# SQL Grundlagen DBS1 (DDL und DML)

Jasmin Fässler 2025

SELECT [DISTINCT] ... FROM ... [JOIN . ON .]   
WHERE ... ORDER BY ... [LIMIT .];

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

**Common Table Expression Rekursiv**

Von 1 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;

## modernes SQL

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

LOWER('HelLo') UPPER('HelLo') LENGTH('HelLo'))

COALESCE(wohnort, 'unbekannt')

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

substr(name, 1, position(',' IN name) - 1) AS nachname

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

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

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

CASE (ohne else returns NULL)  
    WHEN *condition1* THEN *result1* ELSE *result* END;

## Datenbank anlegen, Schema

CREATE TABLE fahrzeug (   
 id bigint PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
 fzg\_typ INTEGER NOT NULL UNIQUE DEFAULT 0,

fahrzeug PRIMARY KEY REFERENCES fahrzeug(id),

CHECK (salaer BETWEEN 1000 AND 20000) *--ist inclusive*

);

ALTER TABLE personen RENAME TO mitarbeiter;

ALTER TABLE mitarbeiter [ADD|MODIFY|DROP] COLUMN   
 col INTEGER default 0;

ALTER TABLE mitarbeiter ALTER col SET DEFAULT 0;

ALTER TABLE mitarb ALTER col SET DATA TYPE int;

## DML – Data Manipulation Language

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

INSERT INTO films SELECT \* FROM table2 WHERE id = 2;  
UPDATE angestellter SET salaer=6000 WHERE   
 persNr=101;

DELETE FROM angestellter WHERE persNr = 1001;

TRUNCATE table1, table2 [ CASCADE | RESTRICT ];

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

# Object relational Mapping

Softwareprogramm→speichern (persistieren) und abfragen von DBMS

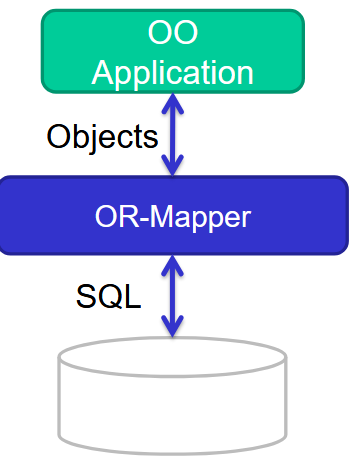
Brücke zwischen Semantik von Software und Datenbank (Datentypen, relational zu objektorientiert)

Programm objektorientiert, Klassen, Objekte & Java Datentypen

DB relational, Tabellen, Tuples, Beziehungen & PostgreSQL Datentypen

Semantische Lücke Programme sind (meist) objekt-orientiert, Datenbanken sind (meist) relational und haben andere Datentypen.

### O/R Mapping Varianten (Lösung für semantische Lücke)

Java Data Objects: JDO, OO-Kompatibilität besser als JPA, kein Lazy-Loading, beliebige Datenquellen

jOOQ: lightweight, SQL-based, type-safe Queries

Java Jakarta Persistence API (JPA): EclipseLink, Hibernate (JPA/Enterprise Java Beans 3.0 konform)

### Funktionen von OR-Mapper

- Laden von relationale Daten als Verbund von Objekten

- Speichern von Änderungen an persistenten Objekten

### Wichtigste Begriffe

Entity = Tabelle, Klasse und Entity Instance = Zeile von Tabelle,

Entity Manager = API für die CRUD Befehle

Persistence Context = Menge von Entity Instances, darin werden Entity Instances und ihre Lifecycles von der Application verwaltet (Entity States)

JPQL = Java Persistent Query Language

### Wie kommen Entities und Daten zusammen?

**Top down (Forward Engineering)** Erstelle Business-Modell und erzeuge DB-Schema | **Bottom up (Reverse Engi.)**  DB-Schema existiert; erzeuge daraus Business-Modell (verwendet) | **Middle out (Mapping First)** Erstelle Metadaten und generiere Java und DB-Schema | **Meet in the middle** Business-Modell und DB-Schema existieren bereits: Erstelle Metadaten

**Meta Model / Model driven** (selten und für uns nicht so wichtig)

# jPA

## Regeln für Entities (Entitätsklasse) (JPA)

Java-Klasse (class) mit Annotation @Entity

– Kann erben und vererben – Kann Interfaces implementieren

– Kann "abstract" sein

Einschränkungen

– muss Public- oder Protected-Konstruktor ohne Argumente geben

– Identity-Angabe erforderlich (Annotation @Id)

– Klasse ist nicht final, keine final Fields oder Methoden (Reflection)

– Zudem: Fields sollen private oder protected sein (getter/setter)

Mapping Regeln case insensitiver, gleicher Name für Tabellen&Columns, alle Attribute persistent gemappt (ausser @Transient), Calendar/Date mit @Temporal als Abbildung auf DATE TIME oder TIMESTAMP

Unterstützte Typen Primitive Typen+Wrappers, Strings, Enums, Byte/Char Arrays, Date/Calendar, beliebige Serializable Klassen (BLOB)

Relationen Referenzen auf Instanzen mit Entity Klassen, Collection<>, Set<>, List<>, Map<> von Entities

Field Access (Attribute direkt in Fields) (Standard)

Alternative: Property Access (Attribute über Get/Set) Annotat. vor getId()  
private long id; @Id public long getId(){}; setId(longid) { }

Annotations bei getters, somit getters/setters Pflicht

## Persistence Unit und Entity ManagerAPI

Persistence.xml beschreibt JPA Persistence Unit

EntityManager API verwaltet Persistency, Context & bietet Lifecycle-Operationen für Entity Instanz an

persist() Make an instance managed and persistent

remove() remove entity instance

refresh() Refresh state of the instance from the database, overwriting changes made to the entity, if any. (DB → managed)

merge() Merge state of given entity into the current persistence context. (detached →managed)

find() Execute a simple PK query

flush() Force synchronization of persistence context to database

clear() Clear persistence context, managed + removed→detached

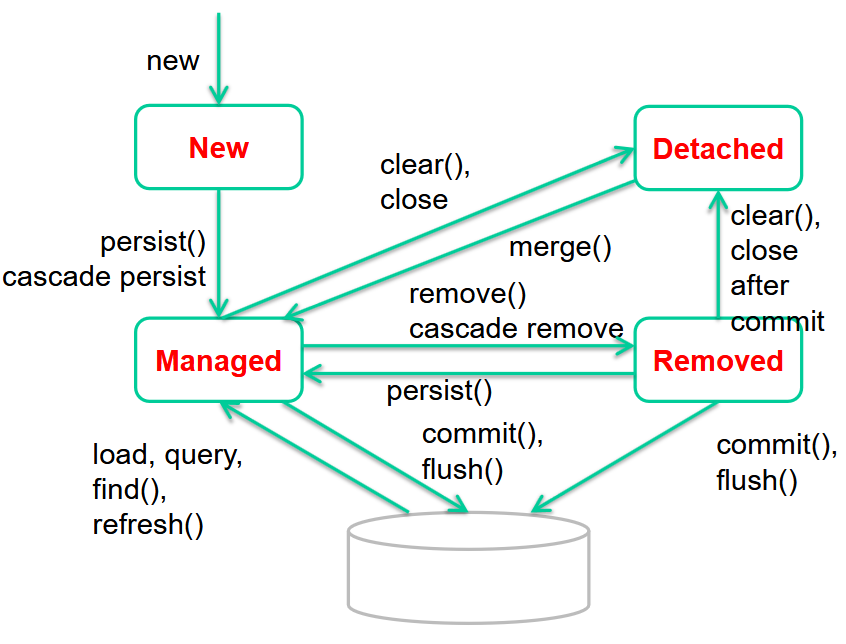
createQuery() Create query instance using dynamic JP QL

createNamedQuery() Create instance for a predefined query

createNativeQuery() Create instance for an SQL query

contains() Determine if entity is managed by persistence context

## **Entity States** (Lebenszyklus Entity Instanz)



New no persistent identity, not associated with persistency context.

Managed persistent id and associated with a persistency context.

Detached persistent identity and not associated, on cache

Removed persist. id, associated, scheduled removal data store. (wie managed aber für die Entfernung aus dem Datenspeicher vorgesehen)

### EntityManagerFactory verwaltet Persistency Unit

Setup (em instanziieren)

*//* Bank *ist der Name der Unit in persistence.xml*

private static EntityManagerFactory factory;

factory = Persistence.createEntityManagerFactory("Bank");

*Neuer Persistence Context (Session):*

EntityManager em = factory.createEntityManager();

try {

em.getTransaction().begin();

*// hier den* Code *von unten einfügen, mache etwas mit em*

em.refresh();

em.getTransaction().commit();

} catch (Exception e) {

em.getTransaction().rollback();

} finally { em.close();

}

//Insert (Code für oben in try Klammer)

BankCustomer customer = new BankCustomer();

customer.setName("Bill");

em.persist(customer); *// PK automatisch serial typ*

// Update, bearbeiten

BankAccount account = em.find(BankAccount.**class**, 1L); *// PK*

account.incBalance(100); *// updates mit Objekt Methode*

//Delete

BankAccount account = em.find(BankAccount.class, 1L);

em.remove(account); eplizit aus Persistenz (dh. DB) entfernen  
*// Alternativ Persistenz durch Erreichbarkeit, Cascade-Option*

//JPQL

Query query = em.createQuery(

"SELECT a FROM BankAccount a"); *// JPA Query Language*

List<BankAccount> list = query.getResultList();

for (BankAccount account : list) {System.out.println(account);}

## Entity Klasse Annotationen

@Entity // persistierbare Klasse

@Table(name = "account") // Name der DB-Tabelle

public class BankAccount {

@Id// Primary Key attribute id

@Column(name = "accountid") // id name aus DB  
 @GeneratedValue(strategy=GenerationType.IDENTITY)

// weitere GenerationType = AUTO, SEQUENCE, TABLE

private long id;

private String name;

private double balance;

@Temporal(TemporalType.TIMESTAMP)

private Calendar creationDate;

private LocalDate birthdate; // ohne Annotation

// Erstellen: LocalDate.of(2020, Month.MARCH, 8)

@Enumerated(EnumType.STRING)

private Currency curr;

@Transient // nicht persistent

private String tempComments;

/\* getter und setter, z.B.: \*/

private long getId() { return id; }

public void setName(String name) {

this.name = name; *// POJO*

} *// public String toString() {} überschreiben*

}

### @Column() Params

Allgemein→ @Column(name="…", unique=true, nullable=true)

private String lastName → length=200   
private BigDecimal salary → scale=10, precision=2

private Calendar nameDate → columnDefinition="TIMESTAMPTZ"

## JPA Entität als CLASS mit Annotationen

### Persistency Context (verwaltet Entity Instanzen zur Laufzeit)

Managed Entities sind in Persist. context + definiert transaktion. Session

Objekte werden nicht automatisch persistiert, entweder explizit über em.persist() / em.remove() oder implizit über @XtoX(cascade = CascadeType.PERSIST) / @XtoX(cascade = CascadeType.REMOVE)

### Ladestrategien

Eager Loading Target Entity direkt mit Beziehung laden, Default bei @OneToOne & @ManyToOne

Lazy Loading Target Entity bei 1. Beziehungs-Zugriff laden, Default bei @OneToMany & @ManyToMany

(Wann es die verknüpften Assoziationen lädt. Eager = kein n+1 Fehler aber es wird mehr im voraus geladen was eventuell gar nicht benötigt wird.)

@OneToOne(fetch = FetchType.LAZY)

@OneToMany(fetch = FetchType.EAGER)

### GenerationTypes für @GeneratedValue siehe Code

IDENTITY (increment von DB), AUTO (JPA wählt aus), TABLE, SEQUENCE

## JPA Bidirectional sync

public class BankAccount { …

public void setCustomer(BankCustomer newCust) {

BankCustomer oldCust = this.customer;

this.customer = newCust;

if(newCust != null && !newCust.containsAccount(this)) {

newCust.addAccount(this);

}

if (oldCust != null && oldCust.containsAccount(this)) {

oldCust.removeAccount(this);

} }…}

public class BankCustomer { …

public void addAccount(BankAccount account) {

this.accounts.add(account); *// könnte NULL Check machen*

if (account.getCustomer() != this) {

account.setCustomer(this);

} }…}

# Beziehungen zwischen Entities

(Annotationen/XML nötig)

Man beachte, dass sich bestimmte Annotationen

– auf Entity/Field/Java-Klassen (hier z.T. Mixed Case) –

oder auf Tabelle/Column beziehen (hier kleingeschrieben).

– auf "eigene" Felder oder Columns beziehen – oder auf andere

Annotationen wie @Entity (@Entity in den Codes↓ nicht vergessen!)

### OneToOne (1 : 1) und (1..1 : 0..1) und (0..1 : 0..1)



public class BankCustomer {

@OneToOne(optional=true) // true ist default

@JoinColumn(name="addressid") // FK bei BankCustomer

private Address address; ...

### Bidirektionale OneToOne (1..1 ↔ 0..1)

... OneToOne BankCustomer +

public class Address {

@OneToOne(mappedBy="address", optional=false)

private BankCustomer customer; // andere Entität

### Inverse OneToOne (0..1 ← 1..1)

public class Address {

@OneToOne

@JoinColumn(name="customer\_address\_id",   
 referencedColumnName="addr..id", insertable=false,   
 updatable=false)

private BankCustomer customer; ...

### ManyToOne-Beziehung (BankAcc. 0..\* → 1 BankCustomer)



public class BankAccount {

@ManyToOne(optional=false)

@JoinColumn(name="customerref") // FK

private BankCustomer customer8; ...

