# o/r-mapping

Programme sind meist objektorientiert während DB meist relational sind. Diese semantische Lücke kann mittels O/R-Mapping geschlossen werden. Damit können relationale Daten als Verbund von Objekten geladen werden und Änderungen an persistenten Objekten gespeichert werden.

**Top down (Forward Engineering):** Erstelle Business-Modell und erzeuge DB-Schema

**Bottom up (DB first):** Erzeuge aus existierended DB-Schema ein Business-Modell

**Middle out (Mapping first):** Erstelle Metadaten und generiere Java und DB-Schema

**Meet in the middle:** Aus existierenden Business-Modell und DB-Schema Metadaten erstellen

## java persistance api (jpa)

**Anbindung an DB**: Konfigurations-File persistence.xml (obligatorisch). Beinhaltet Informationen bezüglich JDBC-Connection Parameter (Treiber, URL, User und Passwort)

Diagram

Description automatically generated**Entity als persistente Objekte:** POJO (Plain Old Java Object), ohne Basisklasse oder Interface, Klasse nicht final, Id-Feld notwendig für PK Abbildung und Parameterloser Konstruktor

**Persistence Unit:** Bestimmt JPA Provider und DB-Anbindung. Menge von Entity Klassen und deren Mapping. Durch persistence.xml beschrieben

**EntityManagerFactory:** Verwaltet eine Persistence Unit

**Persistence Context:** Verwaltet eine Menge von Entity Instanzen zur Laufzeit. Entites ist entweder managed (in Persistence Context) oder new/detached (in keinem Context). Definiert transaktionelle Session

Diagram

Description automatically generated**Entity Manager:** Verwaltet Persistence Context und bietet Lifecycle-Operationen für Entity Instanzen an.

**Entity States:**

|  |  |
| --- | --- |
| New | Keine persistente Idendität und noch nicht mit Persistence Context verbunden |
| Managed | Haben persistente Identität und sind mit Persistence Context verbunden |
| Detached | Haben persistente Identität aber nicht mit Persistence Context verbunden |
| Removed | Haben persistente Identität, sind mit Persistence Context verbunden und für die Entfernung aus Datenspeicher vorgesehen |

**Persistente Objekt:** POJO ohne Basisklasse/Interface, nicht final, hat ID-Feld und Parameterloser Konstruktor, bildet eine DB-Entität ab

**Session:** Übersetzt DB-Konzept des PK in das Konzept von Objekt-Instanzen und verwaltet dessen Identität. Objekte werden durch Id im Cache verwaltet und damit identifiziert.

A screenshot of a computer

Description automatically generated with low confidence

## mapping

**Abbildung zur DB:** OR-Mapping-File orm.xml (optional) oder Annotation im Java-Code z.B @Entity. Zweite Lösung ist nicht mehr POJO.

**Priorität:** 1. orm.xml 2. Annotations im Java Code 3. Default Mapping in JPA-Implementation

**Regeln:** Mapping mit gleichem Namen (Case Insensitiv), Alle Attribute werden persistent gemappt (Ausnahme @Transient), Calendar/Date Java-Typ braucht Zusatzinformation (@Temporal)

**Access Types:**

|  |  |
| --- | --- |
| Field Access | Property Access |
| DB-Attribute direct in Fields abbilden | DB-Attribute über Getter/Setter abbilden |
| @Id private long id; | private long id; @Id public long getId() {  return id;} |
| Statt getXY() auch isXY() möglich |

**Klasse mit @Entity:** Kann erben/vererben, Interfaces implementieren und abstrakt sein.

Muss public/protected Konstruktor ohne Parameter haben, PK Angabe erforderlich (@Id),

keine final Fields/Methoden, Fields sollen private/protected sein

## Entity relations

Icon

Description automatically generated with medium confidence

**1:1 Beziehung**

@Entity   
@Table(name = "bankcustomer")  
public class BankCustomer {  
 @OneToOne(optional = true) *// Nullabe* @JoinColumn(name = "customer\_addressid") *// FK* private Address address;  
 ...

Shape, rectangle

Description automatically generated

Table bankcustomer Table address

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| customerid | name | customer\_addrid |  | addrid | street | … |
| 10 | Bob | 2 |  | 2 | 10 Mark | … |
| 11 | Alice | 3 |  | 3 | 10 Main | … |
| 12 | Anonym | NULL |  |  |  |  |

**A picture containing icon

Description automatically generated**

**Bidirektionale 1:1 Beziehung**

Graphical user interface, text, application, chat or text message

Description automatically generated@Entity  
@Table(name = "bankcustomer")  
public class BankCustomer {  
 @OneToOne *// default optional = true* @JoinColumn(name = "customer\_addressid") *// FK* private Address address;  
 ...  
@Entity  
@Table(name = "address")   
public class Address {  
 @OneToOne(mappedBy = "address",*// Property in BankCustomer*  
 optional = false)  
 private BankCustomer customer;  
 …

A screenshot of a computer

Description automatically generated with low confidence

**Inverse 1:1 Beziehung**

Text

Description automatically generated@Entity  
@Table(name = "address")  
public class Address {  
 @OneToOne  
 @JoinColumn(name = "addressid",  
 referencedColumnName = "customer\_addressid",  
 insertable = false, updatable = false)  
 private BankCustomer customer;

Shape, rectangle

Description automatically generated

Table bankcustomer Table address

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| customerid | name | customer\_addrid |  | addrid | street | … |

A picture containing shape

Description automatically generated

**N:1 Beziehung**

Shape, rectangle

Description automatically generated@Entity  
@Table(name = "bankaccount")  
class BankAccount {  
 @ManyToOne(optional = false)  
 @JoinColumn(name = "customerref") *// FK* private BankCustomer customer;  
 …  
Table bankcustomer Table bankcustomer

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| accountid | balance | customerref |  | customerId | name |

**A picture containing shape

Description automatically generated**

**Bidirektionale Many:1 Beziehung**

Graphical user interface, text, application, chat or text message

Description automatically generated@Entity  
@Table(name = "bankcustomer")  
class BankCustomer {  
 @OneToMany(mappedBy = "customer")  
 private Collection<BankAccount> accounts = new …



**1:N Beziehung**

Background pattern

Description automatically generated@Entity  
@Table(name = "bankcustomer")  
public class BankCustomer {  
 @OneToMany  
 @JoinColumn(name = "customerref", *// FK* referencedColumnName = "customerid")  
 private Collection<BankAccount> accounts = …

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| customerid | name |  | accountid | balance | customerref |

**Icon

Description automatically generated with low confidence**

**N:N Beziehung**

@Entity  
@Table(name = "bankmanager")  
class BankManager {  
 @ManyToMany

@JoinTable(name = "customer\_manager",

joinColumns = {@JoinColumn(name = "managerref")},  
 inverseJoinColumns = {@JoinColumn(name = "customerref")})  
 private Collection<BankCustomer> customers = …

Background pattern

Description automatically generatedcustomer\_manager

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Background pattern  Description automatically generated |  |  | managerref | customerref |  |  |  |
| Bankmanager |  |  |  |  |  | Bankcustomer |  |
| managerid | … |  |  |  |  | customerid | … |

**Shape

Description automatically generated**

**Bidirektionale N:N Beziehung**

Text

Description automatically generated with medium confidence@Entity  
@Table(name = "bankcustomer")  
class BankCustomer {  
 @ManyToMany(mappedBy = "customers")  
 private Collection<BankManager> managers = new …

@Id  
@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
private long managerid;

@Temporal(TemporalType.*DATE*) *// TIME auch möglich*

private Date utilDate;

## ladestrategien

**Eager Loading:** Target Entity wird direct mit der Beziehung geladen. Default bei 1:1 und N:1

**Lazy Loading:** Laden der Entity beim erstem Zugriff auf Beziehung. Default bei 1:N und N:N

Explizite Steuerung mit Annotations

@OneToOne(fetch = FetchType.*LAZY*)  
@OneToMany(fetch = FetchType.*EAGER*)

## vererbung

**Joined Table Mapping (3a. Je eine Tabelle pro Sub- und Superklasse)**

Subklassen werden auf eigene Tabelle abgebildet und enthalten alle Attribute.

Ermöglicht einfache Zugriffe auf Tablle ABER Semantikverlust, Overlapping nicht möglich und Schlüssel-Eindeutigkeit muss ggf. Seperat über mehrere Tabellen kontrolliert werden.

BankCustomer

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CustomerId | Name | Type |  |  |  |
| 1 | Bob | Retail |  |  |  |
| 2 | Alice | Private |  | RetailBankCustomer |  |
|  |  |  |  | CustomerId | Fees |
|  |  |  |  | 1 | 100 |
|  |  |  |  |  |  |
|  |  |  |  | PrivateBankCustomer |  |
|  |  |  |  | CustomerId | EliteOffer |
|  |  |  |  | 2 | SailingTour |

@Entity  
@Inheritance(strategy = InheritanceType.*JOINED*)  
@DiscriminatorColumn(name = "type")  
public abstract class BankCustomer {  
 @Id private int customerId;  
 private String name;  
}  
@Entity  
@DiscriminatorValue("Retail")  
public class RetailBankCustomer extends BankCustomer {  
 private int fees;  
}  
@Entity  
@DiscriminatorValue("Private")  
public class PrivateBankCustomer extends BankCustomer {  
 private String eliteOffer;  
}

**Table Per Class Mapping (3b Je eine Tabelle pro Subklasse)**

Superklasse wird auf eine Tabelle abgebildet, welche zusätzlich ein diskriminierendes Attribut enthält, das den jeweiligen Typ der Subklasse spezifiziert.

Redundanzfrei, overlapping möglich ABER Abfragen sind komplex da viele Tabellen und ein zusätzliches Typ-Attribut notwendig.

RetailBankCustomer PrivateBankCustomer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| CustomerId | Name | Fees |  | CustomerId | Name | EliteOffer |
| 1 | Bob | 10 |  | 2 | Alice | SailingTour |

@Entity  
@Inheritance(strategy = InheritanceType.*TABLE\_PER\_CLASS*)  
public abstract class BankCustomer {  
 @Id private int customerId;  
 private String name;  
}  
@Entity  
public class RetailBankCustomer extends BankCustomer {  
 private int fees;  
}  
@Entity  
public class PrivateBankCustomer extends BankCustomer {  
 private String eliteOffer;  
}

**Single Table Mapping (3c. Superklasse Tabelle)**

Superklasse-Tabelle speichert alle Attribute, auch die der Subklassen. Enthält auch ein diskriminierendes Attribut, welches den Typ der Subklasse spezifiziert.

