Datenbank-Programmierung

Kapitel 6: Top-N Anfragen und Fenster-Anfragen

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2021

http://www.informatik.uni-halle.de/~brass/dbp21/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Top-N Anfragen in SQL schreiben können.
- Die Seitenaufteilung von großen Anfrage-Ergebnissen durchführen (z.B. für Webseiten).
- Die Möglichkeiten von Window Funktions in SQL für eine konkrete Anwendung einschätzen.
- Window Functions in SQL Anfragen einsetzen

  Zumindest einfache Fälle. Die Details komplexer „Window Frame“ Spezifikationen brauchen Sie in dieser Vorlesung nicht zu beherrschen, aber Sie sollten einen Eindruck haben, was möglich ist.

Inhalt

Beispiel-Datenbank

STUDENTEN

SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

BEWERTUNGEN

SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
102	H	2	9
102	Z	1	10
103	H	1	5
103	Z	1	7

AUFGABEN

ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

„Erste *n*“ Anfragen: Einführung (1)

- Gesucht sind die beiden Studenten mit dem besten Ergebnis für Hausaufgabe 1 (allgemein: die ersten *n* bezüglich einer Ordnung: „Top-N Queries“).
- Solche Anfragen sind mit den Möglichkeiten von SQL-92 recht schwierig zu lösen.

  Man kann zählen, wie viele ein schlechteres Ergebnis haben (s.u.).
- Da solche Anfragen aber relativ häufig vorkommen, haben viele Systeme ein spezielles Konstrukt dafür.

  Die Systeme unterscheiden sich in diesem Punkt aber stark (Portabilitätsproblem). Erst im SQL:2008 Standard wurde die `FETCH FIRST`-Klausel aufgenommen.

„Erste n“ Anfragen: Einführung (2)

- Aufgabe: Erste n (hier $n = 2$) Studenten bezüglich der Punkte in Hausaufgabe 1 (bei gleicher Punktzahl: `SID`).
- Klassische Lösung (ohne Spezial-Konstrukt):

```
SELECT S.VORNAME, S.NACHNAME, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H' AND B.ANR = 1
AND    2 > (SELECT COUNT(*) FROM BEWERTUNGEN X
            WHERE  X.ATYP = 'H' AND X.ANR = 1
            AND    (X.PUNKTE > B.PUNKTE OR
                    X.PUNKTE = B.PUNKTE AND
                       X.SID < B.SID))
ORDER  BY B.PUNKTE DESC
```

„Erste n“ Anfragen: Einführung (3)

- Wie oben erläutert, kann man bei neueren Datenbanken die Unteranfrage auch links schreiben:

```
(SELECT COUNT(*) ...) < 2
```

- Die Formel

```
(X.PUNKTE > B.PUNKTE OR
 X.PUNKTE = B.PUNKTE AND X.SID < B.SID)
```

  entspricht dem Sortierkriterium

```
ORDER BY PUNKTE DESC, SID
```

- Es werden also die `X` gezählt, die
  - mehr Punkte als `B` haben, oder
  - bei gleicher Punktzahl eine kleinere `SID`.

„Erste n“ Anfragen: Einführung (4)

- Falls man bei gleicher Punktzahl doch alle auflisten will, also ggf. auch mehr als n (hier $n = 2$):

```
SELECT S.VORNAME, S.NACHNAME, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H' AND B.ANR = 1
AND    2 > (SELECT COUNT(*) FROM BEWERTUNGEN X
            WHERE  X.ATYP = 'H' AND X.ANR = 1
            AND    X.PUNKTE > B.PUNKTE)
ORDER  BY B.PUNKTE DESC
```

- Die Benutzung der `SID` als zusätzliches Sortierkriterium zur Beschränkung auf zwei Ausgabezeilen ist ja willkürlich.

„Erste n“ Anfragen: FETCH FIRST (1)

- Nach dem SQL:2008 Standard wird die gleiche Aufgabe (die beiden Studenten mit der besten Punktzahl für Hausaufgabe 1) so gelöst:

```
SELECT S.VORNAME, S.NACHNAME, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H' AND B.ANR = 1
ORDER  BY B.PUNKTE DESC, S.SID
FETCH  FIRST 2 ROWS ONLY
```

  Um ein definiertes Ergebnis zu haben, wird hier wieder die `SID` zur Entscheidung bei Punkte-Gleichstand verwendet („tiebreak“).

- Wenn man nur das erste Antworttupel will, kann man auch „`FETCH FIRST ROW ONLY`“ schreiben.

  Die `FETCH FIRST`-Klausel kommt ganz ans Ende der Anfrage.

„Erste n“ Anfragen: FETCH FIRST (2)

- Das funktioniert in PostgreSQL ab Version 8.4 (2009).

  [https://www.postgresql.org/docs/8.4/sql-select.html]

- In Oracle funktioniert `FETCH FIRST` ab Version 12c (2013).

  [https://docs.oracle.com/database/121/SQLRF/statements_10002.htm]

- In DB2 funktioniert `FETCH FIRST` schon „ewig“ ($\leq$ 1998).

  [https://www.ibm.com/support/producthub/db2/docs/content/SSEPGG_11.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0059212.html?pos=2]

  Vermutlich kommt diese Syntax von DB2. Sie steht schon im Buch „A Complete Guide to DB2 Universal Database“ von Don Chamberlin (1998).

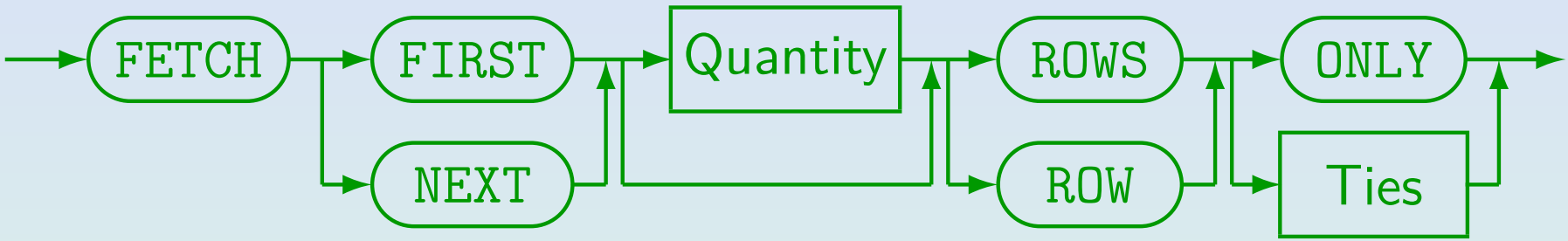
- In der Wikipedia steht eine Übersicht mit 14 Syntax-Varianten der „Top N“ Anfragen für ca. 24 verschiedene Systeme.

  [https://en.wikipedia.org/wiki/Select_(SQL)]

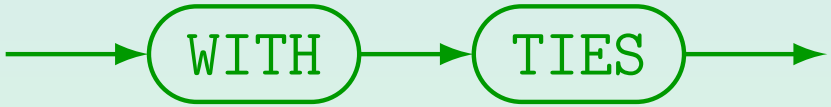
  Das zeigt zumindest die Portabilitätsprobleme.

„Erste *n*“ Anfragen: FETCH FIRST (3)

fetch first clause:



Ties:



- „`FIRST`“ und „`NEXT`“ haben keinen semantischen Unterschied. Man sollte „`NEXT`“ zusammen mit „`OFFSET`“ verwenden (s.u.).
- Wenn man „Quantity“ auslässt, ist `1` gemeint. Der SQL:2011 Standard erlaubt auch Prozentangaben, z.B. „`10 PERCENT`“. Das hat PostgreSQL noch nicht. In SQL:2008 war Quantity einfach eine „unsigned integer“.
- „`WITH TIES`“ wurde in SQL:2011 eingeführt. In PostgreSQL seit Version 13 (2020).

