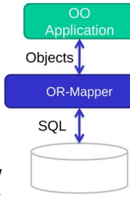
Data Engineering

1 O/R-Mapping....1
2 Stored Procedures & Cursors....1
3 Triggers....2
4 Updatable & Materialized Views....2
5 Datenstrukturen....2
6 Query-Optimierung & Indexe....2
7 Verteilte & replizierte Systeme....3
8 NoSQL....3
9 NoSQL: Key/Value Stores....3
10 NoSQL Document Store Mongo....3
11 NoSQL: Graph Stores....3
12 NoSQL: Column (Family) Store....3
13 DBaaS & GraphQL....4
14 Design Patterns....4

1 O/R-Mapping

1.1 Grundprinzip



Programm: objektorientiert, Klassen, Objekte & Java Datentypen
DB: relational, Tabellen, Tuples, Beziehungen & PostgreSQL Datentypen
Brücke dazwischen ist O/R-Mapping, Mapping zwischen Objekten / Tabellen, Objekte persistieren / abfragen, CRUD-Operationen, Transaktionsmanagement, Abbildung von Datentypn

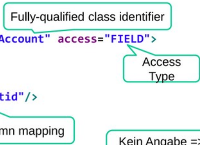
1.2 Produkte in Java

- ***Java Data Objects***: JDO, OO-Kompatibilität besser als JPA, kein Lazy-Loading, beliebige Datenquellen
- ***jOOQ***: lightweight, SQL-based, type-safe Queries
- ***Jakarta Persistence API (JPA)***: *EclipseLink*, OpenJPA, DataNucleus, JBoss, Hibernate (JPA/Enterprise Java Beans 3.0 konform, eigenes API & Query-Lang)

1.3 Jakarta Persistence API (JPA)

JPA Layers (unten→oben): DB, JDBC Driver (3rd Party), JDBC API (Standard Interface), JPA Provider (3rd Party), JPA (Core) (Standard ORM Interface (JEE)), Java App
META-INF/persistence.xml: obligatorisches Anbindungs-Konfigfile, definiert Persistence Unit
META-INF/orm.xml: optionales O/R-Mapping File, für Plain old Java objects (POJO), Prios: 1. orm.xml, 2. Annotations, 3. Default Mapping

```xml
<entity name="bankaccount"
        class="ch.hsr.dbs1.BankAccount" access="FIELD">
  <table name="account"/>
    <attributes>
     <id name="id">
        <column name="accountid"/>
     </id>
  </attributes>
</entity>
```



Entities: DB-Abbildung mit Annotations, kann (ver)erben, Interfaces implementieren & abstract sein, public/protected Konstruktor ohne Argumente, @Id Pflicht, Klasse / Fields / Methode nicht final, Fields private oder protected

```java
@Entity
@Table(name = "account") // optional
public class BankAccount {
  @Id
  @Column(name = "accountid") // optional
  @Gen...Value(strategy=Gen..Type.IDENTITY)
  private long id;
  private String name;
  @Temporal(TemporalType.TIMESTAMP)
  private Calendar creationDate;
  @Enumerated(EnumType.STRING)
  private Currency curr;
  @Transient // nicht persistent
  private String tempComments; /* getter und setter */ }
```

Entity States: *New* (keine persistente Identität & nicht mit Persistenzkontext verbunden), *Managed* (persistente Identität & mit Persistenzkontext verbunden), *Detached* (persistente Identität & nicht mit Persistenzkontext verbunden), *Removed* (wie managed aber für die Entfernung aus dem Datenspeicher vorgesehen)

```java
EntityManagerFactory f = Persistence.
  createEntityManagerFactory("Bank"); //legt Schema an
EntityManager em = f.createEntityManager();
Query q = em.createQuery
("SELECT a FROM BankAccount a");
List<BankAccount> list = q.getResultList();
em.getTransaction().begin();
BankAccount a1 = new BankAccount();
a1.setName("test"); em.persist(name);
BankAc...t a2=em.find(BankAccount.class,1L);
a2.incBalance(100);
BankAc...t a3=em.find(BankAccount.class,2L);
em.remove(a3); em.getTransaction().commit();
em.close();
```

Mapping Regeln: case insensitiver, gleicher Name für Tabellen/Columns, alle Attribute persistent (ausser @Transient), Calendar/Date nur mit @Temporal
Unterstützte Typen: Primitive Typen/Wrappers, Strings, Enums, Byte/Char Arrays, Date/Calendar, beliebige Serializable Klassen, ***Relationen***: Referenzen auf Instanzen mit Entity Klassen, Collection<>, Set<>, List<>, Map<> von Entities, ***@Column Params***: name="", unique=true, nullable= true, length=200, BigDecimal: scale=10 / precision=2, Calendar: columnDefinition="TIMESTAMPTZ"
Field Access: Annotations bei Variablen, getter/setter nicht Pflicht ***Property Access***: Annotations bei getters, somit getters/setters Pflicht

1.4 Beziehungen

BankCustomer 1..1 → 0..1 Address — 1:1

```java
public class BankCustomer {
  @OneToOne(optional=true)
  @JoinColumn(name="addressid")
  private Address address; ...
```

BankCustomer 1..1 ↔ 0..1 Address — 1:1 Bidirectional

```java
public class BankCustomer {
  @OneToOne
  @JoinColumn(name="addressid")
  private Address address; ...
public class Address {
  @OneToOne(mappedBy="address", optional=f)
  private BankCustomer customer; ...
```

BankCustomer 0..1 ← 1..1 Address — 1:1 Invers

```java
public class Address {
  @JoinColumn(name="id",ref...ColumnName=
="addressid", insertable=false, updatable=false)
  private BankCustomer customer; ...
```

BankAccount 0..* → 1..1 BankCustomer — Many:1

```java
public class BankAccount {
  @ManyToOne(optional=false)
  @JoinColumn(name="customerid")
  private BankCustomer customer; ...
```

BankAccount 0..* ↔ 1..1 BankCustomer — Many:1 Bidirektional

```java
public class BankCustomer {
  @OneToMany(mappedBy="customer")
  private Collection<BankAccount> accounts = ...;
```

BankCustomer 1..1 → 0..* BankAccount — 1:Many

```java
public class BankCustomer {
  @OneToMany
  @JoinColumn(name="customerid", referenced
   ColumnName = "id")
  private Collection<BankAccount> accounts = ...;
```

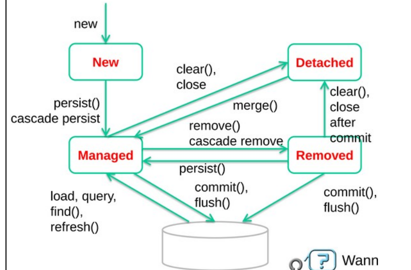
BankManager 0..* → 0..* BankCustomer — Many:Many

```java
public class BankManager {
  @ManyToMany
  @JoinTable(name="customer_manager", join
   Columns={@JoinColumn(name="manager")},
   inverseJoinColumns={@JoinColumn(name=
   "customer")})
  private Collection<BankCustomer> custs = ...;
```

BankManager 0..* ↔ 0..* BankCustomer — Many:Many Bidirectional

```java
class BankCustomer {
  @ManyToMany(mappedBy="custs")
  private Collection<BankManager> managers=...;
```

Eager Loading: Target Entity direkt mit Beziehung laden, Default bei @OneToOne & @ManyToOne
Lazy Loading:Target Entity bei 1. Beziehungs-Zugriff laden, Default bei @OneToMany & @ManyToMany
@XtoX(fetch = FetchType.LAZY/EAGER)
N+1 Performance Problem: bei lazy, z.B. ruft man eine Entität BankCustomer (*1*) auf und pro Entität im Resultatset (*N*) dann die in Beziehung stehende Entität BankAccount, es gibt also N+1 SQL-Queries, **Lösungen**: "join fetch" statt "join" im JPQL, eager verw., Entity Graphs
Top Down (Forward Engineering): Business-Modell, dann DB-Schema, ***Bottom Up***: DB-Schema, dann Business-Modell, ***Middle Out***: Metadaten erstellen, dann Java & DB-Schema, ***Meet in the middle***: Business-Modell & DB-Schema existieren, Metadaten erstellen, ***Meta Model / Model driven***: unwichtig