Zugriffe sind sehr einfach ABER sehr einfach 3 NF verletzt, viele NULL Werte

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Typ | EliteOffer | Fees |
| Bob | Retail | NULL | 100 |
| Alice | Private | SailingTour | NULL |

@Entity  
@Inheritance(strategy = InheritanceType.*SINGLE\_TABLE*)  
@DiscriminatorColumn(name = "type")  
public abstract class BankCustomer {  
 @Id

private String name;  
}

@Entity  
@DiscriminatorValue("Retail")  
public class RetailBankCustomer extends BankCustomer {  
 private int fees;  
}  
@Entity  
@DiscriminatorValue("Private")  
public class PrivateBankCustomer extends BankCustomer {  
 private String eliteOffer;  
}

## anfragen mit jpql

Anfragesprache in Analogie zu SQL, operiert aber auf EntityModell, nicht DB-Modell. Wird von JPA Provider in SQL übersetzt.

**Positional Parameters**

Query pp = em.createQuery("select a from BankAccount a where a.customer.name like ?1 and a.balance >= ?2");  
pp.setParameter(1, "Bob");  
**Named Parameters**

Query np = em.createQuery("select a from BankAccount a where a.customer.name like :name and a.balance >= :lower");  
np.setParameter("lower", 200);

**Dynamic Queries**

Query während Laufzeit gebaut und geprüft

Query query = em.createQuery("SELECT c FROM BankCustomer c WHERE c.name LIKE :customerName");  
query.setParameter("customerName", name);  
query.setMaxResults(1000);  
List<BankCustomer> list = query.getResultList();

**Named Queries**

Statisch bekannt und können von JPA Provider vorgeparst und optimiert werden. @NamedQuery muss bei Entity-Klasse annotiert werden

@NamedQuery(name = "CustomerSearch", query =  
 "SELECT c FROM BankCustomer c WHERE c.name LIKE :customerName")  
@Entity  
public class BankCustomer { … }  
Query query = em.createNamedQuery("CustomerSearch");  
query.setParameter("customerName", name);

**N+1 Problem**

Bei 1:N oder N:N Relationen können Queries riesige Mengen an Objekten laden, da diese weitere Relationen haben können, welche auch geladen werden.

Lösung: FetchType.LAZY und nur Abfragen was nötig ist, d.h Abfrage anpassen

# Stored procedures

Stored Procedures ist der Oberbegriff für User Defined Function, Subroutine und Methoden. Konzept des Procedure Calls. Programme gespeichert (stored) bei oder nahe an den Daten. Grundlagen für Datenbank-Triggers.

**Warum SP:** Domänen-Logik (Funktionen, Datenkapselung), Performanz da Funktionen nahe an Daten, Triggers, Seperation of Concern/Duties

**Wann keine SP:** Wenn Views oder OR Mapper genügen

**Nachteile:** Macht Software-Entwicklungsprozess um ein Element komplexer, Wartbarkeit

**PL/SQL:** Prozedurale Sprache mit eingebettetem SQL; Case insensitive; Default Values

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

**SP Compiler:** Code wird beim Aufruf geparsed und als Pseudocode in der DB gespeichert. Erst bei der ersten Ausführung wird der volle Syntax gecheckt.

Graphical user interface, application

Description automatically generated

**Parameter:**

**IN:** Call by Value, Variablen/Ausdrücke als Argument

**OUT:** Call by Reference, nur Variablen als Argument

**INOUT**: Beides

**RETURN:** Entweder Typ oder Tabelle als Rückgabewert

**Beispiel Function:**

create or replace function *test\_fn*(inout m integer)as $$  
DECLARE  
 CurrAbtNr integer;  
 AngCursor CURSOR (AbtId Abteilung.AbtNr%TYPE) FOR  
 SELECT *Salaer*, *PersNr* FROM Angestellter  
 WHERE Angestellter.AbtNr = AbtId  
 FOR UPDATE;  
 SalSumme DECIMAL (8, 2) := 0;  
 MinSalaer DECIMAL := 7100;  
 AngRec RECORD;  
 BEGIN  
 FOR CurrAbtNr IN 1..2 LOOP  
 OPEN AngCursor (CurrAbtNr);  
 LOOP */\*Iteration über Resulatmenge\*/* FETCH AngCursor INTO AngRec;  
 EXIT WHEN NOT *FOUND*;  
 IF AngRec.*Salaer*<MinSalaer THEN  
 UPDATE Angestellter SET *Salaer* = MinSalaer  
 WHERE CURRENT OF AngCursor;  
 END IF;  
 SalSumme := SalSumme + AngRec.*Salaer*;  
 END LOOP;  
 CLOSE AngCursor;  
 END LOOP;  
END; $$ LANGUAGE plpgsql;

**Beispiel Procedure:**

create or replace procedure *swap*(inout arg1 bigint, inout arg2 bigint) as $$  
 declare temp bigint;  
begin  
 temp:=arg1;  
 arg1:=arg2;  
 arg2:=temp;  
end;  
$$ language plpgsql;  
call *swap*(2,3); *-- To call a procedure, use CALL.*

**Beispiel mit Return:**

CREATE Function ProjektMA(ProjBezeichnung Projekt.Bezeichnung%TYPE)

    RETURNS TABLE

        (   Bezeichnung CHAR,

            ProjLeiter  VARCHAR,

            ProjMA      VARCHAR ) AS

$$BEGIN RETURN QUERY

        SELECT P.Bezeichnung, P.ProjLeiter, P.ProjMA

        from ProjektUebersicht P

        where P.Bezeichnung = ProjBezeichnung;

END;

$$LANGUAGE plpgsql;

## cursor

Select-Statement liefert in der Regel eine Menge von Tupels (Result Set). Der Cursor erlaubt den sequentiellen Zugriff auf die einzelnen Tupel des Result Set.

Diagram

Description automatically generated

**Unbound Cursors:** Curser kann mit jedem Query verwendet werden

DECLARE curs1 refcursor;

**Bound Cursors:** Ein voll spezifizierter Query ist am Cursor gebunden

DECLARE curs2 CURSOR FOR SELECT *\** FROM tenk1;

**Parametrisierter Cursor:** Query mit Parameters, welcher beim öffenen ersetzt wird.

DECLARE curs3 CURSOR (key integer) FOR SELECT *\** FROM tenk1 WHERE id= *key*;

**Beispiel Cursor:**

CREATE FUNCTION *calcSum*(chefnr INTEGER) RETURNS NUMERIC AS $$  
DECLARE  
 SalSumme DECIMAL(10, 2) := 0;  
 curs1 REFCURSOR; *-- unbound*  
BEGINOPEN curs1 FOR EXECUTE 'select salaer from angestellter where chef = $1' USING chefpersnr;LOOP  
 FETCH curs1 INTO angsalaer;  
 EXIT WHEN NOT *FOUND*;  
 SalSumme := SalSumme + angsalaer;  
 END LOOP;

CLOSE curs1;  
 RETURN SalSumme;  
END; $$ LANGUAGE plpgsql;

## triggers

Trigger sind DB-Objekte und immer einer Tabelle zugeordnet, werden in SP programmiert, haben keine Parameter, können nicht direkt aufgerufen werden, werden vom DBMS beim Eintreten eines Events aufgerufen und haben bei der Ausfürung die Rechte des Owners.

**Verwendung:** Implementation von komplexen Konsistenzbedingungen, Sichertheit z.B Änderung von Daten nur während Arbeitszeit an Wochentagen, berechnen von abgeleiteten Attributen, sammeln von Statistik- und Logdaten und für Updatable Views.

Diagram

Description automatically generated**Nachteile:** Machen DB tendenziell langsamer und schwerer wartbar (load, dump/restore)

**Event:** INSERT, UPDATE, DELETE, TRUNCATE

**Before Trigger:** Vor Query ausgeführt und kann Operation überspringen oder Zeile ändern

**After Trigger:** Nach Query ausgeführt und sieht alle Änderung des Events inklusiv Änderungen von anderen Triggers

**Reihenfolge der Ausführung von Triggers:**

1. Alle BEFORE Statement Trigger
2. Für jedes betroffene Tupel
   1. BEFORE ROW Trigger ausführen (Abbruch wenn Trigger-Function null liefert)
   2. Bearbeite Tupel (Locks werden gehalten)
   3. AFTER ROW Trigger ausführen
3. Alle AFTER Statement Trigger

**Trigger-Fn.-Variablen:**

|  |  |
| --- | --- |
| Variablename | Funktion |
| TG\_NAME | Name des Triggers |
| TG\_WHEN | BEFORE oder AFTER |
| TG\_LEVEL | ROW oder STATEMENT |
| TG\_OP | INSERT, UPDATE, DELETE, TRUNCATE |
| TG\_RELID | OID der Tabelle |
| TG\_RELNAME | Name der Tabelle |
| TG\_TABLE\_SCHEMA | Schema der Tabelle |

CREATE OR REPLACE FUNCTION *dt\_trigger\_func*()  
 RETURNS TRIGGER AS $$  
BEGIN  
 IF (*TG\_OP* = 'INSERT') THEN  
 *NEW*.*creation\_date* := *NOW*();  
 ELSIF (*TG\_OP* = 'UPDATE') THEN  
 *NEW*.*modification\_date* := *NOW*();  
 END IF;  
 RETURN *NEW*;  
END $$ LANGUAGE plpgsql;

CREATE TRIGGER dt\_trigger  
 BEFORE INSERT OR UPDATE  
 ON mytable  
 FOR EACH ROW  
EXECUTE PROCEDURE dt\_trigger\_func();

**Instead-of-Trigger:** In Postgres nur auf Views (nicht auf Tabellen), die dann SP auslösen, werden anstelle der ursprünglichen SQL-Operation ausgeführt, können für Modifkationen auf Tables und Views definiert werden, leiten Modifikationen (insert, update, delete) auf Views zur darunterliegenden Tabelle weiter.

CREATE OR REPLACE FUNCTION *abtleiterinfo\_update\_abtchef\_fn*()  
 RETURNS TRIGGER AS $$  
BEGIN  
 IF *TG\_OP* = 'UPDATE' THEN  
 UPDATE abtleitung  
 SET *abtchef*=NEW.*abtchef* WHERE *abtnr* = OLD.*abtnr*;  
 RETURN NEW;  
 END IF;

RAISE EXCEPTION 'Example exception'  
END; $$ LANGUAGE plpgsql;  
CREATE TRIGGER abtleiterinfo\_update\_abtchef  
 INSTEAD OF UPDATE  
 ON abtleiterinfo  
 FOR EACH ROW  
EXECUTE PROCEDURE *abtleiterinfo\_update\_abtchef\_fn*();

**Updatable Views:** Views sind externe Sichten für Applikation (Kapselung) und verringern die Komplexität von Queries. Views sind bei Modifikationsoperationen möglicherweise nicht mehr eindeutig. Eine View ist automatisch aktualisierbar, wenn die View genau einen Eintrag in der FROM-Klausel hat, keine WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET, UNION, INTERSECT, EXCEPT enthalten. Die SELECT-Liste dark keine Aggregationen, Window-Function oder SET-returning Funktion enthalten.

