Kapitel 4

SQL

Generelle Anmerkungen

- SQL: Structured Query Language
- Früherer Name war SEQUEL
- Standardisierte Anfragesprache fuer 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

DRL

- 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

Student

MatrNr	Name	Geburtstag
1	Schmidt	1980-10-12
2	Müller	1982-07-30
3	Klein	1981-03-24
4	Meier	1982-07-30

Anfrage: "Gib mir die gesamte Information über alle Studenten"

```
select *
from   Student;
```

Ergebnis

MatrNr	Name	Geburtstag
1	Schmidt	1980-10-12
2	Müller	1982-07-30
3	Klein	1981-03-24
4	Meier	1982-07-30

Attribute selektieren

Anfrage: "Gib mir die Matrikelnr und den Namen aller Studenten"

```
select MatrNr, Name
from   Student;
```

MatrNr	Name
1	Schmidt
2	Müller
3	Klein
4	Meier

Duplikateliminierung

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

Beispiel

```
select Geburtstag
from   Student;
```

Geburtstag
1980-10-12
1982-07-30
1981-03-24
1982-07-30

```
select distinct Geburtstag
from   Student;
```

Geburtstag
1980-10-12
1982-07-30
1981-03-24

Where Klausel

Anfrage: "Gib mir alle Informationen über Studenten mit einer MatrNr kleiner als 3"

```
select *
from   Student
where  MatrNr < 3;
```

MatrNr	Name	Geburtstag
1	Schmidt	1980-10-12
2	Müller	1982-07-30

Prädikate

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

Beispiel für Between

Anfrage: "Gib mir die Namen aller Studenten, die zwischen 1982-01-01 und 1984-01-01 geboren wurden"

```
select Name
from   Student
where Geburtstag between date '1982-01-01' and date '1984-01-01';
```

ist äquivalent zu

```
select Name
from   Student
where Geburtstag >= date '1982-01-01'
and    Geburtstag <= date '1984-01-01';
```

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 Meier"

```
select *
from   Student
where  Name = 'Meier';
```

Suche mit Jokern (Wildcards)

Anfrage: "Gib mir alle Informationen über Studenten deren Namen mit einem M anfängt"

```
select *
from Student
where Name like 'M%';
```

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   Student
where  Geburtstag 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), 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

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   Vorlesung, Professor;
```

Joins

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

```
select *
from   Vorlesung, Professor
where  ProfPersNr = PersNr;
```

Joins(2)

- Es dürfen beliebig viele Relationnamen 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 R_1 [cross|inner|natural|left outer|right outer|full outer]
join R_2 [on $R_1.A = R_2.B$];

Joins(3)

- Weiteres Problem: Namenskollisionen (gleichnamige Attribute in verschiedenen Relationen) müssen aufgelöst werden
- Beispiel: Join von
  - `Student(MatrNr, Name, Geburtstag)`
  - `besucht(MatrNr, Nr)`
  - `Vorlesung(Nr, Titel, Credits)`

Qualifizierte Attributnamen

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

```
select *
from   Student, besucht, Vorlesung
where  Student.MatrNr = besucht.MatrNr
and    besucht.Nr = Vorlesung.Nr;
```

Kurzform

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

```
select *
from   Student S, besucht B, Vorlesung V
where  S.MatrNr = B.MatrNr
and    B.Nr = V.Nr;
```

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
1	Moerkotte
2	Kemper

Prof2

PersNr	Name
2	Kemper
3	Weikum

Anfrage: "Vereinige beide Listen"

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

PersNr	Name
1	Moerkotte
2	Kemper
3	Weikum

Duplikateliminierung

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

Schnitt

Prof1

PersNr	Name
1	Moerkotte
2	Kemper

Prof2

PersNr	Name
2	Kemper
3	Weikum

Anfrage: "Welche Professoren sind auf beiden Listen"

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

PersNr	Name
2	Kemper

Mengendifferenz

Prof1

PersNr	Name
1	Moerkotte
2	Kemper

Prof2

PersNr	Name
2	Kemper
3	Weikum

Anfrage: "Welche Professoren sind auf der ersten aber nicht auf der zweiten Liste?"

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

PersNr	Name
1	Moerkotte

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   Student
order by Geburtstag desc, Name;
```

MatrNr	Name	Geburtstag
4	Meier	1982-07-30
2	Müller	1982-07-30
3	Klein	1981-03-24
1	Schmidt	1980-10-12

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

Unkorrelierte Unteranfrage

Anfrage: "Gib mir die Namen aller Studenten, die die Vorlesung Nr 5 besuchen"

```
select S.Name
from   Student S
where  S.MatrNr in
       (select B.MatrNr
       from   besucht B
       where  B.Nr = 5);
```

