Merge und andere schnelle Statements

Dr. Andrea Kennel
InfoPunkt Kennel GmbH
CH-8600 Dübendorf

Schlüsselworte:

Merge, Multitable Insert, Analytische Funktionen, hilfreiche Befehle für DWH und ETL

Einleitung

Mit Oracle geht es häufig darum grosse Datenmengen zu speichern und zu verarbeiten. Dazu bietet Oracle diverse spezifische Befehle an, die grosse Datenmengen effizient bearbeiten. Das Problem ist nur, wer diese speziellen Befehle nicht kennt, findet sie in der umfassenden Dokumentation kam und setzt sie daher auch nicht ein. Schade, denn Befehel wie Merge sind nicht nur effizient, sondern auch relativ einfach und elegant.
In diesem Vortrag werden typische Aufgabenstellungen zuerst mit normalem SQL Befehlen gelöst. Dann wird gezeigt wie sie mit speziellen Oracle Befehlen schneller und eleganter gelöst werden können.
Die Konkreten Beispiele sollen helfen diese Befehle besser zu verstehen und auch selber bei ähnlichen Aufgabenstellungen anwenden zu können.

Analytische Funktionen

Aufgabenstellungen Gültigkeitsbereich

In der Tabelle COSTS sind die Preise der Produkte je Verkaufskanal und je Werbeweg (Promotion) gespeichert. Als Datum wird immer angegeben, ab wann diese Preise gelten. Für unser Data Warehouse wollen wir aber die Gültigkeit durch ein VALID_FROM und VALID_TO angeben. Da wir mit ganzen Tagen arbeiten soll das VALID_TO immer ein Tag vor dem nächsten VALID_FROM liegen.
Mit einem Select sollen die Daten vorbereitet werden, damit die neuen Tabelle mit einem CREATE TABLE AS SELECT erstellt werden kann.

Lösung mit einfachem SQL

Diese Aufgabe ist eher knifflig. Zu jedem Record muss ich alle späteren Records finden und von diesen wiederum das kleinste Datum lesen.

Hier der Code:

```
SELECT a.prod_id, a.promo_id, a.channel_id, a.unit_cost, a.unit_price,
       a.time_id valid_from, min(b.time_id) -1 valid_to
FROM costs a, costs b
WHERE a.prod_id = b.prod_id
  AND a.promo_id = b.promo_id
```

```
  AND a.channel_id = b.channel_id
  AND a.time_id < b.time_id
GROUP BY a.prod_id, a.promo_id, a.channel_id, a.unit_cost, a.unit_price, a.time_id
```

Ausgewählte Daten

```
   PROD_ID   PROMO_ID CHANNEL_ID  UNIT_COST UNIT_PRICE VALID_FR VALID_TO
---------- ---------- ---------- ---------- ---------- -------- --------
       116        999          4       9,93      12,73 20.12.01 21.12.01
       116        999          4       9,93      12,73 22.12.01 23.12.01
       116        999          4       9,93      12,73 24.12.01 24.12.01
       116        999          4       9,93      12,73 25.12.01 27.12.01
       116        999          4       9,93      12,73 28.12.01 28.12.01
       116        999          4       9,93      12,73 29.12.01 30.12.01
       116        999          4       9,93      12,73 31.12.01
```

Lösung mit analytischer Funktion

Bei Analytischen Funktionen (OVER) je Teilbereich von Daten (PARTITION BY) eine Sortierung (ORDER BY) vorgenommen werden. Mit der Funktion LEAD kann dann auch den nachfolgenden Datensatz bezug genommen werden.