### ManyToOne Bidirektional (BankAcc. 0..\* ↔ 1 BankCustomer)



ManyToOne BankAccount + BankCustomer

public class BankCustomer {

@OneToMany(mappedBy="customer8",fetch = FetchType.EAGER)

private **Collection**<BankAccount> accounts = new ...;

### OneToMany (BankCustomer 1..1 → 0..\* BankAccount)



public class BankCustomer {

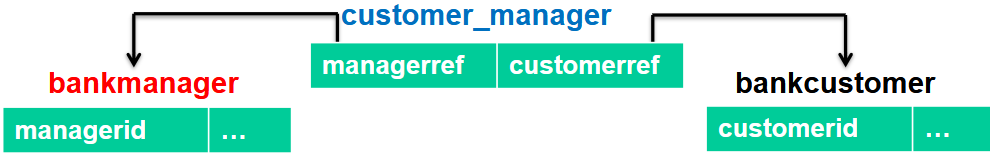
@OneToMany

@JoinColumn(name="customerref", von BankAccount Table

referencedColumnName="customerid") von BankCustomer

private **Collection**<BankAccount> accounts = new ...;

### ManyToMany (0..\* : 0..\*) Zwischentabelle nicht separat nötig



public class BankManager {

@ManyToMany

@JoinTable(name="customer\_manager",

join[Columns={@JoinColumn](mailto:Columns%3D%7b@JoinColumn)(name="managerref")},

[inverseJoinColumns={@JoinColumn](mailto:inverseJoinColumns%3D%7b@JoinColumn)(name="customerref")}

) private **Collection**<BankCustomer> customers =

new ArrayList<>(); ...

### ManyToMany Bidirektional (0..\* : 0..\*)

Public class BankCustomer { // map Inverse zu Bankm.custs

@ManyToMany(mappedBy="customers",fetch = FetchType.EAGER)

private **Collection**<BankManager> managers=…;

JPA Vererbungen

Superklasse bei SINGLE\_TABLE und JOINED:

@Entity

**@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)**

@**DiscriminatorColumn**(name = "type") *Typ Diskriminator*

public **abstract** class BankCustomer {

@Id private String name; // *bei SINGLE\_TABLE*

@Id private int customerId; // *bei JOINED PK*

}

Subklassen bei SINGLE\_TABLE und JOINED:

@Entity

@**DiscriminatorValue**("retail") (kleinschreiben wie SQL)

class RetailBankCustomer extends BankCustomer {

private int fees; }

Superklasse bei TABLE\_PER\_CLASS

@Entity

**@Inheritance(strategy= InheritanceType.TABLE\_PER\_CLASS)**

public class BankCustomer { *// ohne abstract*

@Id private int customerId;

private String name; }

Subklasse bei TABLE\_PER\_CLASS

@Entity

class RetailBankCustomer extends BankCustomer {

private int fees; }

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

fahrzeug (id INT, fzg\_typ TEXT NOT NULL DEFAULT 'typ1')

pkw (*fzg\_id* REFERENCES fahrzeug)

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

fzg\_typ NULL möglich für incomplete, overlapping möglich

JOINED PK von Subklassen ist FK auf PK von Superklasse, JPA: DiscriminatorColumn / DiscriminatorValue wie oben, keine @Id bei Subklassen.

**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

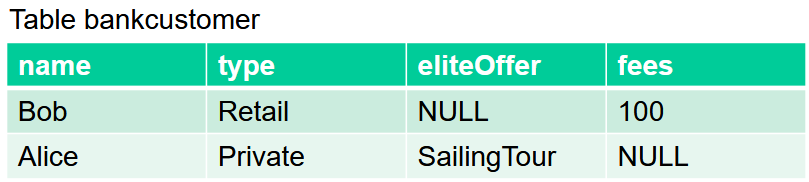
TABLE\_PER\_CLASS Table pro Sub- & nicht-abstrakter-Superklasse, PKs von Subklassen sind keine FKs, alle Attribute von Superklasse sind auch in Subklassen-Tabelle vorhanden

keine DiscriminatorColumn / DiscriminatorValue / @Id bei Subklassen

Mapped Superclass (seltener) wie Table per class, aber ohne Tabelle für Superklasse, inkl. Vererbte Columns. Wirklich wie 3b.

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

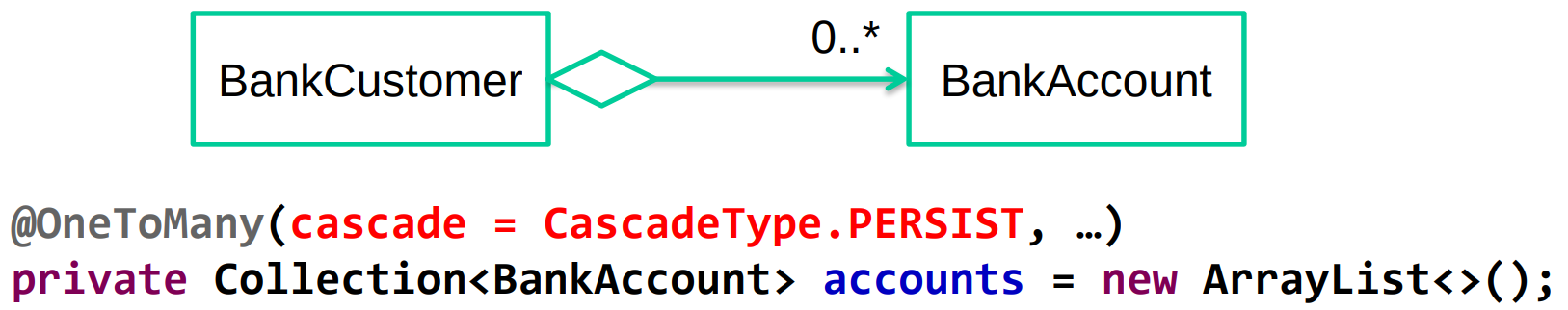
fahrzeug (id INT, fzg\_typ INT NOT NULL, flaeche DECIMAL NULL) overlapping möglich, 3.NF verletzt, viele NULL



SINGLE\_TABLE (default JPA) 1 Table mit Attributen aller Subklassen

Bei JPA mit Discriminator Column. Keine @Id bei Subklassen.

Zusätzliche Möglichkeiten: @SecondaryTable: Beliebig viele Tabellen können auf eine einzige Klasse abgebildet werden. @Embedded: Mehrere Klassen können zusammen auf eine Tabelle abgebildet werden.



# JPQL Abfragen

Anfragesprache in Analogie zu SQL, aber: Operiert auf EntityModell, nicht DB-Modell

Dynamic Query (Query zur Laufzeit gebaut und geprüft)

Query query = em.createQuery("SELECT c FROM BankCustomer c   
 WHERE c.name LIKE **:customerName**");

query.setParameter("**customerName**", "Donald Duck");

query.setMaxResults(1000);

List<BankCustomer> list = query.getResultList();

### Mögliche Queries

select a from BankAccount a // Alias, BankAccount=Classname JPA

where a.accountid = '06543'; // Path expression

// oder where a.balance >= 0 and a.balance <= 1000

select a from BankAccount a **order by** a.balance desc;

select distinct a.id, a.balance from BankAccount a;

//Projektion (Vector) ↑

select c from BankCustomer c where c.address is null

is not empty //null referenz oder not null

select m from BankManagerm where 'Bob' in

(select c.name from m.customersc) //subquery

select a from BankCustomer c **join** c.accounts a

select c.name, sum(a.balance) from BankCustomer c

join c.accounts a **group by** c.name **having** a.balance>= 1000

// Java Ausgabe :

List<**Object[]**> resultList = query.getResultList();

for (Object[] result : resultList) {

System.out.println(result[0] + "\t\tKontostand: " + result[1]);

}

### Positional Parameters (zuerst Query dann Parameter)

select a from BankAccount a where

a.customer.name like **?1** and a.balance >= **?2**

//.name -> l-join/PathExpression

query.setParameter(1, "Bob") *// Parameter setzen*

Named Parameters

select a from BankAccount a where a.customer.name like **:name** and a.balance>= **:lower**

query.setParameter("lower", 100) *// Parameter setzen*

### Named Query

(statisch bekannt, von JPA vorgeparst und optimiert)

@NamedQuery(name = "CustomerSearch", query =

"SELECT c FROM BankCustomerc WHERE c.name LIKE :customerName")

@NamedQuery(name = "Angestellter.hasChef", query = "SELECT a FROM Angestellter a WHERE a.chef IS NOT NULL")

@Entity

public class BankCustomer{ … }

// um NamedQuery abzufragen

Query query= em.createNamedQuery("CustomerSearch");

query.setParameter("customerName", name);

var result = query.getResultList();

# Sicherheit (inkl. SQL DCL = Rechteverwaltung)

## Benutzerrechte

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

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;

DROP ROLE angguest;

Systemprivilegien für Datenbank-Operationen & Systemvariablen

CREATEDB, CREATEROLE, NOCREATEDB, NOCREATEROLE

ALTER ROLE username WITH CREATEROLE;

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

## Row Level Security / Views

Nur selbst erstellte Einträge sehen

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;

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

Sicherheitsüberprüfungen/Auditing mit Triggers

Zugriffsschutz mit Stored Procedures

Client modifiziert über den Aufruf von Stored Procedures die Daten. Stored Procedures kapseln die Daten und können Teile der Geschäftslogik implementieren. Client benötigt keine Privilegien für die Modifikation der darunterliegenden Tabelle.

### Beispiele

Alle dürfen ihre eigenen Daten ansehen: GRANT SELECT ON Angestellter\_V TO PUBLIC;

Abteilungsleiter dürfen die Daten ihrer Mitarbeiter einsehen: GRANT SELECT ON AbtAng\_V TO AbtLeiter\_R;

PersonalChefs dürfen Salärerhöhungen ausführen: GRANT EXEC ON SalaerErhoehung TO Personal-Chef\_R;

### Zugriffsschutz mit Views

Mit Views können vertrauliche Daten geschützt werden: Zuerst wird eine Sicht mit den öffentlichen Attributen definiert (vertikale Filterung), dann wird diese mit der Grant Anweisung allen Benutzern zugänglich gemacht:

CREATE VIEW AngPublic (Persnr, Name, Tel) AS

SELECT Persnr, Name, Tel FROM Angestellter**;**

GRANT SELECT ON AngPublic TO PUBLIC;

# STORED Procedure (PL/pgSQL)

### Rechte bei Stored Procedure

GRANT EXECUTE ON storedprocedurefunction TO anguser;

### Möglichkeiten

INSERT SELECT immer mit INTO!

SELECT .. INTO STRICT muss genau ein Tupel liefern, Exceptions: NO\_DATA\_FOUND, TOO\_MANY\_ROWS

Verstoss gegen Eindeutigkeit eines Schlüssels: UNIQUE\_VIOLATION

Alle anderen Exceptions: OTHERS

Variablen FOUND, ROW\_COUNT, FOUND : bei update

Exceptions (nur in EXCEPTION Block) division\_by\_zero,

angnr IN INT (call by value) angnr OUT INT (call by reference)

Generische Datentypen: anyelement, anycompatible, anyarray

### Operator <> not equal to

Anstatt IF NOT … = … , besser so schreiben: …. <> …..

### Eigenschaften Stored Procedures

Procedures können Transaktionen commit und rollback, CALL

Functions Rückgabetyp wichtig, mit SELECT, INSERT, UPDATE aufrufbar

Konsistenz Trigger Security. Separation of Concerns: Konsistenzprüfung 1x im Backend statt überall im Frontend

SQL/PSM = SQL Persistent Stored Modules

PL/SQL Oracle-spezifisch. Am Nächsten zum SQL/PSM Standard

PL/PGSQL Case sensitive; no defaults; no need for cursors; Function Overloading

UDF User Defined Functions können an jeder Stelle im SQL aufgerufen werden wo eingebaute Funktionen stehen, z.B. «upper(TEXT)»

## Codes Bausteine PL/PGSQL

DROP FUNCTION name(argtype [,…]); *echo(anycompatible);*

COMMENT ON FUNCTION increment(int) is 'Increment 1';

IF expr THEN ... ELSIF other-expression THEN ... ELSE ... END IF;

CASE x when 1, 2 THEN ... END CASE

**FOR** record-variable **IN** query **LOOP**  
 -- statements  
**END LOOP**;

EXIT, CONTINUE, WHILE, FOR

SELECT .. INTO STRICT muss genau ein Tupel liefern

Anz. Tupel > 1: Exception TOO\_MANY\_ROWS

Anz. Tupel = 0: Exception NO\_DATA\_FOUND

Ohne STRICT: erstes Tupel des Resultsets bzw. NULL

### Deklarationen von Variablen Möglichkeiten

DECLARE *-- optionaler Teil*

var1 integer;

var2 integer not null;

var3 integer default 0;

var4 varchar := 'starting text';

var5 angestellter.id**%TYPE**; *Column Attribut Typ*

var6 angestellter**%ROWTYPE**; *Tupel mit Datentypen der Zeile*

var7 **record**; *-- Referenz auf eine Tabelle (Pointer)*

var8 any element; -- generischer Typ gemäss Fn.-Argument

Datentypen in PL/PGSQL

integer, text, numeric, boolean, timestamp, date, bigint, smallint, character, interval, json, xml, uuid, bytea, array

### Parameter

IN call by value, Variablen oder Ausdrücke als Argument.

