Robert Tusveld en Melchior Brandt Corstius

SQL Server 2005 Performance Tuning

ALLES WAT JE ALS DEVELOPER MOET WETEN OVER INDEXOPTIMALISATIE

Iedere applicatie die vandaag de dag wordt ontwikkeld, heeft in meer of mindere mate te maken met het opslaan van informatie. Deze informatie kun je op diverse manieren opslaan, maar vaak zal je hiervoor een database gebruiken. Dit is dan ook de reden dat iedere ontwikkelaar wordt geacht een zekere basiskennis te hebben van databases. Hoe moet je een database ontwerpen? Hoe optimaliseer je een database voor performance? Op welke manier maak je indexen aan en hoe onderhoud je ze? Dit zijn vragen die je als ontwikkelaar moet kunnen beantwoorden.

Dit artikel gaat in op het gebruik en de optimalisatie van indexen in SQL Server. Om het verhaal goed te kunnen begrijpen, volgt eerst een korte inleiding waarin de basisbegrippen en de uitgangspunten van het gebruik van indexen in SQL Server worden behandeld.

Indexen in SQL Server

SQL Server kent twee typen tabellen: *tabellen met een clustered index* en *tabellen zonder een clustered index*, ook wel *heap* genaamd. In een tabel met een clustered index zijn de records geordend in de volgorde van de clustered index. Een record kan worden gevonden op basis van de sleutel van de index. In een heap zijn de records niet geordend. Een record kan worden gevonden op basis van een row identifier (RID), een verwijzing naar de fysieke locatie van het record. In bijna alle gevallen is het aan te raden om bij een tabel een clustered index aan te maken. Slechts in die gevallen dat de tabel gebruikt wordt voor het inlezen, verwerken en weer verwijderen van grote hoeveelheden data, zoals een IIS-log, is een heap de meest geschikte oplossing.

Naast twee typen tabellen zijn ook twee typen indexen te onderscheiden: *clustered indexen* en *non-clustered indexen*. Een clustered index bepaalt de volgorde waarin de data van de tabel worden opgeslagen. Bij een clustered index worden de data van de tabel dus opgeslagen in de volgorde die door de index is aangegeven. De informatie van de clustered index is dan ook de informatie van de tabel en geen redundante informatie. Per tabel kan maximaal één clustered index worden gemaakt; denk dus goed na bij het kiezen van deze sleutel. Een non-clustered index bevat daarentegen alleen de kolommen die in de index zijn gedefinieerd, met daarbij een verwijzing naar de echte data in de tabel. Een non-clustered index bevat dus een kopie van de kolommen van de index en een kopie van de sleutel of een row identifier om in de tabel de daadwerkelijke data te kunnen opzoeken. Het opzoeken van de informatie in de tabel op basis van een sleutel of een row identifier wordt ook wel een *bookmark lookup* genoemd. Het kan voorkomen dat de SQL Server-query optimizer geen betere oplossing kan bedenken dan de gehele tabel van begin tot eind te doorlopen, dit wordt een *table scan* genoemd.

Bij de keuze van de sleutel van een clustered index is het aan te raden de volgende uitgangspunten te hanteren. De gekozen sleutel moet:

- **Uniek zijn.** SQL Server dwingt dit niet af, maar intern maakt SQL Server elk record uniek toegankelijk. Indien de sleutel niet uniek is, dan wordt door SQL Server extra informatie aan ieder record toegevoegd om dit record te kunnen identificeren. Dit kost geheugenruimte en dus performance.
- **Smal zijn**. Aangezien de sleutel van de clustered index in elke non-clustered index is gedupliceerd, zorgt een brede sleutel ervoor dat alle indexen extra geheugen gebruiken en dus extra performance kosten.
- **Statisch zijn.** Aangezien de sleutel van de clustered index in elke non-clustered index is gedupliceerd, zorgt het wijzigen van de sleutel er voor dat alle non-clustered indexen moeten worden gewijzigd. Dit leidt tot een groot aantal schrijfacties en kan ook leiden tot een sterk gefragmenteerde tabel. Beide aspecten hebben een negatief effect op de performance.
- **Oplopend zijn.** Het toevoegen van records onderaan de tabel zorgt voor een optimale wijze van locking van de tabel. Daarnaast zal de tabel na verloop van tijd minimaal gefragmenteerd zijn.