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

Korrelierte Unteranfrage

Anfrage: "Finde alle Professoren für die Assistenten mit verschiedenen Fachgebieten arbeiten"

```
select  distinct P.Name
from    Professor P, Assistent A
where   A.Boss = P.PersNr
and     exists
        (select *
        from    Assistent 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

Student

MatrNr	Name	Geburtstag
1	Schmidt	1980-10-12
2	Müller	1982-07-30
3	Klein	1981-03-24
4	Meier	1982-07-30

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

1
4

Beispiel(2)

Student

MatrNr	Name	Geburtstag
1	Schmidt	1980-10-12
2	Müller	1982-07-30
3	Klein	1981-03-24
4	Meier	1982-07-30

```
select count(distinct Geburtstag)
from   Student;
```

1
3

Min/Max

Anfrage:"Gib mir den Studenten mir der größten MatrNr'"

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

- 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    Student
where   MatrNr =
        (select max(MatrNr)
        from Student);
```

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 Nr, count(*) as Anzahl
from   besucht
group by Nr;
```

Ergebnis

besucht

MatrNr	Nr
1	1
1	2
2	1
2	3
4	1
4	2
4	3

$\rightarrow$

Nr	Anzahl
1	3
2	2
3	2

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  PersNr, Titel, count(*) as Anzahl
from    Vorlesung
group by PersNr;
```

Having

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

Illustration

Anfrage:"Finde alle Professoren die mehr als drei Vorlesungen halten"

```
select  PersNr, count(Nr) as AnzVorl
from    Vorlesung
group by PersNr
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 der Durchschnitt an Credits wert sind 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 viel Credits:

```
create view ÜberSchnittCredit as
select  Nr, ProfPersNr
from    Vorlesung
where   Credits >
        (select avg (Credits)
        from    Vorlesung);
```

Komplizierte Anfrage(3)

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

```
create view VieleAssistenten as
select  Boss
from    Assistent
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    Professor
where PersNr in
        (select PersNr
        from    ÜberSchnittCredit)
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   Überschnitt as (select ...),
       VieleAssistenten as (select ...),
       ÜberSchnittCredit as (select ...),
select Name
from   Professor
where PersNr in
       (select PersNr
       from   ÜberSchnittCredit)
and PersNr in
       (select Boss
       from   VieleAssistenten);
```

Fallunterscheidungen mit CASE

- einfaches case

```
select  case x when 1 then 'rot'
               when 2 then 'grün'
               else 'anders' end
from    R
```

- allgemeines case

```
select  case when x>5 then 'groß'
             when y<3 then 'klein'
             else 'mittel' end
from    R
```

Arbeiten mit Text

- Verbinden
  select 'a' || 'b';
- Zerlegen
  select substring('abcfoo' from 3 for 2);

Typumwandling mit CAST

- explizite Typumwandlung mit `CAST(value AS TYPE)`
- Beispiel
  select cast(x as text) || cast(y as text)
  from R
- 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)`

Rekursion in SQL

Problem: Alle Voraussetzungen eine Vorlesung finden

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

Findet nicht die indirekten Voraussetzungen

Rekursion in SQL (2)

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 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

Rekursion in SQL (3)

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 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

- 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 Professor
values(123456, 'Kossmann', 012);
```

  - Weglassen von Attributwerten:

```
insert into Professor(PersNr, Name)
values(123456, 'Kossmann');
```

Daten einfügen(2)

- Daten aus anderen Relationen kopieren

```
insert into Professor(PersNr, Name)
select PersNr, Name
from Assistent
where PersNr = 111111;
```

Daten ändern

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

```
update Professor
set    ZimmerNr = 121
where  PersNr = 123456;
```

Daten löschen

- Der **delete**-Befehl löscht Daten

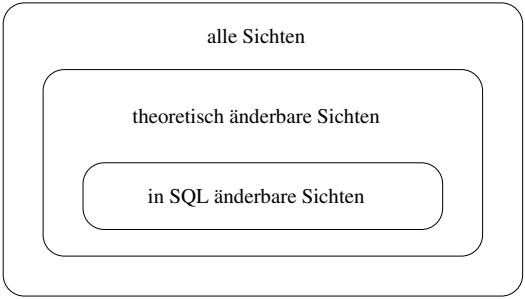
```
delete from Professor
where PersNr = 123456;
```

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

```
delete from Professor;
```

Änderbarkeit von Sichten

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



DDL

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

Relationen anlegen

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

```
create table Professor (
  PersNr      integer,
  Name        varchar(80),
  ZimmerNr    integer
);
```

Datentypen

- *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)

