Goede performance is geen Utopia (2)

Keuzen maken voor de applicatie

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 tweede artikel uit een drieluik onderzoekt de mogelijkheden daarvoor in de applicatie, om te beginnen door het gebruik van stored procedures, die worden verwerkt op de server.

Laten we eens kijken naar de volgende situatie. Een client vraagt een overzicht op scherm of rapport aan waarvoor veel selecties op de database nodig zijn. Mogelijk wordt deze aanvraag gecombineerd met andere bewerkingen. Ook kan het zijn dat uiteindelijk maar weinig gegevens teruggaan naar de client. De meest efficiënte, eenvoudige -is ook: onderhoudbare- en betrouwbare manier om dit te schrijven is doorgaans als volgt:

- de client roept een stored procedure aan die alle selecties en bewerkingen op de server uitvoert. Aan het einde wordt het resultaat teruggegeven aan de client;
- de client verzorgt alleen de opmaak van de teruggekomen gegevens en berekent hooguit nog een totaalregel over de ontvangen records.

Deze werkwijze geeft bijna altijd de beste performance, omdat bijna alles binnen de server kan worden afgehandeld en er dus weinig communicatie tussen client en server nodig is. Verder is de SQL-code vaak beter te lezen dan code van de client of een combinatie van clientcode met nog wat selecties en bewerkingen in de stored procedures. Natuurlijk moet de server voldoende capaciteit hebben om bewerkingen voor soms heel veel clients te kunnen verwerken.

Ook voor minder complexe handelingen -insert een record, verwijder alle records die ouder zijn dan een bepaalde datum- zal een stored procedure meestal (iets) sneller zijn dan het 'losse' SQL-commando.

Een stored procedure is echter niet altijd sneller. Op een scherm kan een gebruiker gegevens opvragen aan de hand van een aantal selectievelden; query by form, bijvoorbeeld: selecteer alle klanten aan de hand van de opgegeven achternaam en/of postcode en/of woonplaats en/of klanttype en/of ... Hiervoor kan een ingewikkeld SQL-commando worden geschreven in de trant van:

```
SELECT naam, straatnaam, huisnummer, postcode,
  woonplaats
FROM klant
WHERE (naam LIKE <opgegeven naam> OR <opgegeven
  naam> is NULL)
AND (postcode LIKE <opgegeven postcode> OR <opgegeven
  postcode> is NULL)
AND etc.
```

Zeer waarschijnlijk zal de optimizer hier een table scan kiezen om de gegevens op te halen. Veel 4GL's kunnen echter een query genereren die alleen de ingevulde velden bevat en waarin de "OR .. is NULL" dus niet nodig is. Deze query wordt geëvalueerd en geoptimaliseerd. Waarschijnlijk is hier wel een goede index te gebruiken en zal de losse query een betere performance geven.

Een ander probleem met bovenstaande query is dat de gebruiker selectieparameters kan inbrengen waaraan zeer veel records voldoen (woonplaats like "A%"). De databaseserver moet dan een

Basistips voor performance (2)

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. Hij beperkt zich tot de 'universele' *vendor*-onafhankelijke aspecten en tips.
In deel 1 (in het vorige nummer van DB/M) stonden de mogelijkheden binnen het rdbms centraal. Dit tweede deel schenkt aandacht aan de rol van de applicatie. De reeks sluit in DB/M 3 af door in te zoomen op de hardware en het belang van het analyseren en testen van de performance.

enorme hoeveelheid gegevens lezen, naar de client versturen en op de client verwerken. Op elk punt (databaseserver, netwerk en op de client) kan dit tot performanceproblemen leiden, op de client mogelijk zelfs tot vastlopen van het programma. Ook andere gebruikers hebben hiervan last; hun responstijden zullen eveneens teruglopen. Om dit probleem te voorkomen kan men voor een bepaald soort functies (bijvoorbeeld het opzoeken van een klantnummer [zie ook 8]) afspreken dat maximaal 100 records worden opgehaald. Zit de gezochte klant er niet bij, dan kan de gebruiker andere of meer zoekcriteria opgeven of mogelijk opdracht geven het aantal terug te geven rijen te verhogen (telkens met 100). De gegevens mogen hierbij ook niet opgevraagd worden met een

Ook voor minder complexe situaties zal een stored procedure meestal sneller zijn

order by clause. In dat geval haalt het dmbs dan toch eerst alle gegevens op, sorteert ze en geeft er slechts 100 terug, in plaats van er 100 op te halen en die gesorteerd terug te geven.

