

SQL

SQL

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

- 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;

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;
```

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 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

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

```
select Semester  
from Studenten;
```

<u>Semester</u>
18
12
10
8
6
3
2
2

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

<u>Semester</u>
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 in der where-Klausel können logisch kombiniert werden mit: AND, OR, NOT
- Als Vergleichsoperatoren können verwendet werden: =, <, ≤, >, ≥, between, like

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;
```

Vergleiche mit mehreren Werten können mit `in` durchgeführt werden

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

- 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';
```

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

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

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

- 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 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), and **unbekannt**(u):

not		and	w	u	f	or	w	u	f
w	f	w	w	u	f	w	w	w	w
u	u	u	u	u	f	u	w	u	u
f	w	f	f	f	f	f	w	u	f

- Im Ergebnis einer SQL-Anfrage tauchen nur Tupel auf, für die die Auswertung der where-Klausel wahr ergibt

- 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;
```

- 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];
```

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

- In dieser Beispielanfrage muß spezifiziert werden woher MatrNr und VorlNr herkommen sollen
- Dazu schreibt man den Relationenname vor den Attributnamen

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

- 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;
```

- 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

Prof1		Prof2	
PersNr	Name	PersNr	Name
2125	Sokrates	2126	Russel
2126	Russel	2137	Kant

Anfrage: “Vereinige beide Listen”

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

PersNr	Name
2125	Sokrates
2126	Russel
2137	Kant

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

Prof1		Prof2	
PersNr	Name	PersNr	Name
2125	Sokrates	2126	Russel
2126	Russel	2137	Kant

Anfrage: “Welche Professoren sind auf beiden Listen”

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

PersNr	Name
2126	Russel

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

- 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)

```
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

- 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

- 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

- 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));
```

- =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);
```

- 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

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

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;
```

$$\frac{1}{8}$$

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;
```

$$\frac{1}{7}$$

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

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

- Funktioniert so nicht!!!

- Aggregatfunktionen reduzieren alle Werte einer Spalte zu einem **einzigem** 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

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

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

- 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;
```

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

→

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

- 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;
```

- 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;
```

- 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

- 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

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

- Finde alle Vorlesungen mit überdurchschnittlich vielen SWS:

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

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

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

- 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);
```

- 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);
```

- 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
```

- Verbinden

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

- Zerlegen

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

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

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

- Achtung: Fehler wenn Umwandlung nicht möglich ist

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 eine 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

```
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 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 aus anderen Relationen kopieren

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

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

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

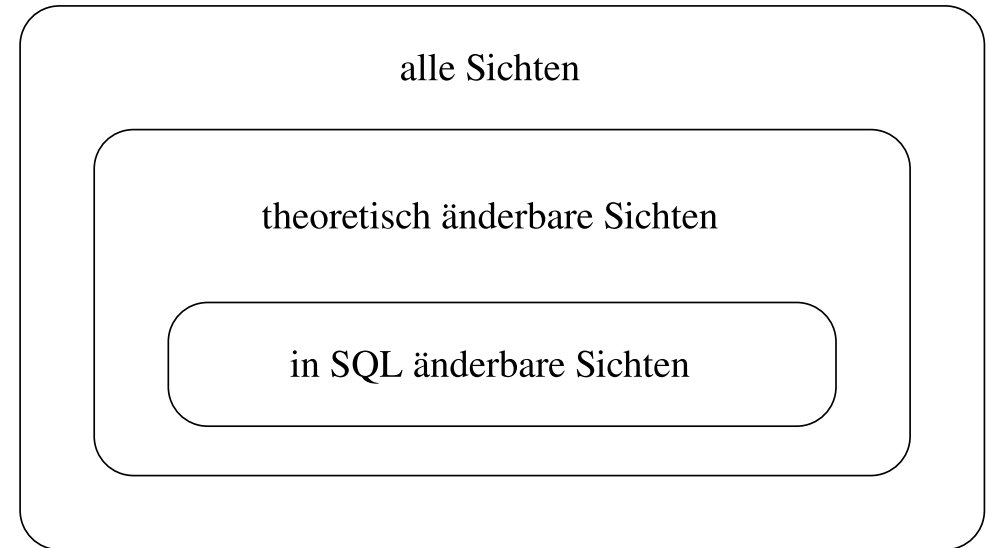
- Der **delete**-Befehl löscht Daten

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

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

```
delete from Professoren;
```

