Advanced SQL

Komplexe Abfragen mit aktuellem SQL

SQL wird oft unterschätzt und kann mehr als einfache Joins und Unterabfragen

Holger@Jakobs.com – http://plausibolo.de

FrOSCon 2016

Vorstellung Holger Jakobs

- studierter Wirtschaftsinformatiker
- jahrzehntelange Erfahrung in der Ausbildung von Software-Entwicklern
- seit 2013 freiberuflich als Berater und Trainer
- Lehrauftrag an der Fachhochschule der Wirtschaft für Betriebssysteme
- LPIC-2-zertifiziert und PRINCE2 Practitioner
- Fan von PostgreSQL, Tcl/Tk, Fossil, …
- Motto: KISS – keep it simple & stupid

ANSI/ISO SQL

- Die SQL-Standards haben sich seit 1986 sehr weit entwickelt.
- Kein RDBMS erfüllt auch nur annähernd den gesamten Standard.
- Es gibt trotzdem sehr große Unterschiede bei der Standardumsetzen.
- Ein Feature nicht umzusetzen ist etwas anderes als klar gegen den Standard zu verstoßen.

1992 und 1999 waren besonders große Sprünge bei den Features

SQL-92

- Datentypen VARCHAR, DATE, TIME, TIMESTAMP
- Typumwandlungen mit CAST
- [NATURAL] JOIN [ON | USING]
- CHECK Constraints
- Information Schema
- SQLSTATE (statt SQLCODE)

1992 und 1999 waren besonders große Sprünge bei den Features

SQL:1999

- Datentyp BOOLEAN
- benutzerdefinierte Typen: CREATE TYPE
- Arrays
- Grundlegende OLAP-Fähigkeiten
  - grouping sets
  - Window-Funktionen
- Common Table Expressions
- rollenbasierte Rechteverwaltung

Common Table Expressions

`WITH`-Schlüsselwort definiert eine virtuelle Tabelle, die nur für die aktuelle Anweisung Gültigkeit hat.

```
WITH cteName1 AS (SELECT …),
cteName2 AS (SELECT … FROM cteName1 …)
SELECT … FROM cteName1 …
   … FROM cteName2 …;
```

In `cteName2` kann auf `cteName1` zugegriffen werden, im Haupt-Select auf beide Common Tables.

Common Table Expressions

Zwecke von CTEs:

- Vereinfachung von komplexen Abfragen durch Aufteilung und sinnvolle Benennung der „Zwischenergebnisse“.
- Vermeidung der Mehrfachermittlung von „Zwischenergebnissen“. Alle CTEs werden nur 1x ausgewertet.
- `VIEW`-ähnliche Konstruktion ohne Recht zur Objekterzeugung.

Common Table Expressions

Arten von CTEs:

- Gewöhnliche CTEs nur zur Vereinfachung und zur Vermeidung von Mehrfachauswertung
- Rekursive CTEs als Schleifenersatz in SQL
  (Nur PostgreSQL verlangt das Schlüsselwort `RECURSIVE`, bei anderen RDBMS ist es wahlfrei).
- Rekursive CTEs dienen oft dazu, in Tabellen abgebildete Hierarchien abzufragen.
- Beispiele: Mitarbeiterhierarchien und Stücklisten

2 CTEs und ein Haupt-Select

```
WITH regionUmsatz AS (
  SELECT region, sum(betrag) AS betragSumme
  FROM v_auftrag   GROUP BY region),

 topRegion AS (SELECT region FROM regionUmsatz
  WHERE betragSumme >
  (SELECT SUM(betragSumme)/10 FROM regionUmsatz))

SELECT region, produkt, SUM(anzahl) as anzahl,
       SUM(betrag) AS betragSumme
FROM v_auftrag
WHERE region IN (SELECT region FROM topRegion)
GROUP BY region, produkt;
```

Rekursive CTE ohne Datenzugriff

Zahlen summieren:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT SUM(n) AS summe FROM t;
```

Rekursive CTE ohne Datenzugriff

Legende von der Erfindung des Schachspiels:

