Praktisch product met SQL front-end als open source beschikbaar

MonetDB keert een DBMS binnenste-buiten

Rick van Rein

Bij het CWI is een unieke database ontwikkeld, MonetDB genaamd. Deze database rijt het hele relationele concept aan flarden om het vervolgens op een vernuftige manier weer anders in elkaar te puzzelen.

MonetDB is altijd een zonderling pakket geweest. Het gooit veel getrouwe gewoontes uit de database-wereld op de schroothoop, maar altijd met een goede onderbouwing. In de jaren negentig was het bijvoorbeeld een raar idee om het werkdeel van een database in het werkgeheugen van de computer te hebben, terwijl tegenwoordige machines dat voor veel toepassingen prima aankunnen. En er is altijd nog de mogelijkheid om *swap* aan te spreken als dat onverhoopt misloopt.
MonetDB is ontworpen als extreem snelle database. Het besteedt daarom veel aandacht aan optimalisaties, en dan met name aan de kernel, het deel waarin database query's op tabellen worden losgelaten. De programmeerregel dat het efficiënt is om code uit een lus te halen en naar buiten te transporteren, wordt voor die kernel veelvuldig toegepast. Dit artikel legt MonetDB daarom van binnen naar buiten uit.

Processoren en geheugens

Huidige processoren hebben een enorme kloksnelheid in vergelijking met de geheugensnelheid. Processoren worden gemeten in termen van GigaHerz, terwijl geheugens snelheden in de orde van tientallen MegaHerz aankunnen. En omdat veel processorinstructies iets met een geheugenlocatie doen, staat een processor dus regelmatig 'verveeld'. Specifiek voor DBMS'en is dat een gegeven dat met onderzoek gestaafd is.
Een uitzondering daarop is als caches hulp bieden. Een L1 cache zit dicht bij de processor en is uitermate snel aan te spreken; een L2 cache zit daar weer omheen en is wat trager maar ook een stuk groter. Het ideaal is om zo veel mogelijk gebruik te maken van zulke caches, zodat het relatief trage RAM minimaal hoeft te worden gebruikt.

Elk proces heeft een zogenaamde 'werkverzameling' van variabelen ofwel geheugenlocaties waarop die enige tijd doorwerkt. Naarmate die werkverzameling kleiner is, is de kans groter dat die in een L2 of zelfs L1 cache past. Voor sommige processen is het daarom aantrekkelijk om een ongebruikelijke implementatie te kiezen die weliswaar meer werk door de processor laat uitvoeren, en die mogelijk vaker dezelfde data aanspreekt, maar die wel een kleinere werkverzameling heeft. Zoals gesteld 'verveelt' een gemiddelde processor zich tijdens traditionele DBMS-bewerkingen.
Ook is het handig gebruik te maken van sequentiële toegangspatronen op de data. De caches rond de processor worden namelijk in burst-mode gevuld met bijvoorbeeld 32 bytes tegelijk.
Als al die data worden aangesproken (zoals bij sequentiële toegang), dan kan de verloren tijd voor het vullen van de cache worden verdeeld over de losse aanspraken op elementen van dat stuk cache. MonetDB kan dit soort dingen allemaal als onderdeel van haar join-operatie verwerken. Verderop zal duidelijk worden dat die operatie voor MonetDB nog veel belangrijker is dan voor een gemiddelde database.

> Het idee van een BAT volgt eigenlijk heel logisch uit de cache-redenatie

Moderne processoren bezitten ook geavanceerde faciliteiten zoals pipelining en parallelle uitvoering van processor-commando's. Daarvoor is het noodzakelijk dat de processor voorspelt wat de volgende stap wordt; klopt de voorspelling niet dan wordt die tak van uitvoering gestaakt en is het werk voor die tak verspild geweest. Zaken die dat ernstig beïnvloeden zijn conditionele sprongen en subroutines. Het genoemde vervelen van processoren wordt door het MonetDB-team geïnterpreteerd als indicatie dat deze zaken veel te veel voorkomen in een gemiddeld DBMS.

Kleine bouwstenen

Om te vermijden dat de processorkracht wordt afgeknepen door de bottleneck van het geheugen kiest MonetDB daarom niet voor een SQL-achtige, generieke implementatie in haar kernel, maar juist hele toegewijde beperkte implementaties, waarvan zonodig varianten zonder veel keuzepunten naast elkaar bestaan.
Op die manier hoeven niet te veel opties en vlaggetjes te worden getest tijdens iets basaals als een join, en kan gewoon doorgejakkerd worden zonder al te veel dingen te doen die caches

herschrijven of die conditionele sprongen maken. Ook worden geen subroutines gebruikt, maar macro-expansies. Er zijn wel wat meer varianten nodig, bijvoorbeeld een join-operatie moet hierdoor voor meerdere types afzonderlijk worden gecompileerd. Maar de binnenste lussen van de database-bewerkingen worden er een stuk sneller van, en dat is wel de meest dankbare plek om zo'n opoffering voor te maken.

