Begriffe

Assoziation / Beziehungstyp	Verknüpft mehrere Entitätstypen. Binäre Assoziationen verknüpfen zwei Entitätstypen; n-wertige Assoziationen verknüpfen n Entitätstypen.
Attribut	Beschreibt eine Eigenschaft einer Entität oder Assoziation. Das Attribut besteht aus einem Namen und einem Datentyp
Attributwert	Ist der Inhalt/Wert eines Attributes. («Fritz» für das Attribute name)
Data Dictionary, Datenkatalog, Metadatenbank	Inhaltsverzeichnis des DBS. Enthält Informationen über Datenbasis (Tabellen, Views,Attribute,Relationen/Beziehungen, Benutzerrechte)
Datenbasis	Die eigentlichen physisch, persistent gespeicherten Anwendungsdaten, kurz Daten, des DBS.
Datenbanksystem	Besteht aus Datenbankmanagementsystem und einer oder mehreren Datenbasen (DB's)
Entität	Eindeutig identifizierbares Objekt der realen Welt (z.B Der Angestellte Fritz Meier)
Entitätsmenge	Gleichartige Menge von Entitäten (z.B Alle Angestellte aus Zürich)
Entitätstyp	Die Menge aller möglichen gleichartig strukturierten Entitäten. Jeder Entitätstyp hat einen eindeutigen Namen und eine Mengen von Attributen. (z.B Angestellter)
Kanonische Lösung	Keine Nullwerte
Schwacher Entitätstyp	Ein Entitätstyp dessen Primärschlüssel von der anderen Entitäten und deren Primärschlüssel abhängig ist
Relation/Beziehung	Verknüpfung mehrerer Entitäten miteinander. (z.B 'Fritz Müller' gehört zur Abteilung 'Entwicklung')
Surrogatschlüssel	Besteht ein Primärschlüssel aus mehreren Attributen, kann auf eine Entität nicht mehr Eindeutig Zugegriffen werden ohne alle Attribute anzugeben. Zudem wird der Index weniger effizient. Eine Lösung stellt hierbei ein künstlicher Primärschlüssel als fortlaufende Nummer dar. (Anwendungsintern)

ANSI 3-Ebenen Modell

1) Externe Ebene: Benutzersicht auf DB. SQL oder über GUI
2) Logische, konzeptionelle Ebene: Beschreibt Daten in Gesamtheit. Beziehungen, Integritätsbedinungen, Zugriffsbedingungen.
3) Interne, physiche Ebene: Speicherstrukturen. Wie und wo Daten in der DB gespeichert werden. Effizienter Zugriff auf Daten (Index, B-Tree, Zugriffsrechte)"

Benutzer (analog zu ANSI 3-Ebenen Modell)

DBA: Installation, Wartung, Backup, Recovery, Speicherplatz, Benutzer, Laden von Daten
DB Entwickler: Design und Entwicklung, Datenmodell, Masken, Applikation, SQL
DB Benutzer: Benutzt Applikation und führt allenfalls einfache SQL aus

Vorteile DBS gegenüber Dateiablage

- kontrollierbare Datenintegrität/Constraints
- Transaktionen
- Mehrbenutzerbetrieb (Sessionmngmt.)
- Kapselung der Daten
- Abfrage Sprache (SQL)
- Zugriffskontrolle / Sicherheit / Authent.

ACID

Atomicity: Transaktion wird ganz oder gar nicht ausgeführt
Consistency: Nach Transaktion sind Daten konsistent
Isolation: Gleichzeitige Ausführungen beeinflussen sich nicht
Durability: Auswirkungen auf Daten bleiben dauerhaft

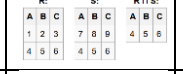
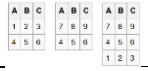
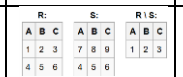
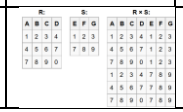
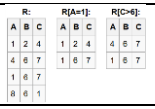
Indexe