Object Identity: Session(=EntityManager =Persistence Context) übersetzt PK zu Objekt Instanz, im Cache durch ID identifiziert
Generierungs-Strategien: **AUTO**, **IDENTITY** (auto-increment, SQL: "id **GENERATED** ALWAYS **AS IDENTITY NOT NULL**"), **SEQUENCE** (SQL: "**CREATE** SEQUENCE seqname;"), **TABLE** (PK DB-Tabelle mit key-name & letztem keyvalue)
Objekte werden nicht automatisch persistiert, entweder explizit über em.persist() / em.remove() oder implizit über @XtoX(cascade = CascadeType.PERSIST) / @XtoX(cascade = CascadeType.REMOVE)

Aggregation:

BankCustomer ◇→ 0..* BankAccount

```java
@OneToMany(cascade = CascadeType.PERSIST, …)
private Collection<BankAccount> accounts = new ArrayList<>();
```

Komposition:

BankCustomer ◆→ 0..* BankAccount

```java
@OneToMany(cascade = { CascadeType.PERSIST, CascadeType.REMOVE }, …)
private Collection<BankAccount> accounts = new ArrayList<>();
```

1.5 Vererbung

Single Table: 1 Table mit Attributen aller Subklassen

```java
@Entity\n@Inheritance(strategy =
InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type")
public abstract class BankCustomer { ... }
@Entity\n@DiscriminatorValue("Retail")
class RetailBankCustomer extends BankCustomer {}
```

Joined Table: Table pro Sub-/Superklasse, PK von Subklassen ist FK auf PK von Superklasse, Java: InheritanceType.JOINED, DiscriminatorColumn / DiscriminatorValue wie oben, keine @Id bei Subklassen
Table per Class: Table pro Sub- & nicht-abstrakter-Superklasse, PKs von Subklassen sind keine FKs, alle Attribute von Superklasse sind auch in Subklassen-Tabelle vorhanden, Java: InheritanceType.TABLE_PER_CLASS, keine DiscriminatorColumn / DiscriminatorValue / @Id bei Subklassen
Mapped Superclass: wie Table per class, aber ohne Tabelle für Superklasse

1.6 Java Persistence Query Language

JPQL, ähnlich wie SQL, aber operiert auf Entity- statt DB-Modell (Entities/Fields statt Tables/Columns), GROUP BY, HAVING, IS NULL/EMPTY & Subqueries möglich

```java
@NamedQuery(name="idbalance", query="select
distinct a.id, a.balance from BankAccount a where
a.balance >= 0 and a.balance <= 1000 order by
a.balance desc");
class BankAccount { ... }
Query q1 = em.createNamedQuery("idbalance");
Query q2 = em.createQuery("select a from
BankAccount a where a.customer.name like ?1
(oder :name)");
Query q3 = em.createQuery("SELECT c.name,
SUM(a.balance) FROM BankCustomer c JOIN
c.accounts a GROUP BY c.customerid");
Positional: q2.setParameter(1, "Bob")
Named: q2.setParameter("name", "Bob")
Limit: q2.setMaxResults(100);
```

1.7 Criteria API

Queries mit Method Chaining und Generics, *Vorteil*: statisch zur Compilezeit geprüft, *Nachteil*: unübersichtliche Schreibweise

```java
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<BankCustomer> c = cb.createQuery(
  BankCustomer.class);
Root<BankCustomer> root = c.from(
  BankCustomer.class);
c.select(root).where(cb.equal(root.get("name"),
  "Bob"));
query = em.createQuery(c);
```

```java
try { em.getTransaction().begin();
BankAccount account = em.find(...);
em.lock(account,
LockModeType.PESSIMISTIC_WRITE); // actions
em.getTransaction().commit(); }
catch(Exception e){em.getTransaction().rollback();}
finally { em.close(); }
```

1.8 Transaktionen

Default Isolation Level: READ COMMITTED (Fuzzy Read, Phantom Read, Serialization Anomaly möglich) anderer Level nützt nicht immer, da separate Connection für Read & Write

2 Stored Procedures & Cursors

User defined function Subroutine, Methode, Konzept des Procedure Calls, gespeichert nahe der Daten, Aufruf von: SQL, andere SPs, Triggers, JPA, JDBC, laufen innerhalb äusserer Transaktion, sind DB-Objekte, eigenes EXECUTE Recht zur Ausführung
Vorteile: **Datenkapselung** (Business-Logik zentral in DB, konsistentes Testing über versch. Apps), **Wiederverwendbarkeit** (ohne Code-Copy in versch. Apps), **Performance** (weniger Netzwerktraffic), **Wartbarkeit** (App-Code muss nicht angepasst werden), **Sicherheit** (weniger SQL-Injection)
Nachteile: SW-Entwicklung komplexer, macht Apps weniger portierbar, nicht benutzen wenn Views/ Subqueries/OR-Mapper reichen
***Programmiersprachen* SPs**: SQL/PSM (SQL Standard Norm), PL/SQL (Oracle proprietär), PL/pgSQL (PostgreSQL), SQL, Java, Python, usw.
SPs können als **Function** (mit Rückgabewert oder void) oder **Procedure** (ohne Rückgabewert) implementiert sein, **Signaturen** haben Parameter, eine Sprache und weitere Angaben wie Kosten, Optimierung, …

```sql
create [or replace] [function | procedure] name (
  [ [argname] argtype [,...] ] )
[ returns rettype | returns setof record ]
language [plpgsql | SQL | ...] [optimizers]
as $$
  -- ... Source Code gemäss "language"
$$;
```

Deklarationen:

```sql
DECLARE
  var1 integer; var2 integer not null;
  var3 integer default 0; var4 RECORD;
  var5 varchar := 'starting text';
  var6 angestellter.id%TYPE;
  var7 angestellter%ROWTYPE;
BEGIN ... EXCEPTION ... END;
-- Kommentar auf einer Linie
/* Kommentar auf mehreren Linien */
```

Kontrollstrukturen:

```sql
IF expression THEN ...
ELSIF expression THEN ...
ELSE       ...      END IF;
IF mod(i, 3) = 0 OR mod(i, 5) = 0 THEN ...
SELECT ... ; IF NOT FOUND THEN ... END IF;

FOR var IN query LOOP ... END LOOP;
FOR i IN 3..999 LOOP ...
CONTINUE WHEN x = 100 END LOOP;

WHILE condition LOOP ... END LOOP;

LOOP ... EXIT WHEN x > 100; END LOOP;

CASE rate WHEN 0.99 THEN price_segment = 'Mass';
  WHEN 2.99 THEN price_segment = 'Mainstream';
  WHEN 4.99 THEN price_segment = 'High End';
  ELSE  price_segment = 'Unspecified'; END CASE;

RAISE [DEBUG|LOG|INFO|NOTICE|WARNING]
'str' [expression, ...];
```

