ABIS NIEUWSBRIEF VOOR DB2 PROFESSIONALS

JAARGANG 10 - NUMMER 4, SEPTEMBER 2017

EXPLORING DB2

OPEN CURSOR

Op Linux, Unix & Windows is Db2 11.1 ondertussen op de meeste plaatsen in gebruik; en de meeste Db2 for z/OS-"shops" zijn stilaan plannen aan het maken voor migratie naar Db2 12. Hoog tijd dus om in dit nummer van Exploring Db2 nog even stil te staan bij enkele interessante (nieuwe) mogelijkheden van versie 11.

We proberen ook sinds juni Db2 (met kleine b) te schrijven i.p.v. DB2 -- een subtiele naamswijziging die Db2 een meer eigentijdse uitstraling moet geven. Want inderdaad: Db2 evolueert mee met z'n tijd, en komt stapsgewijs tegemoet aan de steeds veeleisender verwachtingen van het data-landschap. Dat "stapsgewijs" wordt vanaf Db2 12 trouwens eigentijdser ingevuld, nl. als "continuous delivery". Meer hierover in een volgend nummer.

Alvast veel leesgenot,
Het ABIS Db2-team.

IN DIT NUMMER:

4

CLOSE CURSOR

In een volgende nummer van Exploring Db2 vindt u deel 2 van de bijdrage over JSON, en wordt ook de reeks over de optimizer verdergezet. U kunt zich ook verwachten aan een verslag van de IDUG conference die begin deze maand in Lissabon doorging.

Over welke onderwerpen wilt u meer lezen? Meld het ons: training@abis.be

Het ARRAY datatype *Peter Vanroose (ABIS)*

Db2 11 for z/OS bevat enkele opmerkelijke nieuwe mogelijkheden. Een opvallende SQL-nieuwigheid is het nieuwe datatype "`ARRAY`". Andere RDBMS kennen dit datatype al een poosje; zo heeft b.v. Db2 for LUW reeds sinds versie 9.5 een ARRAY-type (met trouwens iets meer mogelijkheden dan het ARRAY-type van Db2 11 for z/OS).

Een array declareren

Kort samengevat is een ARRAY een *verticale lijst van gegevens van hetzelfde type*.

ARRAY is dus veel meer dan één nieuw datatype: het is zelfs een hele reeks nieuwe datatypes, geparametriseerd als het ware met een "gewoon" datatype zoals INT of VARCHAR(24) of DATE of zo.

Om een "concreet" nieuw ARRAY-datatype te verkrijgen heb je dan ook (eenmalig) een DDL-statement van de volgende vorm nodig:

```
CREATE TYPE array_txt AS varchar(24) ARRAY [ 1000 ] ;
```

Deze instructie bevat essentieel drie vrij te kiezen ingrediënten:

- de (nieuwe) naam voor dit specifieke datatype; hier: `array_txt` die naam is uiteraard vrij te kiezen, mits niet conflicterend met een reeds bestaande naam;
- het basis-type waarop dit array-type is gebouwd; hier: `varchar(24)`
- en, last but not least: de lengte van de array; hier dus 1000.

Wanneer we straks een "object" van het nieuwe type `array_txt` zullen aanmaken, maken we dus eigenlijk meteen 1000 varchar(24)'s aan! Een dergelijk object kan dus gezien worden als een "verticale lijst", een soort kolom dus, van 1000 velden (cellen) van hetzelfde type.

Uiteraard registreert Db2 elk aldus aangemaakt ARRAY-datatype in een catalog-tabel, namelijk in `SYSIBM.SYSDATATYPES`:

```
SELECT name, sourcetype, length, metatype, arraylength
  FROM sysibm.sysdatatypes WHERE definer = USER ;
```

```
NAME       SOURCETYPE  LENGTH      METATYPE  ARRAYLENGTH
---------- ----------- ----------- --------- ------------
ARRAY_TXT  VARCHAR              24 A                 1000
```

(Vreemd genoeg heeft de kolom `arraylength` een andere naam in Db2 LUW, namelijk `array_length`; verder is alles identiek met z/OS.)

Een array aanmaken

Hierboven werd voorlopig in het midden gelaten hoe we een ARRAY-datatype kunnen gebruiken. Laat alvast één zaak duidelijk zijn: in tegenstelling tot built-in datatypes kan een array *niet* gebruikt worden als datatype voor een kolom van een tabel!

Welke andere "objecten" met een datatype bestaan er nog in Db2? Antwoord: variabelen! Alleen... variabelen in Db2 bestaan alleen in de body van een native SQL-procedure, een native SQL-functie of een trigger. Dat is met versie 11 niet veranderd.

Het array-datatype kan dus enkel gebruikt worden binnen de context van SQL PL ! Dat is natuurlijk een zeer belangrijke beperking. Maar lees toch verder, het wordt mogelijk toch nog interessant...