Anmerkung zur Performance

- Der Anfrageoptimierer kann das Wissen nutzen, dass sich der Benutzer nur für eine oder wenige Ergebniszeilen interessiert.

  Z.B. „nested loop join“ statt „merge join“ (mit vorgeschaltetem Sortieren), siehe Vorlesung „Datenbanken II“.

- Es macht also einen Unterschied,
  - ob man die spezielle Klausel verwendet,
  - oder die Anfrage stellt, und dann nur die ersten n Antworten von der Datenbank abholt.

    Das Anwendungsprogramm enthält eine Schleife über den Tupeln im Anfrageergebnis, die man natürlich vorzeitig abbrechen kann.
    Ggf. spart man mit „`FETCH FIRST`“ auch Netzwerkverkehr.

Seitenaufteilung: OFFSET (1)

- Wenn große Anfrageergebnisse im Web angezeigt werden sollen, werden sie üblicherweise auf mehrere Seiten verteilt.

  Die Aufgabe nennt man auch „query result pagination“ (Seiteneinteilung).

- Z.B. enthält jede Seite 20 Anfrageergebnisse.

  Bei einigen Webseiten kann man auch wählen, wie viele Anfrageergebnisse pro Seite angezeigt werden sollen.

- Für die erste Seite verwendet man dann natürlich

```
FETCH FIRST 20 ROWS ONLY
```

- Bei der zweiten Seite muss man 20 Zeilen überspringen, und dann die nächsten 20 Zeilen abfragen:

```
OFFSET 20 ROWS
FETCH NEXT 20 ROWS ONLY
```

Seitenaufteilung: OFFSET (2)

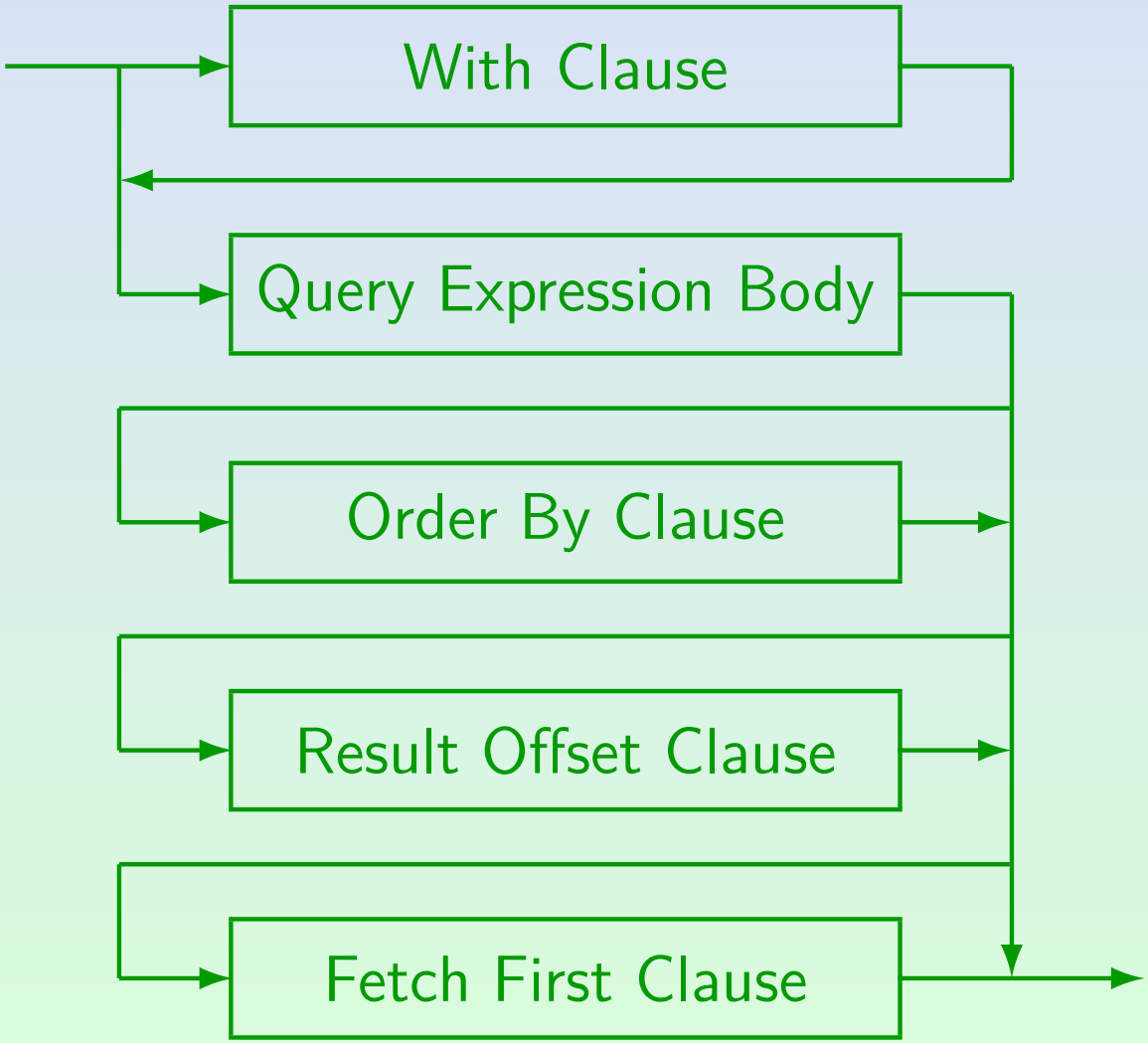
- Nach `OFFSET` wird die Anzahl der zu überspringenden Zeilen angegeben.
- Mit „`OFFSET 0 ROWS`“ bekommt man also die erste Zeile.
- `OFFSET` wurde im SQL:2011 Standard eingeführt.
- PostgreSQL hatte `LIMIT` (eine syntaktische Variante für `FETCH FIRST`) und `OFFSET` schon lange.

  Mindestens seit Version 7.1 (2001), möglicherweise früher. Als in Version 8.4 (2009) das standard-konforme `FETCH FIRST` eingeführt wurde, funktionierte das auch mit `OFFSET`.
- Oracle (ab Version 12c) und DB2 (ab Version 11) unterstützen auch `OFFSET`.

  [https://docs.oracle.com/database/121/SQLRF/statements_10002.htm]
  [https://www.ibm.com/docs/en/db2/11.1?topic=queries-subselect]

Seitenaufteilung: OFFSET (3)

Query Expression:



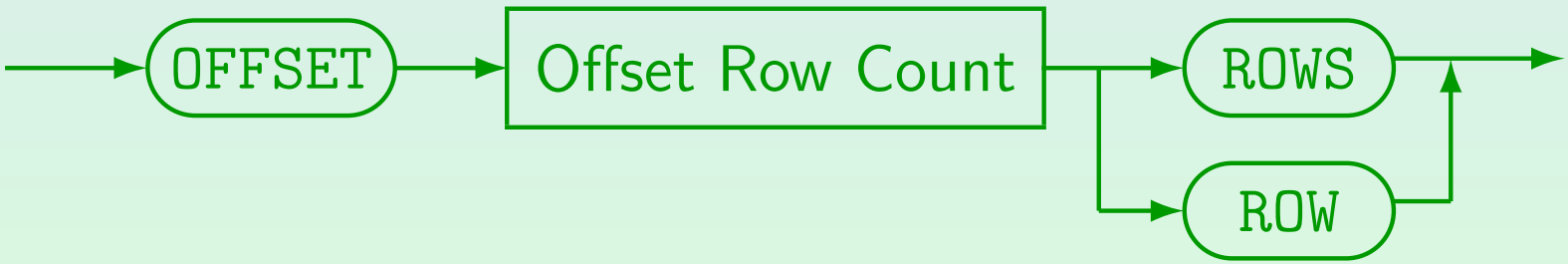
Seitenaufteilung: OFFSET (4)

Erläuterungen zu „Query Expression“:

- Der „Query Expression Body“ ist (etwas vereinfacht) eine Verknüpfung von `SELECT`-Ausdrücken mit `UNION` u.s.w.
- Die Reihenfolge ist also: `ORDER BY`, `OFFSET`, `FETCH FIRST`.

