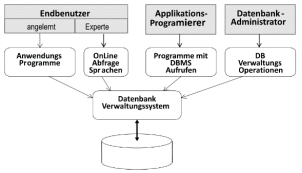
1. GRUNDLAGEN

Datenbanksystem (DBS): besteht aus einem Datenbankmanagementsystem (DBMS, bei einer Objektdatenbank ODBMS und bei einer relationalen Datenbank RDBMS genannt) und einer (oder mehreren) Datenbasen (DB).

Anforderungen an ein DBMS:
- ***Redundanzfreiheit*** *(Jedes Element nur einmal)*
- ***Datenintegrität:*** Datenkonsistenz *(logische Widerspruchsfreiheit)*, Datensicherheit *(physisch)*, Datenschutz *(vor unberechtigtem Zugriff)*
- ***Kapselung:*** Anwendungen greifen nicht direkt auf Daten zu, sondern via DBMS (Datenunabhängigkeit)



ANSI-3-Ebenen-Modell

Logische Ebene: Logische Struktur der Daten, Definition durch logisches Schema «Trägermodell» *(Zugriff auf die Daten durch DBMS von Speichermedium)*

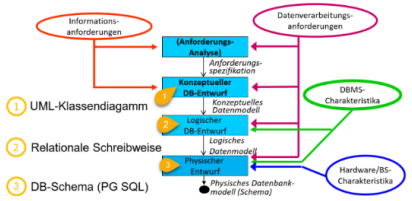
Interne Ebene: Speicherstrukturen, Definition durch internes Schema *(Beziehungen zwischen den Daten, Tabellen etc.)*

Externe Ebene: Sicht einer Benutzerklasse auf Teilmenge der DB, Definition durch externes Schema *(Daten, auf die der Benutzer zugreifen kann)*

Mapping: Zwischen den Ebenen ist eine mehr oder weniger komplexe Abbildung notwendig

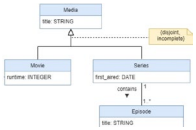
---

DB-Entwurfsprozess



2. UML - DATENMODELLIERUNG

Vererbung: «is-a». Kann ***complete*** oder ***incomplete*** *(Element muss nicht zwingend Instanz einer Subklasse sein, um Instanz der Superklasse zu sein)* und ***disjoint*** *(Objekt ist Instanz von genau einer Unterklasse)* oder ***overlapping*** *(Objekt ist Instanz von x überlappenden Unterklassen)* sein.



Multiplizität / Kardinalität:

i .. j	i bis j	≥ i und ≤ j
*1 .. **	1 oder mehrere	≥ 1
1 (oder 1 .. 1)	Genau 1	= 1
** (oder auch 0 .. *)*	0 oder mehrere	≥ 0
0 .. 1	0 bis 1	≥ 0 und ≤ 1

3. RELATIONALES MODELL

```
Student (
    Id INTEGER PK,
    lang TEXT(3) NOT NULL UNIQUE,
    abt INT REFERENCES TableB
)
```

Abbildung von Assoziationen inkl. Kompositionen:

2.1 | one-to-many | 1..* zu 1: Der Primärschlüssel der 1-Tabelle muss in der *-Tabelle als Fremdschlüssel vorkommen. *(1 oder mehr Studenten gehören zu einer Abteilung)*

```
Abteilung (AbtId INT, Name TEXT)
Student   (StudId INT, Name TEXT, Adresse TEXT,
           AbtID INT NOT NULL REFERENCES Abteilung)
```

2.2 | optionale Assoziation | 0..1 zu 0..*: Wie bei vorheriger, aber mit optionalen Beziehungsattributen. *(0 oder 1 Person leiht 0 oder mehrere Bücher aus)*

```
Person  (PId INT, Name NOT NULL)
Buch    (BuchId, Bezeichnung NOT NULL,
         Ausleihdatum NULL, Ausleiher NULL
         REFERENCES Person)
```

2.2 | optional mit separater Tabelle | 0..1 zu 0..*: Wird verwendet, falls Beziehungen selten sind, damit nicht zu viele NULL-Werte. *(0 oder 1 Person leiht 0 oder mehrere Bücher aus)*

```
Person   (PId INT, Name NOT NULL)
Buch     (BuchId, Bezeichnung NOT NULL)
Ausleihe (BuchId NOT NULL REFERENCES Buch, PId NOT
          NULL REFERENCES Person, Ausleihdatum
          NOT NULL)
```

2.3 | Kinder | 1 zu 0..*: Der PK der abhängigen Tabelle ist entweder ein FK oder enthält Attribute des FK der übergeordneten Tabelle. *(Ein Angestellter hat 0 oder mehr Kinder)*

```
Angestellter (AngId, Name NOT NULL)
-- V1: Abhängige Tabelle, starke Beziehung, Koposition
Kind     (AngId REFERENCES Angestellter,
          Vorname, GebJahr NOT NULL)
-- V2: Unabhängige Tabelle, schwache Beziehung, Aggr.
Kind     (KindId, AngId NOT NULL REFERENCES
          Angestellter, Vorname NOT NULL, GebJahr)
Constraint:  UNIQUE (Kind.AngId, Kind,Vorname)
```

2.4 | many-to-many | 0..* zu 0..* *(0 oder mehr Personen belegen 0 oder mehrere Kurse)*

```
Student   (StudId Name NOT NULL)
Kurs      (KursId, Bezeichnung NOT NULL UNIQUE)
Belegung  (StudId REFERENCES Student, KursId
           REFERENCES Kurs)
```