Let ook op als een gebruiker geautoriseerd is zelf query's uit te voeren op de database, buiten de applicatie om. Hierbij kunnen zij gemakkelijk query's schrijven die een enorm performanceprobleem geven, waarvan ook alle andere gebruikers last van hebben. Beter is de gebruiker alleen de database te laten benaderen via de applicatie. Voor eventuele incidentele vragen kan het onderhoudsteam stored procedures schrijven en testen. Via deze stored procedures (met de goede parameters) kan de gebruiker extra query's uitvoeren zonder dat veel overhead nodig is voor het schrijven van nieuwe programma's.

TRIGGERS, RULES EN CONSTRAINTS

Triggers op tabellen zijn stukjes coding die worden afgevuurd telkens wanneer op een tabel een insert, update of delete wordt uitgevoerd. Rules en constraints bevatten validaties die op de tabel of een kolom gedefinieerd worden, bijvoorbeeld de primary en foreign key constraints of een rule die bepaalt dat de kolom sexe in de tabel Medewerker alleen de waarde M of V mag hebben. Triggers zijn bruikbaar voor ingewikkelde validaties die (bijna) niet in een rule of constraint te definiëren zijn. Verder gebruik ik triggers voor het bijhouden van redundante gegevens (saldo klant, zie elders in dit artikel) of historie [zie 5].

Bedenk dat een trigger enige overhead geeft, zelfs al wordt geen coding uitgevoerd omdat de trigger bijvoorbeeld alleen gebruikt wordt als een bepaalde kolom gewijzigd is. Ook rules en constraints zorgen voor overhead.

Bij client/server-toepassingen op een Windows-pc of een Unix-programma en VT220-terminal als client worden controles vaak tweemaal uitgevoerd:

- eerst als de gebruiker het veld verlaat (is de waarde correct, bestaat deze klant?);
- en daarna nogmaals bij een insert in de database door het afvuren van de trigger, rule of constraint.

Bij blokmode-terminals (3270 van IBM-mainframe of de schermen van een AS400) zal men vaak eerst een heel scherm voltypen, waarna de inhoud door het systeem wordt gecontroleerd en verwerkt. Dit gebeurt meestal ook zo bij webapplicaties als er validaties tegen de database nodig zijn, teneinde communicatie via Internet tussen client en ver verwijderde server te vermijden.

De eerste methode (meteen controleren) is gebruikersvriendelijker (de gebruiker wordt direct op zijn fout gewezen), maar levert wel meer overhead op, omdat een controle tweemaal wordt uitgevoerd: tweemaal het lezen van de klanttabel in de database. Het alternatief -de tweede controle overslaan- is ook minder gewenst, omdat tussen eerste controle en feitelijke update de inhoud van de database gewijzigd kan zijn, waardoor inconsistenties kunnen ontstaan.

Dit alles is natuurlijk een kwestie van standaardisatie of architectuur. De overhead van extra controles -voor de performance, maar ook dubbele coding in de programmatuur- is af te wegen tegen voordelen (gebruikersvriendelijk) en mogelijkheden - op een blokmode-terminal is er geen keuze.

NETTE WHERE CLAUSE

Het schrijven van nette coding leidt zeker tot minder fouten en een beter onderhoudbaar systeem, maar mogelijk ook tot een betere performance of tenminste tot coding waarmee een performanceprobleem gemakkelijker valt te analyseren. Met name van de where clause weten sommige programmeurs een onoverzichtelijke kluwen te maken, waarin zij zelf ook nauwelijks meer een begin of einde ontwaren. De volgende regels gegeven een nette, goed leesbare en onderhoudbare where clause [zie voor een uitgebreide toelichting ook 6].