Laten we dus inzoomen op de body van (b.v.) een stored procedure; daar kunnen we nu schrijven (met het zopas aangemaakte type):

```
DECLARE my_list array_txt;
```

waarmee we een "object", met name een ARRAY-variable genaamd `my_list`, hebben gedeclareerd, en dus eigenlijk gecreëerd. Bemerk: een variabele is uitsluitend een in-memory object (van de procedure).

Hoeveel ruimte neemt deze variabele in? Worden er dadelijk 1000 elementen (van 24+2+1=27 bytes) gereserveerd in het geheugen?

Nee, toch niet: initieel is een array "leeg", dus van effectieve lengte 0. De lengte 1000 is een bovengrens. Arrays zijn dus *dynamisch*! (Zoals arrays in b.v. Java of PHP, maar in tegenstelling tot b.v. arrays in C of in Oracle's PL/SQL.)

De bovengrens (in het voorbeeld: 1000) is er eerder om veiligheids-redenen; we hadden het type array_txt ook kunnen declareren met onbegrensde lengte:

```
CREATE TYPE array_txt2 AS varchar(24) ARRAY [ ] ;
```

Eens aangemaakt, kan een array-variabele gevuld worden. Dit kan hetzij element per element, of in één keer. Enkele voorbeelden:

```
SET my_list[23] = 'value'; -- index kan tussen 1 en 1000 (incl) zijn
SET my_list[i] = my_list[i-1]; -- wanneer "i" een numerieke variabele is
SET my_list = ARRAY['value1', 'value2', 'value3', ...];
SET my_list = ARRAY[ SELECT <expr> FROM <table> WHERE ... ];
```

Vooral het laatste voorbeeld is interessant, en verklaart waarom we arrays best als "verticale lijsten" bekijken: het resultaat van een 1-kolom SELECT-statement kan dadelijk als array opgeslaan worden.

(Db2 LUW laat ook multi-column arrays toe; Db2 z/OS niet.)

In plaats van met SET-statements kan een array ook gevuld worden wanneer het in de INTO-clause van een SELECT wordt gebruikt:

```
SELECT ARRAY_AGG(<expr>) INTO my_list FROM <table> WHERE ... ;
```

Bemerk dat we in dat geval de (nieuwe) aggregate functie ARRAY_AGG moeten gebruiken.

Vermits een array een *gesorteerde* lijst is, voorziet ARRAY_AGG de mogelijkheid om de volgorde te kiezen:

```
ARRAY_AGG(<expr> ORDER BY <expr>)
```

Associatieve arrays

Dit zijn varianten van de hierboven geïntroduceerde objecten; nog steeds "verticale lijsten van hetzelfde datatype", maar nu is de "index" van een element van de lijst niet meer een geheel getal (b.v. tussen 1 en 1000), maar een tekstuele waarde (die we de "key" zullen noemen) van type varchar, b.v. VARCHAR(9). Opnieuw een voorbeeld:

```
CREATE TYPE array_txt3 AS varchar(24) ARRAY [ varchar(9) ] ;
DECLARE my_list3 array_txt3;
SELECT name, sourcetype, length, metatype, arraylength
  FROM sysibm.sysdatatypes WHERE definer = USER ;
```

```
NAME        SOURCETYPE  LENGTH      METATYPE  ARRAYLENGTH
----------- ----------- ----------- --------- ------------
ARRAY_TXT   VARCHAR              24 A                 1000
ARRAY_TXT2  VARCHAR              24 A           2147483647
ARRAY_TXT3  VARCHAR              24 L                    9
```

(Bemerk het metatype: "A" = "gewone" array, "L" = associatieve array.) Behalve VARCHAR(n) kan enkel INT als index-type gebruikt worden, maar dan zijn we eigenlijk terug bij "gewone" arrays van onbegrensde lengte. In de praktijk wordt dus altijd VARCHAR(n) gebruikt, voor een waarde naar keuze voor n.

Een associatieve array kan gevuld worden zoals een "gewone", met enkele kleine, voor de hand liggende verschillen:

```
SET my_list3['key'] = 'value'; -- alvast één array-element gemaakt & gevuld
SET my_list3['key'] = my_list[23] ; -- vorige waarde wordt overschreven!
SET my_list3['Jan Janssens'] = '06-123456'; -- een tweede array-element
```

De twee "globale" manieren om en array te vullen, zijn er nu niet meer. Die onderstellen namelijk een element-volgorde, en dat is nu net wat een associatieve array mist!

Arrays als tabellen

Naast de nieuwe (aggregate) functie ARRAY_AGG, die een (resultaat)kolom converteert naar een array, heeft Db2 11 nog een tweede functie, die net het tegenovergestelde doet, nl. een array converteren naar een (tabel van één) kolom: UNNEST. Een voorbeeld:

```
SELECT x FROM UNNEST(my_array) AS t(x);
```

waarbij de array-variabele my_array uiteraard eerst gevuld moet zijn. Een array kan dus effectief gebruikt worden als een in-memory tabel! Met als belangrijke beperking, uiteraard, dat zo'n tabel maar één kolom kan hebben. Alhoewel: twee kolommen zijn nog net mogelijk; Daarvoor moeten we wel van een associatieve array vertrekken:

```
SELECT t.key, t.val
  FROM UNNEST(my_array3) WITH ORDINALITY AS t(key,val)
```

(Zonder "`with ordinality`" is enkel de "`val`"-kolom beschikbaar.)

