FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Datenbanksysteme II

Prof. Dr. Viktor Leis

Professur für Datenbanken und Informationssysteme

SQL

SQL

- SQL ist die Standardsprache für Datenzugriff
- wird von fast allen Datenverarbeitungsplattformen unterstützt
- deklarative Sprache (es wird nicht spezifiziert wie der Zugriff erfolgt)

ANSI

SQL ist ein ANSI Standard, der wächst:

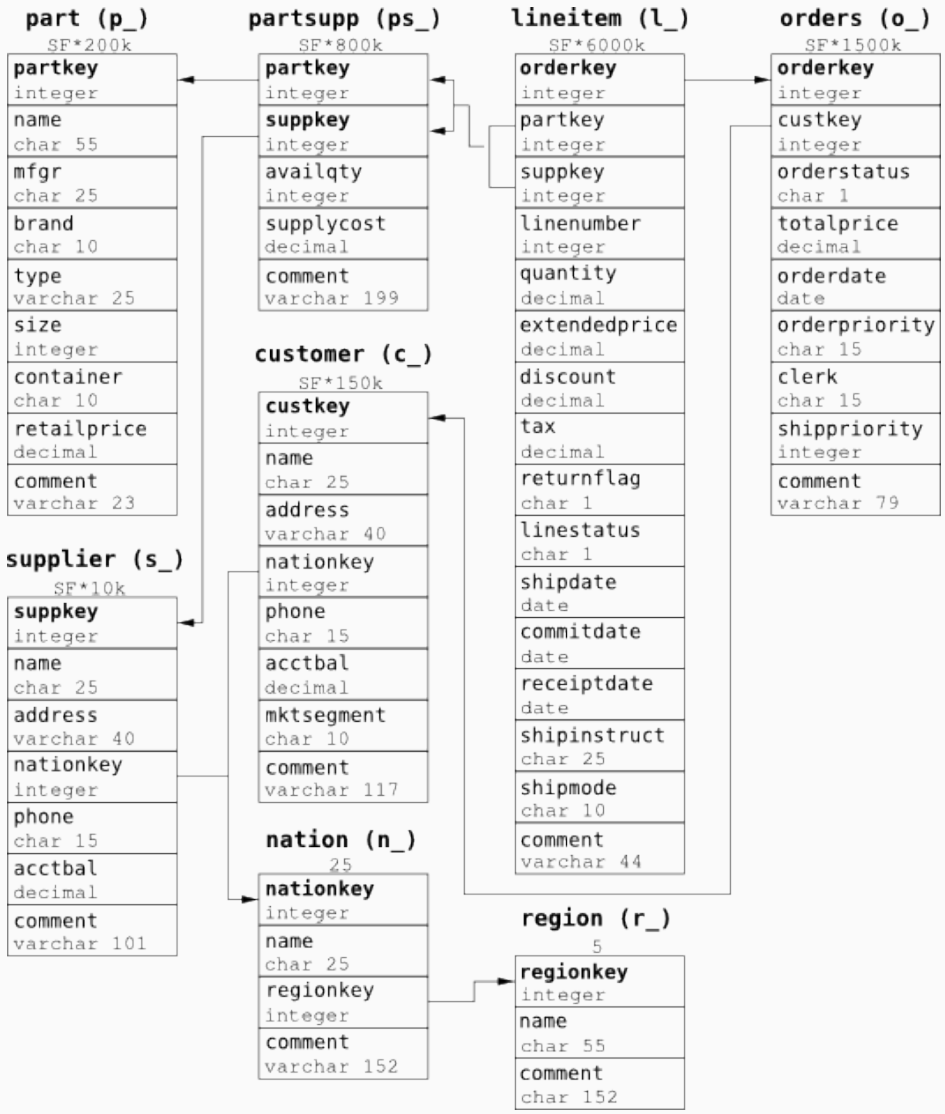
- SQL-92: Grundlagen
- SQL:1999: rekursive CTEs, grouping sets
- SQL:2003: Window Funktionen
- SQL:2006: XML
- SQL:2008: Merge (“Upsert”)
- SQL:2011: time travel, mehr Window Funktionen
- SQL:2016: JSON, Mustererkennung in Zeitserien

Ein Standard?

