Serialiseerbaarheid voor eindgebruiker nauwelijks een nuttig concept

Silly SQL (2): Te braaf in ganzenpas

Rick van Rein

Een DBMS beheert een database onder strakke randvoorwaarden. Een belangrijke voorwaarde is die van serialiseerbaarheid, ofwel isolatie. Maar die is eigenlijk best vreemd.
Met serialiseerbaarheid wordt bedoeld dat het net lijkt alsof alle operaties op een database in een bepaalde volgorde worden uitgevoerd. Ofwel, bij een set geslaagde transacties hoort een volgorde waarin die hetzelfde effect sorteren als de database aangeeft. Dat is zelfs zo wanneer het DBMS in werkelijkheid een aantal transacties tegelijk heeft uitgevoerd.

Ter implementatie van dit concept is het mechanisme van concurrency control uitgevonden: het zodanig met elkaar vervlechten van transacties dat ze voldoen aan deze blijkbaar heel belangrijke eigenschap. Databasefabrikanten spenderen veel energie aan het implementeren van dit mechanisme, want daarmee is de mate van parallellisme te vergroten, en dus ook de hoeveelheid te verwerken transacties per seconde.

Praktische bezwaren

In de praktijk is concurrency control helemaal niet zo'n geweldige booster voor de efficiëntie van een database. Er is namelijk nogal veel extra administratie voor nodig. De stoere insteek die men in het begin met MySQL heeft genomen deed hier expliciet afstand van, door gewoon operaties meteen te verwerken zonder concurrency control. Dat gaf de database zijn goede reputatie van een razendsnelle database.
Het grootste nadeel dat aan de administratie ten behoeve van concurrency control kleeft, is wellicht de centrale aard ervan, want die maakt het potentieel hoge parallellisme van een database in de praktijk moeilijk op te schalen. Operaties voer je in SQL in principe uit op hele tabellen tegelijk, en dus moet je ook die hele tabellen ergens centraal gaan beheren. Dat kan inhouden dat je een lock zet op een hele tabel, maar dan zet je alle transacties die iets met die tabel willen doen stop. Het kan ook inhouden dat je per record een lock zet, maar dan krijg je bij een bulk update heel veel locks, met als gevolg veel tests voor andere transacties, en problemen met ghost updates. Tenslotte is het ook nog mogelijk om een soort generieke lock te maken dat met een expressie beschrijft welke records gewijzigd zijn, maar ook dat veel tijd vergt van andere transacties, die voor een operatie moeten testen of ze daarmee aan zo'n lock zouden morrelen.
De overhead van een aanvaring tussen locks kan ook groot zijn, zelfs als er geen transacties voor te hoeven worden teruggedraaid. Neem een gebruiker die on-line gegevens invoert, maar die halverwege een transactie even koffie gaat halen. Als een andere gebruiker tegelijk de database aanspreekt ten behoeve van een bellende klant, dan kan het zo maar gebeuren dat een normaal razendsnelle opvraag van informatie wordt tegengehouden totdat de koffiedrinker terug is. Of, ook niet fraai; het handmatige invoerwerk van de koffiegenieter wordt tenietgedaan.

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. Deel 1 is gepubliceerd in DB/M 1, 2006.

Het hoeft niet in ganzenpas

Het idee om alles altijd een volgorde te willen geven is best aanvalbaar. Immers, het is een transactie toch niet bekend op welke plek deze terechtkomt. En het is ook niet zo dat iets af te dwingen is over records die in een draaiende transactie *niet*, maar op een later moment *wel* aan een selectiecriterium voldoen.
Het lijkt dus wel alsof serialiseerbaarheid een bepaalde garantie geeft, maar in de praktijk kun je er niet zo veel mee.

Als je ergens zeker van wilt zijn, dan moet je er een trigger voor definiëren. Dan weet je voor nu en voor altijd dat een bepaalde aanpassing, een bepaalde wens wordt geïmplementeerd. Dit in tegenstelling tot gewone query's die worden uitgevoerd op een soort momentopname van de database, zo goed en zo kwaad als dat de kennis samenvat. Triggers maken het mogelijk om invariante eigenschappen in een database af te dwingen; dus het zodanig aanpassen van het standaardgedrag van de database dat altijd aan bepaalde randvoorwaarden wordt voldaan. Een goed systeemontwerp zit vol van zulke invarianten, al is het maar in de vorm van eenvoudige integrity constraints.
Door op deze manier naar een database te kijken wordt het belang van de timing en volgorde van een transactie sterk gerelateerd, geheel conform de gebrekkige controle over de volgorde waarin transacties worden uitgevoerd door een DBMS.

