Van centrale database naar 'centrale' ruggengraat van gerepliceerde systemen

Ritsbare Databases

Rick van Rein

Het klassieke beeld van een database is dat van een centraal server-systeem. Als de data al over nodes verdeeld zijn, dan worden die zo goed mogelijk consistent gehouden. Toch is het soms praktisch om databases, of delen daarvan, af te koppelen voor 'veldwerk'. En er is weinig meer voor nodig dan een geschikt ontwerp.

Denk eens aan een verkoper die de productgegevens mee moet nemen op een PDA of een laptop; die is optimaal gediend als hij direct bij de database kan, of bij een meeneembaar uittreksel, zelfs al loopt die soms een beetje achter. Op zich is het niet moeilijk te realiseren, door zo af en toe een kopie van de database uit het centrale systeem te trekken. Problematischer wordt het wanneer de verkoper ook orders aanneemt van klanten, en deze later wil kunnen invoeren in diezelfde centrale database. Dit type problemen vormt het onderwerp van dit artikel.

Zelfmaken: Koppelscripts

Het herintegreren van een eerder losgekoppelde database is in zijn algemeenheid geen probleem dat goed is op te lossen. Er zal zelf aan de oplossing geprogrammeerd moeten worden, en daarbij zal kennis van het database-ontwerp meegenomen moeten worden. Er zullen dus scripts of iets dergelijks nodig zijn voor het afsplitsen van de database en voor het reïntegreren ervan. Dit kan op een aantal manieren en met betrekking tot welke gewenst wordt, is het goed om een keus te maken voor de rest bepaald wordt. De simpelste maar ook meest starre methode, is aan te nemen dat een laptop of ander 'satelliet' systeem een tijdelijke afsplitsing is die later weer aangekoppeld wordt. Daarbij is in elk geval iets van het synchronisatieproces bekend, in tegenstelling tot bijvoorbeeld de aanname dat elk systeem op elk moment, elk gewenst beetje data kan in- of uitchecken.
Er ontstaan de nodige problemen als meerdere verkopers of veldwerkers data willen kunnen integreren in de centrale database. Stel dat elke thuiswerker dat ging doen, dan zou het einde zoek zijn! Tenzij men vooraf de problemen inventariseert en oplost natuurlijk. Een probleem dat optreedt is dat identifiers (met name auto-incremented integer keys) kunnen clashen bij de reïntegratie. En een identifier aanpassen doet men niet slechts op één plek, dus begrip van het hele datamodel moet eraan te pas komen om dat te verhelpen. Men moet dan maar hopen dat de identifier niet als bestelnummer is uitgegeven aan de klant, of op een andere manier al buiten de database beland is.

Verder kan het gebeuren dat twee partijen dezelfde data aanpassen, of erger nog, het kan gebeuren dat de ene partij data verandert die een andere partij weghaalt uit de database.
De vraag moet beantwoord worden wat zoiets inhoudt in termen van volgorde en consistentie, als beide partijen hun wijzigingen integreren in de centrale database.
Tenslotte; de enige manier om in een dergelijk scenario nog transactionele eigenschappen te waarborgen, is om bij de afsplitsing van een kopie alvast de locks te claimen op de data die later aangepast kunnen worden bij herintegratie. Maar dan zit de centrale database opeens met stapels locks die wel heel lang