ARRAY heeft nog meer mogelijkheden in petto. Meer weten? Zie b.v.
`www.abis.be/resources/presentations/idug20161114sqlpl.pdf`

Optimizer query transforms

Peter Vanroose (ABIS)

In deze tweede bijdrage over de nieuwe mogelijkheden van de optimizer van Db2 11 voor z/OS hebben we het over een drietal onderwerpen die nauw met elkaar in verband staan.

We beginnen met een beschrijving van zgn. "*query transformations*" (herschrijven van een SQL-query door de optimizer) met i.h.b. de belangrijke "*predicate pushdown*".

Daarmee komen we onvermijdelijk terecht bij EXPLAIN en z'n tabellen: oude en ook enkel nieuwe.

Dit brengt ons naadloos bij *virtuele indexen*: via EXPLAIN laten deze denkbeeldige objecten ons toe om te verifiëren of (echte) nieuwe indexen de query-performance zouden kunnen verbeteren.
Ook hiervoor hebben we een (nieuwe) explain-tabel nodig.

Query transformations

De taak van de Db2-optimizer: een SQL (b.v. SELECT) statement vertalen naar het meest efficiënte access-path. Dit houdt onder andere in: welke predikaten via welke index implementeren, chronologie van de andere predikaten, tabel-volgorde bij een join, ...

Bij deze vertaalslag kan het helpen, de SQL eerst ietwat anders (maar equivalent) te formuleren, b.v. een redundant predikaat toevoegen, of een subquery herschrijven als join, om zo de eigenlijke optimizer meer mogelijkheden te geven.

Dit is uiteraard geen nieuw fenomeen in Db2 11, maar de optimizer maakt (vooral sinds Db2 10) steeds meer en steeds ingrijpender gebruik van deze voorbereidende stap.

Hoog tijd dus om het fenomeen onder de loep te nemen!

Transitive closure

Dit is wellicht de bekendste, en ook de oudste vorm van automatic query rewrite. Een eenvoudig voorbeeld kan hierbij volstaan:

```
SELECT t.name, t.creator, s.name AS tblspace, s.bpool, s.partitions
FROM   sysibm.systables t JOIN sysibm.systablespace s
       ON t.tsname = s.name AND t.dbname = s.dbname
WHERE  t.creator = USER AND t.dbname = 'TBD785X'
```

geeft een lijst van al "mijn" tabellen in database TBD785X, met wat info over hun tablespace. Predikaten filteren uitsluitend op `systables`. De optimizer voegt echter volgend "redundant" predikaat toe:

```
  AND  s.dbname = 'TBD785X'
```

Dit is inderdaad een simpele, blinde toevoeging: als `A=B AND B=C` voor komt, kan altijd `A=C` toegevoegd worden. Join-condities spelen uiteraard ook mee, en verder kunnen ook andere dan "="-predikaten blindweg gecombineerd worden via transitive closure, zoals b.v.

```
a < b AND b <= c                  =>    a < c
a = b AND a BETWEEN c AND d       =>    b BETWEEN c AND d
a <= b AND a BETWEEN c AND d      =>    b >= c
```

Outer joins zijn delicater, maar ook daar gebeurt (sinds Db2 11) transitive closure.

Predicate pushdown

Dit is misschien wel het belangrijkste aspect van query rewrite, omdat hierdoor de filtering die in één query block gebeurt, doorgeschoven wordt naar een ander query block (waar de filtering hopelijk efficiënter kan gebeuren).

Een query block is een bouwblok van een volledig SQL-statement, dat op zichzelf ook een volledig SQL-statement is. Voorbeelden zijn: subqueries (in een WHERE clause); common table expressions (CTEs); en de twee aparte select-blokken (de twee "benen") van een UNION (of UNION ALL, of EXCEPT of INTERSECT).

"Predicate pushdown" komt neer op het doorschuiven van een predikaat naar "binnen", dus van een buitenste query block naar een query block daarbinnen. Een simpel voorbeeld:

```
SELECT name, dbname, bpool, partitions, segsize
FROM   sysibm.systablespace
WHERE  dbname LIKE 'TBD%'
  AND  (name,dbname) IN (SELECT tsname,dbname FROM sysibm.systables
                         WHERE creator = USER)
```

Dit geeft me al "mijn" tablespaces in databases met namen TBD...