  Wenn man `OFFSET` verwendet, sollte man `FETCH NEXT` schreiben.

Result Offset Clause:



Der „Offset Row Count“ ist eine nicht-negative ganze Zahl (nach der SQL-Grammatik eine „Simple Value Specification“, das ist insbesondere eine Zahlkonstante, aber auch Programmparameter etc.). Bei PostgreSQL ist `ROW`/`ROWS` nach `OFFSET` optional.

Warnung vor Nichtdeterminismus

- Während es formal möglich wäre, `OFFSET` und `FETCH FIRST` ohne `ORDER BY` zu verwenden, macht das keinen Sinn:
  - Ohne `ORDER BY` ist die Reihenfolge der Ergebniszeilen nicht vorgeschrieben.
  - Sie ergibt sich aus dem Berechnungsalgorithmus, den der Anfrageoptimierer gewählt hat.
  - Bei der nächsten Systemversion oder veränderten Tabellengrößen kann der Optimierer einen anderen Algorithmus wählen.
- Er kann sogar bei einem anderen `OFFSET`-Wert einen anderen Algorithmus wählen! Dann passen die Seiten nicht.
- Man muss also eine `ORDER BY` Sortierung wählen, die die Reihenfolge aller Zeilen genau vorschreibt.

Performance von OFFSET

- Wenn der Optimierer nicht cleverer als ich ist, wird er mit `OFFSET` übersprungene Zeilen im wesentlichen doch berechnen.

  Eventuell braucht man einen Join über einen Fremdschlüssel nicht auszuführen: Man weiss ja, dass es keinen Einfluss auf die Anzahl der Ergebniszeilen hat. Man kann also ein wenig sparen, weil die Spaltenwerte der übersprungenen Zeilen nicht gebraucht werden. Aber man muss sie so weit berechnen, dass man sie zählen kann.

- Für relativ kleine Anfrage-Ergebnisse ist das kein Problem.

- Ansonsten kann man die letzten auf dieser Seite angezeigten Werte für die `ORDER BY` Spalten in den Link zur nächsten Seite einfügen: Dort kann die Anfrage statt `OFFSET` nach Zeilen fragen, die größer in diesen Werten sind.

  Bei „`ORDER BY A, B`“: Größer in `A` oder gleich in `A` und größer in `B`.

Inhalt

„Erste n“ Anfragen: LIMIT (1)

- In MySQL verwendet man die `LIMIT`-Klausel (auch PostgreSQL versteht `LIMIT`):

```
SELECT SID, PUNKTE
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND ANR = 1
ORDER  BY PUNKTE DESC
LIMIT  2
```

- Auch in MySQL kann man Zeilen am Anfang des Anfrage-Ergebnisses überspringen:

```
LIMIT 2 OFFSET 3
```

Überspringt drei Zeilen und gibt dann zwei aus.

„Erste n“ Anfragen: LIMIT (2)

- Beachte: Die erste Zeile hat den Offset 0.

```
LIMIT 2 OFFSET 3
```

  Dies gibt die vierte und fünfte Zeile des Anfrageergebnisses aus.

- Statt `LIMIT` *L* `OFFSET` *O* kann man auch `LIMIT` *O*`,` *L* schreiben (im Beispiel: `LIMIT 3, 2`).

  Bei MySQL gibt es syntaktisch keine Möglichkeit, nur den Offset anzugeben.

  Das Handbuch erklärt, dass man für `LIMIT` ja einen großen Wert angeben kann.

  [https://dev.mysql.com/doc/refman/8.0/en/select.html]

- Wegen der Kompatibilität mit PostgreSQL (und der Verständlichkeit) sollte man aber die Schlüsselworte `LIMIT` und `OFFSET` verwenden.

„Erste n“ Anfragen: TOP

- Microsoft SQL Server:

```
SELECT TOP (2) SID, PUNKTE
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND ANR = 1
ORDER  BY PUNKTE DESC
```

  Aus Gründen der Abwärtskompatibilität kann man die Klammern um die Anzahl der Zeilen auch weglassen (nicht empfohlen).

- `TOP (10) PERCENT`: erste 10% des Ergebnisses.
- `TOP (2) WITH TIES`: auch mehr als 2 Zeilen, wenn die folgenden Zeilen mit der letzten gewählten Zeile in den `ORDER BY`-Attributen übereinstimmen.

„Erste n“ Anfragen: ROWNUM (1)

- Klassisch wird in Oracle die Funktion `ROWNUM` benutzt (zumindest vor Version 12c).
- Die 0-stellige Funktion `ROWNUM` erzeugt eindeutige Nummern für die Ausgabezeilen (beginnend mit 1).

  In der Oracle Dokumentation findet man `ROWNUM` im Abschnitt „Pseudocolumns“, d.h. virtuelle Spalten, die bei Bedarf berechnet werden. In diesem Fall wäre es allerdings eine Spalte der Ausgabetabelle.
- Z.B. kann man damit die Anzahl der Ausgabezeilen begrenzen:

```
SELECT SID, PUNKTE
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND ANR = 1
AND    ROWNUM <= 2
```

„Erste *n*“ Anfragen: ROWNUM (2)

- Die obige Anfrage gibt zwei Bewertungen für Hausaufgabe 1 aus. Es ist nicht vorhersehbar, welche Bewertungen ausgegeben werden.

  Die Auswahl hängt von internen Algorithmen des Anfrageoptimierers ab und kann sich von einer Oracle-Version zur nächsten ändern.

  Genauer funktioniert `ROWNUM` folgendermaßen: Es wird ein Zähler verwendet, der mit 1 initialisiert ist. `ROWNUM` liefert immer den aktuellen Wert des Zählers. Falls eine Tupelkombination die `WHERE`-Bedingung erfüllt hat, wird der Zähler hochgesetzt.

  `ROWNUM` bezieht sich nicht eigentlich auf Ausgabezeilen. Der Unterschied wird bei „`GROUP BY`“-Anfragen deutlich: Hier kann `ROWNUM` unter `WHERE`, aber nicht unter `SELECT` verwendet werden. Bei Anfragen ohne „`GROUP BY`“ kann es natürlich auch unter `SELECT` verwendet werden.

„Erste *n*“ Anfragen: ROWNUM (3)

- Da „ORDER BY“ erst nach der WHERE-Bedingung ausgewertet wird, hat es keinen Einfluss auf ROWNUM.

  Möglicherweise wählt der Optimierer aber einen anderen Auswertungsplan, was dann die ausgewählten Zeilen verändern könnte.

- Es geht aber mit einer Unteranfrage:

```
SELECT SID, PUNKTE
FROM   (SELECT SID, PUNKTE
        FROM   BEWERTUNGEN
        WHERE  ATYP = 'H' AND ANR = 1
        ORDER  BY PUNKTE DESC, SID)
WHERE  ROWNUM <= 2
```

ORDER BY in Unteranfragen ist erst seit SQL:2008 standard-konform.

„Erste n“ Anfragen: ROWNUM (4)

- Beachte: `ROWNUM > 1` kann nicht funktionieren!

  Da die Bedingung am Anfang nicht erfüllt ist, wird `ROWNUM` nicht erhöht, ist also immer 1.

