Advanced SQL-SELECT-Abfragen

Holger Jakobs – holger@jakobs.com

2016-08-26

Inhaltsverzeichnis

1 Das SELECT-Kommando

Das `SELECT`-Kommando ist zunächst einmal ein sehr grundlegendes, denn ohne `SELECT` kann man bei SQL-Datenbanken überhaupt keine Daten lesen. Der Umfang und die Leistungsfähigkeit sind allerdings seit dem Standard SQL:1999 (oder auch SQL-99 genannt) sehr stark gewachsen. Einen Überblick über die diversen SQL-Standards gibt das Dokument „SQL-Standards“ vom selben Autor. Das Dokument „Die SQL-Select-Anweisung“ erläutert die Grundlagen der SQL-Abfrage gemäß der SQL-Standards vor 1999, d. h. alles, was man so im täglichen Leben als SQL-Anwender benötigt.

Mit SQL-99 gab es zwei wesentliche Neuerungen bezüglich `SELECT`:

1. Common Table Expressions (CTEs[1])
2. grundlegende OLAP[2]-Fähigkeiten

Diese Neuerungen werden im vorliegenden Dokument erläutert. Sie können in einigen Fällen sehr hilfreich sein, kommen aber in der täglichen Arbeit nicht so häufig vor. Wenn sie aber angebracht sind, vereinfachen Sie die Arbeit ungeheuer und können auch die Performance der Abfragen um ein Vielfaches steigern. Zudem entlasten sie die Anwendungsprogramme von Arbeit, für die sie nicht gut geeignet sind.

Grundsätzlich sollten sich alle Anwendungsentwickler beim Abfragen von Daten aus relationalen Datenbanksystemen immer fragen, ob eine Verarbeitung der Daten im Anwendungsprogramm sinnvoll ist – oder ob es nicht viel besser wäre, die Datenbank auf eine Weise zu fragen, dass gleich die „fertigen“ Daten an das Anwendungsprogramm geliefert werden. In fast allen Fällen ist letzteres günstiger. Ein Extrembeispiel wäre das Abfragen aller Daten einer Tabelle, nur um beim Abholen der Tabellenzeilen einen Zähler mitlaufen zu lassen. Richtig wäre es, die Datenbank die Tabellenzeilen mittels `COUNT(*)` zählen und dann diese eine Zahl liefern zu lassen. Leider finden Tests oft nur mit kleinen Datenmengen statt, so dass beide Lösungen als quasi gleichwertig erscheinen. Sobald die Datenmenge größer wird – bei einer Liste mit Ländernamen und Hauptstädten beispielsweise ist das nicht anzunehmen, bei den Einzelverbindungsnachweisen eines Mobilfunkproviders aber sehr wohl – liegen zwischen beiden Ansätzen mehrere Größenordnungen an Systembelastung bzw. Laufzeit.

Ähnliche Beispiele gibt es auch bei komplexeren Abfragen, die mit dem klassischen SQL vor 1999 nicht so einfach darzustellen waren. Daher gibt es aus dieser Zeit noch viele Anwendungen, die Informationen im prozeduralen Code berechnen. Das muss man jetzt nicht unbedingt mit einem großen Knall in allen vorhandenen und zufriedenstellend laufenden Anwendungen sofort ändern, aber bei Neuentwicklungen oder bei der Suche nach Performanceproblemen muss man an diese Anwendungen und die zugrunde liegenden SQL-Abfragen ran und sie „richtig“ machen.

Der Einsatz von „Advanced SQL“ setzt allerdings voraus, dass das verwendete Datenbanksystem diese Abfragen unterstützt. Das muss man ggf. in der Dokumentation nachschlagen und zur Sicherheit auch ausprobieren. Datenbanksysteme, die ohnehin den Namen kaum verdienen, sind auch in diesem Bereich eher schwach aufgestellt.

Eine SQL-Datei zum Erzeugen der Datenbank mit Mitarbeitern, Vorgesetzten und Abteilungen, welche u. a. für das Beispiel in Abschnitt 2.4.1 auf Seite 7 vewendet wird, ist als Anhang vorhanden. Bitte auf das nebenstehende Symbol klicken, um sie zu laden. Alle angehängten Dateien sind UTF-8-codiert und haben Newline-Zeichen als Zeilentrenner. Die Tabellen sind wie untenstehend definiert. Die Inhalte sind hier nicht dokumentiert, aber in der Datei im Anhang enthalten. Der „Datentyp“ `SERIAL` ist PostgreSQL-spezifisch, muss

[1] https://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL#Common_table_expression
[2] https://de.wikipedia.org/wiki/Online_Analytical_Processing

ggf. durch eine vergleichbare Angabe zum Füllen nicht angegebener `INTEGER`-Werte aus einer Sequenz oder einen anderen „auto-increment“-Mechanismus ersetzt werden.

```
CREATE TABLE dept (
  dept_name  VARCHAR(100)   NOT NULL UNIQUE,
  budget     NUMERIC(10,2)
    CHECK (budget > 0 OR budget IS NULL),
  boss       INTEGER,
  dept_id    SERIAL         PRIMARY KEY
);

CREATE TABLE job (
  job_name   VARCHAR(50)    NOT NULL,
  job_id     SERIAL         PRIMARY KEY
);

CREATE TABLE emp (
  emp_name   VARCHAR(100)   NOT NULL
    CONSTRAINT "Name darf nicht leer sein!" CHECK (TRIM(emp_name) <> ''),
  birthdate  DATE           NOT NULL,
  job_id     INTEGER        NOT NULL REFERENCES job,
  salary     NUMERIC(10,2)
    CHECK (salary > 1000 OR salary IS NULL AND job_id = 0),
  boss       INTEGER        REFERENCES emp
    CHECK (boss <> emp_id),
  dept_id    INTEGER        REFERENCES dept,
  emp_id     SERIAL         PRIMARY KEY
);
```

Für das Stücklistenbeispiel in Abschnitt 2.4.2 auf Seite 9 gibt es einen Anhang. Die Tabellen sind wie untenstehend definiert. Die Inhalte sind hier nicht dokumentiert, aber in der Datei im Anhang enthalten.

```
CREATE TABLE part (
  partno    SERIAL       PRIMARY KEY,
  descr     VARCHAR(30)  NOT NULL UNIQUE
);

CREATE TABLE sub_part (
  partno    INTEGER      REFERENCES part,
  subpartno INTEGER      REFERENCES part,
  quantity  INTEGER      CHECK (quantity > 0),
  PRIMARY KEY (partno, subpartno)
);
```

Eine weitere SQL-Datei zum Erzeugen der Tabelle mit den Temperaturdaten für das Beispiel zur Zeitreihenanalyse in Abschnitt 3.12 auf Seite 28 ist ebenfalls als Anhang vorhanden. Es handelt sich nur um eine Tabelle der folgenden Struktur.

```
CREATE TABLE gltemps (
    zeit    TIMESTAMP PRIMARY KEY,
    temp_c  REAL
);
```

2 Common Table Expressions

2.1 Zweck

Der Zweck von Common Table Expressions ist zunächst einmal die Vereinfachung von Abfragen, indem komplexe Abfragen in mehrere einfachere Teile zerlegt werden. Ein Beispiel hierfür ist:

```
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;
```

Was soll diese Abfrage ermitteln? Es geht darum, alle Aufträge aus den Top-Regionen aufzulisten, d. h. den Regionen, die zu mehr als 10 % zum Gesamtumsatz beitragen. Hierzu werden zunächst alle Aufträge pro Region aufsummiert mit der Abfrage der ersten CTE:

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

Das Ergebnis steht durch die CTE unter dem Namen `regionUmsatz` zur Verfügung und wird in der zweiten CTE eingesetzt.

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

```
    GROUP BY region)
SELECT region
  FROM regionUmsatz
  WHERE betragSumme > (SELECT SUM(betragSumme)/10 FROM regionUmsatz);
```

Bei diesem `SELECT` wird eine Liste der Regionen erstellt mit der Einschränkung, dass die Region eine höhere Betragssumme hat als die Gesamtsumme geteilt durch 10.

Diese Abfrage hätte auch vollständig ohne CTEs geschrieben werden können, allerdings wären hierfür mehrfach geschachtelte Sub-Selects notwendig gewesen. Mit den CTEs ist die ganze Sache deutlich übersichtlicher. Man kann sich sozusagen schrittweise an die Lösung herantasten und den Zwischenergebnissen aussagekräftige Namen geben.