Het `dbname`-predikaat kan doorgeschoven worden naar de subquery:

```
SELECT name, dbname, bpool, partitions, segsize
FROM   sysibm.systablespace
WHERE  (name,dbname) IN (SELECT tsname,dbname FROM sysibm.systables
                         WHERE creator = USER AND dbname LIKE 'TBD%')
```

De belangrijkste reden waarom het goed is dat de optimizer een dergelijke pushdown voor z'n rekening neemt: we hebben (als schrijver van de SQL) niet altijd de mogelijkheid om de "binnen"-SQL aan te passen. Dit geldt in het bijzonder wanneer die binnen-SQL een view-definitie is, zoals b.v.in het geval van een "shadow catalog":

```
CREATE VIEW sys.tables AS (SELECT * FROM sysibm.systables WHERE creator=USER)
```

Als "gewone" gebruiker krijgen we SELECT-autorisatie op deze view maar niet op de catalog-tabel zelf. Zodat we alleen info over onze eigen tabellen kunnen zien. De volgende query lijkt maar één query block te bevatten, maar (na substitutie van de VIEW-definitie) is er eigenlijk een zgn. nested table expression in de FROM-clause:

```
SELECT * FROM sys.tables WHERE name LIKE 'PERS%'
```

Gelukkig wordt het LIKE-predikaat naar binnen geschoven:

```
SELECT *
FROM   (SELECT * FROM sysibm.systables
        WHERE  creator=USER AND name LIKE 'PERS%')
```

Bij dit eenvoudig voorbeeld worden daarna de twee query blocks uiteraard samengevoegd tot slechts één, maar bij complexere queries zal de nested table expression moeten gematerialiseerd worden, en is het dus belangrijk dat zoveel mogelijk predikaten zo vroeg mogelijk toegepast worden!

Bij elke versie van Db2 werden en worden gevallen toegevoegd die voor predicate pushdown in aanmerking komen. Zo gebeurde er in Db2 10 enkel "pushdown" voor "eenvoudige" (stage-1) predicates naar een table-expressie toe (CTE of NTE, zoals in het voorbeeld hierboven). Pas in Db2 11 gebeurt dit eveneens voor OR-predikaten, voor stage-2 predikaten en voor outer join ON-predikaten. En sinds Db2 12 voor de "benen" van een UNION (ALL), indien dit efficiënter is:

```
CREATE VIEW sys.tables(name,type,dbname,ts,cols,nrpar,nrchi,nrkey) AS (
   SELECT name,type,dbname,tsname,colcount,parents,children,keycolumns
   FROM   sysibm.systables WHERE creator=USER
  UNION ALL
   SELECT name,'S','','',-1,-1,-1,-1
   FROM   sysibm.syssynonyms WHERE creator=USER )
;
SELECT * FROM sys.tables WHERE SUBSTR(name,1,4) = 'PERS'
```

Stage-2 naar stage-1

Een interessant geval van query transformation is het herschrijven van *predicates*, dus WHERE-condities. Zoals bij voorbeeld het laatste predikaat in het vorige voorbeeld (dat stage-2 en dus inefficiënt is).

In het bijzonder herschrijft Db2 11 de volgende twee "beruchte" stage-2 predikaten naar hun u welbekende stage-1 equivalent. Niet langer nodig dus om dat zelf te doen; dit kan soms de leesbaarheid van de query ten goede komen!

`YEAR(datum_kolom) = :hv` => BETWEEN-predikaat met 1 Jan & 31 Dec

`SUBSTR(kol, 1, n)` = :hv => `kol LIKE :hv || '%'`

De optimizer gaat trouwens verder dan dit (of dan wat we zelf syntactisch haalbaar kunnen realiseren). Hoe zou je b.v. het volgende stage-2 predikaat herschrijven naar een stage-1 predikaat?

```
WHERE   SUBSTR(k, 1, 3) <= 'CFA' ?
```

Bedenk dat kolom `k` b.v. lengte 10 kan hebben, en dat dus b.v. ook de kolomwaarde '`CFAZZZZZZZ`' moet weerhouden worden, maar niet '`CFBaaa`'. Maar wel b.v. ook '`CFA000`' en '`CFAzzz`' of één of ander non-printable character op positie 4 of verderop.

Dit is wat de query transformation-fase van de optimizer ervan maakt (althans wanneer de bewuste kolom een EBCDIC-encodering heeft, en wanneer de kolom als datatype CHAR(10) heeft):

```
WHERE   k <=  x'C3C6C1FFFFFFFFFFFFFF'
```

Dit wil je niet zelf moeten schrijven! Niet alleen is het zeer onleesbaar: bovendien zou het fout zijn wanneer men de lengte van kolom k aanpast, of wanneer de kolom geen EBCDIC-encoding heeft.

Drie explain-tabellen

Hoe ontdekken we wanneer een query transformation heeft plaats gevonden, en wat de wijziging precies is? Vraag het EXPLAIN !

Het meest volledige (maar voor de menselijke lezer minst nuttige) antwoord vinden we in DSN_QUERY_TABLE, in kolom NODE_DATA: deze XML-kolom bevat de *parsed queries*: zowel de originele als de getransformeerde.

Een leesbaarder antwoord vinden we in de twee tabellen DSN_FILTER_TABLE en DSN_PREDICAT_TABLE (zie hieronder).