```
WITH RECURSIVE t(n) AS (
    SELECT CAST(1 AS NUMERIC)
  UNION ALL
    SELECT n*2 FROM t
) SELECT n FROM t LIMIT 64;
```

Rekursive CTE mit Datenzugriff

Angestelltenhierarchie:

```
WITH RECURSIVE ang (emp_id, name, boss, boss_name, vollst) AS (
  -- nicht-rekursiver Teil
  SELECT emp_id, emp_name, CAST (NULL AS INTEGER),
      CAST (NULL AS VARCHAR(100)), emp_name
  FROM emp
  WHERE boss IS NULL

UNION ALL

  -- rekursive Unterabfrage
  SELECT a.emp_id, a.emp_name, a.boss, v.boss_name,
      CAST(v.vollst || '->' || a.emp_name AS VARCHAR(100))
  FROM emp a JOIN ang v
  ON a.boss = v.emp_id
)

SELECT * FROM ang
ORDER BY emp_id;
```

Rekursive CTE mit Datenzugriff

Angestelltenhierarchie:

emp_id	name	boss	vollst
1	Hans		Hans
2	Anna	1	Hans->Anna
3	Udo	2	Hans->Anna->Udo
4	Herbert	2	Hans->Anna->Herbert
5	Kai	2	Hans->Anna->Kai
6	Marie	2	Hans->Anna->Marie
7	Mona	2	Hans->Anna->Mona
8	Urs	1	Hans->Urs
9	Uta	8	Hans->Urs->Uta
10	Ulli	8	Hans->Urs->Ulli
11	Udo	8	Hans->Urs->Udo
12	Uwe	8	Hans->Urs->Uwe
13	Mandy	8	Hans->Urs->Mandy

…

OLAP ohne Data Warehouse

- Erweiterung von `GROUP BY` durch `GROUPING SETS`
- Mehrere Gruppierungen gleichzeitig in einer Abfrage
- Auch Gesamtergebnis ist möglich – neben den einzelnen Gruppenergebnissen.

```
SELECT dept_id, dept_name, job_id, job_name, COUNT(*)
FROM emp JOIN dept USING(dept_id) NATURAL JOIN job
GROUP BY GROUPING SETS
  ((dept_id, dept_name), (job_id, job_name), ())
ORDER BY dept_id, job_id;
```

OLAP ohne Data Warehouse

dept_id	dept_name	job_id	job_name	count
1	owner			1
2	sales			6
3	development			6
4	production			8
		0	Inhaber	1
		1	Verkäufer	3
		2	Verkaufsleitung	1
		3	Ingenieur	1
		4	Techniker Entwicklung	2
		5	Zeichner	2
		6	Bürofachkraft	4
		12	Techniker Produktion	4
		13	Produktionsleitung	1
		14	Schichtleitung	2
				21

OLAP ohne Data Warehouse

```
SELECT COALESCE(dept_id, job_id) as nummer,
   CASE GROUPING(dept_name)
   WHEN 1 THEN '' ELSE 'in Abt. ' || dept_name END
|| CASE GROUPING(job_name)
   WHEN 1 THEN '' ELSE 'Job ' || job_name END
|| CASE GROUPING(job_name) + GROUPING(dept_name)
   WHEN 2 THEN 'Gesamtwert' ELSE '' END
AS "Abt./Job", COUNT(*) AS "Anzahl MA"

FROM emp JOIN dept USING(dept_id) NATURAL JOIN job

GROUP BY GROUPING SETS
  ((dept_id, dept_name), (job_id, job_name), ())
```

OLAP ohne Data Warehouse

nummer	Abt./Job	Anzahl MA
1	in Abt. owner	1
2	in Abt. sales	6
3	in Abt. development	6
4	in Abt. production	8
0	Job Inhaber	1
1	Job Verkäufer	3
2	Job Verkaufsleitung	1
3	Job Ingenieur	1
4	Job Techniker Entwicklung	2
5	Job Zeichner	2
6	Job Bürofachkraft	4
12	Job Techniker Produktion	4
13	Job Produktionsleitung	1
14	Job Schichtleitung	2
	Gesamtwert	21

OLAP ohne Data Warehouse

