SQL

SQL

Generelle Anmerkungen

- SQL: Structured Query Language
- Früherer Name war SEQUEL
- Standardisierte Anfragesprache für relationale DBMS: SQL-89, SQL-92, SQL-99
- SQL ist eine deklarative Anfragesprache

Teile von SQL

- Vier große Teile:
  - DRL: Data Retrieval Language
  - DML: Data Manipulation Language
  - DDL: Data Definition Language
  - DCL: Data Control Language

- Die DRL enthält die Kommandos, um Anfragen stellen zu können
- Eine einfache Anfrage besteht aus den drei Klauseln **select**, **from** und **where**

```
select Liste von Attributen
from   Liste von Relationen
where  Prädikat;
```

Ein einfaches Beispiel

Studenten

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Anfrage: “Gib mir die gesamte Information über alle Studenten”

```
select *
from   Studenten;
```

Ergebnis

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Attribute selektieren

Anfrage: “Gib mir die Matrikelnr und den Namen aller Studenten”

```
select MatrNr, Name
from   Studenten;
```

MatrNr	Name
24002	Xenokrates
25403	Jonas
26120	Fichte
26830	Aristoxenos
27550	Schopenhauer
28106	Carnap
29120	Theophrastos
29555	Feuerbach

Duplikateliminierung

- Im Gegensatz zur relationalen Algebra eliminiert SQL keine Duplikate
- Falls Duplikateliminierung erwünscht ist, muss das Schlüsselwort **distinct** benutzt werden

Beispiel

```
select Semester
from   Studenten;
```

Semester
18
12
10
8
6
3
2
2

```
select distinct Semester
from   Studenten;
```

Semester
18
12
10
8
6
3
2

Anfrage: “Gib mir alle Informationen über Studenten in einem Semester größer als 10”

```
select *
from   Studenten
where  Semester > 10;
```

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12

Prädikate

- Prädikate in der where-Klausel können logisch kombiniert werden mit: AND, OR, NOT
- Als Vergleichsoperatoren können verwendet werden: $=, <, \leq, >, \geq$, between, like

Beispiel für Between

Anfrage: “Gib mir die Namen aller Studenten, die im zweiten bis vierten Semester sind”

```
select Name
from   Studenten
where  Semester between 2 and 4;
```

ist äquivalent zu

```
select Name
from   Studenten
where  Semester >= 2
and    Semester <= 4;
```

Vergleich mit Werteliste

Vergleiche mit mehreren Werten können mit `in` durchgeführt werden

```
select *
from   Studenten
where  Semester in (1, 2, 3, 4);
```

Stringvergleiche

- Stringkonstanten müssen in einfachen Anführungszeichen eingeschlossen sein

  Anfrage: “Gib mir alle Informationen über den Studenten mit dem Namen Feuerbach”

```
select *
from   Studenten
where  Name = 'Feuerbach';
```

Suche mit Jokern (Wildcards)

Anfrage: “Gib mir alle Informationen über Studenten, deren Namen mit einem S anfängt”

```
select *
from   Studenten
where  Name like 'S%';
```

Mögliche Joker

- _ steht für ein beliebiges Zeichen
- % steht für eine beliebige Zeichenkette (auch der Länge 0)

Nullwerte

- In SQL gibt es einen speziellen Wert **NULL**
- Dieser Wert existiert für alle verschiedenen Datentypen und repräsentiert unbekannte, nicht verfügbare oder nicht anwendbare Werte
- Auf NULL wird folgendermaßen geprüft:

```
select *
from   Professoren
where  Raum is null;
```

Nullwerte(2)

- Nullwerte werden in arithmetischen Ausdrücken durchgereicht: falls mindestens ein Operand NULL ist, ist das Ergebnis ebenfalls NULL
- SQL hat eine dreiwertige Logik: **wahr**(w), **falsch**(f) und **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

Mehrere Relationen

- Falls mehrere Relationen in der from-Klausel auftauchen, werden sie mit einem Kreuzprodukt verbunden
- Beispiel:

Anfrage: “Gib alle Vorlesungen und Professoren aus”

```
select *
from   Vorlesungen, Professoren;
```

- Kreuzprodukte machen meistens keinen Sinn, interessanter sind Joins
- Joinprädikate werden in der where-Klausel angegeben:

```
select *
from   Vorlesungen, Professoren
where  gelesenVon = PersNr;
```

Joins(2)

- Es dürfen beliebig viele Relationennamen in der from-Klausel stehen
- Wenn keine Kreuzprodukte erwünscht, sollten alle in der where-Klausel gejoint werden
- Die verschiedenen Joinvarianten aus der relationalen Algebra sind auch in SQL möglich:

```
select *
from   R1 [cross | inner | natural | left outer | right outer | full outer]
       join R2 [on R1.A = R2.B];
```

Joins(3)

- Weiteres Problem: Namenskollisionen (gleichnamige Attribute in verschiedenen Relationen) müssen aufgelöst werden
- Beispiel: Join von
  - `Studenten(MatrNr, Name, Semester)`
  - `hören(MatrNr, VorlNr)`
  - `Vorlesungen(VorlNr, Titel, SWS, gelesenVon)`

Qualifizierte Attributnamen

- In dieser Beispielanfrage muss spezifiziert werden, woher MatrNr und VorlNr herkommen sollen
- Dazu schreibt man den Relationennamen vor den Attributnamen

```
select *
from   Studenten, hören, Vorlesungen
where  Studenten.MatrNr = hören.MatrNr
and    hören.VorlNr = Vorlesungen.VorlNr;
```

Kurzform

- Um sich Tipparbeit zu sparen, können die Relationen auch umbenannt werden

```
select *
from   Studenten S, hören H, Vorlesungen V
where  S.MatrNr = H.MatrNr
and    H.VorlNr = V.VorlNr;
```

Mengenoperationen

- In SQL gibt es auch die üblichen Operationen auf Mengen: Vereinigung, Schnitt und Differenz
- Setzen wie in der relationalen Algebra gleiches Schema der verknüpften Relationen voraus

Vereinigung

Prof1

PersNr	Name
2125	Sokrates
2126	Russel

Prof2

PersNr	Name
2126	Russel
2137	Kant

Anfrage: “Vereinige beide Listen”

```
select * from Prof1
union
select * from Prof2;
```

PersNr	Name
2125	Sokrates
2126	Russel
2137	Kant

Duplikateliminierung

- Im Gegensatz zu **select** eliminiert **union** automatisch Duplikate
- Falls Duplikate im Ergebnis erwünscht sind, muss der **union all**-Operator benutzt werden