- neueren Versionen des SQL Standards sind sehr groß und die meisten Systeme unterstützen nur Teilmengen davon
- trotz der Größe sind nicht alle Aspekte spezifiziert
- außerdem gibt es auch Systeme, die vom Standard in Syntax (z.B. Outer Joins in Oracle) und Semantik (leerer String vs. NULL) abweichen
- im folgenden werden wir PostgreSQL verwenden, das sich weitestgehend an den Standard hält

TPC-H

- weit verbreiteter Benchmark für ad-hoc Anfragen
- Datensatz und 22 Anfragen
- Datensatzgenerator ist verfügbar, Größe ist konfigurierbar (*scale factor* 1 ≈ 1 GB)



Schema

```
create table lineitem (
    l_orderkey integer not null,
    l_partkey integer not null,
    l_suppkey integer not null,
    l_linenumber integer not null,
    l_quantity decimal(12,2) not null,
    l_extendedprice decimal(12,2) not null,
    l_discount decimal(12,2) not null,
    l_tax decimal(12,2) not null,
    l_returnflag char(1) not null,
    l_linestatus char(1) not null,
    l_shipdate date not null,
    l_commitdate date not null,
    l_receiptdate date not null,
    l_shipinstruct char(25) not null,
    l_shipmode char(10) not null,
    l_comment text not null
);
```

Datentypen in PostgreSQL: Zahlen

- *Integer mit Vorzeichen*: `smallint` (2 bytes), `integer` (4 bytes), `bigint` (8 bytes)
- *Festkommazahlen mit fester Größe*: `numeric(scale,precision)`, *scale* ist die Anzahl der dezimalen Ziffern, *precision* ist die Anzahl der Nachkommastellen
- *Zahlen beliebiger Größe*: `numeric` (Dezimalzahl beliebiger Größe, sehr langsam)
- *Fließkommazahlen* (IEEE 754): `float` (4 bytes), `double precision` (8 bytes)

Datentypen in PostgreSQL: Nicht-Numerische Typen

- *Zeichenketten*: `varchar(n)` (Maximallänge *n*), `char(n)` (Maximallänge *n*, mit Leerzeichen aufgefüllt, merkwürdige Semantik), `text` (beliebige Größe)
- *weitere Typen*: `bytea` (binäres Array), `timestamp` (8 bytes), `date` (4 bytes), `interval` (16 bytes), `boolean` (1 byte)
- PostgreSQL ist ein klassischer Row-Store, d.h. die Spalten einer Relation werden physikalisch zusammen gespeichert
- `https://www.postgresql.org/docs/current/static/datatype.html`

SQL Webinterface

`http://hyper-db.com/interface.html`

Selektion

```
select o_orderkey, o_orderdate, o_shippriority
from orders
where o_orderdate < date '1995-03-15';
```

Selektion, Joins

```
select o_orderkey, o_orderdate, o_shippriority,
       l_extendedprice, l_discount
from customer, orders, lineitem
where c_mktsegment = 'BUILDING'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-15'
and l_shipdate > date '1995-03-15';
```

Selektion, Join, Group By

```
select o_orderkey, o_orderdate, o_shippriority,
    sum(l_extendedprice * (1 - l_discount))
from customer, orders, lineitem
where c_mktsegment = 'BUILDING'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-15'
and l_shipdate > date '1995-03-15'
group by o_orderkey, o_orderdate, o_shippriority;
```

Selektion, Join, Group By, Ordering/Limit

```
select o_orderkey, o_orderdate, o_shippriority,
    sum(l_extendedprice * (1 - l_discount)) revenue
from customer, orders, lineitem
where c_mktsegment = 'BUILDING'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-15'
and l_shipdate > date '1995-03-15'
group by o_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
limit 10;
```

Selektion, Join, Group By, Ordering/Limit, Kommentare

```
select o_orderkey, o_orderdate, o_shippriority,
    sum(l_extendedprice * (1 - l_discount)) revenue
from customer, orders, lineitem /* this is a
multi line comment */
where c_mktsegment = 'BUILDING'
and c_custkey = o_custkey  -- single line comment
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-15'
and l_shipdate > date '1995-03-15'
group by o_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
limit 10;
```

- Wie viele Einzelartikel wurden von deutschen Händlern in Jahr 1995 ausgeliefert?
- Was sind die 10 Namen und Kontostände von Kunden aus der Region EUROPE im Marktsegment FURNITURE mit den höchsten Kontoständen?

