion A gör rollback – nu har Transaktion B använt ogiltig data.

2.Double Read

1. **Problem:** En SELECT-läsning av ett dataområde fångar samma rad två gånger om rader flyttas under läsningen.
2. **Exempel:**
    1. En SELECT-läsning hämtar rader i en tabell sorterad på ID.
    2. En annan transaktion flyttar en rad längre ner under läsningen.
    3. SELECT-läsningen läser samma rad två gånger.

3.Missing Rows

1. **Problem:** Motsatsen till double read – rader flyttas under läsningen så att vissa rader aldrig läses.
2. **Exempel:**
    1. En SELECT börjar läsa en tabell där rader flyttas uppåt av en annan transaktion.

2. Rader som först låg längre ner i läsningen har nu passerats och missas helt.

Not protected from by default

Dessa problem kräver högre isoleringsnivåer för att förhindras.

4.Non-repeatable Read

4. **Problem:** En SELECT-fråga körs två gånger inom samma transaktion, men datan ändras mellan körningarna.
5. **Exempel:**
    4. Transaktion A läser en kunds saldo.
    5. Transaktion B uppdaterar och committar ett nytt saldo.
    6. Transaktion A läser igen – nu får den ett annat värde än första gången.

5.Phantom Read

4. **Problem:** En SELECT-fråga får olika antal rader vid två körningar eftersom en annan transaktion har infogat eller tagit bort rader.
5. **Exempel:**
    4. Transaktion A kör SELECT * FROM Orders WHERE Amount > 1000.
    5. Transaktion B lägger till en ny order med belopp >1000 och committar.
    6. Transaktion A kör sin SELECT igen och ser en ny rad som inte fanns vid första körningen.

För att skydda sig mot de sista problemen kan man använda **högre isoleringsnivåer** som *Repeatable Read*, *Serializable* eller *Snapshot Isolation* beroende på behovet.

The ACID Properties of Statements & Transactions
https://sqlperformance.com/2014/02/t-sql-queries/confusion-caused-by-trusting-acid

Transaction Isolation Levels (ODBC)
https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-ver15

Dirty read, Non-repeatable read, and Phantom read
https://jennyttt.medium.com/dirty-read-non-repeatable-read-and-phantom-read-bd75dd69d03a

Transaction isolation levels

- Pessimistic isolation levels:
  - READ UNCOMMITTED
  - READ COMMITTED
    - READ_COMMMITTED_SNAPSHOT OFF
  - REPEATABLE READ
  - SERIALIZABLE
- Optimistic (row versioning) isolation levels:
  - READ COMMITTED
    - READ_COMMMITTED_SNAPSHOT ON
  - SNAPSHOT
    - Be aware of possibilities for update conflict errors

Denna slide handlar om **transaction isolation levels** i SQL Server, vilket påverkar hur transaktioner ser data och låser rader.
Pessimistiska isoleringsnivåer (låsbaserade)
1.READ UNCOMMITTED – Ser osparade ändringar (dirty reads).
2.READ COMMITTED – Ser endast committade data, använder delade lås.
3.REPEATABLE READ – Låser lästa rader för att förhindra att andra ändrar dem.
4.SERIALIZABLE – Låser hela sökintervallet, förhindrar både insättning och ändring.
Optimistiska isoleringsnivåer (row versioning)
1.READ COMMITTED SNAPSHOT (RCSI) – Undviker lås genom att använda versionshantering.
2.SNAPSHOT – Ger en konsekvent bild av data, men kan orsaka **update conflict errors**.
Exempel: Skapa en databas och tabell
CREATE DATABASE TestDB; GO USE TestDB; GO CREATE TABLE Customers (ID INT PRIMARY KEY, Name NVARCHAR(50)); INSERT INTO Customers VALUES (1, 'Alice'), (2, 'Bob'); **Testa olika isolation levels**
READ UNCOMMITTED (dirty reads)
Session 1:
BEGIN TRAN UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; -- Vänta utan commit för att simulera osparad ändring **Session 2:**
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SELECT * FROM Customers; -- Kan se "Charlie" trots att transaktionen inte är committad **SERIALIZABLE (förhindrar ny insättning)**
Session 1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN TRAN SELECT * FROM Customers WHERE ID = 1; -- Behåller transaktionen öppen **Session 2:**
INSERT INTO Customers VALUES (3, 'David'); -- Blockeras av Session 1 **Aktivera SNAPSHOT isolation**

