

SQL GRUNDLAGEN DBS1 (DDL UND DML)

```

SELECT [DISTINCT] ... FROM ... [JOIN . ON .]
WHERE ... ORDER BY ... [LIMIT .];
BETWEEN, AND, IN (oder Subquery), OR, <, >, =, !=

```

Common Table Expression Rekursiv

Von 1 Angestellten alle Untergebene 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('HeLo') UPPER('HeLo') LENGTH('Hello')
COALESCE(wohnort, 'unbekannt')
substr(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 und 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 Lifecyclees 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 Engl.) 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 insensitive, gleicher Name für Tabellen&Columns, alle Attribute persistent gemappt (ausser @Transient), Calendar/Date mit @Temporal als Abbildung auf DATE TIME oder TIMESTAMP

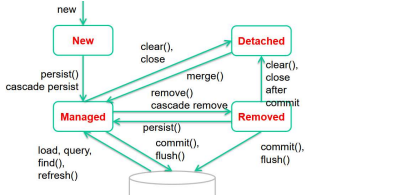
Unterstützte Typen Primitive Typen=Wrappers, Strings, Enums, Byte/Char Arrays, Date/Calendar, beliebige Serializable Klassen (BLOB)

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 get()!
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 Persistence, 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 persistent 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 persistence context.
Managed persistent id and associated with a persistence 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 Persistence Unit

```

Setup (im Instantizierern)
// Bank ist der Name der Unit in persistence.xml
private static EntityManagerFactory factory;
factory = Persistence.createEntityManagerFactory("Bank");
Never 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); explizit aus Persistenz (dh, DB) entfernen
// 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="TIMESTAMP"Z

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 @txo(cascade = CascadeType.PERSIST) / @txo(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 & @ManyToOne
(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; ...
}
public class Address {
  @OneToOne(mappedBy="address", optional=false)
  private BankCustomer customer; // andere Entität
}

```

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)

```

Table bankcustomer
customerid | name | customer_address_id | addressid | street
public class Address {
  @OneToOne
  @JoinColumn(name="customer_address_id",
    referencedColumnName="addr_.id", insertable=false,
    updateable=false)
  private BankCustomer customer; ...
}

```

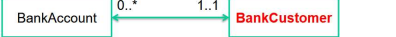
ManyToOne-Beziehung (BankAcc. 0.* → 1 BankCustomer)

```

public class BankAccount {
  @ManyToOne(optional=false)
  @JoinColumn(name="customerref") // FK
  private BankCustomer customer8; ...
}

```

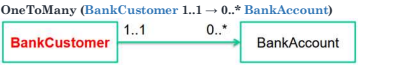
ManyToMany Bidirektional (BankAcc. 0.* ↔ 1 BankCustomer)



```

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