B-Tree, B+Tree (Balanced)	Grosse Datenmengen die häufig ändern (schnell)
ISAM (Index Sequential Access Method)	Einfügen und suchen schnell Aktualisieren langsam
HASH (B-Tree unterlegen)	Nicht im WAL-> Kaputt bei Crash
BRIN (Block Range Index)	Sehr kleiner Index
GIST / GIN	Volltextsuchen

Normalformen (Vorhergehende Normalformen werden immer vorausgesetzt)

1. Normalform: Daten sind Atomar (keine mengenwertige Werte), Komalisten in mehrere Datensätze
2. Normalform: Festlegen, welche Attribute von welchem Primärschlüssel abhängig sind. Jedes Nichtschlüsselattribut voll funktional abhängig vom einem Primarschlüssel. Keine Abhängigkeit zu einem Teilschlüssel.
3. Normalform: Auslagern in mehrere Tabellen mit Relationen (Kein Nichtschlüsselattribut transitiv abhängig (=über zwei Schlüssel abhängig) von einem Schlüsselattribut)
Boyce-Codd Normalform: Aufteilen in Tabellen, damit kein Attribut nur von einem Teil einer zusammengesetzten ID abhängig ist.
4. Normalform: Keine Daten ohne Zusammenhang in einer Tabelle
5. Normalform: Nur eine Abbildung pro Tabelle

Relationale Algebra

Durchschnitt ∩		**Vereinigung U**	
**Differenz **		**Projektion ∏ (SELECT DISTINCT)**	
Kartesisches Produkt X		**Selektion σ (WHERE)**	

SQL

DDL: Data Definition Language

CREATE DATABSE, TABLE, INDEX, VIEW, Temporary Table

```
CREATE DATABASE [dbname] WITH OWNER = 'name'
CREATE TYPE sex type AS ENUM ('male', 'female', 'trans');

CREATE TABLE [tablename] (
  ID SERIAL PRIMARY KEY,
  sex sex_type,
  employeeno INTEGER,
  age INTEGER CHECK age > 18
  name VARCHAR(255) UNIQUE NOT NULL,
  startdate  DATE DEFAULT CURRENT_DATE,
  insertdate TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  paid BOOLEAN default FALSE,
  price REAL NOT NULL, -- that's a float value
  FOREIGN KEY (employeeno) REFERENCES person(personid)
    [ON DELETE {CASCADE|RESTRICT|NO ACTION|SET NULL|SET DEFAULT}]);
CREATE INDEX [indexname] ON [tablename] {(col) | (col1,col2) |
(immutable function(col)) | [USING {btree|gist|...}(col)]}
[WHERE condition];
CREATE VIEW [viewname] AS SELECT name, date, age FROM [t/v] WHERE age > 18
CREATE RULE kurs update AS ON UPDATE TO kursuebersicht
DO INSTEAD UPDATE kurs SET beschreibung =NEW.beschreibung
WHERE kursnummer = NEW.kursnummer;
CREATE TEMPORARY TABLE [tablename] AS SELECT * FROM B
```

ALTER TABLE/COLUMN

```
ALTER TABLE [tablename] ADD COLUMN [colname]  varchar(30);
ALTER TABLE [tablename] DROP COLUMN|CONSTRAINT [name];
ALTER TABLE [tablename] RENAME [colname] TO [new_name];
ALTER TABLE [tablename] ALTER COLUMN [colname] TYPE integer;
ALTER TABLE [tablename] ALTER COLUMN [colname] TYPE sex_type USING
[colname]::sex_type;
ALTER TABLE [tablename] ADD CONSTRAINT fk_one FOREIGN KEY (col) REFERENCES
B(column) ON DELETE CASCADE;
```

DQL: Data Query Language

```
SELECT DISTINCT sex, age FROM [tablename] WHERE age > 18 AND name LIKE
'%name%' AND MAX(price) GROUP BY sex, age ORDER BY age DESC LIMIT 1;
```