Zoals altijd vertelt de optimizer ons alleen iets als we 'm dat ook expliciet vragen. Met name door EXPLAIN uit te voeren. Wanneer we ervoor gezorgd hebben dat ons current schema (naast PLAN_TABLE) de eerder genoemde tabellen bevat, dan worden die gevuld met alle gewenste informatie omtrent query transformations.

Eén voorbeeld zegt meer dan een hoop theoretische uitleg. Laten we b.v. de volgende catalog-query eens proberen, die alle indexen laat zien met namen beginnend met '`TUI`', en ook het aantal gebruikte storage groups telt per index:

```
SELECT i.name, i.padded, i.uniquerule, i.unique_count, p.nr_stogroups
FROM   sysibm.sysindexes i  JOIN
       (SELECT ixname, ixcreator, COUNT(DISTINCT storname) AS nr_stogroups
        FROM   sysibm.sysindexpart  GROUP BY ixname, ixcreator) p
       ON i.name = p.ixname AND i.creator = p.ixcreator
WHERE  SUBSTR(i.name,1,3) = 'TUI'
```

De verwachte chronologie van deze query, voor een naïeve Db2-gebruiker: (a) doorloop de hele tabel `sysindexpart` om partities per index te tellen; (b) filter `sysindexes` op basis van kolom name (maar met een stage-2 predikaat); (c) join de gefilterde `sysindexes` met het gematerialiseerde resultaat van de `GROUP BY` op `sysindexpart`.

Dit is uiteraard allesbehalve efficiënt, zeker wat stap (a) betreft.
EXPLAIN kan ons hopelijk vertellen dat de query herschreven is naar:

```
SELECT i.name, i.padded, i.uniquerule, i.unique_count, p.nr_stogroups
FROM   sysibm.sysindexes i  JOIN
       (SELECT ixname, ixcreator, COUNT(DISTINCT storname) AS nr_stogroups
        FROM   sysibm.sysindexpart
        WHERE  ixname LIKE 'TUI%'  GROUP BY ixname, ixcreator) p
       ON i.name = p.ixname AND i.creator = p.ixcreator
WHERE  i.name LIKE 'TUI%'
```

Hiervoor bekijken we eerst de tabel PLAN_TABLE: een extra filtering in stap (a), queryblock 2 dus, kan blijken uit `ACCESSTYPE='I'` op `sysindexpart` i.p.v. `ACCESSTYPE='R'` (table scan). Ook het vervangen van een stage-2 predikaat op tabel `sysindexes` door de LIKE-variant kan blijken uit ACCESSTYPE='I' voor queryblock 1.

We observeren echter nog steeds een table scan in query block 2, en gelukkig wel het gebruik van index `DSNDXX01` op `sysindexes`:

QB	Meth	Table	AccType	Index	MatchCol	IxOnly	Sort	QBType
1	0		R					select
1	1	SYSINDEXES	I	DSNDXX01	2			select
2	0	SYSINDEXPART	R					tablex
2	3						U G	tablex

Een table scan hoeft niet te betekenen dat er geen “early filtering” gebeurt op `sysindexpart`: misschien is de filter factor te hoog om een index te verantwoorden. Of misschien is er helemaal geen index!

Laten we eens kijken in EXPLAIN-tabel DSN_FILTER_TABLE, die één rij bevat per predikaat:

QB	OrderNo	PredNo	Stage
1	3	2	STAGE1
1	2	3	MATCHING
1	1	4	MATCHING
2	1	5	STAGE1

Het goede nieuws is inderdaad dat geen enkel predikaat als STAGE2 staat aangegeven. We zien echter niet zo goed, over welke predikaten we precies praten (volgnummers 2,3,4,5). Daarvoor bekijken we EXPLAIN-tabel DSN_PREDICAT_TABLE:

QB	PredNo	Type	LeftHSide	RightHSide	FilterFac	Join	Text
1	1	AND			1.00000	N	((I.NAME BETWEEN ...
1	2	BETWEEN	NAME	VALUE	0.04000	N	I.NAME BETWEEN ...
1	3	EQUAL	NAME	IXNAME	0.01000	Y	I.NAME=P.IXNAME
1	4	EQUAL	CREATOR	IXCREATOR	0.01000	Y	I.CREATOR=P.IXCREATOR
2	5	BETWEEN	IXNAME	VALUE	0.10000	N	I.CREATOR=P.IXCREATOR

Hier verschijnt ook predikaat nummer 1; maar dat is gewoon een hiërarchie-aanduiding (“AND”) voor de combinatie van nummers 2, 3 en 4. Nummer 4 hebben we trouwens niet zelf geschreven: die is het gevolg van enerzijds een “transitive closure” in queryblock 1 (want p.ixname = i.name) en anderzijds een predicate pushdown naar queryblock 2!

Bemerk de kolom FILTER_FACTOR: deze getallen geven de door de optimizer geschatte filtering van dit predikaat.