Abbildung von Vererbungen/Generalisierungen

3.a | Je eine Tabelle pro Sub- und Superklasse: Flexibel, redundanzfrei, geeignet für overlapping Vererbung. Jedoch viele Tabellen, komplexe Zugriffe, zusätzliches Typ-Attribut benötigt für einfache Unterscheidung v.a. bei overlapping Vererbungen.

```
Fahrzeug  (FzgId INT, Marke STRING, Gewicht
           DECIMAL, FzgTyp INT NOT NULL)
PKW       (FzgId REFERENCES Fahrzeug, AnzPlaetze
           INT NOT NULL)
LKW       (FzgId REFERENCES Fahrzeug, LadeFlaeche
           DECIMAL NOT NULL)
```

3.b | Eine Tabelle pro Subklasse: Keine direkte Abbildung der Superklasse. Einfache Zugriffe auf die Tabellen. Jedoch Semantikverlust, da nicht mehr klar ist, was gemeinsame Attribute sind, überlappende Vererbungen können nicht abgebildet werden. Primary-Key-Eindeutigkeit muss über mehrere Tabellen kontrolliert werden.

```
PKW       (FzgId INT, Marke STRING, Gewicht
           DECIMAL, AnzPlaetze INT NOT NULL)
LKW       (FzgId INT, Marke STRING, Gewicht
           DECIMAL, LadeFlaeche DECIMAL NOT NULL)
```

3.c | Eine einzige Tabelle für die Superklasse: Subklassen werden nicht explizit abgebildet. Die Tabelle speichert alle Attribute, auch die der Subklassen. Zusätzlich enthält sie ein diskriminierendes Attribut, das den jeweiligen Typ der Subklasse spezifiziert. Einfache Zugriffe, funktioniert auch für overlapping Vererbungen. Aber viele Null-Werte, dritte oder höhere Normalform verletzt

```
Fahrzeug  (FzgId INT, Marke STRING, Gewicht
           DECIMAL, FzgTyp INT NOT NULL, AnzPlaetze
           INT NULL, LadeFlaeche DECIMAL NULL)
```

π Projektion (pi: projection): Relation × Attributfolge -> Relation
σ Selektion (sigma: selection): Relation × Bedingung -> Relation
x Kartesisches Produkt: Relation × Relation -> Relation
⋈ Verbund (join): Relation × Relation -> Relation
ρ Umbenennung (rho: replacement); formal: ρV (E1), ρA ← B (E1)

Relationale Algebra

Begriffe

Wertebereich	D1 = {Grün, Rot, Blau}, D2 = {-69 .. +1337}
Attribut	A1 = Blau *(Wert aus D1)*, D2 = 420 *(Wert aus D2)*
Tupel	T = (Blau, 420) *Record aus zusammengehörigen Attributen*
Relation	Mathematische Menge, vergleichbar mit Entitätsmenge. Eine Tabelle ist eine Visualisierung des Relationenmodells.

Vereinigung, Differenz, «Ausser»
- ***Union/Vereinigung*** ∪: Fügt zwei Tabellen zusammen, UNION ALL entfernt keine Duplikate, Kombination

```
select * from R UNION select * from S
```

- ***Intersect / Durchschnitt*** ∩: Durchschnitt von zwei Tabellen, nur behalten was in beiden vorkommt

```
select * from R INTERSECT select * from S
```

- ***Except / Differenz*** −: Differenz zwischen zwei Tabellen, behalte nur das von der linken Tabellen was nicht in der rechten vorkommt

```
select * from R EXCEPT select * from S
```

Normalisierung

Stellt ***Redundanzfreiheit*** sicher, ***verhindert Anomalien***. Ist ***Verlustlos*** *(alle Infos bleiben nach Zerlegung erhalten)* und ***Abhängigkeitsbewahrend*** *(Funktionale Abhängigkeiten bleiben bewahrt)*

1. Normalform: Attributwerte sind atomar *(z.B. «Hans» und «Muster» statt «Hans Muster»)*, zusätzliche Zeile oder Spalte
2. Normalform: Nichtschlüsselattribute sind von jedem Schlüsselkandidaten voll funktional abhängig *(keine Abhängigkeiten von einem nur einem Teilschlüssel)*. Attribute, die von Teilschlüssel abhängen, zu sep. Tabelle zusammenfassen.
3. Normalform: Keine Abhängigkeit zwischen Nichtschlüssel-attributen. Abhängige Attribute kommen in eigene Tabelle.
Boyce-Codd-Normalform: Nur Abhängigkeiten vom Schlüssel. Jede Determinante ist ein Schlüsselkandidat. *Beispiel:* `Sportler(Name, Verein, Sportart)`, *Sportart hängt von Verein ab, zerlegen in* `Sportler(Name, Verein)` *und* `Verein(Name, Sportart)`
Voll-funktionale Abhängigkeit: Ein Attribut B ist voll funktional abhängig von Attribut A, falls zu jedem A genau ein Wert von B existiert. *Beispiel: ISBN legt eindeutig Autor und Titel fest*
Transitive Abhängigkeit: $A \rightarrow B \cap B \rightarrow C \Rightarrow A \rightarrow C$
Mutations-Anomalien: Unbeabsichtigte Veränderung von Datensätzen wie ***Einfügeanomalie*** *(Eine neue Abteilung kann erst eingefügt werden, wenn mind. 1 Mitarbeiter & Chef bekannt)*, ***Löschanomalie*** *(Löschen aller Mitarbeiter einer Abteilung löscht alle Informationen einer Abteilung)*, ***Änderungsanomalie*** *(Ändern des Abt. name bewirkt dass alle Mitarbeiter auch geändert werden müssen)*.
Denormalisierung: Bei Tabellen in NF können Performanzprobleme bei Anfragen über mehrere Tabellen entstehen, komplizierter Zugriff. Deshalb Denormalisierung in die 2. NF.