Nullwerte

- jeder Wert kann NULL sein, außer das Attribut ist im Schema als `NOT NULL` definiert
- NULL kann unterschiedliches bedeuten, z.B. Geburtsdatum ist NULL:
  - Geburtsdatum ist unbekannt (“unknown”)
  - die Person wurde nie geboren (“not applicable”)
  - die Person wurde am Anfang der Zeit geboren (“application-specific special value”)

Nullwerte (2)

- Nullwerte werden in Ausdrücken durchgereicht: falls ein Operand NULL ist, ist das Ergebnis ebenfalls NULL
- explizites prüfen mit `IS NULL` bzw. `IS NOT NULL`
- dreiwertige Logik: **wahr**(w), **falsch**(f), and **unbekannt**(u):

not	
w	f
u	u
f	w

and	w	u	f
w	w	u	f
u	u	u	f
f	f	f	f

or	w	u	f
w	w	w	w
u	w	u	u
f	w	u	f

- Im Ergebnis einer SQL-Anfrage tauchen nur Tupel auf, für die die Auswertung der where-Klausel wahr ergibt
- Nullwerte können entstehen, wenn die Datenbank überhaupt keine NULL Werte enthält (z.B. bei Outer Joins)

Unteranfragen

- Unteranfragen können in SQL fast überall eingesetzt werden:

```
select n_name,
       (select count(*) from region)
from nation,
     (select *
      from region
      where r_name = 'EUROPE') region
where n_regionkey = r_regionkey
and exists (select 1
            from customer
            where n_nationkey = c_nationkey);
```

Korrelierte Unteranfragen

```
select avg(l_extendedprice)
from lineitem l1
where l_extendedprice =
      (select min(l_extendedprice)
       from lineitem l2
       where l1.l_orderkey = l2.l_orderkey);
```

- Korrelierte Unteranfragen beziehen sich auf Werte der äußeren Anfrage (`l1.l_orderkey`)
- Semantik: führe innere Anfrage für jedes Tupel der äußeren Anfrage aus
- führt oft zu quadratischer Laufzeit

Dekorrelation

Anfragen können so umgeschrieben, dass keine Korrelation vorkommt (nur wenige Systeme tun dies automatisch):

```
select avg(l_extendedprice)
from lineitem l1,
     (select min(l_extendedprice) m, l_orderkey
      from lineitem
      group by l_orderkey) l2
where l1.l_orderkey = l2.l_orderkey
and l_extendedprice = m;
```

Wie würde man die folgende Anfrage ohne Korrelation formulieren?

```
select c1.c_name
from customer c1
where c1.c_mktsegment = 'AUTOMOBILE'
or c1.c_acctbal >
      (select avg(c2.c_acctbal)
       from customer c2
       where c2.c_mktsegment = c1.c_mktsegment);
```

Mengenoperationen

- UNION, EXCEPT, and INTERSECT entfernen Duplikate

```
select n_name from nation where n_regionkey = 2
union
select n_name from nation where n_regionkey in (1, 2)
intersect
select n_name from nation where n_regionkey < 3
except
select n_name from nation where n_nationkey = 21;
```

Mengenoperationen (2)

- UNION ALL: $l + r$
- EXCEPT ALL: $\max(l - r, 0)$
- INTERSECT ALL (obskur): $\min(l, r)$

```
select n_name from nation where n_regionkey = 2
union all
select n_name from nation where n_regionkey in (1, 2)
intersect all
select n_name from nation where n_regionkey < 3
except all
select n_name from nation where n_nationkey = 21;
```

Nützliche Konstrukte

- `case` (conditional expressions)

```
select case when n_nationkey > 5
            then 'large' else 'small' end
from nation;
```