Het hanteren van deze uitgangspunten bij het ontwerpen van een tabel zorgt voor een optimale performance. Voorbeelden van sleutels van een clustered index die voldoen aan deze uitgangspunten zijn een identity-veld, een combinatie van de actuele datum en een identity-veld, of een GUID, maar alleen indien de functie **newsequentialid()** wordt gebruikt. Het gebruik van een normale GUID heeft juist een averechts effect, aangezien een normale GUID een volledig willekeurige waarde heeft. Op termijn leidt dit tot een in hoge mate gefragmenteerde tabel.

Afbeelding 1. Non-clustered index-intersection

Aandachtspunten bij databaseontwerp

Bij het ontwerpen van een database en het definiëren van de bijbehorende indexen moet met een aantal zaken rekening worden gehouden:

1. **Welke informatie wordt in de database opgeslagen**? Een database met geografisch geordende informatie bevat andere indexen dan een database met klanteninformatie.
2. **Op welke manier wordt de informatie in de database gebruikt**? De indexen kunnen zeer goed op het gebruik worden afgestemd, stored procedures kunnen worden geoptimaliseerd op het gebruik en eventueel kan er anders met informatie worden omgegaan als bijvoorbeeld bekend is dat de informatie alleen gelezen wordt.
3. **Test met representatieve hoeveelheden data.** Hoe vaak komt het niet voor dat een ontwikkelaar een query heeft geschreven die razendsnel is bij het testen (met 10 records), terwijl in productie (met 500.000 records) performanceproblemen ontstaan. Door met inhoudelijk representatieve testdata en representatieve hoeveelheden te testen, kan dit soort problemen voorkomen worden.
4. **Als er problemen optreden,** blijf dan zoeken totdat de echte oorzaak is gevonden. Eindgebruikers kunnen niet altijd goed aangeven wat het echte probleem is; vind zelf de query die het probleem veroorzaakt en verbeter deze query. De SQL Profiler kan hierbij helpen.

De meest voor de hand liggende manier waarop performanceproblemen aan het licht komen, zijn klachten van gebruikers. Het kan ook zijn dat het resourcegebruik op de databaseserver erg hoog is; veel CPU-activiteit, veel lees- en schrijfacties en het uitvoeren van stored procedures duurt lang. Door queries met een excessief resourcegebruik te voorkomen, worden ook locking-problemen geminimaliseerd, bijvoorbeeld deadlocks. Locking is in de meeste gevallen een symptoom van een ander probleem en niet het echte probleem.

Tip: Als 10-20% van de meest gebruikte queries met een slechte performance wordt verbeterd, is in de meeste gevallen 80% van de problemen opgelost!

Bezint eer gij begint…

SQL Server verzamelt statistische informatie over indexen en data in de database. Deze statistieken worden door de SQL Server query-optimizer gebruikt om te bepalen wat het efficiëntste executieplan is voor het uitvoeren van een query. Als de statistieken niet overeenkomen met de actuele samenstelling van de data in de database, kiest de query-optimizer mogelijk een minder geschikt executieplan. Dit resulteert in een verminderde performance. Voordat een oplossing voor de problemen wordt bedacht, is het aan te raden om eerst te controleren of deze statistieken wel goed zijn bijgewerkt. Met de query in codevoorbeeld 1 kunnen de statistieken voor een database worden getoond. In de laatste kolom wordt aangegeven wanneer de statistieken voor het laatst zijn bijgewerkt.

```
SELECT
   object_name(si.[object_id])    AS [TableName]
   , CASE
      WHEN si.[stats_id] = 0 then 'Heap'
      WHEN si.[stats_id] = 1 then 'CL'
      WHEN INDEXPROPERTY(si.[object_id], si.[name], 'IsAutoStatistics') = 1
           THEN 'Stats-Auto'
      WHEN INDEXPROPERTY(si.[object_id], si.[name], 'IsHypothetical') = 1
           THEN 'Stats-HIND'
      WHEN INDEXPROPERTY(si.[object_id], si.[name], 'IsStatistics') = 1
           THEN 'Stats-User'
      WHEN si.[stats_id] BETWEEN 2 AND 250 THEN 'NC ' + RIGHT('00' +
 convert(varchar, si.[stats_id]), 3)
      ELSE 'Text/Image'
     END                        AS [IndexType]
   , si.[name]                  AS [IndexName]
   , STATS_DATE (si.[object_id], si.[stats_id])  AS [Last Stats Update]
FROM sys.stats AS si
WHERE OBJECTPROPERTY(si.[object_id], 'IsUserTable') = 1
ORDER BY [Last Stats Update] DESC
GO
```

