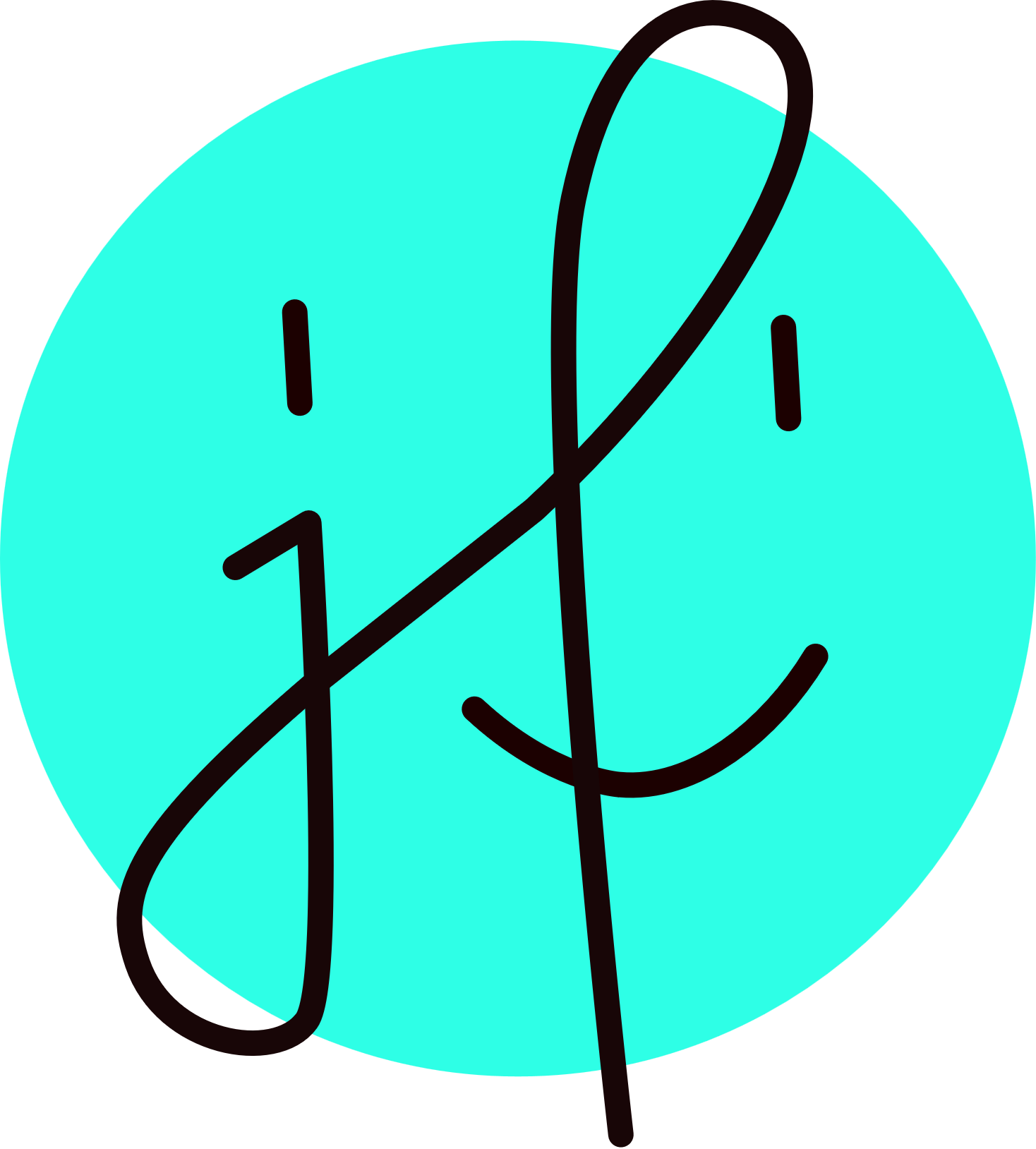
# begriffe / Theorie

FK = Fremdschlüssel | PK=Primary Key=Primärschlüssel | → Pfeil um Text zu sparen z.B. daraus folgt / also /gibt | PG = PostgreSQL



Jasmin Fässler 2024

ACIDAtomicity (Integrity), Consistency, Isolation (Concurrency, Multiuser), Durability (Security) und weiterer Vorteil DB-Systeme im Gegensatz zu Dateisystem: Kapselung

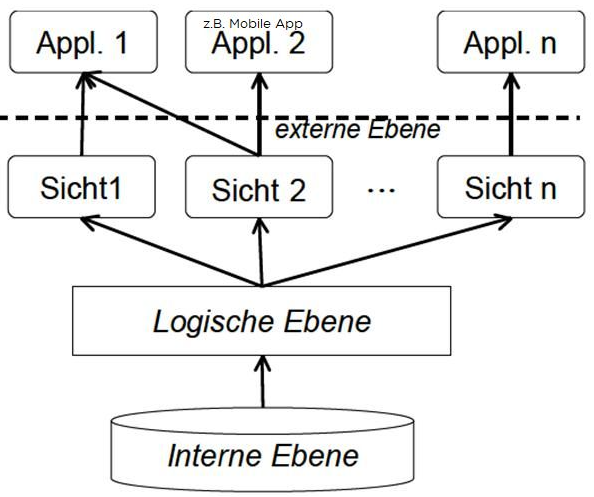
Anforderungen an DBMS Redundanzfreiheit + Datenintegrität

Datenintegrität Datenkonsistenz (logische Widerspruchsfreiheit der Daten), Datensicherheit (Schutz vor physischem Verlust) & Datenschutz (Schutz vor unberechtigtem Zugriff)

Datenmodelle Hierarchisch, Netzwerk, Relationen, Postrelational: Objektrelational (Methoden, Tabellenvererbung), Objektorientiert (analog Programmiersprache)

DBMS-Funktionen Backup & Recovery, SQL/ QuerySprachen, Datentypen, ACID, Tran-saktionen (A), Konsistenz (C), Mehrbenutzerbetrieb (I), Grosse Datenmengen/Sicherheit (D)

### Ansi-3-Ebenen-Modell (Trennung der Ebenen!) Architektur

Externe Ebene (z.B. App, Website zu DB)

* Sicht einer Benutzerklasse auf Teilmenge der DB
* Definition durch externes Schema

Konzeptionelle/Logische Ebene (UML/rel. Schreibw.)