```

bankmanager
managerid | managerref | customerref | bankcustomer | customerid | ...
public class BankManager {
  @ManyToOne
  @JoinColumn(name="customer_manager",
    joinColumns={@JoinColumn(name="managerref")},
    inverseJoinColumns={@JoinColumn(name="customerref")})
  private Collection<BankCustomer> customers =
    new ArrayList<>(); ...
}

```

ManyToMany Bidirektional (0.* : 0.*)
public class BankCustomer { // map Inverse zu Bankm.custs
 @ManyToOne(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(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= 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; }
Superklasse bei TABLE_PER_CLASS
@Entity
@Inheritance(stategy= InheritanceType.TABLE_PER_CLASS)
public class BankCustomer { // ohne abstract
 @Id private int customerId;
 private

DROP ROLE angguest;
Systemprivilegien für Datenbank-Operationen & Systemvariablen
CREATEDB, CREATOROLE, NOCREATEDB, NOCREATOROLE
ALTER ROLE username **WITH** CREATOROLE;
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** (Pernr, Name, Tel) **AS** SELECT Pernr, 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 **INTOI**!

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 Rückgabtyp 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 auferufen

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 ... ELSEIF** 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.xrow%TYPE; 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, nci, 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 ROW, **RETURN NEXT**;

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 (bigint bigint, n bigint, m bigint)
RETURNS bigint **AS** \$\$
DECLARE
x bigint = 0; xx bigint;
BEGIN
if n = 0 **then**
return 1;
end if;
x := bigint % 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 % existient 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 <keine_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 mit TRIGGER Funktion setzen)	Optional (z.B. SELECT-Ergebnisse)	Nein (ausser innerhalb Prozedur)
PERFORM	Funktionen aufrufen, Ergebnisse 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 ctype> 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
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 Resultatmenge
FETCH angcursor **INTO** angsaLaer, angpersnr;
EXIT **WHEN** **NOT FOUND** -- exit when no more row to fetch
salsumme := salsumme + angsaLaer;
RAISE notice 'Angestellter 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 'SaLaersumme: %', salsumme;
END;
RETURN **NEXT** gibt 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 auferufen) | **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 definiert Tga der Funkt. A aufruft -> verändert Tb. Tb definiert Tgb der Funkt. B aufruft -> ändert Tc. Tc hat TG c -> Cascading. Problem wenn TG b 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();
ELSEIF (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 auferufen 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 auferufen mit den Trigger-Fn-Variablen old, new
DROP TRIGGER IF EXISTS trigger ON tablename;
CREATE TRIGGER ourview2Triggerf()
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 & DATENSTRUKTUR

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**
Flieskommazahl **NUMERIC=DECIMAL** → B-Tree, Hash, BRIN
Zeichenketten **TEXT** beliebige Länge, **VARCHAR**(80) maximale Länge, **CHAR**(3) feste Länge → B-Tree, **GIN** **TVECTOR/TSQUERY** → zusätzlich RUM
Binär **BYTEA** (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
Aufzahl-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 PostgreSQL besitzen standardmässig bei 1 aber kann auch im Minus Bereich spezifiziert 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

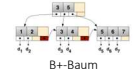
CREATE TABLE sal_emp (
pay_by_quarter integer[4], -- 1 dimensional, "fix"
schedule text[][] -- 2 dimensional variabel/ "unbound"
);