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



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

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

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

- *Integer*: smallint (2 bytes), integer (4 bytes), bigint (8 bytes)
- *Festkommazahlen*: numeric(scale,precision) (*scale* ist die Anzahl Dezimastellen, *precision* 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)
- allen Typen können NULL sein, außer wenn NOT NULL angegeben wird
- smallint/bigint/text/bytea sind Erweiterungen, aber oft unterstützt (z.B. in PostgreSQL)

- Für jede Relation kann ein Primärschlüssel definiert werden

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

- 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

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

- 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  
);
```

- 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)  
);
```

- 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)  
);
```

- 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

- 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)  
);
```

- 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

```
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}]  
);
```

- 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  
);
```

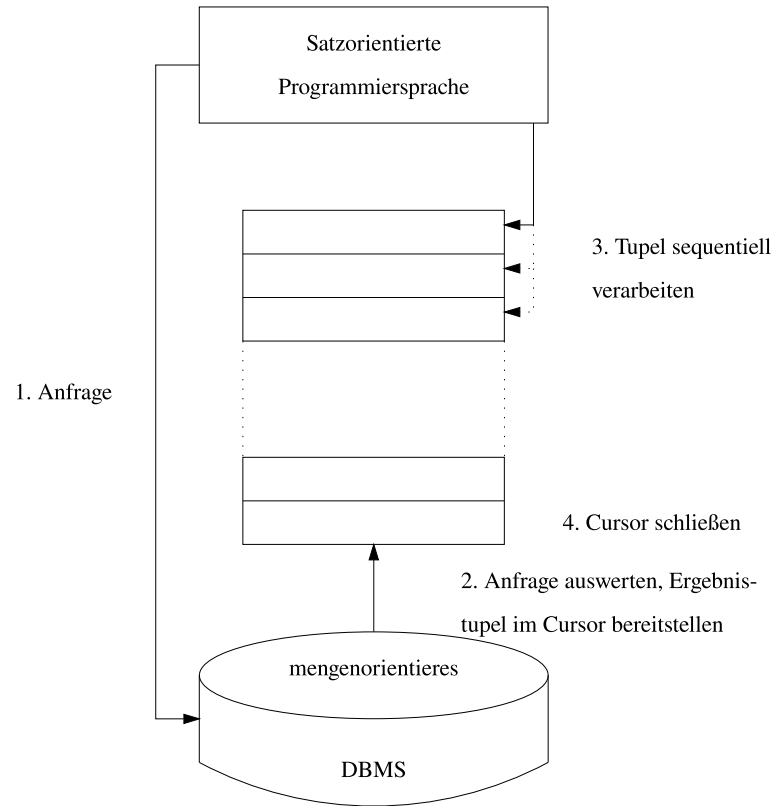
- Indexe beschleunigen den Zugriff auf Relationen (verlangsamen allerdings Änderungsoperationen)
- Die meisten DBMS legen automatisch einen Index auf dem Primärschlüssel an (um schnell die Eindeutigkeit prüfen zu können)
- Weitere Details zu Indexen gibt es später

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

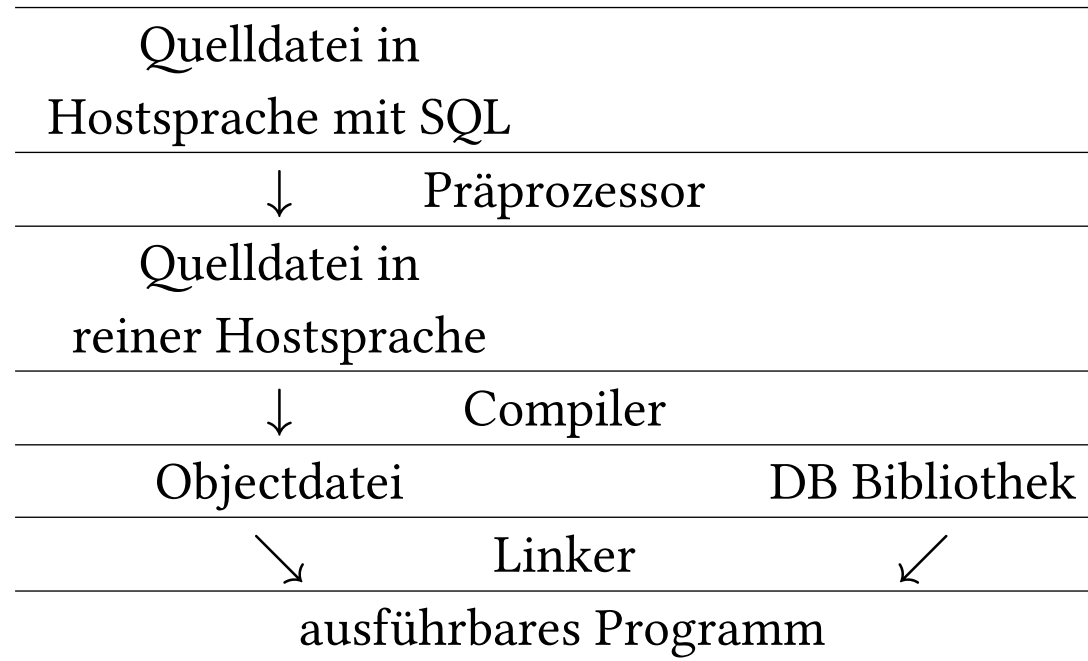
- Relationen, Sichten und Indexe können mit dem **drop**-Befehl wieder entfernt werden:
 - **drop table** *Relation*;
 - **drop view** *Sicht*;
 - **drop index** *Index*;

- 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

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



- Hier werden SQL-Befehl 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

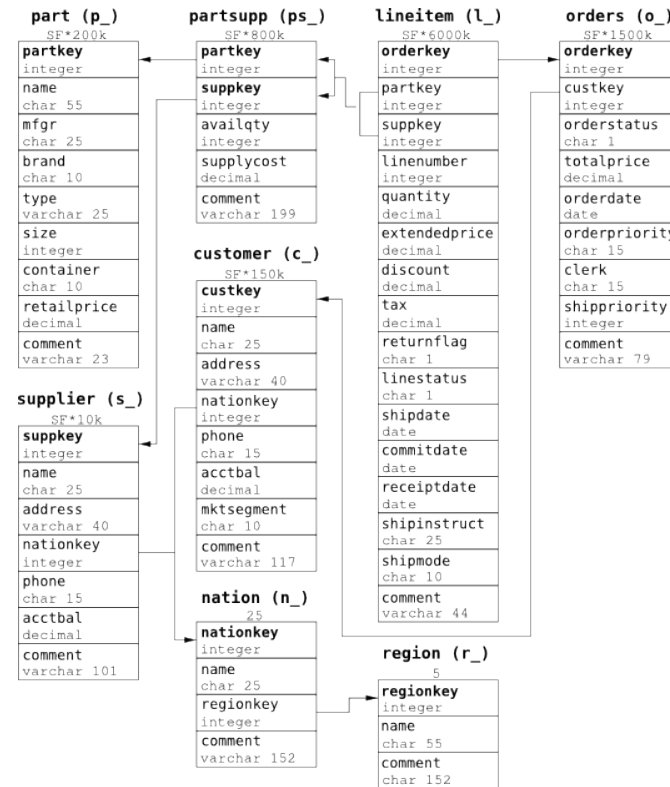


- Wird eingesetzt wenn die Anfragen zur Übersetzungszeit des Programms noch nicht bekannt sind
- Standardisierte Schnittstellen
 - ODBC (Open Database Connectivity)
 - JDBC (für Java)
- Flexibler, aber üblicherweise etwas langsamer als Embedded SQL

- 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 is an ad-hoc, decision support benchmark
- randomly generated data set, data set generator available
- size can be configured (*scale factor* 1 is around 1 GB)



