Partitioneren voor ontwikkelaars

...levert significant betere prestaties op

Jaren geleden speelde ik wat met het partitioneren van tabellen. Ik had er destijds al eens een artikeltje aangeweid. Laatst had ik er echter een discussie over met een collega ontwikkelaar op het project. Partitioning is dus nog steeds actueel, vandaar dat ik e.e.a. maar weer eens heb opgepoetst.

Bij partitioneren denk je vrijwel meteen aan Datawarehouses of anderssoortige VLDB (Very Large Databases). Om verscheidene redenen die ik hierna aan zal geven is partitioneren voor een HTEKDB (huis-tuin-en-keuken-database) ook erg handig. Maar partitioneren is iets voor DBA's, toch? Als je een DBA bent zul je mogelijk bij het lezen van dit artikel de nodige gapen moeten onderdrukken. Tenzij je nog nooit iets met partitionering hebt gedaan. Als je ontwikkelaar bent dan raad ik je aan dit artikel toch eens door te nemen. Want hoewel partitionering gezien wordt als een 'DBA-ding' is het nou juist de ontwikkelaar die meestal het fysiek datamodel ontwikkelt. En het is de ontwerper dan wel ontwikkelaar die het logisch datamodel ontwerpt en bedenkt hoe het gebruikt gaat worden in de applicatie. Over het algemeen zijn zij degenen die contact hebben met de eindgebruikers of de business-analisten die een inschatting kunnen maken van de hoeveelheid data. Een tabel wordt gepartitioneerd bij creatie en het is vaak lastig of vrijwel onmogelijk om dat achteraf te doen. Vooral op een live-systeem met hele grote tabellen.

Een goede ontwerper/ontwikkelaar denkt daarom bij het ERD al na over het toekomstig gedrag van de applicatie. Maar ook bij het daadwerkelijk ontwerpen van het fysiek datamodel is het zinvol om hier nog eens bij stil te staan.

Wat is partitioneren?

Partitioneren is het opdelen van tabellen in partities. Een open deur? Zie het als een extra logische abstractie laag tussen de tabellen en tablespaces. Een tabel bestaat uit segmenten (voor het gemak ook partities genoemd) en die segmenten bevinden zich in de tablespace. Voor de applicatie maakt het functioneel eigenlijk weinig verschil: die 'ziet' gewoon een 'normale' tabel. Maar vanaf Oracle 8i zijn die segmenten afzonderlijk toegankelijk gemaakt. Met partitionering kan de database op basis van een bepaalde sleutel (partition-key) of voorwaarde de rijen verdelen over die segmenten. Daarvoor kunnen bijvoorbeeld datum-ranges worden gebruikt. Bijvoorbeeld de creatie-datum van een order. Orders die in een bepaalde periode zijn aangemaakt komen dan in de partitie van die bepaalde periode. Of interface records met een bepaalde status komen in de partitie die behoort bij betreffende status.

Veel mogelijkheden

Als de optimizer in het geval van een Select, Update of Delete de partion-key (de kolom(men) waarop is/zijn gepartitioneerd) in de where clause vind, dan neemt hij dat mee in de toegangspad bepaling. De optimizer kan dan concluderen dat de betreffende rijen zich allemaal in hetzelfde segment bevinden. De optimizer zal wellicht wat sneller voor full table scans kiezen, maar dan binnen het betreffende segment (partitie) en dat is dan niet erg. Eigenlijk een full-segment-scan dus. Met een beetje geluk moeten juist ook alle rijen binnen die partitie behandeld worden (zie onderstaande voorbeelden). Voor hele grote tabellen is dat dus een uitkomst als toch maar een deel van de data frequent wordt benaderd.
Eigenlijk ziet de optimizer de tabel dus als het ware als een verzameling tabellen die met 'union all' aan elkaar gekoppeld zijn. Zo is het ooit ook in Oracle 7 geïmplementeerd.
Op het gebied van beheer zijn er een schat aan extra mogelijkheden. Veel bewerkingen waar normaal zware (PL/)SQL-code voor nodig is, zijn dan relatief simpele ddl- bewerkingen die dan heel snel zijn.