- `coalesce(a, b)`: ersetze a durch b, wenn a NULL ist
- `cast(a as integer)`: explizite Typkonversion
- `generate_series(begin, end)`: Integersequenz
- `random`: zufällige Fließkommazahl von 0.0 bis 1.0

```
select cast(random()*6 as integer)+1
from generate_series(1,10);
```

Zeichenketten

- Konkatenation:

```
select 'Daten' || 'banken';
```

- einfaches matching:

```
select 'abcfoo' like 'abc%';
```

- reguläre Ausdrücke:

```
select 'abcabc' ~ '(abc)*';
```

- Extraktion von Teilstrings:

```
select substring('abcfoo' from 3 for 2);
```

- Stringersetztung basierend auf regulärem Ausdruck:

```
select regexp_replace('ababfooab', '(ab)+', 'xy', 'g');
                       --    string    pattern   repl. flags)
```

Stichproben

- `bernoulli` Modus: zufällige Tupel
- `system` Modus: zufällige Seiten
- random seed kann mit `repeatable` spezifiziert werden
- in PostgreSQL $\geq$ 9.5:

```
select *
from nation tablesample bernoulli(5) -- 5%
     repeatable (9999);
```

- man kann auch beliebige Tupel ansehen:

```
select *
from nation
limit 10; -- 10 arbitrary rows
```

- Was ist der durchschnittliche Preis (`o_totalprice`) basierend of 1% aller Bestellungen?

Views and Common Table Expressions (CTE)

- analog zu Funktionen in normalen Programmiersprachen
- Wiederverwendung, Abstraktion, Lesbarkeit

```
create view nation_europe as
  select nation.*
  from nation, region
  where n_regionkey = r_regionkey
  and r_name = 'EUROPE';

with old_orders as (
    select *
    from orders
    where o_orderdate < date '2000-01-01')
  select count(*)
  from nation_europe, customer, old_orders
  where n_nationkey = c_nationkey
  and c_custkey = o_custkey;
```

Rekursive Common Table Expressions

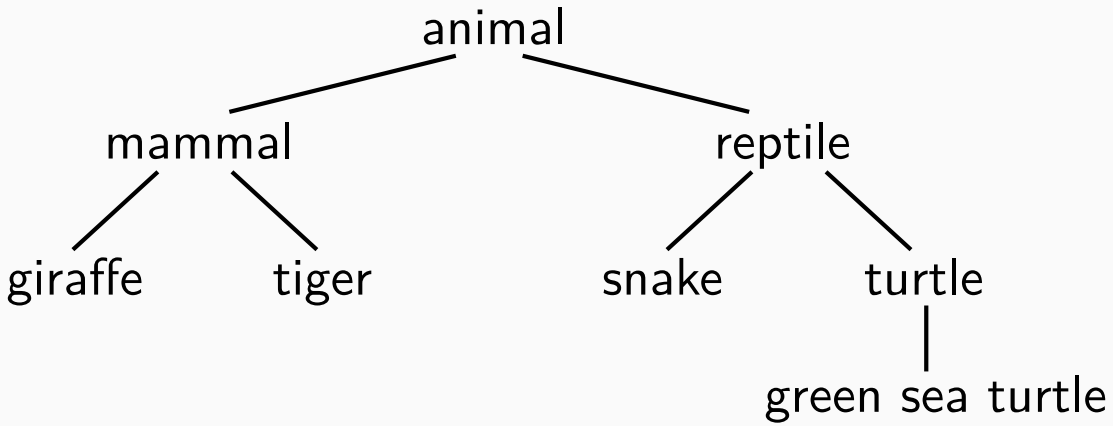
- eher Iteration als Rekursion
- ermöglicht das Traversieren von hierarchischen Daten beliebiger Tiefe (Joins erlauben nur eine konstante Anzahl von Schritten)
- macht SQL Turing-vollständig

```
with recursive r (i) as (
  -- nicht-rekursiver Term:
  select 1
  union all
  -- rekursiver Term:
  select i+1 from r where i < 5)
select * from r;
```