Hier der Code

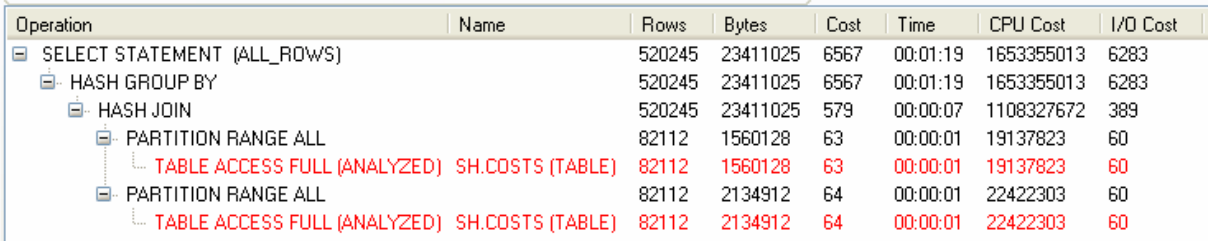
```
SELECT a.prod_id, a.promo_id, a.channel_id, a.unit_cost, a.unit_price,
       a.time_id valid_from,
       LEAD(a.time_id) OVER (PARTITION BY a.prod_id, a.promo_id, a.channel_id
                             ORDER BY a.time_id) -1 valid_to
FROM costs a
```

Ausgewählte Daten

```
   PROD_ID   PROMO_ID CHANNEL_ID  UNIT_COST UNIT_PRICE VALID_FR VALID_TO
---------- ---------- ---------- ---------- ---------- -------- --------
       116        999          4       9,93      12,73 20.12.01 21.12.01
       116        999          4       9,93      12,73 22.12.01 23.12.01
       116        999          4       9,93      12,73 24.12.01 24.12.01
       116        999          4       9,93      12,73 25.12.01 27.12.01
       116        999          4       9,93      12,73 28.12.01 28.12.01
       116        999          4       9,93      12,73 29.12.01 30.12.01
       116        999          4       9,93      12,73 31.12.01
```

Performancevergleich

Executionplan mit Join



Operation	Name	Rows	Bytes	Cost	Time	CPU Cost	I/O Cost
SELECT STATEMENT (ALL_ROWS)		520245	23411025	6567	00:01:19	1653355013	6283
HASH GROUP BY		520245	23411025	6567	00:01:19	1653355013	6283
HASH JOIN		520245	23411025	579	00:00:07	1108327672	389
PARTITION RANGE ALL		82112	1560128	63	00:00:01	19137823	60
TABLE ACCESS FULL (ANALYZED)	SH.COSTS (TABLE)	82112	1560128	63	00:00:01	19137823	60
PARTITION RANGE ALL		82112	2134912	64	00:00:01	22422303	60
TABLE ACCESS FULL (ANALYZED)	SH.COSTS (TABLE)	82112	2134912	64	00:00:01	22422303	60

Laufzeit: 00:00:31.95

Executionplan mit LEAD

Operation	Name	Rows	Bytes	Cost	Time	CPU Cost	I/O Cost
SELECT STATEMENT (ALL_ROWS)		82112	2134912	684	00:00:09	98315824	667
WINDOW SORT		82112	2134912	684	00:00:09	98315824	667
PARTITION RANGE ALL		82112	2134912	64	00:00:01	22422303	60
TABLE ACCESS FULL (ANALYZED)	SH.COSTS (TABLE)	82112	2134912	64	00:00:01	22422303	60

Laufzeit: 00:00:01.48

Mit dem Join muss Oracle zweimal auf dieselben Daten zugreifen und die Tabelle dann noch mit der Joinmethode Hash verknüpfen. Das braucht natürlich Zeit.
Mit Hilfe der analytischen Funktion müssen die Daten nur einmal gelesen werden. Mit der WINDOW Funktion werden die Daten richtig sortiert und der Zugriff auf den Nachfolger kostet quasi nichts mehr.