Beispiele:

```
CREATE OR REPLACE FUNCTION fn
(_customerid IN bankcustomer.customerid%TYPE)
  RETURNS address.zip % TYPE
  LANGUAGE plpgsql AS $$ DECLARE
  name bankcustomer.name % TYPE;
  zip address.zip % TYPE;
  city address.city % TYPE; BEGIN
  SELECT c.name, a.zip, a.city INTO name,
  zip, city FROM bankcustomer c, address a
  WHERE a.addressid = c.customer_addressid
  AND c.customerid = _customerid;
  RAISE NOTICE 'DEBUG: Name: %; Zip: %,
    Ort: %', name, zip, city;
  RETURN zip; END; $$;
CREATE OR REPLACE FUNCTION
check_isbn10(ISBN CHAR(12))
RETURNS boolean LANGUAGE plpgsql AS $$
DECLARE
  pos integer; ascii_val integer;
  checksum integer := 0; digits integer := 1;
  weight integer[]:=ARRAY[10,9,8,7,6,5,4,3,2,1];
BEGIN
  FOR pos IN 1..length(ISBN) LOOP
    ascii_val := ascii(substr(ISBN, pos, 1));
    IF ascii_val = 88 OR ascii_val = 120 THEN
      checksum := checksum + 10; digits := digits + 1;
    ELSIF ascii_val>=48 AND ascii_val<=57 THEN
      checksum := checksum + (ascii_val - 48)
        * weight[digits]; digits := digits + 1;
    END IF; END LOOP;
  IF digits <> 11 THEN RETURN FALSE;
  ELSE RETURN (checksum % 11) = 0;
  END IF; END; $$;
CREATE OR REPLACE FUNCTION isprime(n BIGINT)
RETURNS boolean LANGUAGE plpgsql AS $$
DECLARE
  i BIGINT := 1;
BEGIN
  FOR i IN 2..sqrt(n) LOOP
    IF mod(n, i) = 0 THEN RETURN FALSE; END IF;
  END LOOP; RETURN TRUE;
END; $$;
```

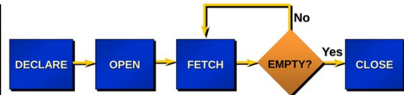
Neben **IN** (call by value, Variablen/Ausdrücke) Parameter sind auch **OUT** (call by ref, nur Variablen) und **IN-OUT** Parameter möglich, variadische Parameter sind Arrays und polymorphe Parameter sind vom Typ *anyelement*, Rückgabe-Typen: bisherige, user-defined (**CREATE TYPE**), void, **SETOF** type, **TABLE**, **Trigger**

```
RETURNS setof foo LANGUAGE plpgsql AS $$
DECLARE r foo%rowtype
BEGIN FOR r IN SELECT * FROM foo;
LOOP ..actions.. RETURN NEXT r; END LOOP;
RETURN; END; $$;
```

	FUNCTION	PROCEDURE
Use in an expression	✔	✘
Return a value	✔	✘
Return values as OUT parameters	✔ (PG Spezialität)	✔ (PG v14)
Return a single result set	✔ (as table fn.)	✔
Return multiple result sets	✘	✔
Can contain a transaction begin/end	✘	✔
Make it run using ...	EXECUTE/SELECT	CALL

```
@NamedStoredProcedureQuery(name = "test",
procedureName = "mysum", parameters = {
@StoredProcedureParameter(mode =
ParameterMode.IN, type=Double.class, name="x")})
```

SELECT liefert in der Regel eine Menge von Tupels, Cursors erlauben sequentiellen Zugriff auf die einzelnen Tupels, SELECT/UPDATE möglich, es gibt Cursors **unbound** (noch ohne FOR), **bound** (mit FOR) und **parametrisiert bound**, ganze Tupels fetchen: Variable vom Typ **RECORD** nach **INTO**




```
DECLARE
  curs1 refcursor; -- Unbound Cursor
  curs2 CURSOR FOR SELECT * FROM abteilung;
  -- oben Bound Cursor, unten parametrisierter bound C.
  curs3 CURSOR (abtid INTEGER) FOR
  SELECT name FROM abteilung WHERE id=abtid;
BEGIN
  OPEN curs1 FOR SELECT * FROM abteilung;
  ... CLOSE curs1; OPEN curs2; ...  CLOSE curs2;
  OPEN curs3 (3);
  LOOP
    FETCH curs3 INTO abtname;
    EXIT WHEN NOT FOUND;
    RAISE NOTICE 'Name: %', abtname;
  END LOOP; CLOSE curs3; END;
```

3 Triggers

Impl. von komplexen Konsistenzbedingungen, Sicherheit (z.B. Änderungszeit beschränken), abgeleitete Attribute berechnen, Statistik- & Logdaten sammeln, Updatable Views, DB-Objekte, an Tabelle zugeordnet, in SPs programmiert, keine Parameter, direkter Aufruf möglich, werden vom DBMS bei DB-Event (INSERT, UPDATE, DELETE, TRUNCATE) aufgerufen, Rechte ihres Owners, **Probleme**: DB wird langsamer & schwerer wartbar, cascading Triggers können Endlosschlaufen erzeugen

```
CREATE TRIGGER emp_stamp
  BEFORE INSERT OR UPDATE
  ON emp FOR EACH ROW
  EXECUTE PROCEDURE emp_stamp();
CREATE FUNCTION emp_stamp()
  RETURNS trigger language plpgsql AS $$
  BEGIN
    IF NEW.empname IS NULL THEN
      RAISE EXCEPTION 'name null' END IF;
    NEW.last_date := current_timestamp;
    NEW.last_user := current_user;
    RETURN NEW;
END; $$;
```

Variablen innerhalb Trigger Function: **TG_NARGS** (Anz. Param.), **TG_ARGV[]** (Param. Array als text), **TG_NAME** (Trigger-Name), **TG_WHEN** (before/after), **TG_LEVEL** (row/statement), **TG_OP** (Event-Typ z.B. insert), **TG_RELID** (Object-ID Tabelle), **TG_RELNAME** (Tabellenname), **TG_TABLE_ SCHEMA** (Tabellen-Schema), **NEW** (Zugriff auf neue Werte einer Zeile, die in eine Tabelle inserted / updated wird) **OLD** (Zugriff auf alte Werte einer Zeile, die aus einer Tabelle deleted / updated wird),

4 Updatable & Materialized Views

View ist automatisch updatable wenn: 1 Eintrag (Tabelle/updatable View) in FROM, kein WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET, UNION, INTERSECT oder EXCEPT, keine Aggregation, Window-Funktion oder SET-Returning-Funktion, mit Triggern können non-updatable Views updatable gemacht werden:

```
create view vi as select abtnr,
avg(salaer)::decimal(10,2) as avg_salaer from
angestellter group by abtnr;
create function vitrigger() returns
trigger language plpgsql as $$ begin
update angestellter set salaer=salaer+
(new.avg_salaer-old.avg_salaer) where abtnr=
old.abtnr; return new; end; $$;
create trigger vitrigger instead of update
on vi for each row execute procedure
vitrigger();
```

Materialized View: virtuelle Tabelle, zwischen-gespeichert, wie normale Tabelle verwenden

```
CREATE MATERIALIZED VIEW mv AS SELECT...
REFRESH  MATERIALIZED VIEW mv;
```

Views sind langsam, mat. Views sind schnell aber potentiell nicht aktuelle, Refresh-Möglichkeiten: **Snapshot** (1x), **Eager** (wenn unterliegende Tabellen geändert werden), **Lazy** (bei Transaction-commits), **user-defined**, **pro Intervall**

5 Datenstrukturen

Basis-Datentypen: boolean, char(n), varchar(n), date, integer, json(b), decimal(p,s), text, time(tz), timestamp(tz), uuid, xml

Abstrakter Datentyp (ADT): Konzept / Impl. Trennen, z.B. Liste als Array, Bestandteile: Name, innere Datenstruktur, Konstruktor, Accessor- / Update- / Hilfs-Funktion, Operator, Index auf Attribut

DOMAIN: user-defined Datentyp, für Datenschema & SP, Constraints wie NOT NULL/CHECK unterstützt

TYPE: zusammengesetzter Typ, für Datenschema & SP, ENUM-Support, keine Constraints

```
CREATE DOMAIN contact_name AS
VARCHAR(255) NOT NULL CHECK (value !~'\s');
CREATE TYPE tb AS ( outer: text, inner: text );
```

Collections/Containers: Gruppierung variabler Anzahl Datenelemente des gleichen Typs,