Partieel geordende tijd

Er bestaan andere manieren om tijd te modelleren dan als een louter sequentiële tijdlijn. Die zijn bijvoorbeeld te vinden in literatuur rond procesalgebra, een tak van wiskunde die zich bezighoudt met beschrijvingen van processen, inclusief hun volgorde, concurrency en onderlinge communicatie of synchronisatie. Diverse procesalgebra's beschouwen tijd niet als iets dat lineair voortschrijdt, maar als iets met vertakkingen en samenkomstmomenten. Het idee van concurrency is in zulke theorieën vaak dat processen over een afzonderlijke vertakking van de tijd voortgaan, en dat ze alleen voor synchronisatie even bijeen komen. Of wanneer er communicatie nodig is, kan het zijn dat alleen de randvoorwaarde wordt gesteld dat het verzendende processtapje op de ene tijdlijn voor het ontvangende processtapje op de andere tijdlijn wordt gesteld.

> Er bestaan andere manieren om tijd te modelleren dan als louter sequentieel

Het algemene concept hierachter is een partiële ordening: een verzameling acties waarvan soms wel, en soms niet bekend is wat aan wat voorafgaat. Als ik in Nederland nies en iemand in Engeland krabt op zijn hoofd, dan doet het er niet zo toe in welke volgorde dat gebeurt, het zijn ongerelateerde gebeurtenissen. Halen we er relativiteitstheorie bij dan blijkt het zelfs van de waarnemer af te kunnen hangen wat als eerste, en wat als tweede gebeurtenis geldt. Eenduidige volgorde-synchronisatie is er alleen als mijn nies telefonisch hoorbaar is voor degene met jeuk. Normaal gesproken zijn gebeurtenissen ongeordend, pas in gevallen waarin sprake is van expliciete communicatie of synchronisatie worden ze geordend.

Serialiseerbaarheid

Zo zou ook een database opgezet kunnen zijn. Weliswaar werkt concurrency control intern zo, maar dat zit zo strak vast aan de wens om alles zo snel mogelijk weer op één (tijd)lijn te krijgen dat het eigenlijk alleen tijdelijk wordt geduld als de ordening partieel is. Dat moet ook wel, want transacties kunnen worden gestopt als er problemen optreden met serialiseerbaarheid.
Serialiseerbaarheid drukt namelijk juist uit dat de ordening compleet of volledig is: voor elk paar handelingen op de database is bekend welke voor welke kwam. Op het moment dat een transactie iets probeert te doen dat de samenvoegbaarheid onmogelijk maakt moet er iets worden gestopt, puur om de serialiseerbaarheid in stand te houden.
Zoals gesteld is serialiseerbaarheid voor de eindgebruiker nauwelijks een nuttig concept, omdat een gebruiker toch vooral de eigen invalshoek ziet, en geordend wil zien.

Serialiseerbaarheid helpt om invarianten te verkrijgen, maar zoals gezien zijn die toch al niet heel hard te maken zonder triggers. Toen we in DB/M nummer 5 van vorig jaar ingingen op ritsbare databases, hadden we het eigenlijk ook over een partieel te ordenen set van updates. Er is toen beschreven hoeveel rek er zit in het splitsen en later herintegreren van databases. Dat bleek nog best te doen te zijn, met de nodige handigheden en niet-standaard ontwerpoverwegingen. In wezen is dit ook wat nodig wordt als een database partieel geordende gebruikers-tijdlijnen ondersteunt.

Leven met variërende data

Het gebruikersconcept van een database die geen serialiseerbaarheid ondersteunt, maar die wel via triggers of regelmatig gedraaide update-scripts de invarianten van het database-ontwerp in stand houdt, lijkt meestal goed te doen. Alle data die we uit een database halen zien we immers als momentopname. Of het nu op een webpagina is, in een database tool of op een SQL-schermpje. Zodra gegevens aan een klant worden opgelezen via de telefoon, veranderen die gegevens in een externe kopie van data, die intern gewoon 'doorleven'. En als de klant hoort dat de ingezonden invoer nog niet verwerkt is, dan kan die klant zich voorstellen dat het een ander proces is dat nog niet gesynchroniseerd is met hetgeen hij hoort, of dat nu te maken heeft met bedrijfsprocessen of met de werking van de database.
Het is alleen in de gevallen waarin we de uitkomsten van een query in een volgende query binnen dezelfde transactie willen gebruiken, dat deze voorzichtigheid niet geboden is – en voor die gevallen was ook wel iets anders te bedenken geweest, bijvoorbeeld door de voorgaande query op te nemen in een volgende (als geneste query dus) in plaats van de uitkomsten daarvan expliciet op te nemen.

Het leven met variabele data is niet zo'n vreemd gegeven bij databasegebruik. Het leven zonder invarianten echter wel. Alleen serialiseerbaarheid helpt daar niet zo veel aan, dat biedt hooguit zekerheid dat de resultaten van een afgeronde query een wijziging over alle records aanbrachten, op het moment waarop die query gedraaid werd – ooit. Maar dat soort zekerheid over een niet nader gedefinieerd moment in het verleden, is zekerheid waar niemand in de praktijk wat aan heeft.
Achteraf bezien zou SQL nooit ontworpen mogen zijn met serialiseerbaarheid als uitgangspunt. 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.