* Logische Struktur der Daten
* Definition durch log. Schema «Trägermodell»

Interne Ebene

* Speicherstrukturen, physische Datenstruktur
* Definition durch internes Schema, DDL

### PostgreSQL ORDBMS nach SQL Standard mit ACID

Client-Server Modell, physische und logische Ebene getrennt, Buffer, WAL Write ahead log zuerst im log speichern danach in DB

### Datenbankmanagement (DBMS) Funktionen

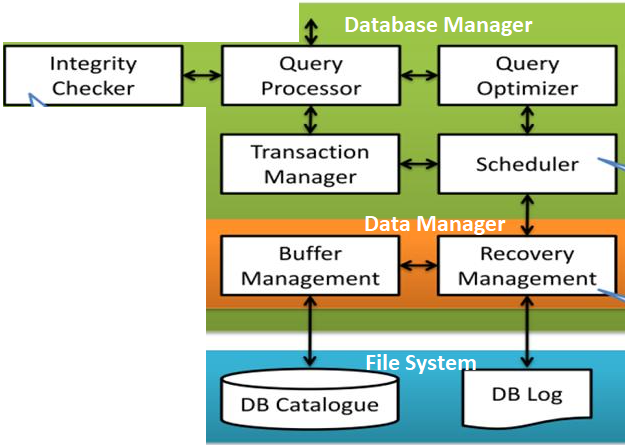
* Speicherung /Persistenz (dauerhaft)
* Transaktionen («alles oder nichts» - Operationen)

DBMS Grafik ACID Zusammenhang

Integrity Checker: C

Scheduler: I,C

Recovery Management: A,D

* Mehrbenutzerbetrieb (Sync parallele Zugriffe)
* Sicherheit: Authentifizierung und Autorisierung
* Backup und Recovery
* Datentypen /Datenstrukturen mit Datenkatalog
* Abfragesprachen (z.B. SQL) mit Optimierer
* Programmierschnittstellen (z.B. JDBC)

### Anforderungen an DBMS

Redundanzfreiheit Jedes Element nur einmal

Datenintegrität

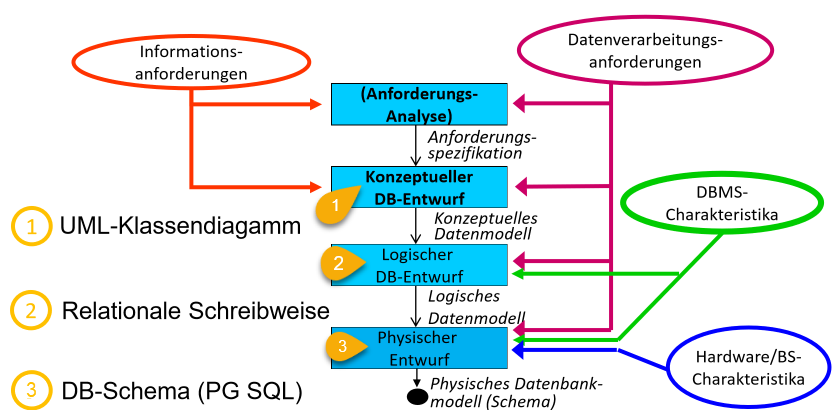
* Datenkonsistenz = logische Widerspruchsfreiheit
* Datensicherheit = Sicherung von physischem Verlust
* Datenschutz = Schutz der Daten   
  vor unberechtigtem Zugriff

### Datenbankmodell (Paradigma)

* Legt Datenstrukturierungs-Konzepte   
  des DBMS fest (Beziehungen)
* Legt die Datenstrukturen fest   
  (Datentypen, Speicherung)

- Auch Paradigma genannt (in UML 2 Meta-Modell)

### Datenbankmodelle

* Hierarchisches Datenmodell (legacy) 1 einzige Hierarchie, Welt nicht modellierbar
* Netzwerkmodell eine vernetzte Hierarchie, Beziehungen zusammen mit Daten, ineffizient
* Relationen Modell 1. Normalform in Tabellen=Relationen, Daten & Anwendungen getrennt, Impedance Mismatch möglich, komplexe Abfragen > Performance Probleme

## Normalformen

1.NF Wertebereiche der Attribute sind atomar, Alle Tabellenspalten beinhalten gleichartige Werte, keine Mengen, hat PK.

2.NF Jedes Nichtschlüsselattribut muss vom Primärschlüssel voll funktional abhängig sein.  
NICHT in 2. NF, wenn sie einen zusammengesetzten Schlüssel hat und mind. ein Nichtschlüsselattribut von nur einem Teil des Schlüssels funktional abhängig ist

3.NF Kein Nichtschlüsselattribut darf von einem Schlüsselkandidaten transitiv abhängig sein.

Boyce CoddKein Attribut von Schlüsselkand. transitiv abhängig, ausser triviale Abhängigkeit. Denormalisierung auf 2. NF + Performanz bei Anfragen über mehrere Tabellen + weniger komplizierter Zugriff - Redundanzen (kann zu Anomalien führen) - braucht mehr Speicher

## Backup und Recovery

WAL write-ahead log, schreibt zuerst in WAL und dann erst in DB

NAS (Network Attached Storage), RAID (Redundant Array of independent Disks)

Fehlersituationen (beim Speichern): 1. Fehler/ROLLBACK, 2. Memory-Fehler/-Verlust, 3. Disk-Fehler-Verlust

Aspekte des DB-Backups Physischer vs. logischer Backup, Full vs. incremental Backup.

