Peter ter Braake

Concurrency en integriteit

SQL SERVER EN LOCKING

Wanneer verschillende gebruikers tegelijkertijd op dezelfde database werken, moet je er rekening mee houden dat ze mogelijkerwijs exact dezelfde gegevens willen wijzigen. In dit artikel beschrijft de auteur hoe SQL Server omgaat met locking en welke mogelijkheden we tot onze beschikking hebben om invloed uit te oefenen op het effect van die locks.

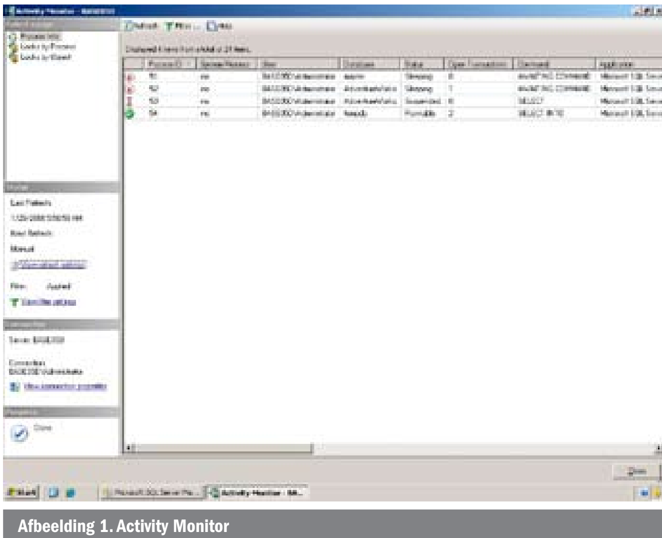
SQL Server 2008 wordt door Microsoft aan de man gebracht als enterprise database. SQL Server moet kleine databases met slechts enkele gebruikers (Express edition) aankunnen, maar ook heel grote bedrijfsbrede systemen met duizenden gebruikers (Enterpise edition). Daarnaast verwachten we een goede performance van OLTP-systemen (online transaction processing) met veel schrijfactiviteit en van datawarehouse-systemen. Nog lastiger zijn de systemen met een mixed workload: veel transacties maar ook langlopende queries. In alle gevallen moet SQL Server de integriteit van de data garanderen en een zo groot mogelijke concurrency (met meer gebruikers tegelijkertijd in de database werken) ondersteunen. In dit artikel kijken we hoe SQL Server dat aanpakt en wat wij als developers kunnen doen om onze applicaties zo efficiënt mogelijk te laten werken.