4. POSTGRESQL

`\c <db>`: Connect to db, `\l`: Alle Datenbanken anzeigen, `\d`: Alle Tabellen anzeigen oder Tabellenschema, `\q`: psql-Shell verlassen

---

DDL – Data Definition Language

Schema: Menge von DB-Objekten *(Tabellen, Views, Berechtigungen...)*, die eine Datenbank in Namensräume unterteilen kann. Hat eine DB-weit eindeutige Schema-ID, per default «public»
Domain: Wertebereich
View: Sicht auf eine oder mehrere Tabellen
Index: Hilfsdatenstruktur für beschleunigte Zugriffe
Owner: Erzeuger der DB-Objekte. Datenbankbenutzer, ausgestattet mit Privilegien um Objekte zu erstellen/löschen.

Funktion für SCHEMA, INDEX, TABLE, VIEW, DATABASE:

Erzeugen	*Ändern*	*Löschen*	*Lösche Inhalt*
CREATE …	ALTER …	DROP …	TRUNCATE …

```
LOGIN UNENCRYPTED PASSWORT 'test'
SUPERUSER NOINHERIT CREATEDB; -- Create role
CREATE DATABASE fahrzeugverwaltung;
CREATE ROLE clients; -- Create group
GRANT SELECT ON ALL TABLES IN SCHEMA public TO clients;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT
ON TABLES TO clients; -- Grant Statement
REVOKE ALL ON SCHEMA public FROM public;
CREATE ROLE webclient WITH LOGIN PASSWORD 'xy' IN ROLE
clients;
CREATE SCHEMA userid [
  <create-table-statement-sequence>
  <create-view-statement-sequence>
  <grant-statement-sequence>
];
CREATE TABLE fahrzeug (
  id INTEGER PRIMARY KEY,
  fzg_typ INTEGER NOT NULL );
CREATE TABLE pwk (
  fahrzeug PRIMARY KEY REFERENCES fahrzeug(id) );
CREATE TABLE mahlzeit (
  menue INTEGER NOT NULL REFERENCES menue(id),
  bstlg INTEGER NOT NULL REFERENCES bestellung(id),
  PRIMARY KEY (menue, bstlg) );
CREATE INDEX XNameAng ON Angestellter (Name);
ALTER TABLE angestellter
  ADD CONSTRAINT name CHECK (alter < 66);
DROP TABLE angestellter -- Tabelle löschen
TRUNCATE TABLE angestellter -- Nur Inhalt löschen
```

Löschen eines Tupels einstellen (default: RESTRICT):

```
ON DELETE CASCADE -- Alle Sub-Tupel werden mitgelöscht
ON DELETE RESTRICT --Supertupel wird nicht gelöscht
ON DELETE SET NULL -- Sub-Tupel werden NULL
ON DELETE SET DEFAULT -- Sub-Tupel werden DEFAULT
ON UPDATE -- Wie DELETE aber bei Änderung Super-Tupel
```

Konsistenzbedingungen / Column Constraints
Definieren Einschränkungen pro Attributwert.
- `CHECK()` Schränkt Wertebereich eines Attributs anhand Bedingung ein.
- `NOT NULL` Attribut muss immer einen Wert haben
- `UNIQUE` Attributwert muss eindeutig sein (erstellt Index)
- `PRIMARY KEY` Primärschlüssel, immer UNIQUE und NOT NULL (erstellt Index)
- `REFERENCES <TableName>` Bedingung für Fremdschlüssel-Beziehung (Tabelle muss Primärschlüssel haben)

Table Constraints
Definieren ***Einschränkungen für 1 oder mehrere Attribute*** einer Tabelle. Entweder via `ALTER TABLE ADD CONSTRAINT <customName>` oder in `CREATE TABLE` nach allen Attributen. Alle Column Constraints können auch als Table Constraints definiert werden.

Datentypen

Boolean: Boolescher Datentyp
Ganzzahlen: `SMALLINT` 2 Byte, `INT` oder `INTEGER` 4 Byte, `BIGINT` 8 Byte
Gleit-/Fliesskommazahlen: `REAL` oder `FLOAT` sind Fliesskomma-Zahlen, `DOUBLE` ist Fliesskomma-Zahl mit 8 Byte, `NUMERIC[(precision,scale)]` und `DECIMAL(precision,scale)` sind Festkomma-Zahlen.
Zeichenketten: `CHAR` oder `CHARACTER(size)` sind Strings mit fixer Länge (<= 2000 Zeichen), `VARCHAR(size)` sind Strings mit variabler Länge
Datum/Zeit: `DATE` ist Jahrhundert, Jahr, Monat, Tag; `DATETIME` ist Date + Time; `TIME` ist Stunde, Minute, Sekunde; `INTERVAL` ist Zeitintervall
Array: `type[]` z.B. `integer[][]` für zweidimensionales Int-Array
Verschiedenes: `BINARY`, `VARBINARY` und `LONGVARBINARY` sind binäre Datentypen, `CLOB`/`BLOB` (Char/Binary Large Objects) für Speicherung von grossen Text- und Binärdaten.
PostgreSQL: `FLOAT` gibt es nicht, `TEXT` für Zeichenketten, `BINARY`, `VARBINARY` und `LONGVARBINARY` sind bytea, `SERIAL` als Alternative zu `CREATE SEQUENCE` (Auto-increment), kein RAW, CLOB/BLOB, DATETIME
Type Casting: `SELECT CAST(42 AS float8)` oder `SELECT 42::float8`
Datum/Zeit: `now()` oder `CURRENT_DATE`
Text: `UPPER()`, `LOWER()`, `SUBSTR()` etc. Wildcard: _ für genau 1 Zeichen, % für mehrere Zeichen
Numerische Funktionen: `SUM()`, `COUNT()`, `ROUND()`, `MOD()`, `TRUNC()`, `ABS()`, `COS()`, `POWER()`, …
Helper-Funktionen: `COALESCE`: Gibt 0 statt NULL zurück

