WWU
MÜNSTER

```
select <Attribut- und Funktionsliste>
from <Relationenliste>
[where <Bedingung>]
[group by <Gruppierungsattribut(e)>]
[having <Gruppenbedingung>]
[order by <Attributliste>];
```

Structured Query Language (SQL)

Datenbanken

wissen.leben

Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Relationale Entwurfstheorie**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Verteilte Datenbanken**
   - Fragmentierung, Replikation, Allokation; CAP
   - Anfragebeantwortung, föderierte Systeme

Relationales Schema

- Relationales Schema (in 3NF/BCNF): Unternehmen → DB anlegen

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

ArbeitetAn	ProjNr	SVN	Std

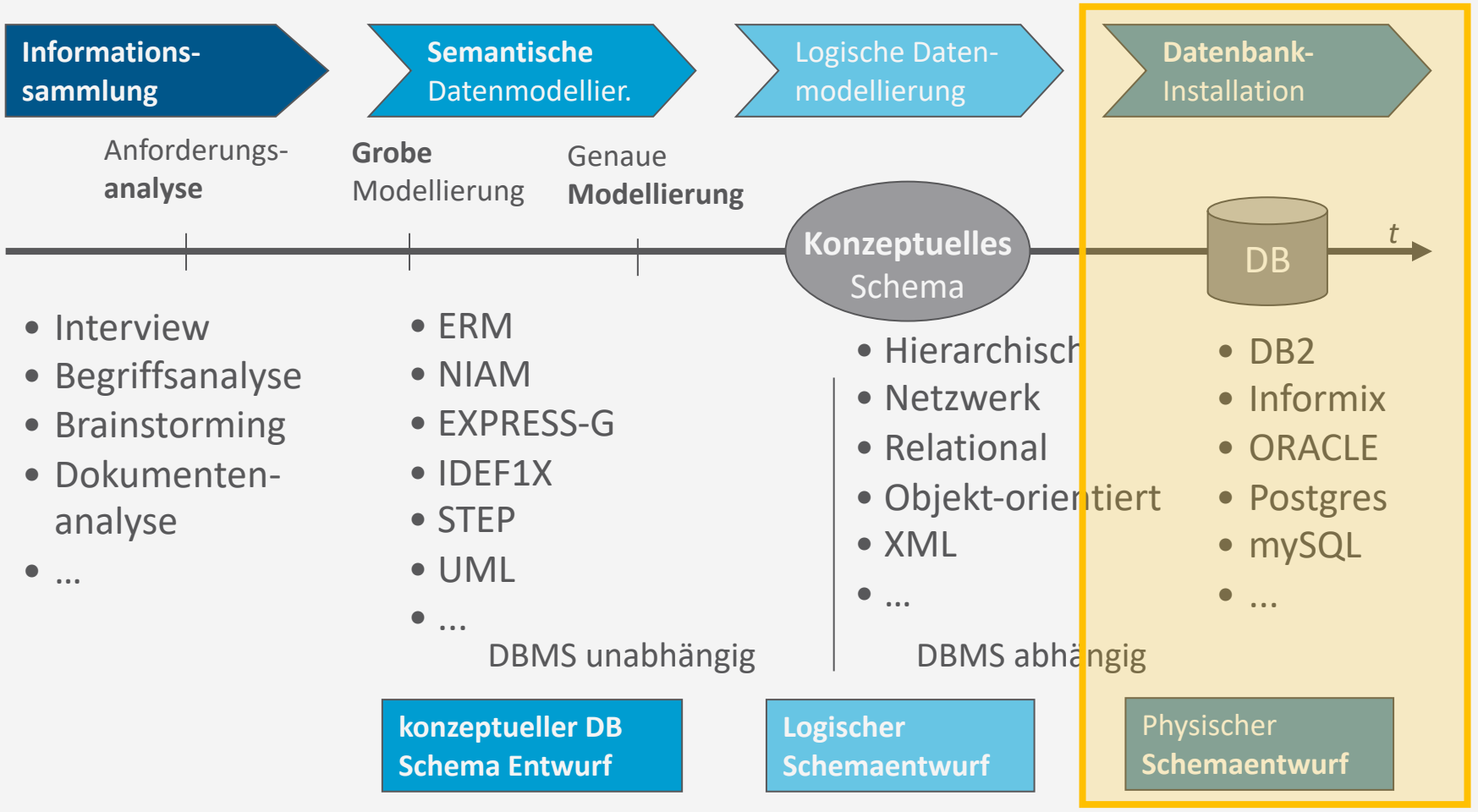
AbtStandort	AbtNr	Standort

Angehörige	Name	Geb	Geschlecht	Grad	SVN

- Relationale Algebra: Anfragen und Datenmanipulation
  - $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$; gruppieren, aggregieren
  - Insert, delete, update

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



SQL - Historie

- 1974: SEQUEL
  - Erster Vorschlag für die Sprache SQL, Entwicklung durch IBM
  - Implementierung für (experimentelles) relationales DBMS: System-R
- 1983: SQL ist de facto Standard
- 1986: SQL-86 / SQL 1
  - Erster offizieller Standard durch ANSI und ISO
- 1989: SQL-89
  - Revision des ersten Standards
- 1992: SQL-92 / SQL 2
  - zweite, deutlich erweiterte Revision
- 2000: SQL 3
  - mit OO-Konzepten, Multimedia, …
- Weitere Revisionen 2003, 2006 (mit XML), 2008, 2011, 2016 (aktuell)

DB-Sprachen

- Definition von DBs:
  - View Definition Languages (VDLs): extern
  - Data Definition Languages (DDLs): logisch
  - Storage Definition Languages (SDLs): intern

- Zugriff auf DBs (Einfügen, Ändern, Löschen und Anfragen von Datensätzen):
  - Data Manipulation Languages (DMLs)
    - Einfüge-, Änderungs- und Löschoperationen: Updates
    - Reine Anfragen: „Queries“
    - Alle Zugriffsarten: „Manipulation“
  - *Data Control Languages* (DCLs)

SQL : Structured Query Language
VDL, DDL, SDL, DML, DCL in einem

Überblick: 5. Structured Query Language (SQL)

A. ***Datendefinition (SQL als DDL)***
   - Schema, Tabellen, Datentypen, Constraints definieren
   - Strukturelle Änderungen mittels drop, alter

B. *Datenmanipulation (SQL als DML)*
   - Anfragen
   - Datenänderungen

C. *Und der Rest*
   - Sichten (SQL als VDL)
   - Rechtevergabe (SQL als DCL)
   - Programmiermethoden

Das SCHEMA Konstrukt

- SCHEMA: Namensraum in einer Datenbank
  - Die meisten Implementierungen akzeptieren auch DATABASE
- Enthält:
  - Eindeutigen Namen
  - Autorisierungsbezeichner, um Nutzer*innen oder Inhaber*innen des Schemas zu identifizieren
  - Deskriptoren für jedes im Schema enthaltene Element:
    - Relationen
    - Wertebereiche
    - Einschränkungen
    - Sichten
    - Autorisierungsinformationen bzw. Zugriffsrechte
    - etc.

Definition eines SCHEMAs

- CREATE SCHEMA definiert eine Hülle für eine DB:
  - Name: `SchemaName`
  - Autorisierung: `[ authorization ] Authorization`
    - Identifiziert Nutzer*in, dem*r das Schema gehört; wenn nicht genannt, wird der*ie derzeitige Nutzer*in als Inhaber*in gesetzt
  - Mögliche Liste von Elementen in Schema: `SchemaElementDefinition`
    - Liste von Tabellen innerhalb des Schemas
      - Über CREATE TABLE (nächste Folie)

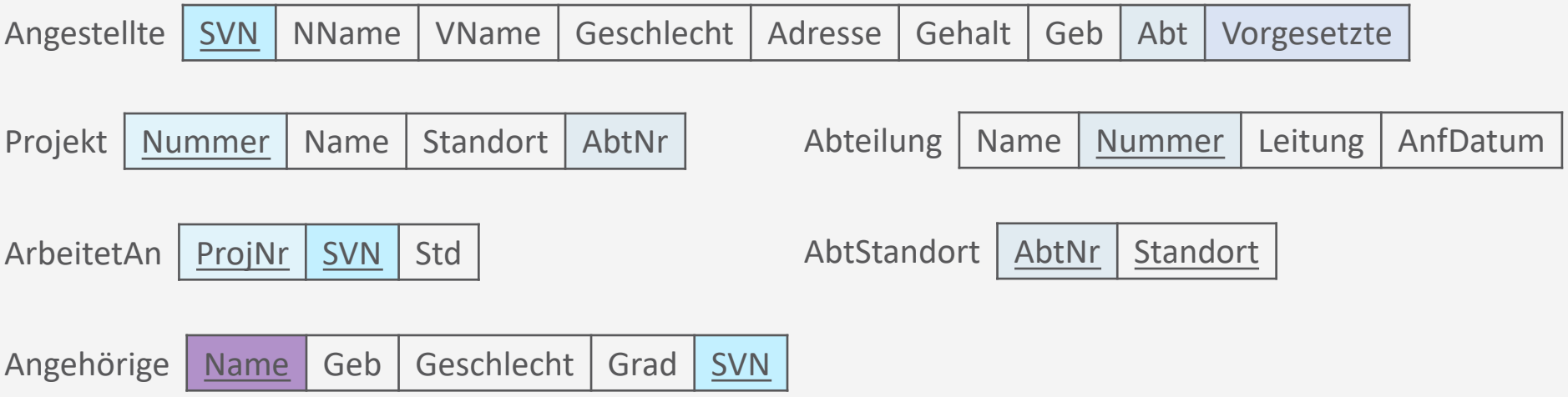
```
create schema [ SchemaName ]
[ [ authorization ] Authorization ]
{ SchemaElementDefinition };
```

Eingeschränkte SQL-Grammatik
(nicht vollständig)

- Beispiele:
  - `create schema Unternehmen authorization JSmith;`
  - `create schema Unternehmen authorization JSmith create table Projekt;`

Relationales Schema: CREATE Beispiele

- Relationales Schema (in 3NF/BCNF): Unternehmen → DB anlegen



Praxis-Test mit DBeaver

- Tabellen definieren

Dbeaver Screenshot

Weitere Online-Ressource:
https://www.w3schools.com/sql/default.asp
Achtung: SQL Befehle teilweise implementierungsabhängig

Beispiele für CREATE TABLE

Einrückung, erweiterter Leeraum dient nur der Übersicht und ist nicht erforderlich.

```
create table Angestellte (
     SVN            char(12)      not null,
     NName          varchar(25)   not null,
     VName          varchar(25)   not null,
     Geschlecht     char,
     Adresse        varchar(60),
     Gehalt         decimal(10,2),
     Geb            date,
     Abt            int           default 1,
     Vorgesetzte    char(12),

     primary key  (SVN),
     foreign key  (Vorgesetzte) references Angestellte(SVN)
);
```

`Abt` ist Fremdschlüssel auf Relation ABTEILUNG, die noch nicht definiert ist → noch nicht definierbar

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Beispiele für CREATE TABLE

```
create table Abteilung (
    Name          varchar(25) not null,
    Nummer        serial,
    Leitung       char(12),
    AnfDatum      date,

    primary key (Nummer),
    unique (Name),
    foreign key (Leitung) references Angestellte(SVN)
);
```

Für eine inkrementell wachsende ID nutzt
- *PostgreSQL* den Datentyp SERIAL, womit ein Int-Attribut angelegt wird, welches mit jeder Einfügung um 1 inkrement wird (Start bei 1)
- *MySQL* den Datentyp INT zusammen mit AUTO_INCREMENT
- *SQL Server* den Datentyp INT zusammen mit IDENTITY(m,n), wobei m=1 der Startwert und n=1 das Inkrement

Abteilung	Name	Nummer	Leitung	AnfDatum

Beispiele für CREATE TABLE

```
create table Angestellte (
    ...
    primary key (SVN),
    foreign key (Vorgesetzte) references Angestellte(SVN)
);
create table Abteilung (
    ...
    primary key (Nummer),
    ...
);
alter table Angestellte
    add foreign key (Abt) references Abteilung(Nummer)
```

Beispiele für CREATE TABLE

```
create table Projekt (
     Name          varchar(25) not null     unique,
     Nummer        int         not null     primary key,
     Standort      varchar(25),
     AbtNr         int         not null     references
                                            Abteilung(Nummer)
);
```

Projekt	Nummer	Name	Standort	AbtNr

Beispiele für CREATE TABLE

```
create table Abteilung (
    ...
    primary key (Nummer),
    ...
);
create table AbtStandort(
    AbtNr       int          not null,
    Standort    varchar(25)  not null,

    primary key (AbtNr, Standort),
    foreign key (AbtNr) references Abteilung(Nummer)
);
```