OUT call by reference, nur Variablen als Argument. INOUT beides

Return Werte void → RETURNS VOID, Scalar → RETURNS DECIMAL,

Tabellen → RETURNS TABLE (abtname VARCHAR, abtMA VARCHAR)

RETURN QUERY / Set → RETURNS SETOF foo, RETURN NEXT r;

### Stored Procedure Vorlage

CREATE [OR REPLACE] [FUNCTION | PROCEDURE] name (

[ [argname] [IN/OUT/INOUT] argtype[,…] ] )

[ RETURNS rettype | RETURNS SETOF RECORD *(SRF)* ]

LANGUAGE [plpgsql | SQL | …] [optimizers]

AS $$

BEGIN  *-- source code gemäss "language"*

END; $$;

### Anwendungsbeispiele (verschiedene Codeschreibweisen möglich)

create [or replace] function hello (**var1** anycompatible)

returns void as $$

begin

raise notice 'Hello World!'; *(ist wie console\_log)*

end;

$$ language plpgsql; *$$ = Signatur für body*

SELECT hello('test'); *-- im tab messages/log zu sehen ‘Hello World! test’*

### PowerModulo Function

CREATE OR REPLACE function power\_modulo (bign bigint, n bigint, m bigint )

RETURNS bigint AS $$

DECLARE

x bigint = 0; xx bigint;

BEGIN

if n = 0 then

return 1;

end if;

x := bign % m; xx := (x \* x) % m;

if n % 2 = 0 then

return power\_modulo(xx, n/2, m);

else

return (x \* power\_modulo(xx, (n-1)/2, m)) % m;

end if;

END;

$$ LANGUAGE plpgsql;

select power\_modulo(2, 4, 5); *-- Erwartetes Ergebnis:* *(2^4) % 5 = 16 % 5 = 1*

## Exception Handling

RAISE level 'format' [. expression [, ...]]

Levels DEBUG, LOG, INFO, NOTICE, WARNING

BEGIN

z:= x / y; *-- RAISE NOTICE 'notice %‘, id;*

EXCEPTION

WHEN division\_by\_zero THEN z:= 0;

END;

DECLARE

angnr angestellter.persnr%TYPE;

*-- Typ von angestellter.persnr*

BEGIN */\*lokaler, anonymer Block \*/*

SELECT angestellter.persnr **INTO STRICT** angnr

FROM angestellter

WHERE angestellter.name = 'Marxer, Markus';

RAISE EXCEPTION 'Text'; *eigene Exception*

EXCEPTION */\*Exception-Handler des lokalen Blocks\*/*

WHEN **NO\_DATA\_FOUND** THEN

*/\*System Exception: SELECT INTO liefert keinen Wert\*/*

RAISE;

WHEN **TOO\_MANY\_ROWS** THEN   
 */\*SELECT INTO\* liefert mehr als einen Wert\*/*

RAISE;

WHEN **others** THEN

RAISE;

END;

## Insert

DECLARE

PNr Projekt.ProjNr%TYPE;

AngNr Angestellter.PersNr%TYPE;

ProzAnt DECIMAL,

BEGIN

INSERT INTO ProjektZuteilung

VALUES (AngNr, PNr, ProzAnt, NULL, NULL);

EXCEPTION

WHEN unique\_violation THEN

*/\* Projektzuteilung existiert bereits\*/*

END;

## Update

BEGIN

UPDATE angestellter SET salaer = salaer + SalIncr

WHERE name = AngName;

IF NOT FOUND THEN *//found= true falls bearbeitet*

RAISE NOTICE 'Ang % existiert nicht', AngName;

END IF;

## Ausführen, entwickeln und Testen von SP

EXECUTE '<<SQL-Abfrage>>';

– innerhalb von PL/pgSQL verwendet, um eine dynamische SQL-Abfrage auszuführen. Dynamisch bedeutet, dass die SQL-Abfrage als String konstruiert ist und zur Laufzeit geändert werden kann.

EXECUTE 'SELECT 1 + ' || i into result;

Table return:

RETURNS TABLE ( abtname VARCHAR, abtMA VARCHAR)

PERFORM <<eine\_Funktion\_oder\_Prozedur>>;

– wird in PL/pgSQL verwendet, um eine Funktion oder Prozedur mit ihren Nebeneffekten auszuführen, ohne notwendigerweise ein Ergebnis zurückzugeben (z.B. Einfügen oder Aktualisieren von Daten).

(Perform ist ähnlich wie execute)

DO <<ein\_anonymer\_PLpgSQL-Block>>

nützlich, um PL/pgSQL-Codefragmente direkt zu testen, ohne eine Prozedur oder Funktion erstellen zu müssen.

DO $$ DECLARE

counter integer := 0;

BEGIN

counter := counter + 1;

RAISE NOTICE ''The value of counter is %'', counter;

END $$;

|  |  |  |  |
| --- | --- | --- | --- |
| **Befehl** | **Zweck** | **Rückgabewert** | **Transaktionen?** |
| EXECUTE | Dynamische SQL- Befehle ausführen (SQL im plpgsql) + TRIGGER function setzen | Optional (z.B. SELECT-Ergebnisse) | Nein (ausser innerhalb Prozedur) |
| PERFORM | Funktionen aufrufen, Ergebnis ignorieren  In PL/PGSQL Code | Kein Wert (wird ignoriert) | Nein |
| CALL | Stored Procedures ausführen (explizit) | Kein direkter Wert (ausser OUT- Parameter) | Ja (explizit) |
| TRIGGER | Automatisch bei Datenänderung ausgelöste Funktionen | Kein direkter Wert (aber Änderung NEW/OLD möglich) | In Transaktion |

## Set Returning Function (SRF)

Datentypen

– RECORD, bzw. %rowtype

– SETOF <type> sowie TABLE (…)

CREATE OR REPLACE FUNCTION getAllFoo()

RETURNS SET OF foo AS $$

DECLARE r foo%rowtype;

BEGIN

FOR r IN SELECT \* FROM foo WHERE fooid > 0

LOOP

*--do something…*

**RETURN NEXT r;**

END LOOP;

RETURN;

END

$$ LANGUAGE ‘plpgsql’;

## Return Table

CREATE OR REPLACE FUNCTION get\_abtma (nr integer)

**RETURNS TABLE (**

**abtname VARCHAR, abtma VARCHAR**

**)**

AS $$

BEGIN

**RETURN QUERY**

SELECT abt.name, ang.name FROM abteilung abt

JOIN angestellter ang ON ang.abtnr=abt.abtnr

WHERE abt.abtnr=nr

ORDER BY abt.name, ang.name;

END; $$

LANGUAGE plpgsql;

select \* from get\_abtma(2);

# Cursor

- mehrere Cursors starten ist möglich

### 3 mögliche Cursor Deklarationen

Unbound Cursor

curs1 refcursor; (Query in Begin Section definieren und erst später verwenden)

Bound cursor

curs2 cursor for SELECT \* FROM abteilung;

Parametrisierter Cursor

curs3 cursor (arg1 type) for <query>:

curs3 cursor (id integer) FOR SELECT \* FROM abteilung;

angcursor **CURSOR** **(abtid IN abteilung.abtnr%TYPE) FOR**

SELECT salaer, persnr FROM angestellter

WHERE angestellter.abtnr = abtid;

Parameter müssen beim Öffnen des Cursors mit aktuellen Werten versehen werden

1. Deklaration des Cursors

DO $$ *-- anonymous block*

DECLARE

currabtnr integer := 1;

**angcursor CURSOR FOR SELECT salaer, persnr FROM  
 angestellter WHERE angestellter.abtnr = currabtnr**

[FOR UPDATE]**;**

salsumme DECIMAL (8, 2) := 0;

angsalaer angestellter.salaer%TYPE;

angpersnr angestellter.persnr%TYPE;

2. Öffnen des Cursors, Verarbeiten der Tuples

BEGIN

**OPEN angcursor ;** *-- SQL-Abfrage starten Resultat→Puffer*

**LOOP** *-- Iteration ueber Resulatmenge*

**FETCH angcursor INTO angsalaer, angpersnr;**

**EXIT WHEN NOT FOUND** *-- exit when no more row to fetch*

salsumme := salsumme + angsalaer;

RAISE notice 'Angstellter persnr: % salaer %',

angpersnr, angsalaer;

*Wenn FOR UPDATE gesetzt:*UPDATE angestellter   
 SET salaer = minsalaer WHERE CURRENT OF angcursor;

**END LOOP;**

3. Cursor schliessen

**CLOSE angcursor;**

RAISE notice 'Salärsumme: % ', salsumme;

END;

RETURN NEXTgibt aktuelle ROW von SELECT zurück

# Triggers

Implementation von komplexen Konsistenzbedingungen, Sicherheit, Sammeln von Statistikdaten. Event ist Tabelle zugeordnet. Trigger ist mit Funktion verbunden.

Events: INSERT, UPDATE, DELETE, TRUNCATE (Del ganze Tab)

Before Triggers (können Inhalt neuer Zeile ändern, wenn Return null dann abgebrochen), After Triggers (können nichts ändern, Return value ignoriert). Ausführung alphabetisch.

Ausführung: FOR EACH statement (einmal bei Eintritt Event für alle aufgerufen) | row (Für jede Datenzeile die betroffen ist ausgeführt. Mit Parametern OLD und NEW)

Ablauf Triggers bei Event: BEFORE FOR EACH Statement, Pro Tupel: BEFORE FOR EACH row (abbruch null), Bearbeiten, AFTER FOR EACH row, AFTER FOR EACH Statement | Diskussion: Triggers machen DB langsam, schwerer wartbar. Trigger kann man ausschalten.

Cascading Trigger Effekt: T a defniert Tga der Funkt. A aufruft -> verändert T b. T b definiert TG b der funk. B aufruft -> ändert T c. T c hat TG c -> Cascading. Problem wenn TG b a aufruft.

## Trigger Variablen

|  |  |
| --- | --- |
| TG\_NAME | Trigger-Name, z.B. Trigger1 oder Trigger 3 |
| TG\_WHEN | BEFORE oder AFTER |
| TG\_LEVEL | ROW oder STATEMENT |
| TG\_OP | INSERT,UPDATE, DELETE,TRUNCATE |
| TG\_RELID | "Object Id" der Tabelle |
| TG\_RELNAME | Name der Tabelle |
| TG\_TABLE\_SCHEMA | Schema der Tabelle |
| NEW, OLD | Attributwerte vom Typ Record |
| TG\_NARGS | Anzahl Parameter |
| TG\_ARGV[] | Array von Parametern als Text |

## Trigger Code

### Trigger Function

CREATE OR REPLACE FUNCTION dt\_trigger\_func()

RETURNS TRIGGER AS $$

DECLARE

arg0 int := tg\_argv[0]::int;

BEGIN

IF (*tg\_op*='INSERT') THEN  
NEW.creation\_date *:= now();*

ELSIF *(tg\_op*='UPDATE'*)* THEN

NEW.modification\_date *:= now();*

END IF;

RETURN NEW; -- passender Record (z.B. NEW/OLD/NULL)

END

$$ LANGUAGE plpgsql;

### Trigger Aufbau

(CREATE|DROP|ALTER) TRIGGER ang\_audit

(BEFORE|AFTER|INSTEAD OF)

(INSERT|UPDATE|DELETE|TRUNCATE) {OR ...}  
ON Angestellter   
[ FOR EACH ROW | FOR EACH STATEMENT (default) ]   
EXECUTE PROCEDURE process\_ang\_audit();

Bei Function: EXECUTE FUNCTION trigger\_fn\_name

Bei Procedure: CALL trigger\_procedure\_name

- NEW ist NULL bei DELETE und Truncate und OLD ist NULL bei INSERT.

- Mit RETURN NULL wird restliche Operation an Row übersprungen (auch nachfolgende Triggers werden nicht aufgerufen für Row).

## Instead of Triggers

Werden zur Realisierung von Updatable Views eingesetzt. Diese werden anstelle der ursprünglichen SQL-Operation ausgeführt. Können für Modifikation auf Tables und Views definiert werden. Leiten insert, update, delete auf Views zur darunterliegenden Tabelle weiter.

Spezieller Row-Trigger, wird anstelle des Inserts/Update/Delete-Operation aufgerufen mit den Trigger-Fn-Variablen old, new

DROP TRIGGER IF EXISTS trigger ON tablename;

CREATE TRIGGER ourview2trigger

**INSTEAD OF** UPDATE ON ourview2

FOR EACH ROW EXECUTE PROCEDURE ourview2triggerfn();

## Views

Für Sicherheit (Rechte), schnellere Abfrage, einfachere Queries

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

SELECT persnr, name, tel, wohnort FROM angestellter;

### UPDATE möglich, wenn

* Keine Join und Set-Returning-Operationen
* Keine Gruppen-Funktionen (min, max)
* Keine WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET, UNION, INTERSECT, EXCEPT
* Keine Aggregation (SUM), Window Function (OVER)

### Weitere Views

CREATE MATERIALIZED VIEW view2; // speichert Kopie der Query

REFRESH MATERIALIZED VIEW view2;

Temporäre Tabellen (CREATE TEMPORARY TABLE):

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

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

### Beispiel Updatable View

create view ourview1 as select name, abtnr, salaer

from angestellter order by abtnr, salaer desc;

update ourview1 set salaer = salaer + 10;

# Datentypen & Datenstrukturen

Datentypen Datensatz (Tupel, Record): einfachste Datenstruktur, Gruppe von inhaltlich zusammengehörigen Elementen bzw. Basis Datentypen. Definitionen eines Datentyps 1. Als Datenstruktur, 2. Als Operationen auf diesen Daten

## Datentypen in PostgreSQL (nicht alle)

Numerische Datentypen Ganzzahl INTEGER=INT, BIGINT Fliesskommazahl NUMERIC=DECIMAL → B-Tree, Hash, BRIN

Zeichenketten TEXT beliebige Länge, VARCHAR(80) maximale Länge, CHAR(3) feste Länge → B-Tree,GIN TSVECTOR/TSQUERY→ zusätzlich RUM

BinärBYTEA (postgres) binäre Datentypen für grosse Daten → kein Index

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 → Date/Time Indexe sind B-Tree, Hash, Brin

BOOLEAN (TRUE oder FALSE) → kein Index sinnvoll

Aufzähl-Datentyp ENUM → Partial Index | Array array[] → GIN, B-Tree

Währung MONEY oder DECIMAL(7,2) → B-Tree | Key-Value hstore → GIN

Dokument JSON, JSONB, JSONPATH → GIN, GIST, Bloom, BRIN,Hash,B-Tree

Raumbezogen/Geometrie PostGIS geometry/geography→ GIST, SP-GIST

## Array

In PostgreSQL Kein eigenständiger Datentyp, sondern Erweiterung jedes Basisdatentyps int, boolean etc.; gibt es mit fixer oder variabler Länge. Mehrdimensional ist standardisiert, sind keine Mengen/Sets

Arrays in Postgres beginnen standardmässig bei 1 aber kann auch im Minus Bereich spezifiert werden!