// 2. Unbound default, nicht definiert, beginnt bei 1
//Einfügen von Array Elementen (zwei Möglichkeiten)
INSERT INTO sal_emp VALUES (
'Bill',
{10000, 10000, 10000, 10000, 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}

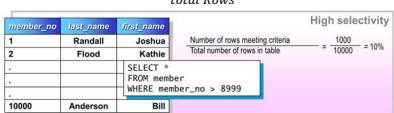
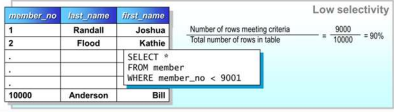
<div>SELECT ARRAY</div> <div><pre>SELECT 1 + (random()*5)::int // Wert zwischen 0-5 FROM generate_series(1,6) ORDER BY 1 //6x ausführen); --gibt z.B. {1,3,5,5,6,6}</pre></div> <div>Array Accessoren</div> <div><p>Index Query: intuitiv wie Koordinate (Start mit 1)</p><pre>select board[1][1] from tictactoe; -- z1 k1</pre><p>Slice Query: Untergrenze:Obergrenze für jede Dimension</p><pre>select board [2:3][1:1] from tictactoe; -- {{z2 k1}, {z3 k1}}</pre><p>Max Board Abkürzung "[2]" vermeiden, besser [1:2]</p><pre>select board[2:3][1:2] from tictactoe ; -- [2]↔[1:2] -- {{z2 k1, z2 k2}, {z3 k1, z3 k2}}</pre><p>Suche mit ANY (any für jedes Array Element)</p><pre>select * from tictactoe where 'z2 k2' = any (board); -- 1;{{z1 k1,z1 k2},{z2 k1,z2 k2},{z3 k1,z3 k2}} //ganze zeile ausgegeben</pre><pre>SELECT array[] FROM table; ==? SELECT array [:] FROM table; SELECT array[1] FROM table; -- Element bei Index 1</pre><div>3x2 Dimensionaler Array</div><div><pre>create table tictactoe as (select 1 as id, ARRAY[['z1 k1', 'z1 k2'], ['z2 k1', 'z2 k2'], ['z3 k1', 'z3 k2']] as board);</pre></div></div>	
<div>DICTIONARIES (KEY VALUE STORES)</div> <div><p>Key Value Pairs, ADT & Datentyp, Bildet "Unique Key" auf Value ab. Für variable, und unvorhersehbare Werte.</p><p>No schema/Schema less, umstritten (consistency, optimization)</p><p>Geeignet für Einfache Datenspeicherung und Datenerfassung, Zeilen mit vielen Attributen, die selten untersucht werden, Semi strukturierte Daten.</p><p>hstore implementiert in PostgresSQL Dictionaries als Datentyp: Keys und Values sind vom Typ TEXT. Unterstützt vom GIST/GIN Index Keys müssen eindeutig sein, sonst werden sie ignoriert.</p><pre>CREATE EXTENSION hstore; CREATE TABLE kvpstable_hstore (id SERIAL PRIMARY KEY, name TEXT, kvp hstore);</pre><pre>INSERT INTO hstoretable (id, kvp) VALUES (1, hstore('key1', 'value1') hstore('key2','value2')) </pre><p>Queries – (Wenn klar ist, dass die linke Operand hstore ist, dann wird der rechte implizit auf hstore gecastet)</p><p>Operatoren -> und <= und <</p><pre>SELECT 'key1>v1, key2>v2':::hstore -> 'key1'; -- v1 -- Does hstore contain key?</pre><pre>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)</pre><pre>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()</pre><div>Helper Funktionen</div><div><pre>SELECT akeys(kvp) FROM List all keys SELECT each(kvp) FROM Get all key value pairs</pre></div></div>	

<div>INDEXE UND SPEICHERSTRUKTUREN + OPTIMIERUNG</div> <div><p>Datentypen und passender Index</p><p>1. Numerisch -> B-Tree, Hash, BRIN</p><p>2. Zeichenketten -> GIN, GIST, Bloom, BRIN, Hash, B-Tree, RUM</p><p>3. Char, Text, Varchar data Type => B-Tree, GIN</p><p>4. Tsvector, Tsqquery => B-Tree, GIN, RUM</p><p>Indexe</p><p>B-tree balanced tree (universell), <= = B+ Baum (verkettet, Blatt zeigt auf nächstes Blatt und d-Daten sind ausserhalb der Blätter im Heap gespeichert),</p><p>Hash (equality search WHERE) =,</p><p>GIST Generalized Search Tree (Array, Volltexteuche, nearest-neighbor), << &< << &< => && SP-GIST space-partitioned GIST (knn (nearest-neighbor), geometrisch), << => << <<</p><pre>SELECT * FROM products WHERE title LIKE 'JASMIN'; CREATE INDEX like_search_tree ON table USING gist(title gist_tmgr_ops);</pre><p>GIN Generalized inverted index (schneller aber mehr Daten als GIST, wenig ändern), Array, JSOVB, Volltext), << &&</p><p>BRIN Block Range Indexes (Range Search) für viele schon sortierte Daten,</p><p>ISAM Indexed Sequential Access Method (wie hash aber kann mehr)</p><p>DROP INDEX [IF EXISTS] indexname;</p><p>Zusammengesetzter Index, Include, funktionaler index</p><pre>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</pre><p>Index für GiST und GIN für Hstore</p><pre>CREATE INDEX kvps_idx ON kvpstable_store USING GIST (col);</pre></div>	