AbtStandort | AbtNr | Standort

Beispiele für CREATE TABLE

```
create table ArbeitetAn (
    ProjNr   int       not null  references Projekt(Nummer),
    SVN      char(12)  not null  references Angestellte(SVN),
    Std      decimal(3,1),

    primary key (SVN, ProjNr)
);
```

ArbeitetAn | ProjNr | SVN | Std

Beispiele für CREATE TABLE

```
create table Angehoerige (
    Name          varchar(25) not null,
    Geschlecht    char,
    Geb           date,
    Grad          varchar(8),
    SVN           char(12)    not null,

    primary key (SVN, Name),
    foreign key (SVN) references Angestellte(SVN)
);
```

Häufig gilt: keine Umlaute

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Definition von Relationen: CREATE TABLE

- CREATE TABLE spezifiziert eine neue Relation
  - Name der Relation: `TableName`
  - Liste von Attributen: `AttributeList`
    - Name der Attribute: `AttrName`
    - Name der Wertebereiche: `Domain`
  - Intrarelationale Einschränkungen
    - NOT NULL, UNIQUE, PRIMARY KEY
    - DEFAULT-Werte zusätzlich angebbar
    - SERIAL für automatisches inkrementieren eines INT (Start bei 1)
  - Interrelationale Einschränkungen
    - REFERENCES, FOREIGN KEY, CHECK (komplexe Bedingung über mehrere Relationen)

```
create table [ TableName ] [(
   [AttributeList]
   [RConstrList]
)];
```

```
AttrName   Domain   [AConstrList]
```

AConstrList:

```
  not null
| default Value
| unique
| primary key AttrName
| references TableName1(AttrName1)
```

RConstrList:

```
  primary key (AttrNameList)
| unique (AttrNameList)
| foreign key (AttrName)
      references TableName1(AttrName1)
| check [Bedingung]
```

Definition von Relationen: CREATE TABLE

```
create table [ TableName ] [(
   [AttributeList]
   [RConstrList]
)];
```

- Jede Relation ist einem Schema implizit (Defaultschema) oder explizit (Punktnotation) zugeordnet
  - Verwendung (und Erweiterung) eines Schemas über Punktnotation:
    - `create table Unternehmen.Angestellte`

Vordefinierte Datentypen in SQL

Achtung: hier große Unterschiede zwischen DBMS

- Numerische Typen:
  - Ganze Zahlen:
    INTEGER / INT, SMALLINT
  - Reelle Zahlen:
    FLOAT, REAL, DOUBLE PRECISION
    - Approximativ
      - Achtung bei Test auf Gleichheit
  - Formatierte Zahlen:
    DECIMAL(i,j), NUMERIC(i,j)
    - Exakt
    - i und j: Dezimalstellen und Nachkommastellen
  - SERIAL als INT mit Auto-Inkrement
- Text-Typen
  - Zeichenketten mit fester Länge:
    CHAR(n)
    - Bei Input mit weniger als n Zeichen wird aufgefüllt
    - Max. n = 255
  - Zeichenketten mit variabler Länge:
    VARCHAR(n)
    - n maximale Länge; kein Auffüllen
    - Max. n systemabhängig
  - In manchen DBMS in der Zwischenzeit synonym

Vordefinierte Datentypen in SQL

Achtung: hier große Unterschiede zwischen DBMS

- Datum, Zeit, Zeitstempel, Intervall
  - DATE ('YYYY-MM-TT')
  - TIME ('hh:mm:ss[.nnnnnnn]')
  - DATETIMEOFFSET ('YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm‘)
  - INTERVAL
    - YEAR, DAY, SECOND...
  - Beispiele
    - `select to_number(to_char(current_date,'J'),'9999999');`
    - `select to_char(to_date('1','J'),'DD-MM-YYYY');`
    - `select date '2020-01-01' + interval '1' day;`

Julianisches Datum für Daten vor 1841, Anzahl an Tagen seit 1. Januar 4712 v.Chr.

dual: In manchen Implementierungen gibt es eine so genannte Default Tabelle **dual** (1 Zeile, 1 Reihe) für Anfragen, die eigentlich keine Tabelle benötigten, wenn das DBMS das FROM Konstrukt zwingend vorsieht

Benutzerdefinierte Datentypen

- Benannte Definitionen (Domains)
  - Können im Schema wie Basisdatentypen verwendet werden
- Warum verwenden?
  - Bessere Lesbarkeit
  - Bessere Wartbarkeit

```
create domain [ DomName ]
    as Domain
   [default Value];
```

- Beispiel:
  - `create domain svn_dom as char(12)`
  - Zusätzlich Default-Wert möglich:
    - `create domain abt_name as char(25) default 'invalid';`

Constraints: Intrarelationale Constraints

- NOT NULL
  - Verbietet die Speicherung von NULL-Werten
- UNIQUE
  - Verlangt Eindeutigkeit von Attributen (NULL max. einmal)
- PRIMARY KEY
  - Verlangt Eindeutigkeit von Attributen (NULL nicht erlaubt)
- CHECK
  - Ermöglicht die Formulierung komplexer Einschränkungen

AConstrList

```
  not null
| default Value
| unique
| primary key AttrName
```

```
  primary key (AttrNameList)
| unique (AttrNameList)
| check [Bedingung]
```

RConstrList

- *Einige SQL Implementierungen:* AUTO_INCREMENT
  - Eindeutige ID beginnend bei 1, erhöht sich automatisch mit jedem INSERT

Constraints: Interrelationale Constraints

- REFERENCES und FOREIGN KEY
  - Spezifikation von Bedingungen zur referenziellen Integrität:
  - Auch für einzelne Attribute nach der Domain-Angabe:
    - REFERENCES Relation (Attrib)
  - Für mehrere Attribute:
    - FOREIGN KEY(Attrib1, Attrib2)
      REFERENCES Relation (Attrib1, Attrib2)

```
references TableName1(AttrName1)
```

```
foreign key (AttrName)
   references TableName1(AttrName1)
```

Constraints: Namen

- CONSTRAINT
  - Erlaubt intra- und interrelationalen Constraints eigene Namen zu geben
  - Damit leichter änderbar oder löschbar
  - Beispiel:
    - ```
      constraint   Angest_PK
      primary key (SVN),

      constraint   Angest_Vorgesetzt_FK
      foreign key (Vorgesetzte) references Angestellte(SVN)
      ```
      ```

Constraints: Reaktion auf Constraintverletzung

- Um Constraintverletzungen bei Änderungen zu vermeiden, Aktion spezifizieren
  - CASCADE
    - Propagiert die durchgeführte Änderung in die referenzierende Relation
  - SET NULL
    - Setzt die (wertebasierte) Referenz auf NULL
  - SET DEFAULT
    - Setzt die (wertebasierte) Referenz auf den für das Attribut vorgesehenen Default-Wert.
  - NO ACTION (manchmal auch: RESTRICT)
    - Verbietet Änderungen in einer referenzierten Relation – solange Abhängigkeiten bestehen
- Für DELETE und UPDATE getrennt spezifizierbar:
  - ON DELETE [Action]
    ON UPDATE [Action]

Constraints: Beispiel 1

```
create table Abteilung (
    Abt             int             not null    default 1,
    ...,
    constraint PK__Angest PRIMARY KEY (SVN),
    constraint FK__Angest_VorgesSVN
       foreign key (Vorgesetzte) references Angestellte(SVN)
       on delete set null
       on update cascade,
    constraint FK__Angest_Abt
       foreign key (Abt) references Abteilung(Nummer)
       on delete set default
);
```

Constraints: Beispiel 2

```
create table Abteilung (
    ...,
    Leitung      char(12)    not null    default '000000000000',
    ...,
    constraint PK__Abt
       primary key (Nummer),
    constraint UQ__AbtName
       unique (Name),
    constraint FK__Abt_Mgr
       foreign key (Leitung) references Angestellte(SVN)
       on delete set default
);
```

Constraints: Beispiel 3

```
create table AbtStandort (
    ...,
    primary key (AbtNummer, AStandort),
    ...,
    constraint FK__Stand_Abt_Nummer
       foreign key (AbtNummer) references Abteilung(AbtNummer)
       on delete cascade
);
```

- Oder nach der Definition der Tabelle(n)

```
alter table Angestellte
add constraint
      Angest_CK check (Gehalt/168 > 11.50);  -- Mindestlohn
```

Katalog: INFORMATION_SCHEMA

- Katalog eines DBMS
  - Beinhaltet Informationen über
    - Namen und Größe von Dateien
    - Namen und Datentypen von Datenelementen
    - Speicherdetails für jede Datei
    - Mappinginformation zwischen Schemata
    - Constraints

- SQL: Spezielles, vordefininiertes Schema **`information_schema`**
  - Enthält Metadaten zur Datenbank:
    - Welche Schemata
    - Welche Tabellen
    - Welche Attribute
    - …
  - Können auch mit SQL abgefragt werden, z.B.:
    - `select * from information_schema.tables;`

Schemata ändern

ALTER, DELETE

Löschen von DB-Konstrukten: DROP

- DROP [SCHEMA | TABLE | VIEW] Name;
  - Löscht ein Schema / Relation / Sicht
- ALTER TABLE TableName
  DROP [COLUMN | CONSTRAINT] Name;
  - Löscht eine Spalte / Einschränkung einer Tabelle

```
drop [schema | table | view] Name;
```

```
alter table TableName
drop [column | constraint | view] Name;
```

- Liste von Objekten möglich
- Kann durch CASCADE oder RESTRICT kontrolliert werden

Sichten (VIEWs) kommen später dran,
da wir dafür SQL Queries brauchen.

Ändern einer Relation: ALTER

- Mögliche Änderungen:
  - Hinzufügen eines neuen Attributs
  - Entfernen eines Attributs
  - Ändern einer Attributdefinition
  - Hinzufügen von zusätzlichen Relationeneinschränkungen
  - Entfernen von Relationeneinschränkungen
- Werte für neue Attribute:
  - Default-Wert, manuelle Zuweisung, NULL

```
alter TableName
[add AttrName Domain]
[alter AttrName drop constr]
[rename TableName1]
```

Ändern einer Relation: ALTER – Beispiele

- Beispiele:
  - `alter table Unternehmen.Angestellte add Job varchar(12);`
  - `alter table Unternehmen.Angestellte alter VorgesSVN drop default;`
  - `alter table Unternehmen.Angestellte rename Unternehmen.Mitarbeitende;`
  - Vorherige Folien: Primärschlüssel / CHECK-Constraint hinzugefügt, Attribut gelöscht

```
alter TableName
[add AttrName Domain]
[alter AttrName drop constr]
[rename TableName1]
```

Zwischenzusammenfassung

- Schema als Menge von Relationen: CREATE SCHEMA
- Relationen: CREATE TABLE
  - Liste von Attributen mit Wertebereichen und Constraints
- Wertebereiche
  - Basisdatentypen: INT, FLOAT, …, CHAR(n), VARCHAR(n), DATE, …
  - Benutzerdefiniert zur besseren Lesbarkeit / Wartbarkeit: CREATE DOMAIN
- Constraints
  - UNIQUE, NOT NULL, DEFAULT, AUTO_INCREMENT, PRIMARY KEY, FOREIGN KEY … REFERENCES, CHECK
  - Namen für Constraints über CONSTRAINT für Lesbarkeit / Wartbarkeit; Aktionen zur Vermeidung von Constraintverletzungen: CASCADE, SET NULL, SET DEFAULT, NO ACTION
- Strukturelle Änderungen: DROP, ALTER

Praxis-Test mit DBeaver

- Tabellen ändern

Dbeaver Screenshot

Weitere Online-Ressource:
https://www.w3schools.com/sql/default.asp
Achtung: SQL Befehle teilweise implementierungsabhängig

Überblick: 5. Structured Query Language (SQL)

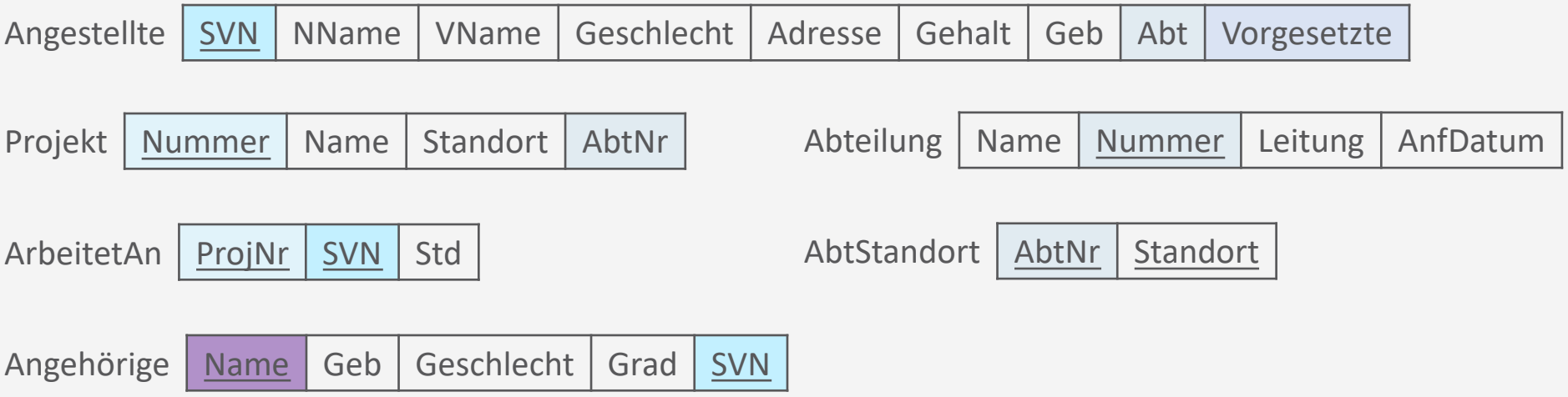
A. Datendefinition (SQL als DDL)
- Schema, Tabellen, Datentypen, Constraints definieren
- Strukturelle Änderungen mittels drop, alter