Virtuele indexen

Bij een rebind (of bij dynamische SQL) kan de optimizer soms “plots” een totaal ander access path kiezen, al dan niet beter dan het vorige. Dit kan het gevolg zijn van gewijzigde statistieken, van een nieuwe index, of van een aanpassing van de cluster sequence van een tabel, of van extra kolommen die aan een index werden toegevoegd.

Willen we niet voor verrassingen komen te staan bij geplande aanpassingen aan indexen, dan zouden we de impact van dergelijke DDL op bestaande access paths eerst moeten uittesten.

Virtuele indexen zijn het perfecte tool hiervoor: deze niet-bestaande objecten, ingevoerd in Db2 10, zijn uitsluidend zichtbaar voor EXPLAIN en hebben dus geen effect op rebinds. In plaats van met CREATE INDEX een rij toe te voegen aan SYSIBM.SYSINDEXES, voeg je namelijk manueel een rij toe aan de explain-tabel

DSN_VIRTUAL_INDEXES

Vanaf dat moment zal EXPLAIN doen alsof deze index bestaat, en kan dus het effect van dit nieuwe object op bestaande access paths nagegaan worden.

De eenvoudigste manier om van een bestaand (static SQL) package het nieuwe access path te zien zonder het package te wijzigen:

REBIND PACKAGE (naam) EXPLAIN(ONLY)

Wil je een index-wijziging (ALTER INDEX) simuleren? Ook daar kan een virtuele index uitkomst bieden. Of eigenlijk twee virtuele indexen: want m.b.v. een "`D`" in de kolom `mode` geef je aan dat je de index beschreven in die rij van `dsn_virtual_indexes` wil verwijderen.

Sinds Db2 11 zijn er enkele nieuwe kolommen toegevoegd aan `dsn_virtual_indexes`: de statistische kolom `datarepeatfactorf`, de kolom `unique_count` (voor indexen met "include"-kolommen), de kolom `sparse` (voor null-suppressed indexen), en `key_target_count` en `ix_extension` (voor XML-indexen en index on expression).

Een voorbeeld: stel dat we de performance van de query

```
SELECT * FROM sysibm.systables WHERE name LIKE 'PERS%'
```

willen verbeteren, door een nieuwe index te plaatsen op de catalog-tabel `sysibm.systables`. De kolom `name` komt weliswaar voor als tweede kolom van de primary key index van deze tabel, maar niet als eerste kolom, zodat het predikaat `name LIKE ?` niet indexeerbaar is.

Vooraleer we een dergelijke index effectief creëren, maken we een virtuele index aan:

```
INSERT INTO dsn_virtual_indexes (tbcreator, tbname, ixcreator, ixname)
  VALUES ('SYSIBM', 'SYSTABLES', 'SYSIBM', 'my_IXNAME') ;
UPDATE dsn_virtual_indexes
  SET enable='Y', mode='C', colcount=1, clustering='N', uniquerule='D',
  pgsize=4, padded='N', indextype='D', colno1=1, ordering1='A',
  nleaf=423, nlevels=3, firstkeycardf=1000, clusterratiof=0.9
  WHERE ixname='my_IXNAME';
```

Bemerk dat we de informatie die anders in SYSIBM.SYSKEYS zou terecht komen, hier via colno1 & ordering1 (en eventueel colno2 enz. voor indexen met meerdere kolommen) moeten aangeven. Omdat `name` de eerste kolom is van `sysibm.systables`, geven we

`colno1` de waarde 1. Waarden voor de statistische kolommen moeten uiteraard zorgvuldig ingevuld worden (of gebruik desnoods de waarde -1).

JSON (vs XML) - en Db2 (I)

Sandy Schillebeeckx (ABIS), Kris Van Thillo (ABIS)

In een aantal vorige edities stonden we reeds uitgebreid stil bij het gebruik van XML in Db2. En ook op onze website zijn een aantal resources beschikbaar die de mogelijkheden uiteenzetten voor het verwerken van XML-documenten in Db2.

In dit -en een volgend- artikel willen we even stil staan bij JSON; en de manier waarop JSON in een Db2-omgeving kan worden gebruikt. In dit eerste artikel introduceren we JSON, en schetsen we kort even de verschillen met XML.

Wat is JSON?

JSON staat voor JavaScript Object Notation. Het is een ECMA internationale standaard sinds 2013, maar bestaat al sinds begin 21ste eeuw. JSON is een open-standaard file formaat dat "leesbare" tekst gebruikt om data objecten, bestaande uit name-value paren en array data types, te transfereren. Het is een veelgebruikt data formaat voor asynchrone client/server communicatie, en vervangt daar XML in AJAX-achtige systemen.

JSON is een taal-onafhankelijk data formaat, dat afgeleid is van JavaScript. Tegenwoordig bevatten vele programmeertalen en databases code om JSON te parsen en te genereren.