## Anomalien (Unregelmässigkeit, Regelwidrigkeit)

Einfügeanomalie

* Tupel kann erst eingefügt werden, wenn andere Tupel (z.B in anderen Tabellen) existieren

Löschanomalie

* Wenn Tupel gelöscht wird, gehen zusätzliche Infos verloren (Mitarbeiter löschen, löscht auch Abteilung)

Änderungsanomalie

* Eine Änderung muss auch an x Orten nachgetragen werden

## Optimierung Eines DBMS wie PostgreSQL

* Abfrageoptimierung, Ineffiziente Abfragen umgestalten, Komplexität reduzieren, «SELECT \*» vermeiden – sowie Just-in-time-Kompilierung (JIT)
* ist SQL-Query adäquat oder muss sie umgeschrieben werden? DB-Schema adäquat?
* Ist die DBMS-Konfiguration der Hardware angepasst? (GUC oder pgtune.leopard.in.ua)
* Tabellen-Statistik→ANALYZE VACUUM, DB-Logs überwachen (Probleme identifizieren)
* Indexoptimierung Sind die (richtigen) Indexe vorhanden?
* Parallele Abfrageausführung: Aktivieren und konfigurieren
* Partitionierung: Native Daten-Partitionierung einrichten
* Verbindungspooling: Datenbank-Verbindungen wiederverwenden (mit Tools)
* Zwischenspeicherung: Häufig abgefragte Daten (MATERIALIZED VIEW) und   
  Queries (PREPARED Statement) speichern
* Hardwareressourcen: CPU, RAM oder SSD hinzufügen

## **Indexe wählen** (nicht ohne Grund SELECT \* und indexe erstellen)

Index-Kandidaten sind

* Spalten, die häufig in Abfragen vorkommen (WHERE-Klausel)
* Spalten/Attribute in Constraints wie z.B. UNIQUE NOT NULL
* Spalten/Attribute, die in JOINS vorkommen, meist Schlüssel (PK, FK, Sekundär-S.)

Weniger gut eignen sich Indexe:

* bei Spalten, die sich häufig ändern
* bei Spalten, mit wenig unterschiedlichen Werten (Boolean, Enumerations)
* viele Daten auf's Mal eingefügt werden (Index löschen, Daten laden& wieder erstellen)
* bei Resultsets, die grösser als ~5-10% einer "grossen" (z.B. >100K) Tabelle sind – bei diesen ist ein Lesen der ganzen Tabelle ("Full Table Scan") schneller

## Relationale Algebra

Kartesisches Produkt SELECT \* FROM R1, R2 von >2 Tabellen Tupel kombiniert

projection SELECT A1, A4 FROM Tabelle

selection SELECT \* FROM Tabelle WHERE A1 > 30 (WHERE)

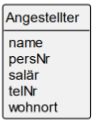
natural join SELECT \* FROM R natural join S gleicher Spaltenname kombiniert

theta-join SELECT \* FROM R join S on R.a < S.b INNER JOIN,equi ∈ theta join

division

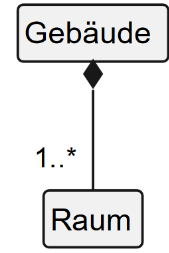
rename SELECT A2 as renamed from Tabelle (Alias)

# UML (datenmodellierung) Domänen modell

Aggregation ◊ (besteht aus, Teil von), Komposition: ♦ (Existenzielle Abhängigkeit des Teils vom Ganzen), Vererbung: Δ

Assoziations-Klasse Gibt der Beziehung eigene Attribute

Basisdatentypen TEXT, INTEGER, DECIMAL(10,2), FLOAT,PHONE\_NR, BOOLEAN, DATE, TIME, DATETIME

PG-Datentypen NUMERIC (10, 2) ist wie DECIMAL, ENUM-Typ

Multiplizität Angabe: 1, 0, 0..1, 1

Angabe der Bedeutung der Beziehung: Assoziations-Name (bearbeite ►) oder mind. eine der beiden Rollen (Projekt-Beteiligter)

## Generaliseriung/Vererbung Constraints

****{disjoint} default (disjunkt)→ Instanz genau einer Unterklasse

{overlapping} (nicht disjoint!) → mehrere Unterklassen gleichzeitig möglich und kann Unterklassenzugehörigkeit wechseln{complete} default → alle Subklassen definiert, Superklasse ist abstrakt

{incomplete} → nicht alle Objekte müssen in eine Subklasse fallen

{ordered} → Reihenfolge, Position festgelegt

# Logisches DB-Modell

## Abbildungsregeln, relationale Schreibweise

**Beziehungen** one-to-many: via Fremdschlüssel, many-to-many mit Zwischentabelle

Keine Semikolons, **Datentypen überall ausser FK angeben**! Datentypen freier als SQL

### Regel 1 : Abbildung von Klassen

1. NF einhalten, Pro UML Klasse eine Tabelle, NULL (optional), Surrogat-Schlüssel wenn nötig.

Student(StudId INT, Name VARCHAR(80), Kürzel CHAR(3) NOT NULL, Wohnort TEXT NOT NULL, Abflugzeit TIME)

### Regel 2.1 Abbildung von one-to-many Assoziationen (1..\* zu 1) (auch 1..\* zu 1..\* )

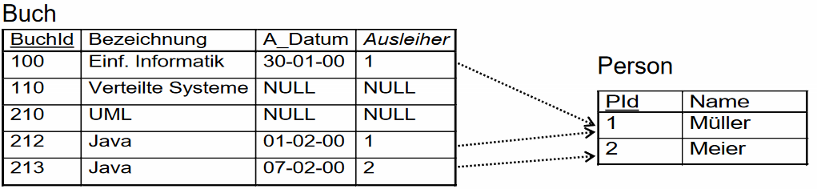
PK der 1-Tabelle muss in \*-Tabelle als FK vorkommen und Eigenschaften der Assoziation können als Attribute der \*-Tabelle geführt werden.

Abteilung (AbtId INT, Name TEXT)

Student (StudId INT, Name TEXT, Adresse TEXT,

*AbtId* INT **NOT NULL** REFERENCES Abteilung) -- DB-logisches Modell!