De theorie, ACID-properties

Een DBMS (databasemanagementsysteem) moet er voor zorgen dat een database aan de zogenaamde ACID-properties voldoet. De letters ACID staan voor Atomicity, Consistency, Isolation en Durability. Met atomicity bedoelen we dat een actie in de database als een geheel moet kunnen worden uitgevoerd en dus volledig slaagt of niet doorgaat. Het schoolvoorbeeld hiervan is het overmaken van geld van de ene bankrekening naar de andere. Hoogstwaarschijnlijk zijn hier twee update-statements voor nodig, één om het saldo van de eerste rekening te verlagen en de ander om het saldo van de tweede rekening te verhogen. Er is hier sprake van één logische handeling (geld overmaken) die twee database-handelingen vereist. Als we deze twee update-statements niet als één geheel kunnen uitvoeren, krijgen we situaties waarbij het geld van de ene rekening wordt afgehaald maar nooit op de andere rekening wordt bijgeschreven. Wie vertrouwt zijn geld toe aan een bank die zo werkt? Uiteraard hebben we het hier over transacties. Door beide update-statements een onderdeel te maken van dezelfde transactie, moeten beide statements volledig en succesvol uitgevoerd worden om de gehele transactie met succes af te ronden. Als de ene update succesvol is uitgevoerd maar de ander niet, zal de eerste update worden teruggedraaid (rollback), zodat beide updates niet zijn uitgevoerd.
De letter C in ACID staat, zoals gezegd, voor consistency. Daarmee bedoelen we dat de data in de database altijd een logische consistente (kloppende) weergave van de werkelijkheid moet zijn. In het hierboven beschreven voorbeeld van een bankoverschrijving kan er geen geld verdwijnen of gegenereerd worden. Het geld staat altijd op de ene of op de andere rekening, nooit op beide tegelijkertijd of, erger nog, op geen van beide. Elke rekening heeft op elk willekeurig moment één eenduidig saldo. Daarmee komen we bij de I van isolation. Verschillende gebruikers die tegelijkertijd met dezelfde data werken moeten uit elkaar gehouden worden, zodat beiden altijd met logisch consistente data werken. Wat gebeurt er als twee mensen met een gedeelde rekening tegelijkertijd bankzaken aan het regelen zijn? De ene vraagt via een pinautomaat het saldo op, terwijl de andere op exact hetzelfde moment via internet een overschrijving doet. Welk saldo krijgt de eerste persoon te zien? De oplossing schuilt hier in locking, een mechanisme dat er voor zorgt dat maar één user tegelijkertijd bij de data kan.
Voordat ik nader in ga op locking in SQL Server, behandel ik eerst de D van durability en de theoretische problemen die je tegenkomt als een database niet aan de ACID-properties voldoet. Met durability bedoelen we dat als gegevens eenmaal door een transactie zijn gewijzigd, ze ook permanent (dat wil zeggen totdat een volgende transactie de data weer wijzigt) in de database zitten. Wat er ook gebeurt, als de overschrijving is gedaan, staat het geld op de nieuwe rekening.
Als een DBMS niet aan de ACID-properties kan voldoen, ontstaan er verschillende problemen. Het meest bekende is de 'dirty read', ofwel het lezen van data die door iemand anders op hetzelfde moment worden gewijzigd. In het eerdergenoemde voorbeeld van een gemeenschappelijke rekening kan één persoon bij een pinautomaat geld opnemen, terwijl de andere via het internet het saldo van de rekening opvraagt. Een dirty read treedt op als de laatste persoon het nieuwe saldo ziet en de eerste persoon vervolgens besluit toch geen geld op te nemen. Er is dan logisch gezien nooit sprake geweest van een nieuw saldo, maar toch is dat wat persoon twee heeft gezien. Naast dirty reads hebben we te maken met 'phantom reads', 'inconsistent analysis' en 'lost updates'. Het gaat hier te ver ze allemaal uitgebreid te beschrijven. Het komt er echter elke keer op neer dat we met een niet-consistente staat van gegevens hebben te maken.

Concurrency control

SQL Server is oorspronkelijk ontworpen als databasesysteem op afdelingsniveau. Dat houdt in dat het om iets kleinere systemen gaat waar enkele mensen tegelijkertijd mee werken en waar de kans dat meer mensen tegelijkertijd dezelfde data benaderen redelijk groot is. Oracle daarentegen is ontworpen als enterprise-database. Hoewel het aantal gebruikers waarschijnlijk groter is, is de kans dat mensen tegelijkertijd met dezelfde data werken een stuk kleiner dan bij afdelingsdatabases waar iedereen ongeveer hetzelfde werk doet. Vanuit deze achtergrond hebben beide systemen een andere oplossing gekozen om de eerdergenoemde problemen te voorkomen. Oracle kiest een optimistische benadering (de kans is klein dat ik daadwerkelijk een probleem tegenkom) in de vorm van versioning. Eenvoudig gezegd komt dit er op neer dat iedereen zijn eigen consistente versie van de data krijgt. SQL Server kiest gezien zijn achtergrond voor de pessimistische benadering, ofwel locking.
Als iemand een resource in de database benadert (een record of een tabel) zet SQL Server een lock op de desbetreffende resource. Dit houdt eigenlijk niets anders in dan dat SQL Server precies bijhoudt wie, wat, waar doet in de database. Deze informatie wordt gebruikt om te beletten dat verschillende gebruikers tegelijkertijd acties op