De redenen voor partitioneren zijn het beste uit te drukken in een aantal situatie-schetsen. Ik noem er een paar:

1. Verwerken van data in een interfacetabel
De interface mogelijkheden van een standaard applicatie zijn vaak beperkt of lastig. Of het implementatie team houdt dat

Partitioneren is handig voor hele grote-, maar ook voor huis-tuin-en-keukendatabases.

liever buiten de deur. Vaak kan er dan wel een tabel gevuld en/of uitgelezen worden. Zo'n interface tabel bevat naast de data-kolommen, vaak ook een id-kolom en een status kolom. De status staat bij nieuw opgevoerde rijen dan op 'Initial' of 'New' of iets dergelijks. Wanneer een rij verwerkt is dan wordt de status van die rij op 'Processed' of 'Complete' gezet. Men is vaak voorzichtig met het verwijderen van de rijen, omdat bij fouten de weg terug tot de initiator getraceerd moet kunnen worden. Weg is immers weg.
Een interface tabel blijft dan maar groeien en met een beetje 'fout geluk' gaat de optimizer voor full-table scans kiezen, omdat hij onvoldoende histogram informatie heeft.
Wanneer de tabel is gepartitioneerd op status en 'row movement' toestaat (waarover later meer), dan betekent een update van 'Initial' naar 'Processed' een verplaatsing van de rij naar een andere partitie. Met als gevolg dat de partitie met de nieuwe rijen steeds leeg gelezen wordt. Deze bevat dan nooit méér rijen dan de grootst mogelijke batch. Laat de Optimizer dan maar vrolijk 'full table scannen', immers de batch zal toch alle rijen in die partitie moeten verwerken!

2. Orders/Requests situaties
Vaak doorloopt een entiteit in de loop van de bedrijfsprocessen een aantal stadia. Totdat het laatste proces is doorlopen en de entiteit alleen maar voor het archief in het systeem blijft. In het sommige systemen zijn die historie-data mogelijk voor allerlei doeleinden nog steeds van belang. De rijen kunnen dan niet zomaar worden verwijderd. Dan kan zo'n tabel tot enorme proporties groeien, zeker bij een aanwas van 100.000 entries per jaar.
Door ook hier weer op status te partitioneren blijft de tabel voor de belangrijkste statussen beperkt: 90% van de queries betreft immers rijen van een bepaalde, niet-'finale' status. Ook dan zijn full-table-scans binnen de betreffende status partitie niet zo'n probleem. En wanneer toch een zware query moet worden gedaan op rijen met een 'finale' status dan zit deze de 'on-line'-queries minder in de weg.

3. Logging
Bij logging is vooral schoning een probleem. Een logging tabel kan natuurlijk in de loop der tijd dramatisch groeien. Met name als tijdens het ontwikkel traject niet gelegenheid is geweest om aandacht aan schoning te besteden of als de applicatie als een dolle informatie in de logging spuwt. Bij logging kun je denken aan het wegschrijven van informatie over de verwerking en gedrag van de applicatie, maar in het geval van integratie ook aan het loggen van binnenkomende berichten.
Voor de opzet van het schonings-mechanisme is het van belang dat wordt nagedacht over wat de aangroei is, hoe vaak de schoningsjob draait en wat de bewaartermijn is. De job die rijen verwijdert kan een behoorlijke systeembelasting tot gevolg hebben: de transacties mogen niet uit het rollback-segment loen, de job mag niet te lang lopen, enzovoort.
Het is echter mogelijk om naast de logging tabel een gepartitioneerde historie-tabel te zetten. Deze tabel wordt dan gepartitioneerd op logging-datum en per week zorgt een job voor de aanmaak van een nieuwe partitie. Als de logging tabel een platte tabel is dan kan het nog slimmer: aan de historie-tabel wordt een lege-partitie toegevoegd waarvan de boven grens op de volgende dag om 0:00 uur ligt (dus trunc(sysdate) + 1). Dan kan de nieuwe partitie verwisseld worden met het segment van de logging tabel. Hoe dat moet komt hieronder.
Partities waarvan de uiterste houdbaarheidsdatum is verstreken kunnen simpel weg gedropt worden (wel eerst even truncaten). Door deze aanpak kost de hele schoning hoogstens een paar minuten, waarbij het opnieuw opbouwen van indexen nog het langst duurt.