### Helper Funktionen

array\_dims(arr) printed array dimensionen: {{5,7},{8,4}} 🡪 [1:2][1:2]

array\_length(colName, 1) length of specified array dimension 🡪 1 = subset index

array\_to\_string(colName, SEPARATOR)

string\_to\_array(text::text, SEPARATOR)

unnest(ARRAY[1,2]) 🡪 eine Row => Spalte 1 = 1, Spalte 2 = 2,

array\_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon'); **→ 2** (gibt Index zurück, performance schlecht?)

array\_prepend(1, ARRAY[2,3]); -- zuerst value dann array

array\_append(ARRAY[1,2], 3); --> {1,2,3}

array\_cat(ARRAY[1,2], ARRAY[3,4]); (concatenate)

### Array Operatoren

ARRAY[1,4,3] @> ARRAY[3,1,3] → t (contains)

ARRAY[1,4,3] && ARRAY[2,1] → t (overlap) *2 im* *Bereich 1-4*

ARRAY[1,2,3] || ARRAY[4,5,6,7] → {1,2,3,4,5,6,7} (concat)

3 || ARRAY[4,5,6] → {3,4,5,6} (empty or 1-dimensional Arr)

SELECT ARRAY[1.1,2.2]::int[] = ARRAY[1,2](is equal)*cast int*

SELECT array\_to\_string(geometry, ';') FROM geometries;

SELECT unnest(geometry) FROM geometries; Werte einzeln

### Definiere Spalte mit Array Typ

CREATE TABLE sal\_emp (

pay\_by\_quarter integer[4], *-- 1 dimensional,* "*fix*"

schedule text[][] *–- 2 dimensional variabel/*"*unbound*"

);

// 2. Unbound default, nicht definiert, beginnt bei 1

//**Einfügen** von Array Elementen (zwei Möglichkeiten)

INSERT INTO sal\_emp VALUES (

'Bill',

'{10000, 10000, 10000, 10000, 10000}',

ARRAY[ ['meeting', 'lunch'], ['training', 'präsi']]

);

### Array Inhalt auslesen

SELECT '{1,2,3}'::int[]; --{1,2,3} Array Syntax

SELECT ARRAY[1,2,3+4]; --{1,2,7}

SELECT ARRAY

SELECT 1 + (random()\*5)::int // Wert zwischen 0-5

FROM generate\_series (1,6) ORDER BY 1 //6x ausführen

); --gibt z.B. {1,3,5,5,6,6}

### Array Accessoren

**Index** **Query**: intuitiv wie Koordinate (Start mit 1)

select board[1][1] from tictactoe;

-- z1 k1

**Slice Query**: Untergrenze:Obergrenze für jede Dimension

select board [2:3][1:1] from tictactoe;

-- {{z2 k1},{z3 k1}}

**Max Bound** Abkürzung "[2]" vermeiden, besser [1:2]

select board[2:3][1:2] from tictactoe ; -- [2]⬄[1:2]

--{{z2 k1,z2 k2},{z3 k1,z3 k2}}

Suche mit ANY (any für jedes Array Element)

select \* from tictactoe where 'z2 k2' = any (board);

-- 1;{{z1 k1,z1 k2},{z2 k1,z2 k2},{z3 k1,z3 k2}} //ganze zeile ausgegeben

SELECT array[] FROM table; ==? SELECT array [:] FROM table;

SELECT array[1] FROM table; -- Element bei Index 1

### 3x2 Dimensionaler Array

create table tictactoe as

(select 1 as id, ARRAY[

['z1 k1', 'z1 k2'], ['z2 k1', 'z2 k2'],   
 ['z3 k1', 'z3 k2'] ] as board);

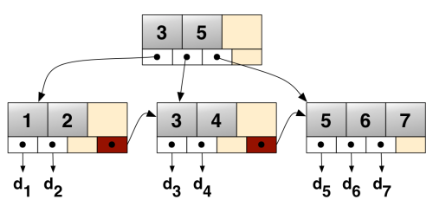
## Dictionaries (Key Value Stores)

Key Value Pairs, ADT && Datentyp, Bildet “Unique Key” auf Value ab. Für variable, und unvorhersehbare Werte.

No schema/Schema less, umstritten (consistency, optimization)

Geeignet für Einfache Datenspeicherung und Datenerfassung, Zeilen mit vielen Attributen, die selten untersucht werden, Semi strukturierte Daten.

hstore implementiert in PostgreSQL Dictionaries als Datentyp: Keys und Values sind vom Typ TEXT. Unterstützt vom GIST/GIN Index Keys müssen eindeutig sein, sonst werden sie ignoriert.



B+-Baum

CREATE EXTENSION hstore;

CREATE TABLE kvptable\_hstore (

id SERIAL PRIMARY KEY, name TEXT,

kvp **HSTORE**

);

INSERT INTO hstoretable (id, kvp) VALUES

(1, hstore('key1', 'value1') || hstore('key2','value2') )

Queries – (Wenn klar ist, dass der linke Operand hstore ist, dann wird der rechte implizit auf hstore gecastet)

### Operatoren -> und <@ und ?

SELECT 'key1=>v1, key2=>v2'::hstore -> 'key1'; -- v1

*-- Does hstore contain key?*

WHERE tags ? 'key' <==> ~~WHERE tags->'key' IS NOT NULL~~

-- Does hstore contain all the specified keys? (?| = or)

'a=>1,b=>2'::hstore ?& ARRAY['a','b'] -> t

// Get value from key (as text)

SELECT mykvpfield->'name' FROM

//Test if right hstore is contained in left hstore:

hstore('a=>b, b=>1, c=>NULL') @> 'b=>1' -> t

//convert hstore to array with %%, %# for 2dimensional

**%%** 'a=>foo, b=>bar'::hstore -> {a,foo,b,bar}

-- oder hstore\_to\_array() hstore\_to\_matrix()

### Helper Funktionen

SELECT akeys(kvp) FROM ...-- List all keys

SELECT each(kvp) FROM ...-- Get all key value pairs

# Indexe und Speicherstrukturen + Optimierung

### Datentypen und passender Index

1. Numerisch → B-Tree, Hash, BRIN
2. Zeichenkette → GIN, GIST, Bloom, BRIN, Hash, B-Tree, RUM
3. Char, Text, Varchar Data Type => B-Tree, GIN
4. Tsvector, Tsquery => B-Tree, GIN, RUM

### Indexe

B-tree balanced 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 Generalized Search Tree (Array, Volltextsuche, nearest-neighbor), << &< <<| &<| <@ ~= && SP-GiST space-partitioned GiST (knn (nearest-neighbor), geometrisch), << ~= <@ <<|

SELECT \* FROM products WHERE title LIKE '%JASMIN';

CREATE INDEX like\_search\_tree ON table   
 USING gist(title gist\_trgm\_ops);

GIN Generalized Inverted Index (schneller aber mehr Daten als GiST, wenig ändern!, Array, JSONB, Volltext), <@ = &&   
BRIN Block Range Indexes (Range Search) für viele schon sortierte Daten, ISAM Indexed Sequential Access Method (wie hash aber kann mehr)

DROP INDEX [IF EXISTS] indexname;

Zusammengesetzter Index, Include, funktionaler index

CREATE INDEX IF NOT EXISTS col12\_idx ON   
 mytable (col1, col2); **INCLUDE (txt**); ON mytable( UPPER(col) );

CREATE INDEX mytable\_col\_idx

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

CREATE EXTENSION btree\_gist;

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

WHERE archived IS NOT NULL; partieller Index

### Index für GiST und GIN für Hstore

CREATE INDEX kvps\_idx ON kvptable\_store USING **GIST** (col);

VACUUM (VERBOSE, ANALYZE) table;

Kein index erstellen für primary key, dort wird automatisch ein index erstellt

Bitmap Index ordnet Attributwerte als Bitmuster (binär), wenig diskrete Werte, geringe Selektivität, AND/OR. PostgreSQL verwendet aber oft intern einen Bitmap Index (bitmap index scan, bitmap heap scan).

## Index Speicherung

Primär-Index Index mit PK (automatisch), immer unique & not null

Cluster = physischer Speicher wie Index, nur 1 pro Index

Clustered Table in Postgres

CLUSTER table USING indexname; Daten werden physisch sequentiell angeordnet gemäss den Index Informationen, One-Time Operation, Changes danach werden nicht automatisch clustered.

Clustered, integrierter Index in MS SQL

Daten werden laufend automatisch physisch sequentiell angeordnet ge-

mäss den Index Informationen, die Blätter enthalten die Daten vom Rest der Columns, nicht nur Referenzen

Nicht-Integrierter Index Blätter haben Referenzen auf die Heap-Daten

Mehrdimensionaler Index z.B. R-Tree für 2D-Geom.

Mehrstufiger Index Erster Index mit grober Filterung, zweiter Index mit exakter Filterung

# Planer: Optimierer von PostgreSQL

Ziel Erzeugt einen optimalen Ausführungsplan (**Execution Plan**) für eine SQL Query (geparst als Baumstruktur).

EXPLAIN: Gibt geplanten Ausführungsplan aus.

EXPLAIN ANALYZE: Führt Query aus und gibt den Plan mit effektiven

Cost: Fiktive Zeiteinheit, beschreibt Aufwändigkeit der Operation

Ausführungskosten aus. table scan (seq scan) kann parallelisiert werden. Suchstring mit %-Zeichen kann nicht optimiert werden. cost 0.15..: Startup Kosten (Sortiervorgänge etc). cost=0.15..23.44: Gesamtkosten. rows =2: Geschätzte Datensatzanzahl. width =2: Geschätzte Datensatzbreite (in Bytes)

## Scans

Table Scan Full Table Scan oder Seq Scan: Z.B.: SELECT \* FROM angestellter

Performance: Langsam. Lohnt sich wenn mehr als 80% der Daten im Resultset sind.

Index Scan Falls in der WHERE Klausel ein Attribut ist, zu dem es einen Index gibt. Lädt einen Tupel Zeiger nach dem anderen aus dem Index und greift sofort auf das entsprechende Tupel in der Tabelle zu.

Bitmap Index Scan (Bitmap Heap Scan): Lädt alle Tupel Zeiger auf einmal aus dem Index, benutzt eine Bitmap Struktur, um sie im Hauptspeicher zu sortieren, und lädt die Tabellen Tupel entsprechend der physischen Speicherreihenfolge.

Index Only Scan Falls Attribut in der Projektion vorkommt. Wegen MVCC muss der Index alle Datenblocks konsultieren, ausser das visibility map Bit in der Indexstruktur ist gesetzt. Z.B.: SELECT abtnr FROM angestellter WHERE abtnr =2; (Covering Index ist eine Eigenschaft des Indexes, Index Only Scan ist eine Eigenschaft der Abfrageausführung.)

## QUERY-ARTEN UND DB-BENCHMARKING

U.a. zur Charakterisierung von Index-Typen, Performance-Benchmarks, etc.

Prädikat: Ausdruck (Expression), der eine TRUE-/FALSE-Bedingung auswertet – kann bei DB auch UNKNOWN sein.

Equality Query (bzw. Equality Prädikat): Q. mit Equal-Operator "=" kann mehrere Zeilen zurückgeben.

Range Query (Bereichsabfrage): Q. mit Operatoren >, >=, <, <=, != BETWEEN. Gibt typischerweise mehrere Zeilen zurück.

Point Query Query mit dem Equal-Operator "=" gibt immer ein einziges Tupel/Wert zurück.

Join Query Query mit JOINs.

Weitere Queries mit Präfix-/Suffix in Texten oder Extremwerte in Zahlen.

## Join Strategien

Nested loop join Die rechte Tabelle wird für jede Zeile in der linken T. gescannt. Ist einfach, kann aber zeitaufwendig sein.

Merge Join Jede T. wird vor dem Start des Joins nach den Join Attributen sortiert. Dann werden die beiden T. parallel gescannt und übereinstimmende Zeilen zusammengefasst. Erfordert explizites Sortieren oder Scannen der T. über Index. (z.B. bei deaktiviertem hash join).

Hash Join Zuerst wird die rechte Tabelle gehasht (ins memory) mit Join Attribut als Hash Key. Dann wird die linke T. gescannt und jede gefundene Zeile als Hash Key verwendet. (Bei Equality Join: ON .. = ..)

Wahl des Algorithmus Falls kein Index 🡪 Hashed Join bei Equi join und (sonst merge join) | Bei Index 🡪 Nested Loop mit covering index, ausser S ist grösser; Index auf Attribut beschleunigen die meisten Operationen, verhindern Full Table Scan; Non equi != <> 🡪 Nested Loop Join, keine Index Optimierung möglich.

Schnellster Exists Query

SELECT \* FROM orders o WHERE EXISTS (  
 SELECT \* FROM orderlines ol   
 INNER JOIN products p ON ol.prod\_id = p.prod\_id

WHERE ol.orderId = o.orderid AND P.category **=** 11 );

Sub-Select Query ist schneller als Distinct und HAVING Query und CTE ist auch schnell.

# Query Index Optimierung

## Interne Ebene (Indexe & Optimierung)

Index Datenstruktur in DBMS, welche Query beschleunigt. Ist im Primärspeicher (Cache)

Data Page Daten einer Tabelle physikalisch gespeichert (als Datei). Enthält Zeilen in unsortierter Reihenfolge. Page Überlauf durch Einfüge und Update Operation | Page Sets (Blocks) = Collection von Pages

Heap = Collection von Page Sets / Pages

Ziel vom Index Abbilden von Werten mit Referenzen (in Memory) auf Daten (Zeilen) und ggf. auf Pages im Sekundärspeicher

Aufbau Page mit Indexen auf eine Tabelle. Diese Tabelle ist in einer Page gespeichert. Die Indizes haben jeweils eine RowId auf einer Zeile in der Page der Tabelle.