Afbeelding 1. Activity Monitor

dezelfde data uitvoeren, waardoor de consistentie geschaad zou worden. Grofweg zijn er twee soorten locks: shared en exclusive. Als iemand een record leest, heeft hij daar een shared lock voor nodig. Als iemand anders tegelijkertijd hetzelfde record wil lezen, is daar ook een shared lock voor nodig. Maar zoals de naam al suggereert, kunnen deze locks tegelijkertijd bestaan, wat inhoudt dat beide gebruikers tegelijkertijd hetzelfde record kunnen lezen. Het wordt lastiger als een derde gebruiker een update wil uitvoeren op dit record. Voor een update is een exclusive lock nodig. Met andere woorden, tijdens een update kan geen enkele andere gebruiker bij het record. Dit houdt in dat iemand die een record wil lezen, zal moeten wachten totdat de transactie die het record verandert klaar is en daarmee zijn exclusive lock opgeeft. De gebruiker die het record wil lezen, ervaart dit wachten als slechte performance.
Daar waar locking er dus voor zorgt dat we geen data-consistentieproblemen hebben, zorgt het er ook voor dat gebruikers die schrijven, gebruikers die lezen in de weg zitten. Maar ook andersom is dat waar (writers block readers and readers block writers). En dus gaat de performance van applicaties achteruit als er meer gebruikers tegelijkertijd bij dezelfde gegevens willen.

Isolation levels

Gelukkig geeft SQL Server developers de kans om het hierboven beschreven locking-mechanisme enigszins te beïnvloeden. We kunnen dit doen via zogenaamde isolation levels. Het default isolation level is Read Committed. Dit houdt dat alleen gegevens gelezen kunnen worden die gecommit zijn, ofwel de laatste transactie die de data heeft gewijzigd, is al beëindigd. Codevoorbeeld 1 en 2 laten dit zien. De code onder stap 1 in codevoorbeeld 1 start een transactie en doet een update. Als we vervolgens, dus voordat we de rollback van stap 3 uitvoeren, vanuit een andere connectie (ander window) de code van codevoorbeeld 2 uitvoeren, zien we dat het simpele select-statement dat daar staat, schijnbaar oneindig bezig is. Als we de Activity Monitor starten (onder de managementfolder in Management Studio) zien we dat deze connectie in de suspended state staat. Dit houdt in dat SQL Server wacht op het wegvallen van de lock die het update-statement heeft. Zodra we vanuit de eerste connectie stap 3 uitvoeren, zal het select-statement een resultaat teruggeven. De performance van de select query is heel slecht, omdat de update-transactie veel tijd nodig heeft en onze query moet wachten totdat de update-transactie is afgerond.

```
--STAP 1
BEGIN TRANSACTION

UPDATE Production.Product
SET Name = 'Adjustable Race 2'
Where ProductID = 1

-- Do some extra work

--STAP 3
ROLLBACK TRANSACTION
```

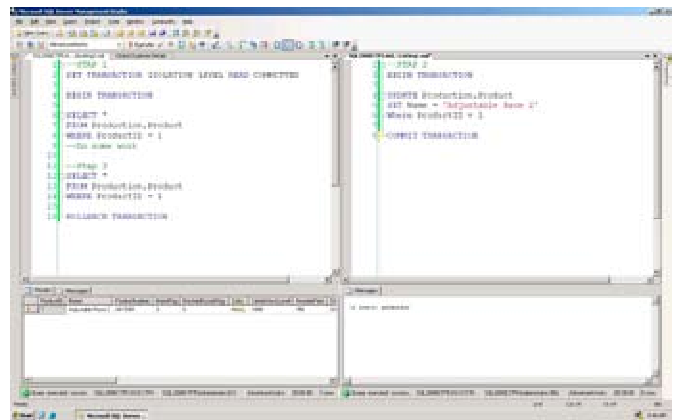
Codevoorbeeld 1.

```
--STAP 2

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

SELECT *
FROM Production.Product
WHERE ProductID = 1
```

Codevoorbeeld 2.