WINDOW FUNCTIONS

```
SELECT persnr, abtnr, avg(salaer) OVER (PARTITION BY abtnr)
FROM angestellter; --compare empl. salary with average of department
```

`row_number()`	Number of the current row within its partion (start 1)
`rank()`	Rank of the current row with gaps
`ntile(num)`	Integer ranging from 1 to the argument value
`lag(offset)`	Value at the row that is offset rows before the current row
`lead(offset)`	Value at the row that is offset row after the current row
`first_value(val)`	Value at the row that is the first row of the window frame

SUBQUERIES (Korreliert, Unkorreliert)

```
SELECT Name, Description FROM Products p WHERE Quantity < 2 * (
  SELECT AVG( Quantity ) FROM SalesOrderItems s WHERE p.ID=s.ProductID );
SELECT Name, Description FROM Products WHERE Quantity < 2 * (
  SELECT AVG( Quantity ) FROM SalesOrderItems ); --unkorreliert
```

EXISTS / IN

```
SELECT * FROM Kunden WHERE EXISTS (SELECT * FROM Aufträge WHERE
Kunden.Kun_Nr = Auftraege.Kun_Nr);
SELECT * FROM Kunden WHERE Kun_Nr IN (SELECT Kun_Nr FROM Auftraege);
```

ALL / ANY, SOME

```
SELECT * FROM angestellter WHERE gehalt < {ALL|ANY,SOME} (SELECT gehalt
FROM angestellter); -- any=some -> at least one row must match
```

UNION (OR), INTERSECT (AND), EXCEPT (OHNE)

```
SELECT ID, Kennzeichen, Farbe FROM Dienstwagen
  {UNION [ALL]|INTERSECT [ALL]|EXCEPT [ALL]} -- ALL= Incl. duplicate
SELECT ID, Kennzeichen, Farbe FROM Fahrzeug;
```

HAVING

```
SELECT ID, name, age, address, salary FROM customers
GROUP BY age HAVING COUNT(age) >= 2;
```

CTE : COMMON TABLE EXPRESSION, WITH (Vereinfachung komplexer Queries)

Wiederverwendbar innerhalb einer Abfrage, rekursiv, Ersatz für Views

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total sales FROM orders GROUP BY region
), top regions AS (
    SELECT region FROM regional sales WHERE total sales >
      (SELECT SUM(total_sales)/10 FROM regional_sales)
) SELECT region, product, SUM(quantity) AS product_units, SUM(amount) AS
product_sales
  FROM orders
  WHERE region IN (SELECT region FROM top_regions)
  GROUP BY region, product;
```

WITH RECURSIVE

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

NULL, CAST/CASE, COALESCE (Nullwerte ersetzen)

```
SELECT * FROM A WHERE A.col IS NOT NULL
SELECT CAST(CASE WHEN paid IS NULL THEN 0 ELSE 1 END AS BOOLEAN) FROM A
select avg(r.date::timestamp - r.date::timestamp) from rueckgabe
SELECT COALESCE(firstname,'no first name')-- Ersetzt null Werte mit dem
jeweilig folgenden Argument: COALESCE(firstname, company, title);
SELECT(null = null); --eine leere Zeile 'unknown', da Dreiwertigkeitslogik
```

DML: Data Manipulation Language

```
INSERT INTO [tablename](col1, col2) VALUES (1, 'Test');
UPDATE [tablename] SET col1=5000 WHERE col1 = 1;
DELETE FROM [tablename] WHERE col1 < 8000 RETURNING *;
TRUNCATE [tablename]; -- performanter wie DELETE
```

DCL: Data Control Language