ALTER DATABASE TestDB SET ALLOW_SNAPSHOT_ISOLATION ON; Nu kan transaktioner läsa gamla versionshanterade rader utan att låsas!
Detta visar hur olika nivåer påverkar **läsningar, låsningar och konkurrenshantering** i SQL Server. 🚀

SQL Server Isolation Levels : A Series
https://sqlperformance.com/2014/07/t-sql-queries/isolation-levels

SET TRANSACTION ISOLATION LEVEL (Transact-SQL)
https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql

Working with row versioning

- Row versioning benefits:
  - Fewer locks
  - Less blocking
- Row versioning aspects:
  - Versioning adds load to tempdb
  - Applications need to handle update conflicts for true SNAPSHOT
- Other considerations:
  - Lock hints still apply
  - Writers still block writers
  - Setting READ_COMMITTED_SNAPSHOT ON requires zero connections to the database

Förklaring av Radversionering i SQL Server

SQL Server stöder radversionering för att öka samtidighet och minska låsning. Det finns två huvudtyper:

1.Snapshot Isolation (SI)

1. Varje transaktion ser en **konsekvent version** av data från transaktionsstart.
2. **Läsningar blockeras inte av skrivningar**, men kan få **update conflicts**.
3. Måste aktiveras **explicit per transaktion**.

2.Read Committed Snapshot Isolation (RCSI)

1. Liknar **Snapshot Isolation**, men fungerar automatiskt på **databasnivå**.
2. **Läsningar ser den senaste committade versionen**, utan att blockeras av skrivare.
3. **Aktiveras på databasenivå** och gäller för alla READ COMMITTED-transaktioner.

Kodexempel för båda metoderna

1. Aktivera Snapshot Isolation

ALTER DATABASE TestDB SET ALLOW_SNAPSHOT_ISOLATION ON; GO **Användning i transaktion**: •SET TRANSACTION ISOLATION LEVEL SNAPSHOT; BEGIN TRAN SELECT * FROM Customers; -- Ser en stabil version av datan vid transaktionsstart WAITFOR DELAY '00:00:05'; -- Simulerar fördröjning UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; -- Kan orsaka update conflict COMMIT; Om någon annan transaktion ändrar Customers innan **denna transaktion committar**, uppstår en **update conflict**.

2. Aktivera Read Committed Snapshot Isolation (RCSI)

ALTER DATABASE TestDB SET READ_COMMITTED_SNAPSHOT ON; GO **Användning i transaktion**: •SET TRANSACTION ISOLATION LEVEL READ COMMITTED; BEGIN TRAN SELECT * FROM Customers; -- Ser senaste committade version utan att blockeras WAITFOR DELAY '00:00:05'; -- Simulerar fördröjning SELECT * FROM Customers; -- Ser den senaste uppdaterade versionen om

någon har committat COMMIT; Här finns **inga update conflicts**, men transaktionen ser den senaste **committade versionen**.

Sammanfattning

•**Snapshot Isolation (SI)** – **Konsekvent snapshot vid transaktionsstart**, men risk för update conflicts.

•**RCSI** – **Senaste committade versionen**, ingen risk för konflikter, men gäller för hela databasen.

Båda metoderna ökar samtidigheten men kräver **tempdb-lagring** för äldre versioner av rader. 🚀

SQL Server stöder radversionering för att hantera samtidighet utan att låsa rader. Det finns två huvudtyper: Snapshot Isolation och Read Committed Snapshot Isolation (RCSI). Snapshot Isolation lagrar en version av dataraden vid transaktionsstart, vilket möjliggör läsningar utan att vänta på låsupplösning. Detta kräver dock explicit aktivering på transaktionsnivå. RCSI fungerar liknande men är inställt på databasenivå och tillämpas automatiskt, vilket gör att läsare inte blockeras av skrivare genom att använda en versionerad rad från tempdb. Båda metoderna ökar samtidigheten men kan öka resursanvändningen eftersom äldre versioner av rader måste lagras.

