Netwerkprotocol geeft externe partijen rechtstreeks toegang tot data

LDAP: niet ACID, wel een complete database

Rick van Rein

Sommigen halen hun neus nogal eens op voor LDAP. Daarbij gebruikt men als belangrijkste argument dat het geen volledige ACID-transacties ondersteunt. Dat LDAP daarnaast van alles wel biedt, en soms handiger is, wordt daarbij over het hoofd gezien.

LDAP is de afkorting van het Lightweight Directory Access Protocol – een protocol dus, maar wel een dat een datamodel definieert voor de over te dragen gegevens, en dat dus ook sterke affiniteit heeft met de database. Hoewel er mensen zijn die beweren dat een vergelijking tussen LDAP en een relationele database is als het vergelijken van appels met peren, is er toch een niveau waarop dat zinnig is – uiteindelijk biedt alle fruit een manier om vitamines binnen te krijgen immers. Wanneer nog besloten moet worden welke opslag- en distributietechniek wordt ingezet is het wel degelijk interessant om LDAP mee te nemen in de overwegingen. Al was het maar omdat het zo'n vreemde eend in de bijt is, en dus verfrissende nieuwe invalshoeken aandraagt.

Een andere invalshoek

Ook wanneer men LDAP vergelijkt met ODBC of met SQL over SSH zijn er nog flinke verschillen. Dat zit met name in het datamodel – LDAP volgt een hiërarchisch model, terwijl SQL natuurlijk relationeel is. De naam voor een LDAP repository is dan ook Directory, en niet Database. De navigatie in zo'n hiërarchische database is ook anders dan in SQL. In plaats van een willekeurige tabel (ofwel een bepaald datatype) aan te grijpen en er alleen die records uit te ziften waar het om te doen is, wandel je vanaf de wortel van de boomstructuur naar een deel van de directory waar vandaan verder gewerkt kan worden. Desgewenst is het mogelijk onder een aangewezen knoop te zoeken. Dat gebeurt dan op basis van velden (of 'attributen' in LDAP-taal), ongeveer zoals dat ook gebruikelijk is in SQL. Het is zo mogelijk om een lijst van records (of 'objecten' in LDAP-taal) te ontvangen. Sterker nog, het is mogelijk om per query aan te geven welke view, dus welke attribuutselectie, doorgegeven moet worden. Daarnaast is het dankzij het hiërarchische karakter van LDAP mogelijk om de verschillende nodes van de boomstructuur onder te brengen op verschillende servers die desgewenst zelfs onder afzonderlijk beheer kunnen vallen. Eén mogelijkheid daartoe is op basis van DNS-domeinen de verschillende stukken LDAP-infrastructuur te verdelen, een andere is dat een LDAP-server van een bepaalde subboom aangeeft dat die elders ondergebracht is; bij een query die zo'n subboom induikt wordt dan een doorverwijzing doorgegeven die door de client dient te worden opgevolgd.

Hoe 'crisp' moet een database zijn?

Een belangrijk verschil tussen het ontwerp achter LDAP en dat achter SQL is dat LDAP doorgaans afzonderlijke objecten behandelt, terwijl SQL optimaal is voor bulk query's. De snelheidsmetingen op SQL-databases zijn doorgaans gebaseerd op massaal doorgevoerde query's en niet op het aanpassen of opvragen van een enkel object. Toch is dat laatste steeds vaker te zien. Regelmatig wordt SQL ingezet onder een website, en voor dat type toepassingen worden veel vaker losse records aangesproken dan hele ritsen van records. Dat komt enerzijds doordat HTTP niet ontworpen is om over een langere tijd informatie op te bouwen, maar juist afzonderlijke fragmenten losstaande data af te leveren; weliswaar is daar omheen te werken, maar dat kost moeite en wordt dus niet zo snel gedaan als het opleveren van onafhankelijke fragmenten.
Anderzijds worden databases achter websites gebruikt om de inhoud ervan te personaliseren – ofwel, precies het record dat de zaakjes van de bezoeker beschrijft wordt getoond, en al het andere niet. Idem geldt dit voor editen, dat doet de bezoeker ook alleen met zijn eigen gegevens.
Kort en goed, er zijn toepassingen van databases waarin het belangrijk is dat objecten snel afzonderlijk kunnen worden aangesproken, en waarin tijdens een sessie niet te veel hoeft te gebeuren. Maar de database moet wel bestand zijn tegen een grote hoeveelheid transacties. Voor deze toepassingen is de wenselijke schaalbaarheid heel anders dan voor een klassiek database-ontwerp: in plaats van wat minder transacties die heel veel werk verzetten zijn er heel veel transacties die wat minder per stuk doen. Ofwel, het principe van OLTP.

