1 OR-Mapper

- Eager Loading default bei OneToOne und ManyToOne
- Lazy Loading default bei OneToMany und ManyToMany
- EntityManager.persist(), EntityManager.remove() bei jedem Element, da nicht transitiv. Implizit bedeutet, dass em.remove() dann alle referenzierten Änderungen löscht. Kaskade bei Relation angeben, damit implizit: @OneToMany(cascade = CascadeType.PERSIST, CascadeType.Remove, ...)
- cascade mit Persist ist Aggregation und mit Remove dann sogar Komposition.
- Java Persistence Querying ist wie SQL aber auf Entity-Modell nicht Db Modell.
- JPQL Named/Positional-Query Parameters werden via Funktion gesetzt ohne SQL-Injektion. Dynamic Querys, können zu Injektion führen. Werden beide zur Laufzeit geprüft. @NamedQuery können sogar statisch geprüft und vom JPA-Provider voroptimiert werden.
- Criteria API kann zur Compile-Zeit geprüft werden. Ohne SQL Injection.

```
EntityManagerFactory factory =
 Persistence.createEntityManagerFactory("Bank");
EntityManager em = factory.createEntityManager();
Query query = em.createQuery("SELECT a from
 BankCustomer c join c.accounts a");
Query query = em.createQuery("SELECT distinct a.id,
 a.balance from BankAccount a order by a.balance desc");

Query query = em.createQuery("SELECT a from BankAccount
 a where a.balance >= 0 and a.balance <= :upper");
 query.setParameter("upper", 2000); List<BankAccount>
 list = query.getResultList();

Query query = em.createQuery("select m from BankManager
 m where 'Bob' in (select c.name from m.customers c)");
Query query = em.createQuery("select c from
 BankCustomer c where c.BankAccount.accountid =
 '0654321'");

Query query =
 em.createNamedQuery("FindPrivateBankCustomers
 OlderThanEqual");
query.setParameter(1, new
 Date(thirtyYearsAgo.getTimeInMillis()));
List<PrivateBankCustomer> customers =
 query.getResultList();

List<BankAccount> list = query.getResultList(); for
 (BankAccount account : list) {
 System.out.println(account); }

CriteriaBuilder criteriaBuilder =
 entityManager.getCriteriaBuilder();
// Query for a List of objects.
CriteriaQuery criteriaQuery =
 criteriaBuilder.createQuery();
Root employee = criteriaQuery.from(Employee.class);
criteriaQuery.where(criteriaBuilder.greaterThan(
 employee.get("salary"), 100000));
Query query = entityManager.createQuery(criteriaQuery);
List<Employee> result = query.getResultList();

em.getTransaction().begin();
BankCustomer customer = new BankCustomer();
 customer.setName("Bill"); em.persist(customer);
em.getTransaction().commit();

//update
BankCustomer customer = new BankCustomer();
 customer.setName("Bill"); em.persist(customer);

//delete
EntityManager em = factory.createEntityManager();
em.getTransaction().begin();
BankAccount account = em.find(BankAccount.class, 1L);
em.remove(account);
em.getTransaction().commit();
em.close();

--Gegen Dirty Reads und Lost Updates, PESSIMISTIC_READ
em.lock(from, LockModeType.PESSIMISTIC_WRITE);
em.lock(to, LockModeType.PESSIMISTIC_WRITE);
```

```
@Entity //sagt, dass man die Klasse speichern kann. Ist
 der Name, bei JPA-Query
@Table(name = "bankcustomer")
@NamedQuery(name="FindPrivateBankCustomersOlder
 ThanEqual", query="SELECT c FROM BankCustomer c WHERE
 c.birthdate <= ?1 ORDER BY c.name")
public class BankCustomer {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
 //entspricht dem serial auf der Db, sonst auto also
 globale Id
/* Alternative
Entsp. einer sep. Tabelle mit KeyName, KeyValue mit
 Namen KeyTable
@GeneratedValue(strategy = GenerationType.TABLE,
 generator ="CustomerGen")
@TableGenerator(name ="CustomerGen", table ="KeyTable",
 pkColumnName ="KeyName", valueColumnName ="KeyValue",
 pkColumnValue ="CustomerKey")

@GeneratedValue(strategy = GenerationType.SEQUENCE,
 generator = "BankCustGen")
@SequenceGenerator(name = "BankCustGen", sequenceName =
 "CustomerIdSeq",allocationSize=1) //die 2 Zeilen
 entsp. Create Sequence cutomeridseq; auf Db.
@Column(name="accountid") //auch unique=true,
 nullable=true, length=200 als parameter moeglich, name
 implizit propertyname
*/
private long customerId;

public void setCustomer(BankCustomer newCustomer) {
   BankCustomer oldCustomer = this.customer;
   this.customer = newCustomer;
   if (newCustomer != null &&
    !newCustomer.containsAccount(this)) {
     newCustomer.addAccount(this);
   }
   if (oldCustomer != null &&
    oldCustomer.containsAccount(this)) {
     oldCustomer.removeAccount(this);
   }}}

@Entity
@Table(name="PET_INFO")
public class Pet {

@Temporal(TemporalType.TIMESTAMP)
private Calendar birthdate;

@Transient
private String notSaved;

@Enumerated(EnumType.STRING)
private PetType type;

@ManyToMany(cascade = { CascadeType.PERSIST,
 CascadeType.MERGE,
   CascadeType.REMOVE, CascadeType.ALL
   },
   fetch = FetchType.LAZY
)
@JoinTable(name = "post_tag",
   joinColumns = {@JoinColumn(name = "post_id")},
   //JoinColumn "referencedColumnName" ist impl. PK
   //auch kommaseparierte mehrere möglich
   inverseJoinColumns = {@JoinColumn(name = "tag_id")}
)
private List<Tag> tags = new ArrayList<>();

//Seite 1:
@ManyToOne(optional = false, fetch = FetchType.EAGER)
@JoinColumn(name="OWNER_ID")
//optional nur für OneToOne und ManyToOne, da ansonsten
 List einfach leer.
Owner owner;

//Seite 2:
@OneToMany(mappedBy="owner")
private List<Pet> pets;

@OneToMany //Gegenseite dann @ManyToOne
 @JoinColumn(mappedBy="friends")
@JoinColumn(name = "addressid",
referencedColumnName = "addressref", //Wenn nicht PK
 von Pet
insertable = false, updatable = false)
private List<Pet> friends = new ArrayList<>();
```