Prof1

PersNr	Name
2125	Sokrates
2126	Russel

Prof2

PersNr	Name
2126	Russel
2137	Kant

Anfrage: “Welche Professoren sind auf beiden Listen”

```
select * from Prof1
intersect
select * from Prof2;
```

PersNr	Name
2126	Russel

Mengendifferenz

Prof1

PersNr	Name
2125	Sokrates
2126	Russel

Prof2

PersNr	Name
2126	Russel
2137	Kant

Anfrage: “Welche Professoren sind auf der ersten, aber nicht auf der zweiten Liste?”

```
select * from Prof1
except
select * from Prof2;
```

PersNr	Name
2125	Sokrates

Sortierung

- Tupel in einer Relation sind nicht (automatisch) sortiert
- Das Ergebnis einer Anfrage kann mit Hilfe der **order by**-Klausel sortiert werden
- Es kann aufsteigend oder absteigend sortiert werden (voreingestellt ist aufsteigend)

Beispiel

```
select *
from   Studenten
order by Semester desc, Name;
```

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29555	Feuerbach	2
29120	Theophrastos	2

Geschachtelte Anfragen

- Anfragen können in anderen Anfragen geschachtelt sein, d.h. es kann mehr als eine select-Klausel geben
- Geschachteltes select kann in der where-Klausel, in der from-Klausel und sogar in einer select-Klausel selbst auftauchen
- Im Prinzip wird in der “inneren” Anfrage ein Zwischenergebnis berechnet, das in der “äußeren” benutzt wird

Select in Where-Klausel

- Zwei verschiedene Arten von Unteranfragen: korrelierte und unkorrelierte
- unkorreliert: Unteranfrage bezieht sich nur auf “eigene” Attribute
- korreliert: Unteranfrage referenziert auch Attribute der äußeren Anfrage

Anfrage: “Gib mir die Namen aller Studenten, die die Vorlesung 5001 hören”

```
select S.Name
from   Studenten S
where  S.MatrNr in
       (select H.MatrNr
        from   hören H
        where  H.VorlNr = 5001);
```

- Unteranfrage wird einmal ausgewertet, für jedes Tupel der äußeren Anfrage wird geprüft, ob die MatrNr im Ergebnis der Unteranfrage vorkommt

Anfrage: “Finde alle Professoren, für die Assistenten mit verschiedenen Fachgebieten arbeiten”

```
select distinct P.Name
from   Professoren P, Assistenten A
where  A.Boss = P.PersNr
and    exists
       (select *
        from   Assistenten B
        where  B.Boss = P.PersNr
        and    A.Fachgebiet <> B.Fachgebiet);
```

- Für jedes Tupel der äußeren Anfrage hat innere Anfrage verschiedene Werte, das exists-Prädikat ist wahr, wenn die Unteranfrage mind. ein Tupel enthält

Existenz- und Allquantor in SQL

- `exists` und `not exists` entspricht semi- und anti-join
- negierter anti-join kann als All-Quantor verwendet werden
- Beispiel: Studenten, die alle 4-SWS Vorlesungen gehört haben

```
select s.*
from   Studenten s
where  not exists
       (select *
        from   Vorlesungen v
        where  v.SWS = 4 and not exists
               (select *
                from   hören h
                where  h.VorlNr = v.VorlNr and h.MatrNr = s.MatrNr));
```

Quantifizierte Vergleiche

- `=some`/`=all`/`>some`/`>all` etc. sind eine Verallgemeinerung von `in`
- Beispiel: Studenten mit dem höchsten Semester

```
select Name
from   Studenten
where  Semester >= all
       (select Semester
        from   Studenten);
```

- Beispiel: Studenten nicht im höchsten Semester

```
select Name
from   Studenten
where  Semester < some
       (select Semester
        from   Studenten);
```

Andere geschachtelte Selects

- Beim Schachteln eines selects in einer select-Klausel muss darauf geachtet werden, dass nur ein Tupel mit einem Attribut zurückgeliefert wird
- Beim Schachteln in einer from-Klausel sind korrelierte Unteranfragen (je nach DBMS) oft nicht erlaubt

Aggregatfunktionen

- Attributwerte (oder ganze Tupel) können auf verschiedene Arten zusammengefasst werden
  - Zählen: count()
  - Aufsummieren: sum()
  - Durchschnitt bilden: avg()
  - Maximum finden: max()
  - Minimum finden: min()

Beispiel

Studenten

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

```
select count(*)
from   Studenten;
```

1
8

Beispiel(2)

Studenten

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

```
select count(distinct Semester)
from   Studenten;
```

1
7

Anfrage:“Gib mir den Studenten mit der größten MatrNr”

```
select Name, max(MatrNr)
from   Studenten;
```

- Funktioniert so nicht!!!

Min/Max(2)

- Aggregatfunktionen reduzieren alle Werte einer Spalte zu einem **einzigen** Wert
- Für das Attribut `MatrNr` sagen wir dem DBMS, dass das Maximum genommen werden soll
- Für das Attribut `Name` geben wir dem DBMS keinerlei Information, wie die ganzen verschiedenen Namen auf einen reduziert werden sollen

Min/Max(3)

- Wie geht es richtig?
- Mit Hilfe einer geschachtelten Anfrage:

```
select MatrNr, Name
from   Studenten
where  MatrNr =
       (select max(MatrNr)
        from   Studenten);
```

Gruppieren

- Manchmal möchte man Tupel in verschiedene Gruppen aufteilen und diese Gruppen getrennt aggregieren

  Anfrage:“Für jede Vorlesung zähle die Anzahl der teilnehmenden Studenten”

```
select VorlNr, count(*) as Anzahl
from   hören
group by VorlNr;
```

Ergebnis

hören

MatrNr	VorlNr
27550	4052
26120	5001
27550	5001
29120	5001
29555	5001
25403	5022
29555	5022
28106	5041
29120	5041
29120	5049
28106	5052
28106	5216
28106	5259

$\rightarrow$

Ergebnis

VorlNr	Anzahl
4052	1
5001	4
5022	2
5041	2
5049	1
5052	1
5216	1
5259	1

Gruppieren(2)

- Alle Attribute, die nicht in der group by-Klausel auftauchen, dürfen nur aggregiert in der select-Klausel stehen
- Z.B. ist folgende Anfrage nicht korrekt (aus dem gleichen Grund wie die erste max-Anfrage):

```
select gelesenVon, Titel, count(*) as Anzahl
from   Vorlesungen
group by gelesenVon;
```

Having

- Die where-Klausel wird vor dem Gruppieren ausgewertet
- Wenn nach der Gruppierung noch weiter ausgefiltert werden soll, muss **having**-Klausel benutzt werden