Als we in codevoorbeeld 2 regel 3 Read Committed veranderen in Read Uncommitted en de stappen opnieuw uitvoeren, zien we een totaal ander gedrag. Het Select-statement uit codevoorbeeld 2 geeft nu wel direct een resultaat terug. Belangrijk om hier op te merken is dat we de nieuwe gewijzigde naam van het product te zien krijgen. Dit is een dirty read. Want wat nu als we in codevoorbeeld 1 de rollback uitvoeren? Logisch gezien heeft het product nooit een andere naam gehad, maar toch hebben we een keer 'Adjustable Race 2' gelezen. We hebben hier een betere performance bereikt ten koste van betrouwbaarheid van de gelezen data. In dit voorbeeld is dat waarschijnlijk een slechte deal, maar in het geval van een langlopende query waar een totaalbedrag wordt berekend, waarvan de orde van grootte van belang is en niet zozeer het exacte bedrag, kan ons dit zeker goed helpen.
In het bovenstaande hebben we een voorbeeld van een schrijfactie gezien die een leesactie blokkeerde. Misschien wel erger is het als een leesactie een schrijfactie in de weg zit. Kijk bijvoorbeeld eens naar afbeelding 2, codevoorbeeld 3 en 4. De code in codevoorbeeld 3 start een transactie en leest een record. Vervolgens wordt vanuit de andere connectie het zojuist gelezen record aangepast. De eerste connectie leest later in dezelfde transactie het record nog eens. Het krijgt nu de veranderde naam te zien. Hoewel deze data gecommit zijn en dus consistent, ontstaat er toch een rare situatie. We noemen dit een non repeatable read wat leidt tot een zogenaamde inconsistent analysis. Gebaseerd op de eerste leesopdracht bepaalt de logica van de code wat verder te doen, maar tegen de tijd dat we dat daadwerkelijk gaan doen, is de situatie al veranderd.

Als we in dit voorbeeld in codevoorbeeld 3 regel 2 Read Committed vervangen door Repeatable Read en de code weer uitvoeren, zien we weer een fundamentele verandering. Het update-statement wordt nu geblokkeerd door de select. Het statement moet wachten tot de transactie waarin gelezen wordt eindigt. We zien nu dus een leesactie die een schrijfactie in de weg zit. En erger nog, in het eerste voorbeeld veranderden we het isolation-level in read uncommitted, waarmee we alleen invloed hadden op de resultaten binnen onze eigen connectie. Nu verandert de ene connectie zijn isolation-level en ondervindt een andere connectie daar hinder van.



Afbeelding 2. Screenshot van een leesactie die een schrijfactie in de weg zit

Het isolation-level repeatable read zorgt er voor dat we binnen een transactie dezelfde data te zien krijgen als we hetzelfde record opnieuw lezen. Wel zouden we in een andere transactie nog een record kunnen toevoegen aan de producttabel. Als we hetzelfde voorbeeld nogmaals uitvoeren, maar dan zonder de where-clauses, dan krijgen we alsnog problemen. Want hoewel de al gelezen records niet veranderen, kunnen er wel meer records gelezen worden bij het tweede select-statement. Als we het isolation-level veranderen in Serializable, krijgt onze transactie een zogenaamd range lock, waardoor andere transactie ook geen inserts meer kunnen doen.

Versioning

De vier bovenstaande isolation-levels zijn beschreven in de ANSI-standaard. Maar zoals eerder gezegd, is er ook een andere aanpak. Oracle is heel succesvol met versioning en vanaf SQL Server 2005 ondersteunt ook SQL Server dit. Met name queries met een lange doorlooptijd die draaien in een OLTP-omgeving kunnen hier voordeel van hebben. Om gebruik te kunnen maken van versioning, moeten we wel eerst aangeven dat we dat gaan doen door middel van onderstaand Alter-database-statement:

```
ALTER DATABASE AdventureWorks
SET ALLOW_SNAPSHOT_ISOLATION ON
```