Row Versioning Concurrency in SQL Server
https://www.red-gate.com/simple-talk/databases/sql-server/t-sql-programming-sql-server/row-versioning-concurrency-in-sql-server/

Transactions

- A logical unit of work, made up of one or more Transact-SQL statements
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability
- Transaction management modes:
  - Auto-commit
  - Explicit transactions
  - Implicit transactions

Demo Transaction semantics

ACID-egenskaper i transaktioner

ACID står för **Atomicity, Consistency, Isolation och Durability**, och säkerställer att databastransaktioner är **pålitliga** och **felfria**.

1.Atomicity (Atomisitet)

1. En transaktion är **"allt eller inget"** – antingen genomförs alla ändringar eller så återställs de.
2. Exempel: BEGIN TRAN UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; ROLLBACK; -- Återställer ändringen

2.Consistency (Konsistens)

1. Transaktionen lämnar databasen i ett **giltigt tillstånd**. Inga constraint-brott får ske.
2. Exempel: INSERT INTO Orders (OrderID, CustomerID) VALUES (1, NULL); -- Misslyckas om CustomerID inte får vara NULL

3.Isolation (Isolering)

1. Skyddar transaktioner från att påverkas av andra samtidigt körande transaktioner.
2. Används med **isoleringsnivåer** (t.ex. READ COMMITTED, SNAPSHOT).
3. Exempel: SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN TRAN; SELECT * FROM Customers; -- Ingen annan kan ändra de lästa raderna innan COMMIT eller ROLLBACK

4.Durability (Varaktighet)

1. När en transaktion har **committats**, finns ändringarna kvar även vid systemkrasch.
2. SQL Server säkerställer detta genom att skriva loggar till **transaction log** (LDF-filen).

Auto-commit i SQL Server

•SQL Server använder **auto-commit-läge som standard**.

•**Varje individuellt SQL-kommando** körs som en transaktion och committas direkt.

•Om en fråga misslyckas, rullas den tillbaka automatiskt.

•**Exempel på auto-commit (ingen explicit transaktion krävs):** UPDATE Customers SET Name = 'David' WHERE ID = 2; -- Committas direkt
•Nackdel: **Ingen kontroll över rollback**, vilket gör att man istället kan använda **explicit transactions** (BEGIN TRAN / COMMIT / ROLLBACK) vid kritiska operationer.
Sammanfattning
•**ACID** säkerställer **stabilitet och säkerhet** i SQL Server-transaktioner.
•**Auto-commit** hanterar transaktioner **implicit**, men saknar rollback-kontroll.
•För **viktiga operationer** bör man använda **explicit transactions** för att undvika oförutsedda problem. 🚀

The ACID Properties of Statements & Transactions
https://sqlperformance.com/2014/02/t-sql-queries/confusion-caused-by-trusting-acid

Working with transactions

- Naming transactions:
  - Label only; no effect on code
- Nesting transactions:
  - Only the state of the outer transaction has any effect
  - @@TRANCOUNT track transaction nesting
- Terminating transactions:
  - Resource error
  - SET XACT_ABORT
  - Connection closure
- Transaction best practices:
  - Keep transactions as short as possible

Demo Concurrency

Här kommer en enkel genomgång av bilden med kodexempel för varje punkt.

1. Naming Transactions (Namnge transaktioner)

•Namn på transaktioner är **bara etiketter** och påverkar inte transaktionens beteende.

•Exempel: BEGIN TRAN MyTransaction UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; COMMIT TRAN MyTransaction;

- Namnet MyTransaction är **bara en etikett** och påverkar inte commit eller rollback.

2. Nesting Transactions (Nästa transaktioner)

•Bara den yttre transaktionen styr commit/rollback.

•@@TRANCOUNT visar djupet på transaktionsnästning.