--	--

<div>VACUUM (VERBOSE, ANALYZE) table;</div> <div>Kein index erstellen für primary key, dort wird automatisch ein index erstellt</div> <div>Bitmap Index ordnet Attributwerte als Bitmuster (binär), wenig diskrete Werte, geringe Selektivität, AND/OR. PostgresSQL verwendet aber oft intern einen Bitmap Index (bitmap index scan, bitmap heap scan).</div>	
<div>INDEXE SPEICHERUNG</div> <div><p>Primär-Index Index mit PK (automatisch), immer unique & not null</p><p>Cluster = physischer Speicher wie Index, nur 1 pro Index</p><p>Clustered table in Postgres</p><p>CLUSTER Table USING indexname; Daten werden physisch sequentiell angeordnet gemäss den Index Informationen, One-Time Operation, Changes danach werden nicht automatisch clustered.</p><p>Clustered, integrierter Index in MS SQL</p><p>Daten werden laufend automatisch physisch sequentiell angeordnet gemäss den Index Informationen, die Blätter enthalten die Daten vom Rest der Columns, nicht nur Referenzen</p><p>Nicht-integrierter Index Blätter haben Referenzen auf die Heap-Daten</p><p>Mehrdimensionaler Index z.B. R-Tree für 2D-Geom.</p><p>Mehrstufiger Index Erster Index mit grober Filterung, zweiter Index mit exakter Filterung</p></div>	
<div>PLANER: OPTIMIERER VON POSTGRESQL</div> <div><p>Ziel Erzeugt einen optimalen Ausführungsplan (Execution Plan) für eine SQL Query (geparst als Baumstruktur).</p><p>EXPLAIN: Gibt geplanten Ausführungsplan aus.</p><p>EXPLAIN ANALYZE: Führt Query aus und gibt den Plan mit effektiven Cost: Fiktive Zeiteltime, beschreibt Aufwändigkeit der Operation</p><p>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)</p></div>	
<div>SCANS</div> <div><p>Table Scan Full Table Scan oder Seq Scan: Z.B.: SELECT * FROM angestellter</p><p>Performance: Langsam. Lohnt sich wenn mehr als 80% der Daten im Resultset sind.</p><p>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.</p><p>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.</p><p>Index Only Scan Falls Attribut in der Projektion vorkommt. Wegen MVCC muss der Index alle Datenblöcke 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.)</p></div>	<div></div> <div>B+ Baum</div>
<div>QUERY-ARTEN UND DB-BENCHMARKING</div> <div><p>U.a. zur Charakterisierung von Index-Typen, Performance-Benchmarks, etc.</p><p>Prädikat: Ausdruck (Expression), der eine TRUE/FALSE-Bedingung auswertet – kann bei DB auch UNKNOWN sein.</p><p>Equality Query (bzw. Equality Prädikat): Q, mit Equal-Operator "=" kann mehrere Zeilen zurückgeben.</p><p>Range Query (Bereichsabfrage): Q, mit Operatoren >, >=, <, <=, != BETWEEN. Gibt typischerweise mehrere Zeilen zurück.</p><p>Point Query Query mit dem Equal-Operator "=" gibt immer ein einziges Tupel/Wert zurück.</p><p>Join Query Query mit JOINS.</p><p>Weitere Queries mit Präfix-/Suffix in Texten oder Extremwerte in Zahlen.</p></div>	
<div>JOIN STRATEGIEN</div> <div><p>Nested loop join Die rechte Tabelle wird für jede Zeile in der linken T. gescannt. Ist einfach, kann aber zeitaufwendig sein.</p><p>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).</p><p>Hash Join Zuerst wird die rechte Tabelle geshat (in 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 =...)</p><p>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.</p><p>Schnellster Exists Query</p><pre>SELECT * FROM orders o WHERE EXISTS (SELECT * FROM orderlines ol INNER JOIN products p ON ol.prod_id = p.prod_id WHERE ol.order_id = o.orderid AND P.category = 11);</pre><p>Sub-Select Query ist schneller als Distinct und HAVING Query und CTE ist auch schnell.</p></div>	

