Twee complexe systemen met verschillend toepassingsgebied

Filesystemen, de 'broertjes' van Databases

Rick van Rein

Hoewel filesystemen hele andere opslagstructuren bieden dan databases verloopt de ontwikkeling niet echt heel verschillend. In een wereld waar gestructureerde data in een DBMS zitten en ongestructureerde gegevens in een filesysteem, is dat wel prettig ook – want we hebben beide nodig in de praktijk.

Databases en filesystemen hebben een fundamenteel verschillende aard. Terwijl een database floreert bij strak gestructureerde data, is de semantiek van een filesysteem beperkt tot het afhandelen van losse files als binaire blobs. Maar als het aankomt op operationele aspecten, zoals backups en de vereiste beschikbaarheid, dan zijn er juist weer veel parallellen te trekken. Met deze focus bespreken we in dit artikel een paar populaire nieuwigheden uit de wereld van filesystemen.

Synchroon of asynchroon

Bij communicatie tussen hardware en software is buffering steevast een ingrediënt. Daarmee is het mogelijk om de hardware voortdurend aan het werk te houden, omdat weg te schrijven data al kant en klaar staan om te worden verzonden. Bij een harde schijf wordt daar bovendien nog scheduling aan toegevoegd, omdat een zijwaartse beweging van de kop relatief veel tijd kost. Een flinke buffer met weg te schrijven diskblokken is dus nuttig, omdat daarmee optimaal gepland kan worden wat de schijf moet gaan doen.

> Journals voorkomen checks niet en dus vertragen ze een herstart nog steeds

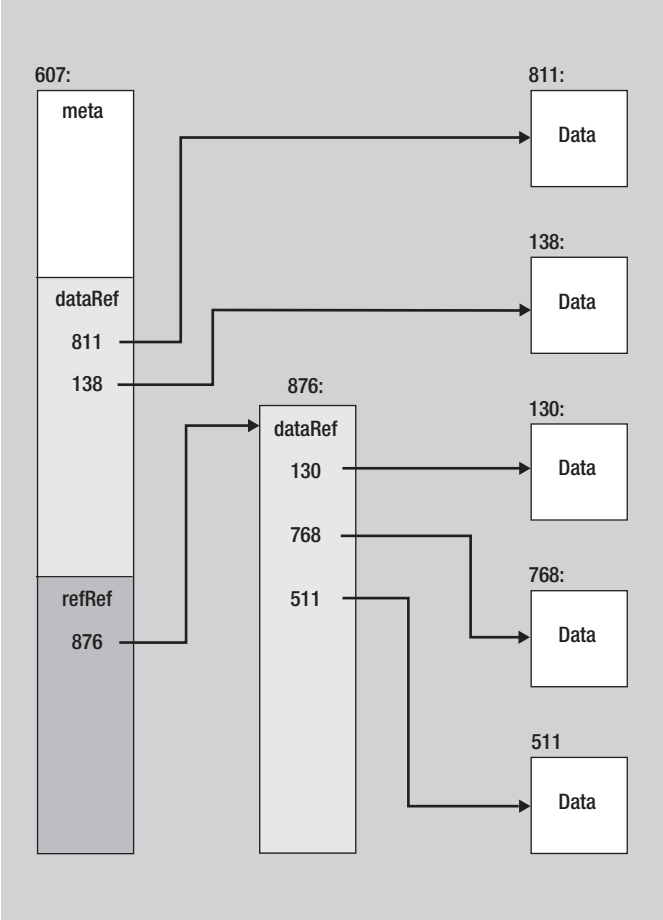
Schedulen kan helaas wel gevolgen hebben die zowel voor filesystemen als voor databases funest zijn, namelijk als de schijf onverwacht uitvalt. Doordat iemand op reset drukte, de spanning uitviel, of zoiets. Dan kan het gebeuren dat de blokken in een verkeerde tijdsvolgorde op de schijf terecht zijn gekomen, met als resultaat een filesysteem met een structuur die niet consistent is. Er is dan een reparatie nodig en die kan op de klassieke manier (de hele schijf aflopen en alles met elkaar vergelijken) erg lang duren, vooral met grote schijven.