CREATE VIEW ourview1 AS  
SELECT *name*, *abtnr*, *salaer*FROM angestellter  
ORDER BY *abtnr*, *salaer* DESC;

**Materialized View:** Zwischengespeicherte View, kann sofort ohne Berechnung aufgerufen werden, wird über eine im DBMS gespeicherte Abfrage definiert.

CREATE MATERIALIZED VIEW aam AS SELECT *\** FROM aa WHERE *a* <= 500000;  
REFRESH MATERIALIZED VIEW aam;

**Temporäre Tabellen:** Erzeugt eine Tabelle, welche am Ender der Transaktion oder Session automatisch gelöscht wird. Verwendet für komplexe Auswertunge, die Zwischendaten halten müssen.

CREATE TEMPORARY TABLE tmp AS  
SELECT *generate\_series*(1,100000) AS a;

# datenstrukturen

**Datentyp:** String/Text, Zahl/Ganzzahl/Fliesskommazahl, Boolean, Datum/Zeit, Aufzähltyp

**Datenstruktur:** Dient der Speicherung und Organisation von Daten

**Datensatz:** Gruppe von inhaltlich zusammengehörigen Elementen (Werte, Felder) bzw. Basis-Datentypen

## abstrakter datentyp (adt)

**Abstrakter Datentyp (ADT):** Datentyp der durch die Opererationen darauf definiert ist, die Zugriff und Verwaltung ermöglichen und realisieren (Liste implementiert als Array oder Stack implementiert als Liste)

**Funktionen:** Konstruktoren, Accessoren sowie Insert/Update/Delete

**Create Domain:** Benutzerdefinierten Datentyp für Datenschema und SP mit Constraints wie NOT NULL, CHECK

create domain contact\_name as  
varchar(255) not null  
check (value !~ '\s');

**Create Type:** Erstellt einen zusammengesetzten Datentyp für Datenschema und SP ohne Constraints-Angaben

create type easteregg as (  
 outer: text,  
 inner: text,  
);  
create type traffic\_light\_t as  
enum('red', 'yellow', 'green');

## collections

Gruppierung einer variable Anzahl von Datenelemeneten im allgemeinen des gleichen Typs oder abgeleitet von einem gemeinsamen Obertyp. Abstrakte Datenstruktur, die keine spezifische Implementierung vorschreibt, obwohl es oft eine konventionelle Implementierung gibt.

**Lineare Kollektionen:** Listen, Arrays, Priority Queues, Heaps

**Assoziative Kollektionen:** Sets, Bags, Multisets, Dictionaries sowieGraphen, Trees

**Beispiel Tabelle erstellen mit 2D Array:**

CREATE TABLE sal\_emp (  
 name TEXT,  
 pay\_by\_quarter INTEGER[], *-- 1 Dim.* schedule TEXT[][] *-- unbound 2 Dim.*);  
**Beispiel Insert:**  
INSERT INTO sal\_emp  
VALUES ('Bill',  
 ARRAY [10000, 10000, 10000, 10000],  
 ARRAY [['meeting', 'lunch'],  
 ['training', 'presentation']]);  
**Beispiel Select:**  
SELECT '{1,2,3}'::INT[]; *-- {1,2,3}*SELECT ARRAY [1,2,3 + 4]; *-- {1,2,7}*SELECT *ARRAY*(  
 SELECT 1 + (*RANDOM*() \* 5)::INT  
 FROM *GENERATE\_SERIES*(1, 6)  
 ORDER BY 1  
 ); *-- gibt z.B. {1,3,5,5,6,6}*

**Index Query:** Intuitiv wie eine Koordinate (Start mit 1 nicht 0)

select *board*[1][1] from tictactoe;

**Slice Query:** Untergrenze:Obergrenze

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

|  |  |
| --- | --- |
| Operatoren | Beispiel |
| Is equal (=) | SELECT ARRAY[1,2,3] = ARRAY[1,2,3]; |
| Contains (@>) | SELECT ARRAY[1,7,4,2,6] @> ARRAY[2,7]; |
| Is contained by (<@) | SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6]; |
| Overlaps (&&) | SELECT ARRAY[1,4,3] && ARRAY[2,1] |

## dictionaries

Schlüssel-Werte Paare, ADT & Datentyp, Bildet “Unique Key” auf Value ab. Umstrittenne Datenstrukture (Zitat: „By using KVP schema you throw DB consistency and optimization down the drain!“).

Möglicherweise geeignet für einfache Datenspeicherung und Datenerfassung, Zeilen mit vielen Attributen, die selten untersucht werden, Semi-strukturierte Daten

**Syntax:** Keys müssen eindeutig sein, doppelte Keys werden ignoriert

**Queries:**

|  |  |
| --- | --- |
| Alle Schlüssel auflisten | SELECT akeys(*mykvpfield*) FROM ...; |
| Alle Key-Value Paare holen | SELECT each(*mykvpfield*) FROM ...; |
| Key holen | SELECT *mykvpfield*->'name' FROM …; |
| Key Value vorhanden | WHERE mykvp @> 'tourism=>zoo'; or  WHERE mykvp @> hstore('tourism’,’zoo’); |

**Beispiel:**

CREATE TABLE test (id integer, col2 hstore, col3 text);  
INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

## graphen/bäume

A picture containing clock, watch

Description automatically generated**Graph:** Ein Graph ist eine Struktur von einem Set von Knoten (Vertices = {V1, V2 ... Vn}) und einem Set von Kanten (Edge = {e1, e1, ... en}). Eine Kante verbindet zwei Knoten repräsentiert als Knoten Paar (z.B e1 = (Vi, Vj)).

**Beispiel:** G = (V, E) wobei

V = {A, B, C, D, E, F}

E = {(A, B), (A, C), (A, D), (C, D), (C, E), (D, E), (E, F)}

**Anwedungen:** Transport, Energie,

Navigationssysteme (kürzester Weg), Geoinformationssysteme (Wasserabfluss)

**Beispiel alle Untergebenen von Mitarbeiter 1010**:

WITH RECURSIVE leafs (idchef) AS (  
 VALUES (1010)  
 UNION ALL  
 SELECT persnr  
 FROM angestellter a,  
 leafs l  
 WHERE a.*chef* = l.idchef  
)  
SELECT persnr,  
 name,  
 chef  
FROM angestellter a,  
 leafs l  
WHERE a.*persnr* = l.idchef;

**Baum:** Ein Baum (Tree) ist ein azyklich verbundener Graph.

**Anwendungen:** Organisation (Führungshierarchie), Stückliste, Bibliothek

Table

Description automatically generated with low confidence

# optimierung und indexe

**Query-Arten:**

|  |  |  |
| --- | --- | --- |
| Bezeichnung | Operator | Rückgabe |
| Equality Query | = | Ein oder mehrere Tupel |
| Range Query | >, >=, <, <=, != | Typischerweise mehrere Tupel |
| Point Query | = | Ein Tupel |
| Join Query | JOINs | Typischerweise mehrere Tupel |

**DB-Benchmarking:** Meistens werden sythetische Daten-Generatoren verwendet um grosse Mengen an Daten zu generieren.

**Kriterien:** Zeit um Index zu erzeugen/aktualisieren, Grösse des Index, Zeit für Query-Abfragen

## indexstrukturen

Indizes sind eine gängige Methode zur Verbesserung der DBMS Performanz und Optimierung. Ein Index ist eine seperate, spezielle Datenstruktur in einem DBMS, welche die Suche/Lesen bzw. eine Query beschleunigt im Vergleich zur sequentiellen Suche in einer Tabelle und auf Kosten zusätzlicher Speicher- und Processing Ressourcen.

**Data Pages:** Daten einer Tabelle physikalisch gespeichert (als Datei), enthält Zeilen in unsortierter Reihenfolge, bei Page-Überlauf (durch Insert/Update) wird ein Page Split gemacht

**Page Sets:** Collection von Pages

**Heap:** Collection von Page Sets / Pages

Diagram

Description automatically generated

**Table Scan:** Scanning aller Pages einer Tabelle, Start bei bei ersten Page der Tabelle und scanne page-to-page und durch alle Rows der Tabelle, Extrahiere die Rows, die die Query Bedingung erfüllen

**Table Scan Performance:** I/O Optimierung durch Lesen der Page-Sets, Langsame Operation bei grösseren Tabellen Index-Optimierung, bei Query mit z.B 80% Result Set Grösse schneller als Zugriff über Index ausser es gibt Index-Only Scan

**Primär-Index:** Index mit PK

**Geclusterter Index:** Clustered sind Daten, die physisch (File-Ebene) sequentiell gespeichert sind sortiert gemäss einem der Daten-Attribute (bzw. Index)

**Integrierter Index:** Index mit Key und Daten (Blätter/Knoten), automatisch geclustered und pro Tabelle nur einmal vorkommend, bei nicht integrierten Index enthalten Blätter nur Referenzen auf die Heap Daten

**Clustering:** Daten einer Tabelle oder eines Indexes sind phyisch sortiert gespeichert gemäss einem Attribut

**B-Bäum:** Geeignet für Hintergrundspeicher/Index-Bedürfnsse, fast optimal für verschiedene Queries und Einfügen

**B+-Bäum:** B-Baum bei dem nur Blätter Daten enthalten, Blätter sind zudem verketter (z.B Linked List), Verkettung erlaubt schnelle Iteration (z.B Range Queries)

**Hash-Index:** Speichert Key-Value Paare basierend auf einer Pseudo-Random-Function, gennant Hash-Funktion, am besten für Point und Multi-Point Queries sofern keine Überlaufketten nötig sind, unbrauchbar für Range-, Präfix- oder Extremwertqueries

Table

Description automatically generated

Muss reorganisiert werden wenn es eine signifikante Menge an Überlaufverkettungen gibt. Um Überlauf zu vermeiden, soll Hash-Speicherplatz nicht voll ausgenutzt werden.