```
CREATE ROLE [user] WITH LOGIN PASSWORD = 'secure'
DROP ROLE [rolename]
CREATE ROLE [group]
ALTER ROLE [role] WITH PASSWORD 'new secure' -- set new password
ALTER ROLE [role] WITH {SUPERUSER|CREATEDB|CREATEROLE|CREATEUSER}
GRANT [group] TO [user] -- add user to group
GRANT {SELECT|INSERT|UPDATE|DELETE|ALL} ON TABLE [tablename] TO
{user|group|PUBLIC} [WITH GRANT OPTION]
REVOKE {SELECT|INSERT|UPDATE|DELETE|ALL} ON TABLE [tablename] FROM
{user|group|PUBLIC} [CASCADE|RESTRICT]
```

REVOKE CASCADE entzieht Rechte auch bei 3th Party. Wurden Rechte weitervergeben und es wird kein CASCADE angegeben, schlägt REVOKE fehl!

Berechtigungen können auf Tabellen, Columns, Views, Sequenzen, Datenbanken, Schemas oder Tablespaces vergeben werden.

Joins

Inner Join `SELECT * FROM A INNER JOIN B ON A.Key = B.Key`	Gibt alle Zeilen zurück, die eine Übereinstimmung in beiden Tabellen haben	
Left Join ⋉ `SELECT * FROM A LEFT JOIN B ON A.Key = B.Key`	Gibt alle Zeilen der linken Tabelle und alle Übereinstimmungen in der Rechte Tabelle zurück	
Right Join ⋊ `SELECT * FROM A RIGHT JOIN B ON A.Key = B.Key`	Gibt alle Zeile der rechten Tabelle und alle Übereinstimmungen in der Linken Tabelle zurück	
Outer Join / Full Join ⋈ `SELECT * FROM A OUTER JOIN B ON A.Key = B.Key`	Gibt alle Zeilen zurück wenn es eine Übereinstimmung in einer der Tabellen gibt	
Excluding Join	`SELECT * FROM A INNER JOIN B ON A.Key = B.Key WHERE B.KEY IS NULL`	

Aggregatsfunktionen		Skalare Funktionen	
`AVG(col)`	Durchschnitt	`UPPER(col)`	Uppercase
`COUNT(col)`	Anzahl Zeilen	`LOWER(col)`	Lowercase
`FIRST(col)`	Erster Wert	`SUBSTR(col, start, length)`	Substring
`LAST(col)`	Letzter Wert	`LENGTH(col)`	Lenge eines Textfeldes
`MAX(col)`	Grösster Wert	`ROUND(col, 2)`	Rundet Dezimalstellen
`MIN(col)`	Kleinster Wert	`NOW()`	Aktuelle Systemzeit
`SUM(col)`	Summe	`FORMAT('%s', NOW())`	

Konvertieren

Number → String	`to_char(125, '999'); -- 9=Wildcard`
String → Date	`to_date('01.01.2016', 'DD.MM.YYYY');`
String → Timestamp	`to_timestamp('01.01.2016 16:00:00', 'DD.MM.YYYY HH24:MI:SS')`
String → Number	`to_number('12,454.8-', '99G999D9S'); -- = -12454.8`

Transaktionen

```
BEGIN TRANSACTION ISOLATION LEVEL [level] ;
  INSERT INTO A VALUES (123, 'test') ;
  SAVEPOINT my_safepoint ;
     INSERT INTO B VALUES (454, 'qwerz');
     ROLLBACK TO my_safepoint ;
  INSERT INTO B VALUES(545, 'bob' ;
COMMIT;
```

Isolation Levels

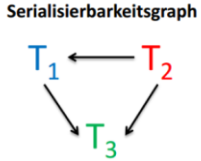
READ UNCOMMITTED: Liest auch Daten, welche UNcommited sind
READ COMMITTED: Liest nur Daten, welche COMMITed sind
REPEATABLE READ: Lesevorgänge gleich wiederholbar
SERIALIZABLE: Nacheinander

Serialisierbarkeit

$r_2(x)\ r_1(x)\ w_1(x)\ r_3(x)\ w_3(x)\ c_3\ w_2(y)\ w_1(y)\ c_1\ c_2$

	T1	T2	T3
1		$r_2(x)$	
2	$r_1(x)$		
3	$w_1(x)$		
4			$r_3(x)$
5			$w_3(x)$
6		$w_2(y)$	
7	$w_1(y)$		

Serialisierbarkeitsgraph



Zyklus => Nicht serialisierbar

Kein Zyklus => Serialisierbar

1) Konfliktpaare bestimmen
- Mindestens eine Write Operation
- Beide Operation auf gleiches Objekt (auch innerhalb derselben Transaktion)

2) Serialisierbarkeitsgraph erstellen
3) Commit Reihenfolge erstellen