De where clause dient steeds geformuleerd te worden met links van de operator (zoals =, < en >), een kolom uit de tabel en rechts van de operator de waarde waarmee deze kolom wordt vergeleken; een constante, een programmavariabele, een andere kolom uit dezelfde rij of een berekening met een combinatie van deze mogelijkheden, bijvoorbeeld: postcode = "1234AA".

De where clause dient te beginnen met de condities voor de primaire sleutel, vervolgens de condities voor eventuele kandidaatsleutels, dan de condities op buitensleutels of op de kolommen

Transacties moeten zo kort mogelijk worden gehouden

waarop (waarschijnlijk) een index staat en daarna de overige condities. Natuurlijk kunnen later indexen worden verwijderd of toegevoegd, zonder dat dit leidt tot aanpassingen in de query.

Bij een join van twee of meer tabellen worden eerst de condities voor de eerste tabel, daarna de condities voor de tweede geschre-

ven enzovoort. Bij de tweede tabel begint men met de conditie die deze tabel joint met de eerste tabel. De condities voor de tweede tabel zijn verder gelijk aan die voor de eerste; maar rechts van de operator kan nu ook een waarde uit de eerste tabel staan, behalve bijvoorbeeld een constante of een programma-variabele.

Daarna worden de condities geschreven voor de derde tabel, die weer een join krijgt met de tweede of de eerste tabel enzovoort.

Voor een goede performance moeten hierbij de tabellen in de volgende volgorde worden benaderd:

- vermeld eerst de condities voor de tabel die de resultaatset zoveel mogelijk beperkt;
- vermeld vervolgens de condities voor de tweede tabel, die daarbij met de eerste tabel wordt gecombineerd;
- vermeld daarna de condities voor de derde tabel, die daarbij met de eerste of de tweede tabel wordt gecombineerd enzovoort.

Met nog een aantal performance-aspecten moeten we rekening bij het schrijven van de where clause, of mogelijk eerder bij het opstellen van het gegevensmodel:

- voer rechts van de operator geen bewerkingen of berekeningen die voor elk record herhaald moeten worden, bijvoorbeeld het ophalen van de systeemtijd (where < sysdate()). Deze bewerking kan ook voor de query worden uitgevoerd, waarbij het resultaat in een programmavariabele wordt geplaatst;
- gebruik links van de operator zoveel mogelijk een kolom van de tabel en geen bewerking op deze kolom (where upper(naam) = "JAN"), omdat dan waarschijnlijk geen index gebruikt kan worden;
- als links en rechts niet hetzelfde type staat (bijvoorbeeld links een integer en rechts een decimal of float), zal het rdbms (links of rechts) een impliciete conversie plaatsen, met ook hier weer de mogelijkheid dat de index niet gebruikt kan worden;
- bedenk dat bij een vraag als <where name like "%jan%"> geen index gebruikt kan worden, maar bij <where name like "jan>%" wel. Vermijd derhalve query's waarbij een wildcard aan het begin staat en wijs de gebruiker erop dat bij bepaalde query's een langere wachttijd onvermijdelijk is.

OPTIMIZERS

Aan de hand van de coding, al of niet volgens bovenstaande aanwijzingen geschreven, moet de optimizer een querypad bepalen. Bij ingewikkelde query's, zoals die met subselects, behoeft dit niet altijd het meest optimale resultaat te zijn. Het kan de moeite lonen de query zelf te splitsen in enkele delen, met een tijdelijke tabel voor de tussenresultaten.

De optimizer bepaalt zijn pad vaak aan de hand van gegevens over tabel grootte en verdeling van sleutels in de index. Hiervoor moeten deze gegevens wel actueel zijn bijgewerkt. Dit kan met een commando "update statistics" (zie ook de performancetips van uw leverancier). Het is dan ook belangrijk dat dit statement (voor elke tabel) wordt uitgevoerd, zeker als in het begin van het gebruik van het systeem de vullingsgraad van de tabellen nog vaak wijzigt.