<div>QUERY INDEX OPTIMIERUNG</div> <div><p>INTERNE EBENE (INDEXE & OPTIMIERUNG)</p><p>Index Datenstruktur in DBMS, welche Query beschleunigt. Ist im Primärspeicher (Cache)</p><p>Data Page Daten einer Tabelle physisch gespeichert (als Datei). Enthält Zeilen in unsortierter Reihenfolge. Page Überlauf durch Einfüge und Update Operation Page Sets (Blocks) = Collection von Pages</p><p>Heap = Collection von Page Sets / Pages</p><p>Ziel vom Index Abbilden von Werten mit Referenzen (in Memory) auf Daten (Zeilen) und ggf. auf Pages im Sekundärspeicher</p><p>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.</p></div>	
<div>STATISTIK</div> <div></div>	

<div>ANALYSE UND STATISTIKEN ...</div> <div><p>DB Statistiken</p><p>Die DBMS führen Statistiken über Anzahl und Verteilung der Daten in den einzelnen Tabellen. In PostgresSQL unter epg_stats gespeichert.</p><p>Abschätzung Kosten: Kommunikationskosten (Nachrichten, Menge Daten), Berechnungskosten (CPU, Pfadlänge), I/O Kosten (Seitenzugriffe), Speicherkosten (Temp. Speicherbelegung).</p><p>Ziele Datenmodellierung: Einfache und klare Semantik, Redundanzfreiheit</p><p>Ziele Datenbank Tuning: Beschleunigung von Abfragen, hoher Durchsatz</p></div>	
<div>ANFRAGEOPTIMIERUNG / QUERY OPTIMIERUNG</div> <div><p>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 Ausführungsplänen. 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</p><p>Optimzier kann mit Hilfe von Histogramm die Selektivität bestimmen. Für Histogram werden nur Stichproben verwendet.</p><p>Anfrageoptimierung Tipps</p><p>JOIN: Klausel Syntax, Ende Anfrage. Attributwertebereiche beachten: Numerisch schneller als Text vergleich, Schnelle Vergleiche an den Anfang, Index erstellen. Kein Selekt *, Unterabfragen durch JOIN ersetzen. UNION durch Where ersetzten.</p><p>Erstelle Indexes</p><p>auf Kolonnen die häufig in where oder join Bedingungen vorkommen und deren Daten eine hohe Selektivität haben.</p></div>	

<div>SELEKTIVITÄT UND DICHTe</div> <div><p>Dichte</p><p>Dichte ist pro Attribut der durchschn. prozentuelle Anteil von Duplikaten.</p><p>Tiefe Selektivität = hohe Dichte. Unique Index hat tiefe Dichte.</p><p>Anzahl distinct Werte</p><p>Anzahl Tuples</p><p>Selektivität</p><p>Prozentueller Anteil der Tupels in einer Tabelle, die von einer Query geliefert werden.</p><p>Hohe Selektivität bei <= 0.1 -> DBMS/Planer verwendet Index</p><p>Tiefe Selektivität bei > 0.1 -> DBMS/Planer macht Table Scan</p><p>Even distribution Histogramm / height balanced Histogramm</p><p>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.</p><p>ohne Histogramm</p></div>	<div></div> <div>High selectivity</div>
<div></div> <div>Low selectivity</div>	

<div>BAUMSTRUKTUREN (JSON)</div> <div><p>Normalisierte Hierarchie-Tabelle als Baumstruktur</p><p>Normalisiert bedeutet: Die Tabelle ist möglichst frei von Redundanzen (Datenwiederholungen) und erfüllt mindestens die 3. Normalform – jedes Attribut hängt nur vom Primärschlüssel ab.</p><p>Hierarchie bedeutet: Die Tabelle bildet eine über-/untergeordnete Struktur ab (z.B. Chef und Mitarbeiter, Oberkategorie und Unterkategorie).</p><p>Tree: azyklisch verbundener Graph</p><p>Rooted Tree: hierarchial, connected, acyclic Ordered Tree: Ordnung der Kinder definiert Binary Tree: jeder Knoten hat 0-2 Kinder,</p><p>Anwendungen Führungshierarchie, Stützknoten, Kataloge</p><p>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</p></div>	
<div>JSON ALS BAUMSTRUKTUR</div> <div><p>Semistrukturierte Daten: SQL/JSON, SQL/XML</p><p>Daten bzw. Dokumente als Baumstrukturen</p><p>JSON Postgresql: json: speichert exakte Kopie des Textes. Jsonb (ab 9.4): speichert geparstes Binärformat, schneller index. Cast: ::json</p><pre>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 and Delete SELECT to_json('2021-03-09':::timestamp @>) #> '{"'}; text</pre><p>SELECT jsonb_pretty(airport) FROM airports</p><pre>WHERE airport ->> 'name' LIKE 'Zürichh'; jsonb_pretty { "name": "Zürich Airport", "region": "Zurich", "navids": { { "name": "Kloten", "ident": "KLO"}} }</pre><p>Build JSON in Query</p><pre>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;</pre><p>Convert Select result to JSON</p><pre>row_to_json() = to_json() SELECT row_to_json(row(row1, row2, row3)) FROM table; SELECT array_to_json(fields) FROM table;</pre><p>Jsonpath @? und @? supported by GIN Index</p><pre>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');</pre></div>	