B. Datenmanipulation (SQL als DML)
- Anfragen
- Datenänderungen

C. Und der Rest
- Sichten (SQL als VDL)
- Rechtevergabe (SQL als DCL)
- Programmiermethoden

Relationales Schema

- Relationales Schema (in 3NF/BCNF): Firma → Anlegen



- Relationale Algebra: Anfragen und Datenmanipulation
  - $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$; gruppieren, aggregieren
  - Insert, delete, update

Anfragen: SELECT-Anweisung

$R \bowtie_{\langle AttrList \rangle} S$
$R \times S$

$\pi_{\langle AttrList \rangle}(R)$

```
select <Attribut- und Funktionsliste>
from <Relationenliste>
[where <Bedingung>]
[group by <Gruppierungsattribut(e)>]
[having <Gruppenbedingung>]
[order by <Attributliste>];
```

$\sigma_{\langle Bedingung \rangle}(R)$
$R \bowtie_{\langle Bedingung \rangle} S$

$\langle Gruppierungsattribute \rangle \mathcal{F}_{\langle Funktion\ Attribut \rangle}(R)$

SELECT …
FROM …
WHERE …

Selektion, Projektion
Mengeneigenschaft und Reihenfolge

```
select [distinct] List
from Table
[where <Condition>]
[order by {Attr [asc | desc]}];
```

SELECT: Grundstruktur

- Umsetzung der Projektion (π)
  - RA: $\pi_{\langle Attributliste \rangle}(R)$
  - SQL: SELECT$\langle Attributliste \rangle$ FROM R;
    - Wählt die Attribute aus der Relation aus, die in der Liste genannt sind
    - Sonderzeichen: * wählt alle Attribute einer Relation (keine Projektion)
  - Achtung: In SQL keine automatische Duplikatseliminierung

```
select List
from Table;
```

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

SELECT: Grundstruktur – Beispiele

```
select List
from Table;
```

- Liefere alle Attributausprägungen der Relation ANGESTELLTE, i.e., den Relationenzustand:
  - $\pi_{SVN,NName,VName,Geschlecht,Adresse,Gehalt,Geb,Abt,Vorgesetzte}(Angestellte)$
  - ```
    select *
    from Angestellte;
    ```
- Liefere Vornamen und Nachnamen aller Angestellten der Firma:
  - $\pi_{VName,NName}(Angestellte)$
  - ```
    select VName, NName
    from Angestellte;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

WHERE-Klausel

- Umsetzung der Selektion (σ) mit anschließender Projektion:
  - RA: $\sigma_{\langle Bedingung \rangle}(R)$
  - SQL: SELECT *
    FROM R
    [WHERE $\langle Bedingung \rangle$];
    - Mit anschließender Projektion, i.e., $\pi_{\langle Attributliste \rangle}\left(\sigma_{\langle Bedingung \rangle}(R)\right)$: $\langle Attributliste \rangle$ anstatt *
  - Beispiel: Liefere Geburtsdatum und Adresse der Angestellten mit Namen John Smith:
    - $\pi_{Geb,Adresse}\left(\sigma_{VName=„John“ \wedge NName=„Smith“}(Angestellte)\right)$
    - ```
      select Geb, Adresse
      from Angestellte
      where VName='John' and NName='Smith';
      ```

```
select List
from Table
[where <Condition>];
```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
```

Relationen als Mengen in SQL

- SQL behandelt Relationen als Multimengen
  - Effekt: Keine automatische Duplikatsentfernung
  - Wenn gewünscht: SELECT DISTINCT $\langle Attributliste \rangle$…
- Gründe
  - Entfernung von Duplikaten ist eine teure Operation
    - Möglichkeit der Implementierung: alle Tupel zunächst zu sortieren, anschließend/dabei Duplikate entfernen
  - Oft nicht nötig
    - Viele praktische Anwendungen von DBs sind so, dass Nutzer*innen Duplikate in der Ergebnismenge wünschen
  - Manchmal falsch
    - Aggregatfunktionen: Duplikatseliminierung könnte Ergebnis verfälschen zu intendiertem Ergebnis

```
select [distinct] List
from Table
[where <Condition>];
```

Gründe für keine automatische Duplikatsentfernung?

Ordnen durch ORDER BY

- Resultatmenge kann für die Ausgabe sortiert werden
  - SELECT … ORDER BY <Attribut> ASC
  - SELECT … ORDER BY <Attribut> DESC
  - ASC / DESC muss nicht angegeben werden:
    - SELECT … ORDER BY <Attribut>
    - Dann greift ein Default-Wert: ASC
      - Wenn nichts angegeben ist, wird aufsteigend sortiert
  - Ordnung auf <Attribut> muss definiert sein
  - Auch mehrere Sortierkriterien möglich

```
select [distinct] List
from Table
[where <Condition>]
[order by {Attr [asc | desc]}];
```

Ordnen durch ORDER BY: Beispiel

```
select [distinct] List
from Table
[where <Condition>]
[order by {Attr [asc | desc]}];
```

- Gebe für alle Angestellte deren Nachname, Vorname und Abteilung sortiert nach Abteilungsnummer, dann Nachname und dann Vorname (lexikographisch aufsteigend) an:
  - ```
    select Abt, NName, VName
    from Angestellte
    order by Abt, NName, VName;
    ```
- Bei absteigender Sortierung der Abteilungsnamen und aufsteigender Sortierung des Rests:
  - ```
    select Abt, NName, VName
    from Angestellte ang
    order by AName DESC, NName ASC, VName ASC;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Relationen kombinieren 1

Kombinierende Operationen: Kartesisches Produkt, Join und Join-Arten, klassische Mengenoperationen (Vereinigung, Schnitt, Differenz)

Umbenennung

Relationen kombinieren: Kartesisches Produkt

- Umsetzung des kartesischen Produkts (×) durch Auflistung der beteiligte Tabellen in FROM
  - RA: $R \times S$
  - SQL: SELECT *
    FROM R, S;

```
select [distinct] List
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

- Beispiel:
  - Liefere alle möglichen Kombinationen von Nachnamen und Abteilungsnamen der Firma geordnet erst nach Abteilung und dann nach Nachname:
    - $\pi_{NName,Name}(ANGESTELLTE \times ABTEILUNG)$
    - ```
      select distinct Name, NName
      from Angestellte, Abteilung
      order by Name, NName;
      ```

Abteilung

Name	Nummer	Leitung	AnfDatum

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Relationen kombinieren: JOIN

- Umsetzung des JOIN (⋈) über zwei Wege möglich:
  1. Implizit über kartesisches Produkt:
     - RA: $\sigma_{\langle Bedingung \rangle}(R \times S)$
     - SQL: SELECT *
       FROM R, S
       WHERE $\langle Bedingung \rangle$;
       - Effiziente Umsetzung als JOIN durch DBMS, auch wenn als $\sigma_{\langle Bedingung \rangle}(R \times S)$ formuliert
  2. Explizit über Schlüsselworte [INNER] JOIN mit Spezifizierung der JOIN-Bedingung über ON:
     - RA: $R \bowtie_{\langle Bedingung \rangle} S$
     - SQL: SELECT *
       FROM R INNER JOIN S ON $\langle Bedingung \rangle$;
- Häufig Verknüpfung über Fremdschlüssel

```
select [distinct] List
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

Abteilung

Name	Nummer	Leitung	AnfDatum

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Relationen kombinieren: JOIN – Beispiele

- Gebe die Nachnamen aller Angestellten mit dem Namen der zugeordneten Abteilung geordnet erst nach Abteilung und dann nach Nachname aus

```
select [distinct] List
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

1. $\pi_{Name,NName}(\sigma_{Nummer=Abt}(Abteilung \times Angestellte))$
   - ```
     select Name, NName
     from Abteilung, Angestellte
     where Nummer = Abt
     order by Name;
     ```
2. $\pi_{Name,NName}(Abteilung \bowtie_{Nummer=Abt} Angestellte)$
   - ```
     select Name, NName
     from Abteilung inner join Angestellte on Nummer = Abt
     order by Name;
     ```

Abteilung

Name	Nummer	Leitung	AnfDatum

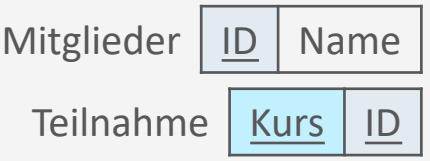
Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Uneindeutigkeit in Statements

- Problem: Gleiche Attributnamen in zu verknüpfenden Tabellen
- Beispiel:
  - Relationen mit Schemata $Mitglieder(\underline{ID}, Name), Teilnahme(\underline{Kurs, ID})$
  - Liefere Namen der Mitglieder, die den Kurs 'Sport101' belegen:
    - $\pi_{Name}(\sigma_{Kurs='Sport101'}(Teilnahme) * Mitglieder)$
    - ```
      select Name
      from Mitglieder, Teilnahme
      where Kurs='Sport101' and ID = ID;
      ```

```
create table Mitglieder (
   ID     int         not null,
   Name   char(20)    not null,
   primary key (ID)
);
create table Teilnahme (
   Kurs   char(25)    not null,
   ID     int         not null,
   primary key(Kurs,ID)
);
```


```

Qualifizierung von Attributen & Aliases

- Über Punktnotation Attributname mit Tabellennamen qualifizieren zur eindeutigen Referenzierung
  - Verbessert Lesbarkeit, da im Statement angegeben ist, woher welches Attribut kommt
  - Beispiel
    - ```
      select Name
      from Mitglieder, Teilnahme
      where Kurs = 'Sport101' and Mitglieder.ID = Teilnahme.ID;
      ```
- Aliases für lange Tabellennamen
  - Dient auch der Übersicht bei komplexen Ausdrücken
  - Beispiel
    - ```
      select Name
      from Mitglieder m, Teilnahme k
      where k.Kurs = 'Sport101' and m.ID = k.ID;
      ```

Mitglieder | ID | Name

Teilnahme | Kurs | ID

Zurück zu Relationen kombinieren: Natürlicher JOIN

- Umsetzung des natürlichen JOINs (⋈ ohne Bedingung) über Schlüsselworte NATURAL JOIN
  - RA: $R \bowtie S$
  - SQL: SELECT *
    FROM R NATURAL JOIN S
    - Wie auch in relationaler Algebra: Keine Spezifizierung der JOIN Bedingung, nur je eine Spalte behalten
- Beispiel
  - Liefere Namen der Mitglieder, die den Kurs 'Sport101' belegen:
    - $\pi_{Name}(\sigma_{Kurs='Sport101'}(Teilnahme) * Mitglieder)$
      bzw. $\pi_{Name}\left(\sigma_{Kurs='Sport101'}(Teilnahme * Mitglieder)\right)$
    - ```
      select Name
      from Mitglieder natural join Teilnahme
      where Kurs='Sport101';
      ```

Mitglieder | ID | Name

Teilnahme | Kurs | ID
      ```

Relationen kombinieren: OUTER JOIN

- Umsetzung der Outer-Join Varianten ($\ltimes$, $\bowtie$, $\rtimes$) über Schlüsselworte RIGHT / LEFT / FULL OUTER JOIN mit Spezifizierung der JOIN Bedingung wieder über ON
  - RA: $R \ltimes_{\langle Bedingung \rangle} S$ | $R \rtimes_{\langle Bedingung \rangle} S$ | $R \bowtie_{\langle Bedingung \rangle} S$
  - SQL: SELECT *
    FROM R [RIGHT | LEFT | FULL] OUTER JOIN S ON $\langle Bedingung \rangle$;
- Beispiel
  - Gebe die Nachnamen aller Angestellten aus und wenn existent, dazu den Namen ihrer Abteilung
    - $\pi_{NName,AbtName}(Abteilung \ltimes_{Nummer=Abt} Angestellte)$
    - ```
      select NName, Name
      from Abteilung right outer join Angestellte on Nummer = Abt
      ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Mehrfach-JOINs

- Umsetzung durch Verkettung
  1. Mittels AND in der WHERE Klausel
  2. Liste expliziter JOIN-Klauseln
     - Exemplarisch für drei Tabellen:
       FROM Table1 INNER JOIN Table2 ON <Join-Bedingung> INNER JOIN Table3 ON <Join-Bedingung2>
       - Man kann auch Klammern setzen, aber schränkt damit das DBMS in seinen Möglichkeiten zur Optimierung ein
     - Ergebnis unabhängig von Auswertungsreihenfolge
       - Aber: Größe der Zwischenergebnisse kann durch Reihenfolge beeinflusst werden

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Mehrfach-JOINs: Beispiel

- Gebe die Projektnummern, die Nummer der verantwortlichen Abteilung sowie den Namen, die Adresse und das Geburtsdatum der jeweiligen Abteilungsleitung für die Projekte am Standort Stafford aus

1.
```
select p.Nummer, p.AbtNr, ang.NName, ang.Adresse, ang.Geb
from Projekt p, Abteilung abt, Angestellte ang
where p.AbtNr = abt.Nummer and abt.Leitung = ang.SVN
            and p.Standort = 'Stafford';
```

2.
```
select p.PNummer, p.Abt, ang.NName, ang.Adresse, ang.Gdatum
from Projekt p inner join Abteilung abt on p.AbtNr = abt.Nummer
               inner join Angestellte ang on abt.Leitung = ang.SVN)
where p.Standort = 'Stafford';
```

Was für Effekte hat es, wenn man zuerst
- nach p.PStandort='Stafford' selektiert?
- die Joins durchführt?

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Self-Join

- Wenn im JOIN die gleiche Tabelle mehrfach vorkommt
  - Nutzung von Aliases um die Vorkommen zu unterscheiden
- Beispiel
  - Führe für jeden Angestellten seinen Nachnamen sowie den Nachnamen seines unmittelbaren Vorgesetzten auf:
    - ```
      select a.NName, v.NName
      from Angestellte a, Angestellte v
      where a.Vorgesetzte = v.SVN;
      ```
  - Problem: Gleiche Attributnamen im Endresultat: `a.NName, v.NName...`
    - Nicht nur bei Self-Joins ein Problem:
      - `select Produkt.Name, Projekt.Name...`

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Umbenennung mittels Namen in SQL-Statement

- Umsetzung der Umbenennung (ρ) von Attributen im *Endresultat* über AS
  - RA: $\rho_{(B_1,\ldots,B_n)}(R)$
  - SQL: SELECT A_1 AS B_1, …, A_n AS B_n FROM R