- Man kann `ROWNUM` aber in einer Unteranfrage generieren, und die Bedingung außen formulieren, z.B.:

```
SELECT SID, PUNKTE
FROM   (SELECT ROWNUM AS NR, SID, PUNKTE
        FROM   (SELECT *
                FROM   BEWERTUNGEN
                WHERE  ATYP = 'H' AND ANR = 1
                ORDER  BY PUNKTE DESC, SID))
WHERE  NR > 3 AND NR <= 5
```

Inhalt

Einführung

- Im SQL:2003 Standard wurden „Window Functions“ eingeführt, die Aggregation innerhalb eines Fensters ausführen können, das über die sortierten Daten läuft (bei Oracle: „Analytic Functions“).
- Damit kann man z.B. Aktienkurse glätten, indem man 3-Tage Durchschnittswerte berechnet.
- Man kann aber auch Positionen innerhalb der geordneten Liste bestimmen.

  Das entspricht einem `count(*)` über einem Fenster, dessen Anfang am Anfang der Liste verankert ist, aber dessen Ende über die Liste hinwegläuft — des Fenster wird also immer größer.

Ranking (1)

- Z.B. alle Abgaben für Hausaufgabe 1 mit
  - Position in der Sortierreihenfolge bzgl. Punkten,
  - laufender Studentennummer, und

    Position bei Sortierung nach `SID`: Nicht besonders sinnvoll, soll aber verschiedene Sortierungen in einer Anfrage demonstrieren.
  - die Ausgabe sortiert nach Namen:

```
SELECT S.NACHNAME, S.VORNAME, B.PUNKTE,
       RANK() OVER (ORDER BY B.PUNKTE DESC),
       RANK() OVER (ORDER BY S.SID)
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H' AND B.ANR = 1
ORDER  BY S.NACHNAME, S.VORNAME
```

Ranking (2)

- `RANK()` ist 1 plus die Anzahl der Tupel, die in den `ORDER BY` Attributen einen echt kleineren Wert haben (bzw. echt größer bei `DESC`).

  Es werden also Tupel mit gleichen Werten in den `ORDER BY`-Attributen („Peers“) nicht mitgezählt. Das führt zu Lücken in der Numerierung. Nach der Lücke synchronisiert sich `RANK()` wieder mit `ROW_NUMBER()`.

- `DENSE_RANK()` ist 1 plus die Anzahl von echt kleineren Werten in den `ORDER BY` Attributen.

  So gibt es keine Lücken, aber Werte sind keine Tupelanzahlen mehr.

- `ROW_NUMBER()` numeriert die Zeilen durch.

  Beginnend mit 1. Bei gleichen Werten in den `ORDER BY` Attributen ist die Numerierung nichtdeterministisch.

Ranking (3)

- Beispiel (falls `ORDER BY PUNKTE DESC`):

NAME	PUNKTE	RANK	DENSE_RANK	ROW_NUMBER
Anna	25	1	1	1
Bettina	20	2	2	2
Christine	20	2	2	3
Dorothea	15	4	3	4
Elisabeth	15	4	3	5
Felicia	15	4	3	6
Georgia	10	7	4	7

Ranking (4)

- `PERCENT_RANK()`: Berechnet als `(RANK()-1)/(N-1)`.

  Dabei ist `N` die Gesamtanzahl der Zeilen (in der Partition, s.u.).

- `CUME_DIST()`: Berechnet als `B/N`.

  Dabei ist `B` die Anzahl von Zeilen mit gleichem oder kleineren Wert, `N` wie oben („cumulative distribution“).

NAME	PUNKTE	RANK	PERCENT_RANK	CUME_DIST
Anna	25	1	0.00	0.14
Bettina	20	2	0.17	0.43
Christine	20	2	0.17	0.43
Dorothea	15	4	0.50	0.86
Elisabeth	15	4	0.50	0.86
Felicia	15	4	0.50	0.86
Georgia	10	7	1.00	1.00

Ranking (5)

- Man kann bei der Sortierung auch Aggregationsterme verwenden (alle Terme, die auch sonst unter SELECT möglich wären):

```
SELECT S.NACHNAME, S.VORNAME, SUM(B.PUNKTE),
       RANK() OVER (ORDER BY SUM(B.PUNKTE) DESC)
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H'
GROUP  BY S.SID, S.NACHNAME, S.VORNAME
ORDER  BY S.NACHNAME, S.VORNAME
```

Es ist nicht verlangt, dass die Terme, nach denen in der OVER-Klausel sortiert wird, auch unter SELECT ausgegeben werden.

Ranking (6)

- Syntaktisch ist

ein Wertausdruck (Term, Expression).

Im SQL:2003 (und auch SQL:2016) Standard unter 6.10 „window function“.

- Allerdings darf diese Form von Wertausdruck nur unter `SELECT` und `ORDER BY` verwendet werden.

Aggregationsterme können ja auch nicht unter `WHERE` verwendet werden.

- „Window Function Type“ ist z.B. `RANK()`.

Auch normale Aggregationsfunktionen sind möglich, z.B. `COUNT(*)`, s.u.

- „Window Specification“ ist z.B. `(ORDER BY PUNKTE DESC)`

Alternativ auch ein in der Anfrage definierter Fenster-Name, s.u..

Partitionierung (1)

- Man kann das Anfrage-Ergebnis auch erst partitionieren (ähnlich zu `GROUP BY`) und dann die Position innerhalb jeder Partition berechnen:

```
SELECT S.NACHNAME, S.VORNAME, B.ANR, B.PUNKTE,
       RANK() OVER (PARTITION BY B.ANR
                    ORDER BY B.PUNKTE DESC)
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H'
ORDER  BY S.NACHNAME, S.VORNAME, B.ANR
```

- Hier wird die Position (nach Punkten) für jede Aufgabe einzeln bestimmt (Ergebnis: nächste Folie).

Partitionierung (2)

- Ergebnis der Anfrage auf der letzten Folie:

NAME	VORNAME	ANR	PUNKTE	RANK
Grau	Michael	1	9	2
Grau	Michael	2	9	1
Sommer	Daniel	1	5	3
Weiss	Lisa	1	10	1
Weiss	Lisa	2	8	2

Z.B. für Hausaufgabe 1 ist Lisa Weiss mit 10 Punkten Erste, dann kommt Michael Grau mit 9 Punkten und dann Daniel Sommer mit 5 Punkten.
In der anderen Partition (Aufgabe 2) kommt Michael Grau (10 Punkte) vor Lisa Weiss (8 Punkte). Daniel Sommer hat nichts abgegeben.

Partitionierung (3)

- Auch die üblichen Aggregations-Funktionen können im OVER-Konstrukt verwendet werden:

```
SELECT S.NACHNAME, S.VORNAME, B.ANR, B.PUNKTE,
       AVG(PUNKTE) OVER (PARTITION BY B.ANR)
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H'
ORDER  BY S.NACHNAME, S.VORNAME, B.ANR
```

- Auf diese Weise wird in jeder Zeile der Punkte-Durchschnitt über alle Abgaben für diese Aufgabe angezeigt (Ergebnis siehe nächste Folie).