Nadat we dit statement hebben uitgevoerd, kunnen we het eerste voorbeeld van codevoorbeeld 1 en 2 nogmaals uitvoeren. Verander in codevoorbeeld 2 regel 3 Read Committed in Snapshot en voer de stappen nogmaals uit. We zien dat het select-statement niet langer hoeft te wachten op de update en dat de teruggegeven waarde voor de naam de laatst bekende gecommitte waarde is. We hebben nu dus ook weer een verbeterde performance van het select-statement zonder het nadeel van de read uncommitted die een dirty read veroorzaakte. Als we in plaats van het bovenstaande Alter-database-statement het onderstaande uitvoeren, krijgen we ongeveer hetzelfde.

```
ALTER DATABASE AdventureWorks
SET READ_COMMITTED_SNAPSHOT ON
```

```
--STAP 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED

BEGIN TRANSACTION

SELECT *
FROM Production.Product
WHERE ProductID = 1
--Do some work

--Stap 3
SELECT *
FROM Production.Product
WHERE ProductID = 1

ROLLBACK TRANSACTION
```

Codevoorbeeld 3.

```
--STAP 2
BEGIN TRANSACTION

UPDATE Production.Product
SET Name = 'Adjustable Race 2'
Where ProductID = 1

COMMIT TRANSACTION
```

Codevoorbeeld 4.

Dit statement heeft tot gevolg dat elke connectie naar de database als default snapshot-isolation krijgt. Er is echter wel een verschil met de eerstgenoemde variant. De eerstgenoemde variant doet namelijk versioning op transactieniveau. Elk statement binnen een transactie leest de gegevens zoals die waren toen de transactie begon. Met read committed snapshot lees je de gegevens zoals die waren toen het statement begon. En tussen het begin van een statement en het begin van de bijbehorende transactie kan een andere transactie uiteraard een update hebben uitgevoerd. Als we gebruikmaken van snapshot-isolation moeten we ons wel bewust zijn van het feit dat SQL Server de verschillende versies van records zo lang als nodig is in TempDB opslaat. TempDB wordt dus meer gebruikt en als we daar geen rekening mee houden (grootte van de database, snelheid van de schijf waarop hij staat) kan dat de performance negatief beïnvloeden.

Query Hints

In het bovenstaande hebben we gekeken hoe het zetten van isolation-levels invloed heeft op welke data we hoe snel lezen. Merk daarbij op dat een SET-statement van invloed is op de connectie en dus geldt voor alle T-SQL die we via die connectie naar SQL Server sturen. We kunnen ook bepalen hoe individuele statements omgaan met locks via zogenaamde locking hints. De hint Readpast bijvoorbeeld zorgt er voor dat een select-statement de records die gelocked zijn gewoon negeert. Dat heeft tot gevolg dat je weer meteen resultaten terugkrijgt, maar misschien niet alle records leest die je had verwacht. Ook de bovengenoemde isolation-levels kom je als hints tegen op statement-niveau. Je kunt in BOL meer lezen over locking hints.

Rekening houden met

Zodra meer gebruikers tegelijkertijd op dezelfde database werken, moet hier rekening mee worden gehouden. SQL Server doet dit door middel van locking. De locks die SQL Server gebruikt, kunnen we beïnvloeden. Ook kunnen we enigszins invloed uitoefenen op het effect van die locks. Dit kan op connection-niveau via isolation-levels en op statement-niveau via hints. Om een efficiënte applicatie te schrijven die gebruikmaakt van SQL Server, moeten we de totale workload van de database kennen en rekening houden met de manier die SQL hanteert om met concurrent users om te gaan.

Peter ter Braake is productspecialist SQL Server bij Compu'Train. Zijn e-mailadres is pbraake@computrain.nl. Compu'Train (www.computrain.nl) is de grootste CPLS in Nederland en marktleider op het gebied van SQL Server-trainingen. Peter is ook vice-voorzitter bij de Nederlandse afdeling van PASS. PASS is een wereldwijde gebruikersgroep voor SQL Server-professionals.

Referenties:

www.microsoft.com/sql

www.sqlpass.nl