```
create table lineitem (  
    l_orderkey integer not null,  
    l_partkey integer not null,  
    l_suppkey integer not null,  
    l_linenummer 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  
);
```

```
$ 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|0|
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, -- single line comment
        sum(l_extendedprice * (1 - l_discount)) revenue,
        o_orderdate, o_shippriority
from customer, orders, lineitem /* this is a
multi line comment */
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;
```

- How many items were shipped by German suppliers in 1995?
- What are the names and account balances of the 10 customers from EUROPE in the FURNITURE market segment who have the highest account balance?

- subqueries can appear (almost) everywhere:

```
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);
```

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

- subquery is correlated if it refers to tuple from outer query (l1.l_orderkey)
- naive execution: execute inner query for every tuple of outer query (quadratic runtime unless index on l_orderkey exists)

- queries can be rewritten to avoid correlation (some systems do this automatically):

```
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;
```

- decorrelate the following query

```
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, and INTERSECT remove duplicates

```
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;
```

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

```
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;
```

- case (conditional expressions)

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

- coalesce(a, b): replace NULL with some other value
- cast (explicit type conversion)
- generate_series(begin,end)
- random (random float from 0 to 1):

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

- concatenation:

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

- simple string matching:

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

- regexp string matching:

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

- extract substring:

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

- regexp-based replacement: (str, pattern, replacement, flags)

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

- sampling modes: `bernoulli` (pick random tuples) or `system` (pick random pages)
- set random seed with optional `repeatable` setting
- supported by PostgreSQL ≥ 9.5 :

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

- it is also possible to get arbitrary tuples:

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

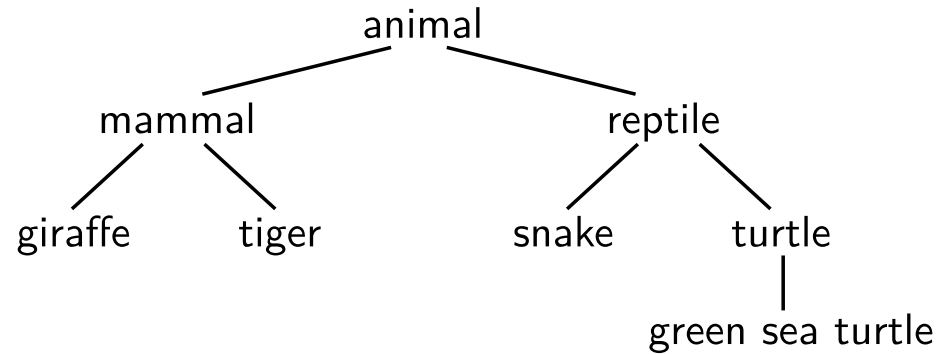
- compute the average `o_totalprice` using a sample of 1% of all orders

- called recursive but is actually iteration
- traverse hierarchical data of *arbitrary* depth (joins only allow a constant number of steps)

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

- algorithm:

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



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, 'grean 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;
```

- compute all descendants of 'reptile'
- compute $10!$ using recursion
- compute the first 20 Fibonacci numbers ($F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2}$)

- for graph-like data UNION ALL may not terminate
- **with recursive** [*non-recursive*] **union** [*recursive*]
- allows one to traverse cyclic structures
- algorithm:

workingTable = *unique*(*evaluateNonRecursive*())

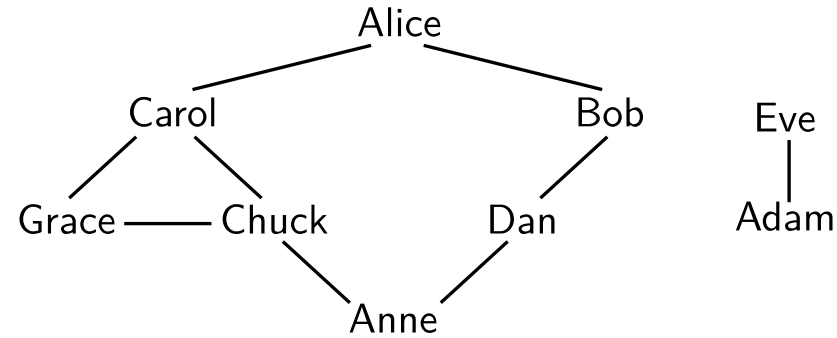
result = *workingTable*

while *workingTable* is not empty

workingTable = *unique*(*evaluateRecursive*(*workingTable*)) \ *result*

result = *result* $\cup$ *workingTable*

output *result*

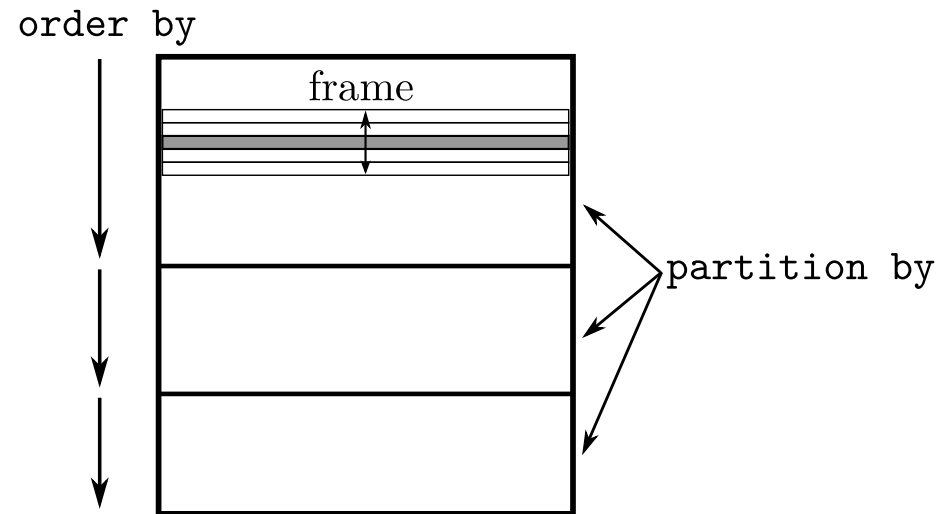


```

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;
  
```

- very versatile feature: time series analysis, ranking, top-k, percentiles, moving averages, cumulative sums
- window functions are evaluated after most other clauses (including `group by`) but before `order by`
- in contrast to aggregation, window functions do not change the input, they only compute additional columns