### Regel 2.2 Abbildung von optionalen Assoziationen (0..1 zu 0..\*)

a) wie 2.1, aber mit optionalen Beziehungsattributen  
• falls Beziehungen häufig vorkommen  
• Null-Werte bei nicht vorhandenen Beziehungen

Person (Pid, Name TEXT NOT NULL)

Buch (BuchId, Bezeichnung NOT NULL, Ausleihedatum DATE NULL,

*Ausleiher* **NULL** REFERENCES Person) --bei sql mit Datentypen

b) mit separater (abhängiger) Beziehungstabelle

• falls Beziehungen selten sind • ‘kanonische‘ Lösung ohne Null-Werte  
• PK = PK der Tabelle mit Kardinalität

Person (PId, Name NOT NULL)

Buch (BuchId, Bezeichnung NOT NULL)

Ausleihe(*BuchId* NOT NULL REFERENCES Buch,

 *PId* NOT NULL REFERENCES Person, Ausleihedatum DATE NOT NULL)

### Regel 2.3 Abbildung von Aggregationen I (Komposition ♦)

z.B. Angestellter ♦1 zu 0..\* Kind

Der Primärschlüssel einer abhängigen Tabelle ist entweder ein Fremdschlüssel oder er enthält Attribute eines Fremdschlüssels einer übergeordneten Tabelle.

Angestellter (AngId INT, Name VARCHAR(80) NOT NULL)

**a) Abhängige Tabelle, starke Ganzes-Teile-Beziehung (Komposition)** ♦

Kind (*AngId* REFERENCES Angestellter, -- NOT NULL UNIQUE default PK

Vorname, GebJahr NOT NULL) mit Datentypen überall ausser FK!

– ORDERED bei Angestellter > Vorname mit Attribut Order

**b) Unabhängige Tabelle, schwache Ganzes-Teile-Beziehung (Aggregation ◊)**

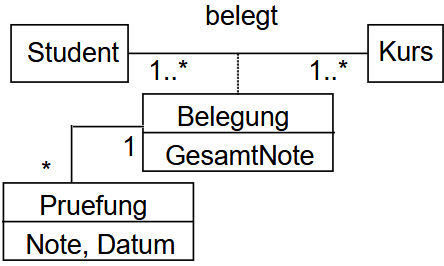
Kind (KindId, *AngId* NOT NULL REFERENCES Angestellter,

Vorname NOT NULL, GebJahr)

*Constraint*: UNIQUE(Kind.AngId, Kind.Vorname) –bei SQL, ON DELETE C.

### Regel 2.4: Abbildung von many-to-many Assoziationen (0..\* zu 0..\*)

mit abhängiger Beziehungs-Tabelle mit zusammengesetztem PK

Student (StudId, Name NOT NULL)

Kurs (KursId, Bezeichnung NOT NULL UNIQUE)

Belegung (*StudId* REFERENCES Student,   
 *KursId* REFERENCES Kurs)

**Regel 2.4a Abbildung von assoziativen Klassen**

Student (StudId INTEGER,   
 Name VARCHAR(80) NOT NULL)

Kurs (KursId INTEGER,   
 Bezeichnung VARCHAR(50))

Belegung (*StudId,**KursId*, GesamtNote DECIMAL)

Pruefung (*StudId* REFERENCES Belegung, *KursId* REFERENCES Belegung, Datum DATE, Note DECIMAL NOT NULL)

Abhängige Beziehungs-Tabelle, erweitert mit den Attributen der assoziativen Klasse.

• Nachteil: Zusammengesetzte Schlüssel können sich über Assoziationen auf andere Tabellen fortpflanzen (Bsp. Tabelle Pruefung)

**2.4b: Abbildung von assoziativen Klassen II**

Assoziative Klasse ersetzen im konz. Modell oder bei der Abbildung auf das DB-Modell

+ keine abhängige Beziehungs-Tabelle Primärschlüssel enthalten keine FK-Attribute

- zusätzliche Unique-Bedingung

Student (StudId, Name)

Kurs (KursId, Bezeichnung)

Belegung (BelId, *StudId*, *KursId*, GesamtNote)

Pruefung (*BelId*, Datum, Note)

Constraints: UNIQUE(Belegung.StudId, Belegung.KursId)

## Regel 3: Abbildung von Vererbungen/Generalisierungen

### Regel 3a je eine Tabelle pro Klasse / eine Tabelle pro Sub- und Superklasse)

fahrzeug (id INT, fzg\_typ INT NOT NULL)

pkw (*fzg\_id* REFERENCES fahrzeug)

lkw (*fzg\_id* REFERENCES fahrzeug, flaeche DECIMAL NOT NULL)

fzg\_typ NULL möglich für incomplete, overlapping möglich

### Regel 3b Eine Tabelle pro Subklasse (Superklassen nicht abgebildet)

pkw (id INT) Schlüssel-Eindeutigkeit separat kontrollieren

lkw (id INT, flaeche DECIMAL NOT NULL) kein overlapping

### Regel 3c Eine einzige Tabelle für Superklasse (Subklassen nicht abgebildet)

fahrzeug (id INT, fzg\_typ INT NOT NULL, flaeche DECIMAL NULL)

overlapping möglich, 3.NF verletzt, viele NULL Werte

# SQL Allgemein (inkl. DDL und DML)

## SQL Queries (DDL)

Reihenfolge im Code

SFW: SELECT field FROM table WHERE field = condition query

WITH – **SELECT** – **FROM** – JOIN – **WHERE** - GROUP BY – HAVING - ORDER BY – LIMIT

SQL Order (mathematische Berechnung im Hintergrund)

FROM + JOIN > WHERE > GROUP BY > HAVING > SELECT (+ WINDOW FUNCTION.) > ORDER BY > LIMIT

LIMIT(x) begrenze auf x Queries

(WITH…) SELECT… FROM… (JOIN…) WHERE… ORDER BY… (LIMIT…)

WHERE-Prädikate

BETWEEN, AND, IN (oder Subquery), OR, < >, =, !=, DISTINCT

Beispiel SELECT name, tel FROM angestellter WHERE tel IS NOT NULL ORDER BY name ASC;

## Datentypen in PostgreSQL (nicht alle)

BOOLEAN (TRUE oder FALSE)

Fliesskommazahl NUMERIC=DECIMAL, z.B. Geld NUMERIC(7,2) *Anz.* *Zahlen vor,nach Punkt*