ACID-transacties zijn niet bijster 'crisp'

Het openen van een verbinding naar een database kan best veel werk betekenen. Er kunnen megabytes aan cache en state voor nodig zijn, en dat zorgt ervoor dat een verbinding niet bepaald meer een lichtgewicht resource is. Dit zit dus danig in de weg voor OLTP. Er zijn natuurlijk mogelijkheden om te werken met een 'connection pool' die vanuit bijvoorbeeld een webserver aan-

spreekbaar is; hiermee voorkomt men dat telkens verbindingen moeten worden geopend en gesloten voor een klein beetje werk. Echter dit lost de problemen maar gedeeltelijk op.
De reden dat er veel opslagruimte nodig is, is de noodzaak van administratie per verbinding. Die administratie gaat over zaken als geclaimde en verkregen locks op delen van de database, de resources die meedingen in een transactie enzovoort. Dat gaat dus om de Isolation-aspecten. Ergens intern moet de server bovendien informatie in de transactie-log opslaan voor het administreren van transacties die onderweg zijn naar gegarandeerde Durability in de database zelf.

Het datamodel bovenop XML is nogal specifiek is per toeleverancier

De Isolation property zorgt er verder voor dat transacties soms moeten worden afgebroken, in een wachtrij gezet, en later opnieuw geprobeerd. Dat is niet prettig voor OLTP, waar 'instant gratification' tot het verwachtingspatroon behoort. Als een transactie voor een webrequest in de wacht wordt gezet vanwege een langdurige handmatig doorgevoerde query, dan is dat zeer merkbaar voor die snelle gast die even iets wil opzoeken: het zal er op lijken dat de website 'down' is.
De situatie waarvoor dit in de wacht zetten soms nodig is, is als de database enigzins voorbarig een serialisatie had gepland; doorgaans doordat op het moment van dat besluit niet alle informatie voorhanden was. Als transactie 1 een schrijflock op record A neemt en daarna transactie 2 een schrijf-lock op B en een leeslock A, dan concludeert het concurrency control-mechanisme in de database bijvoorbeeld dat dezelfde A en B mogen worden gebruikt, op voorwaarde dat transactie 2 vóór transactie 1 in de database-historie terechtkomt. Transacties 1 en 2 kunnen dan wel tegelijkertijd worden uitgevoerd. Maar als transactie 1 later ook een lock op record B probeert te nemen, ontstaat een cyclische tegenspraak die alleen te doorbreken is door één van de transacties af te breken en later opnieuw te starten.
Merk op dat dit soort concurrency control het nodig maakt om door lijsten met uitstaande locks te waden bij elke lock-aanvraag. Dat is overhead, zeker voor de OLTP-situatie waarin veel requests kunnen uitstaan. Als die requests bovendien toch allemaal voor eigen objecten zijn zonder al te veel risico van overlap, dan is veel van dat werk voor niets. Concurrency control vertraagt de OLTP-verwerking dus zonder dat het veel oplevert.
Het beheren van locks voor serialisatie, ofwel de Isolation property, is van slechte invloed op het OLTP-gedrag van een database. Het vergroot de hoeveelheid benodigde administratie, het levert mede daardoor flink wat overhead op in de verwerking en het maakt de responstijd voor afzonderlijke transacties soms erg traag door het afbreken en later opnieuw proberen. Merk op dat een database mogelijk verder gaat dan het opsommen van gelockte records. Het is ter voorkoming van ghost records (die bijvoorbeeld worden aangemaakt terwijl een andere transactie dat soort records aan het updaten of samenvatten was) ideaal om de query's met elkaar te vergelijken, zodat gezien kan worden of bijvoorbeeld een toe te voegen of aan te passen record niet onder de selectiecriteria van een andere query zou vallen. Dat is de meest complete, maar beslist ook de minst efficiënte implementatie van de Isolation-property.
Er bestaan databases, zoals de niet te recente versies van MySQL, die geen transacties bieden en daardoor uitermate geschikt (en populair) zijn voor OLTP-type toepassingen. De beperkingen vallen, zoals uitgelegd, voor OLTP-werk vaak nogal mee. Bovendien blijven de mogelijkheden om overzichtsrapportages te maken behouden. Wanneer rapportages er minder toe doen biedt LDAP een nog eenvoudiger, en dus efficiënter, database-model.