```
select [distinct] {Attr as Name}
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

- Beispiele
  - Beispiel auf vorheriger Folie zu Self-Joins
    - ```
      select a.NName as Untergebene, v.NName as Vorgesetzte,
      from Angestellte a, Angestellte v
      where a.Vorgesetzte=v.SVN;
      ```
  - Nicht-Self-Join-Beispiel von vorheriger Folie:
    - `select Produkt.Name as Produkt, Projekt.Name as Projekt ...`
      ```

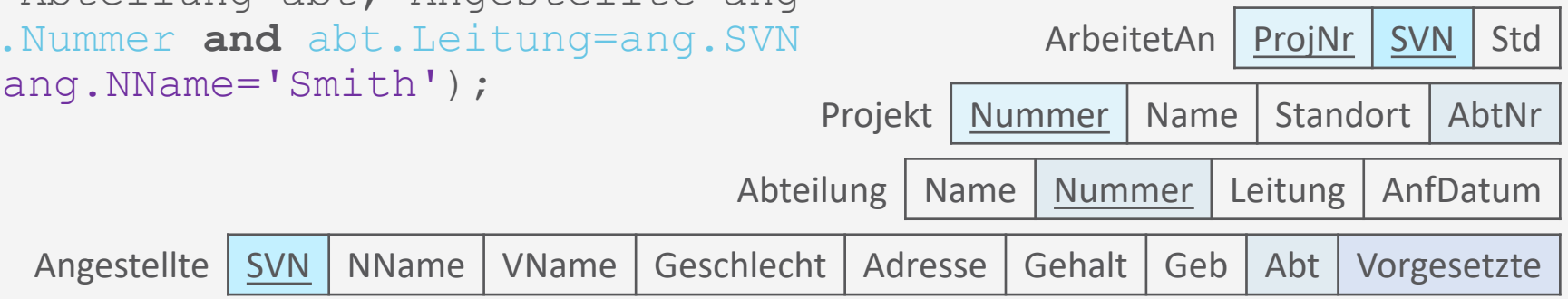
Mengenoperationen in SQL

- Umsetzung der Mengenoperationen (∪,∩, −) über entsprechende Schlüsselworte in Kombination mit SELECT-Ausdrücken zur Bestimmung der zu verarbeitenden Relationen
  - RA: $R \cup S$ $R \cap S$ $R - S$
  - SQL:
    (SELECT * FROM R) UNION (SELECT * FROM S)
    (SELECT * FROM R) INTERSECT (SELECT * FROM S)
    (SELECT * FROM R) MINUS (SELECT * FROM S)
    - Statt MINUS geht auch EXCEPT
  - Ergebnis von Mengen-Operationen: Tupel-Mengen
    - D.h., hier werden Duplikate aus dem Ergebnis entfernt
    - Sollen Duplikate erhalten bleiben, so muss das Schlüsselwort ALL ergänzt werden:
      - UNION ALL, INTERSECT ALL, EXCEPT ALL

```
select-statement
[  union
 | intersect
 | except
 | minus] [all]
select-statement;
```

Mengenoperationen in SQL: Beispiel

- Erstelle eine Liste aller Projektnummern von Projekten, an denen ein Mitarbeiter mit Nachnamen 'Smith' als Mitarbeiter oder Leiter der Abteilung arbeitet, die das Projekt kontrolliert
  -
```
(select p.Nummer
 from Projekt p, ArbeitetAn arb, Angestellte ang
 where p.Nummer=arb.ProjNr and arb.SVN=ang.SVN
            and ang.NName='Smith')
union
(select p.Nummer
 from Projekt p, Abteilung abt, Angestellte ang
 where p.Abt=abt.Nummer and abt.Leitung=ang.SVN
            and ang.NName='Smith');
```



Aggregatfunktion und Gruppierung

Aggregatfunktion und Gruppierung

Aggregatfunktionen

- Umsetzung der Aggregation über Angabe des Funktionsnamens in der Attributliste des SELECT
  - RA: $\mathcal{F}_{\langle Liste\ von\ (Funktion,Attribut)\ Paaren \rangle}(R)$
  - SQL: SELECT $\langle Liste\ von\ Funktion(Attribut) \rangle$ FROM R
  - Standardfunktionen:
    - COUNT($Attribut$) Anzahl der Tupel; mit COUNT(DISTINCT($Attribut$)) Anzahl der verschiedenen Tupel
    - SUM($Attribut$) Summe der Werte der Tupelattribute
    - MIN($Attribut$) Wert des minimalen Tupelattributs
    - MAX($Attribut$) Wert des maximalen Tupelattributs
    - AVG($Attribut$) Durchschnittlicher Wert der Tupelattribute; mit AVG(DISTINCT($Attribut$)) Durchschnitt der verschiedenen Tupel

```
select {Funktion([distinct] Attr)}
from Table;
```

Aggregatfunktionen: Beispiel

```
select {Funktion([distinct] Attr)}
from Table;
```

- Liefere
  - die Summe der Gehälter,
  - das maximale Gehalt,
  - das durchschnittliche Gehalt und
  - das minimale Gehalt

  aller Angestellten
  - $\mathcal{F}_{\mathrm{SUM}(Gehalt),\mathrm{MAX}(Gehalt),\mathrm{AVG}(Gehalt),\mathrm{MIN}(Gehalt)}(Angestellte)$
  - ```
    select sum(Gehalt), max(Gehalt), avg(Gehalt), min(Gehalt)
    from Angestellte;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

Gruppierung mit GROUP BY

- In der Praxis: Aggregatfunktionen häufig auf Gruppen von Tupeln
  - Beispiel: Liefere die Summe der Gehälter, das maximale Gehalt, das durchschnittliche Gehalt und das minimale Gehalt *pro Abteilung*

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList]
[order by {Attr [asc | desc]}];
```

- Umsetzung von Gruppierungen
  - RA: $\langle B_1, \dots, B_m \rangle \mathcal{F}_{\langle Liste\ von\ (Funktion, Attribut)\ Paaren \rangle}(R)$
  - SQL: SELECT $\langle B_1, \dots, B_m \rangle$ $\langle Liste\ von\ Funktion(Attribut) \rangle$
    FROM R
    GROUP BY $B_1, \dots, B_m$
    - Nur was in der GROUP BY Klausel vorkommt, kann im SELECT als weiteres Attribut vorkommen
      - Es muss nicht jedes Attribut im SELECT vorkommen, macht aber die Ergebniszeilen weniger interpretierbar
    - Abarbeitungsreihenfolge: Erst gruppieren, dann aggregieren

Gruppierung mit GROUP BY: Beispiel

- Liefere
  - die Summe der Gehälter,
  - das maximale Gehalt,
  - das durchschnittliche Gehalt und
  - das minimale Gehalt

  der Angestellten *pro Abteilung*
  - $Abt\ \mathcal{F}_{\text{SUM}(Gehalt),\text{MAX}(Gehalt),\text{AVG}(Gehalt),\text{MIN}(Gehalt)}(Angestellte)$
  - **select** Abt, **sum**(Gehalt), **max**(Gehalt), **avg**(Gehalt), **min**(Gehalt)
    from Angestellte
    group by Abt;

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList]
[order by {Attr [asc | desc]}];
```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Gruppierung mit GROUP BY: Beispiel

Wie sortieren wir das Ergebnis absteigend nach Abteilungsgröße?

- Liefere für jede Abteilung die Nummer, die Anzahl der zugehörigen Angestellten und deren Durchschnittsgehalt

- $Abt\,\mathcal{F}_{\mathrm{COUNT}(*),\mathrm{AVG}(Gehalt)}(Angestellte)$
- ```
  select Abt, count(*), avg(Gehalt)
  from Angestellte
  group by Abt;
  ```

VName	NName	SVN	...	Gehalt	Vorgesetzte	Abt
John	Smith	01234567X890		30000	12345678Y901	5
Franklin	Wong	12345678Y901		40000	78901234E567	5
Ramesh	Narayan	23456789Z012		38000	12345678Y901	5
Joyce	English	34567890A123		25000	12345678Y901	5
Alicia	Zelaya	45678901B234	...	25000	56789012C345	4
Jennifer	Wallace	56789012C345		43000	78901234E567	4
Ahmad	Jabbar	67890123D456		25000	56789012C345	4
James	Bong	78901234E567		55000	null	1

Abt	COUNT(*)	AVG(Gehalt)
5	4	33250
4	3	31000
1	1	55000

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
  ```

Abarbeitungsreihenfolge von Aggregation, Gruppierung und JOIN

- Aggregation und Gruppierung werden nach einem JOIN angewendet
- Beispiel:
  - Liefere für jedes Projekt die Projektnummer, den Projektnamen und die Anzahl der daran arbeitenden Angestellten
    - $Nummer, Name\ \mathcal{F}_{\text{COUNT}(*)}(Projekt \bowtie_{Nummer=ProjNr} ArbeitetAn)$
    - ```
      select Nummer, Name, count(*)
      from Projekt, ArbeitetAn
      where Nummer = ProjNr
      group by Nummer, Name;
      ```
  - Erst JOIN, dann Gruppierung, dann Aggregat (COUNT)

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList]
[order by {Attr [asc | desc]}];
```

ArbeitetAn	ProjNr	SVN	Std

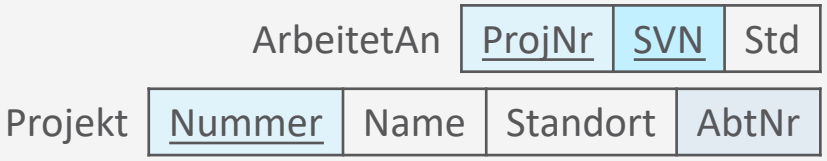
Projekt	Nummer	Name	Standort	AbtNr
```

Bedingungen auf Gruppierungen: HAVING

- HAVING für Bedingungen auf Gruppen in Gruppierungen
  - Entspricht WHERE auf Einzel-Tupeln
  - Tritt nur zusammen mit GROUP BY auf
- Beispiel:
  - Liefere für jedes Projekt, an dem mehr als zwei Angestellte arbeiten, die Projektnummer, den Projektnamen und die Anzahl der daran jeweils arbeitenden Angestellten

```
select Nummer, Name, count(*)
from Projekt, ArbeitetAn
where Nummer = ProjNr
group by Nummer, Name
having count(*) > 2;
```