Anfrage:“Finde alle Professoren, die mehr als drei Vorlesungen halten”

```
select gelesenVon, count(VorlNr) as AnzVorl
from   Vorlesungen
group by gelesenVon
having count(*) > 3;
```

Sichten

- Gehören eigentlich zur DDL
- Werden aber oft verwendet, um Anfragen übersichtlicher zu gestalten, deswegen besprechen wir sie hier
- Stellen eine Art “virtuelle Relation” dar
- Zeigen einen Ausschnitt aus der Datenbank

Sichten(2)

- Vorteile
  - Vereinfachen den Zugriff für bestimmte Benutzergruppen
  - Können eingesetzt werden, um den Zugriff auf die Daten einzuschränken
- Nachteile
  - Nicht auf allen Sichten können Änderungsoperationen ausgeführt werden

Komplizierte Anfrage

Anfrage:“Finde die Namen aller Professoren, die Vorlesungen halten, die mehr als die durchschnittliche Anzahl an SWS haben und die mehr als drei Assistenten beschäftigen”

- Es wird nicht gleich alles auf einmal gemacht, sondern in kleinere übersichtlichere Teile heruntergebrochen
- Diese Teile werden mit Hilfe von Sichten realisiert

Komplizierte Anfrage(2)

- Finde alle Vorlesungen mit überdurchschnittlich vielen SWS:

```
create view ÜberSchnittSWS as
select VorlNr, gelesenVon
from   Vorlesungen
where  SWS >
       (select avg(SWS)
        from   Vorlesungen);
```

Komplizierte Anfrage(3)

- Finde (die PersNr) aller Professoren mit mehr als drei Assistenten:

```
create view VieleAssistenten as
select Boss
from   Assistenten
group by Boss
having count(*) > 3;
```

Komplizierte Anfrage(4)

- Jetzt wird alles zusammengesetzt (dabei können Sichten wie eine herkömmliche Relation angesprochen werden)

```
select Name
from   Professoren
where  PersNr in
       (select gelesenVon
        from   ÜberSchnittSWS)
and    PersNr in
       (select Boss
        from   VieleAssistenten);
```

Lokale Sichten

- Sichten sind global sichtbar, Sichtennamen müssen eindeutig sein
- häufig unpraktisch für die Formulierung von Anfragen
- statt dessen: Sichten nur für eine einzelne Anfrage

```
with ÜberSchnittSWS as (select ...),
     VieleAssistenten as (select ...)
select Name
from   Professoren
where  PersNr in
       (select gelesenVon
        from   ÜberSchnittSWS)
and    PersNr in
       (select Boss
        from   VieleAssistenten);
```

Fallunterscheidungen mit CASE

- einfaches case

```
select case Rang when 'C4' then 'Ordinarius'
                when 'C3' then 'Extraordinarius'
                else 'anders' end
from   Professoren
```

- allgemeines case

```
select case when Semester > 10 then 'Langzeitstudent'
            when Semester < 4 then 'Anfänger'
            else 'mittel' end
from   Studenten
```

Arbeiten mit Text

- Verbinden

```
select 'a' || 'b';
```

- Zerlegen

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

Typumwandlung mit CAST

- explizite Typumwandlung mit `CAST(value AS TYPE)`
- Beispiel

```
select cast(MatrNr as text) ||
' ' || Name
from   Studenten
```

- Achtung: Fehler, wenn Umwandlung nicht möglich ist

NULL-Behandlung

NULL-Werte müssen oft gesondert behandelt werden

- Vergleiche mit NULL liefern NULL
- Test mit `IS NULL`
- expliziter Vergleich mit `x IS NOT DISTINCT FROM y`
- Maskierung: `COALESCE(x, -1)`
- Erzeugung: `NULLIF(x, -1)`

Problem: Alle Voraussetzungen einer Vorlesung finden

```
select Vorgänger
from   voraussetzen, Vorlesungen
where  Nachfolger = VorlNr and
       Titel = 'Der Wiener Kreis'
```

Findet nicht die indirekten Voraussetzungen

Wir können einen Schritt weiter gehen

```
with
  step0 as (
    select Vorgänger
    from   voraussetzen, Vorlesungen
    where  Nachfolger = VorlNr and
           Titel = 'Der Wiener Kreis'),
  step1 as (
    select v.Vorgänger
    from   step0 s, voraussetzen v
    where  s.Vorgänger = v.Nachfolger)
select * from step0 union all select * from step1
```

Wir wollen das so oft machen wie nötig

WITH RECURSIVE statements können sich selbst lesen

```
with recursive rec as (
  select ...                -- Basisfall
  union all
  select ... from rec ...   -- Erweiterung
)
select * from rec
```

Erweiterung wird solange ausgewertet, wie die Erweiterung Tupel findet

Rekursion in SQL (4)

```
with recursive transvor as (
  select Vorgänger
  from   voraussetzen, Vorlesungen
  where  Nachfolger = VorlNr and
         Titel = 'Der Wiener Kreis'
  union all
  select v.Vorgänger
  from   transvor s, voraussetzen v
  where  s.Vorgänger = v.Nachfolger)
select * from transvor
```

Entspricht der Anfrage von Rekursion (2), verallgemeinert für transitive Schritte.

- DML enthält Befehle, um
  - Daten einzufügen
  - Daten zu löschen
  - Daten zu ändern

Daten einfügen

- Daten werden mit dem **insert**-Befehl eingefügt
- Einfügen von konstanten Werten
  - Unter Angabe aller Attributwerte:

```
insert into Professoren
values (2138, 'Kossmann', 'C4', 123);
```

  - Weglassen von Attributwerten:

```
insert into Professoren(PersNr, Name, Rang)
values (2138, 'Kossmann', 'C4');
```

Daten einfügen(2)

- Daten aus anderen Relationen kopieren

```
insert into Professoren(PersNr, Name, Rang)
select PersNr, Name, 'C3'
from   Assistenten
where  PersNr = 3002;
```

Daten ändern

- Änderungen werden mit dem **update**-Befehl vorgenommen

```
update Professoren
set    Raum = 121
where  PersNr = 2138;
```

Daten löschen

- Der **delete**-Befehl löscht Daten

```
delete from Professoren
where PersNr = 2138;
```

- Vorsicht! Das Weglassen der where-Klausel löscht den Inhalt der gesamten Relation

```
delete from Professoren;
```

Änderbarkeit von Sichten

- In SQL
  - nur eine Basisrelation
  - Schlüssel muss vorhanden sein
  - keine Aggregatfunktionen, Gruppierung und Duplikateliminierung
- Allgemein:



- Mit Hilfe der DDL kann das Schema einer Datenbank definiert werden
- Enthält auch Befehle, um den Zugriff auf Daten zu kontrollieren

Relationen anlegen

- Mit dem **create table**-Befehl werden Relationen angelegt

```
create table Professoren (
  PersNr integer,
  Name   varchar(30),
  Rang   char(2),
  Raum   integer
);
```

Datentypen

- *Integer*: `smallint` (2 bytes), `integer` (4 bytes), `bigint` (8 bytes)
- *Festkommazahlen*: `numeric(precision,scale)` (*precision* ist die Gesamtzahl der Dezimalstellen, *scale* ist die Anzahl Ziffern nach dem Dezimaltrenner)
- *floating point numbers*: `float` (4 bytes), `double precision` (8 bytes)
- *strings*: `varchar(n)` (Maximallänge *n*), `char(n)` (Maximallänge *n*, mit Leerzeichen gepadded, seltsame Semantik), `text` (beliebige Länge)
- *andere Typen*: `bytea` (Binärdaten), `timestamp` (8 bytes), `date` (4 bytes), `interval` (16 bytes), `boolean` (1 byte)
- alle Typen können `NULL` sein, außer wenn `NOT NULL` angegeben wird
- smallint/bigint/text/bytea sind Erweiterungen, aber oft unterstützt (z.B. in PostgreSQL)

Schlüssel definieren

- Für jede Relation kann ein Primärschlüssel definiert werden

```
create table Professoren (
  PersNr integer,
  Name   varchar(30),
  Rang   char(2),
  Raum   integer,
  primary key (PersNr)
);
```

Integritätsbedingungen

- Zu den Aufgaben eines DBMS gehört es auch, die Konsistenz der Daten zu sichern
- Semantische Integritätsbedingungen beschreiben Eigenschaften der modellierten Miniwelt
- DBMS kann mit Hilfe von Constraints automatisch diese Bedingungen überprüfen

Constraints

- Neben Primärschlüsseln gibt es eine ganze Reihe weiterer Integritätsbedingungen:
  - not null
  - unique
  - check-Klauseln

Not Null Constraint

- Erzwingt, dass beim Einfügen von Tupeln bestimmte Attributwerte angegeben werden müssen
- Zwingend für Schlüssel

```
create table Professoren (
  PersNr integer not null primary key,
  Name   varchar(30) not null,
  Rang   char(2) not null,
  Raum   integer
);
```

Check-Klauseln

- Durch **check**-Klauseln kann der Wertebereich für Attribute eingeschränkt werden

```
create table Professoren (
  PersNr integer not null primary key,
  Name   varchar(30) not null,
  Rang   char(2) not null,
  Raum   integer,
  check (Raum > 0 and Raum < 99999)
);
```

Check-Klauseln(2)

- In Check-Klauseln können vollständige SQL-Anfragen angegeben werden

```
create table hören (
  MatrNr integer,
  VorlNr integer,
  check (MatrNr not in
         (select P.MatrNr
          from   prüfen P
          where  P.VorlNr = hören.VorlNr
          and    P.Note < 5)),
  primary key (MatrNr, VorlNr)
);
```

Referentielle Integrität

- R und S sind zwei Relationen mit den Schemata $\mathcal{R}$ bzw. $\mathcal{S}$
- κ ist Primärschlüssel von R
- Dann ist $\alpha \subset \mathcal{S}$ ein Fremdschlüssel, wenn für alle Tupel $s \in S$ gilt:
  - $s.\alpha$ enthält entweder nur Nullwerte oder nur Werte ungleich Null
  - Enthält $s.\alpha$ keine Nullwerte, so existiert ein Tupel $r \in R$ mit $s.\alpha = r.\kappa$
- Die Einhaltung dieser Eigenschaften wird *referentielle Integrität* genannt

Referentielle Integrität(2)

- In SQL kann referentielle Integrität durchgesetzt werden:

```
create table Professoren (
  PersNr integer primary key,
  …
);

create table Vorlesungen (
  VorlNr     integer primary key,
  …
  gelesenVon integer not null,
  foreign key (gelesenVon)
  references Professoren(PersNr)
);
```

Referentielle Integrität(3)

- Änderungen an Schlüsselattributen können automatisch propagiert werden
- **set null**: alle Fremdschlüsselwerte, die auf einen Schlüssel zeigen, der geändert oder gelöscht wird, werden auf NULL gesetzt
- **cascade**: alle Fremdschlüsselwerte, die auf einen Schlüssel zeigen, der geändert oder gelöscht wird, werden ebenfalls auf den neuen Wert geändert bzw. gelöscht

Referentielle Integrität(4)

```
create table Vorlesungen (
  VorlNr     integer primary key,
  …
  gelesenVon integer not null,
  foreign key (gelesenVon) references Professoren(PersNr)
    [on delete {set null | cascade}]
    [on update {set null | cascade}]
);
```

Vorgegebene Werte

- Wenn beim Einfügen ein Attributwert nicht spezifiziert wird, dann wird ein vorgegebener Wert (default value) eingesetzt
- Wenn kein bestimmter Wert vorgegeben wird, ist NULL default value

```
create table Assistenten (
  PersNr     integer not null primary key,
  Name       varchar(30) not null,
  Fachgebiet varchar(30) default 'Ideenlehre',
  Boss       integer
);
```

Indexe

- Indexe beschleunigen den Zugriff auf Relationen (verlangsamen allerdings Änderungsoperationen)
- Die meisten DBMS legen automatisch einen Index auf dem Primärschlüssel an (um schnell die Eindeutigkeit prüfen zu können)
- Weitere Details zu Indexen gibt es später

```
create [unique] index Indexname
on table Relation (
  Attribut [asc | desc],
  Attribut [asc | desc], …)
```