Publieke PGP key servers

Hoewel het geen relationele database is, lopen de publieke PGP key servers van pgp.net als rode draad door dit verhaal. Ze vormen een uitstekend voorbeeld van de mogelijkheden die ontstaan wanneer de hier besproken ontwerpprincipes worden gebruikt.
PGP is het systeem van digitaal ondertekenen en coderen van bijvoorbeeld e-mail, waarbij gebruikers elkaars sleutels onderling authenticeren. De houder van een PGP-key kan daar een identiteit, of zelfs meerdere, aan hangen en daar vervolgens bekrachtiging van anderen op vragen. Er zijn verder mogelijkheden om sleutels of identiteits-bekrachtigingen terug te trekken. Sleutels kan men op elk gewenst moment laten downloaden door gebruikers her en der, en na bekrachtiging van een identiteit of de terugtrekking van zo'n zelf eerder gemaakte bekrachtiging kan men de gewijzigde sleutel uploaden naar een key server.
De servers worden, behalve als min of meer centrale vindplaats voor PGP keys, ook gebruikt om paden te vinden van de eigen key naar die van een ander die zojuist e-mail stuurde. Afhankelijk van hoeveel niet-overlappende paden er zijn, en hoe lang die paden zijn, schatten PGP-gebruikers in hoe betrouwbaar ze de link naar zo'n andere persoon vinden. Dit heet het Web of Trust van PGP. Het dient voor die gevallen waarbij 'dit is dezelfde persoon als bij het vorige contact' niet volstaat, zoals bijvoorbeeld bij de verificatie van software-downloads.

stand houden; dat gaat niet goed werken. De database-software is er niet op gebouwd om de controle over locks uit handen te geven en de gebruiker zal het niet verdragen als query's dagenlang worden tegengehouden. Er zit niets anders op dan de ACID-regels in een dergelijk toepassingsscenario te versoepelen.

ACID property's op de helling

De ACID property's bieden de zekerheden die bij een database horen. Welke eigenschappen kunnen behouden blijven, en welke niet? Voor de eigenschappen die verloren dreigen te gaan is een work-around plezierig, en daar gaat de rest van dit artikel op in. Maar eerst is het nuttig de ACID property's afzonderlijk te bespreken.

Atomiciteit.
Dit staat voor het volledig wel of volledig niet uitvoeren van een transactie. Op het moment dat een transactie wordt afgesloten moet bekend zijn of die succesvol is geweest.

> Een belangrijke invalshoek bij het ontwerp is die van monotoniciteit

Voor een afgesplitste database is het uiterst wenselijk deze eigenschap te behouden. Dus zonder kennis van de centrale database zou besloten moeten kunnen worden of een transactie in orde is of niet. Anders zou de situatie kunnen optreden dat de verkoper een succesvolle transactie denkt te boeken terwijl de centrale database bij herintegratie van de verkooptransacties besluit dat het om de een of andere techische reden niet mogelijk is. Dat gaat problemen geven.

Consistentie.
Dit houdt in dat een database na een transactie in een volgens het datamodel acceptabele toestand belandt. Niet elke consistentie-eis zal volgehouden kunnen worden na splitsing; bijvoorbeeld een maximum aantal entry's onder een tabel is niet vol te houden als de verschillende afgesplitste databases dat niet met elkaar uitvechten. Toch is consistentie een belangrijke oorzaak van het falen van een transactie, en zoals genoemd onder Atomiciteit dient dit voorkomen te worden.

Isolatie.
Deze eigenschap wordt ook wel serialiseerbaarheid genoemd, en het komt er op neer dat transacties kunnen worden voorgesteld als een in de tijd opeenvolgende serie transacties. Deze eigenschap wordt doorgaans in ere gehouden door locks, door deadlocks te detecteren en desnoods door conflicterende transacties af te gelasten en later opnieuw te proberen.
Dit alles gaat niet zomaar werken in een gesplitste opzet. Locks distribueren over afsplitsingen van de centrale database gaat niet, zoals boven beschreven, en als alle afzonderlijke databases hun eigen locks lokaal kunnen bekrachtigen is het mogelijk dat transacties op verschillende afsplitsingen andere gevolgen voor dezelfde data hebben. Ook is deadlock-detectie niet mogelijk over een ongekoppeld systeem van database-afsplitsingen.

Durabiliteit.
Deze eigenschap garandeert dat data na een commit daadwerkelijk opgeslagen en oproepbaar zijn; persistentie dus eigenlijk. Dit blijft onveranderd gelden na opsplitsing van een database, al is het natuurlijk wel zo dat de opslag gesplitst is en via een synchronisatie tussen een afsplitsing en de centrale database kan worden overgedragen. Dus persistentie is een lokaal begrip geworden, dat zich bij synchronisatie als een olievlek verspreidt over databases.

