Transformatie van boomstructuur naar tabellen

Zoeken in XML: van XPath naar SQL

Rick van Rein en Maurice van Keulen

Het opslaan en manipuleren van XML-documenten in een gewone relationele database is niet eenvoudig, omdat XML-documenten een boomstructuur bezitten en volstrekt niet lijken te passen in de tabellen van het relationele model. Een eenvoudige transformatie kan dat probleem echter heel snel wegnemen.

XML-documenten en relationele databases hebben een duidelijk een van elkaar verschillende structuur. De eerste zijn gebaseerd op bomen, de tweede op tabellen. Daarnaast speelt de volgorde van elementen in een relationele database geen rol, terwijl de nodes in een XML-document wel degelijk een ordening kennen.

Een boom opzetten

Een voor de hand liggende uitwerking van een boomstructuur (maar zoals verderop duidelijk zal worden geen bijster efficiënte) is die waarin elk record in het relationele model verwijst naar een daarboven gelegen parent record, met NULL als parent-verwijzing voor de wortel. Een XML-document zoals te zien in afbeelding 1 laat zich bijvoorbeeld in zo'n tabel invoeren als:

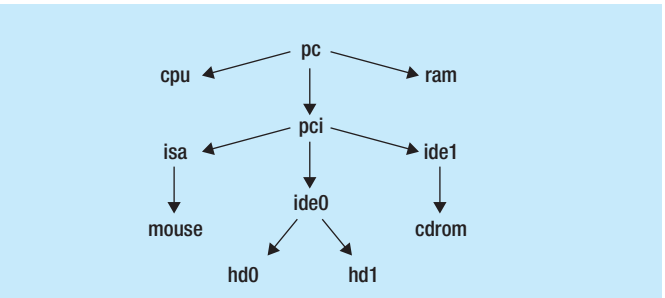
```
create table hardware (node integer not null,
parent integer, tag varchar, ..., primary key (node));
insert into hardware values (1, NULL, 'pc', ...);
insert into hardware values (2, 1, 'cpu', ...);
insert into hardware values (3, 1, 'pci', ...);
...
insert into hardware values (5, 3, 'isa', ...);
...
</>
```

Dit eenvoudige schema maakt het mogelijk om vanuit een node naar de parent of child te navigeren, van daaruit door naar diens parent of child, enzovoort. Hoewel dit in het algemeen nuttige operaties zijn op bomen, is het recursief meerdere joins achter elkaar uitvoeren op potentieel grote tabellen geen recept voor een efficiënte implementatie. Ook speciale primitieven zoals Oracle's CONNECT BY PRIOR geven niet voldoende performance. Zoals deze operatie zich voordoet, lijkt het dat 'onder water' gewoon per stap omhoog in de boom een query wordt uitgevoerd. Bovendien, XPath kent meer navigatieprimitieven, bijvoorbeeld following-sibling, de nodes rechts naast de huidige node met dezelfde parent. Dergelijke primitieven zijn met bovenstaande tabel al helemaal niet te evalueren.

De grote uitdaging van XPath is dat XML-documenten willekeurig diep en breed kunnen worden, alle 13 assen en predikaten goed ondersteund moeten worden en dat een XPath-expressie desondanks in korte tijd uitgevoerd zou moeten worden. Het best haalbare is een performance per stap die lineair schaalbaar is met het aantal nodes in de tabel (complexiteit O(N) met N het aantal nodes in de tabel).

> Het best haalbare is een performance per stap die lineair schaalbaar is

Gelukkig blijkt dit goed te verwezenlijken met gewone SQL-query's. Daarbij is het wel nodig om de data wat handiger op te slaan, namelijk met een pre- en post-order nummer voor elke node. In XML-termen is de pre-order de nummering van de open-tags en de post-order die van de sluit-tags. Dit is hetzelfde als depth-first door de boom wandelen en bij het eerste contact met een node deelt men een pre-order nummer uit en bij het laatste contact een post-order nummer. In afbeelding 2 is de pre-order nummering in groen aangegeven en de post-order nummering in rood. Met behulp van een enkele doorgang door een XML-document kunnen de pre- en post-order nummers bepaald worden.



Afbeelding 1: Een eenvoudig XML-document met daarin een representatie van de hardware in een computer.

Wat is XPath?