Objekte entfernen

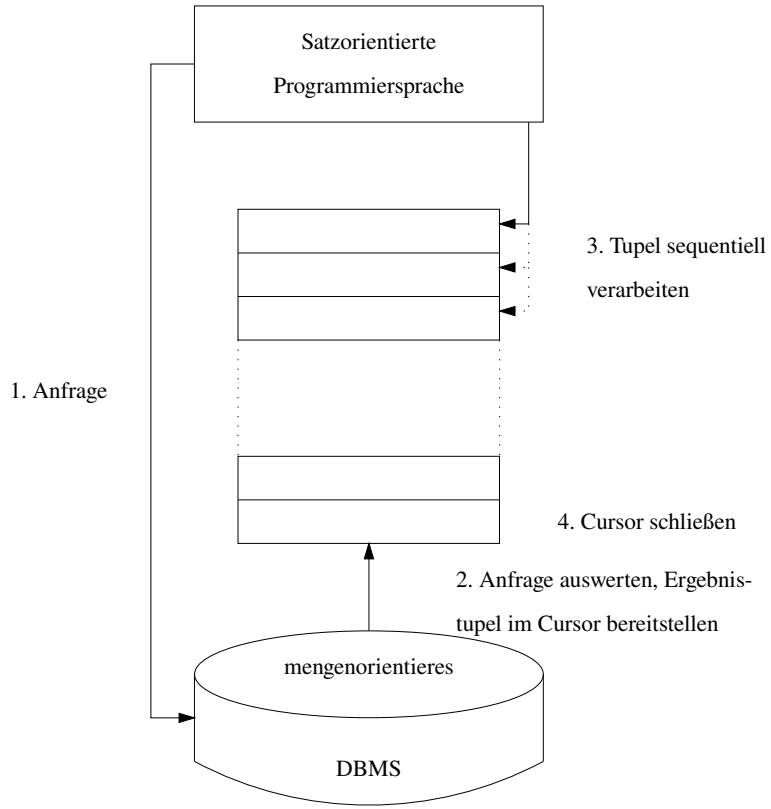
- Relationen, Sichten und Indexe können mit dem **drop**-Befehl wieder entfernt werden:
  - **drop table** *Relation*;
  - **drop view** *Sicht*;
  - **drop index** *Index*;

- Enthält Befehle, um den Fluss von Transaktionen zu steuern
- Eine Transaktion ist eine Menge von Interaktionen zwischen Anwendung/Benutzer und dem DBMS
- Wird später im Rahmen von Transaktionsverwaltung behandelt

Varianten von SQL

- Eine Datenbank kann nicht nur interaktiv benutzt werden
- SQL kann in andere Programmiersprachen eingebettet werden
- Problem: SQL ist mengenorientiert, die meisten Programmiersprachen nicht

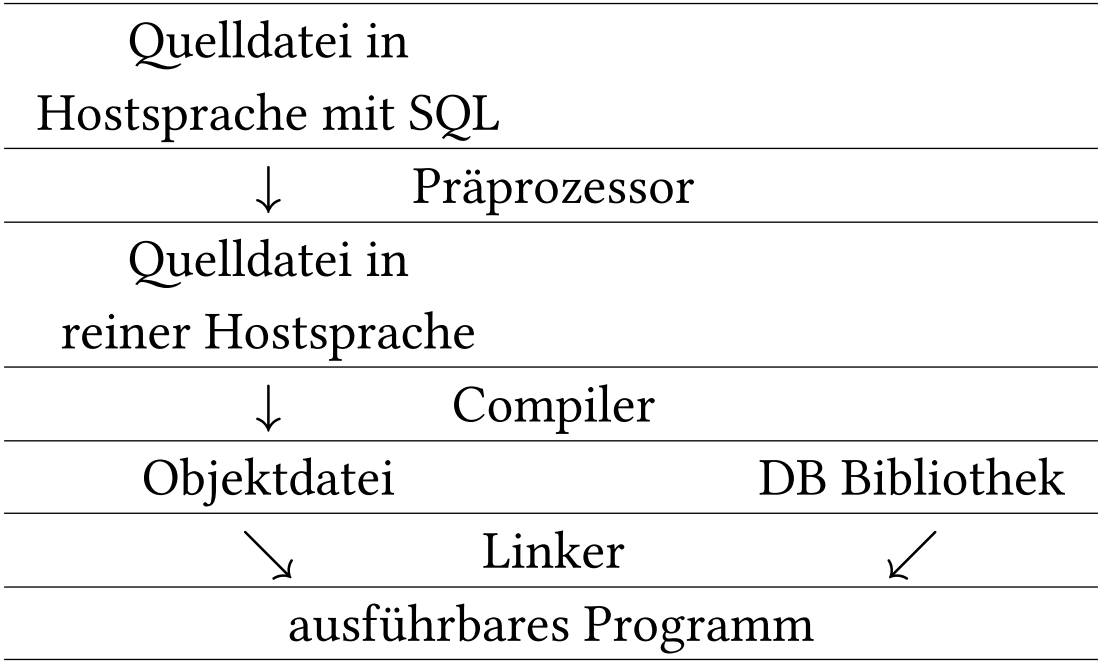
Anfragen in Anwendungen



Embedded SQL

- Hier werden SQL-Befehle direkt in die jeweilige Hostsprache eingebettet (z.B. C, C++, Java, etc.)
- SQL-Befehle werden durch ein vorangestelltes **EXEC SQL** markiert
- Sie werden vom Präprozessor durch Konstrukte der jeweiligen Sprache ersetzt

Embedded SQL(2)



Dynamic SQL

- Wird eingesetzt, wenn die Anfragen zur Übersetzungszeit des Programms noch nicht bekannt sind
- Standardisierte Schnittstellen
  - ODBC (Open Database Connectivity)
  - JDBC (für Java)
- Flexibler, aber üblicherweise etwas langsamer als Embedded SQL

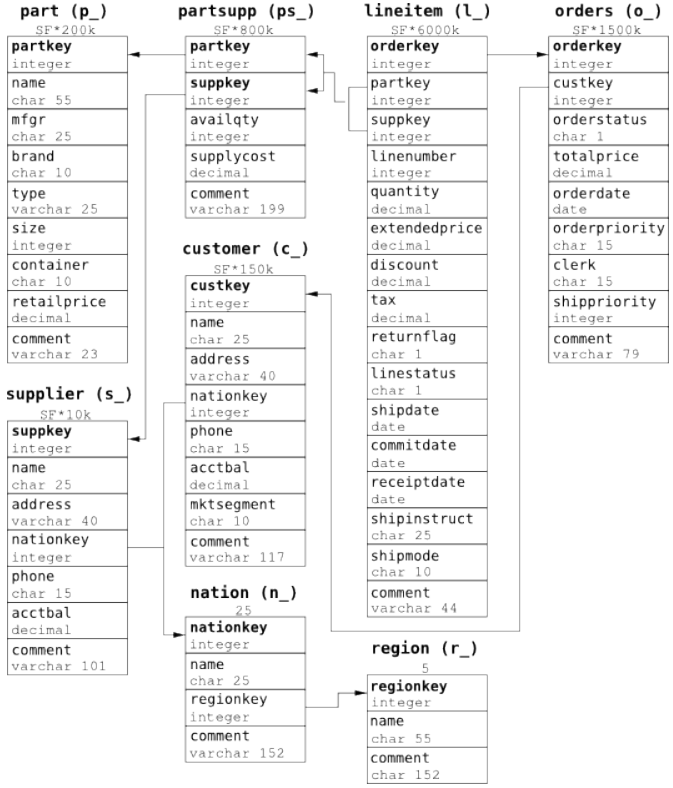
Zusammenfassung

- SQL ist *die* Standardsprache im Umgang mit relationalen Systemen
- SQL enthält Befehle zum Abrufen, Ändern, Einfügen und Löschen von Daten
- Es existieren weitere Befehle, um ein Schema zu definieren, den Zugriff zu kontrollieren und Transaktionen zu steuern