Partitionierung (4)

- Die obige Anfrage verwendet implizit ein „Fenster“ (Eingabe der Aggregationsfunktion), das sich über alle Tupel der Partition erstreckt.
- Daher ist der berechnete Durchschnitt für alle Bewertungen einer Aufgabe gleich:

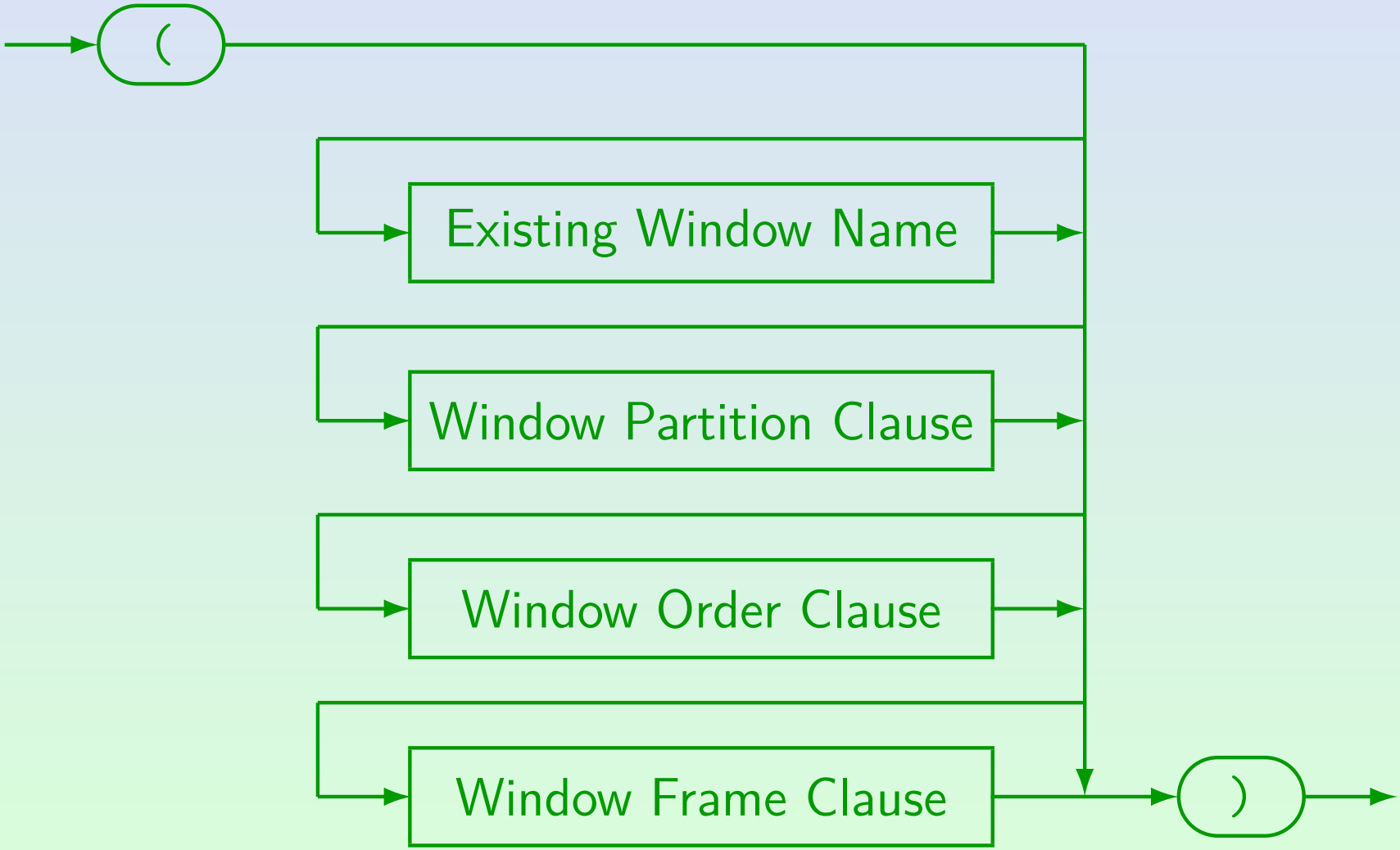
NACHNAME	VORNAME	ANR	PUNKTE	AVG
Grau	Michael	1	9	8.0
Grau	Michael	2	9	8.5
Sommer	Daniel	1	5	8.0
Weiss	Lisa	1	10	8.0
Weiss	Lisa	2	8	8.5

Partitionierung (5)

- Hier zeigt sich ein Unterschied zu `GROUP BY`:
  - Bei `GROUP BY` wird nur eine Ausgabe pro Gruppe erzeugt. Es ist nicht mehr möglich, auf Werte von Attributen, nach denen nicht gruppiert wird, einzeln zuzugreifen.
  - Mit dem `OVER`-Konstrukt wird dagegen eine Ausgabe pro Tupel erzeugt. Man behält daher den vollen Zugriff auf alle Attribute, kann aber das Tupel in einem Kontext sehen. Man bekommt so aggregierten Zugriff auf andere Tupel.

Partitionierung (6)

Window Specification:



Unterstützung in verschiedenen DBMS

- Window Functions wurden im SQL:2003 Standard eingeführt.
- PostgreSQL hat Window Functions seit Version 8.4 (2009)

  [https://www.postgresql.org/docs/9.2/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS]
- Oracle hatte diese Funktionen in Version 8i (1998)

  [https://docs.oracle.com/cd/A84870_01/doc/server.816/a76989/functio2.htm#81409]
- SQLite ab Version 3.25 (2018)

  [https://sqlite.org/windowfunctions.html]

  [https://data-xtractor.com/blog/query-builder/window-functions-support/]
- MariaDB ab 10.2 (2016/17), MySQL ab 8.0.2 (2017/18)

  [https://mariadb.com/kb/en/window-functions-overview/]

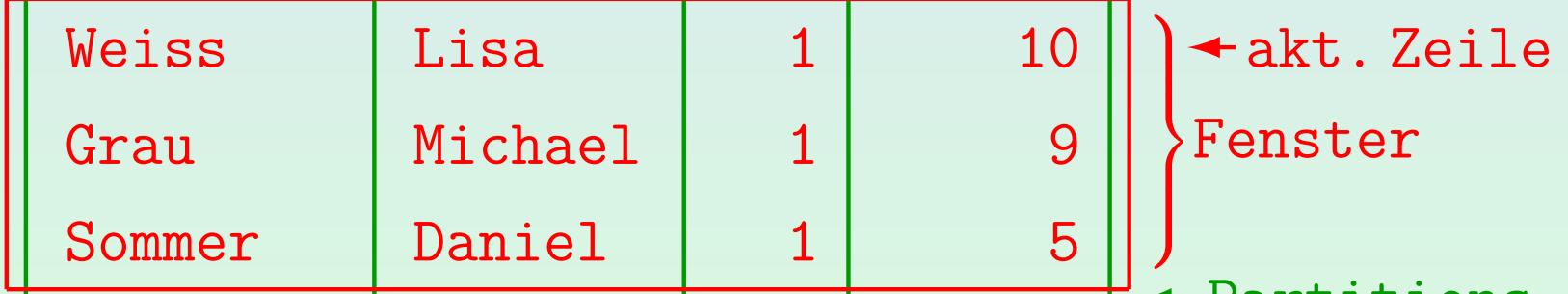
  [https://dev.mysql.com/doc/refman/8.0/en/window-functions.html]

Inhalt

Fenster/Windows (1)

- Ein `OVER`-Ausdruck, z.B. `OVER (PARTITION BY B.ANR)` definiert zu jeder Zeile (bzw. Variablenbelegung) im Ergebnis des „`FROM ... WHERE ... GROUP ... HAVING ...`“ ein Fenster (im Beispiel die ganze Partition):

NACHNAME	VORNAME	ANR	PUNKTE
Weiss	Lisa	1	10
Grau	Michael	1	9
Sommer	Daniel	1	5
Weiss	Lisa	2	8
Grau	Michael	2	9



Fenster/Windows (2)

- Im Beispiel bekommt man für jede der ersten drei Zeilen das gleiche Fenster (die ganze Partition, zu der die aktuelle Zeile gehört).
- Daher liefert z.B.

  ```
  AVG(PUNKTE) OVER (PARTITION BY B.ANR)
  ```

  für diese drei Zeilen den gleichen Wert (8).
- Das ändert sich schon, wenn man zusätzlich eine `ORDER BY` Klausel verwendet, siehe nächste Folie.