JSON syntax: de basis

JSON - voorbeeld

```
{"CourseParticipants": {
    "Participant": [
      {"firstName": "Maria", "lastName": "Meuris"},
      {"firstName": "Bert",  "lastName": "Mak"},
      {"firstName": "Peter", "lastName": "Buenk"}
    ]
  }
}
```

Een JSON *object* is een JavaScript object literal. Het is een lijst van name-value paren, omsloten door accolades (`{ }`). De name-value paren worden van elkaar gescheiden door komma's (`,`); tussen de naam en de waarde staat telkens een dubbelpunt (`:`). Zowel de naam als de waarde moeten tussen dubbele aanhalingstekens (`" "`) staan. Accolades worden ook gebruikt om verdere nesting van de structuur aan te geven.

Een JSON *array* wordt voorgesteld met rechte haken ([ ]). Binnenin bevinden zich de elementen, wat zowel objecten, arrays als scalaire waarden kunnen zijn. De volgorde van de elementen in een array is significant.

De datatypes van de scalaire waarden zijn beperkt tot tekst, numeriek en boolean. Een lege waarde wordt voorgesteld door `null`.

JSON versus XML

Onderstaande is de vertaling van ons JSON-voorbeeld naar XML:

```
<CourseParticipants>
    <Participant>
        <firstName>Maria</firstName>
        <lastName>Meuris</lastName>
    </Participant>
    <Participant>
        <firstName>Bert</firstName>
        <lastName>Mak</lastName>
    </Participant>
    <Participant>
        <firstName>Peter</firstName>
        <lastName>Buenk</lastName>
    </Participant>
</CourseParticipants>
```

Als we beide –eenvoudige– voorbeelden kort bestuderen, blijken duidelijk een aantal verschillen. In wat volgt halen we ze kort even aan, en voegen we een aantal eigenschappen toe die niet onmiddellijk uit bovenstaande voorbeelden blijken.

- ***Leesbaarheid, eenvoud***.

Afhankelijk van een aantal factoren zal eenieder wel één van beide leesbaarder of eenvoudiger vinden. Door het ontbreken van sluittags zoals in XML, en het gebruik van de array notatie is JSON in ieder geval heel wat compacter. Sowieso biedt een JSON-formaat ook gewoon minder mogelijkheden (o.a. geen attributen en commentaar) dan een XML-formaat – en is dus ook *de facto* eenvoudiger. Beide structuren zijn tot op zekere hoogte zelfbeschrijvend. Minimale data is aanwezig om onafhankelijke tools de mogelijkheid te bieden met de inhoud van de documenten aan de slag te gaan. Ook hier is JSON typisch beperkter in mogelijkheden. JSON is ook niet uitbreidbaar, terwijl XML wel die mogelijkheid biedt.

- ***Structuur***.

XML-documenten zijn strikt hiërarchisch van aard. Elk XML document moet verplicht een root-element hebben. Dit hoeft in een JSON-document echter niet zo te zijn. Het volgende object: {“naam”:“value”, “naam2”:“value2”}, is net zo goed geldig in JSON. In de praktijk zullen vele JSON-objecten echter ook een hiërarchische vorm aannemen, gezien ze als vervanging van XML gebruikt worden.

- ***Performance***.

Het processen van JSON data is vaak sneller dan dat van een gelijkwaardig XML document. Ten eerste doordat JSON beknopter is, maar ook omdat JSON in bepaalde programmeertalen als native formaat beschouwd wordt, terwijl XML door een parser behandeld moet worden.

- *Gebruik.*

Gezien al het voorgaande kunnen we besluiten dat JSON te verkiezen is boven XML als het gaat om het snel transfereren van (lichte) data die gebruikt wordt in een JavaScript/OO context. Indien het doel het delen van documenten is, waar je controle wilt leggen op de structuren en datatypes, is XML de betere optie.

JSON en Db2

Het spreekt voor zich dat Db2 ook JSON-stijl documenten kan verwerken. In een volgend nummer van Exploring Db2 staan we zeer concreet stil bij de mogelijkheden die in deze worden geboden..

DOSSIER 11

Het nieuwe extended log adres formaat (RBA/LRSN)

Alle (schrijf)acties in een Db2-omgeving worden gelogd. Log-records worden weggeschreven naar logbestanden die (virtueel) één lange keten vormen. Log-records kunnen enkel teruggevonden worden als hun fysieke lokatie, hun byte-adres in die bestanden dus, gekend is. Dit is b.v. nodig om bij verlies van data (restart/recovery) de uitgevoerde acties opnieuw op de juiste wijze, in de juiste volgorde, en op een consistente manier uit te voeren. Zo'n adres is dus een *sequentie* of volgnummer.

Dit fysiek byte-adres (in de log-keten sinds de installatie van het Db2-subsysteem) wordt enkel in een standaard (no data sharing) omgeving gebruikt, en wordt RBA genoemd: *relative byte address*. Hoe 'hoger' het RBA, hoe recenter de 'gelogde actie'. In een data sharing omgeving -sinds Db2 versie 4- wordt een LRSN-adressering gebruikt om een coördinatie overheen data sharing group members te bewerkstelligen. Dit adres is essentieel een timestamp. Synchronisatie overheen members is uiteraard de verantwoordelijkheid van de sysplex timer.