Al met al klinkt deze opsomming wat zuur. Toch is er met een paar ontwerprichtlijnen vrij goed uit te komen. Want, hoewel het in het initiële database-ontwerp moet worden meegenomen, het is wel degelijk mogelijk om databases af te splitsen en daarna te herintegreren.

Langere identifiers

Om te voorkomen dat twee partijen met een afgesplitste database dezelfde identifiers 'prikken', kan gebruik worden gemaakt van langere, random geprikte identifiers. Deze praktijk is redelijk gebruikelijk; een GUID of UUID is een random getal van 128 bits waarvan het erg veilig is om aan te nemen dat er geen clashes optreden. De kans dat twee van die getallen toevallig dezelfde blijken te zijn is 2^{64}. Deze kans is zo klein dat zelfs cryptografen er tot zeer recentelijk vanuit durfden te gaan dat een clash niet moedwillig te veroorzaken is, laat staan per ongeluk.
Het is wel belangrijk om er voor te zorgen dat de random-waarden goed zijn, en niet bijvoorbeeld een repeterende getallenlijst die op de een of andere manier problemen kan geven. Goede random-getallen helpen tegen het vinden van dezelfde waarden op verschillende systemen, als het desbetreffende schema lokale computerdata incorporeert zoals de CPU-ID of het (globaal unieke) 48-bits nummer van een ethernet-kaart. Hiermee wordt het in principe veilig om de rest door een precisieklok te laten bepalen. Maar beter nog is het om werkelijke random-waarden te gebruiken, zoals afkomstig uit hardware. Er komen langzamerhand processoren en moederborden op de markt die een in hardware geïmplementeerde random-generator bieden; dat is de simpelste en zekerste manier om aan random-informatie te komen, omdat software altijd zo vreselijk deterministisch is.
Het is in principe veilig om aan te nemen dat eenzelfde UUID in twee te synchroniseren databases op hetzelfde object slaan.
De kans dat dezelfde UUID op twee afzonderlijke plaatsen wordt gegenereerd is immers astronomisch klein, vermits de random-getallen netjes worden gegenereerd.
In sommige toepassingen zal de informatie of een UUID nieuw gegenereerd is in een bepaalde database-node expliciet aanwezig zijn. In zo'n geval is het zelfs mogelijk om clashes te detecteren.

Als dat kan is het wel een goed idee. Weliswaar hoeft de kans op een clash niemand normaal gesproken de nachtrust te kosten, maar als het mogelijk is een clash expliciet te detecteren dan is het uitblijven ervan wel extra harde zekerheid. Een gemelde clash kan vervolgens handmatig worden afgehandeld; de kans dat een clash optreedt, vermenigvuldigd met de hoeveelheid handwerk, is zo klein dat er geen automatisch oplossingsmechanisme voor te bouwen is.
Een totaal andere aanpak is om de identifier te splitsen in twee delen, bijvoorbeeld twee integers. De eerste is dan een identifier voor de afgesplitste database-node, de tweede een gewoon nummeringssysteem binnen die node, bijvoorbeeld een auto-incremented integer. Er zijn ook schema's te bedenken waarbij die twee zaken worden geïntegreerd; waarbij elke node gebruik maakt van auto-incremented nummering maar met voldoende ver uit elkaar liggende startpunten.

Het is dus mogelijk om nieuwe records aan te maken zonder risico op een clash van identifiers. Dit is een kwestie van vooruit plannen op splitsing van de database en dan gaat het verder vanzelf. In de publieke PGP key servers (zie kader) werkt het principe van lange random getallen prima. PGP sleutels worden offline gecreëerd, en bevatten aardig wat informatie die zo goed als random is, zoals de public key en een persoonlijk e-mail adres. Door daarover een 'secure hash' te berekenen ontstaat een zogenaamde 'fingerprint', een identifier van 128 of 160 bits. Deze waarde, of de laatste 64 bits ervan, fungeren als identifier van PGP keys in deze publieke key servers.