MonetDB vormt een kleine algebra van dit soort operaties op beperkt varieerbare types. Het idee van een algebra onder een relationele database is niet bijzonder; semantici gebruiken al jaren algebra's en calculi om de precieze werking van bijvoorbeeld SQL te beschrijven. Bijzonder aan MonetDB is alleen wel dat ook in deze algebra te programmeren is, via een intermediate taal die MIL heet. Deze taal is bedoeld om talen als SQL naartoe te vertalen.
Kenmerkend aan een algebra is dat ze operaties op een datatype definieert. Verder is het bevorderlijk voor de manipuleerbaarheid (zoals tabeloperaties van plaats verwisselen ter optimalisatie) als dat datatype eenvoudig is gestructureerd, omdat daarmee minder uitzonderingen voor kunnen komen en dus minder manipulaties uitgesloten hoeven te worden.

BAT's

Het datatype dat ten grondslag ligt aan MonetDB is de Binary Assocation Table, of kortweg BAT. Dat is een extreem simpele database-tabel, bestaande uit maar twee kolommen. De tuples in een BAT heten BUN, Binary UNit, met respectievelijke elementen genaamd head en tail. Head en tail hebben elk een type en elk type kent een NIL-waarde. Het is niet (altijd) zo dat de head een key moet zijn; bijvoorbeeld is er ook een reverse-operatie die head en tail van een BAT verwisselt.
BAT's kunnen een paar eigenschappen bezitten die de implementatie van de operaties in de BAT-algebra kunnen beïnvloeden. Bijvoorbeeld het feit dat een BAT gesorteerd is op de head, of dat de head dan wel de tail geen dubbele elementen bevat, kan heel zinnige informatie zijn.

Het idee van een BAT volgt eigenlijk heel logisch uit de voorgaande cache-redenatie. Een normale tabel heeft met gemak tussen de 5 en 20 kolommen, en dat is regelmatig goed voor tussen de 200 en 2000 bytes per record. Dat is zo veel dat er al geen winst meer te halen is uit de burst-acties op geheugen. Domweg omdat de kans dat attributen die nodig zijn of die een update behoeven binnen dezelfde 32 bytes vallen. Zet dat uit tegen een BAT, bijvoorbeeld een BAT[OID,INT] die als head een object-ID heeft en als tail een integer, dan passen er 4 BUN's in een 32 bytes burst-blok. Bovendien is de kans dat beide helften nodig zijn groot – enerzijds wordt vaak op de OID gezocht, anderzijds is de kans klein dat deze BAT wordt gebruikt terwijl de tail ervan niet nodig is. Een join op zulke BAT's kan in MonetDB zo efficiënt zijn als 20 processorcycli per BUN in de uitkomst, en dan is de geheugen-overhead enorm en loont het dus om daarop te besparen.

Virtuele OID's

Een OID is een interne identifier van MonetDB, die gebruikt wordt als koppelend element tussen BAT's. Elementaire operatoren zoals optellen van twee INT's om een nieuwe INT te vinden kennen een gemultiplexte variant die twee tabellen optelt en een nieuwe tabel oplevert; de twee invoertabellen en de uitvoertabel worden bij elkaar gezocht op een match met de head, en de tail van de uitkomst is dan de som van de tails van de twee inputs. Dat zijn typisch van die eenvoudige maar krachtige dingen die met een algebra uit te halen zijn.
Het is mogelijk om synced BAT's te maken; dat wil zeggen dat BAT's worden gevormd zodanig dat dezelfde OID's op dezelfde positie in de BAT staan. In zulke gevallen zijn joins en

Een SQL query in MIL

De meest voor de hand liggende implementatie van join is er een die over de linker tabel itereert en daar per record een tweede record bij zoekt uit de rechter tabel. Als de tabellen geordend zijn is bij wijze van optimalisatie een soort merging-join mogelijk. Voor beide implementaties wordt alleen wel zo ongeveer de totale lengte van twee records in de processor-cache binnengesleept. Dat doet MonetDB heel anders.