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```



Beispiel: HAVING

Tabelle ist das Ergebnis nach JOIN und Anwendung der GROUP BY Klausel

Name	Nummer	...	SozVersNr	Stunden
ProduktX	1		01234567X890	32,5
ProduktX	1		34567890A123	20,0
ProduktY	2		01234567X890	7,5
ProduktY	2		34567890A123	20,0
ProduktY	2		12345678Y901	10,0
ProduktZ	3		89012345F678	40,0
ProduktZ	3		12345678Y901	10,0
Computerisation	10		12345678Y901	10,0
Computerisation	10	...	45678901B234	10,0
Computerisation	10		67890123D456	35,0
Reorganisation	20		12345678Y901	10,0
Reorganisation	20		56789012C345	15,0
Reorganisation	20		78901234E567	null
Newbenefits	30		67890123D456	5,0
Newbenefits	30		56789012C345	20,0
Newbenefits	30		45678901B234	30,0

Diese Gruppen fliegen raus

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```

```
select Nummer, Name, count(*)
from Projekt, ArbeitetAn
where Nummer = ProjNr
group by Nummer, Name
having count(*) > 2;
```

ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr

Beispiel: HAVING - Ergebnisse

Name	Nummer	...	SozVersNr	Stunden
ProduktY	2		01234567X890	7,5
ProduktY	2		34567890A123	20,0
ProduktY	2		12345678Y901	10,0
Computerisation	10		12345678Y901	10,0
Computerisation	10		45678901B234	10,0
Computerisation	10		67890123D456	35,0
Reorganisation	20		12345678Y901	10,0
Reorganisation	20		56789012C345	15,0
Reorganisation	20		78901234E567	null
Newbenefits	30		67890123D456	5,0
Newbenefits	30		56789012C345	20,0
Newbenefits	30		45678901B234	30,0

Nach Anwendung der HAVING Klausel

Nummer	COUNT(*)
2	3
10	3
20	3
30	3

Nach COUNT und SELECT (PName nicht dargestellt)

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```

```
select Nummer, Name, count(*)
from Projekt, ArbeitetAn
where Nummer = ProjNr
group by Nummer, Name
having count(*) > 2;
```

ArbeitetAn (ProjNr, SVN, Std)

Projekt (Nummer, Name, Standort, AbtNr)

Weiteres Beispiel für Gruppierung

- Liefere für jedes Projekt
  - die Nummer,
  - den Namen und
  - die Anzahl der Angestellten,

  die aus Abteilung 5 an dem betreffenden Projekt arbeiten
  - ```
    select Nummer, Name, count(*)
    from Projekt p, ArbeitetAn arb, Angestellte ang
    where p.Nummer=arb.ProjNr and arb.SVN=ang.SVN and ang.Abt=5
    group by Nummer, Name;
    ```

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```

ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
```

Weitere Klauseln, Tests und Vergleichsmöglichkeiten

String-Tests

Element-Test

Zeichenvergleiche

- Gleichheit von Substrings: LIKE und zwei Sonderzeichen/Platzhalter
  - % : beliebige Anzahl von Zeichen
  - _ : genau ein Zeichen
- Beispiele:
  - Liefere eine Liste der Vor- und Nachnamen aller Angestellten, die in Houston/Texas wohnen
    - ```
      select VName, NName
      from Angestellte
      where Adresse like '%Houston, TX%';
      ```
  - Liefere eine Liste der Vor- und Nachnamen aller Angestellten, deren SVN an der dritten Stelle die Ziffer 8 besitzt
    - ```
      select VName, NName
      from Angestellte
      where SVN like '_ _ 8 _ _ _ _ _ _ _ _ _';
      ```

Operatoren

- Für Zahlen: Arithmetische Operatoren (+, -, *, /)
- Für Zeichenketten: Verbindungsoperator (||)
- Für Datum, Zeit, Zeitstempel und Intervall: Plus und Minus (+, -)
- Vergleichsoperator BETWEEN für Intervall-Prüfung

- Beispiele:
  - 10% Lohnerhöhung testen, i.e., eine Liste aller Angestellten mit Vor- und Nachname und deren Gehalt, um 10% erhöht
  - E-Mail-Liste für Angestellte der Abteillung 5, die Gehalt zwischen 30.000 und 40.000 beziehen (Annahme: Angestellte haben ein Email-Attribut):

Operatoren

- Beispiele:
  - 10% Lohnerhöhung testen:
    - ```
      select ang.VName, ang.NName, 1.1*ang.Gehalt AS PlusGehalt
      from Angestellte ang, ArbeitetAn arb, Projekt p
      where ang.SVN=arb.SVN and arb.PNr=p.Nummer and p.Name='ProductX';
      ```
  - E-Mail-Liste (Annahme: Angestellte haben ein Email-Attribut):
    - ```
      select  VName || '  ' || NName || '  <' || Email || '>'
      from Angestellte
      where (Gehalt between 30000 and 40000) and Abt=5;
      ```
    - Ausgabe bei VName='John', NName='Smith‘, Email='js@blub.de‘, Gehalt=35000, Abt=5:
      'John Smith <js@blub.de>'

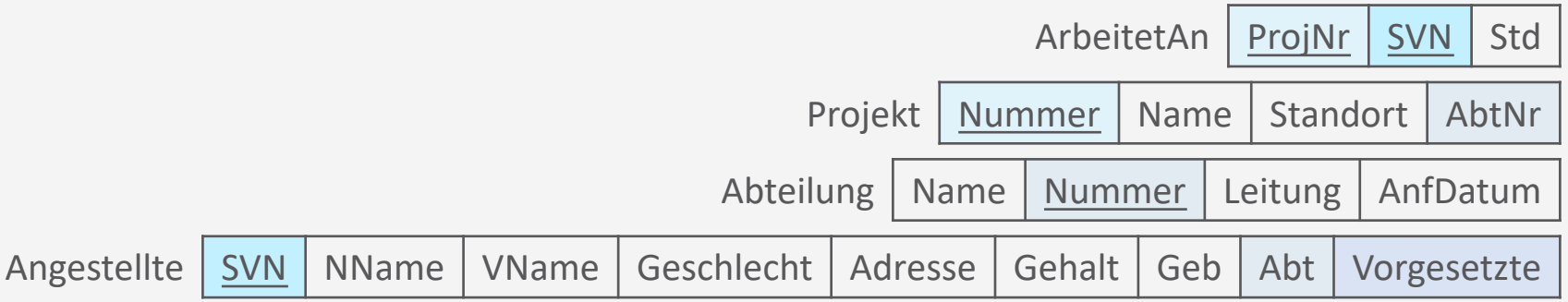
ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Verschachtelte Anfragen und disjunktiver Elementtest: IN

- Innerhalb des WHERE Ausdrucks können wiederum SELECT Ausdrücke formuliert werden, deren Resultate dann im WHERE Ausdruck weiterverwendet werden
  - Üblicherweise für weitere Vergleiche
- Möglichkeit des Vergleichs: disjunktiver Elementtest mittels IN
- Beispiel: Formulierung der UNION-Query (vgl. Folie 60) ohne UNION
  - Erstelle eine Liste aller Projektnummern von Projekten, an denen ein Mitarbeiter mit Nachnamen 'Smith' als Mitarbeiter oder Leiter der Abteilung arbeitet, die das Projekt kontrolliert



Verschachtelte Anfragen und disjunktiver Elementtest: Beispiel

- Erstelle eine Liste aller Projektnummern von Projekten, an denen ein Mitarbeiter mit Nachnamen 'Smith' als Mitarbeiter oder Leiter der Abteilung arbeitet, die das Projekt kontrolliert
  - ```
    select distinct Nummer
    from Projekt
    where Nummer in (select p.Nummer
                     from Projekt p, Abteilung abt, Angestellte ang
                     where p.AbtNr=abt.Nummer and abt.Leitung=ang.SVN
                                  and ang.NName='Smith‘)
                OR
          Nummer in (select arb.ProjNr
                     from ArbeitetAn arb, Angestellter ang
                     where arb.SVN=ang.SVN
                                  and ang.NName='Smith');
    ```
    ```

Verschachtelte Anfragen und disjunktiver Elementtest mit Gruppierung: Bsp.

- Liefere für jede Abteilung mit mehr als fünf Angestellten die Nummer und die Anzahl der Angestellten, die mehr als 40.000 verdienen.
  - ```
    select Abt, COUNT(*)
    from Angestellte
    where Gehalt>=40000 and Abt in (select Abt
                                    from Angestellte
                                    group by Abt
                                    having count(*) >= 5 )
    group by Abt;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

IN-Operator mit Tupeln

- IN-Operator kann auch zum Testen mit Tupeln verwendet werden
  - Tupel-Ausdruck muss UNION-kompatibel zum Resultat des inneren Ausdrucks sein
- Beispiel:
  - Zeige die Sozialversicherungsnummern aller Angestellten, die in einer gleichen Kombination von Projekt und Stunden an einem Projekt arbeiten, an dem auch der Angestellte ‘John Smith‘ mit der SVN 01234567X890 beschäftigt ist.
    - ```
      select distinct SVN
      from ArbeitetAn
      where (ProjNr, Std) IN (select ProjNr, Std
                              from ArbeitetAn
                              where SVN='01234567X890');
      ```

ArbeitetAn

ProjNr	SVN	Std
      ```

Sichtbarkeit von Variablen in verschachtelten Anfragen

- Variablen der äußeren Anfragen in der inneren Anfrage sichtbar (aber nicht anders herum)
  - *Korrelierte* Anfrage: innere Anfrage referenziert auf äußere Anfrage
  - Semantik: Innere Anfrage wird einmal für jedes Tupel ausgewertet
  - Beispiel
    - Liefere die Namen der Angestellten, die einen Angehörigen mit gleichem Vornamen und gleichem Geschlecht wie der Angestellte selbst haben
      - ```
        select a.NName, a.VName
        from Angestellte a
        where a.SVN in (select fam.SozVersNr
                        from Angehoerige fam
                        where a.VName=fam.Name and a.Geschlecht=fam.Geschlecht
        ```