Codevoorbeeld 1. Toon databasestatistieken

```
USE AdventureWorks;
GO

UPDATE STATISTICS Sales.SalesOrderDetail;
GO
```

Codevoorbeeld 2. Bijwerken statistieken

Met behulp van het commando UPDATE STATISTICS <tabelnaam> kunnen de statistieken van een tabel worden bijgewerkt. In codevoorbeeld 2 wordt dit commando getoond.

Tip: *Zorg ervoor dat statistieken regelmatig geactualiseerd worden. Dit kan door een taak te definiëren die de statistieken elke nacht automatisch vernieuwt!*

In sommige gevallen kan een andere query-aanpak voor een performanceverbetering zorgen. Dit kan de query-optimizer helpen met het vinden van een beter executieplan. Bij het schrijven van een query kan vaak niet ingeschat worden wat de beste oplossing is. Probeer eerst de aanpak die het meest voor de hand ligt en blijkt de performance niet acceptabel, overweeg dan de query te herschrijven. Denk hierbij aan wijziging van de volgorde van de WHERE-clause, gebruik van tijdelijke tabellen, views, tabelvariabelen, gebruik van een JOIN in plaats van een sub-query, of andersom. Test, test, test en kijk of de query-optimizer het optimale executieplan kan ontdekken.
Als dit alles niet, of niet genoeg, geholpen heeft, kan de volgende stap worden uitgevoerd; Optimaliseer de indexen!

Optimaliseer de indexen

Er zijn bedrijven die als indexstrategie op iedere kolom in een tabel een non-clustered index aanmaken. Dit lijkt handig, maar levert in de praktijk nauwelijks voordeel op. Sterker nog, het onderhoud van de indexen kost performance. Smalle indexen hebben geen goede toepasbaarheid: na een index-search moet altijd een bookmark lookup uitgevoerd worden om bij de gezochte data te komen. Bookmark-lookups zijn alleen geschikt als het resultaat van de query erg selectief is. Selectief betekent dat het resultaat van een query minder dan 0,5% van het totale aantal records in de tabel bevat. Is de query minder selectief, dan is het vaak voordeliger voor SQL Server om een table-scan uit te voeren. De query optimizer probeert het meest optimale executieplan samen te stellen voor het uitvoeren van iedere query. SQL Server 2005 heeft een tweetal mogelijkheden om indexen te optimaliseren, te weten *non-clustered index-intersection* en *covering*.
Non-clustered index-intersection betreft het koppelen van verschillende indexen zodat toch gezocht kan worden op een combinatie van sleutels die niet in een enkele index aanwezig is. Voorwaarde hiervoor is wel dat beide indexen een gemeenschappelijke unieke kolom bevatten. De query-optimizer kan vervolgens gebruikmaken van deze optimalisatiemogelijkheid. In afbeelding 1 is te zien dat twee indexen worden gecombineerd, met behulp van een INNER JOIN, om het gewenste resultaat samen te stellen.

Covering betreft het afdekken van de gevraagde kolommen van een query (in zowel SELECT-, JOIN-, en WHERE-clause) aan de hand van de informatie uit een index. Een clustered index is altijd een covering index, omdat alle data in deze index zijn opgeslagen. Een non-clustered index is alleen een covering index, indien alle gevraagde kolommen uit de query worden

afgedekt door de index. Omdat de index alle gevraagde informatie bevat, hoeft SQL Server de data niet meer op te zoeken in de tabel. Dit beperkt het aantal IO-acties tot een minimum en verbetert de performance van de query maximaal. Als een covering index nauwelijks blijkt te worden gebruikt, kan het zijn dat de performancewinst niet opweegt tegen de overhead van het onderhoud op de index.