Fenster/Windows (3)

- Mit Sortierung, z.B.

```
OVER (PARTITION BY B.ANR
      ORDER BY B.PUNKTE DESC)
```

erstreckt sich das Fenster vom Anfang der Partition bis zur aktuellen Zeile (plus Peers/Ties, s.u.):

NACHNAME	VORNAME	ANR	PUNKTE
Weiss	Lisa	1	10
Grau	Michael	1	9
Sommer	Daniel	1	5
Grau	Michael	2	9
Weiss	Lisa	2	8

Fenster

← akt. Zeile

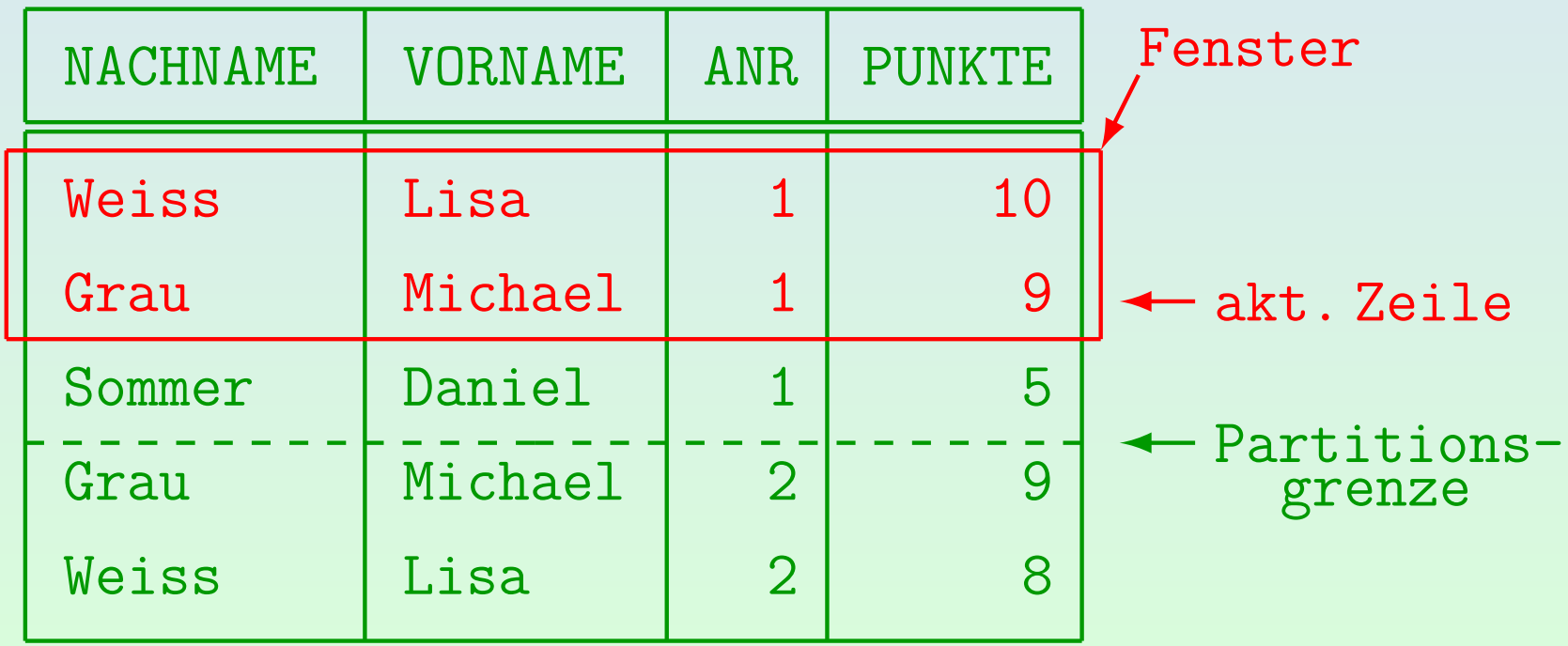
← Partitions-grenze

Fenster/Windows (4)

- Fortsetzung des Beispiels:

```
OVER (PARTITION BY B.ANR
      ORDER BY B.PUNKTE DESC)
```

Fenster für die zweite Zeile:



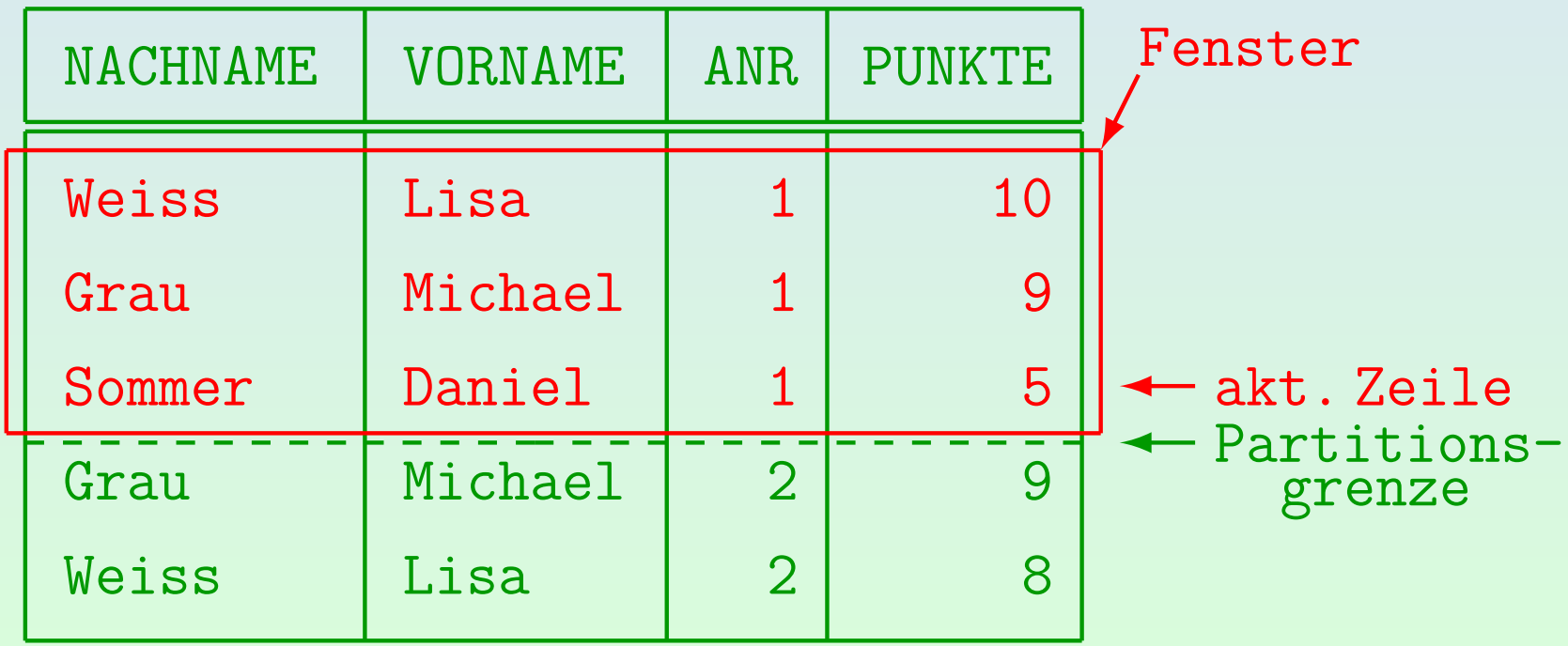
Fenster/Windows (5)

- Fortsetzung des Beispiels:

```
OVER (PARTITION BY B.ANR
      ORDER BY B.PUNKTE DESC)
```

Fenster für die dritte Zeile:

NACHNAME	VORNAME	ANR	PUNKTE
Weiss	Lisa	1	10
Grau	Michael	1	9
Sommer	Daniel	1	5
Grau	Michael	2	9
Weiss	Lisa	2	8