Hoe partitioneeer je een tabel?

Het eerste struikelblok is dit: een tabel kan alleen 'normaal' of 'gepartitioneerd' worden gecreëerd, het is niet mogelijk een normale tabel partitioneren. Met andere woorden, de partition-clausule is voorbestemd voor het 'Create Table' statement.
Overigens is volgens de documentatie een gepartitioneerde tabel wel in een andere (met een andere partitioning-clausule) te transformeren, maar dat heb ik zelf nog niet bij de hand gehad en het voert te ver voor dit artikel.

Als eerste moet je bepalen wat je partitioneringsvoorwaarde is, dus op welke kolom(men) gepartitioneerd moet worden. Voor het gemak houd ik het in het voorbeeld bij een kolom, maar als in de praktijksituatie de verdeling dan nog te grof is, dan is sub-partitioneren op andere kolommen ook mogelijk. Laten we een log tabel maken die is gepartitioneerd op logtijd. Het statement er dan als volgt uit:

```
CREATE TABLE GNL_LOGS
( ID NUMBER(10,0) NOT NULL
, LOG_TIME DATE NOT NULL
, PACKAGE_NAME VARCHAR2(50) NOT NULL
, MODULE_NAME VARCHAR2(50) NOT NULL
, TEXT VARCHAR2(2000) NOT NULL
)
TABLESPACE GNL_DAT_LARGE
PARTITION BY RANGE (LOG_TIME)
( PARTITION GNL_LOGS_2010_07 VALUES LESS THAN (TO_DATE('31-07-2010',
'DD-MM-YYYY')) TABLESPACE GNL_DAT_LARGE
, PARTITION GNL_LOGS_MAXVALUE VALUES LESS THAN (MAXVALUE)
TABLESPACE GNL_DAT_LARGE
)
```

Voor elke partitie wordt een bovengrens van de kolomwaarden die bij die partitie horen aangegeven . Het is daarom van belang om bij het applicatie ontwerp handig met de keuze van deze domeinwaarden om te gaan.
De volgorde is daar bij ook belangrijk: er mogen alleen partities worden toegevoegd met een bovengrens die ligt boven de 'maximale' bovengrens van al toegevoegde partities. In bovengenoemd voorbeeld is er bijvoorbeeld een partitie voor datums eerder dan 31-07-2010', maar ook datums beneden 'Maxvalue'. Maxvalue is een reserved word dat precies betekent wat het zegt: het biedt de mogelijkheid een 'overloop partitie' aan te maken.

Wellicht zijn er indexen nodig. Bijvoorbeeld op de id en log_time kolommen:

```
PROMPT Creating Index 'GNL_LOGS_PK'
CREATE INDEX GNL_LOGS_PK ON GNL_LOGS(ID)
PCTFREE 10
TABLESPACE GNL_IDX_LARGE
LOCAL
( PARTITION GNL_LOGS_2010_07 PCTFREE 10 TABLESPACE GNL_IDX_LARGE
, PARTITION GNL_LOGS_MAXVALUE PCTFREE 10 TABLESPACE GNL_IDX_LARGE
)
/
PROMPT Creating Index 'GNL_LOGS_IDX1'
CREATE INDEX GNL_LOGS_IDX1 ON GNL_LOGS (LOG_TIME)
PCTFREE 10
TABLESPACE GNL_IDX_LARGE
LOCAL
( PARTITION GNL_LOGS_2010_07 PCTFREE 10 TABLESPACE GNL_IDX_LARGE
, PARTITION GNL_LOGS_MAXVALUE PCTFREE 10 TABLESPACE GNL_IDX_LARGE
)
/
```

De eerste index suggereert een primary key. De index is echter local. Dat betekent dat hij is gekoppeld aan de partities van de tabel. Dit wordt in een Primary Key constraint niet toegestaan. Een primary/unique key constraint moet niet-gepartitioneerd zijn.
Hoe onderhoud je een gepartitioneerde tabel?