1.1 Vererbung Single-Table, Joined-Table, Table-Per-Class

Ergibt eine Tabelle mit Spalte type und Inhalt aller Klassen

```
@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; }
```

Ergibt separate Tabellen für alle Klassen. Abstracte Klasse/Tabelle hat nur cutomerid, type, name.

```
//RetailBankCustomer dann auch customerId und fees
@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;}
```

Tabelle mit Namen wie konkrete Klasse inkl. Properties von abstrakter Klasse.

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract ...
```

2 Postgresql Functions

Programme 'gespeichert' (stored) bei oder nahe den Daten. Kapselt Domain Logik, Datenkapselung. Performance. Wiederverwendbarkeit, Feineres Berechtigungsmanagement, haben eigens Recht für Ausführung, Logging möglich. Nachteil Wartbarkeit, Portierbarkeit, ev. genügen Views.

Bei **PL/SQL**: Prozedurale Sprachelemente mit deklarativem SQL mischen. PL/SQL (Oracle) sind Keywords und Variablen case-insensitiv; nahe SQL/PLM

PL/pgSQL (postgres): case-sensitiv, nahe SQL/PLM, moderner. Code wird bei Aufruf geparsed, Pseudocode in Db gespeichert, erst bei Ausführung voller Syntax Check, SQL Statements werden vorkompiliert und wiederverwendet. Nur functions mit out modifier und void Datentyp. Schema anstelle packages.

```
create [or replace] function
name ( [ [ IN | OUT ] [argname] argtype [,...] ] )
[ RETURNS rettype
--kann z.B. void, int, bigint, text, bool, int[].
--setof int dann return next ...; und am Schluss return
 ;.
--wenn OUT vorhanden returns weg lassen.
| RETURNS TABLE ( colname coltype
[,...] )]
AS $$
-- ... Source Code gemaess language
$$ language [plpgsql|SQL|...]
```

```
CREATE OR REPLACE FUNCTION get_AbtMA (nr integer)
  RETURNS TABLE (
 abtname VARCHAR,
 abtMA VARCHAR
  )
AS $$
BEGIN
 RETURN QUERY
  SELECT abt.name, ang.name FROM abteilung abt
  join angestellter ang on ang.abtnr=abt.abtnr
  where abt.abtnr=nr
  order by abt.name, ang.name;
END; $$
LANGUAGE plpgsql;

select * from get_AbtMA(2);
```

```
create or replace function id_as_text (abc personen) --
 personen ist eine Tabelle und Zugriff via abc.Id usw.
```