<div>JSON PATH JSONB_PATH_QUERY AUS ÜBUNG</div> <div><pre>SELECT jsonbpg(data, '\$' ? (@.customer.name == "Peter Staub")) FROM orders_json; Gleiches Mit wildcard SELECT jsonbpg(data, 'strict \$.articles[*].article ? (\$.articles[*].price > 100)') FROM orders_json;</pre></div>	
<div>NOSQL</div> <div></div>	

<div>RDBMS Relationales Datenbanksystem: Single Point of Failure durch Shared Everything. Scaling out ist aufwändig. keine Objekte: aufwändiges OR Mapping.</div> <div>NoSQL Einfaches API (http), grosse Datenvolumen: Skalierbar. Oft nicht relational, schema frei. Oft Architektur basierend auf BASE ()</div> <div>Datenmodelle</div> <div>Aggregation Stores : Key/Value und Document stores.</div> <div>Graph Stores</div> <div>Column Family (Wide Column Stores)</div>	
<div>CAP, BASE & ACID</div> <div><p>CAP wähle 2 aus</p><p>C - Consistency Daten sind die gleichen auf jeder Replikation und jedem Node; verlangt Atomicity, Transaction Isolation. Was zuerst abgesichert wurde, kommt zuerst an. Nicht wie Consistency in ACID.</p><p>A - Availability Daten müssen immer verfügbar/zugreifbar sein</p><p>P - Partition Tolerance DBMS funktioniert auch bei teilweiseem Ausfall von Netzwerk und Nodes. RDBMS: C-P, NoSQL: P-A, Small Data Sets: C-A</p><p>BASE</p><p>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.</p><p>Soft state (Welcher 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.</p></div>	<div><p>Eventuelle Konsistenz: Das System wird konsistent werden, sobald es keine Eingaben mehr erhält</p><p>Unterschied zu ACID</p><p>ACID = Atomarität, Konsistenz, Isoliertheit und Dauerhaftigkeit</p><p>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.</p></div> <div>MONGODB DOCUMENT STORE</div> <div><p>Database = DB/Schema Collection = Tabelle Document = Zeile/Row</p><p>Fields = Spalte/Column _id (objectid) = row id DBRef = Join</p><ul style="list-style-type: none">- Data Partitioning mit Sharding- Availability mittels Replikation und "Raft"-ähnlich Konsens-Algo. Quorum- JSON als grundlegendes 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<p>Modellierungsentscheidung Ausgewogenheit des Datenmodells.</p><p>Stärken: Prototyping (JSON over http), Webfreundlichkeit, DBaaS/Cloud</p><p>Schwächen: Keine Constraints, keine echten Joins langsam. langsame mangelhafte Transaktionen. Security (default offen im Web).</p><p>+/- einfache Query Sprache, No Schema</p><p>Geeignet: analog Column Stores. Schlecht: atomare cross documem q.</p></div> <div>EMBEDDING VS REFERENCING - MONGODB BEZIEHUNGEN</div> <div><p>Denormalisiert (embedding) bei mongoDB bei 1-to-1 oder 1-to-many ({} als Feld) normalisiert «manual references» Referenzen mit _id als Feld. empfohlen für normalisierte Daten. Many-to-many. Grosse Hierarchien.</p></div> <div>SHARDING (FÜR DB DATA PARTITIONING VON MONGODB)</div> <div><p>Spezialfall von Horizontaler Partitionierung wobei Partitionen auf verschiedene Nodes verteilt werden.</p><p>Ziel gute Verteilung der Dok. = gute Auslastung der Nodes.</p><p>Der Shardkey sollte ein Schlüssel sein, der die Daten gleichmässig verteilt.</p><p>Bei MongoDB Hash auf _id "db.events.createindex({_id: 'hashed'})"</p><ul style="list-style-type: none">- 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</div> <div>CONSISTENCY & REPLICAS</div> <div><p>MongoDB Repliziert: Ausfallsicherheit => Replikation mit mind. 3 Knoten.</p><p>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.</p></div> <div>MONGODB SYNTAX DB CRUD</div> <div><pre>CREATE use mydb -- set current database db.collection.insert({name: 'Aurora', gender: 'f'}); WriteResult({ "inserted": 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("58ef8f135") }) db.uncionns.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": "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</pre></div>