**Bitmap Index:** Ordnet Attributwerte als Bitmuster (Bitmap), Geeignet für Attribute mit wenigen diskreten Werten, d.h Wertebereiche mit geringer Kardinalität

**Mehrdimensionale Index:** z.B R-Tree für 2D-Geometrien

**Mehrstufige Index:** Erster index mit grober Filterung, zweiter Index mit exakter Filterung

**Planner von PG: EXPLAIN**

**EXPLAIN:** gibt geplanten Ausführungsplan aus

**EXPLAIN ANALYZE:** führt Query aus und gibt den Plan mit den effektiven Ausführungskosten aus

Nested Loop(cost=4.65..118.62 rows=10 width=488)…

->Bitmap Heap Scan on tenk1 t1(cost=4.36..39.47 rows=10 width=244)…

Recheck Cond: (unique1 < 10)

->Bitmap Index Scan on tenk1\_unique1(cost=0.00..4.36 rows=10 width=0)…

Index Cond: (unique1 < 10)

-> Index Scan using tenk2\_unique2 on tenk2 t2(cost=0.29..7.91 rows=1 width=244)…

Index Cond: (unique2 = t1.unique2)

Planning time: 0.181 ms

Execution time: 0.501 ms

Seq Scan on bank (cost=0.00..17.80 rows=780 width=76)

(actual time=0.015..0.017 rows=6 loops=1)

•start-up cost •total cost •output rows(tuples) •avg row size (bytes) •how many scan

**Ausführungsreihenfolge:**

Nested Loop(){

Bitmap Heap Scan(){ //2

Bitmap Index Scan(){} //1

}

Index Scan (){} //3

}

**Indexe best Practices:** • Nur Indexe erstellen, die verwendet werden • Indexe greifen erst bei hoher Selektivität (< 10% des erwarteten Resultsets)• Attribute im Index kurz halten, keine langen Strings etc. • Clustered Indexes (= Tabellen) eignen gut sich für Bereichsabfragen und Sortierung • "Covering Index" enthält alle Attribute für Abfrage: schnell, aber abwägen zwischen Geschwindigkeit und Index-Grösse

**Kandidaten für Indexe:** • Primär- und Fremdschlüssel • Attribute, auf denen häufig Range Queries ausgeführt werden • Attribute, die häufig in sortierter Reihenfolge gelesen werden •Attribute, die häufig bei Aggregationen gruppiert werden

## anfragenoptimierung

Ziel ist Minimierung der Ressourcen-Nutzung

**Problem:** Wie Query (WAS?) konkret ausführen (WIE?), Kombination der möglichen Ausführungsplände kann nicht linear ansteigen

**Lösungsansätze:** Absolut beste Lösung suchen durch vollständiges austesten oder optimalste Lösung suchen und den Worst Case verhindern mit sog. heuristischen Schätzmethoden

**Heuristik:** Ein Ansatz zu einer Problemlösung, mit der man nicht die perfekte oder optimale Lösung findet, aber nichtsdestotrotz schnell eine ausreichend gute Lösung findet

Text

Description automatically generated with medium confidence

**Logische Optimierung:** Abfrage ändern, sodass bei einem gleichen Resultat effizienter berechnet werden kann. Zwischenergebnisse klein halten (WHERE-Statement). Umformung des Anfrageterms aufgrund von Heuristiken

**Physische Optimierung:** Effiziente Algorithmen auswählen, um die Operationen der relationen Algebra zu implementieren, Für jeden Ausführungsplan gibt es Kommunikations-, Berechnungs-, I/O-, Speicherkosten.

Erzeugung von einem oder mehreren Ausführungsplänen, in denen die abstrakten Algebra-Operatoren durch konkrete Algorithmen ersetzt werden (nutze hierfür den Systemkatalog mit Informationen über vorliegende Index-Strukturen)

**Kostenbasierte Optimierung:** Generiere alle mögliche Ausführungspläne und bewerte deren Kosten (Kostenmodell, Statistiken, Zusatzinformationen, verfügbarer Speicher)

**Kostenabschätzung Join-Operationen:**

|  |  |
| --- | --- |
| Block Nested Loop Join | M+N • (M / (B-2)) |
| Hash Join | 3 • M+N |
| Merge Join Indexiert | M+N |
| Merge Jon ohne Index | M•Log(M) + N•Log(N) + M + N |
| M,N = Seiten p. Relation, B = Puffer-Seiten | |

## analyse

**Abschätzung der Kosten:**

|  |  |
| --- | --- |
| Kosten | Beschreibung |
| Kommunikationskosten | Anzahl der Nachrichten, Menge der zu übertragenden Daten |
| Berechnungskosten | CPU-Kosten, Pfadlängen |
| I/O-Kosten | Seitenzugriffe |
| Speicherungskosten | Temporäre Speicherbelegung im DB-Puffer und auf Hindergrundspeichern |

**Statistiken:** Informationen über Relationen und Indexe, die an Query beteiligt sind. Typischerweise in System-Katalog gehalten und nur periodisch aktualisiert.

**System-Katalog (Data Dictionary):**

|  |  |
| --- | --- |
| Name | Beschreibung |
| Kardinalität | Anzahl Tupel NTuples(R) für jede Relation |
| Verteilungsinformationen | Informationen über Attribute |
| Grösse | Anzahl Seiten NPages(R) für jede Relation |
| Index-Grösse | Anzahl Seiten NPages(I) für jeden Index |
| Index-Höhe | Anzahl Nicht-Blatt-Stufen für jeden Baum-Index I Height(I) |
| Sonstige Angaben | Funktionale Attribut-Abhängigkeiten |

**Selektivität:** Anzahl unterschiedliche gelieferte Tupel relativ zu Gesamtzahl Tupel in Tabelle. Planer entscheidet anhand von Selektivität der Anfrage ob Index oder Table Scan schneller ist. Hohe Selektivität = Tiefer prozentualer Wert, Tiefe Selektivität = Hoher prozentualer Wert.

|  |  |  |  |
| --- | --- | --- | --- |
| **number** | **value(max)** |  | Werte zwischen 210 und 300 ligen in Bucket 10.  => Selektivität=0.1  Werte schwischen 20 und 60 liegen in Buckets 1-4.  => Selektivität=0.4  Faustrgel:  Selektivität ≤ 0.1 **=>** DBMS verwendet Index  Selektivität > 0.1 **=>** DBMS macht Table Scan  Bei Query [inventories-210-300] wird mit dem Einbezug des Even distribution histogram der Index verwendet. Bei der linearen Interpolation hingegen wird ein Table-Scan durchgeführt. |
| 1 | 27 |  |
| 2 | 42 |  |
| 3 | 57 |  |
| 4 | 74 |  |
| 5 | 98 |  |
| 6 | 123 |  |
| 7 | 149 |  |
| 8 | 175 |  |
| 9 | 202 |  |
| 10 | 353 |  |

A picture containing graphical user interface

Description automatically generated

**Dichte:** Durchschnittlicher prozentueller Anteil von Duplikaten. Falls Daten oder Query eine tiefe Selektivität aufweisen, dann ist Dichte hoch (da Selektivität nur unterschiedliche). Unique Index hat tiefe Dichte

A picture containing table

Description automatically generated**Dichteverteilung:** Anzahl Distinct Values / Anzahl Tupel

Chart, histogram

Description automatically generated

**Optimizer:** Mithilfe der Histogramme Selektivität bestimmen, hohe Selektivität = Index, tiefe Selektivität = Table Scan (bzw. je nach Query Index-Only-Scan)

**Statistik:** Optimizer generiert Statistiken für lineare Interpolation. Qualität der Statistik ist unerlässlich für eine gute Qualität der erzeugten Ausführungspläne.

**Join-Strategien:**

|  |  |
| --- | --- |
| Nested Loop Join | Für alle Zeichen der rechten Tabelle (T) scanne Zeile der linken T ob sie matched. Schnell beim Start, einfach implementiert, gut für kleine T |
| Merge Join | Jede T wird vor dem Start nach Join Attributen sortiert; Dann werden die T paralell gescannt und übereinstimmende Zeilen ausgegeben |
| Hash Join | Zuerst wird die recht T in eine Has T geladen mit Join Attributen als Hash Key; Dann wird die linke T gescannt und jede gefundene Zeile als Hash Key verwendet. Nur für Gleichheitsbedingungen verwendbar. Hohe Startkosten, aber schnelle Ausführung |

# verteilte datenbanksysteme

Verteilte Datenbanksysteme bestehen aus kooperierenden DBMS, die auf verschiedenen Computern (Node, Site) eines Netzwerks laufen. Jeder Knoten hat eine autonome Verarbeitungsfunktionalität und kann lokale Applikationen ausführen. Jeder Knoten partizipiert an globalen Applikationen, die Datenzugriffe auf ide verschiedenen Nodes benötigen.

**Vorteile:** Natürlich verteilte DB-Anwendungen (Firmenniederlassung), Verbesserte Verlässlichkeit (reliability) und Verfügbarkeit (availability), Daten gemeinsam nutzen (sharing) bei lokaler Kontrolle, Performance durch Partitionierung und Parallelisierung

**Nachteile:** Verteilung erhöht die Systemkomplexität

**Anforderungen:**

1. Zugriff auf übers Netz verbundene Systeme
2. Verwaltung der verteilten/replizierten Daten im DBMS-Katalog (Zugriffsverwaltung)
3. Transaktions-Manager mit mehr als einem System (Atomarität von verteilen Transaktionen)
4. Ausführungs-Planer mit mehr als einem System

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

Diagram

Description automatically generated

**Homogen:** Alle Knoten haben identische Software und wissen voneinander und arbeiten zumsammen, um Benutzeranfragen zu verarbeiten. Erscheint gegenüber Benutzer als ein System.

**Heterogen:** Knoten können unterschiedliche SW und Schemas aufweisen. Schemaunterschiede sind Hauptproblem für die Ausführung von verteilter Queries und Softwareunterschiede sind Hauptproblem für die Ausführung verteilter Transaktionen Knoten wissen evtl. nichts voneinander und können nur beschränkte Funktionalitäten für Kooperation anbieten

## verteilte datenhaltung

|  |  |
| --- | --- |
| Fragmentierung | Datenbank in logische Fragmente aufteilen und in verschiedenen Knoten speichern |
| Replikation | Gleiche Daten in mehr als einem Knoten speichern |
| Allokation | Prozess, Fragmente oder Replikas von Fragmenten den Knoten zuzuordnen |

|  |  |
| --- | --- |
| Fragmentierung horizontal: disjunkte Daten; Fragmente A picture containing text, shoji, crossword puzzle  Description automatically generated | **A screenshot of a graph  Description automatically generated with low confidence**Fragmentierung vertikal: Zerlegung verlustfrei mit TID |

**Replikation:** Eine Relation oder Fragment einer Relation ist repliziert, wenn sie redundant in mehreren Knoten gespeichert ist. Höhere Verfügbarkeit, Parallelität, reduzierter Datentransfer und schnellere Query-Verarbeitung auf Kosten von höheren Update-Kosten und komplexere Synchronisation

A picture containing text, clipart

Description automatically generated

**Allokation:** Fragmente den Stationen zuordnen

Diagram, engineering drawing

Description automatically generated

**Transparent verteilte Datenhaltung:** Benutzer sieht eine globale Schicht, kennt weder die Fragmente noch Replika. Queries werden auf der Relation definiert, nicht auf Fragmenten.