Die nächsten Folien gehen über den Einführungs-Stoff hinaus, enthalten aber einige nützliche Konstrukte

TPC-H Data Set

- TPC-H ist ein Ad-hoc-Benchmark für Decision Support
- zufällig generierter Datensatz, Generator ist frei verfügbar
- Größe ist konfigurierbar (*Scale Factor* 1 entspricht etwa 1 GB)



```
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
);
```

Laden von CSV-artigen Daten

```
$ head -n 1 lineitem.tbl
1|15519|785|1|17| ... |egular courts above the|
$ psql tpch
tpch=# \copy lineitem from lineitem.tbl delimiter '|'
ERROR:  extra data after last expected column
CONTEXT:  COPY lineitem, line 1: "1|15519|785|1|17|24386.67|0.04|0.02|N|O|
1996-03-13|1996-02-12|1996-03-22|DELIVER IN PERSON|TRUCK|egular courts above
the|"
tpch# \q
$ sed -i 's/|$//' lineitem.tbl
$ psql
tpch=# \copy lineitem from lineitem.tbl delimiter '|'
COPY 600572
```

- https://www.postgresql.org/docs/current/static/sql-copy.html

```
select l_orderkey, -- einzeiliger Kommentar
    sum(l_extendedprice * (1 - l_discount)) revenue,
    o_orderdate, o_shippriority
from customer, orders, lineitem /* das ist ein
mehrzeiliger Kommentar */
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 l_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
limit 10;
```

- Wie viele Artikel wurden 1995 von deutschen Lieferanten verschickt?
- Wie lauten die Namen und Kontostände der 10 Kunden aus EUROPE im Marktsegment FURNITURE mit dem höchsten Kontostand?

- Unteranfragen können (fast) überall auftreten:

```
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);
```

- eine Unteranfrage ist korreliert, wenn sie sich auf Tupel der äußeren Anfrage bezieht (`l1.l_orderkey`)
- naive Ausführung: innere Anfrage für jedes Tupel der äußeren Anfrage auswerten (quadratische Laufzeit, außer es existiert ein Index auf `l_orderkey`)

- Anfragen können so umgeschrieben werden, dass die Korrelation verschwindet (manche Systeme machen das 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;
```

- Dekorreliere die folgende Anfrage

```
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);
```

- UNION, EXCEPT und 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;
```

„Mengen“-Operatoren

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

```
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;
```

Weitere nützliche Konstrukte

- `case` (bedingte Ausdrücke)

```
select case when n_nationkey > 5
            then 'groß' else 'klein' end
from nation;
```

- `coalesce(a, b)`: NULL durch einen anderen Wert ersetzen
- `cast` (explizite Typumwandlung)
- `generate_series(begin,end)`
- `random` (zufällige Gleitkommazahl zwischen 0 und 1):

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

Arbeiten mit Strings

- Konkatenation:

```
select 'a' || 'b';
```

- einfacher Mustervergleich:

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

- Mustervergleich mit regulären Ausdrücken:

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

- Teilstring extrahieren:

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

- Ersetzung mittels regulärer Ausdrücke: `(str, pattern, replacement, flags)`

Arbeiten mit Strings

```
select regexp_replace('ababfooab', '(ab)+', 'xy', 'g');
```

Stichproben (Sampling)

- Sampling-Modi: `bernoulli` (zufällige Tupel auswählen) oder `system` (zufällige Seiten auswählen)
- Zufalls-Seed kann optional mit `repeatable` festgelegt werden
- unterstützt von PostgreSQL ≥ 9.5:

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

- man kann sich auch beliebige Tupel liefern lassen:

```
select *
from nation
limit 10; -- 10 beliebige Zeilen
```

- Berechne den Durchschnitt von `o_totalprice` anhand einer Stichprobe von 1% aller Bestellungen

Rekursive Common Table Expressions

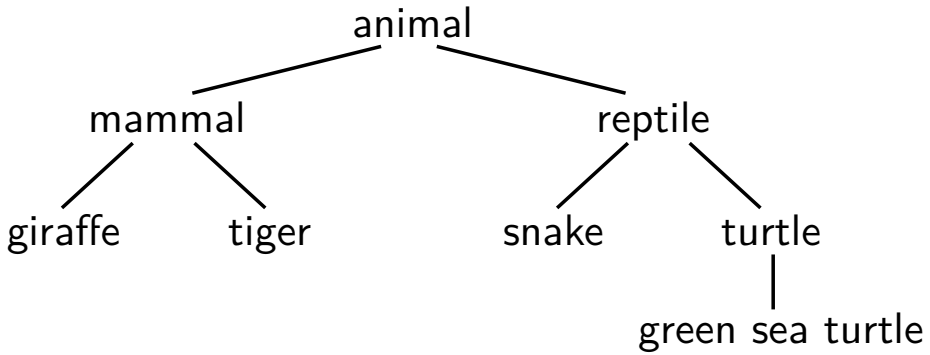
- heißt `recursive`, ist aber eigentlich Iteration
- traversiert hierarchische Daten *beliebiger* Tiefe (Joins erlauben nur eine konstante Anzahl an Schritten)

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

- Algorithmus:

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

WITH RECURSIVE ... UNION ALL




```
with recursive
 animals (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, 'green sea turtle', 7)),
  r as (select * from animals where name = 'turtle'
        union all
        select animals.*
        from r, animals
        where animals.id = r.parent)
select * from r;
```