Weitere Beispiele zu analytischen Funktionen

Wird die Summe mit einerm ORDER BY kombiniert, so werden nur die Datensätze bis zum aktuellen für die Summe berücksichtigt. So erhält man das sogenannte „running total“. Wird aber bei der Summe nur der Teilbereich mit PARTITION angegeben, so wird für diesen Teilbereich die gesamte Summer ermittelt. Ohne Teilbereich bezieht sich die Summe auf alle Datensätze.
Eine weitere interessante Funktion ist RANK mit der eine Rangfolge ermittelt werden kann. Auch hier kann über die PARTITION angegeben werden für welchen Teilbereich die Rangfolge ermittelt werden soll. Das Kriterium für die Rangfolge wird dann im ORDER BY angegeben.

Hier der Code

```
SELECT prod_id, prod_subcategory_id sub_cat, prod_name, prod_list_price price,
       SUM(prod_list_price) OVER (PARTITION BY prod_subcategory_id
                                  ORDER BY prod_list_price) running,
       SUM(prod_list_price) OVER (PARTITION BY prod_subcategory_id) sum_sub_cat,
       SUM(prod_list_price) OVER () sum_total,
       RANK() OVER (ORDER BY prod_list_price) rank
FROM products_2
ORDER BY prod_subcategory_id, prod_list_price
```

```
PROD_ID  SUB_CAT PROD_NAME             PRICE  RUNNING SUM_SUB_CAT  SUM_TOTAL RANK
------- -------- -------------------- ------ -------- ----------- ---------- ----
     46     2031 Standard Mouse        22,99    22,99      189,95     344,91    1
     22     2031 Keyboard              24,99    47,98      189,95     344,91    2
     47     2031 Deluxe Mouse          28,99    76,97      189,95     344,91    3
     27     2031 Speakers- 3"          44,99   121,96      189,95     344,91    7
     32     2031 Speakers- 5"          67,99   189,95      189,95     344,91    9
     38     2032 Internal 6X CD-ROM    29,99    29,99      154,96     344,91    4
     39     2032 Internal 8X CD-ROM    34,99    64,98      154,96     344,91    5
     34     2032 6X CD-ROM             39,99   104,97      154,96     344,91    6
     35     2032 8X CD-ROM             49,99   154,96      154,96     344,91    8
```

Das Merge Statement

Aufgabenstellungen Update

In der Tabelle COSTS sind die Kosten und Preise der Produkte je Verkaufskanal und je Werbeweg (Promotion) gespeichert. Für eine spezielle Hochrechnung sollen in der Kopie dieser Tabelle (COSTS_02) die Kosten neu berechnet werden. Dazu sollen je Monat die Maximalkosten des Produktes berechnet werden. Alle Kosten des Produktes in diesem Monat sollen um 10% dieser Maximalkosten erhöht werden.

Teil der Daten vor dem Update

PROD_ID	TIME_ID	PROMO_ID	CHANNEL_ID	UNIT_COST	UNIT_PRICE
148	08.07.01	999	3	18,32	23,55
148	18.07.01	999	2	17,83	23,69
148	18.07.01	999	4	17,52	22,37
148	23.07.01	999	4	17,52	22,37
148	02.07.01	999	3	18,32	23,55
148	08.07.01	999	4	17,52	22,37
148	02.07.01	999	4	17,52	22,37
148	27.07.01	999	3	18,32	23,55
148	27.07.01	999	4	17,52	22,37
148	27.07.01	999	2	17,83	23,69
148	08.07.01	999	2	17,83	23,69
148	02.07.01	999	2	17,83	23,69
148	23.07.01	999	3	18,32	23,55
148	18.07.01	999	3	18,32	23,55
148	23.07.01	999	2	17,83	23,69

Lösung mit einfachem SQL

Beim Update muss daher im SET wieder auf die Tabelle COSTS_2 zugegriffen werden, um die Kostenerhöhung zu berechnen.