2.2 Syntax und einfache CTEs

Die Syntax des `SELECT`-Kommandos wurde durch SQL-99 erweitert. Das Kommando beginnt jetzt nicht mehr zwingend mit dem Schlüsselwort `SELECT`. Vielmehr können vor dem `SELECT` mittels des Schlüsselworts `WITH` Common Table Expressions festgelegt und weiter unten verwendet werden. Diese stellen virtuelle Tabellen dar, die nur für das laufende Kommando existieren und können innerhalb des `SELECT`-Abschnitts dann wie gewöhnliche Tabellen (Tables) und Sichten (Views) verwendet werden.

Im Gegensatz zu Sichten werden keine besonderen Rechte zur Erzeugung benötigt, weil hierbei keine neuen Objekte in der Datenbank erzeugt werden. Es genügen reine `SELECT`-Rechte. Die virtuellen Tabellen werden nur ein einziges Mal erzeugt, auch wenn sie innerhalb des `SELECT` mehrfach angesprochen werden. Dies ist zu berücksichtigen, wenn die CTEs auch Funktionen enthalten, die ggf. Nebeneffekte haben (auch wenn das grundsätzlich keine besonders gute Idee ist).

```
[ WITH cteName1 AS (SELECT ...) [ , cteName2 AS (SELECT ...) ] ]
SELECT ... FROM cteName1 ... FROM cteName2 ...
```

Im SQL-99 gibt es also eine wahlfreie `WITH`-Klausel (daher in eckigen Klammern dargestellt) mit einer oder mehreren durch Komma getrennten Common Table Expressions, bestehend aus dem CTE-Namen, dem Schlüsselwort `AS` und einer in runden Klammern eingeschlossenen `SELECT`-Anweisung. Weiter hinten stehende `SELECT`-Anweisungen können weiter vorn stehende CTEs wie gewöhnliche Tabellen und Sichten verwenden.

Auf die CTEs kann in weiter hinten festgelegten CTEs und auch im Haupt-Select zugegriffen werden.

2.3 Rekursive CTEs ohne Datenbankzugriff

Neben den gewöhnlichen CTEs gibt es sogenannte rekursive CTEs. Sie bestehen aus zwei Teilen: einem nicht-rekursiven Teil (auch initialer Teil genannt), dem Schlüsselwort `UNION` (oft als `UNION ALL`) und einem rekursiven Teil, in dem auf die Ausgabe der Abfrage selbst wieder zugegriffen wird. Die Auswertung geschieht folgendermaßen:

1. Der initiale Teil wird ausgewertet. Die Zeilen kommen in eine temporäre Arbeitstabelle.

2. Der rekursive Teil wird auf diese Arbeitstabelle angewendet. Sofern er kein leeres Ergebnis liefert, wird die rekursive Abfrage immer weiter wiederholt, wobei das Ergebnis der Arbeitstabelle hinzugefügt wird.

Rekursive CTEs ermöglichen Dinge, die mit gewöhnlichem SQL nicht möglich sind.

Ein extrem simples Beispiel dafür wäre eine Abfrage, die die Zahlen von 1 bis 100 addiert:

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

Durch den rekursiven Ausdruck wird eine Tabelle von 100 Zeilen erzeugt, die von der Hauptabfragen dann per Aggregatfunktion auf eine Zeile reduziert wird.

Ein weiteres, rein mathematisches Beispiel ist die Berechnung von Fakultäten:

```
WITH RECURSIVE temp (n, fakult) AS
(SELECT 0, 1 -- nicht-rekursiver Teil
  UNION ALL
 SELECT n+1, (n+1)*fakult FROM temp -- rekursive Unterabfrage
        WHERE n < 9)
SELECT * FROM temp;
```

Da die Ergebnisspalten vom Typ `INTEGER` sind, laufen diese bei größeren Werten schnell über. Um das zu verhindern, kann man sie im nicht-rekursiven Teil – der ja als erstes ausgewertet wird und damit die Datentypen vorgibt – auf einen anderen Datentyp casten.

Hier ist die CTE zweispaltig und wird mit 0 und 1 initialisiert, während der rekursive Teil die erste Spalte um 1 erhöht und die zweite Spalte den Wert der vorigen Zeile mit diesem erhöhten Wert multipliziert. Links steht also `n` und rechts `n!`. Da hier keine Aggregatfunktion vorliegt, kommen alle Zeilen des rekursiven Ausdrucks zur Ausgabe.

Wichtig ist, dass der rekursive Teil eine Abbruchbedingung enthält, was hier durch die `WHERE`-Klauseln dargestellt wird. Im zweiten Beispiel ist der Wert 9 klein gewählt, weil die Fakultät sehr schnell wächst. Sofern das Datenbanksystem Zahlen beliebiger Genauigkeit unterstützt, kann man auch recht große Werte eintragen, erhält dann aber sehr lange Ergebnisse.

2.4 Rekursive CTEs zum Abfragen von Hierarchien

Rekursive CTEs mit Datenbankzugriff dienen allgemein dazu, in der Datenbank abgebildete Hierarchien darzustellen. Klassische Beispiele hierfür sind:

- Mitarbeiterhierarchien in Unternehmen
- Stücklisten für Produkte

Die Information über die Angestellten und ihre direkten Vorgesetzten kann man mit einer gewöhnlichen Abfrage anzeigen.

```
SELECT a.emp_id, a.emp_name, v.emp_name AS chef
FROM emp a LEFT JOIN emp v ON a.boss=v.emp_id
ORDER BY 1;
```

```
 emp_id | emp_name | chef
--------+----------+-------
      1 | Hans     |
      2 | Anna     | Hans
      3 | Udo      | Anna
      4 | Herbert  | Anna
      5 | Kai      | Anna
      6 | Marie    | Anna
      7 | Mona     | Anna
      8 | Urs      | Hans
      9 | Uta      | Urs
     10 | Ulli     | Urs
     11 | Udo      | Urs
     12 | Uwe      | Urs
     13 | Mandy    | Urs
     14 | Peter    | Hans
     15 | Petra    | Peter
     16 | Paul     | Peter
     17 | Pepe     | Peter
     18 | Patrick  | Peter
     19 | Pam      | Peter
     20 | Patti    | Peter
     21 | Marie    | Peter
```

2.4.1 Beispiel Mitarbeiterhierarchie

Sie kann auch nicht rein tabellarisch, sondern auch baumförmig dargestellt werden, wozu man eine rekursive CTE benötigt.

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

```
    WHERE boss IS NULL
  UNION ALL
    -- rekursive Unterabfrage
    SELECT a.emp_id, a.emp_name, a.boss,
     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;
```

```
 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
     14 | Peter   |    1 | Hans->Peter
     15 | Petra   |   14 | Hans->Peter->Petra
     16 | Paul    |   14 | Hans->Peter->Paul
     17 | Pepe    |   14 | Hans->Peter->Pepe
     18 | Patrick |   14 | Hans->Peter->Patrick
     19 | Pam     |   14 | Hans->Peter->Pam
     20 | Patti   |   14 | Hans->Peter->Patti
     21 | Marie   |   14 | Hans->Peter->Marie
```

Evtl. ist es angebracht, nicht nach der Id, sondern nach der Spalte `vollst` zu sortieren.

```
WITH RECURSIVE ang (emp_id, name, boss, vollst) AS (
    -- nicht-rekursiver Teil
    SELECT emp_id, emp_name, CAST (NULL AS INTEGER), emp_name
    FROM emp
    WHERE boss IS NULL
  UNION ALL
    -- rekursive Unterabfrage
```

```
    SELECT a.emp_id, a.emp_name, a.boss,
     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 vollst;

 emp_id |  name   | boss |        vollst
--------+---------+------+----------------------
      1 | Hans    |      | Hans
      2 | Anna    |    1 | Hans->Anna
      4 | Herbert |    2 | Hans->Anna->Herbert
      5 | Kai     |    2 | Hans->Anna->Kai
      6 | Marie   |    2 | Hans->Anna->Marie
      7 | Mona    |    2 | Hans->Anna->Mona
      3 | Udo     |    2 | Hans->Anna->Udo
     14 | Peter   |    1 | Hans->Peter
     21 | Marie   |   14 | Hans->Peter->Marie
     19 | Pam     |   14 | Hans->Peter->Pam
     18 | Patrick |   14 | Hans->Peter->Patrick
     20 | Patti   |   14 | Hans->Peter->Patti
     16 | Paul    |   14 | Hans->Peter->Paul
     17 | Pepe    |   14 | Hans->Peter->Pepe
     15 | Petra   |   14 | Hans->Peter->Petra
      8 | Urs     |    1 | Hans->Urs
     13 | Mandy   |    8 | Hans->Urs->Mandy
     11 | Udo     |    8 | Hans->Urs->Udo
     10 | Ulli    |    8 | Hans->Urs->Ulli
      9 | Uta     |    8 | Hans->Urs->Uta
     12 | Uwe     |    8 | Hans->Urs->Uwe
```