Het transactiemodel van LDAP

De manier waarop clients onder LDAP uit elkaars 'vaarwater' worden gehouden is uitermate simpel: elke update geschiedt atomair per object. Ofwel, er kan een update worden gezonden die een aantal attributen tegelijkertijd verandert, maar dan wel binnen hetzelfde object. De tussentoestand van de aanpassingen van de attributen wordt nooit aan clients getoond.
Daarnaast biedt LDAP een aantal randvoorwaarden aan de structuur van haar objecten, en zelfs aan de manier waarop die onder elkaar gehangen worden. Binnen een object is bijvoorbeeld aan te geven welke attributen mogen en welke moeten voorkomen. Een object kan meerdere types hebben, waardoor meerdere van die attribuutlijsten ontstaan, en die schuiven op triviale wijze in elkaar. Ook is het mogelijk om per attribuut aan te geven welke syntax het volgt, en of er meerdere van in hetzelfde object mogen voorkomen.

ACID-eigenschappen van transacties

De ACID-eigenschappen worden gebruikt om te definiëren aan welke eigenschappen transacties voor SQL-databases moeten voldoen. De afkorting ACID staat voor:
Atomicity. Een transactie vindt in zijn geheel wel of in zijn geheel niet plaats (en falen of welslagen wordt terug gerapporteerd);
Consistency. Er zijn bepaalde regels die de structurele juistheid van de data vastleggen en die gelden als randvoorwaarde aan een geslaagde transactie;
Isolation. Ook serialiseerbaarheid genoemd – de transacties zijn in een tijdslijn te ordenen, dus ze schijnen strikt achter elkaar uitgevoerd te zijn;
Durability. Wat is opgeslagen is permanent tot het op durable wijze veranderd wordt.

Dit alles is vrij eenvoudig te implementeren. Het volstaat om een lock per object in de LDAP-hiërarchie te gebruiken. Voor een update wordt dat lock aangevraagd, de voorgestelde wijzigingen worden doorgevoerd en daarna gecontroleerd aan de hand van de structurele randvoorwaarden. Mits akkoord wordt het resultaat van de update persistent gemaakt en kan de lock worden vrijgegeven en de client krijgt een succesmelding.

Merk op dat er per object een lock is, en dat er geen noodzaak is om per client een administratie van locks of uitstaande query's bij te houden. Dit betekent dat een update slechts één test hoeft uit te voeren, of hoeft te wachten op het vrijkomen van één lock alvorens het door kan stoten met het werk. Dit gaat uitermate snel, zelfs als er gewacht moet worden omdat een lock op een object gegarandeerd na korte tijd weer vrijkomt, omdat er geen gebruikersinteractie binnen een transactie valt.
LDAP implementeert dus een vorm van structurele Consistency, slechts een minimale vorm van Isolation en Atomicity, en daarnaast uiteraard Durability. Hoewel minder uitgerust dan een volledige SQL-implementatie is het een stuk geschikter voor het OLTP-toepassingsgebied.