Ganzzahl INTEGER / INT, BIGINT

Strings TEXT beliebige Länge, VARCHAR(80) maximale Länge, CHAR(3) feste Länge

Zeit DATE Format tolerant ISO (YYYY:MM:DD), TIME Format (HH:MM:SS),   
TIMESTAMP (YYYY:MM:DD HH:MM:SS),   
TIMESTAMPTZ mit TimeZone '2025-01-21 00:00:00+01'  
NOW() Funktion gibt Zeit zur Laufzeit zurück, CURRENT\_DATE(), CURRENT\_TIME()  
INTERVAL Zeitintervalle

VerschiedenesBINARY, BLOB→**BYTEA** (postgres) binäre Datentypen für grosse Daten

ARRAY OF base\_type array\_name TEXT []

Auto-IncrementSERIAL "create table t1 (id serial primary key);"  
GENERATED AS IDENTITY "create table t2 (id integer primary key generated always as identity);"

## Funktionen (modernes SQL)

Concat (concatenation, zusammenführen)

select 'hello' || ' ' || 'world' from table; -> hello world

String functions / Text Manipulation

select LOWER('HelLo') from table; -> hello

select UPPER('HelLo') from table; -> HELLO

select LENGTH('HelLo') from table; -> 5 (Länge von String)

COALESCE(wohnort, 'unbekannt') -> unbekannt statt NULL

**T**ext extrahieren

substring(string [from <str\_pos>] [for <ext\_char>])

**Wildcards**

\_ für 1 Zeichen, % für mehrere Zeichen, oder Regex mit «~»

LIKE '%B%', LIKE '\_B\_'

**Type-Casting**

now()::text, SUM(salaer)::int, CAST(expression AS target\_type)

Numerisch-mathematische Funktionen

SUM(), COUNT(), ROUND(source, anzahl\_dezimalstellen), MOD, TRUNC(num, precision), ABS(num)→Betrag, COS(), SIN(), POWER(), SQRT(), etc.

CASE if/else, ohne else returns NULL  
    WHEN *condition1* THEN *result1*   
    ELSE *result*  
END;

## Datenbank anlegen | Datenbank, Schema, Tabelle, AttrIbute

CREATE DATABASE foo WITH OWNER = 'bar';

CREATE INDEX indexname ON tbl\_name (attr.);

CREATE TABLE fahrzeug (   
 id INTEGER PRIMARY KEY,   
 fzg\_typ INTEGER NOT NULL

);

CREATE TABLE pwk (

fahrzeug PRIMARY KEY REFERENCES fahrzeug(id)

);

CREATE TABLE angestellter (

persnr INTEGER NOT NULL, --Column Constraint (NOT NULL)

name VARCHAR(20) NOT NULL UNIQUE, tel TEXT NULL,

salaer DECIMAL(7,2) NOT NULL,

eintrittsdatum DATE DEFAULT CURRENT\_DATE,

chef INTEGER NOT NULL REFERENCES angestellter(persnr),

--Table Constraints

PRIMARY KEY (persnr, name), -- Two PKs

CHECK (salaer BETWEEN 1000 AND 20000)

);

**ALTER** TABLE angestellter

ADD CONSTRAINT fk\_ang\_chef -- Table Constraint im Nachhinein

FOREIGN KEY (chef) REFERENCES angestellter(persnr)--Foreign Key

ON DELETE CASCADE,

ADD CONSTRAINT alter\_jung\_constraint CHECK (alter < 66);

ALTER TABLE personen RENAME TO mitarbeiter;

ALTER TABLE mitarbeiter [ADD|MODIFY|DROP] COLUMN alter INTEGER;

## DML – Data Manipulation Language

### Datenbank mit Daten füllen / DML

INSERT INTO angestellter (id, name, chef, datum) VALUES (101, ‘Spring’, NULL, DATE '2005-05-05');

INSERT INTO films SELECT \* FROM table2 WHERE id = 2;

INSERT INTO ang (id, name)   
 VALUES (nextval('s\_angid'), 'Müller')   
 RETURNING id; RETURNING auch bei UPDATE möglich

COPY abteilung (abtnr, name) FROM stdin [WITH (ENCODING 'UTF8')];

  Verkauf

  Marketing

\.

### UPDATE Tupel aktualisieren

UPDATE tablename SET *operations* WHERE *conditions*;  
UPDATE angestellter SET salaer=6000 WHERE persNr=101;

### DELETE Tupel löschen

DELETE FROM angestellter WHERE persNr = 1001;

### TRUNCATE, Löschen aller Inhalte der Tabelle

TRUNCATE table1, table2 [ CASCADE | RESTRICT ]; oder nur 1 Tabelle

### DROP TABLE, Löschen der ganzen Tabelle

DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

### Löschen eines Tupels konfigurieren (default: RESTRICT):

ON DELETE CASCADE - Alle Sub-Tupel (FK) werden auch gelöscht  
ON DELETE RESTRICT - Super-Tupel kann nicht gelöscht werden  
ON DELETE SET NULL – Sub-Tupel werden NULL  
ON DELETE SET DEFAULT – Sub-Tupel werden DEFAULT

ON DELETE RESTRICT/NO ACTION: Fehler/Rollback ist default

ON UPDATE – Wie DELETE aber bei Änderung Super-Tupel

## SELECT QUERIES

|  |  |
| --- | --- |
| ANY Mehrere Tupel, Mindestens ein Wert aus Liste  EXISTS Mehrere Tupel, Gelieferte Tabelle nicht 0  ALL Mehrere Tupel, Alle Werte aus der Liste | IN Mehrere Tupel, Gelieferte Liste enthält |

### Subqueries / Unterabfragen, korreliert oder unkorreliert

Korreliert***/***correlated subquery → executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

SELECT a.name FROM angestellter a

WHERE (

SELECT COUNT(pzt.\*)

FROM projektzuteilung pzt

WHERE pzt.persnr = a.persnr -- a = outer query→korreliert

) >= 2;

Unkorreliert/Normal nested subquery → the inner SELECT query runs first and executes once, returning values to be used by the main query. Unterabfrage selbständig/unabhängig.