        ```

Vergleichsoperatoren ANY und ALL

- Weitere Vergleichsoperatoren für ganze Mengen:
  - <op> ANY | SOME
  - <op> ALL
    - <op> ersetzbar durch ein der Operatoren {=,>, >=, <=, <, <>}
    - v > ALL V liefert bspw. dann TRUE, wenn v größer ist als alle Werte der Menge V
    - = ANY [oder auch = SOME] ist äquivalent zu IN
- Beispiel:
  - Gebe Nach- und Vorname aller aus, die ein höheres Gehalt haben als alle aus Abteilung 5
  - ```
    select NName, VName
    from Angestellte
    where Gehalt > all(select Gehalt
                       from Angestellte
                       where Abt=5 );
    ```
    ```

EXISTS-Test

- EXISTS überprüft, ob das das Resultat einer korrelierten verschachtelten Anfrage leer (FALSE) ist – also kein Tupel enthält – oder nicht (TRUE)
  - Kann auch mit NOT kombiniert werden
- Beispiel:
  - Liefere die Namen der Angestellten, die einen Angehörigen mit gleichem Vornamen und gleichem Geschlecht wie der Angestellte selbst haben
    - ```
      select a.NName, a.VName
      from Angestellte a
      where exists (select fam.SozVersNr
                    from Angehoerige fam
                    where a.SVN=fam.SozVersNr and a.VName=fam.Name and
      a.Geschlecht=fam.Geschlecht );
      ```

Angehörige

Name	Geb	Geschlecht	Grad	SVN

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

UNIQUE zum Duplikattest

- Überprüft, ob eine Multimenge Duplikate enthält (FALSE) oder nicht (TRUE).
- Beispiel:
  - Liefere die Namen der Angestellten, die einen eindeutigen Vornamen haben
    - ```
      select a.NName, a.VName
      from Angestellte a
      where unique (select b.VName
                    from Angestellte b
                    where a.VName=b.VName );
      ```

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Junktoren bei Mengenvergleichen

- Mengenvergleiche können durch Junktoren AND und OR miteinander verknüpft werden
  - Mehrere korrelierte Anfragen
- Beispiel:
  - Erstelle eine Liste mit den Namen der Manager, die mindestens einen Angehörigen haben
    - ```
      select a.NName, a.VName
      from Angestellte a
      where exists (select *
                    from Angehoerige fam
                    where a.SVN=fam.SozVersNr)
            and
            exists (select *
                    from Abteilung abt
                    where a.SVN=abt.Leitung);
      ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Explizite Mengenangaben

- Menge explizit angeben und in der WHERE-Klausel verwenden
- Beispiel:
  - Gib die Sozialversicherungsnummern aller Angestellten aus, die an Projekten mit den Nummern 1, 2 oder 3 arbeiten.
    - ```
      select distinct SVN
      from ArbeitetAn
      where ProjNr in (1, 2, 3);
      ```
      ```

NULL-Test

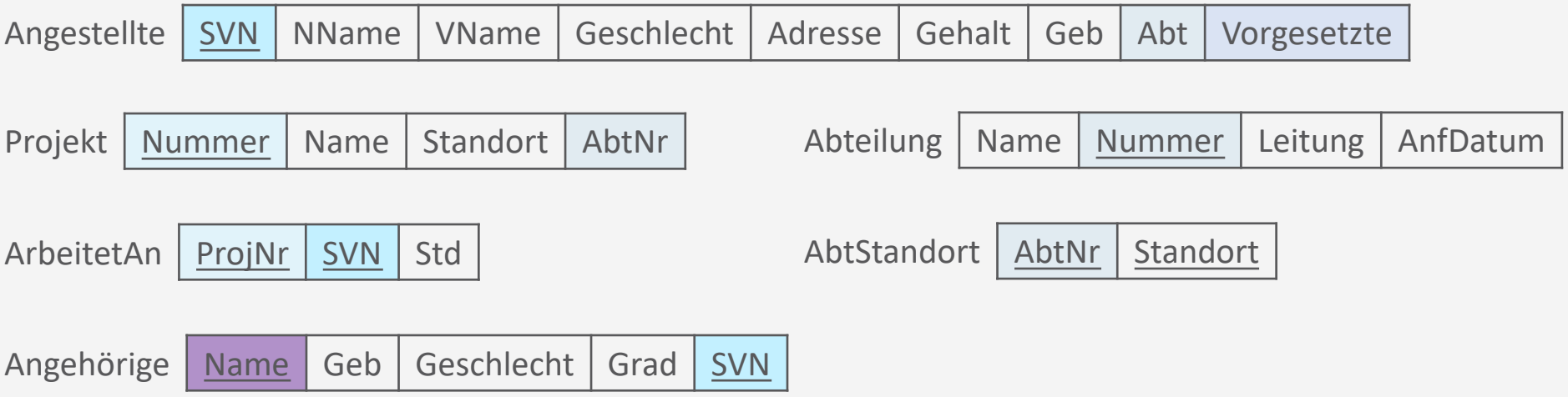
- Prüft, ob der Wert eines Attributs NULL ist
  - Hier kein = möglich! (IS NULL statt = NULL)
- Beispiel:
  - Liefere die Namen der Angestellten, die keinen Vorgesetzten haben.
    - ```
      select NName, VName
      from Angestellte
      where Vorgesetzte is null;
      ```
  - Liefere die Namen der Angestellten, die einen Vorgesetzten haben.
    - ```
      select NName, VName
      from Angestellte
      where Vorgesetzte is not null;
      ```

Daten ändern

INSERT, DELETE, UPDATE

Relationales Schema

- Relationales Schema (in 3NF/BCNF): Firma → Anlegen



- Relationale Algebra: Anfragen und Datenmanipulation
  - $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$; gruppieren, aggregieren
  - Insert, delete, update

Tupel einfügen

- INSERT benötigt:
  - Ziel-Relation
  - Ggf. Attribute (sonst Reihenfolge wie in Schemadefinition)
  - Werteliste oder Anfrage, die Werteliste produziert

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Tupel einfügen: Beispiele für Werteliste

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

- Neuen Angestellten mit allen Werten einfügen
  - ```
    insert into Angestellte
    values ('90123456G789', 'Marini', 'Richard', 'M', '98 Oak
    Forest, Katy, TX', 37000, '30.12.1962', 4, '67890123D456' );
    ```
- Neuen Angestellten mit Teilinformationen einfügen
  - ```
    insert into Angestellte (VName, NName, Abt, SVN)
    values ('Richard', 'Marini', 4, '90123456G789' );
    ```
    - Rest wird auf NULL oder DEFAULT gesetzt, wenn kein Constraint verletzt wird
- Neue Abteilung anlegen: Nummer wird automatisch eingefügt (SERIAL)
  - ```
    insert into Abteilung(Name, Leitung, AnfDatum)
    values ('Facilities', '90123456G789',
    '2019-06-14');
    ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

Tupel einfügen: Beispiele

- Einfügen von Daten aus einer anderen Tabelle:
  - Neue Tabelle:
    - ```
      create table AbtInfo (
      AbtName            varchar(15),
      AnzahlAngest       integer,
      GehaltGesamt       integer
      );
      ```
  - ```
    insert into AbtInfo (AbtName, AnzahlAngest, GehaltGesamt)
        select Name, count(*), sum(Gehalt)
        from (Abteilung inner join Angestellte on Nummer=Abt )
        group by Name;
    ```

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Abweisung von INSERT

- Wenn DB-Integrität verletzt wird
- Mögliche Fehlerquellen
  - Kein Primärschlüssel angegeben
  - Fremdschlüssel existiert nicht in Zieltabelle
  - Kein Wert angegeben trotz NOT NULL
  - CHECK-Constraint verletzt
  - Doppelter Wert trotz UNIQUE

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

Tupel löschen

- DELETE benötigt:
  - Name der Relation, aus der gelöscht werden soll
  - WHERE-Klausel, die bestimmt, welche Tupel gelöscht werden sollen
  - Darf ebenfalls Integrität nicht verletzen
    - Aktionen zur Behandlung (CASCADE, NO ACTION, SET DEFAULT, SET NULL)

```
delete from TableName
where <condition>;
```

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Tupel löschen: Beispiele

```
delete from TableName
where <condition>;
```

- Alle Angehörigen löschen
  - `delete from Angehoerige;`
- Alle Angehörigen des Angestellte mit SVN `34567890A123` löschen
  - ```
    delete from Angehoerige
    where SVN = '34567890A123';
    ```
- Alle Angehörigen der Angestellten löschen, die in Abteilung 4 arbeiten
  - ```
    delete from Angehoerige
    where SVN in(select SVN
                 from Angestellte
                 where Abt = 4 );
    ```

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Tupel aktualisieren

- UPDATE benötigt:
  - Name der Relation
  - SET – Anweisung für die Änderung
    - Kann auch Berechnungen enthalten
  - WHERE-Anweisung für die zu ändernden Tupel
- Reihenfolge mehrerer UPDATE-Anweisungen ist relevant!

```
update TableName
set <statement>
where <condition>;
```

Tupel aktualisieren: Beispiele

```
update TableName
set <statement>
where <condition>;
```

- Projekt mit der Nummer 5 ist jetzt am Standort Bellaire zu finden und wird von Abteilung mit der Nummer 5 betreut
  - ```
    update Projekt
    set Standort='Bellaire', AbtNr=5
    where Nummer=5;
    ```
- Alle Angestellte der Abteilung mit Namen Research erhalten 10% mehr Gehalt
  - ```
    update Angestellte
    set Gehalt=Gehalt * 1.1
    where Abt in(select Nummer
                 from Abteilung
                 where Name = 'Research');
    ```

Merge von Tabellen

- Eine Tabelle kann in eine andere eingefügt werden, wobei hier im WHEN MATCHED über das Einfügen bestimmt
  - Beispiel:

```
merge into AngestellteAll c
using Angestellte a
on (a.SVN = c.SVN)
when matched then
  update
  set
  c.VName = a.VName,
  c.NName = a.NName,
  c.Geb = a.Geb,    ...
when not matched then
  insert values(a.SVN, ..., a.Abt);
```

```
merge into TableName1
using TableName2
on <condition>
when matched then
   update
   set
   <statement>
when not matched then
   insert values ValueList;
```

Zwischenzusammenfassung

- Grundkonstrukte
  - SELECT, FROM, WHERE
- Sortierung
  - ORDER BY: ASC, DESC
- Relationen kombinieren
  - INNER / NATURAL / OUTER JOIN
  - UNION, INTERSECT, MINUS / EXCEPT
- Aggregationen und Gruppierungen
  - COUNT, SUM, AVG, MIN, MAX
  - GROUP BY
  - HAVING
- Weitere Vergleichsmethoden
  - Operatoren (+ - % / * IN LIKE…)
  - IN, ANY, ALL, EXIST, UNIQUE
  - DISTINCT
  - IS NULL / IS NOT NULL
- Datenänderung
  - INSERT
  - UPDATE
    - WHEN MATCHED
  - DELETE

SQL ist ein Standard, der u.a. eine Grammatik definiert
→ Implementierungen (MySQL, PostgreSQL, etc.) setzen den Standard um, woraus Unterschiede in der Nutzung über Systeme hinweg entstehen können

Überblick: 5. Structured Query Language (SQL)

A. Datendefinition (SQL als DDL)
- Schema, Tabellen, Datentypen, Constraints definieren
- Strukturelle Änderungen mittels drop, alter

B. Datenmanipulation (SQL als DML)
- Anfragen
- Datenänderungen

C. Und der Rest
- Sichten (SQL als VDL)
- Rechtevergabe (SQL als DCL)
- Programmiermethoden

Sichten (Views)

SQL als View Definition Language (VDL)
(Machmal auch als Teil der DML aufgefasst)

Sichten: Virtuelle Relationen

- VIEW: aus anderen Relationen abgeleitete Relation
  - Wird über eine SELECT-Anweisung spezifiziert
  - Werden von DB aktuell gehalten
  - Virtuelle Relation: kann, muss aber nicht in der Datenbank abgespeichert werden
  - VIEWs können für Abfragen wie normale Relationen genutzt werden

```
create view ViewName as
<select-statement>
```

ArbeitetAn	ProjNr	SVN	Std

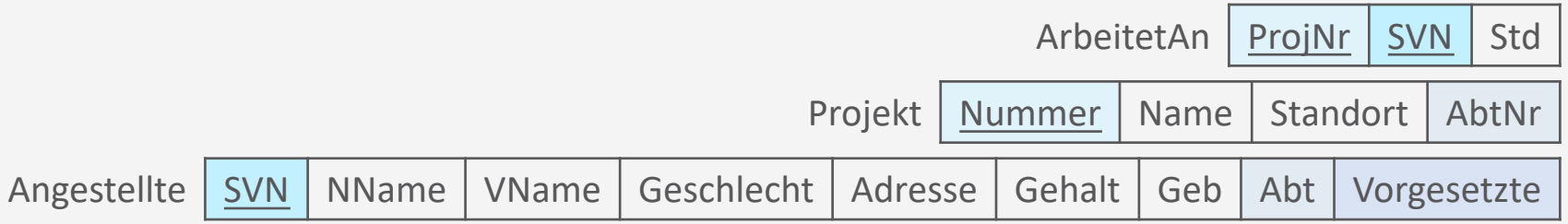
Projekt	Nummer	Name	Standort	AbtNr

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Sichten: Virtuelle Relationen

- Sicht auf Mitarbeitende mit Namen an Projekten
  - ```
    create view ArbeitetAnMitNamen as
    select ang.SVN, VName, NName, Name, Std
    from Angestellte ang, Projekt p, ArbeitetAn arb
    where ang.SVN=arb.SVN and arb.ProjNr=p.Nummer;
    ```
  - Anfrage
    - ```
      select VName, NName, Std as Stunden
      from ArbeitetAnMitNamen
      where SVN = '01234567X890';
      ```

```
create view ViewName as
<select-statement>
```



Verwendung von VIEWs

```
create view ViewName as
<select-statement>
```

- Aktualisierungen/Datenmanipulationen häufig nicht möglich
  - Non-updateable views
- Beispiel: Nutzung von Aggregationsfunktionen:
  - ```
    create view AbtInfo as
       select Abt, count(*) as Cnt, avg(Gehalt) as AvgG
       from Angestellte
       group by Abt;
    ```
  - `insert into AbtInfo values (1,5,20000)` ???
  - `update AbtInfo set Cnt = 5 where Abt = 5` ???
- Beispiel: Sei ein NOT NULL Attribut ohne DEFAULT nicht Teil des Views
  - Kein INSERT

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

Rechte

SQL als Data Control Language (DCL)

Rechtevergabe mittels SQL

- Rechtevergabe an DB-Objekten
- Benutzerprivilegien:
  - Lesen oder Ändern von Relationen oder Spalten
  - Anlegen von Relationenschemata oder Datenbankschemata
  - Weitergabe von Privilegien
- Rechte mittels
  - GRANT (ein ausgewähltes Recht) geben
  - REVOKE (entsprechendes Recht) wieder entziehen
    - SELECT, UPDATE, DELETE, INSERT aber auch
    - EXECUTE, ALTER, CREATE, MANAGE,….., etc

```
grant [Right]
   on Table(AttrList)
   to UserList;
grant [Right]
   on TableList
   to UserList;
```

