‘Universele’ mogelijkheden binnen het rdbms

Goede performance is geen Utopia

Toon Loonen

In de opzet van zijn systeem krijgt de dba, naast de applicatie-architect, te maken met een groot aantal verschillende aspecten. Houdt hij met deze punten rekening, dan ligt een goede performance binnen bereik. Dit artikel gaat allereerst in op de vraag wat een goede performance eigenlijk is. Vervolgens wordt gekeken naar de mogelijkheden daarvoor in het rdbms.

Een goede performance is in eerste instantie dat wat de gebruiker als zodanig zal ervaren. We moeten hierbij rekening houden met verschillende omstandigheden.

- Het intoetsen van een letter of cijfer in een veld op het scherm moet direct zichtbaar zijn op het scherm, dus zonder waarneembare vertraging.
- Opgevraagde eenvoudige gegevens, bijvoorbeeld van één record (artikelgegevens), moeten binnen één seconde op het scherm staan.
- Opslag van nieuwe gegevens of wijzigingen op bestaande gegevens moeten binnen één tot drie seconden worden uitgevoerd, afhankelijk van de hoeveelheid gegevens: updates op één record in één seconde; een order met een aantal orderregels en mogelijk nog andere gerelateerde gegevens -zoals een afleveradres- mag twee tot drie seconden duren.
- Bij het opvragen van een eenvoudig lijstje, een factuur etcetera mag het - afhankelijk van de complexiteit en de hoeveelheid te benaderen gegevens - van enkele seconden tot een minuut duren voordat het lijstje op het scherm gepresenteerd of op de printer afgedrukt wordt.
- Grote batchverwerkingen of overzichten waarin veel gegevens moeten worden doorlopen en/of berekeningen vergen, mogen langer duren. Hierbij is het vaak noodzakelijk enkele uren als doorloop te accepteren of zelfs te aanvaarden dat de verwerking alleen ‘s nachts of in het weekeinde kan worden uitgevoerd.

Een goede performance is in eerste instantie wat de gebruiker als zodanig zal ervaren.

Basistips voor performance (1)

Er is waarschijnlijk geen onderwerp waarover door dba’s zoveel wordt gepraat en niettemin zo weinig is gepubliceerd als over performance in een (r)dbms. In een serie van drie artikelen beschrijft Toon Loonen de belangrijkste punten waarmee men in het database-ontwerp rekening moet houden om tenminste een basis voor een goede performance te leggen.
Hoewel veel performance-aspecten productspecifiek zijn -om die onder de knie te krijgen is het volgen van een cursus (of handboek) “Performance en tuning” van de betrokken leverancier noodzakelijk- zijn er veel aspecten of tips die *vendor*-onafhankelijk zijn. En die moeten ook zeker toegepast worden alvorens naar de leveranciereigen problemen te kijken. Dit drieluik beperkt zich tot deze ‘universele’ aspecten en tips.
In dit eerste deel staan de mogelijkheden binnen het rdbms centraal. Deel 2 schenkt aandacht aan de mogelijkheden in de applicatie, en de reeks sluit af door in te zoomen op de rol van de hardware.

ALGEMENE EN SPECIFIEKE EISEN

Het lijken algemeen aanvaarde eisen, maar toch zien we frequent dat webpagina’s op Internet tot een minuut uittrekken om op het scherm te verschijnen. Normen zijn blijkbaar aan verandering onderhevig.

Soms zullen aan functies heel specifieke performance-eisen gesteld worden. Dit moet dan in het ontwerp van de betreffende functie worden vermeld, terwijl voor de andere functies de (default) eisen uit de

standaards gelden. Deze kunnen globaal overeenkomen met bovenstaande eisen. Binnen één functie kan vaak een groot aantal query's naar een database gaan voordat de taken van de functie zijn uitgevoerd, bijvoorbeeld een aantal validaties (bestaat uit deze en gene code in de referentietabellen) en een aantal inserts en updates, bijvoorbeeld het inserten van order en orderregel en het bijwerken van de voorraad bij het artikel en het uitstaand saldo bij de klant. Om de gebruiker de hierboven omschreven performance te garanderen, zullen de SQL-statements `SELECT`, `INSERT`, `UPDATE` en `DELETE` een fractie van deze tijd mogen duren - vaak minder dan een tiende van een seconde.