Leven zonder ACID

Het is wegens genoemde overwegingen denkbaar dat de keus valt op een zwakker transactiemodel dan het volledige ACID van SQL. Maar dat houdt tegelijk in dat de applicatieprogrammeur een aantal extra verantwoordelijkheden krijgt, namelijk precies die zaken die niet langer door het transactiemodel worden geregeld.
Wanneer een multi-object update nodig is in LDAP, kan een probleem ontstaan als halverwege blijkt dat een update niet mogelijk is. Immers, zo'n update bestaat uit een serie atomaire updates aan losse objecten, en LDAP kan bij de derde rapporteren dat die update niet doorgaat – dan moeten de eerste twee updates ook teruggedraaid worden. Afhankelijk van de rest van het systeemontwerp kan dit wel of niet problematisch uitpakken.
Men moet bedenken dat een handmatig uitgevoerde roll-back kan falen, maar dat de kans daarop eigenlijk miniem is. Een roll-back van een eerste update in een multi-object update kan falen, als het object inmiddels door een andere client is aangepast, maar dat komt in de praktijk zelden voor. Schokkend als dat moge zijn voor database-ontwerpers, is het wel degelijk zinnig af te wegen hoe vaak dat zou gaan gebeuren en of dat niet met een speciaal script of handmatig met een generieke LDAP-tool gerepareerd kan worden. Technisch is dat inferieur, maar commercieel gezien soms de meest billijke oplossing!
Het is ook mogelijk de zaak anders aan te pakken, bijvoorbeeld door zelf enige database-state in een object op te nemen: een vlaggetje dat aangeeft welke processtap volgt, bijvoorbeeld, en wie die mag doorvoeren.
Een andere techniek om multi-object updates toch transactioneel te laten lopen, is het in één object concentreren van de informatie die bepaalt of de omslag als geldig mag worden beschouwd. Zodra daar de succesinformatie wordt afgevlagd, verandert de interpretatie van de hele database.
Een oplossing voor grotere updates die wel heel erg in de lijn van LDAP ligt, is het concentreren van alles wat als eenheid moet veranderen in een enkel object. Dit klinkt raar in de oren van een database-ontwerper die juist gewend is data te splitsen en structuur op te slaan in een veelvoud aan records. Maar bedenk dan wel dat die ontwerpmethode blind vaart op het krachtig kunnen combineren van tabellen in SQL, iets wat niet opgaat voor LDAP. Daarnaast is het in SQL niet handig om meervoudige attributen in dezelfde tabel op te slaan (bijvoorbeeld drie orderregels in een orderobject) terwijl dat in LDAP uitstekend gaat. Wanneer de data niet voor elke query anders gecombineerd hoeven te worden maak je er in LDAP gewoon een enkel object van!

> Het is mogelijk om per query aan te geven welke view doorgegeven moet worden

Tenslotte valt bij deze technieken veel voordeel te behalen uit een wat vreemd aspect aan LDAP updates: behalve de nieuwe attribuutwaarden is het namelijk mogelijk om ook de oude, te vervangen attribuutwaarden op te geven. Wanneer deze niet overeenkomen met de aangetroffen waarden in de attributen zal de update niet slagen. Het is hiermee mogelijk een waarde van een attribuut te testen, simpelweg door deze als oude zowel als nieuwe waarde voor dat attribuut op te nemen. Zonder netto-invloed op het object kan dan toch iets gedaan worden dat lijkt op de WHERE-clause in een UPDATE-statement in SQL.
Waar het op neer komt is dat met de afwezigheid van de gecompliceerde concurrency control uit SQL-databases te leven valt; dat er een aantal technieken bestaat om een applicatie te bouwen met soortgelijke garanties; maar dat dat wel wat meer werk vraagt van de applicatiebouwer. Als de genoemde OLTP-randvoorwaarden gelden dan dus is met wat meerwerk zeker een alternatief voorhanden.

Externe toegankelijkheid

In het voorgaande werd LDAP beschreven als een soort afgeslankte, anders vormgegeven variant op SQL databases. Dat is natuurlijk maar half waar – het is een heel ander soort ontwerp. Er zijn dan ook punten waarop LDAP beter scoort dan LDAP. De voornaamste punten waarop dit gebeurt is bij aanspreken van de beheerde gegevens over een netwerk; bijvoorbeeld om klanten buiten het bedrijf toegang te bieden tot de database. Daarbij spelen vragen omtrent de wijze van verbinden, anderzijds zijn er vragen over de authenticatie en de daarbij toegekende rechten.
Het opbouwen van een verbinding is strikt genomen geen zaak voor SQL, omdat dat slechts een database-vraagtaal is. Alle praktische database-oplossingen bieden echter een client-side library die zulke toegang over een netwerk regelt, en er is doorgaans ook

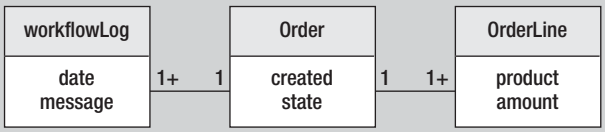
wel een ODBC-driver te vinden. Het leggen van de verbinding is dus niet zo'n groot probleem.
Voor de authenticatie biedt het SQL-model de mogelijkheid om accounts aan te maken voor de afzonderlijke gebruikers, en met *grant* en *revoke* zijn daar rechten aan toe te kennen, zodat bijvoorbeeld een distributeur of reseller wel bij de database kan, maar niet bij allerhande tabellen die intern gebruikt worden.