Neem bijvoorbeeld deze SQL-query:

```
SELECT voetballer.naam, team.naam
FROM voetballer, team
WHERE voetballer.team = team.id
```

In MIL bepaal je eerst de join, en daarna markeer je een paar nieuwe OID-waarden voor de nieuwe deelrelaties. Merk op dat de team.id hier niet wordt gebruikt als OID-waarde in MIL.

```
joined := join (voetballer_team, team_id)
ballenjongens := joined.mark(0).reverse
teams := joined.reverse.mark(0).reverse
```

Hierna kunnen enkele uiterst efficiënte positionele joins worden uitgevoerd. Dat houdt in dat de volgende tijdelijke tabellen allemaal dezelfde data op dezelfde positie bevatten.

```
namen := join (ballenjongens, voetballer_naam)
teamnamen := join (teams, team_naam)
```

Het resultaat ontstaat nu door de namen en teamnamen naast elkaar af te drukken. En dat wordt natuurlijk netjes ondersteund in de BAT algebra:

```
print (namen, teamnamen)
```

MonetDB in de praktijk

MonetDB is gemaakt door een onderzoeksgroep onder leiding van prof. dr. Martin Kersten van het Centrum voor Wiskunde en Informatica in Amsterdam. Het is een zeer praktisch product, onder andere uitgedragen door het spin-off bedrijf Data Distilleries B.V. dat data mining-oplossingen bouwt op bases van MonetDB. Het is voor deze toepassing ingezet bij grote bedrijven zoals Aegis, de Postbank en ABN Amro. De data mining-code is nog steeds eigendom van het bedrijf.

Nadat MonetDB getest was in deze commerciële omgevingen is het uitgebracht als open source code. Ook hier doet MonetDB het anders dan gebruikelijk dus. Bij de open distributie hoort de SQL front-end en te zijner tijd ook de XML front-end.

gemultiplexte operaties zoals de gemultiplexte optelling erg efficiënt te implementeren.
Een OID wordt bij inserts uitgegeven als een oplopende nummerreeks. Dat houdt in dat er, vooral wanneer er niet te veel data verwijderd worden, een vrij goed gevulde oplopende reeks van OID's in de heads van een BAT staat. In het meest extreme geval is de OID zelfs één-op-één te koppelen aan een positie.
En hier komt de Virtuele OID naar voren. Dat is een OID die overeenkomt met de positie, en die dus weggelaten kan worden. Zo'n OID heet dan een VOID. Een grapje van de maker waarschijnlijk – in C en C++ is void het type voor 'niets'. Een BAT mag maar één kolom hebben die VOID is, en meestal is dat de HEAD. De andere kolom blijft over, dus de tabel wordt hierdoor simpel een array. Dat is een meesterzet van de maker, vooral omdat hier regelmatig sequentieel doorheen gelopen zal moeten worden. In termen van een cache passen er maar liefst 8 BUN's van een BAT[VOID,INT] in een 32-bytes burst-read.
MonetDB implementeert het traditionele database-concept dus eigenlijk 'binnenste-buiten': in plaats van de opslag te structureren als een opeenvolging van records, worden de kolommen afzonderlijk opgeslagen in een soort array. Horizontale fragmentatie wordt verticale fragmentatie.

BAT algebra

Het updaten van een BAT is uitermate eenvoudig; er zijn insert statements die een BUN of BAT toevoegen aan een BAT; er zijn delete statements die werken als de head overeenkomt met een gespecificeerde waarde, of zowel head als tail. Verder zijn er replace-operaties die de tail vervangen in BUN's waarvan de head overeenkomt met een gespecificeerde waarde.
Hierbij wordt de head dus als een soort key gebruikt, ook al is het key-concept niet expliciet in MonetDB aanwezig. Om tegemoet te komen aan andere wensen is er ook een reverse operatie die een heleboel dubbele operaties (met tail als key) onnodig maakt, zodat bijvoorbeeld voor sortering alleen een operatie die sorteert op head-waarde nodig is. De MonetDB Intermediate Language MIL waarin deze operaties uit te drukken zijn, zijn dan ook niet primair bedoeld voor programmeergemak. Ze hebben vooral een logische, orthogonale structuur, waardoor allerlei automatisch gegenereerde combinaties op de logisch te verwachten wijze werken – en dat kan van SQL lang niet altijd gezegd worden! Volgens deze redenatie zijn er dan ook algemeen werkzame set-operaties. Daarbij kan wel altijd gekozen worden of de head bepalend is voor wat de set-elementen zijn, of head plus tail. Dankzij reverse zijn, wederom, geen extra combinaties nodig.
Er is een unique die net als in SQL dubbelen verwijdert, er zijn union én intersection-operaties die een stuk algemener werken dan in SQL, en er is zelfs een difference die de elementen bepaalt die wel in één set zitten en niet in een andere. Met deze operaties op BAT's zijn veel bewerkingen op verzamelingen goed te implementeren; alleen inverteren van een verzameling ontbreekt, en dat komt door de (schier) oneindigheid van de omspannen waardedomeinen. De difference-operatie maakt het wel mogelijk veel dingen te implementeren die weleens met een inverse worden gespecificeerd. Voeg hieraan toe dat MonetDB intern onthoudt of een BAT gespeend is van duplicaten, dan is een efficiënte implementatie van de operaties binnen handbereik.