LOGISCH GEGEVENSMODEL, VERTALING NAAR FYSIEKE MODEL

Het logisch gegevensmodel moet in eerste instantie zuiver zijn en worden opgesteld volgens de regels (voor normaliseren enzovoort; zie ook de literatuurverwijzingen 2, 3, 4, 5 en 7). Performance-aspecten komen pas aan de orde bij het vertalen van het logisch naar het fysieke model. Dikwijls, als de twee modellen sterk op elkaar lijken -dus als er weinig afwijkingen zijn tussen logisch en fysiek model- kan men besluiten één model te onderhouden nadat deze vertaling is gemaakt. In dit geval wordt in praktijk het fysieke

Performance-aspecten komen pas aan de orde bij de vertaling van logisch naar fysiek model

model gepromoveerd tot het te onderhouden model. In het gecombineerde model zijn dan wel alle performance-aspecten verwerkt.

Bij de vertaling naar het fysieke model heeft de dba de volgende mogelijkheden om de basis te leggen voor een goede performance:

- plaatsen van indexen;
- distributie van de fysieke database of tabellen over meer schijven;
- toevoegen van redundante gegevens;
- herindeling van tabellen;
- gebruik van de fysieke gegevenstypen.

Deze aspecten worden hierna beschreven. In het volgende artikel komt nog een aantal performance-aspecten aan de orde waarmee tijdens de bouw van de applicatie rekening gehouden moet worden.

TOCH ZIEN WE FREQUENT DAT WEBPAGINA'S OP INTERNET TOT EEN MINUUT UITTREKKEN OM OP HET SCHERM TE VERSCHIJNEN. NORMEN ZIJN BLIJKBAAR AAN VERANDERING ONDERHEVIG.

INDEXEN

In het algemeen gebruik ik een Case-tool voor het vastleggen van het gegevensmodel. Bij het genereren van de `CREATE TABLE`-definities zal dit tool automatisch (unieke) indexen genereren op de primary key en de candidate keys en (niet-unieke) indexen op de foreign keys. Dit is een goede default, maar aan deze keuzen valt vaak nog wat te verbeteren door (in het Case-tool) op te geven waar nog meer indexen moeten komen of waar ze juist niet nodig zijn.

Een index op de `FOREIGN KEY`-kolommen is natuurlijk een default. Vaak worden deze kolommen gebruikt in een join of op een scherm als zoekargument opgegeven. In dit geval is een index inderdaad nodig. Maar het is zeker noodzakelijk na te gaan of:

- deze foreign key inderdaad zo gebruikt wordt - indien niet, dan is deze index niet nodig;
- er nog andere kolommen zijn waarop het systeem of de gebruiker gegevens kan selecteren of sorteren. Ook op die kolommen of een combinatie daarvan kan een index nodig zijn.

Sommige producten bieden verschillende soorten indexen. Voor maximaal één index kan worden aangegeven dat de gegevens in de tabel geheel op deze volgorde worden gehouden: de *clustered index*. Dit kan voordelig zijn als gegevens uit de tabel vaak op deze volgorde gesorteerd of gegroepeerd worden opgevraagd, bijvoorbeeld de orderregels van een order (met `PRIMARY KEY`: ordernummer, orderregelnummer).