```
r1(b) < w2(b)
w2(b) < r3(b)
r2(c) < w4(c)
r4(d) < w4(d)
r2(d) < w4(d)
w3(a) < r5(a)
r5(c) < w4(c)
r2(b) < w2(b)
```

2) Serialisierungsgraph

$$S1' = \underbrace{r1(b)}_{T1}\ \underbrace{r2(b)\ w2(b)\ r2(c)\ r2(d)}_{T2}\ \underbrace{w3(a)\ r3(b)}_{T3}\ \underbrace{r5(c)\ r5(a)}_{T5}\ \underbrace{r4(d)\ w4(d)\ w4(c)}_{T4}$$

Serialisierbar wenn keine gegenseitigen Abhängigkeiten der Transaktionen ($T_1 \Leftrightarrow T_2$)

Locking (X-Lock=Exclusive, S-Lock=Shared)

Two Phase Locking : 1 Growing Phase, 2. Shrinking Phase (UNLOCK())

SLOCK: Wird bei einem lesenden Zugriff abgesetzt. Ein Objekt kann von mehreren Transaktionen gleichzeitig gesperrt werden (allerdings kein XLOCK)

XLOCK: Wird bei einem schreibenden Zugriff abgesetzt. Ein Objekt kann nicht doppelt gesperrt werden.

Deadlock: Mehrere Transaktionen behindern sich gegenseitig beim Locking der jeweilig anderen Ressource.

Optimistisches Locking: Geht von wenigen schreibenden Zugriffen aus; Auf SLOCK wird verzichtet, aber es wird geprüft ob sich ein Zeitstempel geändert hat

Pessimistisches Locking: Geht von vielen schreibenden Zugriffen aus; SLOCK und XLOCKS

Multiuser Probleme

Isolationsgrad	Dirty-Read	Non-Repeatable-Read	Phantom
read uncommitted	ja	ja	ja
read committed	nein	ja	ja
repeatable read	nein	nein	ja
serializable	nein	nein	nein

Dirty Read: Uncommitete Daten werden gelesen

Non Repeatable Read/Fuzzy Read: Daten werden zwischendurch geändert, gelöscht

Phantom Read: Neue Daten werden zwischendurch eingefügt (tritt in Postgres nicht auf)

	Garantiert Serialisierbar	Keine Deadlocks	Keine Cascading Rollbacks	Keine Konflickt-Rollbacks	Hohe Parallelität	Realistisch (ohne Voranalyse)
Two-Phase Locking	x	-	-	x	-	-
Strict 2PL	x	-	x	x	-	x
Preclaiming 2PL	x	x	x	x	-	-
Validation-Based	x	x	-	-	x	x
Timestamp-based	x	x	-	-	x	x
Snapshot Isolation	-	x	x	-	x	x
SSI	x	x	x	-	x	x

Backup

Full Backup: Daten und Log-Files / **Incremental Backup:** Log-File
Export: pg_dump dbname > outfile, **Import:** psql dbname < infile
Mirroring : Server Replikation

Write Ahead Log (WAL)

1) Änderungen der Transaktion in Log schreiben 2) Commit ins Log schreiben 3) Commit in die physiche Datenbank

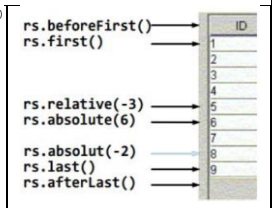
JDBC