Toevoegen van een partitie

Voor het gewoon toevoegen van een extra partitie geldt het volgende statement.

```
ALTER TABLE GNL_LOGS
 ADD PARTITION GNL_LOGS_2010_08 VALUES LESS THAN (TO_DATE('31-08-2010',
'DD-MM-YYYY')) TABLESPACE GNL_DAT_LARGE
/
```

Het statement spreekt denk ik voor zich. Er kunnen echter alleen partities worden toegevoegd met een high-value die groter is dan reeds bestaande 'high-values'. In ons voorbeeld zal dit dus niet werken omdat er geen hogere waarde dan 'MAXVALUE' is. Dan zal de partitie (bijvoorbeeld de 'MAXVALUE'-partitie) moeten worden gesplitst.

Splits een partitie

Het volgende statement splitst een partitie op een opgegeven punt:

```
ALTER TABLE GNL_LOGS SPLIT PARTITION GNL_LOGS_2010_07
AT (TO_DATE('30-06-2010', 'DD-MM-YYYY'))
INTO ( PARTITION GNL_LOGS_2010_06
     , PARTITION GNL_LOGS_2010_07)
```

Het statement splitst de partitie gnl_logs_2010_07 in twee partities met als grens datum '30-06-2010'. Merk op dat de 'into' partities in volgorde van 'highvalue' waarden staan, de eerste partitie heeft dus als 'less than'-waarde '30-06-2010' en de tweede de 'oude' highvalue waarde '31-07-2010'.
Een voordeel van het splitsen van een partitie is dat je geen storage/tablespace-clausule hoeft mee tegeven: de nieuwe partitie komt gewoon in dezelfde tablespace als de 'oude' partitie. Eventuele locale indexen worden mee gesplitst.

Truncate een partitie

Van een tabel kan ook een afzonderlijke partitie ge-'truncate' worden:

```
ALTER TABLE GNL_LOGS TRUNCATE PARTITION GNL_LOGS_2010_06;
```

Dat kan handig zijn als de partitie moet worden gedropt. Het

droppen van een hele grote partitie kan (net als bij een tabel) nogal wat tijd in beslag nemen. Een truncate kan dan helpen.
Drop een partitie
Erg makkelijk bij het schonen van logs: die partitie waarvan de high-value ouder is dan een bepaalde bewaar-periode wordt gedropt:

```
ALTER TABLE GNL_LOGS DROP PARTITION GNL_LOGS_2010_06;
```

Partities uitwisselen

Het splitsen van een erg grote partitie kan ook nog al wat tijd in beslag nemen. Maar het is mogelijk om het segment van een gewone lege tabel met dezelfde layout (bijv. create table bck_logs as select * from gnl_logs where 1=0;) te verwisselen met de te splitsen partitie. De partitie die moet worden gesplitst is dan leeg en alle data zit dan in de dummy tabel. De lege partitie wordt dan gesplitst op een waarde hoger dan de hoogste waarde van de partitie-kolom die nu in de dummy-tabel zit. En naderhand kan de nieuwe partitie weer worden omgewisseld met de dummy tabel. Dan is het splitsen in bijna geen tijd gebeurd. Deze aanpak is overigens ook goed bruikbaar bij het omzetten van een bestaande normale tabel in een gepartitioneerde.