•Exempel: PRINT 'Transaction count: ' + CAST(@@TRANCOUNT AS NVARCHAR); BEGIN TRAN OuterTran; PRINT 'After BEGIN OuterTran: ' + CAST(@@TRANCOUNT AS NVARCHAR); BEGIN TRAN InnerTran; PRINT 'After BEGIN InnerTran: ' + CAST(@@TRANCOUNT AS NVARCHAR); ROLLBACK TRAN OuterTran; -- Rullar tillbaka båda! PRINT 'After ROLLBACK OuterTran: ' + CAST(@@TRANCOUNT AS NVARCHAR);

- Alla BEGIN TRAN ökar @@TRANCOUNT.
- **ROLLBACK på den yttre rullar tillbaka alla innersta transaktioner.**

3. Terminating Transactions (Avbryta transaktioner)

a) Resource error (Fel på resurser)

•Om en constraint bryts, avbryts transaktionen.

•Exempel: BEGIN TRAN INSERT INTO Customers (ID, Name) VALUES (1, 'Duplicate'); -- Förutsätter att ID=1 redan finns COMMIT; -- Kommer aldrig att ske, SQL Server stoppar det

b) SET XACT_ABORT

•Om ett fel uppstår **rullas hela transaktionen tillbaka automatiskt**.

•Exempel: SET XACT_ABORT ON; BEGIN TRAN UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; INSERT INTO Customers (ID, Name) VALUES (1, 'Duplicate'); -- Stoppar hela transaktionen

COMMIT;

c) Connection closure (Stängning av anslutning)

•Om en anslutning stängs **rullas pågående transaktioner tillbaka**.

•Exempel: BEGIN TRAN UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; -- Om användaren stänger SSMS eller förlorar anslutningen rullas transaktionen tillbaka

4. Best Practices (Bästa praxis för transaktioner)

•**Håll transaktioner så korta som möjligt** för att undvika låsning.

•Exempel på **dålig praxis**:

•BEGIN TRAN WAITFOR DELAY '00:00:10'; -- Simulerar långsam transaktion UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; COMMIT;

- Väntetiden gör att andra frågor **blockeras onödigt länge**.

•Exempel på **bra praxis**:

•BEGIN TRAN UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; COMMIT;

- **Snabb commit**, minskar risken för låsning.

Sammanfattning

1.Namn på transaktioner är bara etiketter.

2.Nästa transaktioner (@@TRANCOUNT) påverkas bara av den yttre transaktionen.

3.Avslutas genom fel, XACT_ABORT eller anslutningsbrott.

4.Håll transaktioner korta för att undvika låsproblem.

Dessa koncept är viktiga för **konkurrenshantering** i SQL Server! 🚀

Transactions in SQL Server for beginners
https://www.sqlshack.com/transactions-in-sql-server-for-beginners/

Error and Transaction Handling in SQL Server
https://www.sommarskog.se/error_handling/Part1.html

BEGIN TRANSACTION (Transact-SQL)
https://docs.microsoft.com/en-us/sql/t-sql/language-elements/begin-transaction-transact-sql

COMMIT TRANSACTION (Transact-SQL)
https://docs.microsoft.com/en-us/sql/t-sql/language-elements/commit-transaction-transact-sql

ROLLBACK TRANSACTION (Transact-SQL)
https://docs.microsoft.com/en-us/sql/t-sql/language-elements/rollback-transaction-transact-sql

SET XACT_ABORT (Transact-SQL)
https://docs.microsoft.com/en-us/sql/t-sql/statements/set-xact-abort-transact-sql

Lesson: Locking Internals

- Lock granularity and escalation
- Types of locks
- Lock mode compatibility
- Locking during data modifications
- Deadlocks
- Locking hints
- Monitoring locking and blocking

Lock granularity and escalation

- The lock manager decides at what level locks should be held:
  - Row (RID or KEY)
  - Page (PAGE)
  - Table (OBJECT)
- Each lock costs 128 bytes in memory
  - I.e., 10 million locks costs approx. 1.2 GB memory
- Locks can escalate at run-time
  - Row and page locks escalate to table locks
  - Control at table level with ALTER TABLE SET LOCK_ESCALATION
    - Table (default)
    - Disable (be careful)
    - Partition
  - Control at session or instance level with trace flags 1224 and 1211