```
revoke [Right]
   on Table(AttrList)
   from UserList;
revoke [Right]
   on TableList
   from UserList;
```

Rechtevergabe mittels SQL: Beispiele

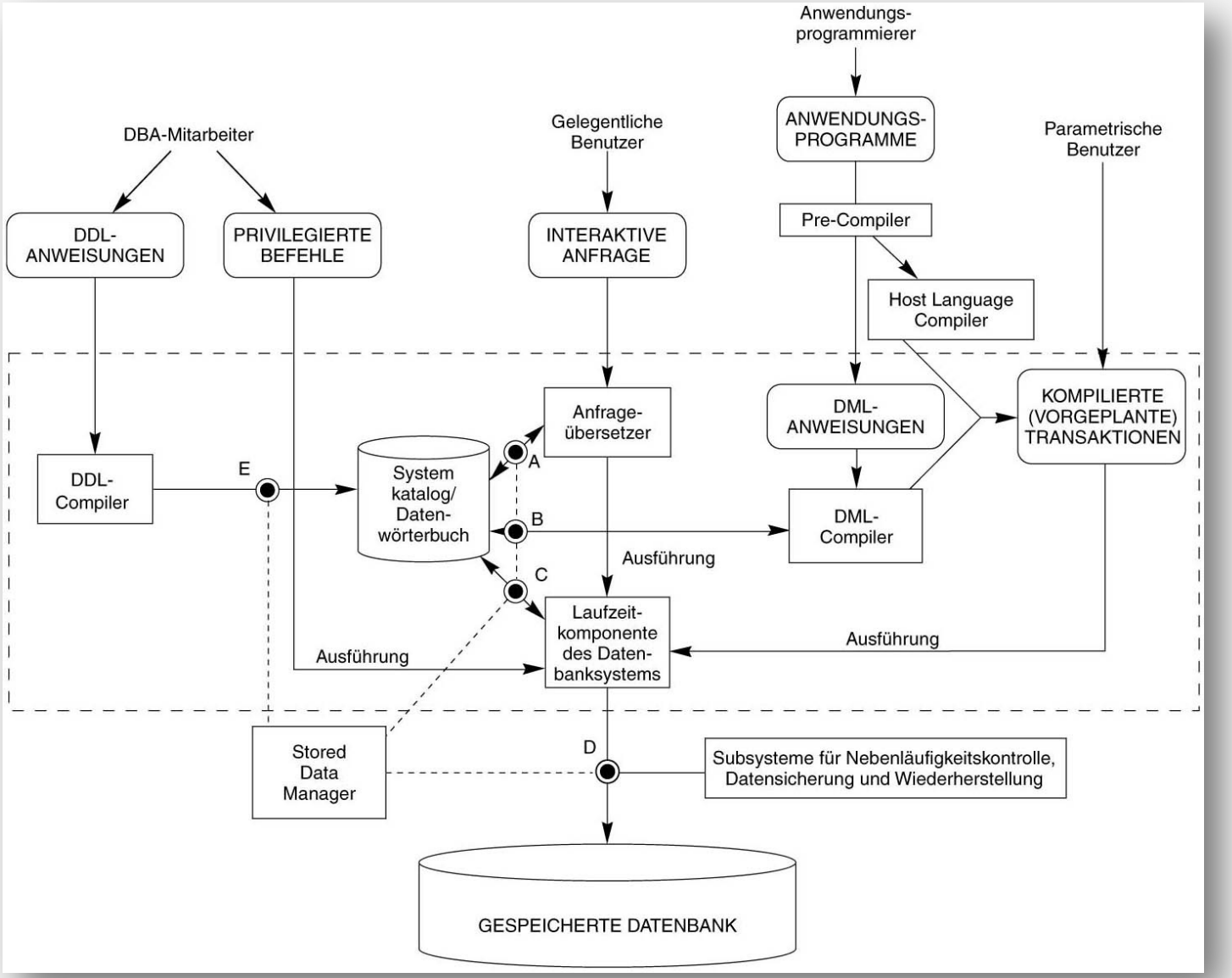
- Rechte auf anfragen und aktualisieren einer Tabelle
  - ```
    grant select, update on Nutzer.Tabelle
    to AndererNutzer;
    ```
- Rechte auf einfügen, anfragen, löschen einer Tabelle; Weitergabe von Rechten
  - ```
    grant insert, select, delete on Angestellter
    to joerg, sabine, harald with grant option;
    ```
- Recht auf aktualisieren von einem Attribut
  - ```
    grant update(Gehalt) on Angestellter
    to chefe;
    ```
- Zurücknehmen des Rechts ein Attribut zu aktualisieren
  - ```
    revoke update(Gehalt) on Angestellte
    from chefe;
    ```

Programmiermethoden

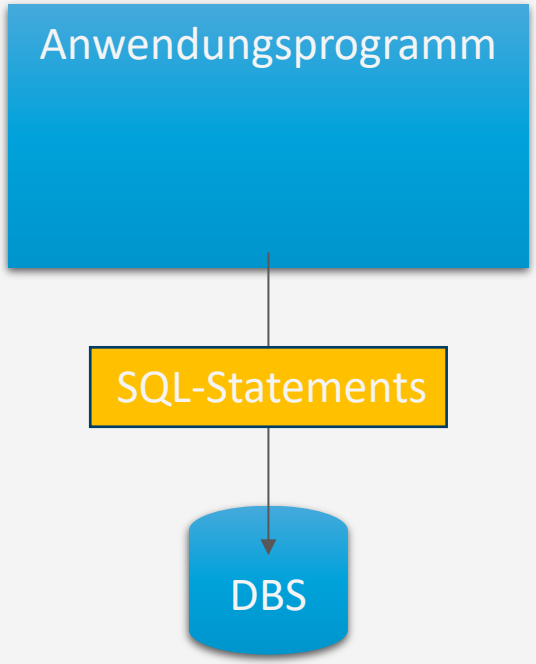
Benutzung von SQL

Erweiterte Systemumgebung

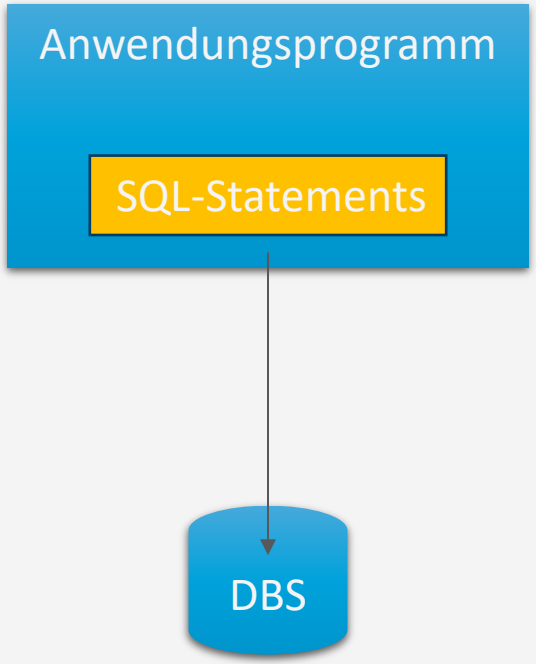
- Zugriffe von
  - DB-Administratoren
  - Anwendungen
  - Nutzer*innen
    - Direkt auf DB
      - Gelegentlich, interaktiv
    - Parametrisch
      - Vorgefertigte Anwendungsprogramme mit beschränktem Kommandovorrat (routinierte, wohldefinierte, formalisierte Befehle)



SQL Programmier-Methoden

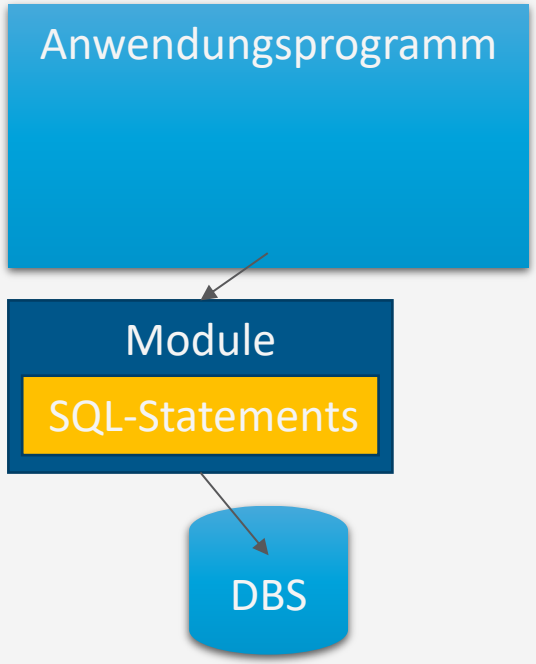


Direkter Aufruf
- Von SQL an die DB

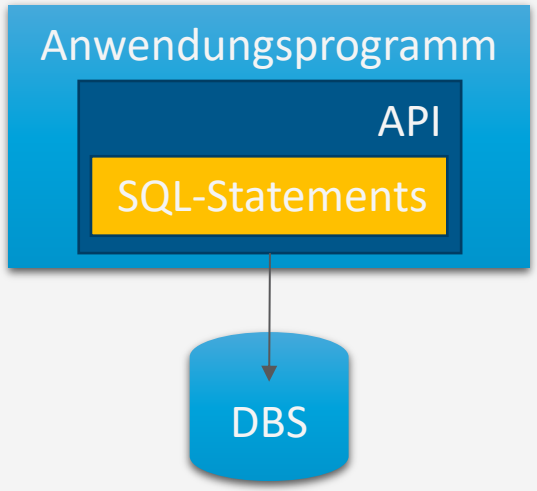


Embedded SQL
- SQL wird in Host-Sprache eingebunden

Dynamic SQL
- Zur Programmlaufzeit zusammengebaut



Module Language
- SQL wird in Module ausgelagert, die von Host-Sprache aus angefragt werden



Call-Level APIs
- Schnittstellen, um aus der Host-Sprache DBs anzusprechen, z.B. SQL/CLI, ODBC, JDBC

Mappings
- Programmierer sieht kein SQL mehr

Impedance Mismatch

- Problem: Datenzugriff unterscheidet sich zwischen SQL und anderen Programmiersprachen
  - SQL: mengenorientiert
  - Andere: verarbeiten einzelne Werte

→ sog. *impedance mismatch*

- Relationales Datenmodell wird von den meisten Programmiersprachen nicht unterstützt

Embedded SQL: "Shared" Variablen

- Transferieren Informationen zwischen Datenbank und Anwendungsprogramm
- Werden in einem DECLARE Abschnitt deklariert:
  - Inhalt des Abschnitts hängt von Programmiersprache ab
- Können im SQL-Statement statt einer Konstante verwendet werden
- Variablen-Name wird mit Doppelpunkt gekennzeichnet
- Spezielle Variable SQLSTATE enthält Fehlercodes
  - '00000': no error condition, '02000': no tuple found

```
exec sql begin declare section;
        { Declaration }
exec sql end declare section;
```

Teil der offiziellen SQL-Grammatik

Shared Variable mit INSERT

- Um Daten in die DB über bzw. aus der DB in die Variablen zu bekommen, EXEC SQL mit INSERT bzw. SELECT kombinieren

```
exec sql insert into TableName(AttrList)
values (VarListColonPrepended)
```

```
exec sql select into VarListColonPrepended
from TableName;
```

```
void setParts() {
   exec sql begin declare section;
      char part[4], project[4], version[10], description[50];
      char sqlstate[6];
   exec sql end declare section;

   /* request part, project, version, description */

   exec sql insert into parts(partno, version, projectno, part_description)
         values (:part, :version, :project, :description);
}
```

Beispiel: Single-Row SELECT Statements

```
int getNumProjects(int minBudget) {
   exec sql begin declare section;
      int num, budget;
      char sqlstate[6];
   exec sql end declare section;
   budget := minBudget;
   exec sql select count(*)
            into :num
            from projects
            where budget >= :budget;

   /* check that SQLSTATE has all 0's    */
   /* and if so print the value of num   */
   if sqlstate == '00000':
      return num;
   else
      return -1:
}
```

Cursor

- Konzept, um durch Ergebnismenge zu navigieren
- 4 Schritte, um einen Cursor zu nutzen:
  1. Cursor Deklaration: **`exec sql declare`** `<cursor>` **`cursor for`** `<query>`
     - <cursor> : Name des Cursor; <query> : SQL-Ausdruck
  2. Cursor Initialisierung: **`exec sql open`** `<cursor>`
     - Initialisiert den Cursor vor dem ersten Tupel, Anfrage wird ausgeführt
  3. Tupel holen: **`exec sql fetch from`** `<cursor>` **`into`** `<variables>`
     - Holt das nächste Tupel und schreibt es in die <variables>
     - kann mehrfach ausgeführt werden
     - wenn keine Tupel mehr da sind: SQLSTATE = '02000'.
  4. Cursor schließen: **`exec sql close`** `<cursor>`

Beispiel: Cursor