5.1 Lineare Collections

Priority Queues/Heaps, Arrays (Liste): viele Hilfsfunktionen, geeignet bei wenig Updates/Inserts, nicht relational, guter Support

```
create table t (arr text[][]);
insert into t values (array[['a1','a2'],['b1','b2']]);
select arr[1:2][1:1] from t; -- {{a1},{b1}}
select arr[2][4] from t; -- b4, Zeile 2, Spalte 4
select array[3,2,1] = array[1,2,3]; -- f
select array[1,7,2] @> array[2,7]; -- t (contains)
select array[1,4,3] && array[2,1]; -- t (>0 gem. Elem.)
select * from unnest(array[row(1, 2), row(3, 4)])
t(a int, b int); -- siehe Tabelle unten
create table session_log (id int, session_os text);
insert into session_log values(641, 'android'),
(641, 'ios'), (642,'android'), (642, 'windows'), (644, 'ios');
select id, array_agg(session_os)
from session_log group by id having
array_agg(session_os) && array['android'];
```

a	b	id	array_agg
1	2	641	{android,ios}
3	4	642	{android,windows}

Helpers: array_dims(arr), array_upper / array_lower(arr,dim), array_length(arr,dim), array_to_string(arr), unnest(arr)

5.2 Assoziative Collections

Sets, Bags/Multisets, Dictionaries (Associative Array, Map, Key-Value Paar, Hash): für variable, unvorhersehbare, unkoordinierbare Werte, schema-less, umstritten, Zeilen mit vielen Attributen, die selten untersucht werden, semi-strukturierte Daten

```
select 'a=>1,a=>2,b=>3'::hstore;--"a"=>"1","b"=>"3"
select akeys('k1=>1, k2=>2'::hstore); -- "{k1,k2}"
select 'k1=>1, k2=>2'::hstore ->'k2' -- 2 vom typ text
select 'k1=>1, k2=>2'::hstore @> 'k1=>1'; -- true
```

5.3 Graphen & Trees

Ein Graph **G**=(V,E) ist ein Set von Vertices (**V**={v1,v2,…,vn}) und Edges (**E**={e1,e2,…,en}), ein Edge verbindet 2 Vertices (**en**={vx,vy})

Eigenschaften: (un)directed, (a)cyclic, Path, Subgraph, Umgang mit rekursiven Abfragen mittels CTE

Anwendungen: Transport, Energie, Navi / GIS

Tree: azyklisch verbundener Graph, **Rooted Tree**: hierarchical, connected, acyclic, **Ordered Tree**: Ordnung der Kinder definiert, **Binary Tree**: jeder Knoten hat 0-2 Kinder, ***Anwendungen***: Führungshierarchie, Stücklisten, Kataloge

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

Implementationen:

	Adjacency List	*Nested Set*	*Materialized Path*
Suchen	Langsam	Eher langsam	Eher langsam
Ancestors suchen	Schnell	Langsam	Schnell
Childs finden	Meist schnell	Langsam	Langsam
Parent finden	Meist schnell	Langsam	Schnell
Aggregierte Queries	Eher schnell	Eher schnell	Eher schnell

5.4 JSON

2 JSON-Datentypen: **json** (exakte Text-Kopie), **jsonb** (geparstes Binärformat, Input langsamer, Verarbeitung schneller, Index-Support)

```
select '{"k1":{"k2":"v1"}}'::json->'k1';-- {"k2":"v1"}
select '{"k1":{"k2":"v1"}}'::json->>'k1';-- {"k2":"v1"}(txt)
select '{"k1":1}'::json || '{"k2":2}';  -- {"k1":1,"k2":2}
select '{"k1":1,"k2":2}'::jsonb @> '{"k2":2}'; -- true
select '{"k1":1,"k2":2}'::jsonb - 'k2'; -- {"k1":1}
select data ? 'canton', data @? '$.canton', -- 2x true
  jsonb_path_match(data, 'exists ($.canton)') -- true
from (values(1, '{"canton": {"sname": "zh", "lname":
  "zürich"}}'::jsonb)) as tmp(id, data);
--jsonb_path_xxx(data jsonb, path jsonpath,
--vars jsonb default '{}', silent boolean default false)
```

Return Bool: exists (Pfad (z.B. $.address) existiert?), match (Pfad-Prädikat (Beispiel oben) existiert?)

Return jsonb setof/array: query, query_first, query_array, **Return text**: jsonb_extract_path_text()

Methoden für JSON Paths: .type(), size, ceiling, double, floor, abs, keyvalue (JSON {id,key,value})

```
select jsonb_path_query(data, '$.articles[*] ?
(@.product == "gadget") ? (@.price > 150)')
from orders_json;
SELECT json_build_object('persnr', a.persnr, 'name',
a.name, 'projnrs', json_agg(p.projnr)) AS ang_json
FROM angestellter a
JOIN projektzuteilung p ON a.persnr = p.persnr
GROUP BY a.persnr, a.name;
```

@ für aktuelles JSON Item, **Vergleichsoperatoren**: ==, != , <>, < , >, <=, >=, exists, like_regex, starts with, is unknown

6 Query-Optimierung & Indexe

Prädikat: Ausdruck, der eine true/false/unknown Bedingung auswertet, ***Query-Arten***: **Equality** (mit = Operator), **Range** (mit >/>=/ </<=/!= Operator), **Point** (mit = Operator, nur ein Return-Tupel), **Join**, **Weiteres** (Text-Präfix/ Suffix, Extremwerte bei Zahlen)

Heap: Collection von Page Sets

Page Sets: Collection von Pages

Data Page: Tabellendaten physikalisch als Datei, Zeilen unsortiert, Page Split wenn Page voll, Row Chaining wenn Zeile nicht Platz in einer Page hat

Sequential Table Scan: bei erster Page starten, alle durchgehen, matching Rows extrahieren, schneller als Index bei 80% Resultset Grösse

6.1 Indexes

Primär-Index: Index mit PK, immer unique & not null

Clustered Table in Postgres: **CLUSTER table USING index**; Daten werden physisch sequentiell angeordnet gemäss den Index Informationen, One-Time Operation, Changes danach werden nicht automatisch clustered

Clustered, integrierter Index in MSSQL: Daten werden laufend automatisch physisch sequentiell angeordnet gemäss den Index Informationen, die Blätter enthalten die Daten vom Rest der Columns, nicht nur Referenzen

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

B-Baum: Balancierter Mehrwege-Suchbaum, Allrounder, fast optimal für viele Queries/Inserts

B+-Baum: Knoten haben Keys (wie gehabt), weitere Ebene mit integrierten Leafs, diese haben Keys nochmals und sind verkettet, schnellere Range Q.

Hash-Index: speichert K/V Paare basierend auf Hash-Funktion, Überlaufkette für versch. Keys mit gleichem Hash verschlechtert Performance, gut bei Point/Multi-Point ohne Überlaufketten, unbrauchbar für Range, Präfix und Extremwert Q.

Bitmap: für Attribute mit geringer Kardinalität (1% der Datenmenge), schnelle AND/OR Anfragen, langsam bei Modifikationen, es wird eine Tabelle mit einer Column pro diskretem Wert erstellt, pro Tupel wird dann die zutreffende Column auf 1 gestellt und die anderen auf 0, für Data Warehouses, Postgres kennt ihn nicht persistent, wird aber intern ggf. verwendet (Bitmap Index/Heap Scan), Alternative: BRIN

GiST: baumartige, balancierte Struktur, Range / Containment Suche, K-Nearest-Neighbour Suche, für geometrische Datentypen (Nachbar), Volltextsuche

SP-GiST: unbalancierter Baum, KNN, 2D/3D-Daten

GIN: Generalized Inverted Index, Liste von Worten, die auf Dokumente zeigen, speichert Duplikate effizient, gut für strukturierte Werte wie Arrays/JSONB/hstore, schneller Zugriff, grosser Speicherbedarf, langsam zum Erstellen/Updaten

BRIN: Block Range Index, speichert min/max Werte als Blöcke, gut bei Range, natürlich benachbarte / sortierte Daten, nutzt Nähe, z.B. PLZ, wenig Diskssp.