Monotoon is verre van eentonig

Een belangrijke invalshoek bij het ontwerp van een splitsbare/ritsbare database is die van monotoniciteit. Dit is een wiskundige term die inhoudt dat ontwikkelingen in dezelfde richting plaatsvinden; bijvoorbeeld een getal dat nooit afneemt in waarde naarmate de tijd vordert. Dat biedt extra zekerheden die soms erg nuttig kunnen zijn.
In termen van een database kan monotoniciteit goed worden vertaald naar 'alleen toevoegen'. Ofwel, er mogen INSERT's worden gedaan maar geen DELETE's. Het integreren van twee tabellen uit verschillende database-nodes wordt hiermee betrekkelijk eenvoudig. Voor sommige toepassingen is het heel natuurlijk om niets te DELETE'n, bijvoorbeeld als data niet worden verwijderd uit historisch besef. Maar in gevallen waarin eigenlijk gewiste data in de weg zouden zitten, kan ook worden gewerkt met een extra attribuut dat aangeeft of het desbetreffende record eigenlijk gewist zou moeten zijn. Een SELECT statement kan daar gemakkelijk een extra test voor opnemen en in de applicatie hoeft het verschil dan niet meer duidelijk te zijn.
Dit alles kan leiden tot een ongewenste ophoping van overtollige data. Als dat een probleem wordt, is het wel mogelijk om overbodig geworden data te verwijderen, maar alleen nadat alle afgesplitste databases er van op de hoogte zijn gebracht. Dit wordt dan een soort 'garbage collection' voor de opgesplitste database.

Dit mag resoluut klinken, maar het wordt daadwerkelijk toegepast. De identiteiten die op PGP-sleutels vermeld staan kunnen door alles en iedereen worden aangemaakt of bekrachtigd en verzonden naar de centrale key servers. Deze ondertekende informatie wordt in principe nooit weggegooid. Er is een mogelijkheid om bekrachtigingen of zelfs hele sleutels terug te draaien, maar daarvoor volstaat het weghalen van de gewraakte informatie niet, omdat die door anderen al kan zijn gedownload en die kunnen zouden het dan weer kunnen uploaden. Wat wel werkt is een expliciete verwijderingsopdracht (uiteraard ondertekend) uit te schrijven en te verzenden aan de key servers. Op een PGP key server staan dus niet alleen bekrachtigingen maar ook door de oorspronkelijke bekrachtigers ondertekende ontkrachtigen van bekrachtigingen.
Merk op dat deze PGP databases, hoewel niet relationeel, een prima voorbeeld bieden van gesplitste monotone databases. Er is een ruggengraat van gerepliceerde servers in allerlei landen, en veel PGP-gebruikers halen er sleutels en identiteitsbekrachtigingen vandaan of uploaden zulke informatie wanneer het hen uitkomt, en van of naar een server die prettig dichtbij draait. Dit werkt allemaal vlekkeloos dankzij een monotoon datamodel.

Veranderingen werken anders

Het updaten van de informatie in een gesplitste database is een hachelijke onderneming. Als een bestand in twee verschillende versies bestaat weet men niet altijd welke versie de nieuwste data bevat. En zelfs als men een timestamp toevoegt met het verandermoment, dan nog kunnen problemen ontstaan wanneer twee mensen in verschillende afsplitsingen van de database wijzigingen aanbrengen. De enige echte oplossing, en daarmee simuleert men locks zonder dat die de hele database-performance onderuit halen door hun langdurigheid, is het verdelen van rechten over de verschillende opgesplitste delen. Zo'n opsplitsing kan rudimentair zijn, door rechten tot wijziging van bepaalde tabellen of kolommen toe te kennen aan bepaalde database-afsplitsingen. Meestal zal dat dan de centrale database worden, en dat houdt in dat de afsplitsingen een read-and-insert-only variant van de database zouden krijgen. Dat is zelden afdoende.