Hier der Code:

```
UPDATE costs_2 a
   SET unit_cost = unit_cost +
       (SELECT max(unit_cost) * 0.1
        FROM costs_2 b
WHERE a.prod_id = b.prod_id
  AND to_char(a.time_id,'YYYY-MM') = to_char(b.time_id,'YYYY-MM'))
```

Teil der Daten nach dem Update

PROD_ID	TIME_ID	PROMO_ID	CHANNEL_ID	UNIT_COST	UNIT_PRICE
148	08.07.01	999	3	20.15	23.55
148	18.07.01	999	2	19.66	23.69
148	18.07.01	999	4	19.35	22.37
148	23.07.01	999	4	19.35	22.37
148	02.07.01	999	3	20.15	23.55
148	08.07.01	999	4	19.35	22.37
148	02.07.01	999	4	19.35	22.37
148	27.07.01	999	3	20.15	23.55
148	27.07.01	999	4	19.35	22.37
148	27.07.01	999	2	19.66	23.69
148	08.07.01	999	2	19.66	23.69

```
    148 02.07.01        999          2      19.66      23.69
    148 23.07.01        999          3      20.15      23.55
    148 18.07.01        999          3      20.15      23.55
    148 23.07.01        999          2      19.66      23.69
```

Lösung mit MERGE

Beim Mergestatement werden im Using die Daten angegeben, die für den Update oder auch für einen Insert gebraucht werden. Das enstpricht der Quelle (source). Die Zieltabelle (destination) wird beim INTO angegeben.
Mit der ON-Klasel wird festgelegt, welche Attribute der Quelle mit dem Ziel übereinstimmen sollen. Stimmen die Werte überein, so wird ein Update gemacht. Nach dem Update wäre es möglich auch ein Insert zu bestimmen, was hier aber nicht gebraucht wird.
In der Lösung mit Update musste für jeden Datensatz in costs_2 die Tabelle costs_2 gelesen werden, um den Betrag zu berechnen, der zu den unit_costs addiert werden soll. Mit dem Merge haben wir die Möglichkeit, diese Wert im Using zu berechnen und haben dann im Update direkt Zugriff auf diesen errechneten Betrag. So hat Oracle die Möglichkeit, die Daten performanter zu lesen.

Hier der Code:

```
MERGE INTO costs_2 d USING
 (SELECT detail.prod_id, detail.time_id, detail.promo_id, detail.channel_id,
month.max_cost
  FROM costs_2 detail,
      (SELECT prod_id, to_char(time_id,'YYYY-MM') month, max(unit_cost) * 0.1
max_cost
       FROM costs_2
       GROUP BY prod_id, to_char(time_id,'YYYY-MM')) month
  WHERE detail.prod_id = month.prod_id
    AND to_char(detail.time_id,'YYYY-MM') = month.month) s
ON (d.prod_id = s.prod_id AND
    d.time_id = s.time_id AND
    d.promo_id = s.promo_id AND
    d.channel_id = s.channel_id)
WHEN MATCHED THEN
UPDATE SET d.unit_cost = d.unit_cost + s.max_cost
```

Wer das Kapitel Analytische Funktionen weiter oben gelesen hat, hat sich nun sicher überlegt, ob der Betrag max_cost einfacher mit einer analytischen Funktion gelesen warden kann.
Das ist so und hier kommt der Code:

```
MERGE INTO costs_2 d USING
 (SELECT detail.prod_id, detail.time_id, detail.promo_id, detail.channel_id,
         sum(unit_cost) OVER (PARTITION BY prod_id, to_char(time_id,'YYYY-MM'))
max_cost
  FROM costs_2 detail) s
ON (d.prod_id = s.prod_id AND
    d.time_id = s.time_id AND
    d.promo_id = s.promo_id AND
    d.channel_id = s.channel_id)
WHEN MATCHED THEN
UPDATE SET d.unit_cost = d.unit_cost + s.max_cost
```

Der folgende Performancevergleich wird zeigen, welche der gewählten Lösungen, die schnellste ist. Sie dürfen gerne schon mal raten.