# Statistik

## Analyse und Statistiken …

DB Statistiken

Die DBMS führen Statistiken über Anzahl und Verteilung der Daten in den einzelnen Tabellen. In PostgreSQL unter «pg\_stats» gespeichert.

Abschätzung Kosten: Kommunikationskosten (#Nachrichten, Menge Daten), Berechnungskosten (CPU, Pfadlänge), I/O Kosten (Seitenzugriffe), Speicherungskosten (Temp. Speicherbelegung).

Ziele Datenmodellierung: Einfache und klare Semantik, Redundanzfreiheit

Ziele Datenbank Tuning: Beschleunigung von Abfragen, hoher Durchsatz

## Anfrageoptimierung / Query Optimierung

Phasen: 1. Übersetzung: finde geeignete interne Darstellung für Query (Syntax Baum). 2. Logische Optimierung: Umformung der Query aufgrund Heuristiken. Ohne Zugriff internes Schema und statistische Daten. Annahme Elemente und Attributwerte gleichverteilt. Suchprädikat unabhängig. 3. Physische Optimierung**:** Erzeugung von einem/mehreren Ausführungslplänen. Einbezug Indexe. Verbesserung der Analyse mit Statistiken und Heuristiken + Kosten. Spezialfall: Kostenbasierte Optimierung, alle Ausführungspläne generieren, Kosten bewerten. 4. Auswahl des günstigsten Plans Basierend auf statistischen Informationen aus dem Katalog Berechnung von Kostenvoranschlägen für jeden möglichen Ausführungsplan und Auswahl des billigsten/günstigsten Plans

Optimizer kann mit Hilfe von Histogramm die Selektivität bestimmen. Für Histogram werden nur Stichproben verwendet.

Anfrageoptimierung Tipps

JOIN: Klausel Syntax, Ende Anfrage. Attributwertebereiche beachten: Numerisch schneller als Text vergleich, Schnelle Vergleiche an den Anfang, Index erstellen. Kein Selekt \*, Unterabfragen durch JOIN ersetzten. UNION durch Where ersetzen.

Erstelle Indexes

auf Kolonnen die häufig in where oder join Bedingungen vorkommen und deren Daten eine hohe Selektivität haben.

## Selektivität und Dichte

Dichte

Dichte ist pro Attribut der durchschn. prozentuelle Anteil von Duplikaten. Tiefe Selektivität = hohe Dichte. Unique Index hat tiefe Dichte.

Selektivität

Prozentueller Anteil der Tupels in einer Tabelle, die von einer Query geliefert werden.

Hohe Selektivität bei <= 0.1 🡪 DBMS/Planer verwendet Index

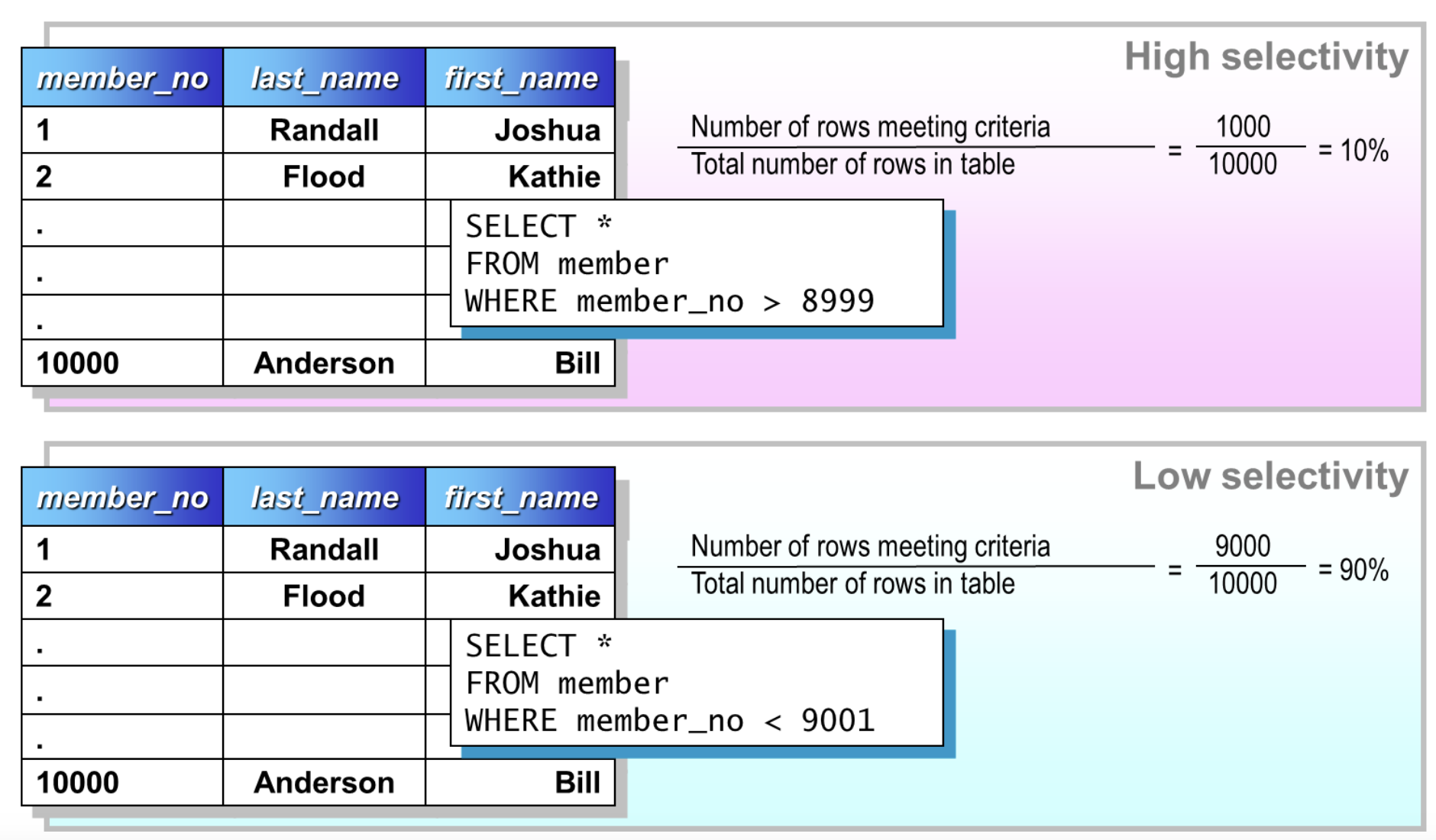
Tiefe Selektivität bei > 0.1 🡪 DBMS/Planer macht Table Scan

Even distribution Histogramm / height balanced Histogramm

Zeigt Verteilung (bei welchen Werten, welche Selektivität. Number = Bucket Nummer (im Beispiel pro Bucket 0.1 Selektivität). Value = Werte von Query.

Number/Bucket 10 = 1 \* 0.1 = 0.1 und Buckets 1-4 = 4\*0.1 = 0.4.

ohne Histogramm

****

# Baumstrukturen (JSON)

### Normalisierte Hierarchie-Tabelle als Baumstruktur

Normalisiert bedeutet: Die Tabelle ist möglichst frei von Redundanzen (Datenwiederholungen) und erfüllt mindestens die 3. Normalform – jedes Attribut hängt nur vom Primärschlüssel ab.

Hierarchie bedeutet: Die Tabelle bildet eine über-/untergeordnete Struktur ab (z.B. Chef und Mitarbeiter, Oberkategorie und Unterkategorie).

Tree: azyklisch verbundener Graph

Rooted Tree: hierarchical, connected, acyclic Ordered Tree: Ordnung der Kinder definiert Binary Tree: jeder Knoten hat 0-2 Kinder,

Anwendungen Führungshierarchie, Stücklisten, Kataloge

Eigenschaften: Parents/Childs/Ancestors, Siblings, Root (Node ohne Parent), Leaf (Node ohne Kinder), Internes Node (nicht Root/Leaf), Tiefe eines Nodes, Grad eines Knotens (Anz. Childs), Höhe des Baums, Teilbaum

## JSON als Baumstruktur

Semistrukturierte Daten: SQL/JSON, SQL/XML

Daten bzw. Dokumente als Baumstrukturen

JSON Postgresql: json: speichert exakte Kopie des Textes. Jsonb (ab 9.4): speichert geparstes Binarärformat, schneller index. Cast: ::json

Select '{"a":1,"b":2}'::json ->> 'b'->2, get obj. as text

'{"a": {"b":"foo"}}'::json -> 'a' Get JSON array/object

SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb --t Contains

SELECT '{"foo": "bar"}'::jsonb ? 'foo'; --t (key exists)

select a || b, select a – b Concatenate und Delete

SELECT to\_json('2021-03-09'::timestamp(0)) #>> '{}'; text

SELECT jsonb\_pretty(airport) FROM airports   
 WHERE airport **->>**'name' LIKE 'Zürich%';

jsonb\_pretty **{** "name": "Zürich Airport", "region": "Zurich", "navaids": [{ "name": "Kloten", "ident": "KLO"}]**}**

### Build JSON in Query

SELECT jsonb\_build\_object(

'persnr', ang.persnr,

'name', ang.name,

'projnr', jsonb\_agg(pz.projnr)

) AS json\_output

FROM angestellter ang

LEFT JOIN projektzuteilung pz ON ang.persnr = pz.persnr

GROUP BY ang.persnr, ang.name;

### Convert Select Result to JSON row\_to\_json() = to\_json()

SELECT row\_to\_json(row(row1, row2, row3)) FROM table;

SELECT array\_to\_json(fields) FROM table;

### Jsonpath @? und @@ supported by GIN Index

select '{"a":[1,2,3,4,5]}' @? '$.a[\*] **?** (@ > 2)'; -- true

select jsonb\_path\_exists('{"a":[1,2,3,4,5]}', '$.a[\*] ? (@ > 2)', silent=>true); --> true (ist gleich wie @?)

select '{"a":[1,2,3,4,5]}' @@ '$.a[\*] > 2'; -- true

select jsonb\_path\_match('{"a":[1,2,3,4,5]}', '$.a[\*] > 2', silent=>true); -- true (ist gleich wie @@)

SELECT airport->>'name' FROM airports WHERE

airport->>'ident' IN ('LSZH', 'KBWI');

## Json Path | Jsonb\_path\_query aus Übung

SELECT jsonbpq(data, '$ ? (@.customer.name == "Peter Staub")') FROM orders\_json;

Gleiches Mit wildcard

SELECT jsonbpq(data, 'strict $.articles[\*].article ? ($.articles[\*].price > 100)') FROM orders\_json;

# NoSQL

RDBMS Relationales Datenbanksystem: Single Point of Failure durch Shared Everything. Scaling out ist aufwändig. keine Objekte: aufwändiges OR Mapping.

NoSQL Einfaches API (http), grosse Datenvolumen: Skalierbar. Oft nicht relational, schema frei. Oft Architektur basierend auf BASE (), Verzicht auf ACID.

Datenmodelle

Aggregation Stores : Key/Value und Document stores.

Graph Stores

Column Family (Wide Column Stores)

## CAP, BASE & ACID

### CAP wähle 2 aus

C - Consistency Daten sind die gleichen auf jeder Replikation und jedem

Node; verlangt Atomicity, Transaction Isolation. Was zuerst abgeschickt wurde, kommt zuerst an. | Nicht wie Consistency in ACID.

A - Availability Daten müssen immer verfügbar/zugreifbar sein

P - Partition Tolerance DBMS funktioniert auch bei teilweisem Ausfall von Netzwerk und Nodes. | RDBMS: C-P, NoSql: P-A, Small Data Sets: C-A

### BASE

Basically Available (Grundsätzlich verfügbar): besagt, dass System die Verfügbarkeit der Daten im Sinne des CAP-Theorems garantiert; auf jede Anfrage eine Antwort. Aber Antwort kann sein, dass Daten nicht verfügbar sind oder in inkonsistenten oder ändernden Zustand befinden.

Soft state (Weicher Zustand): Der Zustand des Systems kann sich im

Laufe der Zeit ändern, d. h. auch in Zeiten ohne Eingaben kann es

aufgrund der "eventuellen Konsistenz" zu Änderungen kommen.

Eventuelle Konsistenz: Das System wird konsistent werden, sobald es keine Eingaben mehr erhält

### Unterschied zu ACID

ACID = Atomarität, Konsistenz, Isoliertheit und Dauerhaftigkeit

Unterschied zwischen dem CAP-Theorem und ACID besteht darin, dass das CAP-Theorem die Herausforderungen in verteilten Systemen betrifft, während ACID die Eigenschaften von Transaktionen in relationalen Datenbanken beschreibt. In ACID müssen alle Eigenschaften erfüllt sein.

# MongoDB Document Store

Database ≈ DB/Schema | Collection ≈ Tabelle | Document ≈ Zeile/Row

Fields ≈ Spalte/Column | \_id (objectId) ≈ row id | DBRef ≈ Join

* Data Partitioning mit Sharding
* Availability mittels Replikation und "Raft"-ähnlich Konsens-Algo. Quorum
* JSON als grundleges Format (JSON/BSON binary serialized JSON format)
* Document Data Model (Schema-less/flexibles Schema)
* MQL = MongoDB Query Language
* Mapped einen Key zu einem strukturierten Dokument
* Keine echten JOINs
* Transaktionen jeweils pro Dokument, atomar, multi-document transactions (MDT) unterstützt

Modellierungsentscheide Ausgewogenheit des Datenmodells.

Stärken: Prototyping (JSON over http), Webfreundlich, DbaaS/Cloud

Schwächen: Keine Constraints, keine echten Joins langsam. langsame mangelhafte Transaktionen. Security (default offen im Web).

+/- einfache Query Sprache, No Schema

Geeignet: analog Column Stores. Schlecht: atomare cross documen q.