Als onderdeel van de XML-familie van standaarden, is XPath bedoeld om trajecten in een XML-document af te leggen: bijvoorbeeld in de patronen die XSL probeert te matchen; in een XPointer die achter een URL kan komen; en natuurlijk binnen XQuery.

Een XPath-expressie begint altijd ergens in het document, mogelijk op meerdere nodes tegelijk, waarna in een aantal door slashes gescheiden stappen relatieve sprongen door het document worden gemaakt. Nodes die een stap niet kunnen maken komen niet meer voor in de uitkomst van zo'n stap. In dit artikel wordt als voorbeeld de volgende XPath-expressie gebruikt:

```
ancestor::pc//mouse
```

De eerste stap bepaalt vanuit een start-node (of set start-nodes) alle hiërarchisch bovenliggende nodes, waaruit alleen diegene overblijven die de tag pc hebben. De tweede stap wandelt van daaruit door naar een onderliggende node mouse, waarin met de dubbele slash is aangegeven dat daar zelfs meerdere niveaus van de XML-boom tussen mogen liggen.

Het belangrijkste van XPath zijn de verplaatsingen langs assen: vanuit een willekeurige node kan men een stap maken naar andere nodes via dertien assen zoals child, ancestor, following, preceding-sibling, etcetera.

Bijvoorbeeld `/descendant::cdrom/parent::*(//cdrom/..` is verkorte notatie hiervoor) op het document van afbeelding 1 geeft de interface waaraan de cdrom gekoppeld is. De query doet dit door naar de descendants van de root (de pc-node) te navigeren die het label 'cdrom' hebben, om vervolgens naar de parent van die nodes te navigeren. XPath-expressies kunnen ook predikaten bevatten, bijvoorbeeld

```
/descendant::ide0/child::*[position()=2]
```

(of `kortweg //ide0/*[2]`).

Deze query vraagt naar het tweede 'kind' van de ide0-interface.

Aan de gegeven tabel kunnen deze gegevens worden toegevoegd met de code:

```
alter table hardware add column pre integer;
alter table hardware add column post integer;
...invullen van pre/post waarden...
alter table hardware alter column pre set not null;
alter table hardware alter column post set not null;
alter table hardware drop column node;
```

De reden om een node te verwijderen is dat zowel pre, als post, als de combinatie van die twee, kunnen dienen als sleutel. Indexen op zulke waarden lijken dus nuttig.

Een aardigheid in deze boomrepresentatie is dat in deze attributen geen uitzonderingen voor NULL-waarden meer nodig zijn; de root van de boom heeft immers net als alle andere nodes gewoon een pre- en post-order nummer. Dit betekent dat de SQL-code die op deze getallen werkt veel eenvoudiger kan worden, doordat er met minder uitzonderingen rekening hoeft te worden gehouden. Zulke uitzonderingen zijn doorgaans een bron voor denk- en onderhoudsfouten en ze maken extra tests noodzakelijk.

Het pre/post-vlak

De toepassing van pre- en post-order nummering wordt onmiddelijk zichtbaar wanneer deze nummers als coördinaten worden gebruikt voor de plaatsing van de nodes in een vlak. Afbeelding 4 toont hoe dat er uit ziet. Wanneer de navigatie start vanuit node ide0, die daarmee de rol van de zogenoemde context node speelt, dan zijn er vijf hoofdassen die men kan gebruiken (zie afbeelding 3). Merk op dat de nodes behorende bij die hoofdassen heel makkelijk te herkennen zijn in het vlak: de ancestors zitten linksboven de context node, de following nodes rechtsboven enzovoort. Dit geldt niet alleen voor ide0, maar voor elke willekeurige context node in het vlak.

Deze eigenschap laat zich gemakkelijk vertalen in simpele vergelijkingen van pre-order en post-order nummers. Om te zoeken naar de ancestors van ide0, kan gezocht worden naar alle nodes waarvoor geldt: pre(ide0) > pre(node) en post(ide0) < post(node). De weg naar een SQL-query is nu niet ver meer.