### Joins

EQUI-JOIN JOIN mit «=», sich selbst, NATURAL JOIN nicht empfohlen

CROSS JOIN Kreuzprodukt (SELECT \* from a,b) *(old join syntax)*

LATERAL JOIN Zweite Tabelle als Subquery

SELECT abtnr, name, maxsalaer FROM abteilung AS abt   
 JOIN LATERAL ( SELECT MAX(salaer) AS maxsalaer FROM angestellter  
 WHERE abtnr=abt.abtnr) **AS ang ON TRUE;**

JOIN= INNER JOIN, mit Bedingung

LEFT/RIGHT OUTER JOINdort wo Bedingung nicht passt = NULL

## Mengenoperationen

UNION / UNION ALL **=** Q1 UNION Q2Fügt zwei Tabellen zusammen, ALL entfernt nicht Duplikate, benötigt genau gleich viel ausgelesene Attribute ('All',)

INTERSECT = Q1 INTERSECT Q2 Durchschnitt zweier Tabellen (EXIST)

EXCEPT Q1 UNION Q2 EXCEPT Q3 Differenz zwei Tabellen (INNER JOIN)

SELECT persnr FROM angestellter a

WHERE a.chef IS NULL INTERSECT

SELECT a.persnr FROM angestellter a

INNER JOIN projektzuteilung p ON a.persnr = p.persnr

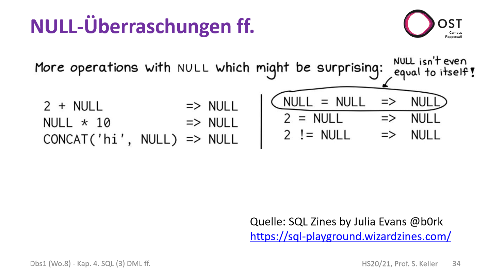
INNER JOIN projekt pr ON p.projnr = pr.projnr

WHERE p.bezeichnung LIKE '%Uranus%'

## Aggregatsfunktionen (GROUP BY) und NULL

MIN(), MAX(), SUM(), AVG(), COUNT()   
nutze HAVING statt WHERE

NULL-Werte werden bei Aggregatfunktionen nicht mit verwendet

SELECT \* FROM table WHERE test != 'hoi' OR test IS **NULL**; 

## Window Functions (zusätzlich zu Aggregatsfunktionen)

Window Function ist abhängig von OVER (PARTITION BY …),

Reihenfolge Window Function ist abhängig von OVER (ORDER BY …), Reihenfolge Ausgabe ist abhängig von ORDER BY

SELECT abtnr, persnr, salaer, RANK() OVER (  
 PARTITION BY abtnr ORDER BY salaer DESC)   
FROM ang ORDER BY 1;

row\_number()Number of current row from its current partition

rank(), dense\_rank(), percent\_rank()Ranking based on order in current partition (dense\_rank is without gaps; percent\_rank is relative rank).

ntile()Returns an integer ranging from 1 to the argument value, dividing the partition as equally as possible.

lag(), lead() access data of the previous row from the current row, nth value relative to current, -nth value relative to current (n defaults to 1) in current partition.

SELECT year, sales, LAG(sales, 1)::INT OVER ( ORDER BY year ) AS previous\_year\_sales FROM sales\_table; LEAD für n folgender Wert

first\_value() / last\_value() / nth\_value() Absolute first/last/nth value in a partition based on order regardless of current position

## Views

Sicherheit: Irrelevante Daten für bestimmte Nutzer entfernen. View ist eine virtuelle Tabelle basierend auf andere Tabellen oder Views. Daten werden zur Ausführzeit aus Tabellendaten hergeleitet. Man kann auch Queries damit vereinfachen.

CREATE VIEW angpublic [(persnr, name, tel, wohnort)] AS

SELECT persnr, name, tel, wohnort FROM angestellter;

### UPDATE möglich, wenn

* Keine Join, Set-Operationen
* Keine Gruppen-Funktionen (min, max)
* Keine GROUP BY, CONNECT BY, START WITH, DISTINCT

### INSERT und UPDATE auf Views

..ermöglichen in PostgreSQL mit **RULE-System** oder **TRIGGERS**.

CREATE RULE teloffice\_insert AS ON INSERT

TO teloffice [ WHERE condition ]

DO [INSTEAD | ALSO] INSERT INTO telverz VALUES (nextval('s\_telid'), NEW.name, NEW.tel, 1);

### Weitere Views

CREATE MATERIALIZED VIEW view2; // speichert Kopie der Query

REFRESH MATERIALIZED VIEW view2;

Row-Level Security (RLS) ist eine Art “System-Views” - Nur User mit entsprecheneden Lese- und Schreibrecht («Policy») auf Rows haben Zugriff

Temporäre Tabellen (CREATE TEMPORARY TABLE):

- Werden gelöscht (dropped) am Ende einer Session oder Transaction

- Andere «permanente» Tabellen mit gleichem Namen sind nicht sichtbar

## CTE (Common Table expression)

Hilfs-Query in einer WITH-Klausel (Temporäre Tabellen während des Statements) Können…

…SELECT, INSERT, UPDATE, DELETE enthalten | …sich auf vorergehende Hilfs-Query beziehen

…anstelle Subqueries verwendet werden | … dem DB-Optimierer helfen und rekursiv sein

### Syntax (Normal)

WITH nameTable1 AS ( SELECT \* FROM myTable )   
SELECT \* FROM nameTable1;

### Rekursiv: zuerst Initialisierung, dann Rekursiver Teil

Von einem Angestellten alle Untergebenen rekursiv auch Unteruntergebene usw.