```
SELECT dept_id, dept_name, job_id, job_name, COUNT(*)
FROM emp JOIN dept USING(dept_id) NATURAL JOIN job
GROUP BY ROLLUP
  ((dept_id, dept_name), (job_id, job_name))
ORDER BY dept_id, job_id;
```

OLAP ohne Data Warehouse

dept_id	dept_name	job_id	job_name	count
1	owner	0	Inhaber	1
1	owner			1
2	sales	1	Verkäufer	3
2	sales	2	Verkaufsleitung	1
2	sales	6	Bürofachkraft	2
2	sales			6
3	development	3	Ingenieur	1
3	development	4	Techniker Entwicklung	2
3	development	5	Zeichner	2
3	development	6	Bürofachkraft	1
3	development			6
4	production	6	Bürofachkraft	1
4	production	12	Techniker Produktion	4
4	production	13	Produktionsleitung	1
4	production	14	Schichtleitung	2
4	production			8
				21

OLAP ohne Date Warehouse

- `WINDOW`-Funktionen bieten neue Abfragemöglichkeiten.
- Die `WINDOW`-Funktionen haben eine neue Ära eingeläutet.
- Idee: Es gibt nicht nur die Gesamttabelle oder die aktuelle Zeile, sondern ein (mehrzeiliges) Fenster, auf das man bei den Abfragen schaut.
- Definition der Fenster geschieht über die `OVER`-Klausel

Syntax der OVER()-Klausel

Folgende Elemente können vorkommen:

- PARTITION BY gibt das Einteilungskriterium an.
- ORDER BY gibt die Sortierreihenfolge in den Partitionen an.
- ROWS … oder RANGE … gibt an, welche Zeilen berücksichtigt werden.
- Die Klammer hinter OVER kann auch leer sein.

ORDER BY hinter OVER

```
SELECT SUM(x)  OVER (ORDER BY x)
FROM generate_series(1, 100) AS t(x);
```

Zeigt die Summen der ersten n Zahlen für n von 1 bis 100.

```
 sum
------
    1
    3
   …
 4950
 5050
```

PARTITION BY hinter OVER

```
SELECT emp_id, emp_name, dept_id, dept_name,
COUNT(*) OVER (PARTITION BY dept_id) AS
anz_ma,
CAST(MIN(salary) OVER (PARTITION BY dept_id)
AS VARCHAR) || ' bis ' || CAST(MAX(salary)
OVER (PARTITION BY dept_id) AS VARCHAR)
AS gehaltsspanne,
CAST(AVG(salary) OVER (PARTITION BY dept_id)
AS NUMERIC(10,2)) AS schnitt
FROM emp JOIN dept USING(dept_id);
```

PARTITION BY hinter OVER

- Immer wieder dieselbe "PARTITION BY"-Klausel ist lästig und fehlerträchtig.
- Darum kann man Fenster-Definitionen benennen und immer wieder verwenden.
- Dies geschieht mit dem Schlüsselwort WINDOW

PARTITION BY hinter OVER

```
SELECT emp_id, emp_name, dept_id, dept_name,
  COUNT(*)          OVER abt AS anz_ma,
  CAST(MIN(salary) OVER abt AS VARCHAR)
  || ' bis ' ||
  CAST(MAX(salary) OVER abt AS VARCHAR)
  AS gehaltsspanne,
  CAST(AVG(salary) OVER abt AS NUMERIC(10,2))
    AS schnitt
FROM emp JOIN dept USING(dept_id)
WINDOW abt as (PARTITION BY dept_id);
```

ROW/RANGE hinter OVER

- Ohne ORDER BY und ohne ROWS/RANGE werden alle Zeilen verwendet.
- Mit ORDER BY, aber ohne ROWS/RANGE werden alle Zeilen vom Tabellenanfang bis zu denen mit demselben Wert wie die aktuelle Zeile verwendet.
  RANGE UNBOUNDED PRECEDING heißt das ausdrücklich
- Es können Ober- und Untergrenze angegeben werden.

ROW/RANGE hinter OVER

Hinter ROWS oder RANGE dürfen stehen:

- UNBOUNDED PRECEDING = BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- BETWEEN … AND …, mit … =
  - UNBOUNDED PRECEDING
  - *n* PRECEDING
  - CURRENT ROW
  - *n* FOLLOWING
  - UNBOUNDED FOLLOWING

Ordnungsfunktionen

- `ROW_NUMBER()`, `RANK()` und `DENSE_RANK()` sind parameterlos.
- Sie benötigen `ORDER BY`.
- `ROW_NUMBER()` nummeriert alle Zeile einfach durch.
- `RANK()` gibt den Rang an, wobei nach gleichwertigen Zeilen Lücken entstehen.
- `DENSE_RANK()` tut dasselbe ohne Lücken.

Ordnungsfunktionen