```
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;
```



- ranking:
 - `rank()`: rank of the current row with gaps
 - `dense_rank()`: rank of the current row without gaps
 - `row_number()`: row number of the current row
 - `ntile(n)`: distribute evenly over buckets (returns integer from 1 to n)
- distribution:
 - `percent_rank()`: relative rank of the current row $((\text{rank} - 1) / (\text{total rows} - 1))$
 - `cume_dist()`: relative rank of peer¹ group $((\text{number of rows preceding or peer with current row}) / (\text{total rows}))$
- navigation in partition:
 - `lag(expr, offset, default)`: evaluate expr on preceding row in partition
 - `lead(expr, offset, default)`: evaluate expr on following row in partition

¹Rows with equal partitioning and ordering values are *peers*.

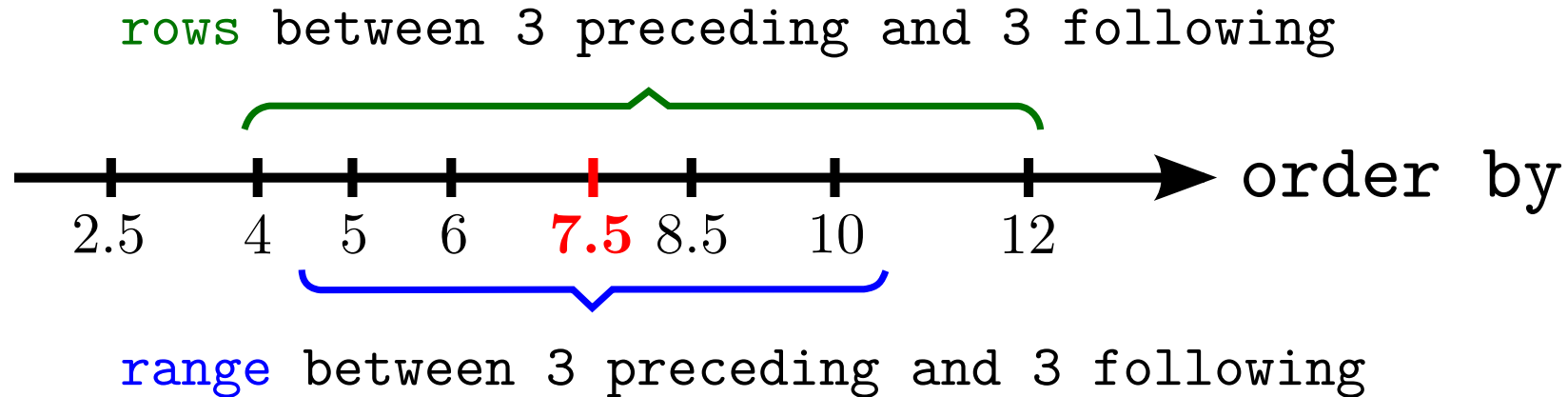
- determine medals based on number of orders, example output:

custkey	count	medal
8761	36	gold
11998	36	gold
8362	35	bronze
4339	35	bronze
388	35	bronze
3151	35	bronze
9454	35	bronze

- yearly (extract(year from o_orderdate) change in revenue percentage (sum(o_totalprice))), example output:

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

- `current row`: the current row (including all peers in range mode)
- `unbounded preceding`: first row in the partition
- `unbounded following`: last row in the partition

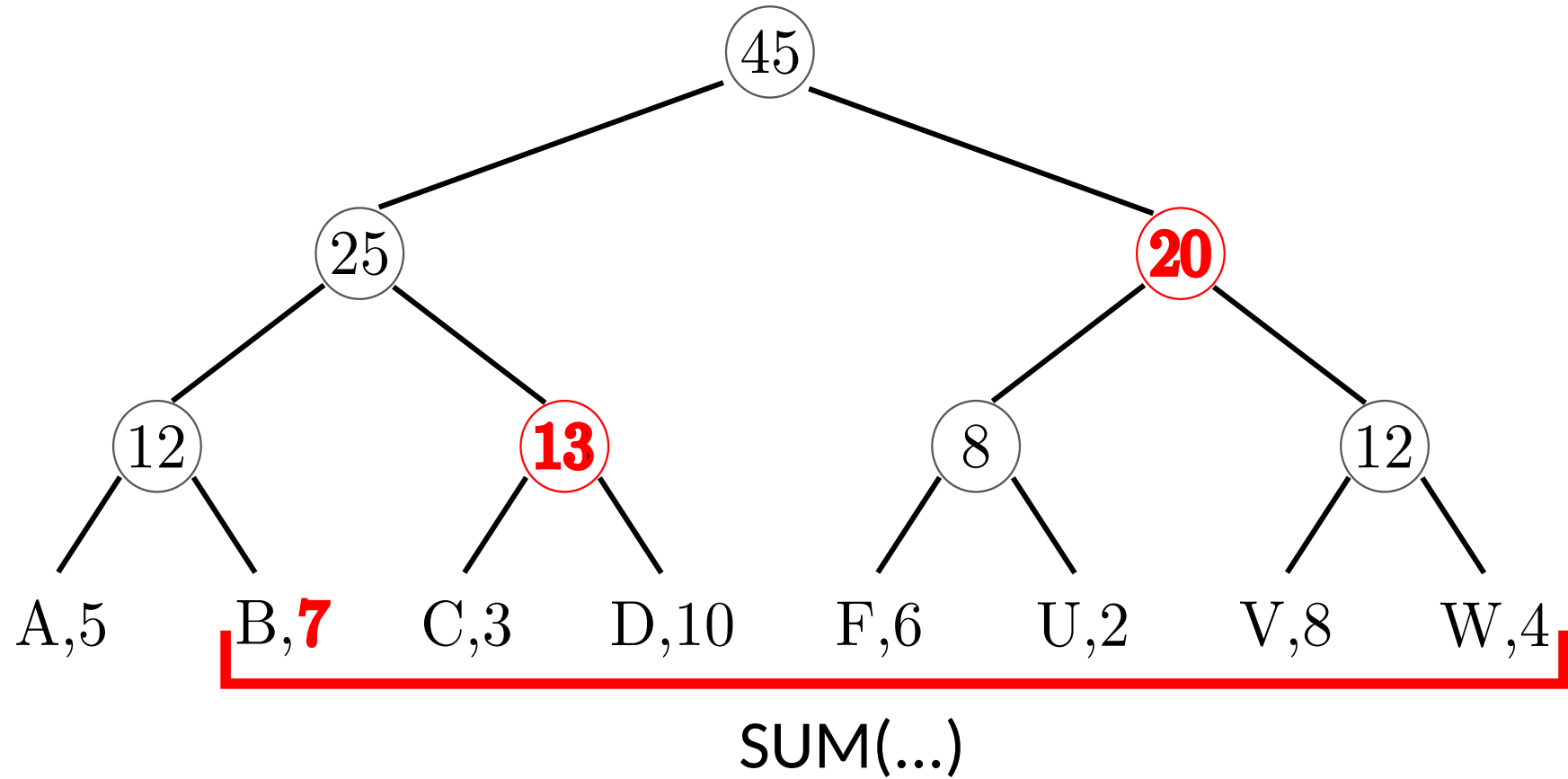


- default frame (when an `order by` clause was specified): range between unbounded preceding and current row
- default frame (when no `order by` clause was specified): range between unbounded preceding and unbounded following
- complex frame specifications are not yet supported by PostgreSQL