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

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

```
create index name on table (a1, a2, ...) using
method include a3 where predicate;
```

Zusammengesetzt: mehrere Attribute, geeignet für Q. auf a1, a1 AND a2 und gewisse Q. auf a2, jedoch nicht Suffix-Q. mit LIKE

INCLUDE: zusätzliches Attribut mitspeichern aber nicht mitindexieren, **Partieller Index**: mit **where**, **mit Funktion/Ausdruck**: z.B. **lower**(a1)

6.2 Query Planer

1. **Transformation**: Syntax-Baum der Query erstellen
2. **Logische Optimierung**: Anfrageterm umformen aufgrund von Heuristiken
3. **Physische Opt.**: Ausführungspläne generieren, Statistiken (Anz. Tupel/Pages, Verteilungsinfos, Index-Grösse/-Höhe), Indexes & Herustiken miteinbez.
4. **Kosten** (Kommunikation, CPU, I/O, Speicher) **berechnen** und **günstigsten Plan wählen**

Table Scans: ***Full Table Scan (Seq Scan)***: ganze Tabelle ab Disk scannen, schnell für kleine Tabellen

Index Scan: scannt alle/einige Zeilen im Index und sucht

dann mit random seek im Heap
Index only scan: scannt alle/einige Zeilen im Index, keine weiteren Zugriffe, da alle Werte im Index
Bitmap Heap/Index Scan: Index Scan lädt alle Tupel-Zeiger aus dem Index, generiert ad-hoc ein Bitmap, wertet dann die Prädikate aus, dann Heap Scan der matching Tupel
Join Strategien:
Nested Loop Join: für alle Zeilen der rechten Tabelle werden alle Zeilen der linken Tabelle nach Match gescannt, schneller Start, gut für kleine Tabellen
Merge Join: Alle Tupel werden vor dem Start nach Join-Attributen sortiert, dann werden sie parallel gescannt und matches ausgegeben, langsamer Start, gut für grosse Tabellen
Hash Join: rechte Tabelle wird in eine Hash-Tabelle geladen mit Join-Attribut als Hash-Key, dann wird die linke Tablle gescannt und jede gefundene Zeile als Hash-Key verwendet, nur für Equality, langsamer Start, schnelle Ausführung
EXPLAIN: geplanten execution plan ausgeben, **EXPLAIN ANALYZE**: Query ausführen und Plan mit effektiven Ausführungskosten ausgeben
(cost=0.15..23.44 **rows**=2 width=8) "0.15"=Startup-Kosten, "0.15..23.44"=Gesamtkosten, "2"=geschätzte Tupel-Anzahl, "8"=geschätzte Tupel-Breite **in** Bytes
Cost: Schätzung der benötigten Ressourcen (wie CPU-Zeit und I/O-Zugriffe)
Selektivität: prozentualer Anteil Tupels, die Query liefert (*Tupel/Total Tupels*), hoch=wenige Tupels, kann geschätzt werden mit Wertebereich & where Kond.
Beispiel: Tabelle hat 1112 Tuples, Wertebereich Attribut quantity zwischen 1 & 353, 237 untersch. Werte, Query:

```
select * from table where quantity between 210 an 300;
```

Ohne Histogramm:

$$\frac{rowsselected}{totalrows} = \frac{(300-210+1)*\frac{237}{353}*\frac{1112}{237}}{1112} = \frac{91}{353} = 0{,}258$$

Mit Histogramm:
Werte zwischen 210 und 300 liegen in Bucket 10
Selektivität = 0.1

number	value (max)
1	27
2	42
3	57
4	74
5	98
6	123
7	149
8	175
9	202
10	353

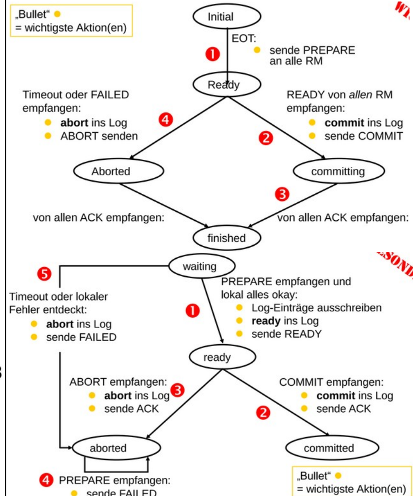
Dichte: durchsch. prozentueller Anteil Duplikate, hoch=viele Duplikate/tiefe Selektivität, tief=unique Idx
Tipps für Indexe und schnelle Queries: JOIN verwenden, numerische Vergleiche statt LIKE, schnellere Vergleiche am Anfang, Attribute statt *, JOIN statt Subquery, UNION statt JOIN, WHERE statt UNION, nur Indexe erstellen die verwendet werden, Covering Index mit allen beteiligen Attributen erstellen wenn möglich, Index für Attribute: PK, FK, Range Q., ORDER BY, GROUP BY

7 Verteilte & replizierte Systeme

Verbessert **Reliability** und **Availability**, erhöht **Komplexität**, **Anforderungen**: Zugriff auf übers Netz verbundene Systeme (Foreign Data Wrapper), Datenverteilungs-Transparenz (Queries bleiben unverändert), Atomarität von verteilten Transaktionen, Query Planer funktioniert verteilt
Homogen / Heterogen: identische Software / Zusammenarbeit / untersch. Softare & Schema (Problem bei verteilten Queries & Transaktionen)
Federated / Unfederated: eigenständige DBs (versch. DBMS) zu einem System verbunden ohne Autonomie zu verlieren / zentrales Mgmt-DBMS, DBs verlieren Autonomie
Fragmentierung: DB in logische Fragmente für versch. Nodes aufteilen, **vertikal**: aufteilen nach Attributen, **horizontal**: aufteilen nach Tupels, in Postgres & MongoDB (Sharding) verwendet
Replikation: gleiche Daten auf mehreren Nodes speichern, bessere Availability, Performance & Parallelität, höhere Update-Kosten & Komplexer Sync
Allokation: (replizierte) Fragmente Nodes zuordnen
Scale up / out: HW-Ress. erhöhen / Nodes adden

7.1 Verteilte Transaktionen

2-Phase Locking Protocol braucht 1 Transaktions-manger & pro Node einen Resource Manager, muss mit Nachrichtenverlust & Absturz von TM/RM zu bestimmten Zeitpunkten umgehen können



Phase 0: Update schreiben
Phase 1: Prepare to commit (TM fragt RM, ob sie die Transaktion mit *commit* abschliessen können, RM antworten mit *ready* und gehen in *prepared* Zustand)
Phase 2: Commit (wenn alle RM prepared, schickt TM commit, RM antworten mit *ack* oder *abort / timeout →* TM schickt *abort* an alle)
Nachteile: hohe Latenz, geringe Parallelität, Ressourcenverbrauch, Möglichkeiten zu Inkonsistenz
Paxos: 2PC Verb., Mehrheitskonsens, Proposers (machen priorisierte Vorschläge), Acceptors (nehmen Vorschlag mit höchster Prio an) & Learners (Exec)

7.2 Verteilte Anfragen

Query Performance Single Node nur durch Disk-I/Os bestimmt, bei Multi Node auch durch Netzwerk-I/Os, bei Fragmentierung werden möglicherweise Daten von anderen Nodes benötigt
Ship Whole: alle Daten zu Query-Node übertragen
Query Pushdown: ganze Query oder Teile werden delegiert, z.B. Predicate (where), Aggregate (avg usw.) und Joins können pushed werden
Fetch-as-needed: bei joins, pro Tupel wird Wert des join Attributs an anderes Node gesendet, das das passende Tupel sendet