## Embedding vs Referencing - MongoDB Beziehungen

Denormalisiert (embedding) bei mongoDB bei 1-to-1 oder 1-to-many ({..} als Feld) | normalisiert «manual references» Referenzen mit \_id als Wert. empfohlen für normalisierte Daten. Many-to-many. Grosse Hierarchien.

## Sharding (für DB Data Partitioning von MongoDB)

Spezialfall von Horizontaler Partitionierung wobei Partitionen auf verschiedene Nodes verteilt werden.

Ziel gute Verteilung der Dok. = gute Auslastung der Nodes.

Der Shardkey sollte ein Schlüssel sein, der die Daten gleichmässig verteilt.

Bei MongoDB Hash auf \_id "db.events.createIndex({\_id: 'hashed'})"

- Daten gleichmässig auf Nodes verteilt

- Feld oder Kombinationen davon, bzw. eine Funktion auf Feld oder Kombinationen - Sollte nicht unique sein (nicht Identifikator/PK)  
- Ein oder mehrere Felder - Numerische Werte

## Concistency & Replicas

MongoDB Repliziert: Ausfallsicherheit => Replikation mit mind. 3 Knoten. MongoDb Cluster mit dem Primary A und den Secondaries B, C, D und E: A read & write. Secondaries keine, ausser Read Preferences ist entsprechend gesetzt. Ausfall: detektieren mit Heartbeat. Wenn ein Replica Set ausfällt, wird ein neuer Primary gewählt. 1 Primary kann lesen und schreiben, Secondary nur lesen.

## MongoDB Syntax | DB CRUD

**CREATE**

use mydb -- set current database

db.collection.**insert**({name: 'Aurora', gender: 'f'});

*WriteResult({ "nInserted" : 1 });*

db.employees.**insertMany([** Hash/ObjectId hier zu kurz/nicht echt!

{"\_id": ObjectId("123"), "name": "Leto"}, {"\_id": ObjectId("456"), "name": "Moneo", "manager": [ObjectId("123"), ObjectId("254")] } **])**

**READ**

$lte = Less than or equal, $ne = not equal to, $not, $all:[], $in:[]

db.collection.**find**({"\_id" : ObjectId("58ef0f135")});

db.unicorns.find**(** { gender: 'f',  
 **$or**:[{vampires: {**$exists**:false}}, {vampires: {**$lte**:0}}],

**$and**: [ {weight:{**$gte**:600}},{weight:{**$lte**:900}} ] },

{**\_id**:0, "name":1} **)** *-- “name”:1 gibt nur den Namen aus*

-- Manager von Moneo mit manuellem Join:

db.collection.findOne(query, projection/select, options)

db.employees.findOne({ "\_id":   
 db.employees.findOne( {"name":"Moneo"} ).manager })

== SELECT \* FROM employees as manager INNER JOIN employees as moneo ON moneo.chef = manager.persnr WHERE moneo.persnr = 1100;

-- Manager von Moneo mit Lookup mit Aggregation:

db.employees.aggregate(**{** "**$lookup**": { "from": "employees", "localField": "manager", "foreignField": "\_id", "as": "managers"

}**}**, **{** "$match": { "name": "Moneo" } **}**, **{** "$project": { "\_id": 0, "manager": 0, "managers": { "\_id": 0, "manager": 0} } **}**)

-- Manager von Moneo mit $graphLookup:

db.employees.**aggregate([ {** "$graphLookup": { "from": "employees", "startWith":"$manager", "connectFromField" : "manager", "connectToField": "\_id","as": "managerDocs", "**maxDepth**": 0 } **},** **{**"$match": {"name":"Moneo"}**},   
{**"$project":{"\_id":0,"name":1,"manager":1,"managerDocs":1}**}**  **])**

*-- Output: { name: 'Moneo', manager: ObjectId('123'),   
 managerDocs: [ { \_id: ObjectId('123'), name: 'Leto' }]}*

UPDATE (upsert=update or insert)

db.coll.update({name: "Mia"},{$**set**: {weight: 590}})

db.update({name: "Pilot"}, {$**inc**: {vampires:-2}},{upsert:false})

db.update({name: "Aurora"}, {$**push**: {loves: "sugar"}})

DELETE

db.collection.**remove**({}) oder drop() --{} leer= alle

# DuckDB (Column-oriented Store)

In-Process Analytical Column Store.

DuckDB: Optimiert für Analyse, Oft als "SQLite for analytics" bezeichnet, Einzelbenutzer OLAP. DuckDB Ist embedded / in-process – also Single User. "Query Pushdown" (filter/projection pushdown) und parallelisiertes Lesen

SQLite : Einzelbenutzer OLTP und mobile Apps.

Haupteigenschaften: Column Store, vektorisierte Ausführung in CPU, keine externen Konfigurationsdateien oder Einstellungen, In-Memory-optimierte Verarbeitung, Abfrageoptimierung, Erweiterbarkeit, SQL-Kompatibilität, ACID, Open-Source (MIT Lizenz), Indexe: Min-Max Index, Adaptive Radix Tree (ART) muss in-memory Platz haben.

* In-Process = läuft innerhalb einer Anwendung, nicht separat.
* In process ist nicht unbedingt in memory, meistens embedded
* serverless = ohne separaten DB Server
* DuckDB Operationen sind atomic, aber bei multi-document transactions macht es eine 2PC Emulation/Guideline
* DuckDB & SQLite sind in process und viel embedded in memory möglich

DuckDB vs. PostgreSQL:

–DuckDB ist ca. 50x schneller als PostgreSQL bei Queries mit Aggregationsfn. und bei wenigen Kolonnen in der Projektion (wohl wegen column-store/vectorized)

– DuckDB ist bei Updates/Deletes langsamer als PostgreSQL

– Bei DuckDB sind die Unterschiede von Kalt- zu Warmstart geringer als bei

PostgreSQL (wohl wegen kleinerem Overhead von DuckDB)

## In-Memory Stores

IMDB: Datenbankmanagementsystem, das Daten vollständig im Hauptspeicher speichert. Dies steht im Gegensatz zu herkömmlichen «On Disk» Datenbanksystemen, die für die Datenspeicherung auf persistenten Medien konzipiert sind. IMDB können eine Grössenordnung (10 1000x) schneller sein, v.a. wenn sie In Memory optimiert, mit Column Architekturen kombiniert sind oder aber spezialisiert sind. In Memory Datenbanksystem steigert die Leistung, indem es alle Datensätze im Speicher hält.

Persistenz, Datenhaltung: Beim Starten der Datenbank lädt das System den gesamten Datenbestand von der Festplatte in den Speicher, damit bei laufendem Betrieb keine Daten nachgeladen werden müssen. Checkpoint Files/Snapshot Images: Geänderte Daten werden in regelmässigen Abständen mit dem persistenten Speicher (Festplatte) abgeglichen. Transaction Logs: Zwischen den einzelnen Checkpoint Vorgängen werden laufende Änderungen in Transaction Logs geschrieben, um nach einem Crash ein Rollforward machen zu können. ACID Prinzip eingehalten.

Use Cases: OLAP, oft mit Column Stores kombiniert. Not Use: viel Einfügen.

## Column-oriented Store

OLTP: Online Transaction Processing: Low data volume, high transaction volume, normalized data, ACID, Require high availability.

OLAP: Online Analytical Processing: High data vol, low transaction val. denormalized data, not necessarily acid, don't usually require high availability.

Row Store Daten: Tuple by tuple auf Disk, Zeilen werden nacheinander gespeichert Optimal für den zeilenweisen Zugriff (z.B. SELECT \*). Vorteile: Seq. Row Scans gute Bandbreiten Ausnützung, ermöglicht einfache horizontale Partitionierung (Parallelisierung), Einfügen. Probleme: Hauptspeicherzugriff (I/O) ist Engpass: Tupel Konstruktion ist aufwändig.

Column Store: enthalten spaltenweise physische Datenstrukturen, Column by column auf Disk, Die Spalten werden nacheinander (komprimiert) gespeichert, Optimal für attributierten Zugriff (z.B. SUM, GROUP BY). Vorteile: Bessere Kompression und Caching. Kleinerer Speicherplatz, automatisch Index auf jede Spalte Probleme: Schreiben und select \*.

Use: Massive Db, read mostly, read intensive (OLAP). kleine DB Grösse auf Disk. Not Use: OLTP, Data Analytics, write intensive dataset

# Graph

### Graph Types in post-relationalen DBMS

Rekursive Abfragen mittels CTE sind eine Möglichkeit mit Graphen in SQL

Umzugehen Anwendungen Transport, Energie, Navigationssysteme (kürzester Weg) / Geoinformationssysteme GIS (Wasserabfluss)

## NoSQL Graph Stores / Database

Graph Graphstrukturen bestehen aus Knoten (oder Vertices) und Kanten

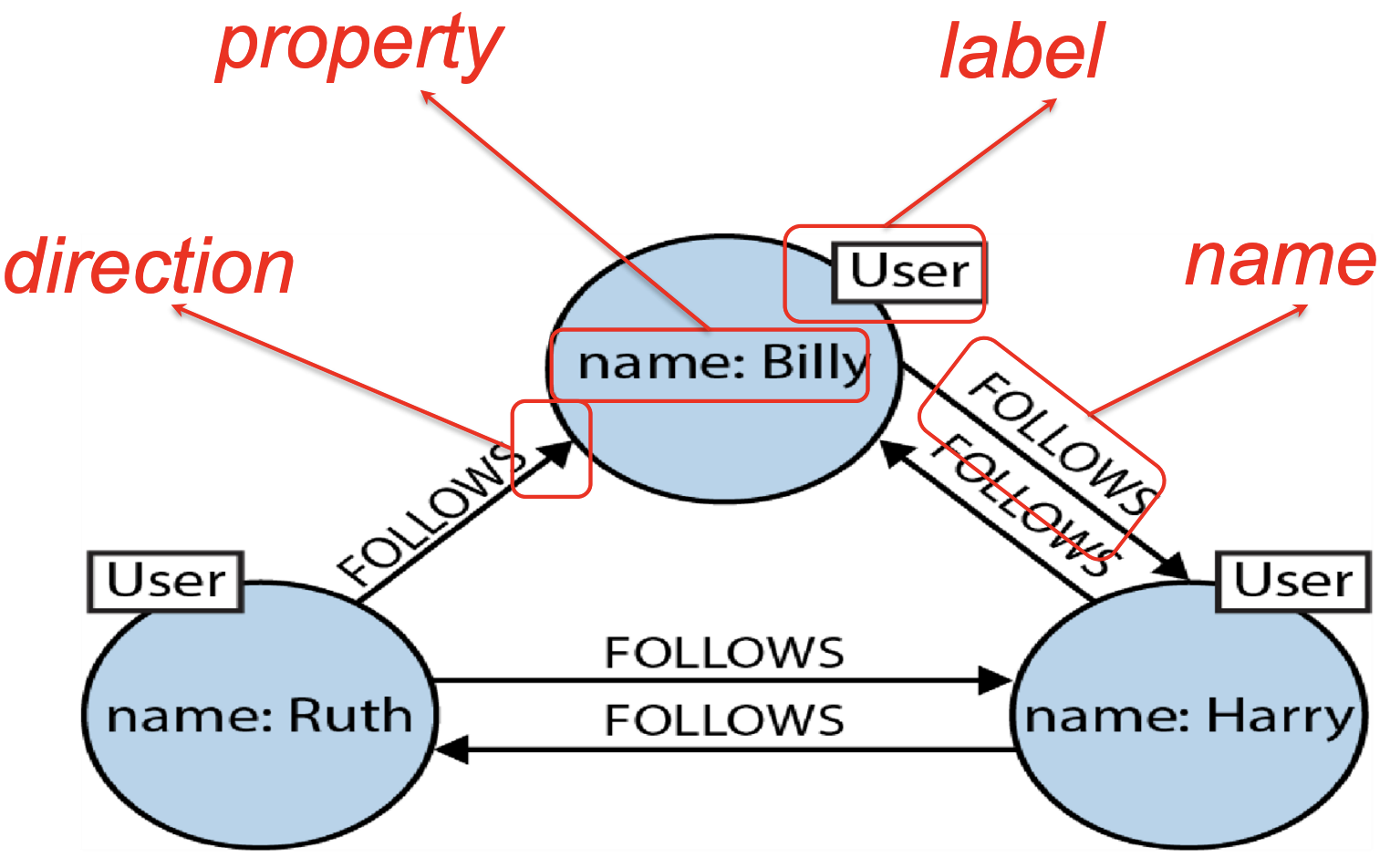
(Verbindungen zwischen den Knoten), die für die Darstellung von

Netzwerkbeziehungen optimiert sind.

Labeled Property Graph Eine Art von Graph, bei dem die Knoten und die Kanten mit Bezeichnungen (Labels) und Eigenschaften (Properties) versehen werden können. Mächtiger als RDF wegen "Properties"!

RDF Triple Das "Resource Description Framework" (RDF) mit "Tripeln" der Struktur Subjekt-Prädikat-Objekt. Standardmodell: Datenaustausch im Web.

Vorteil zu RDBMS: Bei Graphen Datenbanken sind Beziehungen sehr wichtig, diese können durch Labeled Property Graphs sehr gut dargestellt / modelliert werden. Alle Beziehungen sind durch gerichtete Kanten modelliert.