Schlüssel definieren

- Für jede Relation kann ein Primärschlüssel definiert werden

```
create table Professor (
  PersNr      integer,
  Name        varchar(80),
  ZimmerNr    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 Professor (
  PersNr     integer not null primary key,
  Name       varchar(80) not null,
  ZimmerNr   integer
);
```

Check-Klauseln

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

```
create table Professor (
  PersNr     integer not null primary key,
  Name       varchar(80) not null,
  ZimmerNr   integer
  check (ZimmerNr > 0 and ZimmerNr < 99999),
);
```

Check-Klauseln(2)

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

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

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 Professor (
  PersNr  integer primary key,
  ...
);
create table Vorlesung (
  Nr            integer primary key,
  ...
  ProfPerNr     integer not null,
  foreign key (ProfPersNr)
  references Professor(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 Vorlesung (
  Nr          integer primary key,
  ...
  ProfPersNr  integer not null,
  foreign key (ProfPersNr) references Professor(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 Assistent (
  PersNr      integer not null primary key,
  Name        varchar(80) not null,
  Fachgebiet  varchar(200) default 'Informatik'
);
```

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*;

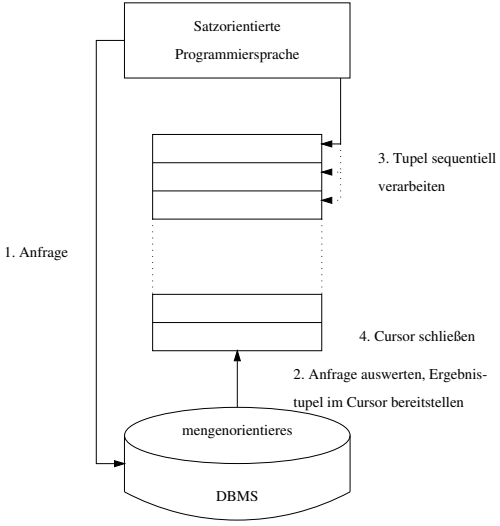
DCL

- 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

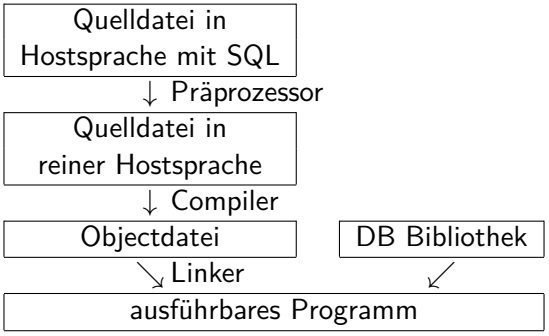
Anfragen in Anwendungen



Embedded SQL

- Hier werden SQL-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

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

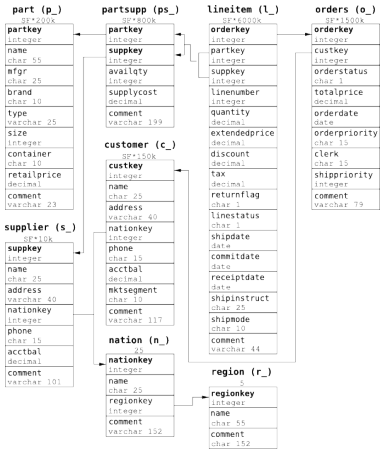
- SQL ist *die* Standardsprache im Umgang mit relationalen Systemen
- SQL enthält Befehle zum Abrufen, Ändern, Einfügen und Löschen von Daten
- Es existieren weitere Befehle, um ein Schema zu definieren, den Zugriff zu kontrollieren und Transaktionen zu steuern

Fortgeschrittene Konzepte in SQL

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

TPC-H Data Set

- 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)



Schema Definition

```
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
);
```

Loading CSV-like Data

```
$ 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|199
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

Basic SQL: Joins, Group By, Ordering

```
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?

- 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

- 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);
```

Correlated Subqueries

```
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 form outer query (`l1.l_orderkey`)
- naive execution: execute inner query for every tuple of outer query (quadratic runtime unless index on `l_orderkey` exists)

Query Decorrelation

- 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);
```

Set Operators

- 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;
```

“Set” Operators

- 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;
```

Miscellaneous Useful Constructs

- 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
```

Working With Strings

- 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

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

Recursive Common Table Expressions

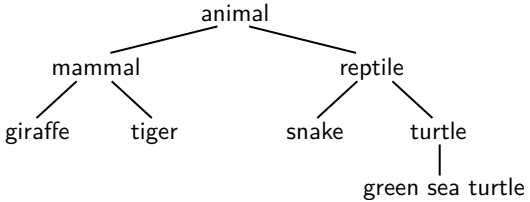
- called `recursive` but is actually iteration
- traverse hierarchical data of *arbitrary* depth (joins only allow a constant number of steps)