Varsågod! Här är en genomgång av **lock escalation** i SQL Server med kodexempel.
Vad är lock escalation?
SQL Server använder **row locks** och **page locks** för att optimera samtidighet.
Men när för många lås används kan de **eskalera** till ett **table lock**, vilket **låser hela tabellen** och kan orsaka blockering.
Standardbeteende:
•**Många row locks** → SQL Server kan eskalera till **page locks**.
•**Många page locks** → Kan eskalera till **table lock**.
1. Kontroll av lock escalation på tabellnivå
Standardbeteende (Table lock, default)
•ALTER TABLE Customers SET (LOCK_ESCALATION = TABLE); Om SQL Server behöver eskalera lås går det direkt till **table lock**.
2. Förhindra lock escalation (Use with caution!)
•ALTER TABLE Customers SET (LOCK_ESCALATION = DISABLE); **Förhindrar** att lås eskalerar från **row/page** till **table lock**.
•**Risk:** Om för många lås tas kan SQL Server använda **för mycket minne** och orsaka prestandaproblem.
3. Partition-level locking
•ALTER TABLE Customers SET (LOCK_ESCALATION = PARTITION); SQL Server kan eskalera lås till **partitioner istället för hela tabellen**.
•Kräver att tabellen är **partitionerad**.
4. Kontroll på session- eller instansnivå
Trace Flag 1224 – Förhindrar lock escalation baserat på antal lås
•DBCC TRACEON(1224, -1); -- Aktiverar på servernivå DBCC TRACEOFF(1224, -1); -- Stänger av

Förhindrar **lock escalation** om minnesgränserna tillåter det.
Trace Flag 1211 – Fullständigt stoppa lock escalation (farligt!)
•DBCC TRACEON(1211, -1); -- Stänger helt av lock escalation DBCC TRACEOFF(1211, -1); -- Återgår till standard SQL Server tillåter **oändligt många lås**, vilket kan orsaka **hög minnesförbrukning**.
5. Exempel på lock escalation i praktiken
•BEGIN TRAN UPDATE Customers SET Name = 'Charlie' WHERE ID < 100000; -- Många rader COMMIT; Om **många rader påverkas**, kan SQL Server **eskalera** till ett **table lock**.
För att inspektera lås:
SELECT request_mode, resource_type, resource_description FROM sys.dm_tran_locks WHERE request_owner_type = 'TRANSACTION'; **Sammanfattning**
•**Lock escalation sker när SQL Server hanterar för många row/page locks.**
•**ALTER TABLE** kan styra om escalation sker till **table eller partition**.
•**Trace Flags 1224 och 1211** styr lock escalation på **session- eller servernivå**.
•**DISABLE lock escalation** kan orsaka **högt minnesutnyttjande**.
Bäst praxis är att **förstå ditt arbetslastmönster** och **inte stänga av lock escalation slentrianmässigt**. 🚀

Lock granularity and hierarchies
https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide#lock-granularity-and-hierarchies

SQL Server, Locks object
https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/sql-server-locks-object

SQL Server Concurrency: Locking, Blocking and Row Versioning (free e-book)
https://www.red-gate.com/library/sql-server-concurrency-locking-blocking-and-row-versioning

Types of locks

- Also known as Lock Modes
- Shared Sh
- Exclusive X
- Update U
- Intent I
- Key-range
- Schema stability Sch-S
- Schema modification Sch-M
- Bulk update BU

All about locking in SQL Server
https://www.sqlshack.com/locking-sql-server/

Lock modes
https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide#lock_modes

Lock mode compatibility

Compatibility between common lock modes

	Existing granted mode					
Requested mode	IS	S	U	IX	SIX	X
Intent shared (IS)	Yes	Yes	Yes	Yes	Yes	No
Shared (S)	Yes	Yes	Yes	No	No	No
Update (U)	Yes	Yes	No	No	No	No
Intent exclusive (IX)	Yes	No	No	Yes	No	No
Shared with intent exclusive (SIX)	Yes	No	No	No	No	No
Exclusive (X)	No	No	No	No	No	No

Lock compatibility
https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide#lock_compatibility

Locking during data modifications

- Relevant data pages located in the Buffer Pool
- Locks for modifications:
  - Update lock on possibly affected rows
    - If the row turn out to qualify, “convert” to exclusive
    - If not, then release update lock and go to next row
      - Intent exclusive lock on pages
      - Intent exclusive lock on table
  - Shared lock on database

Det här handlar om hur SQL Server hanterar låsning vid uppdateringar för att minimera låskonflikter och förbättra prestandan.