DML – Data Manipulation Language

`FROM + JOIN -> WHERE -> GROUP BY -> HAVING -> SELECT (WINDOW FUNCTIONS) -> ORDER BY -> LIMIT`

Einfügen: Fügt einen neuen Datensatz in eine bestehende Tabelle ein. `INSERT INTO <tableName> ([columnName]) VALUES (<value>)` *copy ist schneller als insert into*
Abfragen: Liefert eine Menge von Datensätzen aus einer oder mehreren Tabellen aufgrund von Abfragekriterien. `SELECT <columnName> FROM <tableName>`
Modifizieren: Ändert Daten von bestehenden Datensätzen einer Tabelle. WHERE nicht vergessen, sonst werden alle Rows geändert `UPDATE <tableName> SET <columnName>=<value> WHERE …`
Löschen: Löscht bestehende Datensätze einer Tabelle. `DELETE FROM <tableName> WHERE…`
Group by: Teilt Resultattabelle in Gruppen auf. NULL → eigene Gruppe
Having: Kann nur nach `GROUP BY`-Klausel stehen. Erlaubt Auswahl von Zeilen, die durch Anwendung der `GROUP BY`-Bedingung entstehen. Damit ***lassen sich gewisse Sachen filtern, die sich mit WHERE nicht filtern lassen***. Attribut/Funktion muss in SELECT vorkommen.
Distinct: Gibt nur distinct Werte in der nachfolgenden Spalte an.
Aggregats-/Gruppenfunktionen: *MAX()*, *MIN()*, AVG(), SUM(), COUNT() Liefern nur eine Zeile als Resultat. In SELECT müssen alle Zeilen von einer Gruppenfunktion abhängen. NULL wird ignoriert.

```
INSERT INTO abteilung VALUES (23, 'Verkauf');
UPDATE abteilung SET name='Verkauf' WHERE abtnr=3;
DELETE FROM abteilung WHERE abtnr=21;
SELECT name, salaer, wohnort FROM angestellter WHERE
abtnr=1 AND (salaer>1000 OR wohnort='Luzern') ORDER BY
name, wohnort, salaer;
SELECT DISTINCT wohnort FROM angestellter WHERE wohnort
LIKE 'Zü%'; LIKE '____' -- wäre alles 4-stellige
SELECT MAX(salaer) FROM angestellter;
SELECT abtnr, round(sum(salaer)) as sum_salaer FROM
angestellter GROUP BY abtnr ODER BY abtnr;
SELECT wohnort, COUNT(wohnort) FROM angestellter;
-- Error: column xy must appear in the GROUP BY clause
or be used in an aggregate function. Fix:
SELECT MIN(wohnort), COUNT(wohnort) FROM angestellter;
SELECT abtnr, COUNT(*) AS 'angestellte' FROM
angestellter GROUP BY abtnr HAVING COUNT(*) >= 5;
```

JOINS

Inner Join
Default wenn nichts anderes definiert. Kombiniert Zeilen aus zwei oder mehr Tabellen auf Grundlage einer Bezugsspalte.

```
SELECT abt.name AS abtname, ang.name AS angname FROM Ab-
teilung AS abt INNER JOIN angestellter AS ang ON
abt.abtnr = ang.abtnr ORDER BY abt.name, ang.name;
```

Equi Join / Self Join
Kombiniert Zeilen aus derselben Tabelle auf der Grundlage einer verwandten Spalte.

```
SELECT ang1.name AS "Vorgesetzter", ang2.name AS "Mitar-
beiter" FROM angestellter ang1 INNER JOIN angestellter
ang2 ON ang1.persnr=ang2.chef WHERE ang1.chef IS NULL;
```

Natural Join
Joint basierend auf gleichem Spaltenname

```
SELECT * FROM R natural join S
```

Unterschied Equi-Join und Natural Join: In der neuen Tabelle kommt die verglichene Spalte im Natural Join nur einmal vor, beim Equi-Join werden alle verglichenen Spalten aufgeführt.

Semi Join
Joint **nur**, wenn Tupel existiert

```
WHERE id IN (SELECT id FROM table) WHERE EXISTS (SELECT
1 FROM table WHERE a.id = table.id)
```

Anti Join
Zeigt Rows, die nur in der linken, aber nicht in der rechten Tabelle sind

```
LEFT JOIN table ON R.id = a.id WHERE R.id IS NULLWHERE
id NOT IN (SELECT id FROM R)
```

Join über 3 Tabellen (new style)

```
SELECT a.name, p.bezeichnung, pz.zeitanteil FROM pro-
jektzuteilung AS pz JOIN projekt AS p ON p.projleiter =
pz.persnr JOIN angestellter AS a ON a.persnr = pz.persnr
WHERE pz.zeitanteil > 30;
```

Left outer Join
Ruft alle Zeilen aus der linken Tabelle und die übereinstimmenden Zeilen aus der rechten Tabelle ab. Nicht übereinstimmende Zeilen in der rechten Tabelle haben NULL-Werte.

```
SELECT a.name, p.bezeichnung FROM projekt p LEFT OUTER
JOIN angestellter a ON p.projleiter = a.persnr;
```

Right outer Join
Ruft alle Zeilen aus der rechten Tabelle und die übereinstimmenden Zeilen aus der linken Tabelle ab. Nicht übereinstimmende Zeilen in der linken Tabelle haben NULL-Werte.

```
SELECT a1.name AS chef, a2.name AS untergebener FROM an-
gestellter a1 RIGHT OUTER JOIN angestellter a2 ON
a1.persnr = a2.chef;
```

Full outer Join
Alle Zeilen abrufen, wenn es eine Übereinstimmung entweder in der linken oder der rechten Tabelle gibt. Nicht übereinstimmende Zeilen in beiden Tabellen haben NULL-Werte.

```
SELECT a1.name AS chef, a2.name AS untergebener FROM an-
gestellter a1 FULL JOIN angestellter a2 ON a1.persnr =
a2.chef WHERE a1.persnr IS NULL OR a2.chef IS NULL;
```