Beide waarden -RBA en LRSN- worden sinds jaar en dag opgeslagen als 6 bytes - in de logstructuren, maar ook in de bootstrap dataset, in de fysieke object-pagina's (data/index), ... met als doel: het snel terugvinden van log entries.

Ten gevolge van een hele reeks -ook externe- factoren is het opslaan van een RBA/LRSN als 6 bytes niet langer realistisch. Sneller DASD, meer multi-way processors, toename van het aantal data sharing members per sharing group, ... en dus een toename van logactiviteit, zorgt er voor dat zowel RBA als LRSN een overflowpunt (kunnen) bereiken. En dan is het einde verhaal voor het Db2-subsysteem!

Het RBA van een Db2 subsysteem/data sharing member kan weliswaar worden gereset -de reset-log-RBA procedure zeg maar. Niet evident, vooral omdat hierhoor historiek verloren gaat. Het LRSN kan niet worden gereset (want deze waarde heeft een absolute tijdsinterpretatie); het jaar 2042 zou in deze wel eens een cruciaal jaar kunnen zijn... Eigenaardig trouwens: zelfs als Db2 niet actief is tikt de klok (letterlijk!) en komt het LRSN overflow-moment dichterbij!

In Db2 versie 11 heeft IBM daarom beslist, zowel het RBA-formaat als het LRSN-formaat uit te breiden naar 10 bytes - het zogenaamde extended log address formaat:

- het RBA krijgt 4 extra bytes links (*high order*) van de bestaande waarde; daardoor neemt het maximaal aantal adresseerbare log-entries toe van 256 TB naar 1 YB;

- het LRSN krijgt 1 extra byte links (*high order*) en 3 bytes rechts (*low order*) van de bestaande waarde; we kopen aldus ongeveer 30.000 jaar, en krijgen een veel fijnere log-granulariteit (met als gevolg o.a. minder log processor *spinning* of 'wacht'tijd die men met 6 bytes nodig heeft om duplicate LRSNs te vermijden).

Db2 versie 11 hanteert intern automatisch het extended adresformaat; de 6 bytes variant (basisformaat) wordt altijd geconverteerd van/naar het extended formaat voor processing. Om het extended formaat ook in logstructuren, cataloog (i.h.b. `SYSCOPY`) en datapagina's te doen verschijnen, moet enerzijds de bootstrap dataset worden geconverteerd; en anderzijds de fysieke structuur van de Db2 tablespaces (via het REORG utility) worden aangepast. Deze twee stappen kunnen los van elkaar gebeuren, en kunnen indien gewenst nog enkele jaren uitgesteld worden.

Kris Van Thillo

CURSUSPLANNING, OKT - DEC 2017

SQL & relationele databases: basiskennis	850 EUR	30.10(W), 13.11(L)
Db2 for z/OS basiscursus	1425 EUR	18.10(L)
Db2 for LUW basiscursus	1425 EUR	18.10(L)
SQL workshop	900 EUR	26.10(L), 27.11(W), 11.12(L)
SQL voor gevorderden	500 EUR	17.11(L), 20.12(W)
SQL voor BI rapportering & analyse	950 EUR	15.11(L), 18.12(W))
SQL PL database programmeren	1000 EUR	23.11(L)
Db2 triggers, stored procedures, en User-Defined Functions	500 EUR	op aanvraag
Db2 for z/OS: programmeren voor gevorderden	1000 EUR	09.11(L)
Db2 for z/OS: SQL performance	1500 EUR	13.12(L)
XML in Db2		op aanvraag
Db2 for z/OS: database administratie	2100 EUR	10.10(W), 18.12(L)
Db2 for z/OS: installation & migration		op aanvraag
Db2 for z/OS: data recovery	1080 EUR	10.10(UK), 05.12(UK)
Db2 for z/OS: using RACF		op aanvraag
Db2 for z/OS: monitoring & tuning systems' performance		op aanvraag
Db2 for LUW DBA - Kernvaardigheden	2000 EUR	19.12(W)
JDBC	500 EUR	16.10(L)
Db2 10 for LUW: new features		op aanvraag
Db2 11 for z/OS: changes & new features	525 EUR	16.10(L)
Db2 12 for z/OS: changes & new features		op aanvraag
Data warehouse concepten	500 EUR	12.10(W)
Dimensionaal modelleren	2120 EUR	12.12 (Hilversum)
Big Data concepten	500 EUR	30.10(L), 13.11(W)
Big Data in de praktijk: text analytics	475 EUR	22.11(L)
Regular expressions	275 EUR	30.11(L, avondsessie)

Plaats: (L) = Leuven, (W) = Woerden, (UK) = High Wycombe (bij Londen);
alle cursussen ook beschikbaar op aanvraag en/of op uw lokatie;
Voor details en andere cursussen, zie `http://www.abis.be/html/nlall.html`
Pour détails et d'autres cours, voir `http://www.abis.be/html/frall.html`
For details and other courses, see `http://www.abis.be/html/enall.html`