Update Lock (U) på potentiellt påverkade rader

•När en transaktion ska uppdatera en rad börjar den med ett **Update Lock (U)**.

•Detta lås är ett mellansteg mellan **Shared Lock (S)** och **Exclusive Lock (X)**.

•Fördelen med **Update Lock** är att det förhindrar **deadlocks**, eftersom det hindrar andra transaktioner från att ta ett **Exclusive Lock** på samma rad, men tillåter läsning.

Om raden kvalificerar sig för uppdatering

•Update Lock konverteras till ett **Exclusive Lock (X)**, vilket innebär att inga andra transaktioner kan läsa eller modifiera raden förrän transaktionen är klar.

Om raden inte kvalificerar sig

•**Update Lock släpps**, och transaktionen fortsätter till nästa rad utan att låsa onödiga resurser.

Intent Exclusive Lock (IX) på sidor och tabeller

•När en transaktion behöver ett **Exclusive Lock** på en rad, markerar SQL Server detta genom att först placera ett **Intent Exclusive Lock (IX)** på:

- **Sidan** där raden finns
- **Tabellen** som innehåller raden

Detta förhindrar att andra transaktioner får **Shared Lock (S)** på hela sidan eller tabellen, vilket säkerställer att transaktionen kan genomföra sin ändring utan konflikter.

Sammanfattningsvis gör denna mekanism att SQL Server kan hantera uppdateringar effektivt genom att minimera lås och konflikter medan databasen fortfarande skyddas mot inkonsekventa ändringar.

Optimized locking

- Implemented in Azure SQL Database
  - Planned for vNext
- Improves performance/concurrency
  - Fewer X locks held for modifications, no "trail of unnecessary X-locks" is left behind during the modification statement
  - Improves concurrency
  - Reduces memory requirements
- On by default for Azure SQL Database
  - Not all regions, at the time of writing (2023-02-14)
- Requires Accelerated Database Recovery (ADR)
  - And Read Committed Snapshot to realize full benefits

```
SELECT DATABASEPROPERTYEX('Adventureworks', 'IsOptimizedLockingOn') AS optimized_locking_on
```

Lock-eskalering är en mekanism i SQL Server där flera finfördelade lås, såsom rad- eller sidlås, automatiskt konverteras till ett mer omfattande lås, vanligtvis ett tabellås. Detta görs för att minska systemresursanvändningen och förbättra prestanda. Men i vissa situationer kan detta leda till blockeringar och påverka samtidigheten negativt.

Standardbeteende i traditionella databaser:

I traditionella SQL Server-databaser sker lock-eskalering när en enskild Transact-SQL-sats förvärvar minst 5 000 lås på en enda icke-partitionerad tabell eller index. När detta tröskelvärde nås, eskalerar SQL Server låsen till ett tabellås för att minska overhead och minnesanvändning. Detta standardbeteende gäller fortfarande i många SQL Server-implementationer. □cite□ turn0search0□

Optimerad låsning i moderna databaser:

I nyare SQL Server-versioner, särskilt i Azure SQL Database och SQL Database i Microsoft Fabric, har en funktion kallad "optimerad låsning" introducerats. Denna funktion minskar minnesanvändningen för lås och undviker lock-eskaleringar, vilket möjliggör mer samtidig åtkomst till tabeller. Optimerad låsning består av två huvudkomponenter:

1.Transaction ID (TID) Låsning: Varje rad märks med den senaste transaktions-ID:n som modifierade den. Istället för att hålla många radlås, används ett enda lås på TID, vilket minskar antalet nödvändiga lås.