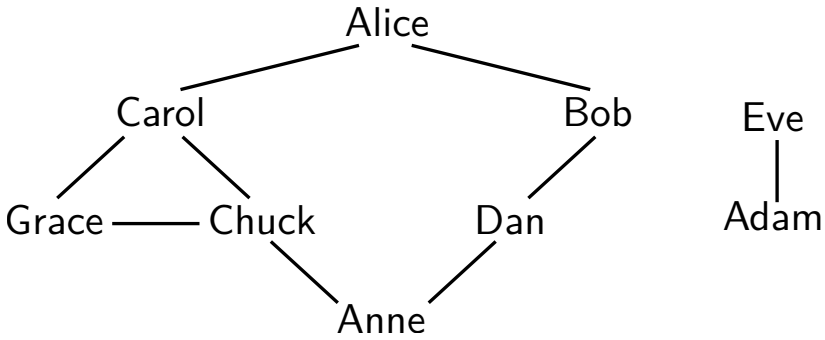
- Berechne alle Nachfahren von ‘reptile’
- Berechne 10! mittels Rekursion
- Berechne die ersten 20 Fibonacci-Zahlen ($F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2}$)

Rekursive Common Table Expressions mit UNION

- bei graphartigen Daten terminiert UNION ALL eventuell nicht
- **`with recursive`** *`[non-recursive]`* **`union`** *`[recursive]`*
- erlaubt das Traversieren zyklischer Strukturen
- Algorithmus:

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

WITH RECURSIVE ... UNION

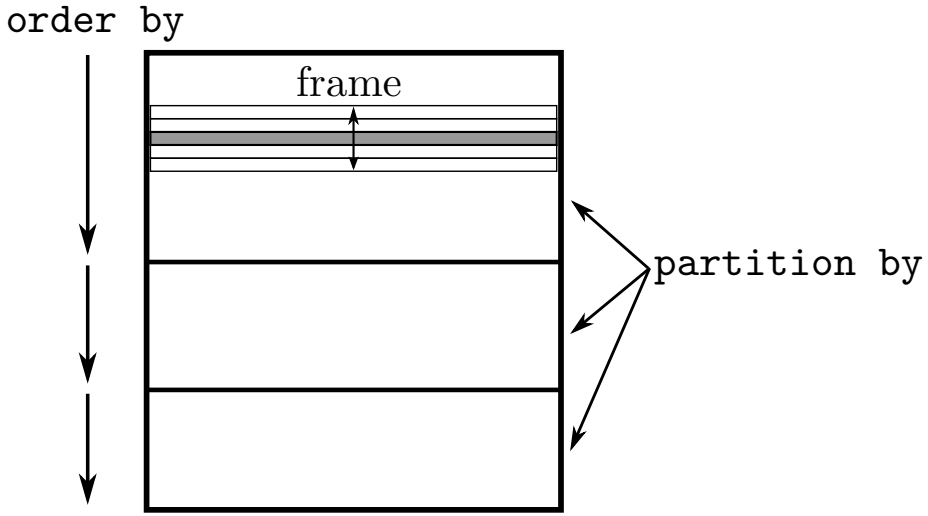



```
with recursive
 friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
   ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
   ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
 friendship (name, friend) as -- Freundschaft ist symmetrisch
    (select a, b from friends 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 Functions

- sehr vielseitiges Feature: Zeitreihenanalyse, Ranking, Top-k, Perzentile, gleitende Durchschnitte, kumulative Summen
- Window Functions werden nach den meisten anderen Klauseln (inklusive `group by`) ausgewertet, aber vor `order by`
- im Gegensatz zur Aggregation verändern Window Functions die Eingabe nicht, sie berechnen nur zusätzliche Spalten

```
select o_custkey, o_orderdate,
  sum(o_totalprice) over       -- Window Function
    (partition by o_custkey    -- Partitionierungsklausel
     order by o_orderdate      -- Sortierklausel
     range between unbounded preceding
      and current row)         -- Framing-Klausel
from customer;
```



Window Functions ohne Framing

- ranking:
  - `rank()`: Rang der aktuellen Zeile mit Lücken
  - `dense_rank()`: Rang der aktuellen Zeile ohne Lücken
  - `row_number()`: Zeilennummer der aktuellen Zeile
  - `ntile(n)`: gleichmäßig auf Buckets verteilen (liefert Ganzzahl von 1 bis `n`)
- Verteilung:
  - `percent_rank()`: relativer Rang der aktuellen Zeile ((Rang - 1) / (Anzahl Zeilen - 1))
  - `cume_dist()`: relativer Rang der Peer-Gruppe[1] ((Anzahl Zeilen vor oder peer zur aktuellen Zeile) / (Anzahl Zeilen))
- Navigation in der Partition:
  - `lag(expr, offset, default)`: `expr` auf der vorhergehenden Zeile der Partition auswerten
  - `lead(expr, offset, default)`: `expr` auf der nachfolgenden Zeile der Partition auswerten

[1] Zeilen mit gleichen Partitionierungs- und Sortierwerten heißen *Peers*.

- Bestimme Medaillen anhand der Anzahl der Bestellungen, Beispielausgabe:

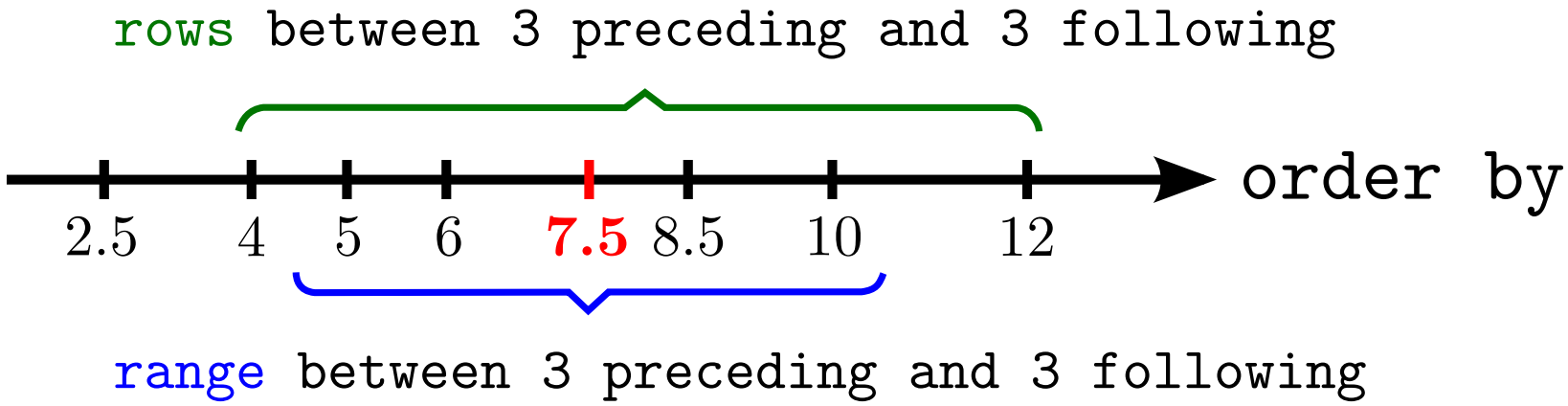
```
 custkey | count | medal
---------+-------+--------
    8761 |    36 | gold
   11998 |    36 | gold
    8362 |    35 | bronze
    4339 |    35 | bronze
     388 |    35 | bronze
    3151 |    35 | bronze
    9454 |    35 | bronze
```

- Berechne die jährliche (`extract(year from o_orderdate)`) prozentuale Umsatzänderung (`sum(o_totalprice)`), 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
```