2.4.2 Beispiel Stückliste

Als weiteres Beispiel sei hier eine Stückliste angegeben, wozu es eine Tabelle `parts` mit den Daten für die fertigen Produkte, die Baugruppen und die Einzelteile gibt. Wie diese jeweils zusammengesetzt werden, steht in der Tabelle `sub_part`, d. h. dort ist jeweils angegeben, aus welchen Teilen in welcher Stückzahl sich eine Baugruppe bzw. ein Endprodukt zusammensetzt. Die Einzelteile, die eingekauft werden, bestehen nicht aus weiteren Teilen, während die Endprodukte in keine weiteren Teile Eingang finden.

Wie finden wir eine Liste unserer Verkaufsprodukte (wir handeln nicht mit Baugruppen oder Einzelteilen)?

```
SELECT partno, descr
FROM part
WHERE partno NOT IN (
  SELECT subpartno FROM sub_part
);
```

```
 partno |     descr
--------+---------------
     10 | Sitzgruppe
     15 | Tennisschläger
```

Welche Teile kaufen wir ein, um daraus unsere Produkte herzustellen? Die erste Variante prüft, ob die `partno` auch nicht als zusammengesetztes Teil in der Tabelle `sub_part` enthalten ist. Die zweite Variante prüft, ob es in `sub_part` keine Zeile gibt, bei der die `partno` aus dem Haupt-`SELECT` mit der `partno` aus dem Sub-`SELECT` übereinstimmt. Dies ist ein „korreliertes Sub-Query“ und daher nicht performance-optimal. Die dritte Variante zählt die Einzelteile, aus denen sich ein Teil zusammensetzt. Setzt es sich aus 0 Teilen zusammen, ist es ein Einzelteil und keine Baugruppe.

```
SELECT partno, descr
FROM part
WHERE partno NOT IN (
  SELECT partno FROM sub_part
) ORDER BY partno;
```

```
SELECT partno, descr
FROM part
WHERE NOT EXISTS (
  SELECT * FROM sub_part
  WHERE part.partno = sub_part.partno
) ORDER BY partno;
```

```
SELECT partno, descr
FROM part NATURAL LEFT JOIN sub_part
GROUP BY partno
HAVING COUNT(quantity) = 0
ORDER BY partno;
```

```
 partno |     descr
--------+---------------
      3 | Tischplatte
      5 | Schraube
      6 | Sitz
      7 | Lehne
      9 | Lehnenhalter
     11 | Tischbeinrohr
```

```
     12 | Tischbeinkappe
     13 | Stuhlbeinrohr
     14 | Stuhlbeinkappe
     16 | Rahmen
     17 | Sehne
     18 | Griffband
```

Welche Baugruppen gibt es, d. h. welche Teile setzen sich aus mehreren anderen Teilen zusammen, sind aber noch keine verkaufsfertigen Endprodukte? Diese existieren also sowohl als Baugruppe (kein Einzelteil) als auch als Bestandteil anderer Teile (kein Endprodukt). Natürlich könnten wir auch Stuhl und Tisch einzeln verkaufen, aber sie gehen eben auch in das Produkt „Sitzgruppe“ ein, was die Abfrage dazu veranlasst, sie als Nicht-Endprodukte anzusehen. Möchte man das anders, müsste man eine entsprechende Information bei den verkaufsfähigen Teilen hinterlegen, z. B. in Form eines `BOOLEAN`-Felds.

```
SELECT partno, descr
FROM part
WHERE partno IN (
  SELECT subpartno FROM sub_part
) AND partno IN (
  SELECT partno    FROM sub_part
);
```

```
 partno |   descr
--------+-----------
      1 | Tisch
      2 | Stuhl
      4 | Tischbein
      8 | Stuhlbein
```

Welche Teile benötigen wir in welcher Anzahl, um ein bestimmtes Produkt herzustellen? Die rekursive Stückliste bringt es an den Tag. In der Spalte `main` sehen wir die Nummer des Endprodukts, in der Spalte `partno` die Nummer des aktuellen Endprodukts, der Baugruppe bzw. des Teils. Die Spalten `subpartno` und `quantity` geben an, welches Teil bzw. welche Baugruppe in welcher Anzahl in das aktuelle Teil eingeht. Die Spalte `level` gibt die Rekursionstiefe an.

```
WITH RECURSIVE included (main, partno, subpartno, quantity, level) AS (
    -- nur die Endprodukte im nicht-rekursiven Teil der Abfrage
    SELECT partno AS main, partno, subpartno, quantity, 1 AS level
      FROM sub_part
      WHERE partno NOT IN (SELECT subpartno FROM sub_part)
  UNION ALL
```

```
    -- über die Zuordnung zu untergeordneten Teilen verknüpfen
    SELECT i.main, i.subpartno, s.subpartno, s.quantity, level + 1
    FROM included i JOIN sub_part s ON i.subpartno = s.partno
)
-- Haupt-Abfrage fragt gesamte CTE ab
SELECT * FROM included;
```

```
 main | partno | subpartno | quantity | level
------+--------+-----------+----------+-------
   15 |     15 |        16 |        1 |     1
   15 |     15 |        17 |        2 |     1
   15 |     15 |        18 |        1 |     1
   10 |     10 |         1 |        1 |     1
   10 |     10 |         2 |        4 |     1
   10 |      1 |         3 |        1 |     2
   10 |      1 |         4 |        4 |     2
   10 |      1 |         5 |       16 |     2
   10 |      2 |         5 |       20 |     2
   10 |      2 |         6 |        1 |     2
   10 |      2 |         7 |        1 |     2
   10 |      2 |         8 |        4 |     2
   10 |      2 |         9 |        2 |     2
   10 |      4 |        11 |        1 |     3
   10 |      4 |        12 |        1 |     3
   10 |      8 |        13 |        1 |     3
   10 |      8 |        14 |        1 |     3
```

Tatsächlich benötigen wird die Zwischenprodukte nicht, weil die Einkäufer diese nicht bestellen müssen, so dass wir diese aus der Liste herausnehmen können. Ermittelt werden müssen sie allerdings, weil sonst die weiter unten im Baum liegenden Knoten nicht gefunden werden können. Zudem müssen wir die Anzahlen entsprechend multiplizieren und am Ende summieren, um feststellen zu können, wie viele von den Einzelteilen benötigt werden – hier gut zu sehen am Beispiel der Schrauben. Ein Stuhl braucht laut Liste 20 Schrauben, ein Tisch 16. Zur Sitzgruppe gehören 1 Tisch und 4 Stühle, so dass insgesamt 96 Schrauben benötigt werden.

```
WITH RECURSIVE included (main, partno, subpartno, quantity) AS (
  -- 1. CTE mit Hierarchie ähnlich oben, aber ohne die Spalte Level,
  -- dafür aber mit Multiplikation der Anzahlen im rekursiven Teil
  SELECT partno AS main, partno, subpartno, quantity
    FROM sub_part
    WHERE partno NOT IN (SELECT subpartno FROM sub_part)
  UNION ALL
```

```
  SELECT i.main, i.subpartno, s.subpartno, i.quantity * s.quantity
  FROM included i JOIN sub_part s ON i.subpartno = s.partno
), partlist (main, partno, descr, quantity) AS (
  -- 2. CTE zum Hinzufügen der Teilebezeichnungen und zum
  -- Summieren über die Teile
  SELECT main, subpartno, descr, SUM(quantity) AS quantity
  FROM included i JOIN part p ON i.subpartno = p.partno
  WHERE subpartno NOT IN (SELECT partno FROM sub_part)
  GROUP BY main, subpartno, descr
)
-- Haupt-Select eingeschränkt auf die Sitzgruppe
SELECT * from partlist
WHERE main = (SELECT partno FROM part WHERE descr = 'Sitzgruppe');
```

main	partno	descr	quantity
10	3	Tischplatte	1
10	5	Schraube	96
10	6	Sitz	4
10	7	Lehne	4
10	9	Lehnenhalter	8
10	11	Tischbeinrohr	4
10	12	Tischbeinkappe	4
10	13	Stuhlbeinrohr	16
10	14	Stuhlbeinkappe	16

Die Anzahlen aus dieser Tabelle sind dann nur noch mit der Anzahl der zu fertigenden Sitzgruppen zu multiplizieren.