2.Lock After Qualification (LAQ): Denna optimering utvärderar frågevillkor med den senaste committade versionen av raden utan att förvärva ett lås initialt, vilket förbättrar samtidigheten. Med optimerad låsning kan till exempel en uppdatering av 1 000 rader kräva 1 000 exklusiva radlås, men varje lås frigörs så snart raden har uppdaterats, och endast ett TID-lås hålls tills transaktionen är klar. Detta minskar minnesanvändningen för lås och minskar sannolikheten för lock-eskaleringar, vilket förbättrar samtidigheten i arbetsbelastningen. □cite□turn0search1□

Tillgänglighet:

Optimerad låsning är för närvarande tillgänglig i Azure SQL Database och SQL Database i Microsoft Fabric. Den är inte tillgänglig i Azure SQL Managed Instance eller i traditionella SQL Server-installationer. För att kontrollera om optimerad låsning är aktiverad i din databas kan du använda följande fråga:
SELECT IsOptimizedLockingOn = DATABASEPROPERTYEX(DB_NAME(), 'IsOptimizedLockingOn'); Resultatet blir 1 om optimerad låsning är aktiverad, 0 om den är inaktiverad, och NULL om funktionen inte är tillgänglig.
Sammanfattningsvis är lock-eskalering en standardmekanism i SQL Server för att hantera resurser effektivt, men den kan påverka samtidigheten negativt i vissa scenarier. Moderna funktioner som optimerad låsning i Azure SQL Database syftar till att förbättra detta genom att minska behovet av lock-eskaleringar och förbättra prestanda i miljöer med hög samtidighet.

Optimized locking
https://learn.microsoft.com/en-us/sql/relational-databases/performance/optimized-locking?view=azuresqldb-current

Deadlocks

- Deadlocks are resolved by the Lock Manager:
  - Runs every five seconds by default
  - Frequency increases as deadlocks are detected
  - Deadlock victim is selected and terminated
- Deadlock priority can be set
  - SET DEADLOCK PRIORITY
- Manage using Extended Events
  - xml_deadlock_report

Demo Deadlocks

Deadlock-prioritet och hantering i SQL Server

1. Deadlock Priority – Vad det gör

När en **deadlock** inträffar väljer SQL Server **en transaktion att avbryta** för att lösa konflikten. Standardbeteendet är att SQL Server **väljer den transaktion som har använt minst resurser som "offer"**.

Men vi kan **styra vilken transaktion som ska avbrytas** genom att sätta **DEADLOCK PRIORITY**.

2. Användning av SET DEADLOCK PRIORITY

Vi kan ange prioritet med tre nivåer:

•**LOW** → Transaktionen är mer benägen att bli offer.

•**NORMAL** (default) → Standardbeteende, SQL Server bestämmer.

•**HIGH** → Transaktionen har högre prioritet och **offras sist**.

Exempel 1 – Sätta låg prioritet på en transaktion

•SET DEADLOCK PRIORITY LOW; BEGIN TRAN UPDATE Customers SET Name = 'Charlie' WHERE ID = 1; WAITFOR DELAY '00:00:05'; -- Simulerar långsam transaktion COMMIT; Om en deadlock uppstår **offras denna transaktion först**.

Exempel 2 – Sätta hög prioritet på en viktig transaktion

•SET DEADLOCK PRIORITY HIGH; BEGIN TRAN UPDATE Orders SET Status = 'Shipped' WHERE OrderID = 100; COMMIT; **Denna transaktion är skyddad** och kommer **inte** att bli offer om en deadlock uppstår.

3. Deadlock-analys med Extended Events (xml_deadlock_report)

SQL Server har **Extended Events** för att logga och analysera deadlocks.

Huvudeventet som används är **xml_deadlock_report**, som fångar detaljer om varje deadlock.

Skapa en Extended Event-session för att logga deadlocks

•CREATE EVENT SESSION DeadlockMonitor ON SERVER ADD EVENT sqlserver.xml_deadlock_report ADD TARGET package0.event_file (SET filename =

'C:\Deadlocks\DeadlockReport.xel') WITH (STARTUP_STATE = ON); GO ALTER EVENT SESSION DeadlockMonitor ON SERVER STATE = START; **Detta loggar alla deadlocks till en .xel-fil** som kan analyseras i SQL Server Management Studio.

