Semantiek is stevig gefundeerd in verzamelingsleer

Silly SQL (6): Alles afzonderlijk verpakt

Rick van Rein

Bij de recente verhoging van het Duitse BTW-tarief zullen heel wat databases een update hebben gedraaid waarbij alle prijzen met een factor 119/116 zijn vermenigvuldigd. Dat soort massale query's zijn het ontwerpdoel van SQL geweest. Daarnaast heeft het model de verfijndheid om alleen tarieven voor Duitsers aan te passen, en eventuele tarieven voor buitenlandse afnemers ongemoeid te laten. SQL is hier precies het juiste gereedschap, als we tenminste de andere afleveringen van Silly SQL even buiten beschouwing laten.

Moderne databases zijn echter een stuk 'hipper' dan voorzien toen SQL ontworpen werd. Er zijn web-interfaces, en daarmee kan soepel worden rondgeklikt. De LAT-relaties tussen de records in een datamodel maken het goed mogelijk en heel aantrekkelijk om gerelateerde records op bevel op te roepen. Echter, heel veel interacties met zo'n database zijn niet langer karakteristiek SQL, omdat ze eigenlijk op losse records werken, in plaats van op verzamelingen records. Alsof je in plaats van een tros druiven een heleboel losse druiven koopt, allemaal afzonderlijk in plastic verpakt.

Fundament van verzamelingsleer

De semantiek van SQL is stevig gefundeerd in de verzamelingsleer, ofwel set theory. De selecties van data zijn in gewone 'set comprehensions' te beschrijven: de verzameling van alle p van type Persoon waarvoor geldt dat p.naam met 'Rick' begint. De and en or operatoren vertalen naar doorsnede en vereniging van verzamelingen. Voor joins en dergelijke zijn formele definities gemaakt die wederom vertalen naar de verzamelingsleer. Alles in SQL is een verzameling.

Het nut hiervan is dat er optimalisaties mogelijk zijn op basis van de wiskundige kennis over deze goedonderzochte presentatiewijze van informatie. Er zijn vele wetten die toegepast kunnen worden op het niveau waarop met verzamelingen wordt gestoeid, en de vrij starre formaten waarin SQL de oorspronkelijke set operaties voorstelt helpen om daarin patronen te herkennen. De engine van een RDBMS is daardoor in staat om te voorkomen dat al te vaak door lange ritsen van records gezocht moet worden door ze allemaal af te lopen. Praktische zaken als keys en indices helpen daarbij natuurlijk enorm.

> SQL is een standaard, maar wel een oude. In een serie 'Silly SQL' artikelen zijn zaken besproken die, in retrospect bezien, een stuk handiger hadden gekund. Dit is het laatste deel van de serie.

> De alombekende volgorde die getallen hebben is niet nuttig voor een key

Maar kijk nu naar de tendens om online in mini-stapjes door data te browsen; dus in plaats van met enige moeite een complexe query opstellen die precies doet wat we willen, gaan we handmatig (en dus tijdrovend) door databases navigeren op zoek naar de data die we denken nodig te hebben. Daarbij worden regelmatig pagina's gepresenteerd met slechts één record, of anders een samenstelsel van slechts enkele. Ook als we lijsten van records bekijken dan worden deze omwille van de browservriendelijkheid afgeleverd in hapbare brokjes.

Simplistische overhead

Elke keer dat zo'n query wordt overgedragen naar de engine begint het hele optimalisatieproces opnieuw. Maar in tegenstelling tot de query die alle BTW-tarieven voor de hele database aanpast, gaat het hier om een relatief klein stukje werk. Al te veel moeite in de optimalisatie steken is dus niet bijster slim, want het zal vaak meer werk zijn dan het bespaart. Maar als engine moet je dat maar net kunnen inschatten. Zonder de data erbij gehaald te hebben is dat niet altijd eenvoudig. Er wordt dus onnodig tijd gestoken in overhead.

En dan hebben we het nog niet gehad over het andere schoolvoorbeeld van investeren in complexe query's, te weten transacties en concurrency control. Het hele systeem van locks, rollbacks en two-phase commit is bedoeld om complexe query's te beschermen. Zodat elke bestelling ofwel tegen het oude, ofwel tegen het nieuwe BTW-tarief wordt afgehandeld. Maar om de overhead van transacties toe te passen op de query voor de 'show my profile' webpagina, dat gaat wat ver. Niet voor niets is MySQL een

populaire engine onder websites: deze deed oorspronkelijk gewoon niet aan transacties en was in verhouding dus supersnel. Maar MySQL of een andere SQL-database, dat mag niet baten; ze hebben allemaal te maken met de overhead van het op verzamelingen gebaseerde model. Ze maken vrijwel allemaal gebruik van onhandige datatypes die alleen in het grote geheel van verzamelingsleer en relationeel databeheer nuttig zijn.

Zoeken naar een record

Wat zijn de eigenschappen van een identifier, of key, van een record? In SQL kan elke combinatie van attributen worden gebruikt als key, waarna de waarde als foreign key mag worden gebruikt in allerlei andere records. Maar om te voorkomen dat dezelfde data op meerdere plaatsen in een datamodel voorkomen, is het veel gebruikelijker om voor keys afzonderlijke velden te gebruiken met data die geen betekenis hebben voor de toepassing die draait, en die doorgaans ook niet getoond worden in query-output.
Dus nogmaals, welke eigenschappen moet zo'n key hebben?
We gebruiken meestal getallen, maar is dat niet wat veel van het goede? Getallen lopen lekker handig op, maar is die volgorde van belang bij de toepassing als foreign key? Wordt er bijvoorbeeld ooit een rapport op zo'n key gesorteerd? Welnee.
De noodzakelijke eigenschap van een key is heel eenvoudig: er moet een vergelijkingsoperatie zijn die keihard 'ja' of 'nee' geeft als je wilt weten of twee keys gelijk zijn. De alombekende volgorde die getallen hebben, is niet nuttig voor een key; in een index kan een willekeurig aspect worden aangegrepen om een 'groter dan' relatie te implementeren, die hoeft niets met verwachtingen van programmeurs of wiskundigen te maken te hebben. Om een bizar voorbeeld te geven: in mijn proefschrift gebruikte ik (in een casus over het oude Egypte) hiëroglyfen als key. Hoewel dat geen praktische implementatie kent, is het wel degelijk valide als key, omdat een eenvoudige gelijkheidsoperator te definiëren is op de spelwijze van de recordnaam.