Window Functions: Framing

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



- Standard-Frame (wenn eine `order by`-Klausel angegeben ist): `range between unbounded preceding and current row`
- Standard-Frame (wenn keine `order by`-Klausel angegeben ist): `range between unbounded preceding and unbounded following`
- komplexe Frame-Spezifikationen werden von PostgreSQL noch nicht unterstützt

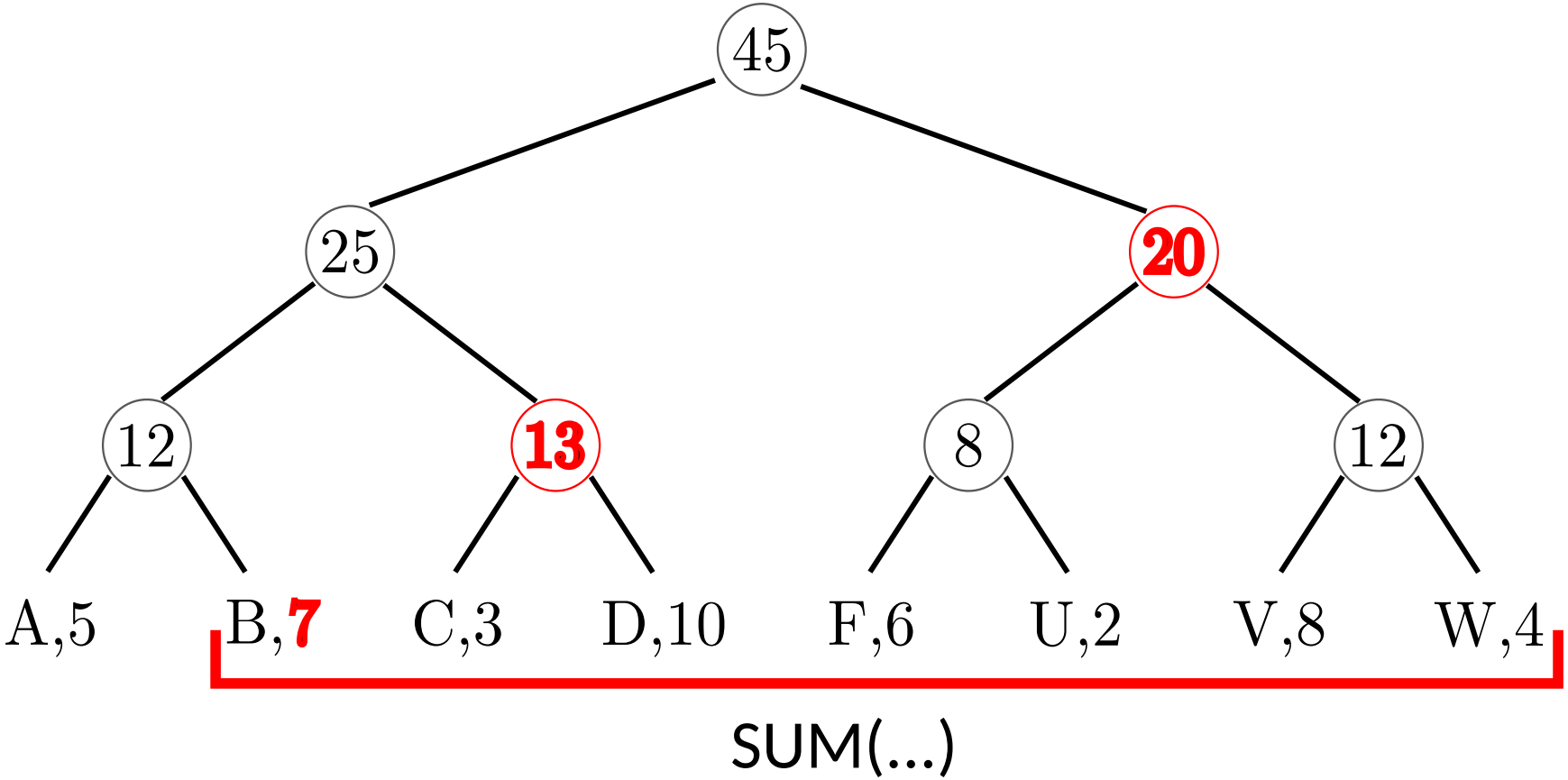
Window Functions mit Framing

- Aggregate (`min`, `max`, `sum`, ...):
  Aggregat über alle Tupel im aktuellen Frame berechnen
- Navigation im Frame:
  `first_value(expr)`, `last_value(expr)`, `nth_value(expr, nth)`: `expr` auf der ersten/letzten/`nth` Zeile des Frames auswerten

- Berechne die kumulierten Ausgaben der Kunden (`sum(o_totalprice)`) über die Zeit (`o_orderdate`)
- Berechne für jeden Kunden aus GERMANY die kumulierten Ausgaben (`sum(o_totalprice)`) pro 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
71 | 1995 |  1261012.10
```


45
25
20
12
13
8
12
A,5
B,7
C,3
D,10
F,6
U,2
V,8
W,4
SUM(...)

Statistische Aggregate

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

Ordered-Set-Aggregate

- Aggregatfunktionen, die Materialisierung/Sortierung erfordern und eine spezielle Syntax haben
- `mode()`: häufigster Wert
- `percentile_disc(p)`: diskretes Perzentil berechnen ($p \in [0, 1]$)
- `percentile_cont(p)`: stetiges Perzentil berechnen ($p \in [0, 1]$), interpoliert ggf., nur für numerische Datentypen

```
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;
```

- Aggregation über mehrere Dimensionen, z.B. Umsatz pro Jahr, pro Kunde, pro Lieferant
- mehrere Gruppierungen angeben:
  `group by grouping sets ((a, b), (a), ())`
- hierarchische Gruppierungen:
  `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)` ist äquivalent zu
  `group by grouping sets ((a, b), (a), (b), ())`

- Aggregiere den Umsatz (`sum(o_totalprice)`): insgesamt, pro Region (`r_name`), pro Nation (`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 |                      |
```