Lateral Join
Wird benutzt um Tabellen/Funktionen von inneren Queries wieder zu verwenden. Bei kleinen DBs schneller als Inner Join

```
SELECT avg_aufw_mitarb, avg_aufwand / dauer AS auf-
wand_pro_tag_pro_mitarb FROM projekt, LATERAL (SELECT
aufwand / anz_mitarb AS avg_aufw_mitarb) AVG
```

Unterabfragen

Um weitere Abfragen in einem `WHERE` zu machen. `ORDER BY/UNION` sind nicht erlaubt. Eher CTE-Queries verwenden, da mächtiger. Performance ist bei beiden Arten gleich.
Korrelierend: Abhängig von der Elternabfrage
```
SELECT c1, c2 FROM table1 WHERE c1 = (SELECT x FROM table2 WHERE y = table1.c2);
```
Unkorrelierend: Unabhängig von der Elternabfrage, könnte separat stehen und funktionieren. `SELECT c1, c2 FROM table1 WHERE c1 = (SELECT MAX(x) FROM table2);`

- `IN:` Mehrere Tupel. gelieferte Liste enthält Element
- `EXISTS:` Mehrere Tupel, gelieferte Tabelle nicht 0
- `ANY:` Mehrere Tupel, mindestens ein Wert aus Liste
- `ALL:` Mehrere Tupel, alle Werte aus der Liste

Window Functions

Funktionen oder Kalkulationen, die auf ein ***«Daten-Fenster»*** angewendet werden. ***Mächtiger*** als `GROUP BY`, ***Tupel*** bleiben im Resultset ***erhalten***. Neue ***Möglichkeiten*** wie Analyse von Logs oder Zeiterfassung. Pro `OVER(PARTITION BY …)`-Spalte gibt es eine neue Tabelle, ***Reihenfolge Ausgabe*** ist abhängig von `ORDER BY`.

```
SELECT id, …, window_fn_name([attr]) OVER (
  PARTITION BY attr_name ORDER BY attr_name
) AS alias FROM table_name;
```

```
SELECT abtnr, persnr, salaer, RANK() OVER (PARTITION BY abtnr ORDER BY salaer DESC FROM angestellter);
```

```
SELECT persnr, name, salaer, salaer - lead(salaer, 1, salaer) OVER (ORDER BY salaer)
FROM angestellter ORDER BY salaer DESC;
```

```
SELECT salesperson, sale_date, sales, RANK() OVER (PARTITION BY salesperson ORDER BY sales DESC) AS sales_rank
FROM sales WHERE RANK() OVER (PARTITION BY salesperson ORDER BY sales DESC) <= 3;
-- Assign a rank to each row within sales, then filter to include only rank of 1,2 or 3
```

Window-Functions:
`MIN()`, `MAX()`, `AVG()`, `SUM()`, `COUNT()`, `RANK()` *Rangfolge*, `LAG(attr, #offset, defaultValue)` *Offset davor*, `LEAD()` *Offset danach*, `NTILE()` *Gleichmässige Aufteilung von Zeilen*, `ROW_NUMBER()` *Nummerierung*

Common Table expression CTE

Hilfs-Query in einer `WITH`-Klausel (Temporäre Tabellen während des Statements. Query-Name immer im `FROM`. Können:

- `SELECT`, `INSERT`, `UPDATE`, `DELETE` enthalten
- Sich auf vorgehende Hilfs-Queries beziehen
- Anstelle von Subqueries verwendet werden
- Dem DB-Optimierer helfen
- Rekursiv sein

```
WITH queryName AS ( SELECT * FROM myTable )
SELECT * FROM queryName; -- normal, ohne recursion
```

```
WITH tmptable(name, bezeichnung, zeitanteil) AS (
  SELECT name, bezeichnung, zeitanteil
  FROM angestellter a
  JOIN projektzuteilung pz ON pz.persnr=a.persnr
  JOIN projekt p ON p.projnr=pz.projnr
)
SELECT name AS "Mitarb.", bezeichnung AS "Projekt", zeitanteil AS "Zeit" FROM tmptable; --normal ohne recursion
```

```
-- mit recursion
WITH RECURSIVE untergebene(persnr, name, chef) AS (
  SELECT A.persnr, A.name, A.chef FROM angestellter A
  WHERE A.chef = 1010 UNION ALL -- recursive term
    SELECT A.persnr, A.name, A.chef FROM angestellter A
  INNER JOIN untergebene B ON B.persnr = A.chef
)
SELECT * FROM untergebene ORDER BY chef, persnr;
```

Views

Eine View ist eine ***virtuelle Tabelle*** basierend auf anderen Tabellen oder Views. Daten werden ***zur Ausführzeit*** aus Tabellendaten hergeleitet. Man kann auch Queries damit ***vereinfachen***. Spaltennamen können auch anders heissen wie in Originaltabelle. ***Sicherheit:*** Irrelevante Daten für bestimmte Nutzer entfernen. `CREATE VIEW AngPublic AS SELECT * FROM Angestellter; -- Korrektheit wird überprüft`

Updatable View: wenn weder `JOIN`, SET-Operationen noch `GROUP`-Funktionen enthalten sind: `GROUP BY`, `CONNECT BY`, `START WITH`, `DISTINCT`, `UNION`, `INTERSECT`. Spalten dürfen keine Funktionen sein.