Rekursive Common Table Expressions: Auswertungsalgorithmus

workingTable = *evaluateNonRecursive*()
output *workingTable*
while *workingTable* is not empty
    workingTable = *evaluateRecursive*(*workingTable*)
    output *workingTable*

Beispiel: WITH RECURSIVE … UNION ALL




```
with recursive
 animal (id, name, parent) as (values (1, 'animal', null),
        (2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
        (5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
        (8, 'grean sea turtle', 7)),
  r as (select * from animal where name = 'turtle'
        union all
        select animal.*
        from r, animal
        where r.parent = animal.id)
select * from r;
```

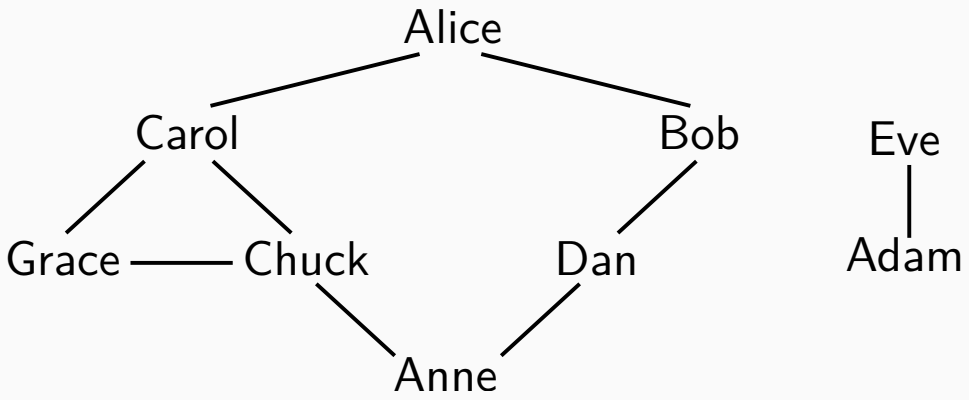
- Nachfolger von ’reptile’
- 10!
- Fibonacci Zahlen ($F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$)

WITH RECURSIVE … UNION

- für zyklische Daten terminiert UNION ALL nicht
- with recursive `[non-recursive]` **`union`** `[recursive]` ermöglicht das Traversieren von zyklischen Daten
- Auswertungsalgorithmus:

  $workingTable = unique(evaluateNonRecursive())$
  $result = workingTable$
  while *workingTable* is not empty
    $workingTable = unique(evaluateRecursive(workingTable)) \setminus result$
    $result = result \cup workingTable$
  output *result*

Beispiel: WITH RECURSIVE … UNION



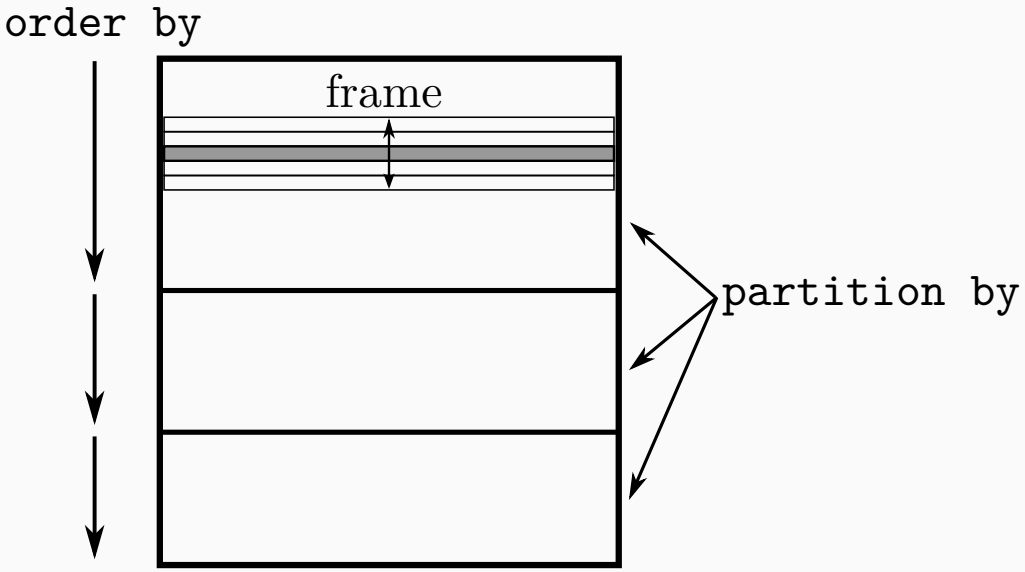

```
with recursive
 friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
   ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
   ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
 friendship (name, friend) as -- friendship is symmetric
    (select a, b from friends union all select b, a from friends),
 r as (select 'Alice' as name
       union
       select friendship.name from r, friendship
       where r.name = friendship.friend)
select * from r;
```