Databases die op een filesysteem draaien (doordat hun back-end naar files wegschrijft in plaats van naar een rauwe partitie) moeten hier rekening mee houden – als ze wordt gevraagd een commit te doen, dan moeten ze zeker stellen dat de veranderingen synchroon worden gemaakt, dat wil zeggen dat de I/O naar de harde schijf daadwerkelijk heeft plaatsgevonden. Operating systemen bieden daarvoor kernel-aanroepen zoals sync of fsync – de eerste om alle gebufferde blokken weg te schrijven naar schijf, de tweede om dat te doen voorzover het een specifieke file betreft.

Journaling

Filesystemen beheren files. Een file is een ding met een naam, en bestaande uit een reeks blokken met daarin data die verder meestal niet worden geïnterppreteerd door een filesysteem, zie afbeelding 1. Nu zijn er doorgaans kernel-aanroepen als mmap, die de blokken in een file weergeeft als blokken in het virtuele geheugen, maar dan zonder ze alvast in te laden. Linux doet dit bijvoorbeeld standaard met de programma's die ze draait. Pas wanneer een programmablok wordt aangesproken wordt het dan van schijf gehaald en daadwerkelijk in RAM vastgehouden. Moet een programma uitgeswapt worden, dan volstaat het om de desbetreffende pagina terug te zetten op 'niet ingeladen' en de onderliggende fysieke RAM te hergebruiken. Om dit efficiënt te laten verlopen is het alleen wel nodig dat de datablokken van een file echt hele blokken zijn – de metadata met bijvoorbeeld de naam van de file moet dus elders worden opgeslagen, niet in die blokken zelf. Ook om snel een directory door te kunnen lopen is dat overigens wenselijk.

Als je dus een file begint te schrijven dan moeten er twee dingen gebeuren; de metadata moeten geschreven worden en de file moet gevuld worden met data. De traditionele aanpak hiervoor is dat de naam eerst moet worden vastgelegd, omdat dan de datablokken niet zonder bestemming los komen te hangen. Om dit te voorkomen wordt het aanmaken van filenamen daarom vaak synchroon uitgevoerd, terwijl het eigenlijke dataverkeer via de



Afbeelding 1: Een file op schijf bevat meta-informatie plus een aantal referenties naar blokken die geheel voor data bestemd zijn. Om hele grote files te ondersteunen zijn zelfs diverse niveaus van indirectie mogelijk om de lijst van blokken op te slaan. Los van deze opzet zijn er ook nog directory entry's, die een naam vermelden met een verwijzing naar blokken als dit blok 607. Niet alle types blokken worden op dezelfde manier behandeld qua synchroniciteit.

diskbuffer gaat. En dat vertraagt de zaak best wanneer je bijvoorbeeld een grote tarball of zipfile uitpakt, of een grote directory recursief verwijdert of verplaatst naar een andere disk of partitie.

Journaling filesystemen pakken dit probleem aan met een soort 'transaction log', hoewel dat niet gebeurt in de stricte zin die we van databases gewend zijn. Het is met de transaction log doorgaans niet mogelijk om een volledige disk-interactie na te spelen, maar wel om de operaties op metadata na te spelen.
Dat heeft als effect dat er een filenaam terug te vinden is bij datablokken die anders ongelinkt zouden kunnen rondzwerven na een crash.
Het hele idee van een journal is dat de te wijzigen porties geheugen dicht bij elkaar op de harde schijf liggen; hierdoor is het mogelijk om snel allerlei grootschalige operaties op metadata te verwerken. Het daadwerkelijk over de hele schijf aanpassen van directory entry's gebeurt dan asynchroon, dus via het buffergeheugen. Na een crash hoeft alleen de logfile te worden afgespeeld om alle namen weer bij alle diskblokken te vinden.