8 NoSQL

Einfaches API (HTTP), für grosse Datenvolumen, skalierbar, oft als Cloud-Storage, nicht-relational, Schema-frei, BASE Architektur, kein ACID, keine Query-Norm, Open Source
Weitere Kategorien: In-Memory (Columnar) Stores, Object-oriented DBMS, Array Stores (Rasterfragmentierung (GIS), VectorDB)
ACID: Atomicity, Consistency, Isolation, Durability, traditionelle relationale Datenbanken, hohe Konsistenz, zuverlässige & vorhersehbare Transaktionen
BASE: **Basically Available** (Antwort auf jede Anfrage, kann auch sein, dass die Daten nicht verfügbar / inkonsistent / sich ändernd sind), **Soft-State** (Systemzustand kann sich auch ohne Eingaben ändern), **Eventually-Consistent** (System wird schliesslich konsistent ohne Eingaben), bei modernen Internet-Systemen
CAP: **Consistency** (Linearisierbarkeit im Transaktionsprotokoll), **Availability** (jeder nicht ausgefallene Node muss antworten), **Partition Tolerance** (funkt. bei teilw. Netzwerk-/Node-Ausfall), pick two (RDBMS=CP, NoSQL=AP), verteilte Systeme, optimiert für Skalierbarkeit & Verfügbarkeit
Ziele: Hochleistungsrechnen (z.B. HPC), Hochverfügbarkeit (HA), geringe Latenz

9 NoSQL: Key/Value Stores

Einfache Hash-Tabelle, auf die nur über ihren PK zugegriffen werden kann, im Grunde Struktur mit ID & Value, Value ist Text ("CLOB/BLOB") oder Basisdatentyp, Konsistenz nur bei Operationen mit einem einzigen Key
Produkte: Redis, Amazon DynamoDB, Project Voldemort, Oracle Berkeley, Riak, Memcached, HamsterDB
Anwendungsbereiche: No-schema, Caching, Session-Infos speichern, User-Profiles/-Settings, Shopping Carts
Not to use: Beziehungen zwischen Daten, Multi-Operations-Transaktionen, Datenabfragen, Oper. nach Gruppen

10 NoSQL Document Store Mongo

Document Data Model (schema-less), einfache MongoDB Query Language (MQL), SQL-Frontend, Partitioning mit Sharding, Availability mit Replikation & Quorum-Konsens-Algo. (Raft-ähnlich), Dokumente haben KV-Paare und dynamisches Schema (logisch analog JSON), auf Disk in BSON (binäres JSON), keine Constraints für Felder & referntielle Integrität
RDBMS → MongoDB: Table → Collection, Row → Document, rowid → _id,objectid
Vorteile: Prototyping, Web-freundlich (json over http)
Nachteile: keine echten Joins, langsame/mangelhafte Transaktionen, Default offen im Web

```
var class = { _id: ObjectId ("509980df3"),
course: {code: "DatEng", title: "Data Engineering"},
year: 2024, students: ["Peter", "Manuel"] }
```

_id: PK, unique, immutable, indexiert, kann von jedem Typ ausser Array/Regex sein, üblich sind: ObjectId (4B Sek. seit Unix-Epoche, 3B Maschinen-Kennung, 2B Prozesskennung, 3B Counter Start random), natürliches Attribut, auto-incremeted Zahl, immer erstes Feld
Transaktion auf einzelnes Dokument ist atomar, bei mehreren Docs/Collections gibt es multi-document transactions (MDT), nicht empfohlen vom Hersteller
Replica Set: Primary wird elected aufgrund Prio, RAM oder näher an anderen Servern, alle Requests gehen dahin, Daten werden repliziert und bei Ausfall wird neuer Primary elected, **Write Concerns**: Unacked / Acked / Journaled / Replica Acked / Majority, **Read Concerns**: local / available / majority / linearizable / snapshot
Start jeweils db.unicorns.:

```
insert({"name": "Arris", "gender": "m",
"loves": ["apple","grape"], "weight": 610});
find({"gender": "m", "loves": {$in: ["apple", "carrot"]}});
find({"gender": "m", "weight": {$gte:600, $lte:900}});
update({"name": "Arris"}, {$set: {"weight": 590}},
{"upsert": false});
remove({"name": "Arris"}); // {} zum Collection leeren

// {"manager": ObjectId("xxx")}
db.employees.findOne({"_id":
db.employees.findOne({"name":"Moneo"}).manager});
```

```
// {"manager": "Leto"}
db.employees.aggregate([{"$graphLookup": {
  "from": "employees", "startWith": "$manager",
  "connectFromField": "manager",
  "connectToField": "name", "as": "managerDocs",
  "maxDepth": 0 }},
  { "$match": {"name": "Moneo"}},
  { "$project": { "_id": false, "name": true,
    "manager": true, "managerDocs": true }}]);
db.employees.aggregate([{"$lookup": {
  "from": "employees", "localField": "manager",
  "foreignField": "name", "as": "managers" }},
  { "$match": {"name": "Moneo"}},
  { "$project": {"_id": false, "manager": false,
    "managers": { "_id": false, "manager": false }}}]);
```

10.1.1 DB Modellierungs-Entscheide

Applikationsbedürfnisse: queries, updates, data processing, Performance, Data Retrieval Patterns
Sharding: automatische horizontale Fragmentierung & Verteilung auf versch. Nodes, reduziert I/O
Sharding Key: sollte Daten gleichmässig auf Nodes verteilen, Funktion auf ein Feld oder eine Kombination von Feldern, sollte nicht unique sein (ID/PK geht nicht), kann eine Hash Funktion sein (dann geht ID/PK doch), Mongo Default: Hash auf _id Feld
Embedding vs. Referencing: im Doc speichern vs. mit Referenzen verlinken, wenn die Doc-Grösse den zugewiesenen Platz übersteigt, wird es relocated, was langsam ist, referencing braucht weniger Platz, aber ist langsamer

11 NoSQL: Graph Stores

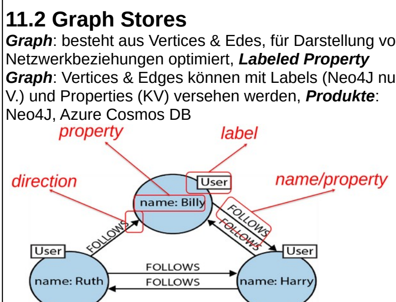
11.1 Triple Stores (RDF)

Resource Description Framework mit Tripeln (Subjekt-Prädikat-Objekt), Standardmodell für Datenaustausch im Web, ***Produkte***: MarkLogic, Virtuoso, Apache Jena, GraphDB, Amazon Neptune, Stardog, AllegroGraph

```
_:markus rdf:type ex:Person.
_:markus ex:vorname "Markus".
_:vreni rdf:type ex:Person.
_:vreni ex:vorname "Vreni".
<http://x.ch/Markus> <http://x.ch/vorname> "Markus".
<http://x.ch/Vreni> <http://x.ch/vorname> "Vreni".
```

11.2 Graph Stores

Graph: besteht aus Vertices & Edes, für Darstellung von Netzwerkbeziehungen optimiert, ***Labeled Property Graph***: Vertices & Edges können mit Labels (Neo4J nur V.) und Properties (KV) versehen werden, ***Produkte***: Neo4J, Azure Cosmos DB



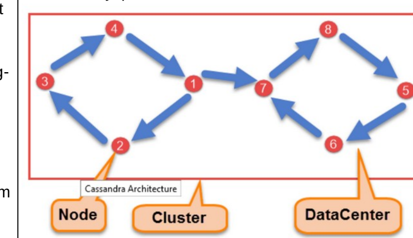
Cypher Query: deklarativ, Struktur & Keywords inspiriert von SQL (select → return, from → match, where → where), joins sind implizit
Use Cases: verbundene Daten, Routing, Dispatch, standortbasierte Services, Empfehlungs-Engines, Knowledge Graphs, semantische Netze
Not use: wenige Sprünge in Graphen, alle oder Teilmenge von Entitäten aktualisieren