Window Funktionen

- vielseitige Funktionalität: Zeitreihenanalyse, Ranking, Top-k, Perzentile, gleitender Durchschnitt, kumulative Summen
- Window Funktionen werden *nach* fast den meisten Elementen der Anfrage (einschließlich `group by`) aber vor `order by`
- im Gegensatz zu Aggregation, ändert eine Window Funktion die Eingabe nicht, sondern berechnet eine zusätzliche Spalte (deswegen “Funktion”)

Syntax und Konzepte

```
select o_custkey, o_orderdate,
  sum(o_totalprice) over      -- window function
    (partition by o_custkey   -- partitioning clause
     order by o_orderdate     -- ordering clause
     range between unbounded preceding
      and current row)        -- framing clause
from customer;
```



Window Funktionen, die Framing ignorieren

- Rang:
  - `rank()`: Rang der aktuellen Zeile mit Lücken
  - `dense_rank()`: Rank der aktuellen Zeile ohne Lücken
  - `row_number()`: Zeilennummer
  - `ntile(n)`: Aufteilung in gleichverteilte Gruppen (1 bis `n`)
- Verteilung:
  - `percent_rank()`: relativer Rang der aktuellen Zeile ((rank - 1) / (total rows - 1))
  - `cume_dist()`: relativer Rang der peer[1] Gruppe ((number of rows preceding or peer with current row) / (total rows))
- Navigation in Partition:
  - `lag(expr, offset, default)`: evaluiere `expr` auf vorheriger Zeile in der aktuellen Partition
  - `lead(expr, offset, default)`: evaluiere `expr` auf folgender Zeile in der aktuellen Partition

[1] Zeilen mit identischen Partitionierungs- und Sortierungswerten sind *peers*.

- Vergeben sie Medallien an Kunden auf Basis der Anzahl von Bestellungen. Beispielausgabe:

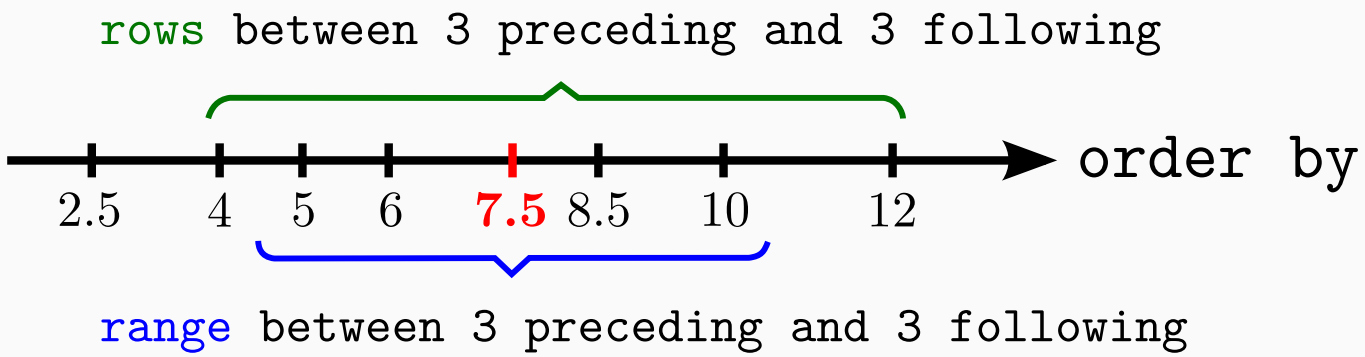
```
 custkey | count | medal
---------+-------+--------
    8761 |    36 | gold
   11998 |    36 | gold
    8362 |    35 | bronze
    4339 |    35 | bronze
     388 |    35 | bronze
    3151 |    35 | bronze
    9454 |    35 | bronze
```