**Diagram

Description automatically generated**

## verteilte Transaktionen

Transaktionen auf einer Datenbank ist immer lokal und an einen Session gebunden. Verteilte Transaktionen müssen Koordiniert werden. Klassische Lösung wäre das Two-Phase Commit Protocol (2PC Protocol). Das 2PC-Protokoll sichert die atomare Ausführung von verteilten Transaktionen.

Diagram

Description automatically generated

Diagram

Description automatically generated

**Komponente:**

|  |  |
| --- | --- |
| Applikation | Löst Transaktion aus |
| Transaction Manager (TM) | Node, der Transaktionen über mehrere andere Nodes koordiniert |
| Resource Manager (RM) | z.B DBMS, Message Queue System: Nodes mit lokalen Transaktionen |

|  |  |
| --- | --- |
| Phase | Beschreibung |
| 0 | Update schreiben |
| 1 | Prepare to Commit  TM fragt RM, ob sie die Transaktion mit COMMIT abschliessen kann. Falls ja (READY) gehen RM in Zustand PREPARED |
| 2 | Commit  Hat TM von allen RM READY erhalten, schickt TM COMMIT an alle RM mit Aufforderung, Datenänderung lokal zu schreiben.  Falls RM mit ABORT oder gar nicht nach einer bestimmten Zeit antwortet (Timeout), schickt TM ein ABORT an alle RM und diese machen die Änderungen der Transktion rückgängig (Undo).  Haben die RM ihre lokalen Transaktionen abgeschlossen, schicken sie ein ACK an den TM |

**Zustandsübergänge TM:**Diagram

Description automatically generated

**Zustandsübergänge RM:**

Diagram

Description automatically generated

**Fehlersituation:**  Das 2PC-Protokoll muss mit allen Fehlersituationen umgehen können und das System in einem konsistenten Zustand halten.

**Absturz des TM:**

|  |  |
| --- | --- |
| Vor senden von COMMIT | Rückgängigmachen der Transaktion durch versenden von ABORT |
| Nach RM ein READY mitgeteilt hat | Blockierung von RM |

Absturz des Koordinators TM ist eines der Hauptproblem des 2PC-Protokolls.

**Absturz des RM:**

|  |  |
| --- | --- |
| Kein READY-Eintrag zu Transaktion T | RM führt ABORT durch und teilt dies dem Koordinator mit (FAILED) |
| READY-Eintrag aber kein COMMIT | RM fragt TM, was aus Transaktion geworden ist. TM teilt COMMIT oder ABORT mit, was beim RM zu einem Redo oder Undo der Transkation führt |
| COMMIT-Eintrag vorhanden | RM weiss one nachfragen, dass ein lokales Redo der Transaktion nötig ist |

**Nachricht verloren:**

|  |  |
| --- | --- |
| PREPARE/READY/FAILED-Nachricht des TM an RM geht verloren | Nach Timout geht TM davon aus, dass betreffender RM nicht funktionsfähig ist und sendet ABORT-Nachricht an alle RM |
| RM erhält im READY Zusatnd keine Nachricht vom TM | RM ist blockiert, bis COMMIT/ABORT von TM konnt, da der RM nicht selbst entscheiden kann |

## verteilte anfragen/db-modell

**Zentralisiertes DBMS:** Kosten der Query bestimmt von der Anzahl Disk-I/O

**Verteiltes DBMS:** Kosten der Query bestimmt von Anzahl Netzwerk I/O und Performance Gewinn durch paralleles Abarbeiten auf mehreren Knoten

**Ship Whole:** Query wird local ausgeführt, d.h alle Daten werden übertragen

**Push Down:** Von der ganzen Query oder Teilen davon

**Fetch-as-needed:** Für jedes Tupel der lokalen Relation wird der Wert des Join-Attributs an den anderen Knoten geschickt, der das passende Tupel überträgt

**Art des (Daten-)Verteilens:** Partitionierung → Vertikal, Horizontal Shardhing (Horiz. über mehrere Nodes verteilt) Replikation → Master-Slave, Master-Master Peer-to-Peer

## cap theorem

**Cap Theorem:**

|  |  |
| --- | --- |
| **C**onsistency | Daten sind die Gleichen auf jeder Replikation und jedem Node; verlangt Atomicity, Transaction Isolation. Jeder Lesevorgang erhält den letzten Schreibvorgang oder Fehler. |
| **A**vailability | Daten müssen immer verfügbar/zugreifbar sein. Jede Anfrage erhält eine (fehlerfreie) Antwort, ohne die Garantie, dass sie den letzten Schreibvorgang enthält. |
| **P**artition Tolerance | DBMS funktioniert auch bei teilwesem Ausfall von Netzwerk und Nodes. Das System arbeitet weiter, obwohl eine beliebige Anzahl von Nachrichten durch das Netzwerk zwischen Knoten fallen gelassen (oder verzögert) werden.  Wenn ein Netzwerkpartitionsfehler auftritt, muss entschieden werden, ob dies der Fall ist |

**ACID:**

|  |  |
| --- | --- |
| **A**tomicity | Vollständig oder gar nicht durchgeführt |
| **C**onsistency | Von einem konsistenten Zustand in einen andern,  Vor und nach der Transaktion sind die Daten konsistent |
| **I**solation | So ausführen als ob isoliert  Transaktionen beinflussen sich gengenseitig nicht |
| **D**urability | Auswirkungen auf Dateien bleiben dauerhaft bestehen und nicht durch Fehler verloren |

* Den Vorgang abbrechen und somit die Verfügbarkeit verringern, aber die Konsistenz sicherstellen (RDBMS) **C&P**
* Mit der Operation fortfahren und somit Verfügbarkeit bereitstellen, aber Inkonsistenz riskieren (NoSQL) **A&P**

Diagram

Description automatically generated

Nur zwei der drei Punkte können gleichzeitig erfüllt sein.

**Base Theorem:**

|  |  |
| --- | --- |
| **B**asically **A**vailable | Response zu jedem Request auch wenn fehlerhaft oder inkonsistente Daten |
| **S**oft-State | Es kann immer Änderungen geben, auch ohne Input |
| **E**ventually consistent | System wird eventuell konsistent wenn es keinen Input erhält |

**Cap und Base** sind schlecht definierbar/abgrenzbar. Wichtiger ist: Konsistenz kann in verteilten DBs durch mehr Knoten erhöht werden zum Preis einer erhöten Latenz. **Durability** kann man z.T. auch opfern zugunsten der Performanz

**Wichtige Ziele von CAP:**

1. High Performance Computing HPC z.B mehr Performance durch Parallelisierung
2. Hochverfügbarkeit (High Availability HA) z.B mittels Failover
3. Geringe Verzögerung (Latency)

# Nosql

**RDBMS**: Single Point of Failure durch Shared Everything. Scaling out ist aufwändig. Aufwändiges OR Mapping da keine Objekte

**NoSQL**: Einfaches API (http), Konzipiert fürgrosse Datenvolumen: Skalierbar mit Partitionierung und Replikation auf verteilen Nodes. Oft nicht relational, schema frei. Oft Architektur basierend auf BASE, Verzicht auf ACID. Open Source

**Kategorien**: Key/Value Stores, Document Stores, Graph Stores, Column-Family Stores

**Aggregations-Datenmodelle:** Arrays (Array Stores), Key-Value (Key/Value Stores), Dokumenten (Dokument-Datenbanken)

Table

Description automatically generated

## db-modellierungs-entscheide

Ausgewogenheit des Datenmodells: Applikationsbedürfnisse, Performance Anforderungen, Data Retrieval Patterns. Beziehung zwischen Objekten (Embedding vs. Referencing)

reduziert werden, indem Daten auf mehrere Server verteilt werden.

Bestimmen des Shardkey’s mit Ziel einer guten Verteilung des Dokuments, d.h gute Auslastung der Nodes.

**Guter Shard Key:**

1. Daten gleichmässig auf Nodes verteilt
2. Feld oder Kombinationsn davon, bzw. eine Funktion auf Feld oder Kombinationen
3. Sollte nicht unique sein, also fallen Identifikator/PK hier weg
4. Kann Hash-Funktion sein auf ein Feld oder Kombination davon

Ein Shard Key kann typischerweise nicht mehr verändert werden ohne neu aufgesetzt zu werden (Datenbank-Ausfall)

**Sharding:** Spezialfall von horizontaler Partitioning wobei Partitionen auf verschiedene Nodes verteilt werden. «Sharded» bedeutet automatisch horizontal partitioniert. Durch Sharding kann I/O

## key-value stores

Eine simple Hash Tabelle zugänglich nur mit dem Primary Key. Tabelle hat 2 Kolumnen ID und Value. Der Value kann Text, JSON, XML oder sonst etwas sein.

**Operationen:** Value mit Key holen, Value mit Key schreiben (überschreiben falls vorhanden), Key/Value löschen

**Konsistenz:** Anwendbar nur auf Operationen auf einem einzelnen Key (get, put, delete), Optimistisches schreiben kann durchgeführt werden aber teuer zu implementieren, Eventuell Konsistent, Risiko dass neuer Write den alten Write überschreibt in einem Konflikt

**Geeignet:** Session Informationen, User Profiles, Warenkorb

**Ungeeignet:** Beziehungen zwischen Daten, Multioperation Transaktionen. (Amazon DynamoDB, Riak, Redis, HamsterDB

## document stores / mongodb

Mappt ein Key zu einem strukturierten Dokument, Flexibles Schema, Dokument in JSON oder BSON Formaten gespeichert

**Glossary:**

|  |  |
| --- | --- |
| RDBMS | MongoDB |
| Instanz, DB/Schema | Instanz, Database |
| Table | Collection |
| Row | Document |
| rowid | \_id, objectId |

**Eigenschaften:** Document Data Model (Schemalos), Eigene Query Language, Data Partitioning mittels Sharding, Availability mittels Replikation, JSON als Verkehrssprache

**Data Model:**

|  |  |
| --- | --- |
| MongoDB Installation | Hostet eine nummber von Databases |
| Database | Physische Datencontainer von einem Set von Collections |
| Collection | Enthält ein Set von Documents |
| Document | Enthält ein Set von Key-Value Paaren  Logisch analog zu einem JSON Object |

A picture containing graphical user interface

Description automatically generatedDocuments haben ein dynamisches Schema, es gibt keine vorgesetztes Schema von einer Collection, Documents sind selbstbeschreibend. In einer Collection müssen documents nicht die gleiche Struktur haben. Gemeinsame Felder in verschiedenen Documents können verschiedene Datentypen haben.