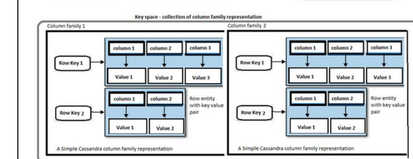
```
CREATE(keanu:Person{name:'Keanu R.',born:1964})
MATCH (k {name: 'Keanu R.'}) RETURN k
MATCH (p:Person) RETURN p.name LIMIT 10
MATCH (tom:Person {name:"Tom Hanks"})
-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors)
RETURN coActors.name
MATCH (bacon:Person {name:"Kevin Bacon"})
-[*1..4]-(hollywood) RETURN DISTINCT hollywood
MATCH (n) DETACH DELETE n
MATCH (tom:Person {name:"Tom Hanks"})
-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
(coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-
(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

12 NoSQL: Column (Family) Store

12.1 Column Family Store Cassandra

Row Key mit Column Family, welche Paare von Column Keys & Column Values hat,
Alternativen: Google BigTable, Apache Hbase & Cassandra, Hypertable, Yahoo PNUTS
Use Cases: Event Logging, CMS, Blogging, Website Analytics, **Not to use**: ACID, Aggregation (z.B. sum/avg), Prototypes (teure Column Änderungen)
Hochverfügbar, inkrementelle Skalierbarkeit, optimistische Replication, eventuel consistency, keine Transaktionen, Atomarität auf dem Row Level, wenig Administration, DB → Keyspace





Partitioning: jeder Datensatz einer Tabelle hat einen Partition Key, der bei der Erstellung der Tabelle definiert wird, durch Hashing wird der Partition Key in einen Token umgewandelt, dessen Wert in einem kontinuierlichen Ring ist, jedes Node bekommt einen Range dieses Rings, somit können alle Nodes jederzeit berechnen, auf welchem Node ein Key ist
Indexing: andere Columns als die Keys der Column Family können indexiert werden

```
create columnfamily customer (key varchar primary
key, name varchar);
insert into customer values ('mfowler','Martin F.');
select key, name from customer;
```

12.2 Column Store

Jede Column ist separat gespeichert
OLTP: Online Transaction Processing, wenige Daten, viele Transaktionen, normalisierte Daten, ACID-compliant, HA zwingend
OLAP: viele Daten, wenige Transaktionen, denormalisierte Daten, nicht unbedingt ACID-compliant, HA nicht zwingend
Row Store: Normalfall, zeilenweise Speicherung, gut für z.B. select *, gute Bandbreiten-Ausnützung zwischen CPU-Kern und RAM, einfaches horizontales Partitioning, RAM-Zugriff ist Engpass, Tupel-Konstruktion ist aufwändig, da immer alle Spalten gelesen werden müssen, gut bei OLTP, schlecht bei OLAP
Column Store: spaltenweise (komprimierte) Speicherung, gut für attributierten Zugriff, z.B. sum, group by, besseres Caching, weniger Speicherverbrauch, meist automatisch Index auf jede Column, schreiben langsamer, gut bei OLAP, schlecht bei OLTP
Komprimierung: reduziert Speicherzugriffe, weniger vergebliche Cache-Zugriffe, Operationen sind zum Teil direkt auf komprimierten Daten möglich, Default bei grossen Datentypen, günstig für beschränkte Kardinalitäten, Optionen: zip, pglz, lz4, RLE
Dictionary Encoding:

Rec ID	fname
...	...
39	John
40	Mary
41	Jane
42	John
43	Peter
...	...

Dictionary for "fname"

Value ID	Value
...	...
23	John
24	Mary
25	Jane
26	Peter
...	...

Attribute Vector for "fname"

position	Value ID
...	...
39	23
40	24
41	25
42	23
43	26
...	...

12.2.1 In-Memory Stores (IMDBMS)

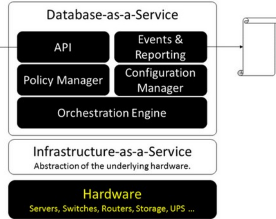
Daten werden beim Start vollständig in den RAM geladen, im Gegensatz zu On-Disk 10-1000x schneller, wenn In-Memory optimiert, mit Column-Architekturen kombiniert oder spezialisiert, geänderte Daten werden regelmässig auf die Disk geschrieben, laufende Änderungen werden in Transaction Logs geschrieben (für Rollforward bei Crash), ACID-compliant, keine Puffer-Caches, weniger Overhead wegen direkter CPU-Anbindung, oft einfacher designed, daher weniger CPU-/RAM-Bedarf, gut bei OLAP, oft mit Column Store kombiniert, schlecht bei OLTP, **Beispiele**: SAP HANA, Oracle TimesTen

12.2.2 DuckDB

In-Process analytical column store, "SQLite for analytics", **vektorisierte** (pull-based Iteration über Row-Value-Gruppe=Vektor) Ausführung in CPU (vs. Volcano-Ausführung (zeilenweise)), keine externen Konfigurationsdateien oder Einstellungen, **In-Memory optimierte Verarbeitung**, **Abfrageoptimierung**, erweiterbar, **SQL**-kompatibel, **ACID**-compliant, **Min-max Index** autom. Erzeugt für Columns aller Datentypen, **Adaptive Radix Tree** (ART) autom. erzeugt für PK & unique Constraints, **direkt lesen**: Pandas & Postgres (Postgres scanner Duck Extension, filter_pushdown für where-Pushdown kann aktiviert werden), **parallelisiert lesen**: CSV, TSV, JSON, Apache Arrow, Parquet

13 DBaaS & GraphQL

Web Services verwenden **http** für einfachen Zugriff auf remote Ressourcen (**API**), Ansatz ist in heterogenen, verteilten, lose verbundenen Umgebungen von Vorteil, mit DBaaS aus der Cloud können Web-Client-Anwendungen **CRUD auf Daten** ausführen und Views & SPs aufrufen, **kein IaaS** (höhere Abstraktionsebene als VMs), **kein PaaS** (generische Schnittstellen, Authentifizierung & Autorisierung)
Architektu



SQL vs. NoSQL: Cheatsheet for AWS, Azure, and Google Cloud
ml4devs.com/datastores

Data Type	Use Case	Type	aws	Azure	Google Cloud	Cloud Agnostic
Structured	ACID Transactions (OLTP)	Relational	RDS, Aurora	Azure SQL Database	Cloud SQL, Cloud Spanner	SQL Server, Oracle, DB2, MySQL, PostgreSQL
Structured	Analytics (OLAP)	Columnar	RedShift	Azure Synapse	BigQuery	Snowflake, ClickHouse, Druid, Pinot, Databricks
Semi-structured	Dictionary	Key-Value	DynamoDB	Cosmos DB	BigTable	Redis, ScyllaDB, Ignite
Semi-structured	Cache	In-memory	ElastiCache	Azure Cache for Redis	Memory-store	Redis, Memcached, Hazelcast, Ignite
Semi-structured	2-D Key-Value	Wide Column	Keyspaces	Cosmos DB	BigTable	HBase, Cassandra, ScyllaDB
Semi-structured	Time Series	Time Series	Timestream	Cosmos DB	BigTable, BigQuery	TimescaleDB, OpenTSDB, InfluxDB, ScyllaDB
Semi-structured	Audit Trail	Immutable Ledger	Quantum Ledger Database (QLDB)	Azure SQL Database Ledger	✗	Hyperledger Fabric
Semi-structured	Location & Geo-entities	Geospatial	Keyspaces	Cosmos DB	BigTable, BigQuery	Solr, PostGIS, MongoDB (GeoJSON)
Semi-structured	Entity-Relationships	Graph	Neptune	Cosmos DB	JanusGraph + BigTable	OrientDB, Neo4J, Giraph
Semi-structured	Nested Objects (XML, JSON)	Document	Document DB	Cosmos DB	Firestore	MongoDB, Couchbase, Solr
Semi-structured	Full Text Search / (Rich) Text	Text Search	Open-Search, Cloud-Search	Cognitive Search	Search APIs on Datastores	Elastic-Search, Solr, Elassandra
Unstructured		Blob	S3	Blob Storage	Cloud Storage	HDFS, MinIO

© Satish Chandra Gupta
CC BY-NC-ND 4.0 International License
creativecommons.org/licenses/by-nc-nd/4.0/
scgupta.me
twitter.com/scgupta
linkedin.com/in/scgupta

13.1 GraphQL

deklarative Datenabfrage- und Manipulations-Sprache für HTTP-APIs, **schema-basiert**, Beschreibung der Daten als API, wickelt alle Anfragen über einen **einzigen Endpoint** ab, macht es einfach, APIs weiterzuentwickeln (Versioning), unterstützt von **allen Elefanten**, Resultsets repräsentieren **JSON-Bäume**, jedoch **Graphen-Nomenklatur** (Nodes/Edges) für verschachtelte Strukturen und Paging, hat sonst **wenig mit Graphen zu tun**
SDL: Schema Definition Language, ist eine DDL, Schema ist Collection von object types mit fields, wobei jedes field einen type hat, der type kann scalar (int, float, string, boolean, id, custom) oder ein anderer object type sein, weitere types: enum, list, union, interface, etc., der type Query dient zum CRUD der Daten, mutation type (CUD) ist optional
Vorteile schema-first ggü. code-first: Schema-Definition erhöht Lesbarkeit, verbessert Kommuni-kation zwischen Frontend-/Backend-Teams, ist Doku-mentation, rasche SW-Entwicklung (API-Mocking)
Client-SW (JavaScript): Relay (React, modern), Apollo Client (schema-first)
Server-SW: Apollo Server (JS), Prisma.io (JS, code-first), Hasura (Postgres, schema-first), PostGraphile (Postgres, schema-first), AWS AppSync (schema-first), GraphQL.NET, HotChocolate (.NET)
Best Practices: null für Felder nicht erlauben, DocStrings (""") verwenden, zusammengehörtige Attribute zu einem type zusammenfassen, entscheide wie mit Fehler umgehen in GraphQL
Vergleich zu SQL: weniger mächtig auch mit Erweiterungen, kontrollierbar (garantierte Performance und Availability)
Vergleich zu REST: weniger Roundtrips (N+1 Problem entschärft), nur nötige Daten werden geliefert, weniger strukturelle Einheit aller APIs, kein Multi-User Support, komplexer

13.1.1 PostGraphile

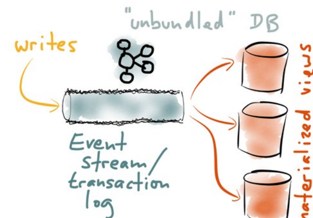
Zwischenstück Postgres – GraphQL, NodeJS, API unter /graphql, Web-GUI unter /graphiql, Resolvers sind hauptsächlich SQL, aus der gegebenen Tabelle ang macht PostGraphile folgendes: **Type Ang** (PascalCase, Singular), **umbenannte Felder** (camelCase ohne _), ergänzt **Feld nodeId** falls PK existiert, ergänzt je ein **Feld für jede FK-Beziehung** (z.B. abteilungByAbtnr), erzeugt als Root Query Types: **Connection allAngs** (mit Pagination, condition, filter & order), **Felder für jeden Unique Constraint** (z.B. angByPersnr), ein **foo(nodeId: ID!)** (! =obligatorisches Attribut), erzeugt als Root **Mutation Types** (deaktivierbar): createAng, updateAng, deleteAng
Pagination: für nummerierte und sequentielle Seiten oder unendliches Scrollen, man könnte mit offset vorherige Seiten überspringen, das wird aber immer langsamer, je grösser, besser ist die Seek-Methode mit einem where Statement oder einem Cursor
Graphen-Nomenklatur: verschachtelte Strukturen in paginated Resultsets müssen benannt werden, Connection=Edge-Listen, eigentliches Objekt=Node, hat Metadaten und Cursor, Client merkt sich zuletzt erhaltenen Cursor und verwendet ihn in nachfolgender Query, wenn nur paginated Node-Listen interessieren, verwende man nodes ohne edges, Simple Collection: ohne Edges/Nodes, wenn Cursor/Seek-based Pagination nicht interessiert