3 OLAP mit SQL

3.1 Zweck

OLAP steht für „Online Analytical Processing“ und ist eigentlich die Domäne von Data Warehouses, während SQL-Datenbanken ihren Schwerpunkt auf OLTP – Online Transaction Processing – legen. OLTP dient dazu, normalisiert gespeicherte, volatile Daten zu bearbeiten und auszugeben. OLAP hingegen dient dazu, nicht zwingend normalisiert gespeicherte (teilweise sogar stark denormalisierte) historische Datenbestände auszuwerten. Weil die historischen Datenbestände sich nicht mehr verändern, treten die bei der Bearbeitung unnormalisiert gespeicherter Daten üblichen Probleme nicht auf. Oft werden OLAP-Daten in einem sogenannten Schneeflockenschema abgelegt[3].

[3] https://de.wikipedia.org/wiki/Schneeflockenschema

3.2 Gruppierungsoperationen

Im SQL bis 1999 gab es zwar die Klausel `GROUP BY`, aber es konnte immer nur nach einem Kriterium gruppiert werden. Der Zusatz `GROUPING SETS` ermöglicht nun, mehrere Gruppierungen gleichzeitig in einer einzigen Abfragen durchzuführen, und auch, ein Gesamtergebnis zu bekommen.

Um zu ermitteln, wie viele Personen in den einzelnen Abteilungen arbeiten, wie viele Personen bestimmte Aufgabenbezeichnungen haben und gleichzeitig, wie viele Mitarbeiter es insgesamt im Unternehmen gibt, verwendet man folgende Abfrage:

```
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;
```

Es werden Abteilungsnummer und -bezeichnung, Jobnummer und -bezeichnung sowie die Anzahl als Spalten erfragt. Um an all diese Spalten zu kommen, müssen die Tabellen wie gezeigt mit `JOIN`s verbunden werden.

```
 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
```

Um nicht so viele `NULL`-Werte zu erhalten, kann man die Bezeichnungsspalten zusammenfassen. Die Abteilungs- und Tätigkeitsbezeichnungen landen dann in einer Spalte, weil bei der Aufteilung auf zwei ohnehin eine der beiden immer leer ist. Zusätzlich wird die Gesamtergebniszeile ordentlich benannt.

```
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), ())
ORDER BY dept_id, job_id;
```

```
 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
```

Um auch noch alle Zwischen- und Kombinationsergebnisse zu erhalten, verwendet man statt `GROUPING SETS` das Schlüsselwort `ROLLUP`. So erhält man die Mitarbeiteranzahl pro Abteilung und innerhalb der Abteilungen pro Tätigkeit.

Weil die Gesamtsumme ohnehin enthalten ist, gibt man die leeren Klammern, die oben

den dritten Eintrag bei `GROUPING SETS` gebildet hat, nicht mehr separat an.

```
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;
```

```
 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
```

Möchte man zusätzlich auch noch alle Anzahlen pro Tätigkeit – auch abteilungsübergreifend wie beispielsweise bei Bürofachkräften – erhalten, so ersetzt man `ROLLUP` durch `CUBE`.

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

```
 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
  |             |      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
```

3.3 Window Functions

Es gibt SQL-Fachleute, die davon sprechen, dass es ein SQL vor der Einführung der Window-Funktionen gab und jetzt eines nach der Einführung derselben. [4]

Tatsächlich ermöglichen die Window-Funktionen Abfragen, die ansonsten nur unter allergrößten Anstrengungen und mit vielen Klimmzügen erreichbar wären – wenn überhaupt. Leider hält die Verbreitung ihrer Anwendung bislang gar nicht Schritt mit dem Umfang an Vorteilen, die sie bieten. Dabei bieten mittlerweile sehr viele Datenbanksysteme Window-Funktionen an.

Die Grundidee ist eigentlich einfach: Es gibt ein Fenster (window), das nicht wie bislang nur eine Zeile hoch ist, sondern größer ist. Auf diese Weise „sieht“ man bei der Verarbeitung einer Zeile auch die Daten anderer Zeilen. Das können alle Zeilen sein, alle Zeilen vom Beginn der Tabelle bis zur aktuellen Zeile, alle Zeilen ab der aktuellen Zeile, die aktuelle Zeile und eine bestimmte Anfang von Zeilen davor und/oder danach usw.

Über dieses Fenster kann man dann Aggregatfunktionen ausführen, die sich normalerweise nur auf die gesamte Tabelle oder auf eine Gruppe (mittels `GROUP BY`) beziehen können, wodurch die Daten der einzelnen Zeile verloren gehen. Nicht so bei den Fenstern: Die Aggregatfunktionen beziehen sich auf das entsprechend beschriebene Fenster, ohne dass die

[4] Dimitri Fontaine: *There was SQL before window functions and SQL after window functions: that's how powerful this tool is.* http://tapoueh.org/tags/window-functions [2016-07-24]

Daten der aktuellen Zeile verloren gehen.

3.4 Syntax der `OVER()`-Klausel

Die `OVER()`-Klausel gibt an, über welche Fenster die jeweiligen Aggregatfunktionen ihre Werte berechnen. Sie hat immer eine Klammer, die folgende Elemente enthalten kann:

1. *window partition clause* – `PARTITION BY` gibt an, aufgrund welchen Kriteriums die Daten partitioniert werden.

2. *window order clause* – `ORDER BY` gibt an, wie die Daten in den Partitionen sortiert werden. Das ist für viele Funktionen notwendig.

3. *window frame clause* – `ROWS` ... oder `RANGE` ... gibt an, welche Zeilen aus der Partition bei der Wertermittlung herangezogen werden. Wenn man diese Klausel weglässt, sind es alle, aber das kann unerwünscht sein, so dass man beispielsweise alle vor oder ab der aktuellen Zeile oder eine bestimmte Anzahl vor und nach der aktuellen Zeile wählen kann.

Alle Elemente können auch fehlen, was zu einer leeren Klammer hinter `OVER` führt. Dann werden alle Zeilen der Tabelle zur Berechnung herangezogen, ähnlich wie das bei den klassischen Aggregatfunktionen der Fall ist. Sobald eine `ORDER BY`-Klausel vorhanden ist, ist die fehlende Frame Clause gleichbedeutend mit `RANGE UNBOUNDED PRECEDING`.

3.5 Verwendung von `ORDER BY`

Um auf das Beispiels der Summierung der ersten 100 Zahlen zurück zu kommen (siehe Abschnitt 2.3 auf Seite 6), hier eine andere Lösung[5]:

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

Die PostgreSQL-spezifische Funktion `generate_series()` liefert eine Tabelle der Zahlen von der Unter- zur Obergrenze. Bei anderen Datenbanksystemen kann man sich ggf. mit einer realen Tabelle `nums` mit einer Spalte `x` behelfen, die man mit einer ausreichenden Menge aufsteigender Zahlen füllt und dann mittels `SELECT x FROM nums ORDER BY 1 LIMIT 100` abfragt, um die ersten 100 Zahlen zu erhalten.

Bislang wurden noch keine Window-Funktionen benötigt. Möchte man jedoch alle Zwischenergebnisse erhalten, so aggregiert man nicht (nur) über die Gesamttabelle, sondern über das Fenster zwischen dem Tabellenbeginn (unbegrenzte Zeilen vor der aktuellen Zeile) und der aktuellen Zeile.

