Bottleneck is niet de database maar de taal

Silly SQL: Computers die mensentaal spreken

Rick van Rein

SQL is ontworpen als een taal die 'gewone mensen' moesten kunnen begrijpen. In een tijdperk waarin een terminal-interactie de luxe opvolger was van ponsbanden, was dat wellicht een aardig idee, maar de meeste huidige computergebruikers gruwelen van terminal-sessies. Als we niet kunnen klikken dan komen we welhaast ons bed niet meer uit voor een interactie met de computer. Sterker nog, al dat klikken is voor velen ook nog te complex.

In tegenstelling tot talen die tussen mensen ontstaan als grootste gemene deler van wat men van elkaar begrijpen kan, worden computertalen expliciet ontworpen. De doelstelling daarbij is meestal om de taal helder en goed gestructureerd te maken. Dat is er de oorzaak van dat programmeurs relatief snel fouten kunnen vinden als ze eenmaal de moeite hebben genomen om een programmeertaal te leren kennen. Het is daarbij vooral handig als een keuze in een programma aanwijsbaar is, en niet verspreid ligt over allerlei plekken; een overweging die we ook wel kennen van het ontwerp van een database-schema.

Geneste query's

SQL moest en zou echter op een natuurlijke taal lijken. Het gevolg daarvan is dat de plekken waar dingen worden geregeld niet altijd logisch lijken voor een programmeur; in plaats van SELECT X FROM Y WHERE Z zou die liever schrijven GIVEN Y SELECT X WHERE Z. Vooral voor geneste query's verheldert die volgorde flink. Of neem het gebruik van meerdere keywords in zelden gebruikte statements zoals ALTER TABLE; ook

SQL is een standaard, maar wel een oude. In een serie 'Silly SQL' artikelen bespreken we de dingen die, in retrospect bezien, een stuk handiger hadden gekund.

ervaren databasebeheerders vergeten gemakkelijk of ze RENAME AS of RENAME TO moeten gebruiken om een kolom van naam te doen veranderen. En of het nou zo handig is dat we TABLE met DROP verwijderen en een record met DELETE?
Het bizarre aan de wijze waarop SQL zich heeft ontwikkeld is dat al deze natuurlijke taal tegenwoordig door een computer wordt gegenereerd, overgezonden via een wrapper als ODBC of JDBC, en aan de andere kant geïnterpreteerd door een andere computer. Alsof computers niets beters met hun tijd te doen hebben dan uitgebreide conversaties te houden in proper English. Misschien nog erger dan de commandotaal is het feit dat de uitvoer in tabelvorm wordt aangeboden, en geïnterpreteerd moet worden door middel van row-walks en cursors.
Ook de constructies die SQL kent, zoals geneste query's, lijken ontworpen te zijn om tegemoet te komen aan de beperkte (want niet systematische) vermogens van een mens achter een toetsenbord. Er zijn regelmatig query's die je eigenlijk niet zomaar kunt uitdrukken, omdat je het resultaat uit de buitenste query zou willen verwerken in de binnenste query. En dan is het wel heel hard blokken op een manier om de database uit te leggen hoe wij het als programmeur logisch vinden iets op te bouwen. De bottleneck is daarbij niet de database op zich, maar de *taal* waarin wordt gesproken met die database: SQL.

Optimaliseren

Het ligt nu voor de hand om te roepen dat de database omwille van optimalisatie werkt met nesting en de mogelijkheden van query's inperkt. De werkelijkheid is natuurlijk andersom: er was eerst een taal met enige inperkingen op de uitdrukkingsvrijheid, en op basis van die inperking is het mogelijk gebleken om optimalisaties in te voeren. En dat is een algemeen beeld – de precieze semantiek van een taalconstructie maken het mogelijk om aannames te maken over de gedragingen van die constructie, en die te optimaliseren.

In de praktijk wordt een query vaak dynamisch geoptimaliseerd – als onderdeel van het verwerken van een query wordt getoetst of aan bepaalde randvoorwaarden is voldaan, zodat een optimalisatie mogelijk wordt. Deze aanpak maakt een groter aantal optimalisaties kansrijk dan wanneer slechts van een starre, altijd geldende semantiek per taalconstructie zou worden uitgegaan. Voor bulk query's kan het bovendien zeer goed uitkomen om vooraf even na te denken over de aanstaande klus. En hoewel met ander resultaat, is dit ook mogelijk bij een andere semantiek van de query-taal.
Wat juist indruist tegen optimaliseerbaarheid is de complexiteit

van de taal, en die wordt weer veroorzaakt door gebrek aan orthogonaliteit – in plaats van enkele eenvoudige componenten te zoeken die naar wens op iedere wijze gecombineerd kunnen worden is gekozen voor tamelijk grote (zins)structuren die erg veel varianten vertonen en waarvan de interacties dus snel ingewikkeld worden, en dus moeilijk optimaliseerbaar. Gegeven dit standpunt, dus dat een mensentaal onnodig is en dat het niet louter omwille van optimalisatie nodig is om de huidige taalstructuren te gebruiken, zouden we eens de tijd moeten nemen om te bezinnen. Wellicht komen we dan al heel snel uit op iets als MIL, de huistaal van MonetDB dat we een poosje terug bespraken.