Een opsplitsing tussen interne en externe gebruikers volstaat maar zelden. Wanneer de externe gebruikers resellers zijn dan is het wel zo netjes om te voorkomen dat ze elkaars data kunnen zien, en helemaal dat ze die kunnen modificeren. Want resellers staan regelmatig op concurrerende voet met elkaar. Daarnaast spelen er wat privacy-regelingen die het op zijn minst ongepast maken als account-specifieke gegevens tussen resellers overdraagbaar worden.
Nodig is dus een toegangsmodel waarin elke externe gebruiker een eigen view op de beheerde gegevens krijgt. Omdat SQL in haar basis werkt met tabellen waarin juist alle informatie geconcentreerd wordt, is dat geen eenvoudig probleem. Denk maar eens na wat voor onzinnige aanpak het zou zijn om per reseller een order-tabel aan te maken. Het enige dat met het SQL-model voor accounts mogelijk zou zijn, is het aanmaken van een view per reseller. Maar dat lost de problemen met het updaten en inserten van nieuwe records nog altijd niet op.
Wat daardoor vaak gebeurt, is dat resellers alleen via een web-interface bij de bedrijfs-database kunnen. Dat klinkt heel aardig, maar het is een ramp voor resellers die willen automatiseren. Niet alleen omdat men zo'n interface dan moet parsen en dat dus elke verandering moet worden doorgevoerd bij elke reseller, en niet alleen omdat zo'n parser specifiek is voor elke toeleverancier van de reseller, maar vooral omdat het de bedieningsmogelijkheden limiteert tot de fantasie en ondersteuningsbereidheid van de toeleverancier.
Een tussenoplossing biedt XML, maar ook hier geldt weer dat het datamodel bovenop XML, dus het XML-schema, nogal specifiek is per toeleverancier. Ook is het model van XPath weer een hele andere manier om een gegevenscollectie te benaderen, en dat resulteert toch altijd weer in specifieke vertaalproblemen.

LDAP is ontworpen als netwerkprotocol waarachter toevallig gegevens zijn opgeslagen, en dat is op dit punt goed te merken. Allereerst zijn de gegevens van de accounts opgenomen in de LDAP-hiërarchie, waardoor ze benaderbaar worden voor normale interacties. Maar bovendien zijn toegangsrechten schaalbaarder te regelen, door slim gebruik te maken van verwijzingen naar de in dezelfde directory opgeslagen accounts. Weliswaar is niet gestandaardiseerd welke view op de data beschikbaar wordt gemaakt aan welke inloggers, maar het is met praktische implementaties zoals OpenLDAP mogelijk om met een vast aantal regels soortgelijke gedragingen te regelen voor afzonderlijke resellers, en wel zodanig dat ze elkaar niet kunnen zien. Dat schaalt dus veel beter dan het voorbeeld van een reseller-specifieke view op een repository. De combinatie van betere schaalbaarheid van accounts en de directe toegang tot de data, maakt LDAP een zeer interessante optie wanneer externe contacten bij een deel van de interne data moeten kunnen.

Verschillende database-modellen

De vertaling van SQL records naar een LDAP database is geen rechtstreekse vertaling. Neem bijvoorbeeld het onderstaande ER-diagram.



In LDAP zou men deze records kunnen combineren in een enkel object, zoals het volgende data-object laat zien:

objectClass: workflowObject
objectClass: order
objectClass: orderLine
createTimeStamp: 20050301191256Z
workflowState: inTransit
orderItem: 12 shaman-license
orderItem: 33 login-token
workflowLogEntry: Thu Mar 01 19:12:56 2005: Order submitted
workflowLogEntry: Thu Mar 03 13:12:12 2005: Order shipped

De objecten in LDAP zijn veel flexibeler van vorm dan een record in een SQL-database, en daardoor is het mogelijk meer data bij elkaar te houden. Dit maakt de data sneller aanspreekbaar, maar ook vermindert het de noodzaak van transacties die meerdere objecten tegelijk aanpassen.

Conclusies

SQL is geoptimaliseerd voor grootschalige operaties op data. Voor toepassingen zoals OLTP is het soms mogelijk en wenselijk om wat minder krachtige faciliteiten te hebben, met name op het gebied van concurrency control. LDAP biedt een netwerkprotocol met zulke eigenschappen, dat bovendien gestandaardiseerd is. Dankzij die standaard, en ook dankzij een betere schaalbaarheid van accounts dan bij SQL, is het mogelijk om externe partijen rechtstreeks toegang tot opgeslagen data te bieden, zonder tussenkomst van een dataformaat zoals XML of HTML.

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