De publieke PGP key service draait niet op een enkele server

Al wat verfijnder is dan een model waarin de rechten per record worden toegekend aan de maker ervan. Die maker zal dan moeten worden vermeld in zulke records. Of beter nog, een record zal een eigenaar moeten hebben, die initieel de maker van het record is. Een eigenaar kan namelijk later worden veranderd. In systemen voor orderverwerking (zoals het verkopersscenario) is dat even zinnig als in ticket-systemen voor e-mail helpdesks. Het hoeft dus niet strijdig te zijn met de belangen van het systeem zelf om zulke

attributen op te nemen, en het is ook zeker uit te leggen aan de eindgebruikers die zich best kunnen voorstellen wat de gevolgen zijn van het splitsen van een database.
Eigenaren kunnen overigens ook groepen zijn (denk aan technische support) en in sommige gevallen zal het mogelijk zijn dat een persoon uit zo'n groep een ticket of record naar zich toe trekt. Dat zal dan tijdens een geschikte synchronisatie moeten gebeuren. Het kan bijvoorbeeld handig zijn als het desbetreffende synchronisatie-script taken vervult als 'zorg dat een support member altijd tien tickets open heeft staan'. Het toekennen van een record aan een andere eigenaar van een record zal doorgaans door de eigenaar worden gedaan, of als het een andere macht is dan zal het toch pas hard kunnen worden gemaakt als de database-afsplitsing van de eigenaar wordt gesynchroniseerd met de centrale database. Anders kunnen er weer inconsistenties optreden.
En hiermee komt een volgende verfijning naar boven; per record dient ook een state te worden opgeslagen, in een state diagram of workflow-proces. De eigenaar van een record kan een state transitie doorvoeren en bij synchronisatie met het centrale systeem kan de eigenaar bij de nieuwe state worden ingesteld. Zo kan per state een eigenaar – persoon of groep – worden aangewezen, wat een prima implementatie biedt van een workflow-systeem. Het is in een workflow in ieder geval heel normaal dat de toegang tot data gelimiteerd wordt gehouden.

Redundantie onder handbereik

De publieke PGP key service draait niet op een enkele server; het is een netwerk van gerepliceerde databases, de ruggengraat waarmee lokale installaties van PGP communiceren over keys. De Nederlandse node is bijvoorbeeld te vinden op www.keys.nl.pgp.net:11371/ – maar er zijn er ook diverse in andere landen. Ze updaten elkaar bij wijzigingen, zodat men inzendingen alleen maar naar een handig dichtbijgelegen node hoeft te verzenden.
De stap van een centrale database met afsplitsmogelijkheden naar een 'centrale' ruggengraat van gerepliceerde systemen is een voor de hand liggende volgende stap. Een ontwerp dat geschikt is voor fragmentatie kan ook vaak gebruikt worden voor een redundante opzet met een centrale ruggengraat in plaats van een enkele centrale server. Immers, vertraagde updates zijn al acceptabel en er is een mechanisme om nieuwe informatie en veranderingen van her en der te accepteren; dan is de stap naar een ruggengraat van servers die elkaar updates sturen ook niet ver meer.
Dit is dus nog iets om in een vroeg stadium van het ontwerp te overwegen. Het vereist wellicht wat specifieke server-naar-server scripts en niet alleen client-naar-server scripts om de synchronisaties door te voeren. Maar dat die scripts op elkaar zullen lijken ligt wel voor de hand.

Bezint aleer ge begint

Er bestaan enkele goede technieken die het mogelijk maken om een database in stukken te knippen en later weer samen te voegen. Hoewel dat met moderne communicatieprotocollen zoals GPRS en UMTS niet de enige mogelijkheid is, kan het helpen problemen vanuit deze andere invalshoek te bezien. Het is belangrijk deze overwegingen al vroeg te betrekken bij het ontwerp van een datamodel, omdat sommige ontwerpbeslissingen erdoor bepaald worden. Maar dat daar een aangenaam flexibel systeem uit voortkomt wordt aangetoond door de PGP public key server infrastructuur die in dit verhaal als leidraad diende.

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