FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Datenbanksysteme I

Prof. Dr. Viktor Leis

Professur für Datenbanken und Informationssysteme

SQL

Generelle Anmerkungen

- SQL: Structured Query Language
- Ursprünglicher Name war SEQUEL
- Standardisierte Anfragesprache für relationale DBMS: SQL-89, SQL-92, SQL-99, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016, SQL:2019
- Trotz Standardisierung gibt es leider signifikante Abweichungen zwischen verschiedenen Herstellern
- 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

SQL Web-Interface

https://hyper-db.com/interface.html

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 1982-01-01 and 1984-01-01;
```

ist äquivalent zu

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

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    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
  - `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

Limit/Top-K Anfragen

- manchmal will man die Anzahl der Ergebnisse beschränken
- Syntax je nach DBMS unterschiedlich
- PostgreSQL:

```
select  *
from    Student
order by Geburtstag desc
limit   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

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

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;
```

count
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;
```

count
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;
```

Kommentare

```
select Name       -- einzeiliger Kommentar
from Student      /* mehr-
zeiliger Kemmentar */
where Geburtstag > '1982-01-01';
```

Klauseln: Reihenfolge

Syntax:

1. select
2. distinct
3. from
4. where
5. group by
6. having
7. order by
8. limit

Ausführungsreihenfolge:

1. from
2. where
3. group by
4. having
5. select
6. distinct
7. order by
8. limit

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);
```

Temporäre Sichten mit WITH

```
with    ÜberSchnittCredit as (
        select  Nr, ProfPersNr PersNr
        from    Vorlesung
        where   Credits >
                (select avg (Credits)
                from    Vorlesung)),
        VieleAssistenten as (
        select  Boss
        from    Assistent
        group by Boss
        having count(*) > 3)
select  Name
from    Professor
where PersNr in (select PersNr from ÜberSchnittCredit)
and PersNr in (select Boss from VieleAssistenten);
```

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 einfügen(3)

- Daten können auch direkt aus einer Datei einfügt werden
- Syntax nicht standardisiert, hier PostgreSQL
- `\copy Professor from 'prof.csv'`
- effizienter als einzelne INSERT Anfragen, viele Optionen
- `copy` erlaubt auch das Exportieren einer Relation in eine Datei

Daten ändern

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

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

Daten löschen

- Der **delete**-Befehl löscht Daten

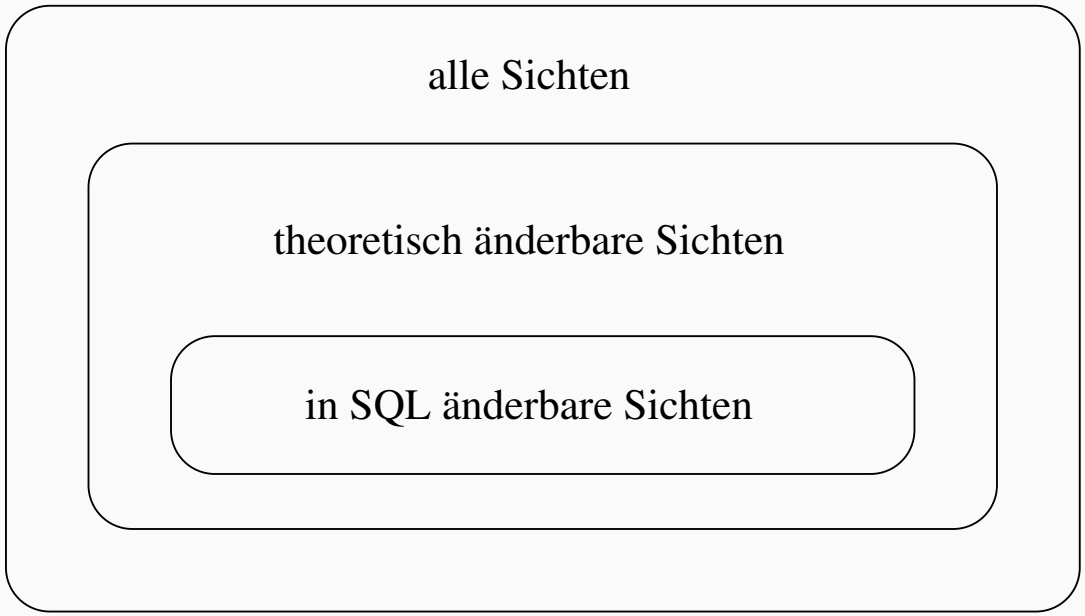
```
delete from Professor
where  PersNr = 123456;
```

- 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 Duplikateliminerung
- 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 in PostgreSQL: Zahlen

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

Datentypen in PostgreSQL: Nicht-Numerische Typen

- *Zeichenketten*: `varchar(n)` (Maximallänge *n*), `char(n)` (Maximallänge *n*, mit Leerzeichen aufgefüllt, merkwürdige Semantik), `text` (beliebige Größe)
- *Wahrheitswerte* `boolean` (1 byte)
- *Zeit*: `timestamp` (8 bytes), `date` (4 bytes), `interval` (16 bytes)

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*;

Stored Procedures

- “Stored Procedures” erlauben es Programmlogik im Datenbanksystem auszuführen
- verbindet SQL mit “normalen” Programmierkonstrukten wie Schleifen, IF-Statements, etc.
- oft deutlich effizienter als Implementierung in einer normalen Programmiersprache
- unterschiedliche Systeme verwenden leider unterschiedliche Sprachen: PL/pgSQL (PostgreSQL), PL/SQL (Oracle), Transact-SQL (SQL Server)

Stored Procedures: PL/pgSQL

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- first try to update the key
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- not there, so try to insert the key
        -- if someone else inserts the same key concurrently,
        -- we could get a unique-key failure
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- Do nothing, and loop to try the UPDATE again.
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

Trigger

- Stored Procedures können auch automatisch ausgeführt werden wenn sich die Datenbasis ändert
- z.B. bei jedem INSERT, UPDATE oder DELETE auf einer bestimmten Tabelle wird eine bestimmte Stored Procedure ausgeführt
- eine typische Anwendung ist das Auditing (Protokollierung aller Änderungen)

Trigger (PostgreSQL)

```
CREATE TABLE emp (empname text, salary integer, last_date timestamp, last_user text);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when they must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

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

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