Tip: Cover niet elke query, maar beperk je tot veel gebruikte, trage queries en vind de juiste balans om tot een optimale performance te komen.

Tip: Om covering van non-clustered index-intersection te kunnen realiseren, is het beter om een paar brede, in plaats van veel smalle non-clustered indexen te hebben. De SQL Server query-optimizer heeft hierdoor meer opties om de queries te optimaliseren.

In SQL Server 2005 is het mogelijk om non-clustered indexen uit te breiden met extra informatie om covering te realiseren. Deze extra informatie maakt geen deel uit van de sleutel van de index, maar zorgt ervoor dat geen additionele bookmark-lookup uitgevoerd hoeft te worden. Bij het creëren van een index kan extra informatie worden toegevoegd met behulp van het INCLUDE-statement. Alle SQL Server-datatypen worden hierbij ondersteund. Codevoorbeeld 3 geeft de syntax aan.
Let wel, de informatie in de INCLUDE-kolommen wordt dus gedupliceerd in de index, maar er zijn situaties waar dit erg handig kan zijn! Geïndexeerde views op een enkele tabel kennen een vergelijkbare optie als INCLUDE, maar bieden zelfs meer mogelijkheden. Geïndexeerde views kunnen kolommen bevatten die bestaan uit functies, berekeningen of aggregaties. Het gebruik van geïndexeerde views bij complexe joins is niet aan te raden. Aangezien er in dat geval veel data moeten worden gedupliceerd, kan dit negatieven gevolgen hebben voor de performance. Dus … gebruik ze, maar gebruik ze verstandig.
De Database Engine Tuning Advisor (DTA), die standaard onderdeel uitmaakt van SQL Server 2005, kan je helpen bij het optimaliseren van de aanwezige indexen. Maak gebruik van de adviezen die de DTA aandraagt. Beoordeel echter wel in hoeverre de adviezen bijdragen tot performanceverbetering; het onderhouden van indexen kost namelijk ook performance. Een fractionele verbetering in performance van de query kan hierdoor zorgen voor een verslechtering van de performance van andere databaseonderdelen. Bedenk en test dus goed of de aanbeveling wel nodig is.

Optimaliseer AND-queries

Elk deel van een AND-statement limiteert het aantal records in de resultaatset van de query, waarbij elke conditie in het statement 'waar' moet zijn. De query-optimizer beoordeelt alle delen van de WHERE-clause en bepaalt vervolgens wat de te gebruiken indexstrategie is. Er zijn drie opties:

- **Een van de condities is selectief**
  In dit geval is het aan te raden een clustered of non-clustered index te maken op basis van deze selectieve sleutel. De query-optimizer voert dan een bookmark-lookup uit om de gehele resultaatset samen te stellen.