```
try(Connection con = DriverManager.getConnection(db, u, p)) {
 con.setAutoCommit(false);
 con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
// or TRANSACTION_REPEATABLE_READ, TRANSACTION_READ_COMMITTED,
//TRANSACTION_READ_UNCOMMITTED          //SQL Exception Class
 con.commit();                         SQLWarning w = statement.getWanings();
} catch (SQLException exception){      w.getSQLState(); w.getErrorCode();
  con.rollback();                      w.getMessage(); w= w.getNextWarning();
  throw exception;}
```

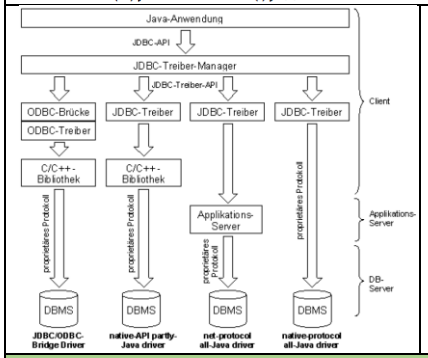
```
1  import java.sql.*;
2  public class InsertCoffees {
3      public static void main(String args[]) {
4          final String user = "jdbctut", password = "jdbctut";
5          final String database = "jdbc:postgresql://localhost/jdbctut";
6          final String query = "select COF_NAME, PRICE from COFFEES";
7          final String INSERT_COFFEE = "insert into COFFEES values(?, ?, ?, ?, ?)";
8          try (Connection con = DriverManager.getConnection(database, user, password)
9              try (Statement stmt = con.createStatement();
10                 PreparedStatement insertStmt = con.prepareStatement(INSERT_COFFEE))
11
12                 /* insert statements */
13                 insertStmt.setInt(1, 1234); insertStmt.addBatch();
14                 insertStmt.setInt(1, 4321); insertStmt.addBatch();
15                 insertStmt.executeBatch();
16
17                 /* query statement*/
18                 ResultSet rs = stmt.executeQuery(query);
19                 while (rs.next()) {
20                     String s = rs.getString("COF_NAME");
21                     float f = rs.getFloat("PRICE");
22             }}
23         } catch (SQLException ex) { System.err.println(ex.getMessage());
24 }}}
```

```
int getColumnDisplaySize(ResultSetMetaData rsm, int column)
  throws SQLException {
   return Math.max(rsm.getColumnDisplaySize(column),
     rsm.getColumnName(column).length());
}

try { // ResulSet rs
  ResultSetMetaData rsm = rs.getMetaData();
  for (int i = 1; i <= rsm.getColumnCount(); i++) {
    String name = rsm.getColumnLabel(i);
    int size = getColumnDisplaySize(rsm, i);
  }
  // Connection con
  DatabaseMetaData dbm = con.getMetaData();
  dbm.getDriverVersion();
  dbm.getDatabaseProductName();
  dbm.getDatabaseProductVersion();
} catch (SQLException e) {
    e.printStackTrace();
}
```