Use Cases Connected Data, Routing, Dispatch, and Location Based Services, Recommendation Engines When Not to Use: update all or a subset of entities, Big Data

### Cypher Query Language

MATCH pattern → FROM

WHERE condition/pattern → WHERE

SELECT → RETURN it

Knoten (), labeled Beziehungen [:.], Verbindungen --,<--,-->,<-->

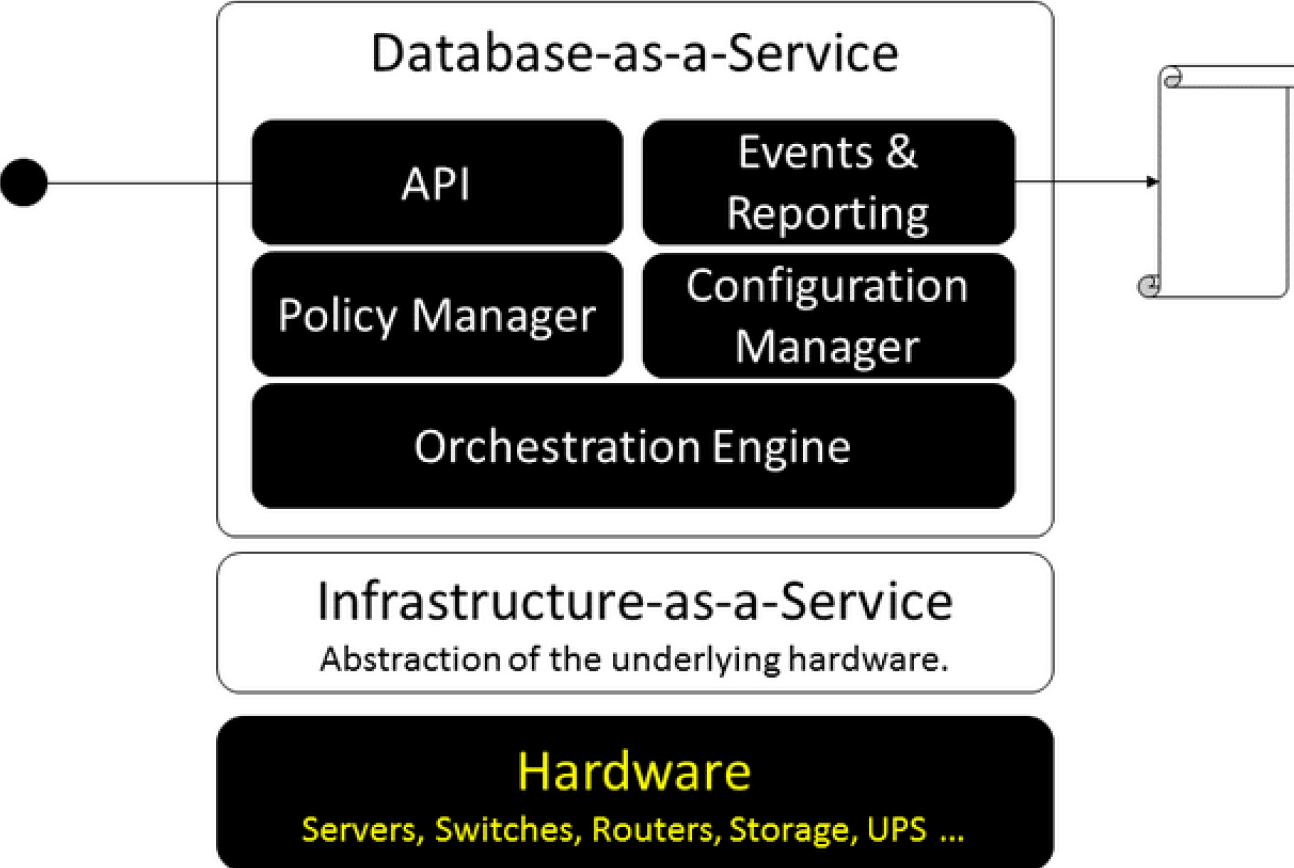
Movie: Label/Klasse | title und released = Attribute/properties

CREATE (alias:**Movie** {title:'Matrix', released:1999})

MATCH (:Movie {title: 'Wall'})<-[:ACTED\_IN]-(actor:Person)   
 RETURN actor.name AS actor

## DBaaS = Cloud-DB-Dienste

Cloud-basierter Ansatz zur Speicherung und Verwaltung von strukturierten Daten. Stellt flexiblere, skalierbare On-Demand-Plattform zur Verfügung, die sich per Self Service und einem einfachen Management organisieren lässt. Abstrahiert für Benutzer. Weder rein Infrastruktur noch Plattform.

Aufbau DBaaS

(Web-)Client-Anwendungen können Daten aus der DB abfragen und verändern (CRUD). Zudem können Clients Views, User Defined Functions und Stored Procedures aufrufen. Tiefe "Total Cost of Ownership", einfach erweiterbar.

Pagination Durch Ergebnisse einer (grossen) Datenabfrage blättern  
Umsetzung mit OFFSET skaliert nicht, besser ist SEEK (FETCH NEXT 25 ROWS ONLY). Best Practice ist Cursor, robust bei Datenänderungen während blättern. | Pagination erfordert eindeutige Sortierreihenfolge.

## GraphQL (nicht Graph store)

Ist eine deklarative Datenabfrage- und Manipulations-Sprache für API (über’s Web, http) und Laufzeitumgebung. Ist Schema-basiert mit Typen; nicht «No-Schema». | KEINE Abfragesprache für Graph Stores, sondern DB-unabhängige DDL/DML | Ist eine REST/OpenAPI-Alternative, die vom Bedürfnis von Facebook entstanden ist, dessen «SocialGraph» als «Timeline/Feed» darzustellen. | Bei REST gibt es verschiedene Endpoints, bei GraphQL einen einzigen Endpoint. | SQL ist mächtiger aber GraphQL ist kontrollierbar (garantierte Performance und Availability). | «Resolvers» sind der Code, der nötig ist, um auf die konkreten Daten zuzugreifen

Schema first vs. code first: Ein Schema (DB) erhöht Lesbarkeit, verbessert Kommunikation (Front/Backend Teams). Ist Dokumentation, ermöglicht rasche SW-Entwicklung. PostGraphile = Schema first, GraphQL kann beides.

### PostGraphile

PostGraphile = [www.graphile.org](http://www.graphile.org) und nicht GraphiQL (UI für GraphQL).

Ist eine Bibliothek für "pluggable” GraphQL-APIs | Implementiert GraphQL als Webservice/API mit PostgreSQL als Datenspeicher und NodeJS als http-Server. Resolvers sind hauptsächlich SQL. Aus gegebenem DB-Schema wird folgendes generiert: (Query) Types allAngestellters (inkl.pagination, condition/filter und order) und Felder für jeden Unique Constraint (angestellterByPersnr) Mutations Types createAngestellter, updateAngestellter, deleteAngestellter (inkl. Resolvers)

Inflection = umbenennen von PostgreSQL in Postgraphile Felder. Es macht aus \_ camelCase und aus PK eine nodeid, ergänzt Fremdschlüssel-beziehungen (abteilungByAbtnr), Tabellennamen mit Grossbuchstabe.

Filter: {field { [equalTo |greaterThan | lessThan]

### GraphQL Syntax

|  |  |
| --- | --- |
| // Fields mit Parameter  {  human(id: "1000") {  name  height  }  } | // Output  {  "data": {  "human": {  "name":"Luke",  "height": 1.72  }}} |

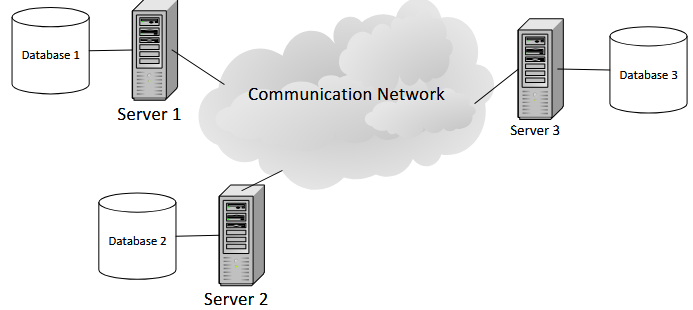
# Verteilte Datenbanksysteme

Verteiltes Datenbanksystem besteht aus kooperierenden DBMS, die auf verschiedenen Computern (Synonyme: Knoten/Node, Site, Station) eines Netzwerks laufen.

Vorher Performance und Transaktionssicherheit

Neu Hochverfügbarkeit (high availability): Verlässlichkeit (reliability) und Verfügbarkeit (availability)

* Jeder Knoten hat eine autonome Verarbeitungsfunktionalität und kann lokale Applikationen ausführen
* Jeder Knoten partizipiert an globalen Applikationen, die Datenzugriff auf die verschiedenen Nodes benötigen



Ein verteiltes DBMS ist eine logisch integrierte Sammlung von Daten, die physikalisch verteilt über die Knoten eines Computernetzwerk sind. Möglicherweise mit unterschiedlicher Software und lokalem Schema.

Jeder Knoten hat eine autonome Verarbeitungsfunktionalität und kann lokale Applikationen ausführen. Jeder Knoten partizipiert an globalen Applikationen, die Datenzugriff auf die verschiedenen Nodes benötigen. Jeder Node = Server + DB, Server zum Teil integriert.

### Anforderungen an verteilte DBMS

1. Zugriff auf übers Netz verbundene Systeme (mit SQL/MED «SQL Management of External Data» = FDW «Foreign Data Wrapper»)  
2. Verwaltung der verteilten und replizierten Daten im DBMS Katalog -> Datenverteilungs-Transparenz: Man sieht Query nicht an, dass sie auf verschiedenen Knoten ausgeführt wird. 3. Transaktion Manager mit mehreren Systemen: Atomarität von verteilten Transaktionen. 4. Ausführungsplaner mit mehreren Systemen.

### Homogen und Heterogen

Homogen globales Schema & identische SW, Alle Knoten wissen voneinander und arbeiten zusammen. Erscheint gegenüber dem Benutzer als ein System. | Heterogen unterschiedliche SW (Problem verteilte Transaktionen) und Schemas (Problem verteilte Queries). Knoten wissen evtl. nichts voneinander und können nur beschränkte Funktionalitäten für die Kooperation anbieten.

### Verteilung von Datenspeicher / Memory / CPU

Tightly coupled einheitliches Schema, Loosely coupled mehrere Firmen |

Shared Everything monolithisches System vs. Shared Nothing

Scaling up vs Scaling out (HW verbessern)

Bewährt: Scaling out mit Shared Nothing auf verteilten "Nodes"/DBMS Instanzen.

### Verteilte Datenhaltung

Replikation die gleichen Daten in mehr als einem Knoten zu speichern (ist redundant) +Höhere Verfügbarkeit + schnellere Query(parallelität) +reduzierter Datentransfer -höhere Updatekosten -komplexe Synchronisation. Implementation: in PostgreSQL Write-Ahead Logging oder Physical Streaming Replication, MongoDB

Partitionierung DB in logische Fragmente aufzuteilen, die dann in verschiedenen Knoten gespeichert werden können

erlauben, horizontale Partitionierung=Sharding (wie MongoDB, gleiche Spalten pro Tabelle) oder vertikale Partitionierung (wie Wide Column Stores, gleiche Zeilen in Tabelle), Wiederholung Id (Column Storage).

Allokation der Prozess, Fragmente=Shards (oder Replikas davon) den Knoten=Stationen zuzuordnen

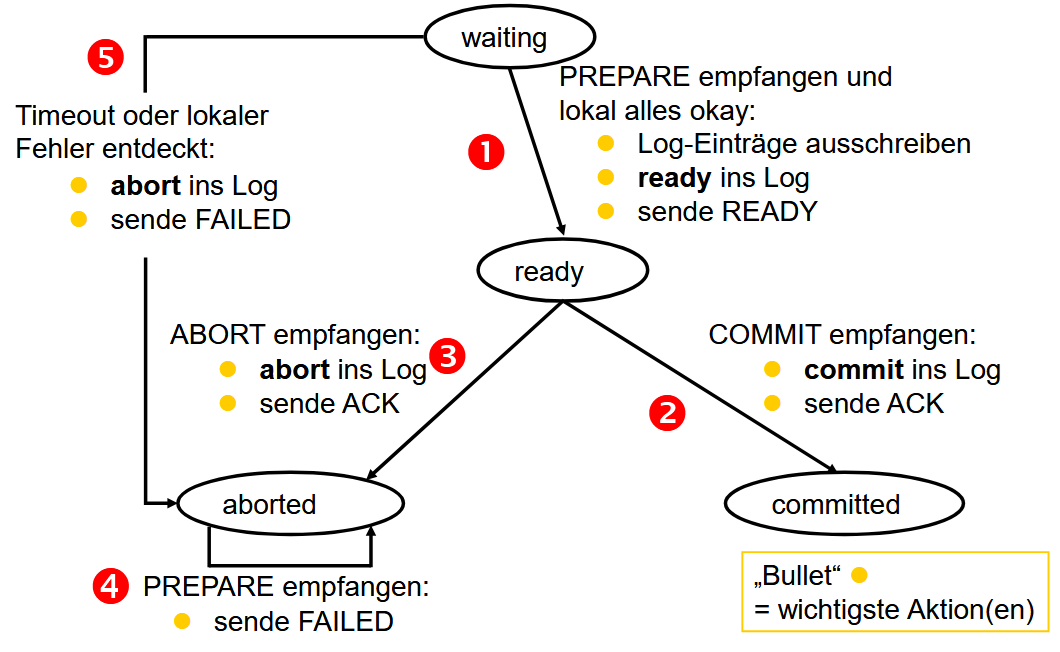
Transparente verteilte Datenhaltung

Der Benutzer sieht eine globale Sicht. Benutzer kennt weder die Fragmente noch die Replika. Queries werden auf der Relation definiert, nicht auf den Fragmenten.

Transaction Manager

## Verteilte Anfrage-Verarbeitung

### Distributed Query Processing

Horizontale (range) Partitionierung Tupels mit salaer < 5000 in Zürich, Tupels mit salaer >=5000 in Genf

Vertikale Partitionierung Tupels mit persnr, und name in Zürich, Tupels mit persnr, salaer in Genf | Verwandt mit Wide Column Stores

Zentralisierte DBMS Kosten der Query bestimmt von Anzahl Disk-I/O.

Verteilte DBMS Kosten bestimmt von Anzahl Netzwerk I/O's und Performance Gewinn durch paralleles Abarbeiten auf mehreren Knoten.