```
query q1 {
  allAngestellters(
    filter: { and: [
      { salaer: { greaterThan: "5000" } }
      { salaer: { lessThan: "8000" } } ]
      wohnort: { equalTo: "Luzern" } }
    orderBy: SALAER_ASC )
  { totalCount nodes { name salaer } } }
query q2 { allProjektsList {
    projnr bezeichnung
    angestellterByProjleiter { name } } }
mutation q3 { updateAngestellterByPersnr (
    input: {angestellterPatch: {bonus: "100"},
      persnr: 5000}) { clientMutationId } }
mutation q4 {   deleteAngestellterByPersnr (
    input: {persnr: 5000}) {
    clientMutationId deletedAngestellterId } }
```


```
{ "data": {
    "allAngestellters": { "totalCount": 6,
      "nodes": [
        { "name": "Pauli, Monika", "salaer": "5089.00" },
        { "name": "Schnell, Marie", "salaer": "5100.00" },
        { "name": "Danuser, Vreni", "salaer": "5100.00" },
        { "name": "Gschwind, Fritz", "salaer": "5900.00" },
        { "name": "Wehrli, Anton", "salaer": "5980.00" }]}}}
```


14 Design Patterns

Bewährte Lösungsschablone für wiederkehrende Entwurfsprobleme, Kombi von Konzepten und Tools, Rezept mit Struktur "How it works, when to use it, examples", Unterschied zu Framework, Framework Architektur und Best Practice
Data Mapper: trennt Business Logic und Datenzugriff strikt, bildet Entität in DB-Schema ab, Flexibilität bei DB-Änderungen, mehr Boilerplate-Code & Komplexität → mehr Entwicklungsaufwand, Implementationen: JPA, Hibernate, Nhibernate, SQLAlchemy, Entity Framework, jOOQ, Doctrine
Active Record: verbindet Business Logic (Klassen) direkt mit Datenzugriff (Entities/Tabellen), was zu einer direkteren und oft einfacheren Umsetzung führt, wenig Boilerplate-Code & einfach → wenig Entwicklungsaufwand, unflexibel bei komplexen Business Logics, schwierige Testbarkeit, Implementationen: jOOQ, SQLAlchemy, Ruby on Rails
CQRS: Command-Query Responsibility Segregation, Real-time analytics, gut für OLAP, schlecht für OLTP, verteilte Systeme, No-schema-Approach, trennt ein Domain Model in zwei separate Teile (Write / Command & Read / Query), R/W werden von untersch. Komponenten separat behandelt, so können beide oder eine davon optimiert werden, kann mit Views implementiert sein, komplex, aufwändige Koordination / Sync der beiden Modelle, passt zu GraphQL, Datawarehouses, Star Schema
Event Sourcing: Reihe von Änderungen im Zustand einer Anwendung, Events umfassen alles, was erforderlich ist, um den aktuellen Zustand zu restoren, Events sind immutable, append-only, unlöschbar (dafür Storno-Transaktionen), unabhängig von CQRS, aber oft in Kombination damit verwendet, schema-less, **Projection**: effizient abfragbare Repräsentation eines Events, View eines Zustands, durch Projector erzeugt, Snapshots können beschleunigen, **Vorteile**: Performance, History, Replication, Logging / Monitoring, kein Impendance Mismatch (OO→RM) dank No-schema, **Probleme**: keine Event-Validierung, keine IDE-Unterstützung, Schemaänderung schwer nachvollziehbar, Gefahr der fehlenden Separation von Business und Event, erzeugen der Projections kann zu Verzögerungen wurden und werden evtl. konsistent



Evolutionary Database Design: auch Database Change Management, Integration DB in CI/CD, agil, Design ist verwoben mit SW-Entwicklung/-Testing-/-Delivery, häufige Releases, DB-Design muss nicht zwingend vorab geschehen, sondern gleichzeitig, Schema Migration: strukturierte, kontrollierte Änderungen am DB-Schema, Refactoring: kleine Änderungen an der DB (Tabellenstruktur, Daten, SPs, Triggers), ohne das Verhalten zu ändern, Best Practices: Entwickler haben eigene DB-Instanz, integrieren kontinuierlich Änderungen und können DBs bei Bedarf aktualisieren, DB-Zugriffe im Applikationscode sind getrennt, alle DB-Artefakte sind im Repo mit dem Code, alle Änderungen sind Migrationen/Refactorings, Prozess ist automatisiert, Migrations-Tools: Alembic, LiquiBase, Flyway