```
Statement s = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
rs.first(); rs.updateInt("Age", 100); rs.updateRow(); // update
rs.moveToInsertRow(); rs.updateInt("Bal",200); rs.insertRow(); // insert
rs.absolute(1); rs.deleteRow(); // delete
```



1)JDBC-ODBC Brücke
2)Native plattformeigene JDBC Treiber
3)Universeller JDBC Treiber (JAVA)
4)Direkte Netzwerktreiber (JAVA)

Constraints (Column, Table)

NOT NULL, UNIQUE, CHECK, PRIMARY KEY, FOREIGN KEY, CONSTRAINT mit Namen

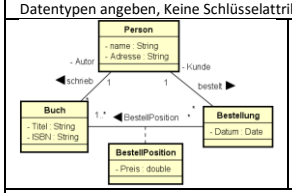
```
CREATE TABLE product (
    product_no INTEGER,
    order_no INTEGER,
    description VARCHAR(20),
    discounted_price INTEGER,
    price INTEGER NOT NULL,
    CHECK(price > discounted_price),
    UNIQUE (product_no, description),
    PRIMARY KEY(product_no),
    FOREIGN KEY(order no) REFERENCES order(id),
    CONSTRAINT constraint name {CHECK(a < b) | UNIQUE()},
}
```

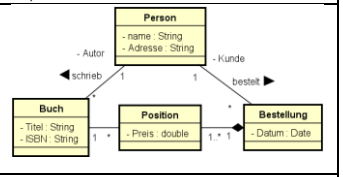
Datenbankentwurf Prozess

1) Informations-, Datenverarbeitungsanforderungen 2) konzeptionelles Domänen Modell 3) logisches, relationales Modell 4) physisches DB Modell 5) Internes Schema oder spez DB.

Logischen Domainmodell (UML)

Datentypen angeben, Keine Schlüsselattribute, n-m mit Assoziationstabellen





Aggregation und Komposition: Ist-Teil-von-Ganzem Beziehung wobei Komposition stärker bindet. (Wird der Parent gelöscht, muss auch der Child gelöscht werden. CASCADE)

Relationale Schreibweise

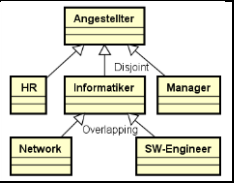
Person(id INTEGER PK, name String NOT NULL, email String NOT NULL UNIQUE);
Hund(id INTEGER PK, name String NOT NULL, PersID REFERENCES Person);
Bauer(id INTEGER PK, hofNr INTEGER NOT NULL CHECK > 0, PersID REFERENCES Person);
PrivatPerson(id INTEGER PK, ausweisNr INTEGER, PersID REFERENCES Person);

REFERENCES verweisst implizit auf den Primärschlüssel. Ansonsten REFERENCES Person(name);

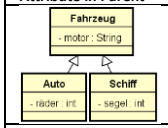
Vererbung / Abbildungsregeln

Overlapping: Attribute können zu mehreren Tabellen gehören.

Disjunkt: Attribute können zu genau einer Tabelle gehören.

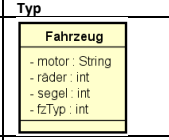


Gemeinsame Attribute in Parent



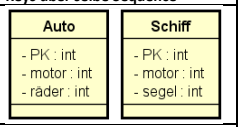
Vorteil: Strikte Trennung. Keine Redundanzen
Nachteil: Mehrere Joins nötig

Alles im Parent und mit Typ



Vorteil: Keine Joins, alles in einer Tabelle
Nachteil: Constraint muss Zwitter unterbinden

Alles in Kindtabelle und Primary Keys über selbe Sequence

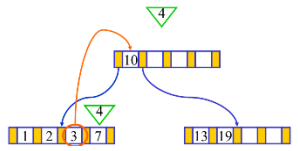


Vorteil: Guter Mix zwischen den anderen beiden Lösungen
Nachteil: Reduntante Felder

B-Tree

m: max. Anzahl Element pro Knotenpunkt muss >3 sein
N: Anzahl Knotenpunkte
- Knotenpunkt:
  - Ein Knotenpunkt beinhaltet mindestens m/2 Elemente (ausser root >=1)
  - Ein Knotenpunkt hat maximal m+1 Unterknoten (sonst rebalancing)
  - Der Baum ist überall gleich hoch (sonst rebalancing)
    max. Höhe: h=1+[log (m/2) (N/2)]

Einfügen:
1. Suchen nach Ordnungsplatz, einfügen
2. Falls Überlauf, teilen
2.1. Neuer Knoten mit Zahlen rechts von Mittlerer
2.2. Mittlerer Eintrag in Vaterknoten
2.3. Neuer Knoten mit Vaterknoten verknüpfen
3. Falls Vaterknoten überfüllt:
3.1. Falls Root, neue Root anlegen
3.2. Ansonsten wie bei 2.



4 und 7 ergeben neuen Unterknoten