Soft Updates

Veel minder bekend dan journaling, maar beslist een zinnig alternatief ervoor, is het mechanisme van Soft Updates zoals dat in de Fast FileSystems van BSD-systemen is ingebouwd. Soft Updates gaan ervan uit dat je na een crash best de laatste seconden (tot aan een minuut) van je interactie met de schijf kwijt mag raken. Het rare is dat de sync-aanroep hieronder asynchroon wordt, en meteen terugkeert. Niet meteen iets wat je onder een database zou wensen – daaronder wil je toch wel zekerheid van schijfopslag hebben voordat je een COMMIT positief beantwoordt! Maar van de andere kant; als het hele systeem stabiel is en in een onbemande ruimte of zelfs een brandkast staat, dan kan er ook weinig meer misgaan dan een stroomstoring – en een UPS die het een minuut uithoudt kan dan zorgen voor voldoende rek om de disk buffers te legen.
De randvoorwaarde waaronder blokken aan de harde schijf worden aangeboden is dat de volgorde nooit een inconsistent filesysteem mag opleveren. De structuur moet altijd correct zijn. Dat houdt in dat er geen recovery nodig is na een herstart, ongeacht het moment waarop de zaak gecrasht is. De aanroepen van de kernel-functie sync worden gebruikt om de zaken daarmee overeenkomstig te ordenen, en de rest is dan de vrijheid van de scheduler die de blokken naar de schijf wegschrijft.

Gedistribueerde filesystemen maken vaak gebruik van lokale cache op elk werkstation

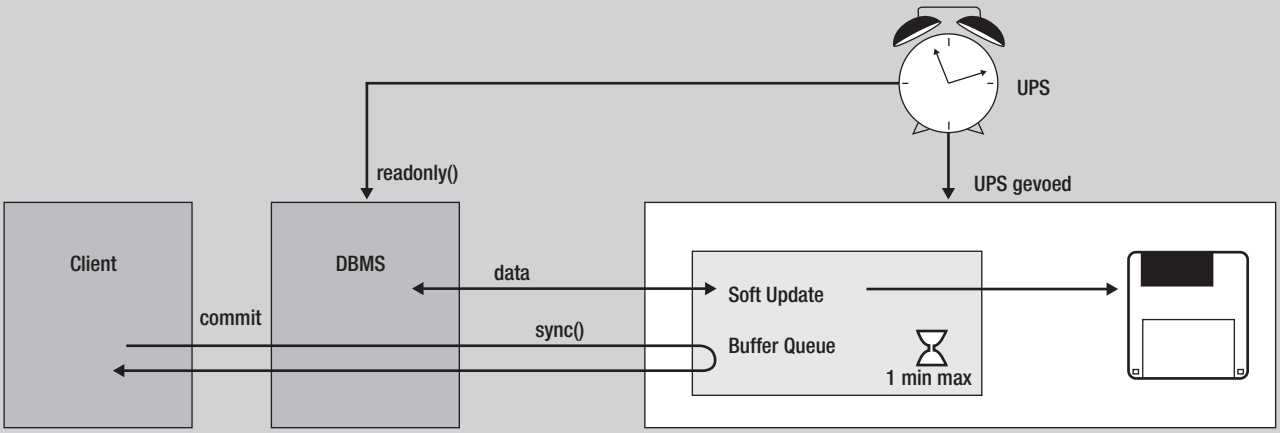
Op een drukbezet systeem bespaart dit 40 procent tot 70 procent op het aantal schrijfacties. Doordat filesystemen onder Soft Updates altijd consistent blijven, kunnen ze onmiddelijk na de herstart worden gebruikt, zonder dat er checks nodig zijn. Journals helpen misschien om sneller te checken, maar ze voorkomen de checks niet en dus vertragen ze een herstart nog steeds. Met Soft Updates kun je onmiddelijk doorstarten, maar daarbij maak je dus wel een stapje terug in de tijd.

Snapshots

Hetzelfde FFS dat ook Soft Updates aankan, kan tegenwoordig ook snapshots van de partitie maken. Dit is werkelijk waar het naar klinkt – een image van de schijf, hardbevroren en dus consistent weergevend hoe de schijf er uit zag op het moment van de snapshot – en dat terwijl de rest van het filesysteem vrolijk doorwerkt en verandert. Tijdens het snapshot maken mag alleen