[5] Die Funktion `generate_series()` ist PostgreSQL-spezifisch und muss ggf. durch eine Hilfstabelle mit den Zahlen 1 bis 100 ersetzt werden

```
SELECT SUM(x) OVER
  (ORDER BY x ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM generate_series(1, 100) AS t(x);

 sum
------
    1
    3
    6
   10
   15
  ...
 4851
 4950
 5050
```

Wir fragen also die Summe der (einzigen) Spalte `x` ab, wobei das Schlüsselwort `OVER` angibt, dass nicht über die gesamte Tabelle aggregiert wird. Das `ORDER BY` ist bei realen Tabellen – also nicht per Funktion generierten wie hier – notwendig, um ein reproduzierbares Ergebnis zu erhalten. Nebeneffekt des `ORDER BY` ist eine Veränderung des Fensters von der Gesamttabelle zu `UNBOUNDED PRECEDING`. Mit leerer `OVER()`-Klammer wird in jeder Zeile 5050 ausgegeben.

Die Klausel `BETWEEN` gibt an, welche Zeilen im Aggregat zu berücksichtigen sind, das sind hier alle Zeilen zuvor `UNBOUNDED PRECEDING` bis zur aktuellen Zeile `CURRENT ROW`

```
SELECT x,
  SUM(x) OVER (ORDER BY x
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS summe_bis,
  SUM(x) OVER (ORDER BY x
    ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS summe_ab
FROM generate_series(1, 100) AS t(x);

  x  | summe_bis | summe_ab
-----+-----------+----------
   1 |         1 |     5050
   2 |         3 |     5049
   3 |         6 |     5047
   4 |        10 |     5044
   5 |        15 |     5040
 ...
  98 |      4851 |      297
  99 |      4950 |      199
 100 |      5050 |      100
```

Hier wird zusätzlich die laufende Zeilennummer ausgegeben (`x`) und nicht nur die laufende Summe der Zahlen von 1 bis zur laufenden Zahl, sondern auch die Summe der laufenden Zahl bis 100 ausgegeben. Es kann also problemlos mehrere Fenster (Windows) geben, über die Aggregate gebildet werden.

Die `BETWEEN`-Klausel für die dritte Spalte gibt an, dass alle Zeilen ab der aktuellen Zeile (`CURRENT ROW`) ohne Begrenzung (`UNBOUNDED FOLLOWING`) für das Aggregat verwendet werden sollen. Dies kann man auch kürzer fassen: `ROWS UNBOUNDED PRECEDING` genügt.

3.6 Verwendung von `PARTITION BY`

Möchte man mehrere Untergruppen bilden – beispielsweise über Abteilungen oder Jahre aggregieren –, so verwendet man die Klausel `PARTITION BY`. Auf diese Weise wird für jede Untergruppe ein Aggregatwert gebildet und ausgegeben.

Hier werden für jede Abteilung die Anzahl Mitarbeiter, die Gehaltsspanne und der Durchschnittswert der Gehälter ermittelt.

```
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);
```

Weil das ein wenig umständlich erscheint, die `PARTITION BY`-Klausel jedes Mal komplett zu wiederholen, kann man diese benennen und auf den Namen Bezug nehmen.

```
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);
```

Dies hat mehrere Vorteile: Es spart Schreibarbeit und vermeidet Schreibfehler. Man sieht auf den ersten Blick, dass überall exakt das gleiche Fenster verwendet wird.

3.7 Verwendung von ROWS und RANGE

In den Beispielen der vorigen Abschnitte wurde ROWS bereits verwendet. Ohne eine Einschränkung, wie ROWS und RANGE sie darstellen, werden alle Zeilen verwendet, sofern keine ORDER BY-Klausel vorhanden ist, weil dann alle Zeilen Peers der aktuellen Zeile sind. Sobald eine ORDER BY-Klausel verwendet wird, ist RANGE UNBOUNDED PRECEDING der default-Wert. Hinter den beiden Schlüsselwörtern ROWS und RANGE darf stehen:

- UNBOUNDED PRECEDING. Dann werden alle Zeilen vom Tabellenbeginn bis zur aktuellen Zeile verwendet. Dies ist gleichbedeutend mit der ausführlicheren Klausel BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.
- BETWEEN ... AND ..., wobei an der Stelle der ... jeweils stehen kann:
  - UNBOUNDED PRECEDING
  - *n* PRECEDING
  - CURRENT ROW
  - *n* FOLLOWING
  - UNBOUNDED FOLLOWING

  Hier ist ***n*** eine Anzahl Zeilen bezogen auf die aktuelle Zeile.

Bei Verwendung von RANGE sind bei CURRENT ROW alle Zeilen eingeschlossen, welche – bezogen auf ORDER BY – denselben Wert wie die aktuelle Zeile haben, bei ROW ist exakt bei der aktuellen Zeile Schluss.

3.8 Ordnungsfunktionen RANK und DENSE_RANK

Die parameterlosen Ordnungsfunktionen (ordinal functions) geben eine Ordnung der Werte innerhalb einer Partition oder eines Fensters zurück. Hierzu benötigen sie eine ORDER BY-Klausel.

ROW_NUMBER() gibt die laufende Nummer der Zeile innerhalb eines Fensters zurück, sofern in der zugehörigen OVER-Klausel etwas steht, ansonsten die laufende Nummer über das gesamte Ergebnis. Die durch die ORDER BY-Angabe in der WINDOW-Definition der OVER-Klausel erzeugte Reihenfolge kann durch eine ORDER BY-Klausel über die Gesamttabelle geändert werden.

RANK() gibt den Rang einer Zeile innerhalb eines Fensters zurück, welches durch die OVER-Klausel mit ORDER BY definiert wird.

```
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;

 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
```

Hier sind die Mitarbeiter nach Abteilung und innerhalb der Abteilung nach ihrem Gehalt absteigend angegeben. Mitarbeiter mit Rang 1 verdienen innerhalb ihrer Abteilung das meiste. Solche mit gleichem Gehalt kommen auf denselben Rang, wobei danach in der Nummerierung eine Lücke entsteht. Möchte man diese Lücke nicht, so verwendet man `DENSE_RANK`, hier in der Spalte `d_rang` gezeigt.

```
SELECT emp_id, emp_name, dept_name,
    CAST(PERCENT_RANK() OVER abt * 100 AS NUMERIC(10,2)) AS percent_rank,
    CAST(CUME_DIST()    OVER abt * 100 AS NUMERIC(10,2)) AS cume_dist
FROM emp JOIN dept USING(dept_id) NATURAL JOIN job
WINDOW abt as (PARTITION BY dept_id ORDER BY salary);

 emp_id | emp_name |  dept_name  | percent_rank | cume_dist
--------+----------+-------------+--------------+-----------
```

```
 1 | Hans     | owner       |          0.00 |    100.00
 5 | Kai      | sales       |          0.00 |     16.67
 6 | Marie    | sales       |         20.00 |     33.33
 7 | Mona     | sales       |         40.00 |     50.00
 3 | Udo      | sales       |         60.00 |     66.67
 4 | Herbert  | sales       |         80.00 |     83.33
 2 | Anna     | sales       |        100.00 |    100.00
11 | Udo      | development |          0.00 |     16.67
10 | Ulli     | development |         20.00 |     33.33
13 | Mandy    | development |         40.00 |     50.00
 9 | Uta      | development |         60.00 |     83.33
12 | Uwe      | development |         60.00 |     83.33
 8 | Urs      | development |        100.00 |    100.00
21 | Marie    | production  |          0.00 |     25.00
18 | Patrick  | production  |          0.00 |     25.00
17 | Pepe     | production  |         28.57 |     37.50
19 | Pam      | production  |         42.86 |     62.50
16 | Paul     | production  |         42.86 |     62.50
15 | Petra    | production  |         71.43 |     75.00
20 | Patti    | production  |         85.71 |     87.50
14 | Peter    | production  |        100.00 |    100.00
```

Beide Funktionen liefern einen Wert zwischen 0 und 1 zurück, so dass es – gerade beim `PERCENT_RANK()` – sinnvoll erscheint, diese mit 100 zu multiplizieren, um einen Prozentwert zu erhalten. Um nicht zu viele Nachkommastellen zu erhalten, verwenden wir die `CAST`-Funktion auf `NUMERIC(10,2)`.

`PERCENT_RANK()` berechnet sich als $(rk-1)/(nr-1)$, während `CUME_DIST()` als np/nr berechnet wird. Hier ist rk der `RANK()`-Wert, nr die Anzahl Zeilen im aktuellen Fenster und np die Anzahl Zeilen vor der aktuellen oder mit gleichem Wert.