Reorganisatie van een tabel kan ook soms verbeteringen geven. Dit is sterk productafhankelijk. Bijvoorbeeld als in een tabel de gegevens op een bepaalde volorde liggen (orders op ordernummer) en oude geheel verwerkte orders uit het systeem worden verwijderd. Als nog enkele oude maar niet verwerkte orders blijven staan, kan het zijn dat een groot aantal datapages slechts één (oude) rij bevat. Deze pagina kan dan niet voor nieuwe gegevens worden gebruikt. Zo zal een tabel scan over deze tabel veel bijna lege pages lezen en dus veel I/O moeten doen. Na reorganisatie zal een table scan veel sneller zijn. Voor het prikken op zo'n tabel zal voordeel minder groot zijn. Bedenk ook dat bij reorganisatie de tabel waarschijnlijk niet door het systeem gebruikt kan worden (tijdelijk gelockt is).

LOCKING EN DEADLOCKS

Als een transactie naar de database wordt geschreven, mogen andere processen (gebruikers) deze gegevens niet zien zolang de transactie niet geheel op de database is verwerkt. Dit wordt door de programmatuur aangegeven door aan het begin van de transactie het commando "begin work" en aan het einde het commando "commit work" of "rollback work" naar de server te sturen. Wil een andere gebruiker deze gegevens wil opvragen, dan zal hij even moeten wachten. De gebruiker zal niet weten waarop hij wacht; hij zit maar naar zijn zandlopertje te staren, en hij ervaart deze situatie gewoon als een slechte performance. Transacties moeten daarom zo kort mogelijk worden gehouden. Dat betekent:

- online functies mogen nooit een transactie open hebben staan terwijl de gebruiker gegevens op het scherm inbrengt of wijzigt. Meestal zal zo'n functie alleen maar lezen -voor validaties tegen de database of om referentiegegevens op te halen; hier-

Grenzen van optimizing

Een performance-cursus van de leverancier zal dieper ingaan op de specifieke problemen bij de optimizer van zijn product. Maak hiervan echter verstandig en beperkt gebruik. Bedenk dat in een volgende versie de optimizer gewijzigd (verbeterd) zal zijn en betere of in elk geval andere conclusies kan trekken. Ik heb meegemaakt dat we een systeem volledig getuned hadden. Alle query's waren uiteindelijk zo geschreven, dat de optimizer het meest efficiënte pad koos om de gegevens op te halen. De optimizer van de volgende versie van het systeem was waarschijnlijk wel beter, maar de query's werden opnieuw bekeken en het kon hierbij eigenlijk niet meer beter; alleen, op enkele punten kan het wel anders. En in ons geval was dat altijd slechter.
Mijn ervaring is dan ook dat bij een zeer goed getuned systeem altijd een goede performancetest noodzakelijk is als op een nieuwe release van het dmbs wordt overgestapt om te controleren of de performance nergens terugloopt.

Andere overhead

Ga na wat nog meer in de architectuur/infrastructuur is opgenomen behalve de client-toepassing zelf (4 GL- of C-programma op pc, in Unix of andere applicatieserver) en het rdbms. Netwerkrouters of koppelingen (mogelijk loopt een deel van de communicatie via Internet), TP-monitors of object brokers en gedistribueerde dbms'en leveren overhead. Dit kan in een gedistribueerde omgeving nodig zijn, maar het kan ook de responstijden sterk nadelig beïnvloeden. Als deze overhead voor een systeem niet nodig is, laat haar dan weg.

voor is geen transactie nodig- en aan het einde -bij een klik op de verwerktoets (of bij Ctrl-S)- zullen de ingetoetste gegevens in één transactie op de database worden verwerkt (optimistic concurrency control);
- batchfuncties die tegelijk met online functies kunnen draaien en die grote hoeveelheden gegevens verwerken, zoals een tape of diskette met orders inlezen, moeten per eenheid (order of zelfs per orderregel) een transactie definiëren.