**Document Beispiel:**

var example = {

    \_id: ObjectId("509980df3"),

    course: {

        code: "Dbs2",

        title: "Advanced DB"

    },

    year: 2016,

    students: [

        "Peter",

        "Manuel",

    ],

    no\_of\_st: 34

}

Das Document enthält Values von verschiedenen Typen. Primary Key \_id ist von ObjectId type, course ist ein Subdocument und students ist ein Array von Strings.

**Document Structure:** Dokuments sind auf der Harddrive im BSON format gespeichert, eine binäre Repräsentation von einem JSON mit zusätzlichen Typ-Informationen. MongoDB documents bestehen aus Field-Value Paare und haben folgende Struktur:

{

    field1: value1, field2: value2, ..., fieldn:valueN

}

Value von einem Feld kann von jedem JSON/BSOn Typ sein, inklusive anderer Documents, Arrays und Arrays von Dokumenten. Es gibt keine Restriktionen von Field Names.

**\_id:** Das Feld \_id ist reserviert als Primary Key in einem Document. Der Value muss in der Collection unveränderlich und unique sein. Kann von jedem Typ sein ausser Array oder Regex. Immer das erste Feld in einem Document. Gewöhnliche Optionen für Speicherung von Werten für \_id sind ObjectId, ein natürlicher einzigartiger Identifizierer oder ein autoinkrementierende Zahl. Standartmässig ein ObjectId, falls not von User spezifiziert.

**ObjectId:** Ein ObjectId ist ein 12 Byte BSON type, bestehend aus 4 Bytes für die Sekunden seit der Unix epoch (00:00:00 UTC 1 January 1970), 3 Bytes für Maschinenidentifizierer, 2 Byte für Prozess Id und ein 3 Byte counter, welcher von einer zufälligen Zahl startet. ObjectId sind klein, höchstwahrscheinlich unique und schnell generiert.

**MongoDB Embedding:**

Embedded documents capture relationships by storing related data objects within a single document. Denormalized data structures allow reading and manipulating related data in a single DB operation. Embedding is the **preferred technique** in the case of: **1 zu 1 , 1 zu \*** with no extensive overlapping of objects on the many side, Potential disadvantage of embedding is a possibly uncontrolled growth of a document through adding new objects on the many side

**MongoDB Referencing:**

Manual references where you save the \_id field of one document in another document as a reference. Representing relationships by referencing produces normalized data models. Preferred Technique: When embedding leads to data redundancy but does not provide sufficient read performance advantages to outweigh the consequences of data duplication, For representing **many – to many relationships**, To represent **large hierarchical structures**

|  |  |
| --- | --- |
| Vorteile | Nachteile |
| Einfache Query-Language | Keine Constraints |
| Kein Schema | Keine echten Joins |
| Prototyping | Langsame Transaktionen |
| Web-Freundlichkeit | Security: Default offen im Web |
| DbaaS/Cloud |  |

## graph stores / neo4j

Eine Graphdatenbank (GDB) ist eine Datenbank, die Graphstrukturen für semantische Abfragen mit Knoten, Kanten und Eigenschaften zur Darstellung und Speicherung von Daten verwendet.

**Vorteile:** Bei Graph Datenbanken sind Beziehungen sehr wichtig, diese können durch Labeled Property Graph sehr gut modelliert werden. Alle Beziehungen sind durch Kanten modelliert.

**Use Cases:** Connected Data, Routing, Recommendation Engines

**When not to use:** Wenige “Sprünge” in einem Graphen (tabellarische DBMS hier besser), Big Data, Alle oder eine Teilmenge von Entitäten aktualisieren.

**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 Kanten modelliert.

**Diagram

Description automatically generatedNeo4J:**

**Text

Description automatically generated**

**Diagram

Description automatically generated with medium confidence**

Löschen aller Nodes und Relationen (implizit)

|  |  |
| --- | --- |
| Bezeichnung | Beschreibung |
| Labeled Property  Graph-Modell | Knoten und Kanten – wobei im Unterschied zu einfachen Graphen beide aus Objekte mit darin eingebetteten Properties bestehen |
| Node/Knoten | Hat ein Set von Labels und Properties |
| Edge/Kante | Hat Namen, Set von Properties und ist gerichtet |
| Property | Key:Value wobei Value ein (Array von) primitiver Typ ist |
| Typsystem | Knote, Kante, Pfad, Map, Liste, Ganzzahl, Float, Boolean, String |

**Diagram, schematic

Description automatically generated**

Join’s sind hier implizit, da Abfragen auf (Beziehungs-/Pfad-) Muster. Entsprechen zum SQL

## column family stores

Eine Column Family Store besitzt eine Datenstruktur mit einem Row Key und Gruppen von Key-Value Paaren. Eine Gruppe bildet die Column Family. Nicht dasselbe wie Column Store Databases.

Diagram

Description automatically generated

**Anwendungen:** Cassandra, Hbase

**Cassandra:** Hohe Verfügbarkeits Column Family Store, Viel verteilte DBMS Properties, Keine starken Konsistenzen (Eventual consistency), Keine Transaktionen im traditionellen Sinn, Minimale Administration

|  |  |
| --- | --- |
| Begriff | Bedeutung |
| Keyspace | aka DB in einem Node |
| Node | Ort der gespeicherten Daten |
| Data Center | Sammlung von Nodes; Viele Nodes sind kategorisiert als DC |
| Cluster | Sammlung von Data Center |

Shape, arrow

Description automatically generated

**Cassandra Data Model:**

**A picture containing diagram

Description automatically generated**

**Cassandra Column:** Basic unit of storage in Cassandra, Key-Value Paar gespeichert mit timestamp, um Schreibkonflikte, veraltete Daten, etc. zu lösen

**Cassandra Column Family:** Eine Row ist eine Sammlng von Kolumnen gebunden an einem Schlüssel. Eine Kolumnenfamilie ist eine Sammlung von gruppierten Rows

Text, letter

Description automatically generated

**Cassandra Query Language (CQL):**

CREATE COLUMNFAMILY Customer (

    KEY varchar PRIMARY KEY,

    name varchar,

    city varchar,

    web varchar

);

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

    ('mfowler', 'Martin Fowler', 'Boston','www.martinfowler.com');

SELECT \* FROM Customer;

SELECT name, web FROM Customer;

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

**Geeignet:** Event Logging, Content Management Systems, Counters in Web Applications, Wenn Queries nicht alle Felder abgefragt werden, Für attributierte Zugriffe (SUM, etc.)

**Ungeeignet:** Systeme welche ACID Transaktionen für Schreiben/Lesen brauchen,

Write-intensive Datensets/Applikationen, OLTP

## olap / oltp, row vs column store

**OLTP (Online Transaction Processing):** Datenverwaltungs-optimiertes System

**OLAP (Online analytical Processing):** Lese-optimiertes System

|  |  |
| --- | --- |
| OLTP | OLAP |
| Low volume of data | High volume of data |
| High volume of transactions | Low volume of transactions |
| Typically normalized data | Denormalized data |
| ACID compliance | Not necesaarily ACID-complient |
| Require high availability | Don’t usually require high availability |

**Encoding:** Komprimierung der Spalten reduziert die Anzahl der Speicerzugriffe, führt zu weniger vergeblichen Cache-Zugriffen durch mehr Info. über eine Cache-Zeile, Operationen z.T direkt auf komprimierten Daten möglich (Vectorized Exec. Schema)

**Run Length Encoding (RLE):** Zählen wie oft gleicher Wert vorkommt und Start und Ende in neuer Tabelle speichern. Kompressionsrate 5-15x

**Application

Description automatically generated with medium confidence**

**Dictionary Encoding:** Eine Spalte in in ein Dictionary und Attributvektor unterteilt. Dictionary speichert alle eindeutigen Werte mit impliziter WerteID. Attributvektor speichert Wertidentifikation für alle Einträge in der Spalte. Position ist implizit gespeichert (offsetting)

A picture containing chart

Description automatically generated

**Row Store vs Column Store:**

|  |  |
| --- | --- |
| Row Store | Column Store |
| Seq. Row Scan gute Bandbreiten-Ausnützung zwischen CPU und RAM | Gleicher Datentyp, d.h bessere Kompression und Caching |
| Einfaches horizontales Partitioning | Kleinerer Speicherplatz |
| Einfügen | Meist automatisch Index auf jede Spalte |
| Problem: Hauptspeicherzugriff ist Engpass: Tupel Konstruktion ist auswändig (Lesen aller Spalten) | Problem: Schreiben, «Select \*» vermeiden |

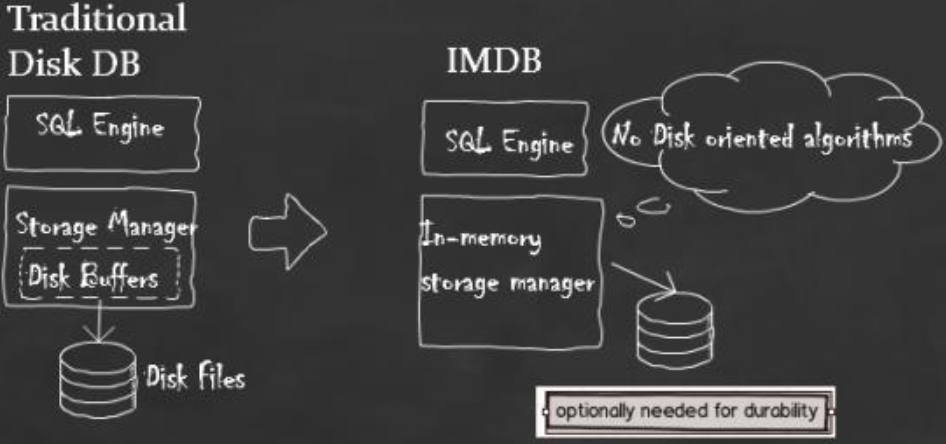
## in-memory stores

Ein In-Memory Datenbanksystem (IMDB) ist ein Datenbankmanagementsystem, das Daten vollständig im Hauptspeicher speichert. Dies steht im Gegensatz zu herkönnlichen On-Disk Datenbanksystemen, die für die Datenspeicherung auf persistenten Medien konzipiert sind. IMDB können eine Grössenordnungen (10-1000x) schneller sein, v.a wenn sie In-Memory optimiert, mit Column-Architekturen kombiniert sind oder aber spezialisiert sind. ACID-Prinzip wird hier eingehalten.