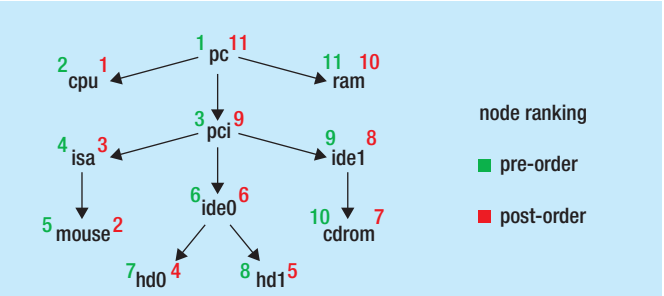
Een concrete XPath-expressie om vanuit een context node zoals ide0 de muis te vinden die bij dezelfde pc hoort is bijvoorbeeld:

```
context/ancestor::pc//mouse
```

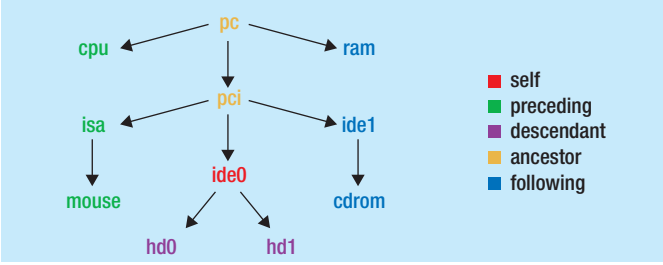
Deze expressie identificeert eerst alle ancestors van ide0 (de gele nodes in afbeelding 3 en 4), vervolgens wordt daaruit de node met tag pc geselecteerd, en van daaruit worden alle descendants van die pc-node gezocht die mouse als tag hebben. Het is overigens geen overbodige luxe om in die laatste stap van descendant gebruik te maken (de dubbele slash in XPath), omdat een muis soms onder pci/isa hangt, soms onder pci/pci.hub/usb/usb.hub, of soms met een nog vreemder pad. Dit is dus wel degelijk iets dat een programmeur opschrijft zonder daarbij lui te zijn.

Omzetten van XPath naar SQL

Een XPath-expressie is om te zetten in SQL per axis step, waarbij elk tussenresultaat een node-verzameling is. We nemen aan dat de node(s) van waaruit we navigeren in een tabel 'context' zitten,



Afbeelding 2: De XML-boom van afbeelding 1, na toevoeging van pre- en post-order nummering.



Afbeelding 3: Startend vanuit node ide0 valt een XML-boom in vijf partities uiteen ten behoeve van XPath-expressies.

bijvoorbeeld één rij voor ide0. Voor het voorbeeld wordt dit:

```
context/ancestor::pc//mouse
       a            b     c
```

In SQL-termen kan het volgende raamwerk gehanteerd worden:

```
select distinct c.pre, ...
from context a, hardware b, hardware c
where 'b is een ancestor van a'
and 'tagname van b is pc'
and 'c is een descendant van b'
and 'tagname van c is mouse'
order by d.pre
```

Met als resultaat:

```
select distinct c.pre, ...
from context a, hardware b, hardware c
where b.pre < a.pre and b.post > a.post
and b.tag = 'pc'
and c.pre > b.pre and c.post < b.post
and c.tag = 'mouse'
order by c.pre
```

En hiermee is de XPath-expressie geïmplementeerd in zuiver relationele termen! Er zijn geen herhaalde query's voor nodig, geen speciale SQL-operatoren en al helemaal geen goocheldoos om de aansluiting tussen XML en relationeel te bewerkstelligen. Alle operaties zijn simpele vergelijkingen op integer- of string-waardige attributen, die goed ondersteund worden door indices. Kortom, allemaal dingen die zelfs een lichtgewicht database als MySQL zonder meer al heel goed kan!

Volledig XPath

XPath bestaat uit meer dan alleen verplaatsing langs assen en kent ook meer assen dan de vijf waarop we ons tot nu toe gericht hebben. Die extra assen zijn eenvoudig toe te voegen. Assensamenstellingen zoals ancestor-or-self zijn te evalueren door simpelweg de 'kleiner dan' en 'groter dan' van ancestor te vervangen door 'kleiner-gelijk' en 'groter-gelijk'. Xpath-predikaten die naar attribuutwaarden refereren of het bestaan van bepaalde nodes testen, zijn ook heel gemakkelijk in SQL op te schrijven. Predikaten op posities zoals 'position()=2', zijn helaas niet goed met deze aanpak te doen. Deze vergen nader onderzoek. Voor assen als parent en following-sibling, is het parent-attribuut intact gelaten. Op zich is de parent afleidbaar uit de pre- en post-nummering, maar dat is niet op een voldoende efficiënte manier te evalueren en daarom is het wel praktisch om het cachende parent-attribuut in stand te houden:
de parent van een node is, van alle nodes met een lager pre-order nummer en een hoger post-order nummer, diegene waarvan de afstand tussen pre- en post-nummer minimaal is.