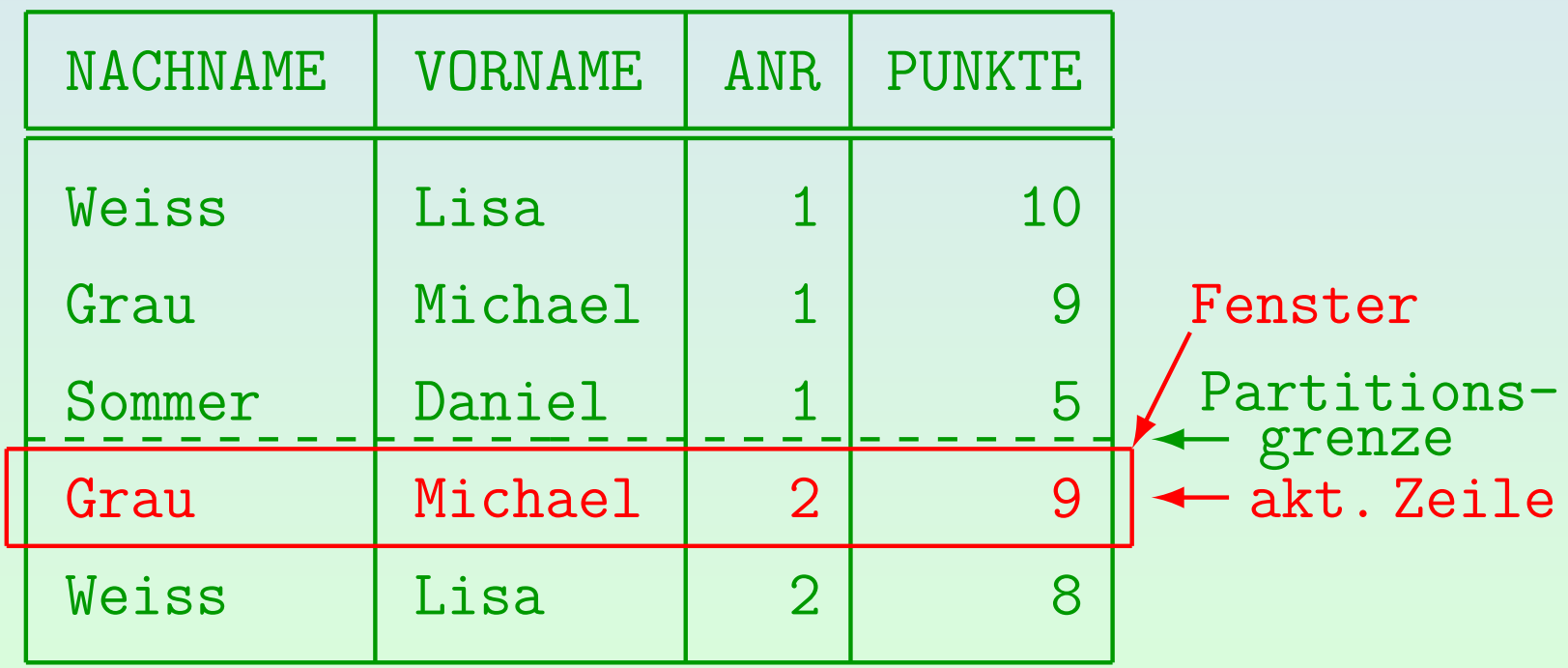
Fenster/Windows (6)

- Fortsetzung des Beispiels:

```
OVER (PARTITION BY B.ANR
      ORDER BY B.PUNKTE DESC)
```

Fenster für die vierte Zeile:

NACHNAME	VORNAME	ANR	PUNKTE
Weiss	Lisa	1	10
Grau	Michael	1	9
Sommer	Daniel	1	5
Grau	Michael	2	9
Weiss	Lisa	2	8



Fenster/Windows (7)

- Fortsetzung des Beispiels:

```
OVER (PARTITION BY B.ANR
      ORDER BY B.PUNKTE DESC)
```

Fenster für die fünfte Zeile:

NACHNAME	VORNAME	ANR	PUNKTE
Weiss	Lisa	1	10
Grau	Michael	1	9
Sommer	Daniel	1	5
Grau	Michael	2	9
Weiss	Lisa	2	8

Fenster

Partitions-grenze

akt. Zeile

Fenster/Windows (8)

- Z.B. bekommt man bei

```
COUNT(*)
OVER (PARTITION BY B.ANR
      ORDER BY B.PUNKTE DESC)
```

folgendes Ergebnis:

NACHNAME	VORNAME	ANR	PUNKTE	COUNT
Weiss	Lisa	1	10	1
Grau	Michael	1	9	2
Sommer	Daniel	1	5	3
Grau	Michael	2	9	1
Weiss	Lisa	2	8	2

Fenster/Windows (9)

- Im Beispiel sind zufällig alle Punktzahlen in einer Partition verschieden.
- Sollten zwei Zeilen aber in den `PARTITION BY` und den `ORDER BY` Attributen übereinstimmen, würde das Fenster diese Zeilen als Block behandeln.

  Das Fenster kann dann also ggf. über die aktuelle Zeile hinaus reichen.
- Bei expliziten Rahmenspezifikationen (s.u.) kann man wählen, ob solche „Ties“ im Fenster mit enthalten sind, oder nicht.

  Wenn man das Schlüsselwort `ROWS` verwendet, sind sie nicht enthalten, bei `RANGE` schon. Es gibt auch „`EXCLUDE TIES`“ (s.u.).
- Natürlich kann man auch einfach die Liste der `ORDER BY` Attribute verlängern, z.B. die `SID` noch mit hinzufügen.

Fenster/Windows (10)

- Beispiel für Einschluss von „Ties“:

```
COUNT(*) OVER (ORDER BY B.PUNKTE DESC) AS "COUNT"
```

NAME	PUNKTE	COUNT	COUNT_ROWS	COUNT_RANGE
Anna	25	1	1	1
Bettina	20	3	2	3
Christine	20	3	3	3
Dorothea	15	6	4	6
Elisabeth	15	6	5	6
Felicia	15	6	6	6
Georgia	10	7	7	7

COUNT_ROWS (s.u): `COUNT(*) OVER (ORDER BY PUNKTE DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)`. COUNT_RANGE: Mit `RANGE` statt `ROWS` ergibt sich das gleiche Ergebnis wie ohne explizite Fenster-Spezifikation (s.u.).

Fenster/Windows (11)

- Die Erweiterung des Fensters um jeweils eine Zeile ist z.B. nützlich, wenn man jeweils eine laufende Summe (Zwischensumme bis zur aktuellen Zeile) mit ausgeben will.

```
SELECT S.NACHNAME, S.VORNAME, B.ANR, B.PUNKTE,
       SUM(PUNKTE) OVER (PARTITION BY B.SID
                         ORDER BY B.ANR)
       AS SUMME
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H'
ORDER  BY S.NACHNAME, S.VORNAME, S.SID, B.ANR
```

Hier wird in jeder Zeile die Summe der Hausaufgabenpunkte für den jeweiligen Studenten bis zu der Aufgabe der Zeile angezeigt.

Fenster/Windows (12)

- Ergebnis mit laufenden Summen:

```
SUM(PUNKTE) OVER (PARTITION BY B.SID
                  ORDER BY B.ANR)
```

NACHNAME	VORNAME	ANR	PUNKTE	SUMME
Grau	Michael	1	9.0	9.0
Grau	Michael	2	9.0	18.0
Sommer	Daniel	1	5.0	5.0
Weiss	Lisa	1	10.0	10.0
Weiss	Lisa	2	8.0	18.0

- Typisches Beispiel für laufende Summen:
  Aktueller Kontostand, wenn jede Zeile eine Buchung ist.

  Laufende Summen erleichtern auch Kontrollen.

Auswertung von OVER-Ausdrücken (1)

- Man kann sich die Auswertung so vorstellen:
  - Zunächst wertet das System

    `FROM ... WHERE ... GROUP ... HAVING ...`

    aus mit einer klassischen `SELECT`-Klausel, die alle im Folgenden benötigten Attribute enthält.

    Im Standard heißt dieser Ausdruck (mit noch einer optionalen `WINDOW`-Klausel am Ende) eine „table expression“.
  - Das Ergebnis wird sortiert nach den `PARTITION BY`-Attributen, gefolgt von den `ORDER BY`-Attributen.