Natuurlijk zijn er de nodige middelen om te bepalen of een element in een set voorkomt; wederom is er de keuze te zoeken naar een head alleen of naar head plus tail. En natuurlijk is er een mogelijkheid om de tail bij een head te bepalen. En wegens genoemde de snelle vindbaarheid van een positie in de tabel is er ook een mogelijkheid die positie eerst te bepalen en daarmee op zoek te gaan naar tail-waarden. Dit is ideaal voor implementaties bovenop MIL die het optimaal haalbare uit de database kernel willen halen.

> De fragmenten van allerhande SQL-joins liggen dus voor het oprapen

Het select-commando is eenvoudig; het selecteert een waardenbereik. Toch zijn hiermee allerlei vergelijkingsoperaties te maken. Er gaat een BAT in, desgewenst een ondergrens en desgewenst een bovengrens, en er komt een BAT uit met daarin de juiste elementen. Deze operatie is een typisch voorbeeld van eent die afzonderlijk gebouwd wordt voor verschillende types – een integer en een string vergelijk je nu eenmaal niet op dezelfde manier.
BAT's zijn dus niet alleen de op schijf opgeslagen gegevens, het zijn ook de tussenresultaten. Zo'n tussenresultaat is een niet-persistente BAT. Door toevoegingen aan een persistente BAT te maken zijn zulke tussenresultaten wel persistent te maken. En om op een tijdelijke versie van een persistente BAT te kunnen werken volstaat het om een kopie te maken. Wederom is de logica

verbluffend simpel, en veel helderder en combineerbaarder dan die van SQL: ideaal dus voor codegeneratie.
Specifiek voor de ondersteuning van SQL-achtige constructies kent de BAT-algebra operaties voor join, semijoin en outerjoin. Deze koppelen typisch twee BAT's op basis van één van de twee componenten en geven als resultaat een BAT met de andere componenten erin. Verder is er een zogenaamde 'theta join' functie die joins maakt op basis van andere vergelijkingen dan gelijkheid. De fragmenten van allerhande SQL-joins liggen dus voor het oprapen. In het kader geven we een voorbeeld van zo'n MIL-implementatie van een SQL-query met een ingebouwde join.

Doe-het-zelf faciliteiten

In dit hele betoog is nog niets gezegd over indexes. Terwijl die zo belangrijk zijn bij de optimalisatie van SQL, spelen ze in MonetDB geen enkele rol. Dat komt doordat ze bovenop MIL zelf te maken zijn. Immers, wat een index doet is de opslagplek vinden bij een bepaalde waarde of set waarden. Zo'n zoektocht is in MonetDB goed te implementeren met de bestaande BAT die al niets te veel bevat; als die bovendien 'synced' is met de gezochte records kan er razendsnel worden overgestapt op die andere BAT. En als dat wat te kort door de bocht is, zijn er de nodige tussenvormen mogelijk met MonetDB.
Ook een doe-het-zelf faciliteit is de transactie. Er is ondersteuning voor globale transacties door middel van sync, commit en abort statements, maar die zijn coöperatief, ofwel die kun je gebruiken als je wilt of niet als je niet wilt. Hiermee is het bijvoorbeeld mogelijk om read only-applicaties te maken die niet zo sterk hangen aan de isolerende eigenschap van een RDBMS; met name data mining is hier een voorbeeld van. MonetDB hoeft niet op het niveau van MIL te worden aangesproken, het krijgt hier alleen veel aandacht omdat het een kijkje in de keuken biedt. Er is gewoon een toplaag die SQL implementeert, dus het kan ook worden gebruikt als vervanger van dat standaardgereedschap. Er is verder een native XML interface in de maak, want XML opslaan in SQL is een noodgreep die eigenlijk niet ideaal is. Verder is er een (gesloten) implementatie van een data mining-toepassing.

Tenslotte

MonetDB is een obstinate maar desondanks goed doordachte wijze van implementatie voor het relationele database-concept. De MIL taal zou weleens een veel prettiger ondergrond kunnen zijn om code voor te genereren dan SQL; dat komt dan niet omdat MIL zo'n hoog niveau van abstractie biedt, maar omdat het veel beter mogelijk is deelbewerkingen samen te stellen zonder dat de structuur van de taal in de weg gaat zitten.
MonetDB is gratis beschikbaar als open source-toepassing. Het is te vinden op monetdb.cwi.nl – hoewel het nu geldt als 'redelijk stabiel' wordt er hard aan gewerkt het te verbeteren. MonetDB is er eentje om in de gaten te houden!

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