WITH RECURSIVE untergebene (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 untergebene ORDER BY chef, persnr;

# Transaktionen

Pro Session maximum 1 Transaktion, NESTED Transaktion nicht unterstützt(savepoints)

Fault Tolerance Bei Server-chrash kann Operation wiederholt werden oder wird ganz gecancelt, Write-Ahead Log zur dauerhaftbarkeit nach Commit

Concurrency Isolation der Transaktionen, Parallelität wird ermöglicht

* A*tomicity*: Transaktion vollständig oder gar nicht
* C*onsistency*: Konsistenter Zustand von Daten bleibt erhalten
* I*solation*: Transaktion wie von anderen Isoliert ausgeführt werden
* D*urability* Alle Änderungen sind persistent, bei Fehler nicht verloren

**Gründe für Abort**

* Explizit durch ROLLBACK oder ABORT
* Unzulässige Verzahnung mit anderen nebenläufigen Transaktionen, Deadlock
* Applikationsabbruch, Systemabsturz, Fehler

|  |  |
| --- | --- |
| BEGIN [TRANSACTION];  COMMIT [TRANSACTION];  ROLLBACK [TRANSACTION]; | SAVEPOINT transaktionsname;  ROLLBACK TO transaktionsname; |

## A picture containing shape Description automatically generatedSerialisierbarkeit

wenn, nebenläufige Ausführung, gleich wie serielle Ausführung

Muss azyklisch sein -> Serialisierbarkeit keine Schlaufen

Beispiel S = r1(x) r1(y) r2(x) r2(y) w1(x) w2(y) c1 c2

Konfliktpaarer1(x) < w1(x), r1(y) < w2(y), r2(x) < w1(x), r2(x) < w2(x)

Zyklus Serialisierbarkeitsgraph→Nicht serialisierbar | *2x gleiches Paar und gleiche Zahl mögl.*

Topologische Sortierung (Halbordnung) bestimmt **Commit-Reihenfolge** (T1→T2 =>c1→c2)

### Implementation der Isolation

|  |  |
| --- | --- |
| **Pessimistische Verfahren (2PL)**   * Sperrprotokolle * Besser bei hoher Konflikt-Wahrscheinlichkeit   1.Phase Growing 2.Phase Shrinking | **Optimistische Verfahren (Isolation, SI, SSI)**   * Konfliktbehebung im Nachhinein * Besser bei kleiner Konfliktwahrscheinlichkeit * Abbruch bei Konflikt |
| **Commit Resultate:** | |
| Success Änderungen atomar und durable gespeichert | FailureAlle temporären Änderungen werden abgebrochen (abort) |

### Locking

unlock(x), Gibt lock wieder frei (*Wichtig richtiger Buchstabe*)

Exclusive Lock (X) xlock(x)

* Für Schreibe- oder Lesezugriffe, Nur eine Transaktion *xlock*

Shared Lock (S) slock(x)

* Nur für Lese-Zugriffe, Mehrere Transaktionen *slock*

## Isolation Levels

BEGIN;

SET TRANSACTION ISOLATION LEVEL [SERIALIZABLE|READ UNCOMMITTED|..];

Serializable ist am besten, Parallelität limitiert, Effizienz mit schwächeren Levels gesteigert, auf Kosten Korrektheit. Fehler schwer nachvollziehbar

|  |  |
| --- | --- |
| 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  Vollständig (korrekte) Isolation ACID |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Read Uncommited | Read Committed | 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 |
| Write Skew | möglich | möglich | möglich | möglich\* |
| Deadlock |  |  | möglich | unmöglich |
| Cascading Rollback |  |  |  | unmöglich |
| Serialization Anomaly | möglich | möglich | möglich | unmöglich |

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

Dirty Read (auch Read Skew) – Daten Lesen von anderer nicht committed T

Fuzzy Read = nonrepeatable read = inconsistent analysis – Lese gleiche Daten mehrmals, sehe aber andere Werte, gelesene Daten ändern sich durch andere T

Phantom Read – Gleiche SELECT neue/gelöschte Rows

Serializable Kann r, w Konfliktpaare blockieren, (bei Postgresql mit SSI)

Read Committed Kann w, w Konfliktpaare blockieren

## Concurrency Control Techniques MVCC

Postgresql nutzt haupstächlich MVCC (Multi-Version Concurrency Control) = SnapshotIsolation und unterstützt SSI Serializable Snapshot Isolation und OCC (Optimistic Concurrency Control)

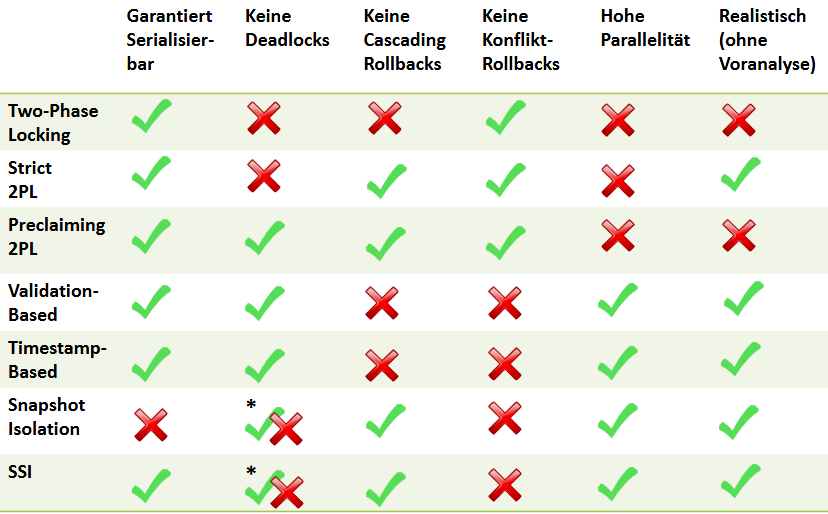
### Grundlagen und Prinzip Multi-Version Concurrency Control (MVCC)

* Mehrere Versionen pro DB-Objekt
* Jede T hat Timestamp des Startzeitpunkts
* write(x): neue Version xt, t = Start-TimeStamp
* read(x): letzte Version xt, mit grösst. t ≤ Start-TimeStamp

PostgreSQL unterstützt MVCC

* Verfahren:
* Update -> Tupels X-Lock (row locking) -> Deadlocks möglich
* Lesen -> keine Locks, nicht überprüft
* Jedes Update führt zu neuer Version des Tupels

Verhalten

* READ COMMITTED: read nur mit derselben Verison (dort implementiert PostgreSQL SI)
* REPEATABLE READ od. SERIALIZABLE: ganze T nur mit derselben Version
* Serialisierbar nur mit Level SERIALIZABLE

\* Deadlock in PostgresSQL mit Snapshot Isolation

XLOCK = upgraded lock, SLOCK = shared lock (bei read)

Lösung gegen Deadlocks: Preclaiming Two-Phase Locking.