Het gebruik van getallen als key zit echter even diep in ons denken geworteld als een paardenbloem in ons gazon. Een nieuwe key is gewoon de maximale key plus één, lekker simpel. Tenminste, zo lijkt het. Een maximum kan inhouden dat de hele tabel moet worden doorzocht; als de database het maximum in een cache opslaat dan moet het doorzoeken van de tabel toch nog plaats vinden bij het verwijderen van het (of een) record met de maximale waarde. Om het doorzoeken op te lossen is een key soms aan te maken als 'autoincrement', maar dat is alleen maar het instandhouden van een principieel onlogische werkwijze.

Indices als doekje voor het bloeden

Zoals gesteld gebruiken we SQL steeds vaker om losse records op te zoeken. Daarvoor wordt een key gebruikt van zo'n record. En dat is helemaal niet handig. In een onnozele implementatie zouden we de hele tabel moeten doorzoeken naar het record; in een tabel van lengte N zou dat tijd kosten met complexiteitsorde O(N). Beter gaat het wanneer een index wordt gebruikt; door binair zoeken kan dan in O(log N) worden gezocht in diezelfde tabel. Dat betekent dat elke verdubbeling van de tabelgrootte een vaste extra tijd met zich mee brengt, zodat groei niet zo hard afgestraft wordt.
Het probleem hiermee is natuurlijk dat O(log N) niet zo goed is als het optimum, O(1) ofwel een vaste tijd die niet afhangt van de tabelgrootte. Dat zou overeenkomen met de toegangstijd voor een pointer in een programmeertaal die wordt gevolgd naar een volgend object. Als een index bestaat uit een hashtabel is dat ook wel te halen in een database, maar het blijft onlogisch dat daarvoor een index nodig is; die vergt namelijk onderhoud bij wijzigingen, en vertraagt dus de werking op een manier die in een programmeertaal niet voorkomt.
Het idee van een programmeertaal, waarin een pointer doorgaans gewoon een geheugenadres bevat, kan eenvoudig worden overgenomen in een database. Een record wordt dan niet beschreven op basis van data in de attributen, ook niet door een getalletje dat los van de inhoudelijke data geprikt is, maar door de positie van dat record op de schijf. Een bloknummer en daarbinnen een offset, bijvoorbeeld. Dit lost ook meteen het probleem op van het prikken van een nieuwe, unieke key-waarde: op het moment dat een nieuw record wordt ingeruimd kan de ruimte voor de nieuwe data meteen worden teruggekeerd. Jawel, wederom net als in een programmeertaal.

JDBC? Wat een onzin!

Het bovenstaande argumenteert eigenlijk voor objectgeoriënteerde databases achter web-interfaces. En daarbij is het natuurlijk belangrijk dat de ODBMS ook werkelijk objectgeoriënteerd is in haar opslagstructuren, in plaats van alles naar een RDBMS-opslagstructuur te vertalen.

> Een werkelijke aanpak van het probleem van objectopslag doe je niet orthogonaal

In die zin is de aanpak van Java ten hemel schreiend: het inpakken van SQL-query's in Java-calls, met als doel objecten aan databases te koppelen is vaak een manier om losse objecten aan te spreken, precies datgene waarvoor SQL niet bedoeld is. JDBC is echter een prima manier om Java geaccepteerd te krijgen in omgevingen waarin moet worden geprogrammeerd bovenop een bestaande RDBMS; het is echter niet de beste manier om een database binnen de Java-wereld beschikbaar te maken.
ODBC is natuurlijk een soortgelijk losbollig idee. Een werkelijke aanpak van het probleem van objectopslag onder een applicatie doe je niet orthogonaal, door SQL te importeren; het is beter een taal daadwerkelijk uit te breiden met database-faciliteiten.
Maar of dit ooit gebeurt op een wijze die uitwisselbaar is tussen (vaak concurrerende) programmeertalen is een grote vraag.

En daarom zullen we het nog wel even met SQL moeten stellen. Al deze interfaces zouden een stuk bruikbaarder worden wanneer een RDBMS aan SQL-gebruikers een *handle* voor de opslaglocatie van een record ter beschikking zou stellen, zoals hierboven uitgelegd. Dan zouden indices kunnen vervallen voor het volgen van referenties tussen objecten, en zou het database-gebruik veel sneller kunnen worden. Oracle kan het een beetje, maar standaard SQL is er niet toe in staat.

Conclusie

SQL is bedoeld voor operaties op grote gegevensverzamelingen. De hedendaagse praktijk van inbedding in web-interfaces mag gebruikersvriendelijk zijn, maar het sluit totaal niet aan bij het model van SQL en de werkwijze van DBMS'en. Het zou beter zijn om voor dergelijke toepassingen gebruik te maken van object-georiënteerde databases, en dan bedoelen we niet een wrapper zoals JDBC of ODBC maar een echte, directe ondersteuning van objecten en vooral referenties naar objecten. Dit is in SQL normaliter niet mogelijk.
Achteraf bezien had SQL nooit voor web-interfaces gebruikt 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.