Visa loggade deadlocks

•SELECT * FROM sys.fn_xe_file_target_read_file('C:\Deadlocks\DeadlockReport*.xel', NULL, NULL, NULL); **Ger detaljer om vilka objekt och transaktioner som orsakade deadlocken**.

Sammanfattning

1.SET DEADLOCK PRIORITY låter dig styra **vilken transaktion som ska offras** vid en deadlock.

2.xml_deadlock_report loggar deadlocks så att du kan analysera dem med **Extended Events**.

3.Extended Events är **bästa sättet att proaktivt identifiera och åtgärda deadlocks** i SQL Server. 🚀

Deadlocks
https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide#deadlocks

What are SQL Server deadlocks and how to monitor them
https://www.sqlshack.com/what-are-sql-server-deadlocks-and-how-to-monitor-them/

Locking hints

- Hints affecting Lock Mode:
  - ROWLOCK
  - PAGLOCK
  - TABLOCK
  - TABLOCKX
  - UPDLOCK
  - XLOCK
  - READPAST
- Hints affecting Isolation level at table level:
  - READCOMMITTED
  - READCOMMITTEDLOCK
  - READUNCOMMITTED or NOLOCK
  - REPEATABLEREAD
  - SERIALIZABLE or HOLDLOCK

```
CREATE DATABASE LockingDemo;
GO
USE LockingDemo;
GO
CREATE TABLE Customers (
   ID INT PRIMARY KEY,
   Name NVARCHAR(50)
);
INSERT INTO Customers VALUES (1, 'Alice'), (2, 'Bob'), (3, 'Charlie’);

--ROWLOCK
UPDATE Customers WITH (ROWLOCK) SET Name = 'David' WHERE ID = 1;


--PAGLOCK
UPDATE Customers WITH (PAGLOCK) SET Name = 'Eve' WHERE ID = 2;

--TABLOCK
SELECT * FROM Customers WITH (TABLOCK);


--READCOMMITTED – Standardläge, läser endast committad data
SELECT * FROM Customers WITH (READCOMMITTED);

--READUNCOMMITTED / NOLOCK – Tillåter dirty reads
```

```
SELECT * FROM Customers WITH (READUNCOMMITTED);
-- Eller ekvivalent:
SELECT * FROM Customers WITH (NOLOCK);
```

Hints (Transact-SQL) – Table
https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table

Understanding the SQL Server NOLOCK hint
https://www.mssqltips.com/sqlservertip/2470/understanding-the-sql-server-nolock-hint/

Using NOLOCK? Here's How You'll Get the Wrong Query Results.
https://www.brentozar.com/archive/2018/10/using-nolock-heres-how-youll-get-the-wrong-query-results/

"But NOLOCK Is Okay When My Data Isn't Changing, Right?"
https://www.brentozar.com/archive/2019/08/but-nolock-is-okay-when-the-data-isnt-changing-right/

Monitoring locking and blocking

- sys.dm_tran_locks
- Download any of below
  - sp_whoisactive
  - beta_lockinfo
  - sp_blitzlock
- Activity Monitor in SSMS
- Troubleshoot deadlock by capturing below event in an Extended Event trace
  - sqlserver.xml_deadlock_report

Demo Locking information

sp_whoisactive
http://whoisactive.com/

beta_lockinfo
https://www.sommarskog.se/sqlutil/beta_lockinfo.html

sp_blitzlock
https://github.com/BrentOzarULTD/SQL-Server-First-Responder-Kit/blob/main/sp_BlitzLock.sql

V1 Lab 5: Concurrency and transaction handling

- Exercise 1: Implement Snapshot Isolation
- Exercise 2: Implement Partition Level Locking

Estimated Time: 30 minutes

Lab 5: Concurrency and transaction handling

- Ex 1: Improve concurrency
- Ex 2: Capture deadlock information into a trace

Estimated Time: 30 minutes