# Sicherheit (inkl. SQL DCL = Rechteverwaltung)

## Benutzerrechte

Bei PostgreSQL hat jeder Benutzer die Summe der folgenden Rechte  
- alle Rechte, die ihm selbst zugeteilt wurden

- alle Rechte, mind. einer Gruppe / Rolle zugeteilt wurden, in der er Mitglied ist

- alle Rechte der Pseudo-Gruppe/Pseudo-Rolle public

### Privileges for new user / group

CREATE ROLE user\_or\_group [WITH LOGIN PASSWORD '123']

WITH [CREATEDB|NOCREATEDB] | [CREATEROLE|NOCREATEROLE];

CREATE ROLE group; standard = CREATE auf public schema

DROP USER IF EXISTS angguest;

CREATE ROLE angguest WITH LOGIN PASSWORD 'angguest' IN ROLE group;

REVOKE CREATE ON SCHEMA public **FROM** public; wichtig!

GRANT [ALL|SELECT|CREATE|INSERT|UPDATE]

ON [TABLE|SCHEMA] angpublic,projekt **TO** angguest;

REVOKE ALL ON SCHEMA angpublic FROM u;

|  |  |
| --- | --- |
| ALTER ROLE user RENAME TO user2;  SET ROLE bob; Benutzer wechseln  DROP ROLE angguest; | RESET ROLE; zurückwechseln |

Systemprivilegien für Datenbank-Operationen & Systemvariablen

CREATEDB, CREATEROLE, NOCREATEDB, NOCREATEROLE

ALTER ROLE username WITH CREATEROLE;

current\_timestamp TIMESTAMP, current\_user ROLLE, session\_user

Nur selbst erstellte Einträge sehen/create.. bei GRANT

CREATE POLICY policy\_teachers\_see\_own\_exams ON exams FOR ALL TO PUBLIC USING (teacher\_pguser\_attr = current\_user);

ALTER TABLE exams ENABLE ROW LEVEL SECURITY;

# Indexe und Speicherstrukturen + Optimierung

B-tree (universell), B+ Baum (verkettet, Blatt zeigt auf nächstes Blatt und d=Daten sind ausserhalb der Blätter im Heap gespeichert), Hash (equality search WHERE =), GiST (Array, Volltextsuche), SP-GiST (knn (nearest-neighbor), geometrisch), GIN (schneller aber mehr Daten als GiST, wenig ändern!), BRIN (Range Search), ISAM (wie hash aber kann mehr)

DROP INDEX [IF EXISTS] indexname;

### zusammengesetzte Schlüssel

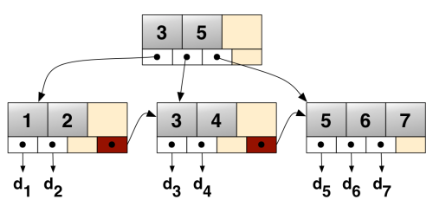
CREATE INDEX mytable\_col12\_idx ON mytable (col1, col2);

### Index mit Include

CREATE INDEX magic\_idx ON test (nr,id) **INCLUDE (txt**);

### Funktionaler Index

CREATE INDEX mytable\_col\_part\_idx



B+-Baum

ON mytable ( UPPER(col) ); UPPER nur als Beispiel

### weitere Index-Typen

CREATE INDEX mytable\_col\_idx

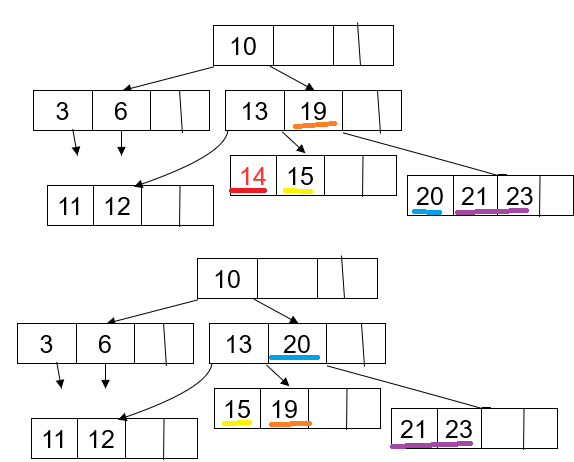
ON mytable USING **btree** (col);

CREATE EXTENSION btree\_gist;

CREATE INDEX mytable\_col\_idx2

ON mytable USING **gist** (col);

CREATE UNIQUE INDEX name ON table (attribut) not null, unique

CLUSTER angestellter USING angestellter\_pkey;

CREATE INDEX mytable\_col\_part\_idx ON mytable (col)

WHERE archived IS NOT NULL; partieller Index

B-Baum mit m=2, k=4

### Beispiel B-Baum Lösche die 14

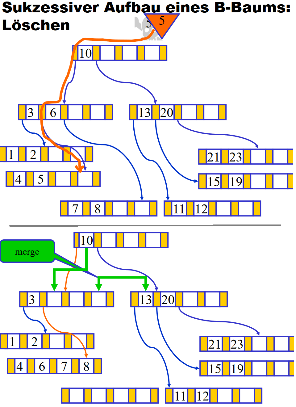
1. Unterlauf bei 14

2. Nachrücken /Verschiebung =

merge [15, 19, 20, 21, 23]

 in Mitte aufteilen (bei 20)

Beispiel 2, lösche die 5

1. Unterlauf bei 5

2. merge [4, 6, 7, 8]

3. Unterlauf bei Vater **3**

4. merge Elternknoten mit Kind von rechts [3, 10, 13, 20]

5. Schrumpfung freier Knoten (leere Knoten löschen)

### Einfügen (Ausschnitt)

Knoten/Blatt überfüllt>aufteilen wie folgt:

1. neuer Knoten mit Keys rechts vom mittlerem Key (/rechts von Mitte)

2. mittl.Key (/links von Mitte) = Vaterknoten (Rechtsbewegung)

3. Wenn Vaterknoten überfüllt →nochmals

- falls Wurzel → neue Wurzel (Baum wächst)

- wiederholen