Resource Manager

### Optimierung von verteilten Anfragen

Predicate Pushdown Operationen wie WHERE, JOIN, AGGREGATE werden nicht auf dem lokalen Server, sondern direkt auf dem Remote-Server ausgeführt. =Aggregate Pushdown =Filter pushdown

Fetch as needed bei JOINs   
- Daten vom Remote-Server werden erst dann geholt, wenn sie im Join wirklich gebraucht werden. - Statt alle Daten auf einmal zu laden ("eager fetching"), wird nur abgefragt, was durch die Join-Operation tatsächlich benötigt wird.

## Verteilte Transaktionen (Operationen)

2PC, Raft und Paxos sind Protokolle und Algorithmen, um Datenkonsistenz und Fehlertoleranz sicherzustellen.

Transaktion auf einer Datenbank ist immer lokal und an eine Session gebunden, Die verteilten (lokalen) Transaktionen müssen koordiniert werden

Paxos Transaktionen sind hier dann gültig, wenn ein Mehrheitskonsens erreicht wird, wodurch die Fehlertoleranz gegenüber 2PC verbessert wird

Raft Vereinfacht Konsens durch explizite Führungsrollen, klare Log-Replikation und ein leicht verständliches Fehler- und Wahlmanagement — ohne auf die Sicherheitseigenschaften von Paxos zu verzichten.

2PC-Protokoll Sichert atomare Ausführung von verteilten (lokalen) Transaktionen.

### 2PC

Transaction Manager (**TM**) ist Node, der Transaktionen über mehrere andere Nodes (**RM**) koordiniert. Resource Manager (**RM**) sind Nodes mit lokalen Transaktionen. | Aufbau SW-Systeme: Application Systeme = TM, DBMS (Server/Message Queues) = RMs

Vorteil: Bewährt, vielfach implementiert, Atomarität und Konsistenz.

Nachteil: aufwändige Inserts/Update (langsam), skaliert schlecht, blockiert.

➔ nur Einsetzen wenn sich die Komplexität lohnt

### Phasen von 2PC

Phase 0: Update schreiben (alle führen Transaktion aus)

Phase 1 Prepare to Commit: TM fragt RM an, ob sie die Transaktion mit COMMIT abschliessen können. Falls ja (Ready ins Log) RM gehen in Zustand Prepared. Falls Nein Abort

Phase 2 Commit (log und senden): alle RM Ready gesendet: TM schickt Commit an alle RM: Transaktion wird festgeschrieben. Abort oder keine Antwort: TM schickt ABORT an alle RM, Transaktion wird undo. Sobald Transaktion lokal abgeschlossen, schicken alle RM ACK an TM. Wenn er alle erhalten hat (EndTransaction) ins log.

## 2PC-Fehlersituation

2PC muss mit Fehlersituationen umgehen können 🡪 System in einem konsistenten Zustand halten. Hauptproblem des 2PC ist Ausfall des TM. Ausgefallener TM/RM findet im Log vorheringen Zustand.

### Recovery des TM (=Koordinator):

TM Ausfall nach READY 🡪 Blockierung RM.

TM war Zustand EndTransaction (Ende Phase 2) 🡪 nichts zu tun, da keine Subtransaktionen mehr möglich.

TM war Committing oder Aborted (<commit T> oder <abort T> im Log) (Mitte Phase 2) 🡪 Commit oder Abort Meldung an alle RM's.

Keine der obigen Fälle Dann hat der TM das Prepare an seine RM geschickt, aber noch nicht von allen ein Ready oder Abort erhalten. 🡪 Abort an alle RM schicken. Transaktion wird zurückgesetzt.

### Recovery des RM:

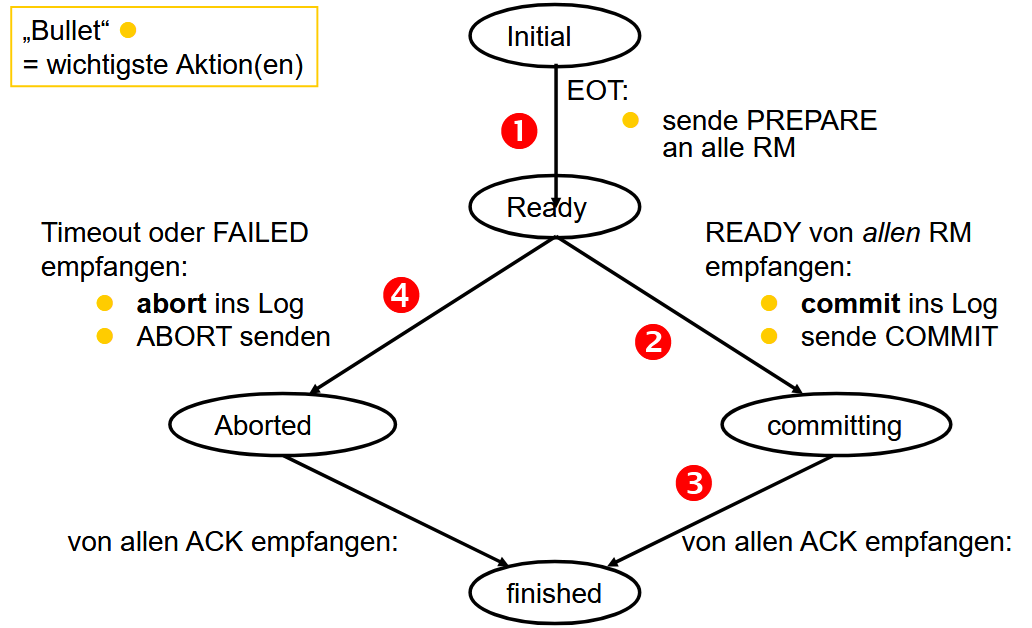
RM im Zustand Ready 🡪 RM muss warten, bis er vom TM weitere Anweisungen erhält. RM fragt TM, was aus Transaktion geworden ist; TM teilt COMMIT oder ABORT mit, was beim RM zu einem Redo oder Undo der Transaktion führt.

RM Zustand Committed (Log enthält <commit T> Eintrag) 🡪 RM führt Redo (T) mit Write-Ahead Log aus. |

RM Zustand Aborted (Log enthält <abort T> Eintrag) 🡪 RM führt ein Undo (T) aus.

Zusätzlich gilt Antwortet ein RM innerhalb eines Timeout-Intervalls nicht auf die <prepare>-Nachricht, gilt der RM als abgestürzt 🡪 der Koordinator bricht die Transaktion ab und schickt eine <abort>-Nachricht an alle Agenten.

Inkonsistenter Zustand

Wenn RM oder RM und TM abgestürzt ist und der Log von dem letzten Zustand ist nicht korrekt im Log gespeichert (sogenannt: unflushed). Deshalb eignen sich diese Zustandsdiagramme nicht für korrektes Recovery.

## NoSQL wide Column Store / column Family

Anwendung: Cassanda, Apache Hbase | hochverfügbar/high availability |RDBMS + Einträge hinzufügen / bearbeiten sehr einfach - Liest pro Query zu viel Daten |gute Komprimierung | eventuelle Konsistenz | keine Transaktionen, Atomicity auf Zeilenebene

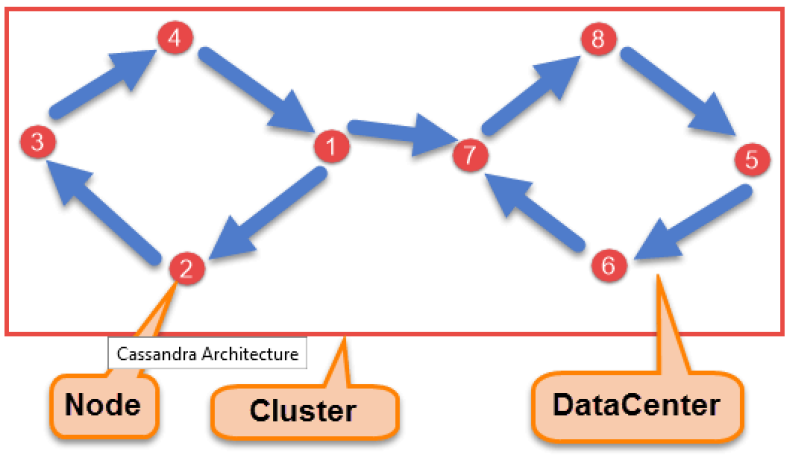
Inhalte werden spaltenweise (und nicht zeilenweise) physisch abspeichert -> effizient, wenn viele Zeilen und wenige Spalten aggregiert werden oder eine Spalte für alle Zeilen ändert.

Column Family (=Table in RDBMS, Bsp. Bestellung ist eine Table) – key ist gemapped zu einem Set von Columns.

### Cassandra

Node: Speicherort (= Replica) für Daten | DataCenter: Collection von Nodes, Cluster: Collection von DataCenters

speichert intern automatisch Zeitstempel zum Eintrag (timestamp / ts)



Base-Unit ist eine Spalte (Key-Value). Eine Row ist eine Collection von Spalten linked zu einem Key. Rows können beliebig viele Spalten haben (nicht wie bei RDBMS, wo jede Spalte min. Null-wert enthalten muss. Eine Super Spalte besteht aus einem Namen und einem value, welche eine map von Spalten ist. In den Clustern gibt es keine Master Nodes. Read und Write kann von jedem Node durchgeführt werden.

Geeignet: Event Logging, Content Management System, Blogging, Web Analytics. Schlecht: ACID Anforderungen, Datenaggregation (SUM, AVG) muss auf Client-Seite geschehen, Daten von allen Rows abfragen

### Cassandra Query Language

CREATE COLUMNFAMILY Customer (

KEY varchar PRIMARY KEY, name varchar, city varchar,

web varchar );

INSERT INTO Customer ( KEY,name,city,web ) VALUES

(‘mfowler ', 'Martin', 'Boston', 'www.martinfo');

SELECT \* FROM Customer; SELECT name , web FROM Customer;

SELECT name , web FROM Customer WHERE city ='Boston’;

# Database Design Pattern

### Data Persistence Pattern

Data Mapper CRUD-Funktionen, die auf Objekten operieren, die in einer DB gespeichert werden. Bildet Entität ab in DB-Schema. Typischerweise OR/Mapper. «Weiss nichts von DB».

Vorteile Flexibilität bei der Datenbankänderung ohne Einfluss auf die Geschäftslogik.

Nachteile Mehr Boilerplate-Code und Komplexität und mehr Entwicklungsaufwand

Beispiele: JPA, SQLAlchemy (Python) Modus, Entity Framework/LINQ

Active Record

Verbindet Geschäftslogik (Klassen) direkt mit Datenzugriff (Entitäten/ Tabellen), was zu einer direkteren und oft einfacheren Umsetzung führt

Speichert Daten in einer Relationalen Datenbank («one object-one record»).

(Klasse & Tabelle eng verbunden.)

Vorteile: Einfachheit, weniger Boilerplate-Code, d.h. schnelle Entwicklung

Nachteile: Weniger flexibel bei komplexen Geschäftslogiken, schwierigere Testbarkeit

Beispiele: Django (Python), Rails (Ruby), "ActiveRecord" (.NET), JOOQ (beide)

### Architektur Patterns

Command-Query Responsibility Segregation (CQRS)

Trennt ein Domänenmodell in zwei separate Teile – ein Read-Modell und ein Write-Modell. Geeignet: Real-time Analytics, Viele Schreibprozesse (>1:100) oder mehr Leseoperationen. – Verteilte Systeme – No-Schema Approach – Gut in Kombination mit GraphQL (dort Query Type für read/Queries und Mutation Type für write). Schemata für Datawarehouses.

Vorteile: Da Schreibbefehle und Abfragen separat behandelt werden, können sie von unterschiedlichen Komponenten unter Verwendung von unterschiedlichen Datenquellen ausgeführt werden, was eine Optimierung beider ermöglicht.

Nachteile: Unnötige Komplexität der Zweiteilung in query/command models – Aufwändige Koordination und Synchronisation der beiden Modelle

Event Sourcing

Reihe von Änderungen im Zustand einer Anwendung → aktueller Zustand kann durch Ablauf aller Events hergestellt werden. Events werden nicht gelöscht, sondern mit Stornoevent storniert. Events sind unveränderlich (ähnlich WAL-Log, "append-only"). Oft in Kombination mit CQRS verwendet.

Vorteile: High-Performance bei Schreiboperationen – History – Replikation

– Logging/Monitoring – Kein Impedance Mismatch (OO→RM) "dank" No-Schema

Nachteile: Keine Validierung der Events, keine IDE-Untersützung (wegen No-Schema) – Schemaänderungen (Data Conversion) schwer nachvollziehbar – – Gefahr der fehlenden Separation von Business und Event («Actor») Code – Erzeugen der «Projections» kann zu Verzögerungen führen (oder werden "eventuell Konsistent"), d.h. potentiell langsamere Leseoperationen, Probleme bei der Query-Optimierung

Evolutionary Database Design (Database Change Management)

Ziel ist Integration der DB in den kontinuierlichen Integrations- und - Auslieferungsprozess (Continuous Integration & Delivery) und für den evolutionären Entwicklungsprozess. DB-Design muss nicht zwingend vorab geschehen, sondern gleichzeitig. Alle DB-Artefakte sind mit dem Applikations-Code im Repository (Versionsmanagement). Alle DB-Änderungen sind Migrationen, Jeder SW-Entwickler bekommt eigene DB-Instanz, SW-Entwickler integrieren kontinuierlich DB-Änderungen. Alle DB-Änderungen sind DB-Refactorings.

Beispiele: Alembic (Python), LiquiBase (Java), Flyway (Java)

### Weiteres

Es fehlt horizontale und vertikale Skalierung=scaling up und scaling out

@onetomany oder @manytoone fehler

Mongodb ist unübersichtlich, besser mit Beispiel input und dann funktionen

Diese letzte Seite hier vom cheatsheet, war nicht an der Prüfung gefragt.