Alleen een batchfunctie die uitsluitend op de database draait ('s nachts, geen andere batch en geen online gebruikers) mag wel grote transacties gebruiken. Dit is zelfs beter, omdat elke transactie enige overhead geeft en een klein aantal grote transacties sneller verwerkt zal worden dan een groot aantal kleine transacties. Een commando om een tabel in zijn geheel te locken kan gebruikt worden om extra overhead van het locken van individuele pages of records tijdens de transactie te vermijden.

Biedt het rdbms de keuze tussen record locking (row level locking) of page level locking, dan zal page level locking altijd een betere performance geven. Alleen lopen we de kans dat een andere gebruiker op gelockte gegevens moet wachten. Ook de kans op een deadlock is bij page level locking groter. Een deadlock doet zich voor als gebruiker A wacht op een gegeven dat door gebruiker B is gelockt en gebruiker B wacht op een gegeven dat door gebruiker A is gelockt. Dit wordt door het rdbms gesignaleerd, waarna de transactie van één van de twee gebruikers wordt afgebroken en teruggedraaid met een foutmelding aan de gebruiker. Deze kan daarna proberen de transactie opnieuw aan het systeem aan te bieden, wat waarschijnlijk wel goed zal aflopen. Bij een deadlock op transacties in een belangrijk batchproces (verwerking van een financiële transactie bij een bank) zal het batchprogramma de deadlockstatus moeten afvangen en de transactie opnieuw aan het rdbms aanbieden.

BATCHVERWERKING

Kleine lijsten of rapporten, die binnen enkele seconden samengesteld kunnen worden, kunnen het beste direct door het online gedeelte van het systeem worden samengesteld en opgemaakt. Wordt de doorlooptijd langer, dan is het beter dat deze opdracht wordt afgesplitst naar een batchverwerking. Een batchverwerkingsfunctie kan eenvoudig worden gebouwd als volgt [zie ook 8]:

- tabel in de database (de batchqueue) met als kolommen onder meer: userid-aanvrager, datum-tijd-aanvraag, datum-tijd-verwerking, prioriteit, status, rapportcode, parameters;
- bij het online gedeelte van de functie geeft de gebruiker eventuele selectieparameters op en wordt een record in deze tabel geplaatst;
- afhankelijk van de omgeving of programmeertaal draait de batchverwerking op de Unix- of NT-databaseserver of op een speciaal hiervoor ingerichte pc. Dit programma draait continu of bijvoorbeeld dagelijks van 1 uur 's nachts tot 10 uur 's avonds (buiten de backup-tijd). Dit programma leest in de batchqueue geplaatste opdrachten en voert deze uit.

Omdat de opdrachten na elkaar worden uitgevoerd, wordt voorkomen dat een systeem verstopt raakt en de online gebruikers een slechte performance van hun functies ervaren, wanneer een gebruiker een tiental rapporten direct na elkaar opstart. Ook kan van zeer zware functies worden aangegeven dat deze alleen 's nachts of in het weekend mogen draaien. Dit geldt zeker voor functies die intensief wijzigen op de database, bijvoorbeeld een schoningsrun; niet alleen vanwege de belasting van de machine, maar ook in verband met de kans op lockingproblemen.

Verder kunnen we de batchfunctie onder Unix en de hierbij afgevuurde SQL-opdrachten in de databaseserver een lagere prioriteit geven, zodat online gebruikers altijd eerst worden geholpen en een betere performance ervaren.

Naast het geven van online opdrachten kan de batchverwerker elke nacht of weekend beginnen met een aantal vaste opdrachten die dagelijks respectievelijk zijn uit te voeren, zoals vaste rapportages, data-import- en export functies en het opschonen van de gegevens.

In het derde en laatste deel van dit artikel zal worden ingegaan op de relatie tussen hardware en databaseperformance en op het traject van analyse en testen. ●

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.

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