**Datenhaltung:** Beim Start der DB lädt System den gesamten Datenbestand der Festplatte in den Speicher, damit bei laufendem Betrieb keine Daten nachgeladen werden muss

**Checkoint Files:** Geänderte Daten werden in regemässigen 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.



**Geeignet:** OLAP, Oft mit Column Stores kombiniert

**Ungeeignet:** Häufiges Einfügen (Einfügen in komprimierte Spalten teurer als bei zeilenorientierten Struturen)

**Beispiele von IMDB:** SAP HANA, Oracle TimesTen, IBM SolidDB, MonetDB

## column oriented in-memory store

**MonetDB:** MonetDB ist eine In-Memory Kolumnen-orientierte DBMS entwickelt für hohe Performance bei komplexen Abfragen mit grossen Datenmengen.

**MonetDB Eigenschaften:** MonetDB ist keine In-Memory Datenbank im eigentlichen Sinne. MonetDB verwendet "Memory-Mapped File Arrays" bzw. Memory-mapped Dateien. Wenn MonetDB eine memory-mapped Datei verwendet, kann sie die Daten direkt im Speicher auf der Festplatte abbilden (Array). Bei einer SQL-Query, wird sie auf eine solche Datei abgebildet und dann vom Kernel des Betriebssystems in den Speicher geladen. Wenn die Datensätze nicht mehr verwendet werden, wird ihr Speicherplatzfreigeben (virtuelle Speicherverwaltung).

**Technisches:** Jede Spalte ist auf eine Datei gemapped. Spalten sind komprimiert (in-memory optimised), Attribute von Spalten sind automatisch indexiert. Multi-Core Parallel Execution. Reines OLAP-System

**Vergleich mit RDBMS:** Hier speichern In-Memory Stores die Daten in regelmässigen Abständen dauerhaft, während RDBMS in der Regel vor jeder Transaktion als abgeschlossen markiert. Dies bedeutet, dass RDBMS potentiell langsamer sind, weil sie häufiger committen, während In-Memory Stores normalerweise ein Zeitfenster haben, in dem Datenverlust auftreten kann

## monetdb vs postgres

• MonetDB ist schnell bzw. schneller als z.B. PostgreSQL, wenn es um

Aggregationsfunktionen geht (GROUP BY, SUM/AVG) und wenn nur wenige Ausgabe

Felder (Columns, Attributes) benötigt werden.

• Beide Server sind typischerweise bei der ersten Query-Durchführung (= Kaltstart)

langsamer und dann bei der wiederholten Query-Durchführung schneller.

• Diese Unterschiede von Kalt- zu Warmstart sind bei MonetDB geringer; dies wenn die

Daten im Memory Platz haben und besonders nach dem allerersten Mal, wenn sie

geladen sind.

• Klarer Vorteil von MonetDB trotz aufwändiger Zusammenstellung aller Spalten.

PostgreSQL braucht trotz vorhandener Indexe länger, um die Teilresultate mittels

Nested Loop Join und Hash Join zusammenzusetzen

# database as a service (dbaas)

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.

**Definition:** Cloud-DB-Dienste sind weder reine Infrastrukture noch reine Plattform. Cloud-DB-Dienste sind nicht Infrastructure as a Service (IaaS), weil DB sich einer höheren Abstraktionsebene befinden als virtuelle Maschinen. Sind wie Platform as a Service (PaaS) stark mit der umgebenden Plattform verwoben, bieten im Gegensatz aber generische Schnittstellen an.

Graphical user interface, text

Description automatically generated

## graphql

GraphQL ist eine deklarative Datenabfrage- und Manipulations-Sprache für API’s und Laufzeitumgebung, ist Schema-basiert mit Typen, stellt eine Beschreibung der Daten ebenfalls als API zur Verfügung, gibt den Clients die Möglichkeit, nach dem zu fragen was gebraucht wird, wickelt alle Anfragen über einen einzigen «Endpoint» ab.

**Schema-First vs. Code-First:** Eine Schema-Definition erhöht Lesbarkeit, verbessert Kommunikation zwischen Front-/Backend, ist gleichzeitig Dokumentation der Daten und API und ermöglicht rasche SW-Entwicklung (Mocking von API)

**PostGraphile:** Bibliothek für «pluggable» GraphQL-API und implementiert GraphQL als Webservice / API mit PostgreSQL als Datenspeicher und NodeJS alas HTTP-Server

**Prinzip:** Schema-First, bzw. DB-First, d.h aus vorgegebenem DB-Schema werden Queries und Mutations (inkl. Resolver) generiert. Resolver sind der Code (SQL), der nötig ist, um auf die konkreten Daten zuzugreifen.

**Schema:** Gegebene Tabelle Angestellter macht PostGraphile folgendes

* Erzeugt einen Type Angestellter, benennt Felder in camelCase ohne «\_», ergänzt ein Feld nodeId als Globally Unique Identifier falls PK existiert und ergänzt je ein Feld für jede Fremdschlüssel-Beziehung, z.B abteilungByAbtnr
* Erzeugt als Root «Query Types» eine Connection allAngestellters inkl. Pagination, condition/filter und order, einige Felder für jeden Unique Constraint der Tabelle, z.B angestellterByPersNr, ein foo(nodeId: ID!)-Feld («!» = obligatorisches Attribut)
* Erzeugt als Root «Mutation Types» (C(R)UD), zb. createAngestellterm, updateAngestellter und deleteAngestellter

**Client-Sicht:** In paginierten Resultsets müssen verschachtelte Strukturen benannt werden (Graphen-Nomenklatur). «Connection» steht in GraphQL für Listen von Edges. Eine Node repräsentiert das eigentliche Objekt. Wenn nur panigierte Listen Nodes interessieren, verwende man «nodes» mit «s», ohne «edges». Wenn Cursor/Seek-Based Pagination gar nicht interessiert, kann man Edges/Node/Nodes weglassen. Eine Edge hat Metadaten über ein Objekt in einer Liste, ein Node enthält einen «cursor», um eine Paginierung ausgehend von diesem Objekt zu ermöglichen. Der Client merkt sich den zuletzt erhaltenen Cursor und verwendet diesen dann in der folgenden Query. «pageInfo» enthält Info, ob noch mehr Datenseiten zu holen sind.

**Pagination:** Durch Ergebnisse einer grossen Datenabfrage blättern. Anwendungsfälle aus Usability-Perspektive – Nummerierte Seiten. Bsp. Google Suchmaschinen-Resultatseite (SERP) Typische Lösung: Vorherige Seiten überspringen mit OFFSET Probleme mit OFFSET: – OFFSET wird immer langsamer, je grösser er ist; – die Seiten können sich durch das Einfügen neuer Daten verschieben, da die Sortierung bei jeder Abfrage neu durchgeführt wird

**SQL vs REST vs GraphQL:**

SQL hat eine generelle Abfragesprache, beliebige Applikationen mit normalisierten Daten, z.B Nutzerverwaltung von Firmen

|  |  |
| --- | --- |
| Vorteile | Nachteile |
| Normiert und standardisiert  Weit verbreitete Abfragesprache | Keine direkte Abfrage über HTTP(s)  Für schnelle Prototypen nicht geeignet |

RESTful: Applikationen mit wenigen Klassen, welche oft mit allen Inhalten ausgeliefert werden, z.B Twitter. REST ist kein Standard, eher statisch, hat keine expliziten Join und ist dadurch aber auch viel schneller

|  |  |
| --- | --- |
| Vorteil | Nachteil |
| Weit verbreitet  API strukturell gleich ein fach zu brauchen | Viele Roundtrips notwendig um Daten abzufragen(N+1 Problem)  Resultat enthält z.T unnötigerweise alle Objekte und diese alle Felder |

GraphQL: Webapplikation mit Joins/diversen verknüpften Daten, von welchen man meist nicht alle Felder braucht, z.B. Frontend für die Nutzerverwaltung einer Firma. GraphQL ist Web-orientiert, dynamisch (Standard lässt absichtlich vieles Offen) und einfach zu verwenden. Graphen als Denkmodell für Daten.

|  |  |
| --- | --- |
| Vorteil | Nachteil |
| Sämtliche benötigten Daten werden in einem Roundtrip geliefert.  Die vom Client definierte Datenstruktur ist deklarativ und typisiert.  Client ist vom Server entkoppelt: Der Client definiert die verlangten Daten und nicht der Server | Abfragesprache ist nur teilweise normiert, Grundelemente meist gleich  Erweiterungen wie bspw. Filters sind Backend-spezifisch |

# json

Wieso sollte eine Datenbank JSON Datenstrukturen speichern?

1. Database schema flexibility; unpredictable, fast changing data structures
2. Lots of attributes of a table that are rarely used
3. Making the database “webfriendly”
4. Matching structure for hierarchic, nested original data (“polyglot persistence”)
5. Migrating from NoSQL (now with real transactions!) or migrating from Entity-Attribute-Value model

Wann keine JSON Datenstrukturen brauchen?