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), Index: 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 Data: Tuple by tuple auf Disk, Zeilen werden nacheinander gespeichert. Optimal für den zeilenweisen Zugriff (z.B. SELECT *). **Vorteile:** Seq. Row Scans gute Bandbreiten ausnutzung, 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. Umzögern **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.

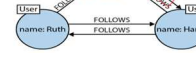
Labeled Property Graph Eine Art von Graph, bei dem die Knoten und die Kanten mit Bezeichnungen (Labels) und Eigenschaften (Properties) versehen werden können. Mächtiger als RDF wegen "Properties"!

RDF Triple Das "Resource Description Framework" (RDF) mit "Triples" 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

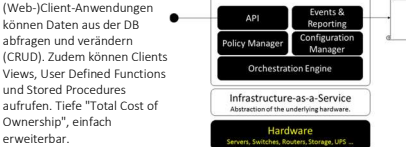


Knoten (.), labeled Beziehungen [.:], Verbindungen --, <-->, -->, <-->
Movie: Label/Klasse | title und released = Attribute/properties
CREATE (alias:Movie {title:'Matrix', released:1999})
MATCH (:Movie {title:'Wall'})<[:ACTED_IN]-(actor:Person)
RETURN actor.name AS actor

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



Pagination Durch Ergebnisse einer (grossen) Datenabfrage blättern Umsetzung mit OFFSET statt cursor, 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/Feeds» darzustellen. | Bei **REST** gibt es verschiedene Endpoints, bei GraphQL ein einziger 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 und nicht GraphQL (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 all Angestellter (inkl.pagination, condition/filter and order) und Felder für jeden Unique Constraint (angestellterByPernr) Mutations Types create Angestellter, update Angestellter, delete Angestellter (inkl. Resolvers)

Infection = umbenennen von PostgreSQL in Postgraphile Felder. Es macht aus _camelCase und aus PK eine nodeid, ergänzt Fremdschlüssel-beziehungen (abteilungByAbtrn), Tabellennamen mit Grossbuchstabe. Filter: {field | [equalTo | greaterThan | lessThan]

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

VERTEILTE DATENBAUSYSTEME

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 Updatelkosten-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.

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 | Verwand 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.

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 vorherigen 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.

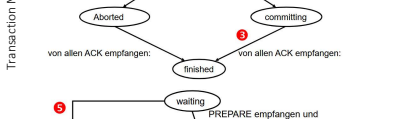
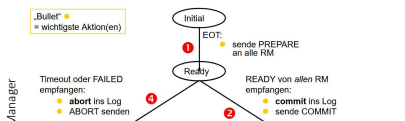
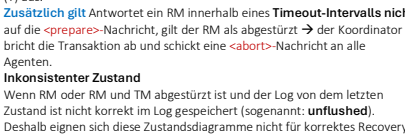
Recovery des RM:

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 TM 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: Cassandra, Apache Hbase | hochverfügbar/high availability | [RDBMS + Einträge hinzufügen / bearbeiten sehr einfach - Liest pro Query zu viele Daten | geringe Komprimierung | eventuelle Konsistenz | keine Transaktionen, Atomicity auf Zeilenebene

Inhalte werden spaltenweise (und nicht zeilenweise) physisch abgespeichert -> 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 ('mfwoler', '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 ORM/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, schwierigerer 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-Unterstützung (wegen No-Schema) – Schemaänderungen (Data Center) 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-Refactoring.

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

Weiteres
Begriff

	RDBMS	NoSQL-System
Complex / multiple joins	X	
Schema flexibility		X
High performance	X	X
Different transaction levels	X	
Linear scalability		X
Advanced security needs	X	
Advanced query functionality needs		X
Efficient calculation of aggregate functions		X