    So stehen die Tupel einer Partition hintereinander, und in richtiger Reihenfolge.

Auswertung von OVER-Ausdrücken (2)

- Auswertung mit `OVER`, Forts.:
  - Dann wird Ergebnis durchlaufen, und für jede Zeile ihr zugehöriges Fenster duchgegangen, wobei die jeweilige Aggregationsfunktion ausgewertet wird.

    Die Zeilen des Fensters könnte man mit einer geschachtelten Schleife durchlaufen, die je nach Rahmenspezifikation (s.u.) des Fensters auch über die aktuelle Zeile hinaus reichen kann.
  - Das Ergebnis wird den Zeilen jeweils hinzugefügt.
  - Gibt es mehrere `OVER`-Ausdrücke, so sortiert man ggf. neu und wiederholt den Vorgang.
  - Auch für `ORDER BY` am Ende wird neu sortiert.

Rahmen-Spezifikation (1)

- Sei eine Tabelle mit Aktienkursen gegeben:

AKTIENKURSE		
Aktie	Datum	Wert
Santa Claus AG	01.12.2011	100.00
Santa Claus AG	02.12.2011	105.00
Santa Claus AG	03.12.2011	104.00
Santa Claus AG	04.12.2011	106.00
⋮	⋮	⋮
Easter Eggs AG	01.12.2011	50.00
Easter Eggs AG	02.12.2011	40.00
Easter Eggs AG	03.12.2011	45.00
⋮	⋮	⋮

Rahmen-Spezifikation (2)

- Folgende Anfrage berechnet die Durchschnittswerte über jeweils drei Tage:

```
SELECT AKTIE, DATUM,
       AVG(WERT) OVER (PARTITION BY AKTIE
                       ORDER BY DATUM
                       ROWS BETWEEN
                            1 PRECEDING AND
                            1 FOLLOWING)
FROM   AKTIENKURSE
ORDER  BY AKTIE, DATUM
```

Vereinfachend wird hier zunächst nicht berücksichtigt, dass es nicht an allen Tagen einen Aktienkurs gibt.

Rahmen-Spezifikation (3)

- Anfrage-Ergebnis, falls die Tabelle nur die obigen sieben Zeilen enthält:

AKTIENKURSE		
Aktie	Datum	AVG
Easter Eggs AG	01.12.2011	45.00
Easter Eggs AG	02.12.2011	45.00
Easter Eggs AG	03.12.2011	42.50
Santa Claus AG	01.12.2011	102.50
Santa Claus AG	02.12.2011	103.00
Santa Claus AG	03.12.2011	105.00
Santa Claus AG	04.12.2011	105.00

Bei der ersten Zeile gibt es keine vorangehende Zeile, es wird dann nur der Durchschnitt über dieser Zeile und der nächsten gebildet.

Rahmen-Spezifikation (4)

- Statt *n* `PRECEDING/FOLLOWING` ist auch möglich:
  - `UNBOUNDED PRECEDING/FOLLOWING`
  - `CURRENT ROW`

    Es gibt Einschränkungen, so dass das Intervall sinnvoll (nicht leer) ist. Man kann statt `BETWEEN` auch nur den Anfang spezifizieren, dann ist das Ende implizit bei der aktuellen Zeile, z.B. „`ROWS 3 PRECEDING`“.
- Statt `ROWS` kann man auch `RANGE` verwenden, um einen Bereich für den Attributwert des `ORDER BY` Attributes anzugeben (`RANGE` schließt „Peers“ ein, `ROWS` nicht).

  Wenn man `PRECEDING` bzw. `FOLLOWING` mit einem Wert verwendet, darf es nur ein `ORDER BY` Attribut geben, das auch von einem numerischen, Datums/Zeit- oder Intervall-Datentyp sein muss (so dass eine Addition bzw. Subtraktion des angegebenen Abstands zum Wert des `ORDER BY` Attributes möglich ist).

Rahmen-Spezifikation (5)

- Nach dem Standard ist z.B.

```
RANK() OVER (ORDER BY PUNKTE DESC)
```

äquivalent zu:

```
( COUNT(*) OVER (ORDER BY PUNKTE DESC
                 RANGE UNBOUNDED PRECEEDING)
- COUNT(*) OVER (ORDER BY PUNKTE DESC
                 RANGE CURRENT ROW)
+ 1)
```

- Wenn nach `ROW` oder `RANGE` nur ein Wert angegeben wird, ist das der Startpunkt des Fensters. Endpunkt: `CURRENT ROW`.
- Wegen `RANGE` werden die „Peers“ jeweils mitgezählt.

  `RANGE CURRENT ROW` ist also nicht eine Zeile, sondern alle, die in den `ORDER BY` Attributen den gleichen Wert wie die aktuelle Zeile haben.

Rahmen-Spezifikation (6)

- Man kann aus dem Fenster Zeilen ausschließen:
  - `EXCLUDE CURRENT ROW`
  - `EXCLUDE GROUP`

    Damit werden die aktuelle Zeile und alle Zeilen mit den gleichen Werten in den `ORDER BY`-Attributen entfernt.
  - `EXCLUDE TIES`

    Dies entfernt Zeilen mit den gleichen Werten in den `ORDER BY` Attributen, aber nicht die aktuelle Zeile.
  - `EXCLUDE NO OTHERS`

    Dies entfernt gar keine Zeilen (es könnten vorher Zeilen aufgrund von anderen Regeln entfernt worden sein, deswegen diese Bezeichnung).

Benannte Fenster

- Es ist auch möglich, einen Namen für Fenster in einer WINDOW-Klausel einzuführen, und diesen dann nach OVER zu benutzen:

```
SELECT AKTIE, DATUM,
       AVG(WERT) OVER UMGEBUNG
FROM   AKTIENKURSE
WINDOW UMGEBUNG AS (PARTITION BY AKTIE
                    ORDER BY DATUM
                    ROWS BETWEEN
                        1 PRECEDING AND
                        1 FOLLOWING)
ORDER  BY AKTIE, DATUM
```

Das ist nützlich, wenn man mehrere Aggregationen über dem gleichen Fenster ausführen will. WINDOW kommt nach der HAVING-Klausel.

Literatur/Quellen

- Kemper/Eickler: Datenbanksysteme, 7. Aufl., Kap. 17, Oldenbourg, 2009.
- Elmasri/Navathe: Fundamentals of Database Systems, 3rd. Ed., Addison-Wesley, 2000. Kapitel 26: Data Warehousing and Data Mining (pp. 841–872).
- ISO/IEC 9075-2: Information Technology - Database Languages - SQL - Part 2 (2003).
- Oracle Database SQL Language Reference, 11g Release 2 (11.2), `http://download.oracle.com/docs/cd/E11882_01/server.112/e26088/toc.htm`
- Oracle Database Data Warehousing Guide, 11g Release 2 (11.2). `http://docs.oracle.com/cd/E11882_01/server.112/e25554/toc.htm`
- IBM DB2 Database for Linux, UNIX and Windows — Informationszentrale. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp`
- Transact SQL Reference (Database Engine), SQL Server 2008 R2. `http://msdn.microsoft.com/en-us/library/bb510741.aspx`
- Cristian Scutaru (Data Xtractor): Database Products with Window Functions Support `https://data-xtractor.com/blog/query-builder/window-functions-support/`
- Chaudhuri, S./Dayal, U.: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record, Vol. 26, No. 1, March 1997.
- Wikipedia: Data Warehouse, OLAP, OLAP Cube, Essbase, MDX. `http://en.wikipedia.org/wiki/Data_Warehouses`
- Deßloch: Recent Developments for Data Models (Folien, TU Kaiserslautern, 2010). `http://wwwlgis.informatik.uni-kl.de/cms/courses/datenmodelle/`