Database op Drijfzand

Een recept voor een bizarre database-machine, die niemand zal durven bouwen maar die wel tot nadenken stemt.
Men neme een computer, uitgerust met RAID en een stabiel operating systeem. FreeBSD en OpenBSD behoren tot de meest stabiele systemen van dit moment. Men draait deze computer in een fysiek afgesloten serverruimte.
Men installeert op dit systeem een database, met de dataopslag op FFS met Soft Updates geactiveerd. Men bepaalt de maximumtijd die de cache achterloopt, en men gebruike een UPS die daarop gedimensioneerd is. Men regelt dat de UPS bij uitval van de hoofdspanning de database uit de lucht schiet, danwel naar read-only modus overschakelt.
Deze machine reageert als het goed is veel sneller op COMMIT's, omdat een sync van het filesysteem meteen klaar is (zie afbeelding 2). Dankzij de UPS-integratie zijn de kansen voor het verloren gaan van database-inhoud verder tot een minimum beperkt. Of dat minimum acceptabel is mag iedereen zelf bepalen.



Afbeelding 2: Een database met een zwaar versnelde sync()-operatie kan veel sneller committen. Een UPS kan net dat beetje extra zekerheid bieden om Soft Updates tijdig weg te schrijven voor de hele zaak spanningsloos raakt.

een kleine seconde even niet geschreven worden. En om het allemaal nog gekker te maken kunnen er een stuk of twintig van dit soort snapshots worden gemaakt van elk filesysteem.
Een applicatie die voortdurend files schrijft, zoals bijvoorbeeld een mailserver, kan onmogelijk rekening houden met de volgorde waarin een onafhankelijke backup wordt gemaakt. Het kan daardoor best zijn dat de backup een file niet vindt die op een bepaald moment geschreven is, maar wel een andere file die op een later moment geschreven is. Een snapshot is daardoor ideaal voor het maken van backups. Vooral omdat het precies op de juiste plaats zit. In een database kunnen we tijdelijk de schrijfacties op het zijspoor van de transaction log zetten, terwijl we de ruwe data opslaan. Ook een mailserver zou je nog kunnen vertellen dat hij tijdelijk geen e-mail mag aannemen (want SMTP geeft ook een soort COMMIT voor het overnemen van de verantwoording over een e-mail). Maar veel fraaier is het als zo'n server gewoon kan doorstomen met wat hij aan het doen is, en kan vertrouwen op een generiek systeem voor het maken van snapshots die een consistent beeld geven van een filesysteem op een bepaald moment.

Er is nog een andere toepassing van snapshots die de robuustheid van een systeem ten goede komt. Omdat namelijk zo veel mogelijk bestaande blokken uit het filesysteem rechtstreeks worden ingelinkt in een snapshot, is het mogelijk er een reparatie van een filesysteem mee uit te voeren. Na een crash van FFS met Soft Updates kunnen hooguit wat resources (zaken als datablokken en inodes) ongebruikt bezet zijn geraakt, en dat soort dingen kunnen worden gerepareerd door de filesystem-check te draaien op een snapshot. Dus terwijl de rest van het systeem allang weer in de lucht is, draait op de achtergrond een procesje dat nog even de rommel aanveegt.

Distributie

Filesystemen hoeven niet lokaal te blijven. Netwerk-filesystemen zijn al heel lang in omloop, met NFS als de facto standaard. Ook filesystemen kunnen dus in een client/server-interactie worden gebruikt. Dat zou eventueel kunnen op basis van SAN of NAS, maar een gedistribueerd filesysteem zit doorgaans boven het niveau van datablokken die worden overgestuurd – systemen als NFS gebruiken een netwerkprotocol om operaties op het

filesysteem aan te vragen bij een fileserver die dat filesysteem draait.
Een interessant alternatief voor NFS is Andrew, ook wel AFS geheten. Dit systeem is gebouwd voor en door CMU (Carnegie Mellon University in Amerika), waar het 7000 werkplekken op de campus voorziet van centrale opslagfaciliteiten. Schaalbaarheid was een belangrijk ontwerpcriterium voor AFS, alsmede veiligheid. NFS en SMB/CIFS nemen bijvoorbeeld aan dat het LAN betrouwbaar is, en die aanname heeft men niet willen maken met Andrew, omdat er een hele campus op aangesloten was.