```
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 ... 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, '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;
```

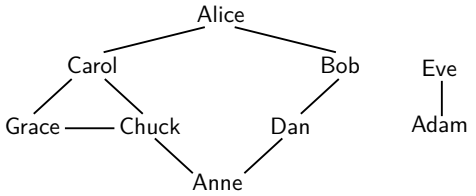
- compute all descendants of 'reptile'
- compute 10! using recursion
- compute the first 20 Fibonacci numbers ($F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$)

Recursive Common Table Expressions with UNION

- 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
    $workingTable = unique(evaluateRecursive(workingTable)) \setminus result$
    $result = result \cup workingTable$
  output $result$

WITH RECURSIVE ... UNION



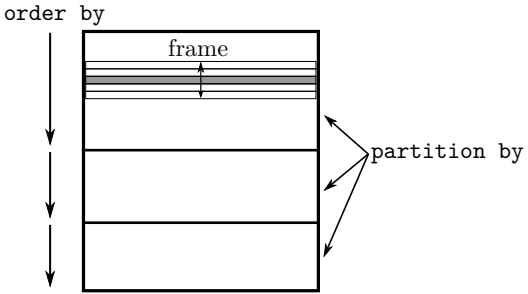

```
with recursive
 friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
   ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
   ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
 friendship (name, friend) as -- friendship is symmetric
    (select a, b from friends union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

- 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

Window Functions: Concepts

```
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 Functions Framing That Ignore Framing

- 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 ((rank - 1) / (total rows - 1))
  - `cume_dist()`: relative rank of peer[1] group ((number of rows preceding or peer with current row) / (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

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

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

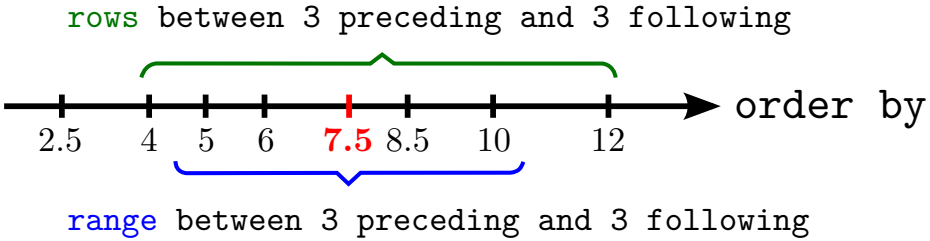
```
 custkey | count | medal
---------+-------+--------
    8761 |    36 | gold
   11998 |    36 | gold
    8362 |    35 | bronze
    4339 |    35 | bronze
     388 |    35 | bronze
    3151 |    35 | bronze
    9454 |    35 | bronze
```

- 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
```

Window Functions: Framing

- `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

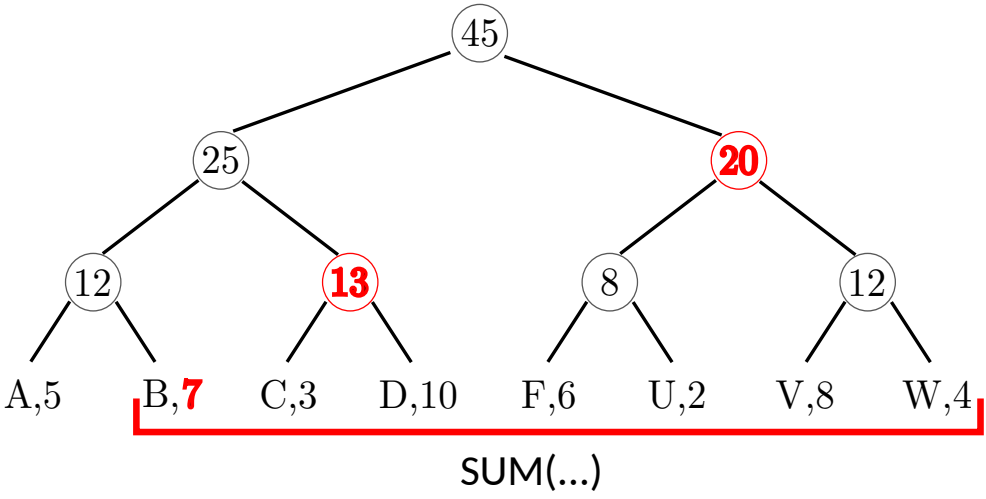
Window Functions With Framing

- 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
```

Efficient Evaluation Using Segment Tree



Statistical Aggregates

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

Ordered-Set Aggregates

- 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;
```

Grouping Sets, Rollup, Cube

- 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 |                        |
```