```
void getAllProjects() {
   exec sql begin declare section;
      char project[4], description[50];
      char SQLSTATE[6];
   exec sql end declare section;
   exec sql declare execCursor cursor for
      select projectno, description
      from projects;
   exec sql open execCursor;
   while (1) {
      exec sql fetch from execCursor
         into :project, :description;
      if (!(strcmp(SQLSTATE, "02000")) break;
      printf("projectno: %s, description: %s",
            project, description);
   }
   exec sql close execCursor;
}
```

Dynamic SQL

- Standard für Anwendungsprogramme, die SQL-Statements zur Laufzeit erstellen und absenden
- Es ist also der DB vorab unbekannt …
  - Ob ein Statement Daten holen oder speichern will
  - Wie viele Variablen benutzt werden, und welchen Typ sie haben
- Eigenschaften der SQL-Statements durch Deskriptor beschreibbar
- Zwei Möglichkeiten:
  - Execute Immediate:
    - Statement wird direkt ausgeführt
  - Prepare and Execute:
    - Das gleiche Statement wird mehrfach ausgeführt (mit verschiedenen Parametern)
    - Zwischenergebnisse der Vorbereitung werden behalten (z.B. Ausführungsplan)

(J. Melton, A. R. Simon: SQL:1999 Understanding Relational Language Components, 2002)

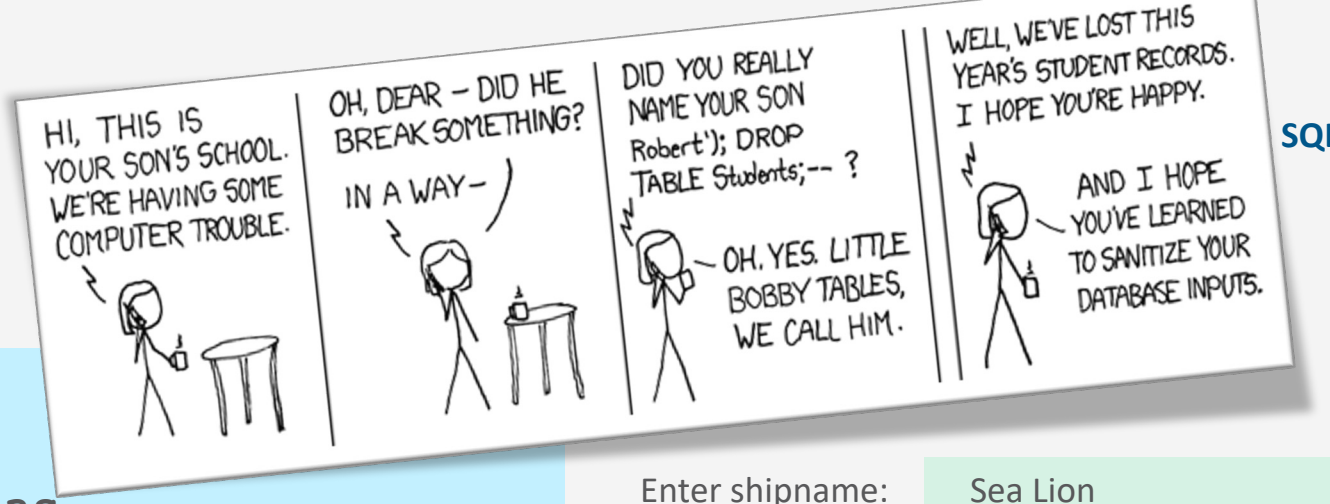
Dynamic SQL: Beispiel

```
/* execute statement only once */
exec sql execute immediate "UPDATE projects
                            SET budget = 10 000 000
                            WHERE projectno = 'PJ47'";

/* prepare and execute statement */
dynstmt = "DYN1";
temp = "UPDATE projects
        SET budget = 1 000 000
        WHERE projectno = ?";
exec sql prepare :dynstmt from :temp;

prjno = "PJ47";
exec sql execute :dynstmt using :prjno;
```

Nachteile der Dynamik: SQL Code Injection




```
create procedure search_orders
       @custid    nchar(5)   = null,
       @shipname nvarchar(40) = null as
declare @sql nvarchar(4000)

select @sql =' SELECT * ' +
            ' FROM dbo.Orders WHERE 1 = 1 '

if @custid is not null
       select @sql = @sql +
            ' AND CustomerID LIKE ''' + @custid + '''
if @shipname is not null
       select @sql = @sql +
            ' AND ShipName LIKE ''' + @shipname + '''
exec(@sql)
```

Enter shipname: Sea Lion

➔

```
exec(
select * from dbo.Orders
where 1 = 1 and ShipName like
'Sea Lion'
)
```

Enter shipname: `;drop table orders;

➔

```
exec(
select * from dbo.Orders
where 1 = 1 and ShipName like
'';
drop table ORDERS;
)
```

Module Language

- Anwendungsprogramm und SQL-Statements werden getrennt
  - Modul enthält Methoden und Deklarationen von Cursorn und temporären Tabellen, wird in einer Datenbank gespeichert
  - Anwendung kann die Methoden des Moduls aufrufen
  - Sog. Linker kombiniert SQL Statements und Anwendungsprogramm
- Beispiel rechts

```
module projects_module
   names are ascii     language C
   schema user_schema authorization user

procedure num_projects
   (  :budget     integer,
      :num        integer, sqlstate )
   select   count(*)
   into     :num
   from     projects
   where    budget >= :budget;
...
```


(J. Melton, A. R. Simon: SQL:1999 Understanding Relational Language Components, 2002)

Call-Level APIs

- ODBC (Open Database Connectivity)
  - Basiert auf informellem DBMS-Hersteller-Standard aus 1992
  - Microsoft adaptiert die Schnittstelle und nennt es ODBC
- SQL/CLI
  - Formales Konsortium (SQL Access Group) übernimmt die Entwicklung, nennt es CLI (Call-Level Interface)
  - Ergebnis wurde als Teil des SQL-92 Standards 1995 veröffentlicht, ist Teil 3 von SQL:1999
  - ODBC ist SQL/CLI sehr ähnlich
- JDBC (Java Database Connectivity)
  - Schnittstelle speziell für Java Anwendungen
  - JDBC wurde durch allgemeine APIs wie ODBC und SQL/CLI stark beeinflusst

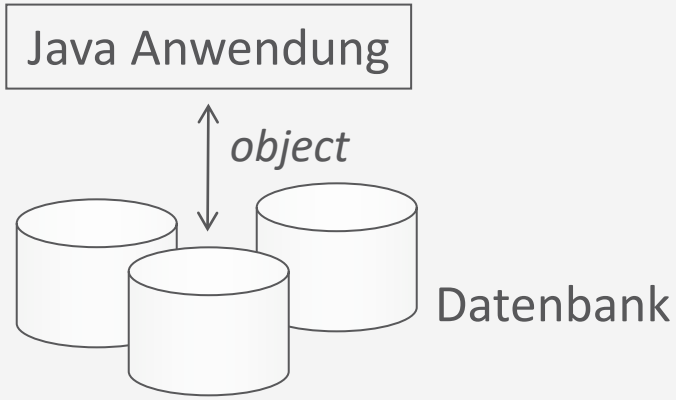
(J. Melton, A. R. Simon: SQL:1999 Understanding Relational Language Components, 2002)

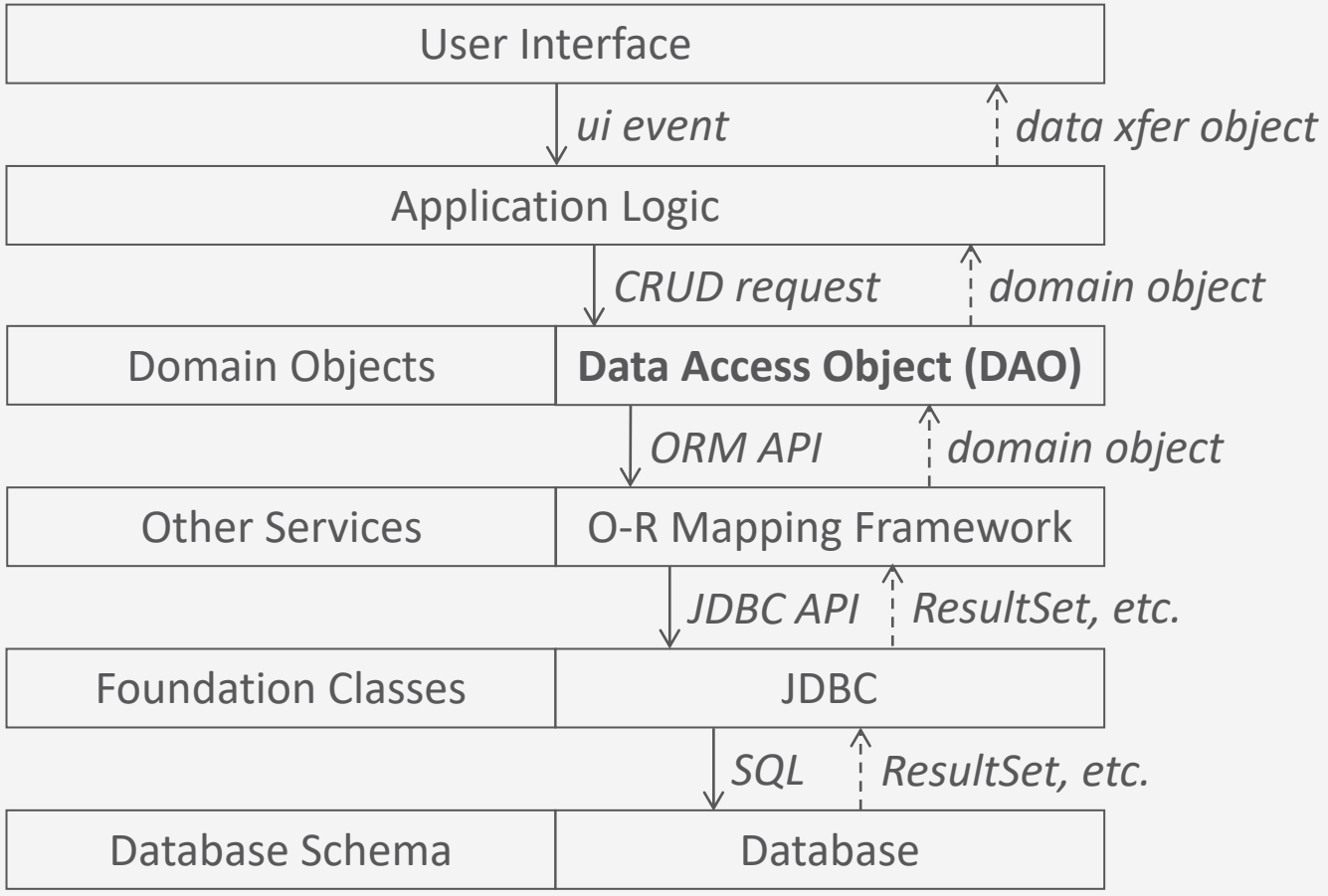
Beispiel: JDBC

```
class Employee {
   public static void main (String args []) throws SQLException {
      // Load the Oracle JDBC driver
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
      Connection conn = DriverManager.getConnection
         ("jdbc:oracle:thin:" + "@cip-s.kbs.uni-hannover.de:1521:dbs1",
               "scott", "tiger9i");
      Statement stmt = conn.createStatement();
      // Select the ENAME column from the EMP table
      ResultSet rset = stmt.executeQuery("select ENAME from EMP");
      // Iterate through the result and print the employee names
      while (rset.next())
         System.out.println(rset.getString(1));
      rset.close();
      stmt.close();
      conn.close();
   }
}
```

Object-relational Mappings (ORM)

- Ziel:
  - Persistierung in objektorientierter Anwendung
  - Framework verbirgt SQL vor Entwickler
  - Mapping: OO-Datenobjekte → Datenbankschemata und passende SQL-Statements





Zwischenzusammenfassung

- Views
  - Abgeleitete Relation
- Rechtevergabe
  - GRANT, REVOKE
- Programmiermethoden
  - Direkter Aufruf: Von SQL an die Datenbank
  - Embedded SQL: SQL wird in die Host-Sprache eingebunden
  - Dynamic SQL: Wird zur Programmlaufzeit zusammengebaut
  - Module Language: SQL wird in Module ausgelagert, die von Host-Sprache aus angefragt werden
  - Call-Level APIs: Schnittstellen, um aus der Host-Sprache Datenbanken anzusprechen
  - Mappings: Verbergen SQL vor Programmierer

Überblick: 5. Structured Query Language (SQL)

A. *Datendefinition (SQL als DDL)*
   - Schema, Tabellen, Datentypen, Constraints definieren
   - Strukturelle Änderungen mittels drop, alter

B. *Datenmanipulation (SQL als DML)*
   - Anfragen
   - Datenänderungen

C. *Und der Rest*
   - Sichten (SQL als VDL)
   - Rechtevergabe (SQL als DCL)
   - Programmiermethoden

→ Anfragenverarbeitung