```
SELECT ROW_NUMBER() OVER () AS lfdnr,
ROW_NUMBER() OVER abt AS nr_in_abt,
emp_id, emp_name, dept_name,
RANK() OVER abt AS rang,
DENSE_RANK() OVER abt AS d_rang

FROM emp JOIN dept USING(dept_id)

WINDOW abt as (PARTITION BY dept_id
ORDER BY salary DESC)

ORDER BY dept_id, salary DESC;
```

Ordnungsfunktionen

lfdnr	nr_in_abt	emp_id	emp_name	dept_name	rang	d_rang
1	1	1	Hans	owner	1	1
2	1	2	Anna	sales	1	1
3	2	4	Herbert	sales	2	2
4	3	3	Udo	sales	3	3
5	4	7	Mona	sales	4	4
6	5	6	Marie	sales	5	5
7	6	5	Kai	sales	6	6
8	1	8	Urs	development	1	1
9	2	9	Uta	development	2	2
10	3	12	Uwe	development	2	2
11	4	13	Mandy	development	4	3
12	5	10	Ulli	development	5	4
13	6	11	Udo	development	6	5
14	1	14	Peter	production	1	1
15	2	20	Patti	production	2	2
16	3	15	Petra	production	3	3
17	4	19	Pam	production	4	4
18	5	16	Paul	production	4	4
19	6	17	Pepe	production	6	5
20	7	21	Marie	production	7	6
21	8	18	Patrick	production	7	6

Happy Birthday mit CTEs und Window-Funktionen

- In der Firma ist es üblich, dass Kollegen Geburts-tagsfeiern ausrichten, und zwar ist immer derjenige zuständig, der als nächstes Geburtstag hat.
- Wir benötigen also eine Liste mit den Mitarbeitern und dem Kollegen, der vor ihnen als letztes Geburts-tag hat.
- Natürlich muss auch angegeben sein, an welchem Tag und Monat die Feier stattfindet.
- Dieses Ausrichten der Geburtstagsfeier wird inner-halb der einzelnen Abteilungen geregelt.

Happy Birthday mit CTEs und Window-Funktionen

- t1 ermittelt das Geburtsdatum ohne Jahr als Spalte bd

```
t1 AS (SELECT emp_id, emp_name,
dept_id, dept_name, birthdate,
SUBSTRING(CAST(birthdate AS VARCHAR) FROM 6 FOR
5) AS bd
FROM emp JOIN dept USING(dept_id))
```

-

```
t2 AS (SELECT emp_id, emp_name,
dept_id, dept_name, bd,
COALESCE(LAG (bd)        OVER dept,
         LAST_VALUE(bd) OVER dept) AS geb_bd
FROM t1 WINDOW dept AS
  (PARTITION BY dept_id ORDER BY bd
   ROWS BETWEEN CURRENT ROW
        AND UNBOUNDED FOLLOWING)
```

Happy Birthday mit CTEs und Window-Funktionen

- Haupt-SELECT mit Self-Join über t2:

```
SELECT a.emp_id, a.emp_name, a.dept_name, a.bd,
  g.emp_id as geb_id,
  g.emp_name as geb_name,
  g.bd
FROM t2 a JOIN t2 g
  ON a.dept_id = g.dept_id
 AND a.geb_bd  = g.bd;
```

- Hierbei ist a der ausrichtende Angestellte und g das Geburtstagskind.
- Wie man sieht, kann SQL sogar bei solch kniffligen Fragestellungen eine Antwort geben – ganz ohne prozedurale/objektorientierte Programmierung.