Materialized View: Resultat von Views wird gecached, nicht automatisch aktualisiert. `CREATE MATERIALIZED VIEW name …; REFRESH MATERIALIZED VIEW name …;`
Row-Level Security (RLS): Eine Art «System-Views». Nur User mit entsprechendem Lese- und Schreibrecht («Policy»)
Temporäre Tabellen: Werden am Ende einer Session oder Transaction gelöscht. Andere «permanente» Tabellen mit gleichem Namen sind nicht sichtbar. `CREATE TEMPORARY TABLE;`

Null-Werte

Null sind ***missing data***. Entweder drin lassen, herausfiltern oder mit `COALESCE(spaltenname, neuerWert)` einen Wert zuweisen. Vergleiche mit NULL-Werten ergeben UNKNOWN. Sie werden in Aggregationen nicht gezählt.

```
SELECT NULL IS NULL -- true
SELECT NULL = NULL -- [null]
```

Sicherheit (DCL - Data Control Language)

Typische Sicherheitslücken: Standard-Passwörter, SQL-Injection, Cross-Site-Scripting.
System-Sicherheit: Authentisierung, Privilegien, Kontrolle von System-Ressourcen, Auditing, Transportsicherheit
Daten-Sicherheit: Zugriffskontrolle, Auditing von Zugriffsoperationen, DCL ist nicht standardisiert.
Massnahmen gegen Exploits: User-Input eingrenzen (Typ prüfen), Escape von nicht numerischem Input.
Datenzugriff abstrahieren: Stored Procedures, eigene User für bestimmte Operationen verwenden
Benutzerverwaltung: Jedem User sind Rechte zugeordnet für Datenbankoperationen und Verwaltung von DB-Objekten
Schema: Fasst DB-Objekte in einer DB zusammen. Eine Datenbank kann n-Schemas haben. Default: public
Rolle (ROLE): Oberbegriff für User oder Gruppen. Gelten für den ganzen Cluster (über alle DBs). Rolle kann n-Schemas besitzen.
Benutzer: `ROLE` mit `LOGIN` oder `USER` (optional `IN ROLE`)
```
CREATE ROLE user WITH LOGIN PASSWORD 'xyz';
CREATE USER user IN ROLE group;
```
Gruppe: `ROLE` ohne `LOGIN` (`CREATE ROLE group;`)
Systemprivilegien: erlauben Zugriff auf DB-Operationen `CREATEDB`, `CREATEROLE`: `CREATE ROLE user CREATEDB NOCREATEROLE; ALTER ROLE user WITH CREATEROLE;`
Datenprivilegien: Erlauben Zugriff auf Datenobjekte, Grantor gewährt Privilegien, Grantee erhält Privilegien `GRANT priv ON object TO {user|group|PUBLIC} [WITH GRANT OPTION]; REVOKE priv FROM {user|group|PUBLIC} priv:[SELECT|INSERT|UPDATE|REFERENCES|TRIGGER|DELETE|CREATE|ALL], object: [SCHEMA|TABLE|DATABASE].`
Gruppe: Globale Objekte, nicht in einem Schema. Haben nur Objektprivilegien. `GRANT Gruppe TO user;`
Rechte sollten wenn möglich nur Gruppen gegeben werden

Transaktionen

Pro Session max 1 Transaktion. `NESTED` Transaktionen sind nicht unterstützt. Nutzen:

- ***Fault Tolerance:*** Bei Server-Crash kann Operation wiederholt werden oder wird ganz gecancelt (nicht nur Hälfte durchgeführt)
- ***Concurrency:*** Isolation der Transaktionen, Parallelität wird ermöglicht.
- **A** ***Atomicity:*** Vollständig oder gar nicht
- **C** ***Consistency:*** Konsistenter Zustand bleibt erhalten
- **I** ***Isolation:*** Transaktion soll von anderen isoliert sein
- **D** ***Durability:*** Alle Änderungen sind persistent

```
BEGIN [TRANSACTION]; -- Kurznotation BEGINN;
COMMIT [TRANSACTION]; -- Kurznotation COMMIT;
ROLLBACK [TRANSACTION]; -- Kurznotation ROLLBACK;
SAVEPOINT xy;
ROLLBACK TO xy;
RELEASE xy; -- Safepoint löschen
COMMIT;
```

Commit Resultate: Success *(Änderungen atomar und durable gespeichert)* oder Failure *(Alle temporären Änderungen werden abgebrochen)*

Gründe für Abort: Explizit durch `ROLLBACK` oder `ABORT`, unzulässige Verzahnung mit anderen nebenläufigen Transaktionen, Deadlock, Applikationsabbruch, Systemabsturz, Fehler.

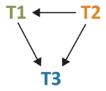
Serialisierbarkeit

Wenn ***parallele*** Ausführung gleich wie ***serielle*** Ausführung. Muss ***azyklisch*** sein (Keine Schlaufen)

Konfliktpaare: Verbindung ziehen zwischen r und w vom gleichen Buchstaben ziehen. Falls Überschneidung zwischen gleichem Buchstaben in $T1$ und $T2$, ist es ***nicht serialisierbar***.

Beispiel: ***(r=read, w=write, c=commit, T=Transaktion, S=Serialisierung)***
- $T1 = r1(x)\ w1(x)\ r1(y)\ w1(y)\ c1$
- $T2 = r2(x)\ w2(x)\ r2(y)\ w2(y)\ c2$
- $S = r1(x)\ w1(x)\ r1(y)\ w1(y)\ c1\ r2(x)\ w2(x)\ r2(y)\ w2(y)\ c2$ **OK**
- $S = r1(x)\ r2(x)\ w1(x)\ w2(x)\ r2(y)\ r1(y)\ w1(y)\ w2(y)\ c2\ c1$ **NOK**
- $S = r1(x)\ w1(x)\ r2(x)\ r1(y)\ w2(x)\ w1(y)\ c1\ r2(y)\ w2(y)\ c2$ **OK**
- $S = r2(x)\ r1(x)\ w1(x)\ r3(x)\ w3(x)\ c3\ w2(y)\ w1(y)\ c1\ c2$ **OK**
- Serialisierbar: $(r \rightarrow w \mid r \rightarrow r \rightarrow w \rightarrow w \mid r1 \rightarrow r1 \rightarrow w1 \rightarrow w2)$
- Nicht serialisierbar: $(r1 \rightarrow r2 \rightarrow w1 \rightarrow w2 \mid r1 \rightarrow r2 \rightarrow w1 \rightarrow w2)$