- aggregates (min, max, sum, ...): compute aggregate over all tuples in current frame
- navigation in frame:
 `first_value(expr)`, `last_value(expr)`, `nth_value(expr, nth)`: evaluate expr on first/
 last/nth row of the frame

- compute the cumulative customer spending (`sum(o_totalprice)`) over time (`o_orderdate`)
- for each customer from GERMANY compute the cumulative spending (`sum(o_totalprice)`) by year (`extract(year from o_orderdate)`), example output:

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



- `stddev_samp(expr)`: standard deviation
- `corr(x, y)`: correlation
- `regr_slope(y, x)`: linear regression slope
- `regr_intercept(y, x)`: linear regression intercept

- aggregate functions that require materialization/sorting and have special syntax
- `mode()`: most frequently occurring value
- `percentile_disc(p)`: compute discrete percentile ($p \in [0, 1]$)
- `percentile_cont(p)`: compute continuous percentile ($p \in [0, 1]$), may interpolate, only works on numeric data types

```
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;
```

- aggregate across multiple dimensions, e.g., revenue by year, by customer, by supplier
- specify multiple groupings:

group by grouping sets ((a, b), (a), ())

- hierarchical groupings:

group by rollup (a, b)

- both are equivalent to:

```
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;
```

- all (2^n) groupings:

group by cube (a, b) is equivalent to

group by grouping sets ((a, b), (a), (b), ())

- aggregate revenue (sum(o_totalprice)): total, by region (r_name), by name (n_name), example output:

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		