Dynamisch script

Een voorbeeld van een toepassing waar SQL een sta-in-de-weg is, en dus een ongeschikte client/server-interface, is bijvoorbeeld te vinden in sommige scripts die dynamisch SQL statements genereren omdat ze telkens voor andere doeleinden worden aangeroepen. Ga uit van een aantal tabellen die samenhangen via normale foreign key's, en ga uit van een aantal voorwaarden die daaraan gesteld kunnen worden, of niet, al naar gelang de invoer die een gebruiker bijvoorbeeld in een webform maakt. Het is knap lastig om SQL te genereren voor zo'n dynamische situatie. Zo'n dynamisch 'query-end' script kan niet zomaar een FROM clause gebruiken met alle tabellen die mogelijkerwijs mee zouden doen in een voorwaarde op de gezochte items; als er namelijk geen voorwaarde op wordt losgelaten dan ontstaat een enorm kruisproduct van al die onbenutte tabellen met de rest van de data. Dus moet aan een basis-selectie telkens een losse rand-voorwaarde worden toegevoegd en tegelijk de tabel of tabellen waarop die betrekking heeft. Bovendien moet vanuit de door die randvoorwaarde bijgezochte data een join met de basis-data worden gemaakt, omdat anders alsnog een ongewenst kruis-product ontstaat. Dit alles is tamelijk lastig dus. Echt vervelend wordt het wanneer zulke randvoorwaarden met AND en OR gecombineerd zouden moeten worden – de tabellen toevoegen en voorwaarden daarop loslaten gaat nog wel, maar welke JOIN kan worden gebruikt om ze samen te voegen?

Het volgende voorbeeld van een heraldische database licht dit toe aan de hand van een persoon, waarvan een deel van de naam bekend is en een nummer gevraagd wordt.

```
SELECT p.nummer
FROM persoon p
WHERE p.naam LIKE 'Jan %'
```

Om hieraan een test toe te voegen of de persoon met een andere persoon van hetzelfde geslacht getrouwd is moeten we op diverse plaatsen iets toevoegen:

```
SELECT p.nummer
FROM persoon p, persoon q
WHERE p.naam LIKE 'Jan %'
AND p.partner = q.nummer
AND p.geslacht = q.geslacht
```

Als we bovendien willen weten of de hoofdpersonen van deze query een dochtertje genaamd Bep heeft dan moeten we toevoegen:

```
SELECT p.nummer
FROM persoon p, persoon q, persoon r
WHERE p.naam LIKE 'Jan %'
AND p.partner = q.nummer
AND p.geslacht = q.geslacht
AND r.vader = p.nummer
AND r.naam LIKE 'Bep %'
```

Stel nu dat degene die in deze heraldiekdatabase zoekt nu niets gepresenteerd krijgt, dan zou die kunnen kiezen om de AND tussen de laatste twee tests in een OR te veranderen: Ofwel onze hoofdpersoon is met iemand van hetzelfde geslacht getrouwd, ofwel onze persoon heeft een dochtertje genaamd Bep. Wat dan niet gaat werken is:

```
SELECT p.nummer
FROM persoon p, persoon q, persoon r
WHERE p.naam LIKE 'Jan %'
AND ( ( p.partner = q.nummer AND p.geslacht =
                q.geslacht) OR ( r.vader = p.nummer
                        AND r.naam LIKE 'Bep %') )
```

De oorzaak van dat dit spaak loopt is dat elke persoon r volstaat als aan de voorwaarde voor q is ontstaan, zodat er weer een ongewenst kruisproduct ontstaat. De ontwerper van de heraldische database kan zich hier het hoofd over breken, op zoek naar een alternatieve formulering. In dit geval zijn zowel q als r personen, dus daar is nog wel uit te komen, maar in het algemeen geldt dat niet en dan moet de hele structuur van de query worden omgegooid. Bijvoorbeeld met een geneste query. Altijd leuk om SQL leesbaar te houden voor de gemiddelde medemens.
Dit probleem ontstaat doordat SQL per se een tabel bij FROM gedeclareerd wil zien, en niet gelokaliseerd bij de randvoorwaar-de. En dat is weer een gevolg van de nogal grote taalconstructie van de SELECT/FROM/WHERE query. En dat was weer ingegeven door de wens een natuurlijke taal te benaderen.

Kort en goed, SQL is ontworpen als taal voor mensen, en is daarin schromelijk mislukt. Weliswaar hebben bouwers van databases daar de nodige mouwen aan gepast en optimalisaties voor bedacht, maar het blijft een rare constructie, die ontwikkelaars onnodige hoofdbrekens kost. Achteraf bezien zou SQL nooit als natuurlijk taal ontworpen hebben mogen worden. Maar achteraf heb je natuurlijk makkelijk praten.

Rick van Rein
Dr. Ir. H. van Rein (rick@openfortress.nl) is ontwikkelaar en beheerder bij OpenFortress Digital signatures.