Optimaal XPath

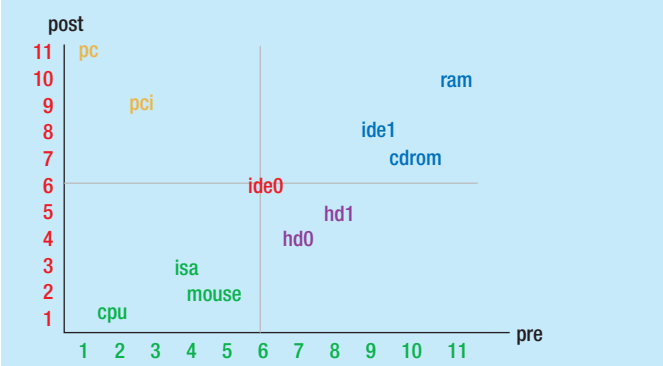
De grote winst van de geïntroduceerde techniek is vooral het vinden van alles in de partities uit afbeelding 3 binnen een enkele tabeldoorgang. We maken hierbij gebruik van zeer nuttige eigenschappen van de nummering, eigenschappen waarvan een techniek als Oracle's CONNECT BY PRIOR geen gebruik van kan maken. Bovendien is XPath een beperkt taaltje waarin slechts enkele, voor XML-documenten zinnige, constructies mogelijk zijn. Dit geeft extra kennis over de aanspreekwijze van de database, en daarmee extra mogelijkheden voor een geoptimaliseerde verwerking.

> XPath is bedoeld om trajecten in een XML-document af te leggen

Een laatste argument dat pleit in het voordeel van deze aanpak, is dat alle operaties intern in de database plaatsvinden. De tussenwaarden a/b worden intern in de database bewaard, en niet eruit gehaald om er later weer ingestopt te worden, zoals vaak bij naïeve implementaties nodig is. De a/b/c join kan erg groot worden, maar interne optimalisaties in de database maken het mogelijk om hier slimme dingen mee te doen.
Onder andere ligt daarvoor ruimte verscholen achter de qualifier DISTINCT in bovenstaande query; immers, als er meerdere paden gevonden worden naar dezelfde node, dan zullen al die nodes normaal gesproken ook optreden in het eindresultaat van de query. En dat is niet conform de XML-semantiek. Ook is er strikt

Geen puur theoretisch onderzoek

In dit artikel wordt de uitkomst besproken van een gezamenlijk onderzoeksproject van de Universiteit Twente en de Universiteit van Konstanz in Duitsland. Niet alleen is in dit onderzoek gewerkt aan praktische vragen over het efficiënt bevragen van XML, maar bovendien zijn er aanpassingen gemaakt aan een paar open source databases om de principes te demonstreren. De handschoen is toegeworpen; commerciële aanbieders worden expliciet uitgedaagd om hun code aan te passen op basis van deze techniek.



Afbeelding 4: Het pre/post-vlak behorende bij de XML-boom met nummering volgens afbeelding 2.