Konfliktpaare im untersten Beispiel:
- $r2(x) \leftarrow w1(x)$
- $r2(x) \leftarrow w3(x)$
- $r1(x) \leftarrow w1(x)$
- $r1(x) \leftarrow w3(x)$
- $w1(x) \leftarrow w3(x)$
- $w2(y) \leftarrow w1(y)$



Implementation der Isolation
- ***Pessimistische Verfahren:*** Benutzte Daten werden gesperrt, besser bei hoher Konflikt-Wahrscheinlichkeit
- ***Optimistische Verfahren:*** Konfliktbehebung im Nachhinein *(Rollback)*, besser bei kleiner Konfliktwahrscheinlichkeit

Locking
Locks sind dafür da, dass nur eine Transaktion eine bestimmte Zeile/Tabelle ändern oder zugreifen kann. Somit bleibt die ***Datenintegrität*** gewahrt. Garantiert aber keine Serialisierbarkeit bei zu frühem einsetzen von `unlock()`

- ***Exklusive Lock (X):*** Für Schreib- oder Lesezugriffe, nur eine Transaktion `xlock(x)`
- ***Shared Lock (S):*** Nur für Lesezugriffe, mehrere Transaktionen `slock(x)`
- ***Freigabe:*** `unlock(x)` gibt Lock wieder frei.

	S	X
S	✓	✗
X	✗	✗

Lock-Verträglichkeit

Two Phase Locking (2PL)
Garantiert Serialisierbarkeit. ***Nachteile:*** Deadlocks und Cascading Rollbacks sind möglich, unklarer Beginn der Shrinking Phase.

- ***Phase 1 (Growing Phase):*** Objekte werden gesperrt
- ***Phase 2 (Shrinking Phase):*** Nach erstem `unlock` kein `lock` mehr

Strict Two Phase Locking (S2PL)
Alle Sperren nach Ende der Transaktion freigeben. ***Vorteile:*** Kein Cascading Rollback, kein unklarer Beginn der Shrinking Phase. ***Nachteile:*** Deadlocks sind möglich, Parallelität wird unnötig eingeschränkt.

Preclaiming Two-Phase Locking
Alle Locks am Anfang & am Ende der Transaktion gleichzeitig sperren bzw. freigeben. ***Vorteil:*** Keine Deadlocks. ***Nachteil:*** Transaktion muss im Vorherein wissen, welche Sperren nötig sind - unrealistisch

Deadlock Szenario
Gegenseitige Locks. Wird entweder durch ***Timeout*** *(poor man solution)* oder durch die ***Erkennung*** *(Abbruch einzelner Transaktionen)* abgebrochen. Für Analyse dient Betriebsmittelgraph und Wartegraph.

Deadlock-Auflösung: Scheduler erkennt Zyklus im Wartegraph. Strategien zur Auswahl des Zyklus:

- ***Min. des Rücksetzaufwands***, T mit wenigsten Locks
- ***Max. frei gewordener Ressourcen***: T mit meisten Locks
- ***Vermeidung Starvation***: Nicht immer gleiche T rollbacken
- ***Mehrfache Zyklen:*** T die an meisten Zykeln beteiligt ist
- ***Cascading Rollbacks:*** Implizite Rollbacks von weiteren T

Isolation

Isolation Levels: Serializable ist am Besten. Parallelität limitiert, Effizienz mit schwächeren Levels gesteigert, auf Kosten der Korrektheit. Fehler schwer nachvollziehbar. ***Für T:*** `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;` ***Für Session:*** `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE;`

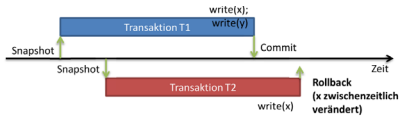
4 Levels nach ANSI SQL-92 Standard
- `READ UNCOMMITTED:` Lesezugriffe nicht synchronisiert (keine Read-lock), Read ignoriert jegliche Sperren
- `READ COMMITTED:` Lesezugriffe nur kurz/temporär synchronisiert (default), setzt für gesamte T Write-Lock, Read-lock nur kurzfristig
- `REPEATABLE READ:` Einzelne Zugriffe ROWS sind synchronisiert, Read und Write Lock für die gesamte T
- `SERIALIZABLE:` Vollständige Isolation nach ACID

	Read Uncommited	Read Commited	Repeatable Read	Serializable
Dirty Write	möglich*	möglich*	möglich*	unmöglich
Dirty Read	möglich	unmöglich	unmöglich	unmöglich
Lost Update	möglich	möglich	unmöglich	unmöglich
Fuzzy Read	möglich	möglich	unmöglich	unmöglich
Phantom Read	möglich	möglich	möglich	unmöglich
Read Skew	möglich	möglich	unmöglich	unmöglich
Write Skew	möglich	möglich	möglich	möglich*
Deadlock			möglich	unmöglich
Cascadin Rollback				unmöglich

* Möglich aber nur in SQL92, PostgreSQL Version >= 9.1 verhindert dies

- ***Dirty Read:*** Lese Daten von anderen, nicht committed T's
- ***Fuzzy Read:*** Lese gleiche Daten mehrmals, sehe aber andere Werte, gelesene Daten ändern sich durch andere T
- ***Phantom Read:*** Entdecke durch SELECT neue/gelöschte Rows nach INSERT/DELETE
- ***Serializable:*** Kann r,w Konfliktpaare blockieren
- ***Read Committed:*** Kann w,w Konfliktpaare blockieren.