Performancevergleich

Executionplan mit Update

Operation	Name	Rows	Bytes	Cost	Time	CPU Cost	I/O Cost
UPDATE STATEMENT (ALL_ROWS)		84069	2942415	88	00:00:02	22013531	84
UPDATE	SH.COSTS_2	0	0	0	00:00:00	0	0
TABLE ACCESS FULL	SH.COSTS_2 (TABLE)	84069	2942415	88	00:00:02	22013531	84
SORT AGGREGATE		1	35	0	00:00:00	0	0
TABLE ACCESS FULL	SH.COSTS_2 (TABLE)	8	280	88	00:00:02	22056035	84

Laufzeit: 00:45:40.00

Executionplan mit Merge und Join

Operation	Name	Rows	Bytes	Cost	Time	CPU Cost	I/O Cost
MERGE STATEMENT (ALL_ROWS)		3354306	130817934	3059	00:00:37	602908213	2955
MERGE	SH.COSTS_2	0	0	0	00:00:00	0	0
VIEW		0	0	0	00:00:00	0	0
HASH JOIN		3354306	553460490	3059	00:00:37	602908213	2955
HASH JOIN		266800	35751200	863	00:00:11	106377963	845
TABLE ACCESS FULL	SH.COSTS_2 (TABLE)	90521	4345008	88	00:00:02	21687071	84
TABLE ACCESS FULL	SH.COSTS_2 (TABLE)	90521	7784806	88	00:00:02	25307911	84
VIEW		90521	2806151	101	00:00:02	96471726	84
SORT GROUP BY		90521	3168235	101	00:00:02	96471726	84
TABLE ACCESS FULL	SH.COSTS_2 (TABLE)	90521	3168235	88	00:00:02	23497491	84

Laufzeit: 00:00:08.62

Executionplan mit Merge und analytische Funktion

Operation	Name	Rows	Bytes	Cost	Time	CPU Cost	I/O Cost
MERGE STATEMENT (ALL_ROWS)		266801	10405239	2270	00:00:28	203054961	2235
MERGE	SH.COSTS_2	0	0	0	00:00:00	0	0
VIEW		0	0	0	00:00:00	0	0
HASH JOIN		266801	39219747	2270	00:00:28	203054961	2235
VIEW		90521	5521781	1439	00:00:18	117773216	1419
WINDOW SORT		90521	5521781	1439	00:00:18	117773216	1419
TABLE ACCESS FULL	SH.COSTS_2 (TABLE)	90521	5521781	88	00:00:02	23497491	84
TABLE ACCESS FULL	SH.COSTS_2 (TABLE)	90521	7784806	88	00:00:02	25307911	84

Laufzeit: 00:00:08.46

Zwischen dem UPDATE und MERGE gibt es einen markanten Unterschied.

Allgemeine Hinweise zu MERGE

In der Regel wird das MERGE dann gebraucht, wenn einige Datensätze mit UPDATE geändert und andere mit INSERT eingefügt werden müssen. Dieses Problem haben wir in unserem Datawarehouse beim Laden der Daten aus der Stating in das Star-Schema. Bestehende Kennzahlen können sich im Produktiven System verändern und für neue Tage oder neue Produkte entstehen neue Kennzahlen. All diese Kennzahlen werden zuerst in die Staging geschrieben und bereinigt. Dann müssen sie im Star-Schema ergänzt werden. Gibt es zur Kombination der Dimensionswerte bereits einen Eintrag für die Kennzahl, so wird diese überschrieben. Existiert noch keine Kennzahl, so wird diese eingefügt.
Ohne Merge müsste zuerst festgestellt werden, welche Datensätze in der Staging neu sind. Diese könnten dann eingefügt und die anderen aktualisiert werden. Das wären aber mehrere

SQL-Befehle, die immer viele Daten lesen. Das MERGE ist daher nicht nur einfacher, sondern klar auch schneller.