genomen nog een ordening nodig om de records in de document-volgorde te krijgen, zoals de Xpath-standaard voorschrijft, maar dat geldt alleen voor het eindresultaat en niet voor de tussen-resultaten. Dus de database zou tijdelijk kunnen afwijken en naderhand sorteren als dat sneller mocht blijken te zijn.
De performance van een normale database op XPath query's gebruik makend van de geïntroduceerde nummering is snel, maar niet het meest optimale wat men zou kunnen halen. Op zich heeft een database natuurlijk van zichzelf geen kennis van bomen en wat voor slimme dingen er mee te doen zijn. Er is bijvoorbeeld een boom-bewuste JOIN bedacht, genaamd staircase join, die de performance nog een orde van grootte kan verbeteren. De JOIN is geïmplementeerd voor en toegevoegd aan de veelgebruikte public-domain PostgreSQL database, alsmede voor de ook public-domain onderzoeksdatabase MonetDB.
Zonder in details te willen treden over deze geavanceerde optimalisatie, is het wel interessant te melden dat hiervoor de pre- en post-nummering als speciaal datatype is toegevoegd (iets waarvoor PostgreSQL modulaire uitbreidingsmogelijkheden biedt) – het is nog steeds gewoon een INTEGER, maar met de toegevoegde kennis dat het gaat om een nummering binnen een boom. Een test als C.PRE < D.PRE is hiermee onmiddelijk te ontmaskeren als een operatie op een boom, en dus als een goede aanleiding tot optimalisatie in de zin van het inzetten van de staircase join.

Dynamische XML-documenten

Het hier beschreven onderzoek is uitgevoerd op een statisch XML-document. Meestal wil men meerdere documenten opslaan in de database. Een effectieve manier om dit te doen is simpelweg een attribute document-ID toevoegen aan de tabel en in alle query's ook eisen in de WHERE-clause dat de gezochte nodes dezelfde document-ID hebben als de nodes in de context. Een index op document-ID of een gecombineerde op document-ID en pre-order nummer, maken dit een efficiënte ondersteuning voor meerdere documenten.
Ook zijn INSERT- en DELETE-operaties buiten beschouwing gelaten. Dit lijkt echter ook niet tot grote problemen te leiden; voor een INSERT hoeft alleen een tussennummer bijgemaakt te worden op de juiste plaats in de pre- en post-nummering, waarna een nieuw record met die nummers kan worden toegevoegd. Een DELETE van een record komt neer op verwijdering. Het verlagen van de er opvolgende pre- en post-nummers is evenwel niet noodzakelijk. Bijvoorbeeld het inpluggen van een toetsenbord na de muis (met pre=5 en post=2) gaat als volgt:

```
begin;
...ontdek dat plugin na pre=5 en post=2 terecht
                                                komt...
update hardware set pre  = pre +1 where pre  > 5;
update hardware set post = post+1 where post > 2;
insert into hardware (pre, post, tag, ...) values
                                (5+1, 2+1, 'kbd', ...);
commit;
```

Het fraaie van deze UPDATE-operaties is dat ze goed kunnen combineren met concurrency control, gegeven de kennis dat het om een boomnummeringsschema gaat. Operaties als 'verhoog-alles-boven-zoveel' zijn zo eenvoudig dat een database daar heel goed gespecialiseerde ondersteuning voor kan aanmaken. Dat zou inhouden dat alleen de INSERT hierboven echt transactioneel moet worden uitgevoerd, en dat komt eigenlijk op eenvoudige atomiciteit neer. Zelfs dit grenst aan de mogelijkheden van een eenvoudige MySQL database.

XPath is klaar voor toepassing

Wanneer XML-documenten eenmaal in een database zijn opgeslagen met de hier beschreven pre- en post-nummering, dan blijken de specifieke operaties die XPath van een database vraagt eenvoudig om te zetten in normale relationele query's in standaard SQL. Dit is bovendien uiterst efficiënt.
Het enige punt waarop nog aardig wat te verbeteren valt, is het optreden van dubbele uitkomsten en het kwijtraken van de document-ordening onderweg. Dit is te optimaliseren door in te grijpen in de interne structuren van de database, maar ook dit blijkt goed mogelijk door bijvoorbeeld het toevoegen van een speciale join zoals de staircase join.

Rick van Rein en **Maurice van Keulen**
Dr. ir. H. van Rein (rick@openfortress.nl) is ontwikkelaar en beheerder bij OpenFortress Digital signatures.
Dr. ir. M. van Keulen (m.vankeulen@utwente.nl) is universitair docent aan de Universiteit Twente.

Meer informatie op Internet

- de introductie van de XPath-accelerator; www.inf.uni-konstanz.de/~grust/files/xpath-accel.pdf
- de staircase join optimaliseert query's op bomen; http://db.cs.utwente.nl/Publications/PaperStore/db-utwente-0000003277.pdf
- de implementatie van de staircase join in PostgreSQL; http://db.cs.utwente.nl/Publications/PaperStore/db-utwente-0000003577.pdf