Alternative: Optimistische Verfahren ohne Locks:
- Jede Transaktion sieht Snapshot zu Start-Zeitpunkt
- Bei Änderung im Commit prüfen, dass Objekte unverändert sind, wie zum Snapshot-Zeitpunkt, sonst Rollback



- ***Multi-Version Concurrency Control (MVCC):*** Mehrere Versionen x pro DB-Objekt, jede T hat Timestamp t des Startzeitpunkts. `write(x):` neue Version x_t, `read(x):` letzte Version x_t, mit grösstem t. Verfahren: Write -> Tupels X-Lock -> Deadlocks möglich. Lesen -> keine Locks, nicht überprüft. Jedes Update führt zu neuer Version des Tupels, spätere T wird wiederholt.
- ***Verhalten:*** `write` blockiert keine `read`, `read` blockiert keine `write`. `write` blockiert `write`, welche dasselbe Tupel modifizieren. `READ COMMITTED:` `read` nur mit derselben Version. `REPEATABLE READ` oder `SERIALIZABLE`: ganze T nur mit derselben Version. Serialisierbar nur mit Leven `SERIALIZABLE`

	Garantiert Serialisierbar	Keine Deadlocks	Keine Cascading Rollbacks	Keine Konflikt-Rollbacks	Hohe Parallelität	Realistisch (ohne Voranalyse)
Two-Phase Locking	✓	✗	✗	✓	✗	✗
Strict 2PL	✓	✗	✓	✓	✗	✓
Preclaiming 2PL	✓	✓	✓	✓	✗	✗
Validation-Based	✓	✓	✗	✗	✓	✓
Timestamp-Based	✓	✓	✗	✗	✓	✓
Snapshot Isolation	✗	*✓✗	✓	✗	✓	✓
SSI	✓	*✓✗	✓	✗	✓	✓

* Deadlock in PostgreSQL mit Snapshot Isolation

Log-Files

Write-Ahead Log (WAL): Änderung der T in Log schreiben (Flush), danach Commit (atomar) in Log schreiben (Flush), danach in-place-Updates in DB machen.
Aufbau: `[LSN, TaID, PageID, Redo, Undo, PrevLSN]`

Nach Absturz Recovery: Replay/Undo nach Log für alle committed Transaktionen *(Wenn beim Absturz noch aktiv – redo, sonst undo)*

Backup

Planung: Maximale Ausfallzeit, Zeit für Recovery. Was und wann soll gesichert werden? Volles Backup, inkrementeller Backup (nur logs), Export, Online/Offline Backup? Spiegelung des Log-Files durch DBMS oder OS?

Backup-Arten
- ***Logischer Backup «SQL Dump»:*** Blockiert keine schreibende oder lesende T. Für mittelgrosse Datenmengen, interkompatibel mit neuen PG-Versionen und anderen Maschinen.
- ***Physischer Backup – File-System:*** Datenbank muss gestoppt werden, schneller als logisches Backup, passt nur zu derselben «Major Version» von PG.
- ***Andere:*** Cloud, ContinousArchiving, Snapshot, Agent

Indexe

Beschleunigt SELECT auf Kosten `INSERT/UPDATE/DELETE`
Primär-Index: Auf PK, Sekundär-Index: Auf nicht PK
Data Pages: Heap, ***Index Pages:*** Suchbaum (B-Baum).

Arten
- ***ISAM*** *(Index-Sequential Access Method)*: Einfügen und Suchen ist einfach und schnell, aktualisieren schlecht. Daten werden über die Indexspalte aufsteigend sortiert.
- ***B-Bäume:*** Balanciert, gecluserter Index. Geeignet für Hintergrundspeicher, fast optimal für Queries und Einfügen. k=2 und m=4 -> 2 bis 4 Einträge, max degree 5. *Einfügen: Suche Schlüssel, füge Schlüssel ein, falls Blatt überfüllt – neuer Knoten mit mittleren und rechts liegenden Einträgen erstellen, kleinster Knoten in Vaterknoten verschieben.*
- ***B+-Bäume:*** Referenz nur in den Blättern, Blätter sind verkettet.
- ***Hash:*** Ordnet Key zu Einträgen. Problem: Overflow.
- ***Spezielle Speicherstrukturen bei PostgreSQL***

Indexe in PostgreSQL
- ***B-Tree:*** Default. Universell, Bereichsabfragen, Vergleiche, Mustersuche
- ***Hash:*** Ab PG Version 9.3 nicht nutzen. B-Tree ist in fast allen Aspekten besser.
- ***GiST:*** Range/Containment Search, KNN Search *(Generalized Search Tree)*
- ***GIN:*** Gut für Arrays, speichert effizient Duplikate, VS GiST: erstellen, update langsamer, Zugriff schneller *(Generalized Inverted Index)*
- ***BRIN:*** Speichert min/max-Werte als «Blöcke», gut für «Range Search». Gut für natürlich benachbarte oder sortierte Daten, kleinerer Disk-Verbrauch. *(Block Range Index)*

Index-Variationen:
- ***Zusammengesetzter Index:*** Index über mehrere Attribute / Columns *(erlaubt suchen über col1 ohne col2, aber nicht umgekehrt)*
  ```
  CREATE INDEX idx ON tablename USING btree(col1);
  CREATE INDEX idx ON tablename ((col1 - col2));
  ```
- ***Index mit INCLUDE:***
  ```
  CREATE INDEX idx ON tablename(col1) INCLUDE(col2);
  ```
- ***Partieller Index:*** Queries beziehen sich auf eine Selektion
  ```
  CREATE INDEX idx ON table(c1) WHERE c2 IS NOT NULL;
  ```
- ***Funktionaler Index:*** Index mit Ausdruck
  ```
  CREATE INDEX idx ON tablename(function(col));
  ```

Index-Spalten Dos: häufig in wheres & joins, constraints unique not null. Don'ts: häufig ändernde Spalten, bool & enums, viele INSERTs