- jährliche (`extract(year from o_orderdate)`) Änderung des Umsatzes (`sum(o_totalprice)`) in Prozent, Beispielausgabe:

```
  y   |     revenue     | pctchange
------+-----------------+-----------
 1992 | 3249822143.71 |
 1993 | 3186680293.06 |      -1.94
 1994 | 3276391729.79 |       2.82
 1995 | 3269894993.32 |      -0.20
 1996 | 3227878999.30 |      -1.28
 1997 | 3212138221.07 |      -0.49
 1998 | 1933789650.38 |     -39.80
```

Framing

- `current row`: aktuelle Zeile (inklusive peers im `range` Modus)
- `unbounded preceding`: erste Zeile in der Partition
- `unbounded following`: letzte Zeile in der Partition



Default Frame

- default frame ohne `order by` Ausdruck: `range between unbounded preceding and current row`
- default frame mit `order by` Ausdruck: `range between unbounded preceding and unbounded following` (komplette Partition)

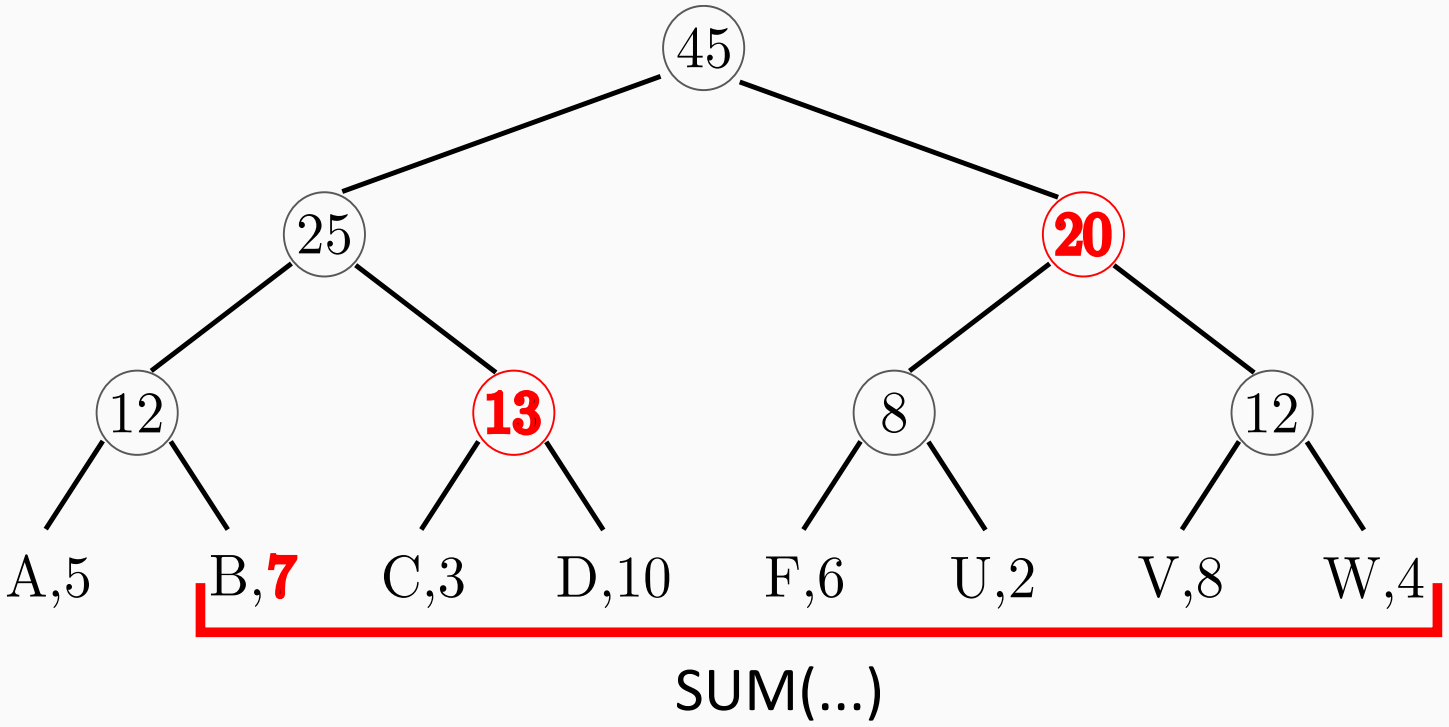
Window Funktionen Mit Framing

- Aggregate (`min`, `max`, `sum`, ...):
  berechne Aggregate über alle Tupel im aktuellen Frame
- Navigation im Frame:
  `first_value(expr)`, `last_value(expr)`,
  `nth_value(expr, nth)`: evaluiere `expr` auf
  erstem/letzten/`nth` Tupel des Frames

- Kumulativer Umsatz (`sum(o_totalprice)`) über die Zeit (`o_orderdate`)
- Kumulativer Umsatz (`sum(o_totalprice)`) nach Kunden aus GERMANY nach Jahr (`extract(year from o_orderdate)`), Beispielausgabe:

```
 custkey |  yr  | running_sum
---------+------+-------------
      62 | 1992 |   169991.32
      62 | 1993 |   344376.79
      62 | 1994 |   433638.98
      62 | 1995 |   960047.31
      62 | 1996 |  1372061.28
      62 | 1997 |  1658247.25
      62 | 1998 |  2055669.94
      71 | 1992 |   403017.41
      71 | 1993 |   751256.86
      71 | 1994 |  1021446.72
```

Efficient Evaluation Using Segment Tree



Statistische Aggregate

- `stddev_samp(expr)`: Standardabweichung
- `corr(x, y)`: Korrelation
- `regr_slope(y, x)`: lineare Regression (Steigung)