1. When you need constraints, like from a database or format schema.
2. When you could use a table i.e nomalized attributes

**Von relationaler Tabelle nach JSON:**

SELECT *JSONB\_BUILD\_OBJECT*('persnr', ang.*persnr*,  
 'name', *MIN*(ang.*name*),  
 'projects',  
 *JSONB\_AGG*(*TRIM*(proj.*bezeichnung*))  
 ) AS objects  
FROM angestellter ang  
 LEFT OUTER JOIN projektzuteilung pz ON ang.*persnr* = pz.*persnr* LEFT OUTER JOIN projekt p ON pz.*projnr* = p.*projnr*GROUP BY ang.*persnr*ORDER BY ang.*persnr*;

**JSON-Daten abfragen:**

SELECT *persnr*, *angwithproj*FROM angprojj  
WHERE *angwithproj* ->> 'persnr' = 1001::TEXT;  
  
SELECT *persnr*, *angwithproj*FROM angprojj  
WHERE (*angwithproj* ->> 'name') LIKE 'Marxer%';  
  
SELECT *persnr*, *JSONB\_PRETTY*(*angwithproj*)  
FROM angprojj  
WHERE *angwithproj* -> 'projects' @> *TO\_JSONB*('Uranus'::TEXT)

SELECT *persnr*, *angwithproj* -> 'projects' AS projects  
FROM angprojj;  
  
SELECT persnr, angwithproj ->> 'name' AS persname, value AS projname  
FROM angprojj, *JSONB\_ARRAY\_ELEMENTS\_TEXT*(*angwithproj* -> 'projects');  
  
SELECT persnr, angwithproj ->> 'name' AS persname, value AS projname  
FROM angprojj  
CROSS JOIN LATERAL *JSONB\_ARRAY\_ELEMENTS\_TEXT*(*angwithproj* -> 'projects');

# examples

## mongodb

|  |  |
| --- | --- |
| **Lookup mit Aggregation (= aka Joins!) ab Version 3.2:**  [{     $lookup: {          from: 'person',          localField: '\_id',          foreignField: 'address',          as: 'people'      }  }, {      $match: {          "street": "Blumenstrasse 13"      }  }, {      $project: {          people: {              name: 1          }     } }] | **Entspricht Left Join mittels Subquery:**  SELECT *\**, *holidays* FROM absences WHERE *holidays* IN (SELECT *name*, *date* FROM holidays WHERE *year* = 2018); |

**MongoDB CRUD Queries:**

db.collection.insert({ name: 'Aurora', gender: 'f', weight: 450 })

WriteResult({ "nInserted": 1 })

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

db.collection.find({ gender: 'f', $and: [{ weight: { $gte: 600 } }, { weight: { $lte: 900 } }] })

db.collection.remove({… });

**MongoDB Join Queries:**

db.unicorns.find({gender:"f",loves:{ $all:["apple","carrot"]}})

db.unicorns.find({gender:"f",loves:{ $in:["apple","carrot"]}})

db.unicorns.find({$or:[{vampires: {$exists:false}},{vampires: {$lte:0}}]})

db.unicorns.find({gender:"m",$and:[{weight:{$gte:600}},{weight:{$lte:900}}]})

db.unicorns.find({gender:"f",$or:[{loves:"apple"}, {loves:"carrot"}]},{\_id:0})

db.unicorns.update({name: "Roooooodles"}, {$set: {weight: 590}}, {upsert:false})

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

|  |  |
| --- | --- |
| **// GraphLookup**  db.employees.aggregate([    { $graphLookup: {      from: "employees",      startWith: "$manager",      connectFromField: "manager",      connectToField: "name",      as: "managerDocs",      maxDepth: 0    } },    { $match: { "name":"Moneo"} },    { $project: { \_id:0, name:1, manager:1, managerDocs:1 } } ]) | **// Lookup**  {   $lookup: {      from: <collection to join>      localField: <field from the input document>      forgeinField: <field from join documents>      as: <output array> }}  db.employees.aggregate({      $lookup: {          from: "employees",          localField: "manager",          foreignField: "name",          as: "managers"      }  }, {      $match: {          "name": "Moneo"      }  }, {      $project: {          \_id: 0,          manager: 0,          managers: {              \_id: 0,              manager: 0}} }) |

## neo4j

**Neo4J Create:**

CREATE

(alice:User {username: ‘Alice’}),

(nodeName:Type {Properties} )

(bob:User {username: ‘Bob’}),

(alice)-[:ALIAS\_OF]->(bob)

**Neo4J Match:**

MATCH (tom {name: "Tom Hanks"}) RETURN tom

MATCH (people:Person) RETURN people.name ORDER BY people.name ASC

MATCH (tom:Person) WHERE tom.name = "Tom Hanks" RETURN tom

MATCH (movies:Movie) WHERE movies.released >= 1990 AND movies.released <= 1999

RETURN movies.title

**Neo4J Match mit Verbindung:**

MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED\_IN]->(tomHanksMovies) RETURN tom,tomHanksMovies

MATCH (bacon:Person {name:"Kevin Bacon"})-[\*1..4]-(hollywood) RETURN DISTINCT Hollywood

So werden alle Knoten gefunden, die über bis zu 4 Beziehungs-Hüpfer (Pfade durch den Graph mit Länge 1 bis 4) etwas mit Kevin Bacon zu tun haben

MATCH p=shortestPath( (bacon:Person {name:"Kevin Bacon"})-[\*]-(meg:Person {name:"Meg Ryan"}) ) RETURN p

Ermittelt die kürzeste Verbindung zwischen Kevin Bacon und Meg Ryan.

**Neo4J Daten löschen:**

MATCH (a:Person),(m:Movie) OPTIONAL MATCH (a)-[r1]-(), (m)-[r2]-() DELETE a,r1,m,r2

Löschen über komplette Abbildung von Beziehungen

MATCH (n) DETACH DELETE n

**Left Outer Join:**

MATCH (m1:Movie), (m2:Movie)

WHERE m1.released = m2.released

AND NOT m1 = m2

RETURN m1.title, m1.released, m2.title

## graphql

Alle Angestelle mit 5000.- Lohn

Graphical user interface

Description automatically generated with medium confidence

Angestellter erstellen und Daten rückgeben

Text

Description automatically generated

Angestellter nach PersNr löschen

Graphical user interface, application

Description automatically generated

Alle Angestellter mit Lohn zwischen 5000.- und 8000.- und wohnhaft in Luzern

Text

Description automatically generated with medium confidence

## datenstrukturen graph trees

**Nested Set:**

A picture containing funnel chart

Description automatically generated

**Adjacency List:**

Diagram

Description automatically generated

INSERT INTO clots(id,name,level,left,right)VALUES(1,Cloting,1,22);

**Methode zur Ausgabe:**

WITH p AS (  
 SELECT *lft*, *rght* FROM clothing  
 WHERE *name* = 'Dresses'  
)  
SELECT a.*id*, a.*name*FROM clothing a,  
 p  
WHERE a.*lft* BETWEEN p.lft AND p.rght  
ORDER BY a.*lft*

**Ausgabe:** Dresses, Evening Gowns, Sun Dresses

**Adjazenzliste:**

INSERT INTO clots(*id*,*name*,*parent\_fk*)VALUES(1,*Cloting*,null);  
INSERT INTO clots(*id*,*name*,*parent\_fk*)VALUES(2,*Men*,1);  
INSERT INTO clots(*id*,*name*,*parent\_fk*)VALUES(3,*Suits*,2);  
INSERT INTO clots(*id*,*name*,*parent\_fk*)VALUES(4,*Slack*,3);  
INSERT INTO clots(*id*,*name*,*parent\_fk*)VALUES(5,*Jackets*,3);

**Methode zur Ausgabe:**

WITH RECURSIVE a AS (  
 SELECT *id*, *name*, *parent\_fk* FROM clothing  
 WHERE *name* = 'Sun Dresses'  
 UNION ALL  
 SELECT p.*id*, p.*name*, p.*parent\_fk* FROM a  
 JOIN clothing p ON a.parent\_fk = p.*id*)  
SELECT *\**FROM a  
ORDER BY 1

**Ausgabe:** Clothing, Womens, Dresses, Sun Dresses

**Methode zur Ausgabe mit Level und parent\_fk:**

WITH RECURSIVE a (id, parent\_fk, name, level)  
 AS (SELECT *id*, *parent\_fk*, *name*, 1  
 FROM cloths  
 WHERE *name* = 'Women'  
 UNION ALL  
 SELECT p.*id*, p.*parent\_fk*, p.*name*, a.level + 1  
 FROM a  
 JOIN cloths p ON p.*parent\_fk* = a.**id**)  
SELECT *\**FROM a  
ORDER BY level;

**Ausgabe:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **parent\_fk** | **name** | **level** |
| 6 | 1 | Women | 1 |
| 7 | 6 | Dresses | 2 |
| 10 | 6 | Skirts | 2 |
| 11 | 6 | Blouses | 2 |
| 8 | 7 | Evening Gowns | 3 |
| 9 | 7 | Sun Dresses | 3 |

## common table expression (cte)

Mit CTE kann zeilenweise ausgegeben werden. CTE erlaubt Abfragen quasi als Parameter und können rekursiv sein. Existieren nur während dem Statement.

Sämtliche Untergebene von Steiner (1010) finden:

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

## i/o kosten suche

Gegeben ist eine Relation R(P, A, B). R hat 2‘000‘000 Tupels. 80 Tupels belegen einen Block (Total 25‘000 Blöcke).

Wie viele IO’s durchschnittlich benötigt die Suche nach Tupels (und dem Lesen der gefundenen Tupels) mit einem Wert von A=a0. Annahme: es existieren ca. 24’000 Tupel mit A=a0.

**a)** Wenn kein Index auf R definiert ist Lineare Suche auf R (100% der Blöcke durchsuchen): 25‘000 IO’s sind nötig

Bei b) und c) können Sie die davon ausgehen, dass das Wurzelelement des Indexes jeweils bereits im Speicher gecacht ist. FanOut=200.

**b)** Wenn ein Clustered Index auf dem Attribut A definiert ist?

Annahme Fill-Faktor=1.0 Index Block 1 enthält 200 Einträge; 40’000 im Level 2. dh 1 Block-IO’s für das Laden des korrekten Index-Blocks und 1 IO fürs erste Tupel. Je nach der Verteilung sind für die 24’000 Tupels sind 299-300 Operationen nötig.

**c)** Wenn ein (Nonclustered) Index auf dem Attribut A definiert ist. Die Tupels mit A=a0 sind beliebig auf die Blöcke auf dem Heap verteilt. FanOut=200?

Index Block enthält 200 Einträge; 40’000 im Level 1, 1’600'000’000 im Level 2, d.h 2 Block-IO’s für das Laden des korrekten Index-Blocks, + 299-300 IO’s für Laden der benachbarten Index-Blöcke + 24'000 IO’s für Laden der Daten

## ausführungsplan

Gegeben ist das Schema: Orders (OrderId, OrderDate, …) mit ca. 13‘000 Tupels. OrderDate zwischen 2004 und 2010

OrderLines(OrderId, OderLineId, Quantity, ...) 65‘000Tupels

Unique Index auf Order.OrderId und (OrderLines.OrderId , OrderLines.OrderLineId)

F: Selektivität hoch bei where mit Unique Index?

A: Nested Loop Join

F: Selektivtät hoch bei where ohne Index?

A: Hash Join Table Scan über Orders und Table Scan OrderLines

F: Selektivität tief mit Indexen oben?

A: Hash Join mit Full Table Access auf Orders und Index-Zugriff auf OrderLines