3.9 Nachbarn mit `LEAD` und `LAG`

An benachbarte Zeilen kommt man mittels der Funktionen `LEAD` und `LAG`. Man blickt sozusagen auf vorige Zeilen zurück oder auf folgende Zeilen voraus. Auf diese Weise kann man bei Zeitreihendaten leicht Vergleiche zwischen Perioden machen, beispielsweise einen Wert mit dem Vorjahreswert vergleichen.

Tatsächlich muss es sich nicht um unmittelbare Nachbarzeilen handeln, auf die man zugreift, sondern der Abstand kann angegeben werden.

```
SELECT emp_id, emp_name, dept_name,
  salary                AS gehalt,
  LAG(salary)  OVER abt AS besser,
  LEAD(salary) OVER abt AS schlechter
```

```
FROM emp JOIN dept USING(dept_id) NATURAL JOIN job
WINDOW abt as (PARTITION BY dept_id ORDER BY salary DESC);

 emp_id | emp_name |  dept_name  | gehalt  | besser  | schlechter
--------+----------+-------------+---------+---------+------------
      1 | Hans     | owner       |         |         |
      2 | Anna     | sales       | 3800.00 |         |    2750.00
      4 | Herbert  | sales       | 2750.00 | 3800.00 |    2700.00
      3 | Udo      | sales       | 2700.00 | 2750.00 |    2300.00
      7 | Mona     | sales       | 2300.00 | 2700.00 |    2200.00
      6 | Marie    | sales       | 2200.00 | 2300.00 |    2000.00
      5 | Kai      | sales       | 2000.00 | 2200.00 |
      8 | Urs      | development | 3800.00 |         |    2600.00
      9 | Uta      | development | 2600.00 | 3800.00 |    2600.00
     12 | Uwe      | development | 2600.00 | 2600.00 |    2200.00
     13 | Mandy    | development | 2200.00 | 2600.00 |    2100.00
     10 | Ulli     | development | 2100.00 | 2200.00 |    2000.00
     11 | Udo      | development | 2000.00 | 2100.00 |
     14 | Peter    | production  | 3500.00 |         |    2600.00
     20 | Patti    | production  | 2600.00 | 3500.00 |    2500.00
     15 | Petra    | production  | 2500.00 | 2600.00 |    2400.00
     19 | Pam      | production  | 2400.00 | 2500.00 |    2400.00
     16 | Paul     | production  | 2400.00 | 2400.00 |    2300.00
     17 | Pepe     | production  | 2300.00 | 2400.00 |    2200.00
     21 | Marie    | production  | 2200.00 | 2300.00 |    2200.00
     18 | Patrick  | production  | 2200.00 | 2200.00 |
```

Wir erfahren also, wer innerhalb derselben Abteilung der mit dem nächsthöheren und der mit dem nächstniedrigeren Gehalt ist. Der Höchstverdienende hat natürlich niemanden, der mehr verdient; genauso wie der am wenigsten Verdienende keinen hat, der weniger verdient. Das zeigt sich durch die `NULL`-Werte.

Solche Vergleiche mit benachbarten oder noch weiter entfernten Daten sind insbesondere bei Zeitreihen beliebt, hier am Beispiel von Produktionsmengenwerten über viele Jahre gezeigt. Neben dem Jahr und den aktuellen Produktionswerten werden die Veränderungen gegenüber dem Vorjahr und dem 5 Jahre alten Wert absolut und in Prozent aufgeführt. Der Einfachheit wegen wird hier eine Common Table Expression (siehe Abschnitt 2 auf Seite 4) eingesetzt.

```
WITH t AS (SELECT year, prod,
  prod - LAG(prod)    OVER (ORDER BY year) AS ch_prev,
  prod - LAG(prod, 5) OVER (ORDER BY year) AS ch_5
  FROM v_oil)
SELECT
```

```
  *,
  CAST(ch_prev * 100.0 / prod AS NUMERIC(10,2)) AS percent_prev,
  CAST(ch_5    * 100.0 / prod AS NUMERIC(10,2)) AS percent_5
FROM t
ORDER BY year;

 year | prod  | ch_prev | ch_5  | percent_prev | percent_5
------+-------+---------+-------+--------------+-----------
 1965 | 18683 |         |       |              |
 1966 | 20322 |    1639 |       |         8.07 |
 1967 | 21750 |    1428 |       |         6.57 |
 1968 | 23491 |    1741 |       |         7.41 |
 1969 | 24986 |    1495 |       |         5.98 |
 1970 | 27162 |    2176 |  8479 |         8.01 |     31.22
 1971 | 29576 |    2414 |  9254 |         8.16 |     31.29
 1972 | 31706 |    2130 |  9956 |         6.72 |     31.40
 ...
 2006 | 39655 |     254 |  2458 |         0.64 |      6.20
 2007 | 39102 |    -553 |  3046 |        -1.41 |      7.79
 2008 | 39707 |     605 |  1986 |         1.52 |      5.00
 2009 | 38408 |   -1299 |  -781 |        -3.38 |     -2.03
 2010 | 39513 |    1105 |   112 |         2.80 |      0.28
```

3.10 Spitzenreiter und Schlusslicht mit FIRST_VALUE und LAST_VALUE

Hiermit kann man den jeweils größten und kleinsten Wert aus einem Fenster ermitteln. Das ist besonders hilfreich, wenn man für die einzelnen Zeilen ermitteln möchte, wie viel geringer ihr Wert gegenüber dem größten bzw. wie viel höher ihr Wert gegenüber dem kleinsten Wert ist.

```
WITH t AS (
  SELECT emp_id, emp_name, dept_name,
    salary                AS gehalt,
    salary * 100.0 / FIRST_VALUE(salary) OVER abt   AS von_max,
    salary * 100.0 / LAST_VALUE(salary)  OVER abt_f AS von_min
  FROM emp JOIN dept USING(dept_id)
  WINDOW
    abt   AS (PARTITION BY dept_id ORDER BY salary DESC),
    abt_f AS (PARTITION BY dept_id ORDER BY salary DESC
              ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
)
SELECT emp_id, emp_name, dept_name, gehalt,
  CAST(von_max AS NUMERIC(10,2)),
```

```
  CAST(von_min AS NUMERIC(10,2))
FROM t;
```

```
 emp_id | emp_name |  dept_name  | gehalt  | von_max | von_min
--------+----------+-------------+---------+---------+---------
      1 | Hans     | owner       |         |         |
      2 | Anna     | sales       | 3800.00 |  100.00 |  190.00
      4 | Herbert  | sales       | 2750.00 |   72.37 |  137.50
      3 | Udo      | sales       | 2700.00 |   71.05 |  135.00
      7 | Mona     | sales       | 2300.00 |   60.53 |  115.00
      6 | Marie    | sales       | 2200.00 |   57.89 |  110.00
      5 | Kai      | sales       | 2000.00 |   52.63 |  100.00
      8 | Urs      | development | 3800.00 |  100.00 |  190.00
      9 | Uta      | development | 2600.00 |   68.42 |  130.00
     12 | Uwe      | development | 2600.00 |   68.42 |  130.00
     13 | Mandy    | development | 2200.00 |   57.89 |  110.00
     10 | Ulli     | development | 2100.00 |   55.26 |  105.00
     11 | Udo      | development | 2000.00 |   52.63 |  100.00
     14 | Peter    | production  | 3500.00 |  100.00 |  159.09
     20 | Patti    | production  | 2600.00 |   74.29 |  118.18
     15 | Petra    | production  | 2500.00 |   71.43 |  113.64
     19 | Pam      | production  | 2400.00 |   68.57 |  109.09
     16 | Paul     | production  | 2400.00 |   68.57 |  109.09
     17 | Pepe     | production  | 2300.00 |   65.71 |  104.55
     21 | Marie    | production  | 2200.00 |   62.86 |  100.00
     18 | Patrick  | production  | 2200.00 |   62.86 |  100.00
```

Neben dem ersten und dem letzten Wert kann auch der n-te Wert bestimmt werden mit der Funktion `NTH_VALUE(`*`spalte`*`, `*`n`*. Wer ist also der zweitälteste im Betrieb?

```
SELECT * FROM emp
WHERE birthdate IN (
  SELECT NTH_VALUE(birthdate, 2) OVER (ORDER BY birthdate) FROM emp
);
```

Natürlich ist es auch leicht möglich, die drei Jüngsten zu ermitteln, aber dazu benötigen wir gar kein „Advanced SQL“, sondern kommen mit Standardmitteln aus:

```
SELECT * FROM emp
WHERE birthdate IN (
  SELECT birthdate
  FROM emp
  ORDER BY birthdate DESC
  LIMIT 3
);
```

3.11 Geburtstagsfeier ausrichten

Nun zu einem etwas komplexeren Beispiel, bei dem mehrere der bislang erläuterten Funktionen verwendet werden.