Multitable Insert

Aufgabenstellungen Daten auf mehrere Tabellen verteilen

In der Tabelle COSTS_2 sind die Preise und neu berechneten Kosten der Produkte je Verkaufskanal und je Werbeweg (Promotion) gespeichert. Diese Daten sollen je Produktegruppe auf neue Tabellen verteilt werden.

Lösung mit einfachem SQL

Mit einem normalen INSERT INTO SELECT kann die Aufgabe sehr einfach gelöst werden. Der Nachteil: die Tabelle COSTS_2 wird 5 mal gelesen.

Hier der Code:

```
INSERT INTO costs_201 (prod_id, time_id, promo_id, channel_id, unit_cost,
unit_price)
SELECT c.prod_id, c.time_id, c.promo_id, c.channel_id, c.unit_cost, c.unit_price
FROM costs_2 c, products p
WHERE c.prod_id = p.prod_id
  AND p.prod_category_id = 201;
INSERT INTO costs_202 (prod_id, time_id, promo_id, channel_id, unit_cost,
unit_price)
SELECT c.prod_id, c.time_id, c.promo_id, c.channel_id, c.unit_cost, c.unit_price
FROM costs_2 c, products p
WHERE c.prod_id = p.prod_id
  AND p.prod_category_id = 202;
…
```

Lösung mit Multitable Insert

Das Multitable Insert erlaubt das Resultat eines Selects nach definierbaren Kriterien in verschiedene Tabellen zu verteilen.

Hier der Code:

```
INSERT ALL
WHEN prod_category_id = 201 THEN
   INTO costs_201 VALUES (prod_id, time_id, promo_id, channel_id,
                          unit_cost, unit_price)
WHEN prod_category_id = 202 THEN
   INTO costs_202 VALUES (prod_id, time_id, promo_id, channel_id,
                          unit_cost, unit_price)
WHEN prod_category_id = 203 THEN
   INTO costs_203 VALUES (prod_id, time_id, promo_id, channel_id,
                          unit_cost, unit_price)
WHEN prod_category_id = 204 THEN
   INTO costs_204 VALUES (prod_id, time_id, promo_id, channel_id,
                          unit_cost, unit_price)
WHEN prod_category_id = 205 THEN
   INTO costs_205 VALUES (prod_id, time_id, promo_id, channel_id,
                          unit_cost, unit_price)
SELECT c.prod_id, c.time_id, c.promo_id, c.channel_id, c.unit_cost, c.unit_price,
```

```
      p.prod_category_id
FROM costs_2 c, products p
WHERE c.prod_id = p.prod_id
```

Performancevergleich

Es ist offensichtlich, dass mit einfachem SQL die Basisdaten 5 mal gelesen werden müssen und mit Multitable Insert nur gerade einmal. Der Performancevergleich des obigen Beispiels zeigt aber, dass beide Lösungen ähnlich schnell sind. Ein Multitable Insert lohnt sich also vor allem dann, wenn der Select entsprechend komplex und langsam ist. Dann kann es von grossem Vorteil sein, wenn diese Daten nur genau einaml gelesen werden müssen.

Schlussfolgerung

In der Praxis zeigt sich häufig, dass neuere Möglichkeiten von Oracle nicht genützt werden, weil sie nicht bekannt sind. Funktionen wie das MERGE-Statement wird dann hervor geholt, wenn es um Data Warehousing geht. Doch unser Beispiel ziegt, dass es auch als Ersatz für ein UPDATE-Statement sinnvoll eingesetzt werden kann.

So lohnt es sich zwischendurch externe Wissen zu holen; sei es in Form von Weiterbildung, von Konferenzbesuchen, von Codereview oder von externen Spezialisten und Spezialistinnen, die neue Ideen einbringen. Gerne helfen wir dabei.

Kontaktadresse:

Dr. Andrea Kennel
InfoPunkt Kennel GmbH
Bahnhofstr. 48
CH-8600 Dübendorf

Telefon: +41(0)44-820 71 40
E-Mail info@infokennel.ch
Internet: www.infokennel.ch