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
   ON <object> ( column [ ASC | DESC ] [ ,...n ] )
   [ INCLUDE ( column_name [ ,...n ] ) ]
   [ WITH ( <relational_index_option> [ ,...n ] ) ]
   [ ON { partition_scheme_name ( column_name )
      | filegroup_name
      | default
      }
   ]
[ ; ]
```

Codevoorbeeld 3. CREATE INDEX-statement

- **Een combinatie van condities is gezamenlijk selectief**
  Dit geval is vergelijkbaar met de bovenstaande optie. Een clustered of non-clustered index op basis van deze combinatie van sleutels is ook hier de beste oplossing. De query-optimizer voert dan eveneens een bookmark-lookup uit om de resultaatset samen te stellen. Neem als volgorde van de AND-delen de meest selectieve als eerste, of gebruik de meest logische en meest gebruikte volgorde.
- **Geen combinatie van condities is selectief**
  Zoals in de voorgaande paragrafen is aangeven, is een bookmark-lookup hiervoor niet aan te raden. In dit geval is covering de beste oplossing.

Als de resultaatsets en de opgegeven parameters dusdanig variëren dat de query-optimizer elke keer een andere index wil gebruiken, is het aan te raden om meer indexen aan te maken die allemaal de query coveren. Dit is echter afhankelijk van de query en moet dus voor elke situatie opnieuw worden bekeken. Kies ervoor om alleen die queries te optimaliseren die het belangrijkste (veelgebruikt) en het meest tijdrovend (in uitvoering) zijn. Optimaliseer deze queries door de indexen aan te passen of uit te breiden of door nieuwe indexen aan te maken. Als gevolg van deze optimalisatie zullen ook andere queries gebruikmaken van gewijzigde en nieuwe indexen, waardoor deze queries ook beter performen.

Optimaliseer OR-queries

Wat doet een OR-query eigenlijk? In een OR-query worden als eerste de individuele resultaatsets bepaald, daarna worden deze resultaatsets samengevoegd en als laatste worden de dubbele records verwijderd. In plaats van een OR-constructie kan in sommige gevallen ook gebruik worden gemaakt van UNION. UNION is sneller dan OR. Let wel op, de werking van beide constructies is verschillend. Het verschil zit in het verwijderen van dubbele records. OR verwijdert de dubbele records op basis van de row identifier of de sleutel van de clustered index, UNION doet het op basis van dubbele records in de resultaatset. Het overstappen van OR naar UNION kan dus leiden tot een ander resultaat. Zorg daarom voor een grondige test van de aangepaste queries.

TIP: *Als bij voorbaat bekend is dat er geen dubbele records aanwezig zijn in de resultaatsets, kan de performance echt goed worden verbeterd. Gebruik in dit geval UNION ALL in plaats van OR. De stap die de dubbele records uit de resultaatset verwijdert, hoeft niet meer uitgevoerd te worden en de performance verbetert aanzienlijk.*

De belangrijkste tips voor ontwikkelaars

Zoals aangegeven zou iedere ontwikkelaar een hoeveelheid basiskennis van databases moeten hebben. Om er voor te zorgen dat de performance op een acceptabel niveau komt en blijft liggen, moet je deze kennis toepassen bij het ontwerpen van tabellen en indexen en het schrijven van queries.
Het is aan te raden de volgende uitgangspunten te hanteren bij het ontwerpen of optimaliseren van een database:

1. Creëer voor elke tabel een clustered index op basis van de principes: uniek, smal, statisch en oplopend.
2. Creëer unieke non-clustered indexen voor alle andere unieke sleutels die aanwezig zijn in de tabel.
3. Creëer een non-clustered index voor iedere kolom die een foreign key-relatie heeft met een andere tabel.
4. Test de werking van de database en maak hierbij gebruik van realistische testdata. Gebruik de aanbevelingen van de Database Engine Tuning Advisor. Ga wel verstandig met deze aanbevelingen om.
5. Overweeg de toepassing van covering om de performance van veelgebruikte en de meest tijdrovende queries te verbeteren. Indexen die meer kolommen bevatten, bieden meer mogelijkheden om queries te coveren.
6. Zorg ervoor dat de statistieken altijd bijgewerkt zijn.
7. Gebruik het executieplan om inzicht te krijgen in potentiële problemen
8. Overweeg het gebruik van UNION (ALL) in plaats van OR
9. Test, test, test.

Het toepassen van deze uitgangspunten betekent niet dat er nooit meer performanceproblemen optreden, maar ze dragen wel bij een stabielere database met een beter voorspelbaar gedrag. Als indexoptimalisatie is doorgevoerd en er is nog steeds behoefte aan performanceverbetering, denk dan als vervolgstap aan de optimalisatie van stored procedures.

Robert Tusveld is als softwarearchitect werkzaam bij Capgemini (www.nl.capgemini.com). Hij heeft zich gespecialiseerd op het gebied van softwareontwikkeling met behulp van producten en technologieën van Microsoft. Robert is te bereiken via Robert.Tusveld@Capgemini.com.

Melchior Brandt Corstius is softwarearchitect bij Capgemini. Hij houdt zich voornamelijk bezig met Service Oriented Architecture op het .Net-platform. Melchior is te bereiken via Melchior.BrandtCorstius@Capgemini.com.

Referenties

www.sqlskills.com
www.sqljunkies.com
www.sqlteam.com
www.microsoft.com/SQL/2005
msdn.microsoft.com/SQL