> Een applicatie die voortdurend files schrijft kan onmogelijk rekening houden met de volgorde van een onafhankelijke backup

De ontwerpeis is daarom geweest dat op de werkplekken willekeurige code zou moeten kunnen draaien, en dat alleen de file servers betrouwbaar hoefden te zijn om het geheel veilig te houden.
Een verschil tussen de client/server-interactie van databases en van filesystemen is dat een database een snapshot geeft van de data die het vindt, terwijl een filesysteem een verbinding blijft veronderstellen. Wanneer je lokaal iets aanpast gaat dat terug naar de fileserver, en als er op de fileserver iets aangepast wordt zul je dat ook te zien krijgen wanneer je er opnieuw naar kijkt. De client/server-interactie van een filesysteem is dus meer 'statefull' dan die van een database met haar klanten. In een database levert dit weleens semantische problemen, zoals records die nieuw aangemaakt worden maar die volgens de serialisatievolgorde eigenlijk gelockt zouden moeten zijn door een andere transactie. Verder is het voor fileservers veel gemakkelijker dan voor databases om de last te verspreiden over meerdere machines. Dat is voor een deel te verklaren uit de veel eenvoudiger semantiek van een file; een update is niets meer dan een schrijfoperatie, en op basis van het wijzigingstijdstip is altijd te zien welke versie de nieuwste is.

Ontkoppeling en reïntegratie

Gedistribueerde filesystemen maken vaak gebruik van lokale cache op elk werkstation, zodat de server wordt ontlast. Die werken vrij goed doordat mensen en programma's de neiging hebben om kortstondig aan een beperkte hoeveelheid data (de 'werkset') te sleutelen. Op het moment dat die in een cache zit, kan de client ook enige tijd afgekoppeld werken. Denk bijvoorbeeld aan een laptop die mee gaat naar een berghut en die na het weekend met het filesysteem wordt gereïntegreerd.

Coda is een afsplitsing van AFS waaraan nog steeds ontwikkeld wordt; het voegt ten opzichte van Andrew zaken toe als het aanwijzen van files die beslist in de lokale cache van een werkstation moeten komen te staan, zodat na afkoppeling voldoende informatie beschikbaar is om door te werken. Dit soort dingen gebeurt op dit moment nog via het soort tools waar alleen een techneut zich lekker bij voelt, maar dat is een kwestie van tijd. Veel belangrijker is dat Coda op heel wat systemen (Win95 en later, Linux, BSD, Mac OS X) beschikbaar is. CMU omschrijft Coda zelf als stabiel genoeg voor experimentele omgevingen met enige tientallen werkplekken.

Een parallelle wereld

Databases en filesystemen zijn beide complexe systemen voor de opslag van gegevens. Toch is hun toepassingsgebied zo verschillend, en met name hebben ze totaal andere semantiek, dat beide bestaansrecht hebben. Het is dan ook prettig te zien dat filesystemen allesbehalve ondergeschoven kindjes zijn. Er wordt nog steeds hard aan geïnnoveerd, met soms verrassende concepten tot gevolg.

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

Alle files laten intrekken bij een database?

Er zijn weleens mensen die beweren dat alle data in een database moeten. Van Oracle verwacht je zoiets, en ook Microsoft heeft in elk geval serieuze plannen in deze richting met WinFS, maar is het werkelijk zo nuttig om alles in een database te stoppen? Als een filesysteem soortgelijke robuustheid kan bieden als een database valt dat nog te bezien.
Een relationele database is met name goed in het aflopen van tabellen met gelijksoortige gegevens. De soortgelijkheid is in een filesysteem vaak ver te zoeken, en het aflopen ervan eigenlijk ook, want in een filesysteem worden de objecten vaak per stuk aangesproken. En de bouwers van een filesysteem kunnen dat ook echt wel efficiënt implementeren.

Een database moet zich daarbij laten leiden door algemene principes, zonder enige kennis van de specifieke structuur van filesystemen, of het gemiddelde gedrag van haar gebruikers. Al die dingen zitten uiteraard wel verweven in een filesysteem, want zo'n ding bouw je niet zonder flink wat benchmarks en test-runs te draaien. Voor het aanspreken van de ongesorteerde 'troep' die we allemaal liefkozend aanduiden als onze home directory, lijkt een database dus helemaal niet zo'n goed idee.