In der Firma ist es üblich, dass Kollegen Geburtstagsfeiern 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 Geburtstag hat. Natürlich muss auch angegeben sein, an welchem Tag und Monat die Feier stattfindet. Dieses Ausrichten der Geburtstagsfeier wird innerhalb der einzelnen Abteilungen geregelt. Die einzige „Abteilung“ mit einer Person – der Inhaber – macht nicht mit, weil er alleine feiern müsste.

```
WITH t1 AS (
  -- t1 ermittelt das Geburtsdatum ohne Jahr als Spalte bd
  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 (
  -- t2 holt in die Spalte geb_bd den Geburtstag des Geburtstagskinds.
  -- Wer als letztes im Jahr Geburtstag hat, richtet für den aus, der
  -- als erstes im Folgejahr Geburtstag hat.
  SELECT emp_id, emp_name, dept_id, dept_name, bd,
    COALESCE(LEAD (bd) OVER dept, FIRST_VALUE(bd) OVER dept) AS geb_bd
    FROM t1
    WINDOW dept AS (PARTITION BY dept_id ORDER BY bd)
)
-- Self-JOIN von t2, so dass Name des Geburtstagskinds und Name
-- des Ausrichters nebeneinander stehen.
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 AS geb_bd
-- a = Angestellter, g = Geburtstagskind
FROM t2 a JOIN t2 g ON a.dept_id = g.dept_id AND a.geb_bd = g.bd
  WHERE a.emp_id <> g.emp_id -- Abteilungen mit 1 Person machen nicht mit
ORDER BY dept_name, a.bd;
```

```
 emp_id | emp_name |  dept_name  |  bd   | geb_id | geb_name | geb_bd
--------+----------+-------------+-------+--------+----------+--------
     11 | Ulf      | development | 02-28 |     12 | Uwe      | 03-25
     12 | Uwe      | development | 03-25 |     10 | Ulli     | 06-18
     10 | Ulli     | development | 06-18 |      9 | Uta      | 09-17
      9 | Uta      | development | 09-17 |      8 | Urs      | 09-23
      8 | Urs      | development | 09-23 |     13 | Ulani    | 12-08
     13 | Ulani    | development | 12-08 |     11 | Ulf      | 02-28
     18 | Patrick  | production  | 01-17 |     21 | Paola    | 02-13
```

```
 21 | Paola    | production | 02-13 |     15 | Petra    | 02-27
 15 | Petra    | production | 02-27 |     16 | Paul     | 04-13
 16 | Paul     | production | 04-13 |     17 | Pepe     | 05-03
 17 | Pepe     | production | 05-03 |     20 | Patti    | 10-12
 20 | Patti    | production | 10-12 |     19 | Pam      | 11-05
 19 | Pam      | production | 11-05 |     14 | Peter    | 12-02
 14 | Peter    | production | 12-02 |     18 | Patrick  | 01-17
  3 | Udo      | sales      | 02-14 |      2 | Anna     | 03-04
  2 | Anna     | sales      | 03-04 |      4 | Herbert  | 04-24
  4 | Herbert  | sales      | 04-24 |      7 | Mona     | 05-14
  7 | Mona     | sales      | 05-14 |      6 | Mara     | 11-26
  6 | Mara     | sales      | 11-26 |      5 | Kai      | 12-08
  5 | Kai      | sales      | 12-08 |      3 | Udo      | 02-14
```

3.12 Zeitreihenauswertung und Einsatz von FILTER

Bei der Auswertung von Zeitreihen, beispielsweise Messergebnissen über längere Zeiträume, kann zeitgemäßes SQL eine große Hilfe sein und Daten passend aufbereiten. Als Beispiel sollen gemessene Temperaturen aus den letzten 10 Jahren dienen, die als einzelne Werte mit Zeitstempel vorliegen, wobei bis zu 24 Werte pro Tag vorliegen können. Die Tabelle sieht so aus.

```
CREATE TABLE gltemps (
    zeit    TIMESTAMP PRIMARY KEY,
    temp_c  REAL
);
```

Diese Werte sollen verdichtet werden zu Maximal-, Minimal- und Durchschnittstemperaturen pro Tag und pro Monat. Dies geschieht mit folgenden Views, die man ggf. auch materialisieren kann, da sich die historischen Werte niemals mehr ändern werden.

```
-- VIEW MIT MIN-, MAX- UND AVG-WERTEN PRO TAG
CREATE VIEW v_temp_day AS
SELECT
  CAST(zeit AS DATE)                AS zeit,
  CAST(MAX(temp_c) AS NUMERIC(3,1)) AS maxtemp,
  CAST(MAX(temp_c) AS NUMERIC(3,1)) AS mintemp,
  CAST(AVG(temp_c) AS NUMERIC(3,1)) AS avgtemp
FROM gltemps
GROUP BY CAST(zeit AS date);


-- VIEW MIT MIN-, MAX- UND AVG-WERTEN PRO MONAT
```

```
CREATE VIEW v_temp_month AS
SELECT
  CAST(DATE_TRUNC('month', zeit) AS DATE) AS month,
  CAST(MAX(temp_c) AS NUMERIC(3,1))       AS maxtemp,
  CAST(MIN(temp_c) AS NUMERIC(3,1))       AS mintemp,
  CAST(AVG(temp_c) AS NUMERIC(3,1))       AS avgtemp
FROM gltemps
GROUP BY DATE_TRUNC('month', zeit);
```

Die Funktion `DATE_TRUNC` ist nicht bei allen Datenbanksystemen vorhanden. Sie setzt beim „Abschneiden“ (truncate) auf den Monat den Tag auf 01 und die Uhrzeit auf 00 : 00. Anschließend wird der Wert nach `DATE` gecastet, so dass nur das Datum des Monatsersten übrig bleibt. Die `CAST`-Operationen nach `NUMERIC(3,1)` finden statt, damit die Zahlen nicht durch viele Nachkommastellen unübersichtlich werden. Bei höheren Genauigkeitsanforderungen lässt man das weg.

Möchte man die Daten unvollständiger Jahre nicht haben, also nur Daten für Jahre anzeigen, in denen Daten für alle 12 Monate vorliegen, so filtert man entsprechend:

```
-- VIEW MIT MIN-, MAX- UND AVG-WERTEN PRO MONAT
-- NUR MIT JAHREN, FÜR DIE ALLE 12 WERTE VORLIEGEN
DROP VIEW v_temp_month;
CREATE VIEW v_temp_month AS
SELECT
  CAST(DATE_TRUNC('month', zeit) AS DATE) AS month,
  CAST(MAX(temp_c) AS NUMERIC(3,1))       AS maxtemp,
  CAST(MIN(temp_c) AS NUMERIC(3,1))       AS mintemp,
  CAST(AVG(temp_c) AS NUMERIC(3,1))       AS avgtemp
FROM gltemps
WHERE EXTRACT(YEAR FROM zeit) IN (
  SELECT EXTRACT(YEAR FROM zeit)
  FROM gltemps
  GROUP BY EXTRACT(YEAR FROM zeit)
  HAVING COUNT(DISTINCT EXTRACT(MONTH FROM zeit)) = 12
)
GROUP BY DATE_TRUNC('month', zeit);
```

Mittels der bereits vorgestellten Window-Funktionen `LEAD` and `LAG` (siehe Abschnitt 3.9 auf Seite 23) kann man Werte mit Vor- und Folgejahreswerten vergleichen, indem man als Abstand 12 angibt.

```
-- VERGLEICH MIT VORJAHRESWERTEN DESSELBEN MONATS
SELECT month,
  LAG(avgtemp, 12)  OVER () AS vorjahr,
```

```
  avgtemp                    AS aktuell,
  LEAD(avgtemp, 12) OVER () AS folgejahr
FROM v_temp_month
ORDER BY 1;
```