- `regr_intercept(y, x)`: lineare Regression (y-Achsenabschnitt)

Ordered-Set Aggregate

- sortierende Aggregatfunktionen mit spezieller Syntax
- `mode()`: häufigster Wert
- `percentile_disc(p)`: diskreter Perzentil ($p \in [0, 1]$)
- `percentile_cont(p)`: interpolierendes Perzentil ($p \in [0, 1]$) auf numerischen Daten

```
select percentile_cont(0.5)
    within group (order by o_totalprice)
from orders;
```

```
select o_custkey,
percentile_cont(0.5) within group (order by o_totalprice)
from orders
group by o_custkey;
```

Grouping Sets, Rollup, Cube

- Aggregation über gleichzeitig über mehrere Dimensionen, z.B. Umsatz nach Jahr, Kunde, Zulieferer
- explizit:
  `group by grouping sets ((a, b), (a), ())`
- hierarchisch: `group by rollup (a, b)`
- beide sind äquivalent zu:

```
select a, b, sum(x) from r group by a, b
union all
select a, null, sum(x) from r group by a
union all
select null, null, sum(x) from r;
```

- alle (2^n) Gruppierungen:
  `group by cube (a, b)` entspricht
  `group by grouping sets ((a, b), (a), (b), ())`

- Umsatz (`sum(o_totalprice)`): gesamt, nach Region (`r_name`), nach Land (`n_name`), Beispielausgabe:

```
     revenue     |   region      |        nation
-----------------+---------------+-----------------------
   836330704.31 | AFRICA        | ALGERIA
   902849428.98 | AFRICA        | ETHIOPIA
   784205751.27 | AFRICA        | KENYA
   893122668.52 | AFRICA        | MOROCCO
   852278134.31 | AFRICA        | MOZAMBIQUE
  4268786687.39 | AFRICA        |
...
 21356596030.63 |               |
```

Referenzen

- SQL reference (PostgreSQL): `https://www.postgresql.org/docs/current/static/sql.html`
- modern SQL: `https://modern-sql.com/`
- basic SQL (in German):
  Datenbanksysteme: Eine Einführung, Alfons Kemper und Andre Eickler, 10th edition, 2015
- Joe Celko's SQL for Smarties, Joe Celko, 5th edition, 2014
- SQL cookbook, Anthony Molinaro, 2005