De meeste indexen bevatten een B-tree-structuur. Het hoogste niveau van deze index heeft één pagina met een grove indeling van de index. Deze verwijst naar een tweede niveau met een verfijndere indeling etcetera tot -op het laagste niveau- de gegevens zelf worden gelezen. Bij een grote tabel gaat het al gauw om in totaal vier niveaus en vergt het dus ook

vier fysieke I/O's om een record op te halen. Bij een index met grote kolommen passen er minder rijen in de indexpagina's en moet eerder een extra niveau worden ingezet. Dit gaat onder meer op bij een naam van veertig posities of een index met een groot aantal kolommen. Het kan lonend zijn een gegenereerd nummer als primaire sleutel te introduceren en alle verwijzingen -foreign keys in andere tabellen- naar dit nummer te doen in plaats van naar de lange logische sleutel.

Er zijn ook indexen die de pagina (page) waar de gegevens staan, berekenen door uit de sleutelgegevens een getal uit te rekenen. Dit getal (modulo het aantal pages van de tabel) geeft dan de page aan waar de gegevens zijn te vinden. Dit

Bedenk eerst wat conceptueel correct is en documenteer dit in het logisch model

is snel; er is voor de index geen ruimte en geen I/O nodig. Nadeel is echter dat de sleutel volledig bekend moet zijn, terwijl de index meestal niet gebruikt kan worden om de gegevens meteen gesorteerd op te halen. Bovendien kan hierbij een pagina vol raken. De nieuwe gegevens moeten vervolgens naar een overflow-gebied, en dat brengt wel weer extra I/O voort. Bij deze indexen is het dan ook zaak goed bij te houden of de overflow niet te veel gegevens bevat. Deze wijze van indexering is niet bij alle leveranciers beschikbaar.

Mogelijk is de primaire sleutel een gegenereerd volgnummer en worden alle records toegevoegd aan de tabel op deze volgorde; de tabel ligt gesorteerd op deze sleutel. Dan zal telkens de meeste activiteit op de laatste pagina van de tabel plaatsvinden (*hot spots*). Is het een tabel waarbij veel gebruikers tegelijk nieuwe gegevens invoeren, dan kan dit tot vertragingen door locking en soms zelfs tot deadlocks leiden.

Indien alle gegevens die het systeem opvraagt al in een index zitten, zal het dbms vaak alleen de index pages lezen en niet meer het laagste niveau, de data pages zelf. Dit versnelt het opvragen. Het kan daarom soms de moeite waard zijn een extra veel gebruikte kolom toe te voegen aan een index of een index speciaal voor dit doel te definiëren.

Bedenk dat indexen ook voor overhead zorgen bij inserts en deletes en soms bij updates. Vermijd daarom indexen die toch bijna nooit gebruikt worden. Is een index alleen nodig voor een enkel rapport of (nachtelijke) batchverwerking, dan is het ook mogelijk aan het begin van deze verwerking de index op te bouwen en aan het einde weer te verwijderen. Als vuistregel wordt wel eens aangegeven dat je niet meer dan drie tot vijf indexen moet definiëren. Maar dit hangt natuurlijk sterk af van het gebruik van de tabel.

Indexen in kleine tabellen

Kleine tabellen vragen niet om indexen, behalve die welke ook de unieke waarde van een sleutel bewaken. Ik hanteer de volgende regel:

- is de tabel kleiner dan 8 pages (van 2 Kb), dan geen indexen;
- is de tabel groter dan 32 pages, dan wel indexen.

Daartussen ligt een grijs gebied: als de tabel voornamelijk gelezen wordt, dan wel indexen; wordt er voornamelijk gewijzigd, dan geen indexen. Als de tabel veel indexen nodig zou hebben, dan maar liever helemaal geen indexen.