Wie viel schöner wäre es, wenn man die Monatswerte eines jeden Jahres nebeneinander hätte – also links die Jahreszahl und daneben genau 12 weitere Spalten? Mit SQL kein Problem:

```
-- MONATSWERTE NEBENEINANDER
CREATE VIEW v_jahrestabelle AS
WITH t AS (
  SELECT EXTRACT(YEAR FROM month) AS year, avgtemp,
    EXTRACT(MONTH FROM month) as month
  FROM v_temp_month
)
SELECT year,
  MIN(avgtemp) FILTER (WHERE month =  1) AS jan,
  MIN(avgtemp) FILTER (WHERE month =  2) AS feb,
  MIN(avgtemp) FILTER (WHERE month =  3) AS mar,
  MIN(avgtemp) FILTER (WHERE month =  4) AS apr,
  MIN(avgtemp) FILTER (WHERE month =  5) AS mai,
  MIN(avgtemp) FILTER (WHERE month =  6) AS jun,
  MIN(avgtemp) FILTER (WHERE month =  7) AS jul,
  MIN(avgtemp) FILTER (WHERE month =  8) AS aug,
  MIN(avgtemp) FILTER (WHERE month =  9) AS sep,
  MIN(avgtemp) FILTER (WHERE month = 10) AS okt,
  MIN(avgtemp) FILTER (WHERE month = 11) AS nov,
  MIN(avgtemp) FILTER (WHERE month = 12) AS dez
FROM t
GROUP BY year;
```

```
SELECT * FROM v_jahrestabelle ORDER BY year ;
 year | jan  | feb  | mar | apr  | mai  | jun  | jul  | aug  | sep  | okt  | nov | dec
------+------+------+-----+------+------+------+------+------+------+------+-----+------
 2007 |  4.5 |  6.2 | 7.9 | 13.9 | 15.3 | 18.1 | 17.8 | 17.1 | 13.6 |  9.9 | 5.8 |  3.4
 2008 |  5.9 |  5.1 | 5.8 |  8.8 | 16.6 | 17.2 | 18.2 | 18.4 | 13.1 | 10.3 | 6.6 |  2.1
 2009 | -0.4 |  3.2 | 6.1 | 13.3 | 14.6 | 15.9 | 19.1 | 19.3 | 15.5 | 10.0 | 9.7 |  2.5
 2010 | -1.0 |  1.9 | 6.1 | 10.2 | 11.2 | 17.5 | 21.7 | 17.3 | 13.6 |  9.7 | 6.3 | -1.6
 2011 |  2.9 |  4.1 | 7.1 | 13.5 | 15.4 | 17.1 | 16.5 | 18.2 | 16.6 | 11.2 | 7.4 |  5.5
 2012 |  4.1 | -0.2 | 8.9 |  9.1 | 15.1 | 15.8 | 17.7 | 19.6 | 13.9 | 10.2 | 6.8 |  4.2
 2013 |  2.0 |  1.0 | 2.4 |  9.5 | 12.2 | 16.4 | 20.1 | 18.5 | 14.7 | 12.2 | 5.7 |  5.6
 2014 |  5.4 |  6.4 | 8.8 | 12.4 | 14.0 | 17.6 | 20.6 | 17.0 | 16.5 | 13.8 | 9.0 |  4.7
 2015 |  3.9 |  3.2 | 7.0 | 10.7 | 14.1 | 17.5 | 19.2 | 20.6 | 14.5 | 10.1 | 9.9 |  9.6
```

Es muss aus syntaktischen Gründen eine Aggregatfunktion angewendet werden, weil sonst `FILTER` nicht erlaubt ist. Da ohnehin nur ein einziger Wert durch den Filter kommt, ist es egal, ob man `MAX` oder `MIN` nimmt. `AVG` wäre auch möglich, aber es ist rechenintensiver und daher langsamer.

Nun, wo die Daten so gut aufbereitet sind, kann man auch die Daten noch leichter mit den Vorjahreswerten vergleichen – hier am Beispiel des Januars gezeigt.

```
-- VERGLEICH MIT DEM DURCHSCHNITT DER LETZTEN UND FOLGENDEN 2 JAHRE
-- (also über insgesamt 5 Jahre)
SELECT
  year,
  CAST(AVG(jan) OVER (ORDER BY year ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)
     AS NUMERIC(3,1)) as schnitt,
  jan
FROM v_jahrestabelle;

-- VERGLEICH MIT DEM DURCHSCHNITT DER LETZTEN UND FOLGENDEN 3 JAHRE
SELECT
  year,
  CAST(AVG(jan) OVER (ORDER BY year ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING)
    AS NUMERIC(3,1)) as dreidavor,
  CAST(AVG(jan) OVER (ORDER BY year ROWS BETWEEN 1 FOLLOWING AND 3 FOLLOWING)
    AS NUMERIC(3,1)) as dreidanach,
  jan
FROM v_jahrestabelle;
```

Dies gelingt deshalb so leicht, weil die Werte jetzt unmittelbar hintereinander stehen. In der vorigen Tabelle mit dem 12er-Abstand wäre das nur sehr umständlich möglich, indem man die einzelnen Werte mit mehreren `LEAD`- bzw. `LAG`-Aufrufen ermitteln und „von Hand“ den Durchschnitt errechnet.

3.13 Gruppeneinteilung mit `NTILE`

Die `NTILE(`*`n`*`)`-Funktion ermöglicht es festzustellen, in welchen von *`n`* Töpfen der Wert der jeweiligen Zeile fällt, wenn man den Gesamtwertebereich über möglichst gleich große Töpfe verteilt.

Es werden also *`n`* Töpfe gebildet. Die ersten `COUNT(*) /` *`n`* Werte fallen in den ersten Topf, danach möglichst gleich viele in den zweiten Topf usw. Wenn die Gesamtanzahl der Zeilen durch die Anzahl Töpfe teilbar ist, enthalten alle Töpfe gleich viele Zeilen – andernfalls wird so gut wie möglich verteilt.

Wir können beispielsweise die Monate der letzten Jahre einteilen nach Temperatur, indem wir 12 Töpfe bilden – für jeden Kalendermonat einen. Bei stets gleicher Temperaturverteilung wären alle Januare die kältesten, alle Julis die wärmsten, aber das stimmt

natürlich nicht. So mancher Februar kommt in den ersten Topf, weil er kälter war als ein Januar (nicht unbedingt der vor ihm), genauso wie mancher August wärmer war als ein Juli.

```
-- NTILE - EINTEILUNG DER MONATE IN 12 TÖPFE
SELECT month, avgtemp, NTILE(12)
  OVER (ORDER BY avgtemp)
FROM v_temp_month
ORDER BY 3;
```

```
   month    | avgtemp | ntile
------------+---------+-------
 2010-12-01 |    -1.6 |     1
 2010-01-01 |    -1.0 |     1
 2009-01-01 |    -0.4 |     1
 2012-02-01 |    -0.2 |     1
 2013-02-01 |     1.0 |     1
 2010-02-01 |     1.9 |     1
 2013-01-01 |     2.0 |     1
 2008-12-01 |     2.1 |     1
 2013-03-01 |     2.4 |     1
 2009-12-01 |     2.5 |     2
 2011-01-01 |     2.9 |     2
 2015-02-01 |     3.2 |     2
 2009-02-01 |     3.2 |     2
...
 2007-08-01 |    17.1 |    10
 2008-06-01 |    17.2 |    10
 2010-08-01 |    17.3 |    10
 2010-06-01 |    17.5 |    11
 2015-06-01 |    17.5 |    11
 2014-06-01 |    17.6 |    11
 2012-07-01 |    17.7 |    11
 2007-07-01 |    17.8 |    11
 2007-06-01 |    18.1 |    11
 2008-07-01 |    18.2 |    11
 2011-08-01 |    18.2 |    11
 2008-08-01 |    18.4 |    11
 2013-08-01 |    18.5 |    12
 2009-07-01 |    19.1 |    12
 2015-07-01 |    19.2 |    12
 2009-08-01 |    19.3 |    12
 2012-08-01 |    19.6 |    12
```

```
2013-07-01 |    20.1 |    12
2015-08-01 |    20.6 |    12
2014-07-01 |    20.6 |    12
2010-07-01 |    21.7 |    12
```

Es wird der Monat angezeigt, die Durchschnittstemperatur und der Topf, in den er fällt.