Het omwissel-statement luidt:

```
ALTER TABLE GNL_LOGS EXCHANGE PARTITION GNL_LOGS_MAXVALUE WITH TABLE
BCK_LOGS;
```

Indexen opnieuw opbouwen

Bij bijna alle onderhoudsoperaties op gepartitioneerde tabellen met indexen, zorgt er voor dat de indexen 'UNUSABLE' worden. Controleer de status van de indexen maar eens met 'SELECT INDEX_NAME, PARTITION_NAME, STATUS FROM USER_IND_PARTITIONS;'.
De indexen zullen daarom opnieuw opbouwd moeten worden. Gaat het om lokale indexen dan kunnen die alleen per partitie worden ge-'rebuild' (maar dat hoeft dan dus ook niet voor de hele tabel).:

```
ALTER INDEX GNL_LOGS_PK REBUILD PARTITION GNL_LOGS_MAXVALUE;
```

Toestaan van rij-verplaatsing

Standaard is het updaten van de partitonerings-kolom niet toegestaan. Immers het gevolg kan zijn dat de rij niet meer aan de partitionerings-voorwaarde voldoet: de rij hoort niet meer in die partitie thuis. Toch is dat in het geval van een status-partitionering nu net de bedoeling: de status moet gewijzigd kunnen worden en dan moet de rij ook van de ene naar de andere partitie worden verplaatst. Daarvoor zal 'rij-verplaatsing' moeten worden toegestaan:

```
ALTER TABLE GNL_LOGS ENABLE ROW MOVEMENT;
```

Merk hierbij wel op dat een wijziging van de status dan eigenlijk een delete in combinatie met een insert in één transactie is. Dit betekent dus wat performance verlies bij updates.
Selecteren uit een partitie
Wat zit er nu in een partitie? Het is uiteraard mogelijk een select te doen op de tabel en dan zelf de partitioneringsvoorwaarde in de where-clause opgeven. Maar het volgende werkt echter ook:

```
SELECT * FROM GNL_LOGS PARTITION ( GNL_LOGS_2010_07);
```

Hiermee wordt dan een full-segment-scan gedaan. Je kan daar dan uiteraard op fijn-filteren met extra query-voorwaarden. Maar je beperkt de zoek opdracht meteen al tot een bepaald segment. En dat scheelt een stuk in de performance.

Conclusie

Met een beetje moeite zijn de prestaties van een applicatie significant op te krikken door slim tabellen te partitioneren. In elk geval is het gedrag beter voorspelbaar te maken. Veel full table scans zijn dan ineens niet erg meer omdat ze toch op alle rijen in die ene partitie betrekking hebben. Daarnaast zijn zware onderhouds-operaties hanteerbaarder geworden. Bedenk daarom bij een nieuwe applicatie op voorhand hoe bepaalde grote/centrale tabellen waarschijnlijk benaderd gaan worden. Kijk bij bestaande slecht of minder goed presterende applicaties of het opdelen van tabellen mogelijk kan helpen.
Als al bekend is dat veel zware queries maar een beperkte set van rijen raakt dan kan het veel schelen. Belangrijkste conclusie is wat mij betreft, dat veel nieuwe Oracle functionaliteit voor grote dataware-house systemen wordt bedacht en ontwikkeld, maar ook voor applicatie-ontwikkelaars handig zijn. Partioning is in de op een volgende versies na 8.0 nog verder verbeterd. Zo is in 11gR2 de SQL Advisor uitgebreid met een Partitioning Advisor. En kun je ook REF-partioneren: dit houdt in dat een child-table de partitionering overerft op basis van primary-foreign-key relaties. Je kunt dan partitioneren op kolommen in de parent table die niet over worden genomen in de child table. Maar vooral stoer vind ik ook de mogelijkheid om te partitioneren op de nieuwe Virtuele Kolommen. Deze zijn immers niet opgeslagen bij de tabel, de waarden bestaan alleen als metadata. Maar je kunt de partitionering er nu wel op baseren.

Martien van den Akker is Technisch Architect Integratie bij Darwin IT