DISTRIBUTIE OVER MEER SCHIJVEN

De performance van een (zwaar gebruikt) systeem met een zeer grote database valt verder te verbeteren door delen van de database over verschillende fysieke schijven te verdelen. Bijvoorbeeld een schijf voor de gewone gegevens en indexen, een tweede schijf voor de logging, een derde schijf voor werktabellen of tijdelijke tabellen en nog een vierde schijf voor de backups, mits die niet direct naar tape gaan. Zo zullen veel acties tegelijk van twee schijven gebruik maken:

- een insert of update verdeelt de I/O over de data- en de log-schijf;
- een `SELECT` met `ORDER BY`-optie verdeelt de I/O over de dataschijf en de schijf met werktabellen voor de sortering;
- de backup zal lezen op de dataschijf en schrijven op de backup-schijf enzovoort.

Een stap verder is grote en/of veel gebruikte tabellen te verdelen over meer schijven, bijvoorbeeld een schijf voor de data en een tweede schijf voor de indexen. Een andere methode is het splitsen van de gegevens van een grote tabel. Bijvoorbeeld alle bankrekening-records worden over tien schijven verdeeld, gebaseerd op de laatste positie van het bankrekeningnummer. Omdat dit een checkdigit is (voor de 11-proef) mag van dit nummer een gelijkmatige verdeling verwacht worden.

Gebruik van dit soort distributies maakt regelmatige controle van de verdeling nodig. Eventueel kan deze later worden bijgesteld aan de hand van de gevonden cijfers.

HERINDELING LOGISCHE ENTITEITEN IN FYSIEKE TABELLEN

Bij het fysiek ontwerp kan ook gekeken worden of gegevens in de tabellen anders verdeeld worden dan in het logisch model.

- Verticale distributie op basis van de kolommen: als een tabel een grote tekstkolom bevat of als een deel van de kolommen vaak leeg is of relatief weinig gelezen wordt, dan kan dit deel naar een tweede tabel worden afgesplitst. Er ontstaat dan een 1-op-1 (of 1-op-0,1) relatie tussen beide tabellen. Nadeel: de werkwijze vergt extra I/O als gegevens van beide tabellen nodig zijn.[3]
- Horizontale distributie op basis van de rijen: is een deel van de gegevens niet zo vaak meer nodig, dan kan dat deel

worden verplaatst naar een andere tabel of naar een archief op cd-rom. Te denken valt aan afgehandelde gegevens van meer dan een maand of jaar oud die nog maar zelden geraadpleegd worden. Bedenk dat prikken op een kleine tabel via een index nauwelijks sneller is dan op een grote tabel, terwijl het doorlopen van de hele tabel (table scan) wel significant sneller zal zijn. Nadeel is dat men beide tabellen moet lezen als zowel oude als nieuwe gegevens nodig zijn. Zo'n union zal, in combinatie met een join op andere tabellen, tot significant complexere coding leiden.

REDUNDANTE GEGEVENS

Vaak moeten we een beroep doen op gegevens die berekend moeten worden uit een veelheid van andere gegevens. Bijvoorbeeld het uitstaande saldo van een

Vermijd indexen die toch bijna nooit gebruikt worden

klant dat is te becijferen door de gegevens van alle facturen, orders en betalingen te combineren.[2] Als dit gegeven vaak nodig is, kan het raadzaam zijn om het redundant bij de klant op te slaan. Bij het opvragen van de klantgegevens inclusief saldo kan het systeem volstaan met het opvragen van één klantrecord in plaats van alle op het saldo gebaseerde gegevens. Nadeel is een extra I/O bij invoering of wijziging van een factuur, order of betaling. Tevens moet deze coding zo geschreven worden dat de kans op fouten in het bijwerken van het saldo minimaal (= nul?!) is. Ik zelf heb hierbij voorkeur voor triggers op tabellen waarop het betreffende saldo gebaseerd is: `INSERT`, `UPDATE` en `DELETE`.

FYSIEKE GEGEVENSTYPEN

Overweeg bij het opstellen van het fysieke model goed in welke typen de gegevens worden opgeslagen. Een getal (aantal, volgnummer en dergelijke) is in een *integer*, *float* of *numeric/decimal* type op te slaan. Een integer zal een betere performance geven, zeker als dit attribuut ook in een index staat of als men hierop sorteert.

Gebruik de optie van `NULL / NOT NULL` correct. De eerste (toegestaan) heeft enige overhead ten opzichte van de laatste. Gebruik dus NULL alleen als de kolom echt optioneel gevuld kan worden.

Bedenk bij dit alles eerst wat conceptueel correct is en documenteer dit in het logisch model. Documenteer goed de overwegingen als bij het opstellen van het fysieke model wordt afgeweken van het logische model, zodat dit later geen vragen gaat oproepen en mogelijk zelfs wordt teruggedraaid.

GEDISTRIBUEERDE SYSTEMEN

Gebruikt een organisatie meer systemen die onderling gegevens uitwisselen, dan zal een systeem A vaak gegevens nodig hebben van systeem B. 'Facturering' put bijvoorbeeld uit data van 'Klantbeheer'. Deze gegevens kunnen bij de huidige stand van techniek online worden opgevraagd, zelfs al staan ze op een andere computer, mogelijk op een rdbms van een andere leverancier, of zelfs in een ERP-pakket.[1] Zo'n opvraging duurt echter altijd langer dan wanneer deze gegevens lokaal staan. Mocht het ophalen onaanvaardbaar lang duren, dan kan men een kopie (replicatie) van de gegevens van het andere systeem in het eigen systeem plaatsen. Ook kan men met zo'n kopie werken als het risico te groot is dat het andere systeem of het tussenliggende netwerk down is, waardoor de gegevens niet opgehaald kunnen worden. De keerzijde van de medaille is natuurlijk dat deze redundante gegevens bijgewerkt moeten worden op een wijziging van de gegevens in het andere systeem. De meest betrouwbare manier hiervoor is gebruik te maken van de replicatiefunctionaliteit van het gebruikte dbms.

In het volgende deel van dit artikel wordt ingegaan op de mogelijkheden in de applicatie voor een goede databaseperformance. ●

Literatuur

1. Loonen, *Gegevenskoppelingen in een 4GL/RDBMS omgeving.* Database Magazine 8/1996.
2. Loonen, *Gebruik van afgeleide gegevens in ontwerp en bouw.* DB/M 1/1997.
3. Loonen, *Modelleren van subtypes.* DB/M 1/1999.
4. Loonen, *Naamgeving van objecten in een RDBMS.* DB/M 4/1998.
5. Loonen, *Mutatierapportage, de tijdgeest van de database.* DB/M 3/1999.
6. Loonen, *Het schrijven van een nette WHERE clause.* DB/M 7/1997.
7. Loonen, *Gegevensmodel van een distributed data dictionary.* DB/M 4/1997.
8. Loonen, *Ontwikkelstraat, hergebruik door inzet van architectuur.* Software Release 7-8/2000.
9. Smit e.a., *In memory-database, einde van een religiestrijd?* DB/M 3/2000.
10. R.F. van der Lans, *Tijd was rijp voor veelbelovend TimesTen.* DB/M 6/2001.

Persoonsnamen opslaan

Voor het opslaan van langere, maximaal 24 of 36 posities gedefinieerd character-gegevens kan beter een *varchar* dan een *char* type gebruikt worden. Het kan bijvoorbeeld gaan om een persoonsnaam. Omdat een naam meestal niet langer is dan tien posities, bespaart dit extra ruimte. Zo passen meer records op een page, waardoor de performance verbetert. Een varchar heeft in het dbms echter enige overhead ten opzichte van een char, zeker als bij een update de lengte wijzigt. Gebruik de varchar daarom niet als de kolom meestal volledig of bijna volledig gevuld is, bijvoorbeeld bij een postcode.

Toon Loonen (toon.loonen@cgey.nl of toon.loonen@inter.nl.net) is als consultant werkzaam bij Cap Gemini Ernst & Young.