```
DECLARE
 var1 int := 0; --nur dann var1 := var1 + 1 möglich!!!
 var2 int not null; var5 angestellter.id%type; -- Spalte
   Id
 var6 angestellter%rowtyp; var7 record;
BEGIN
 BEGIN
  SELECT .. INTO STRICT var1 FROM tbl --muss genau ein
   Tupel
 EXCEPTION
  WHEN division_by_zero THEN z:= 0;
  WHEN NO_DATA_FOUND THEN z := 0; --Bei SELECT INTO
   STRICT
  WHEN TOO_MANY_ROWS THEN z := 0; --Bei SELECT INTO
   STRICT
  WHEN UNIQUE_VIOLATION THEN z := 0; --Bei INSERT,
   UPDATE wegen PK Violation
  WHEN FOREIGN_KEY_VIOLATION THEN z:=0; --Bei INSERT,
   UPDATE wegen Foreign-Key Violation
 END; --Oder OTHERS. Falls Fehler sprung zu Exception
  und dann weiter nach End.
IF NOT EXISTS (SELECT 1 FROM angestellter WHERE persnr =
  anr) THEN
 RAISE EXCEPTION 'ang not exist';
END IF;

UPDATE Angestellter SET Salaer = Salaer + SalIncr WHERE
 Name = AngName;
IF NOT FOUND THEN --FOUND wird bei UPDATE, DELETE true/
 false wenn Tuppel bearbeitet
ELSE RAISE NOTICE 'name ist: %', name; --RAISE NOTICE
 macht ein Logeintrag, RAISE EXCEPTION bricht ab.
END IF;

INSERT INTO films (code, title, did, date_prod, kind)
 VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy')
    ;
END;
```

3 Funktionen

Erlauben komplexe Berechnung nur mit einem Aufruf. Separate Berechtigungen. Können vorkompiliert werden.

```
string || string/number -- String concat, oder array
 concat
upper(string), lower(string --upper/lower-case
substring(string [from int] [for int]) -- substring('
 Thomas' from 2 for 3) = hom
substr('alphabet', 3, 2) = ph
length(string) --Numbers of chars
now() --Date typ
mod(int, int) --Modulo
coalesce (SalaerSumme ,0) --NULL ersetzen
```

4 Loops/Condition

```
IF ( SalIncr < 0 OR SalIncr > 2000 OR X IS NULL) THEN
 ... END IF;
IF expression THEN statements ELSIF other-expression
 THEN statements ELSE statements END IF;
FOR i in 1..10 LOOP ... END LOOP;
FOR record-var IN query LOOP return next record-var; END
  LOOP; return; --record-var must be declared as record
 or %rowtyp return next adds result row and return;
 finish
--EXIT, CONTINUE möglich aber auch z.B. EXIT WHEN y >
 4000000;
FOR i IN 1 .. array_upper(a, 1) LOOP ... END LOOP; --
 Loop array a
WHILE i < 100 AND b > 1 LOOP ... END LOOP;
FOREACH i IN ARRAY a LOOP ... END LOOP;
```

5 Array

Array Index fängt immer bei 1 an. Mit 2:3 dann 2. + 3. Zeile.

```
unnest(anyarray) --Macht aus array row für loop
array_ndims(anyarray) oder array_dims(anyarray) --
 Dimension in int oder text
array_length(anyarray, int) --Länge von array, der
 dimension int
array_prepend(anyelement, anyarray) oder array_append(
 anyarray, anyelement)
array_to_string(anyarray, text [, text]) -- array,
 delimi, null string
new_arr := array_append(new_arr, 4) oder arr[i] := 1;

SELECT (array[1,2,3,4,5,6])[2:5] dann {2,3,4,5} --Slice
 Query, fängt bei 1 an.
```

```
INSERT INTO sal_emp VALUES ('Bill', ARRAY[10000, 10000],
  ARRAY[['meeting', 'lunch'],['train', 'present']]
INSERT INTO geometries VALUES (4, 'Line B', ARRAY[ARRAY
 [6,2],ARRAY[5,5]]); --ist dann {{6,2},{5,5}}
SELECT ARRAY(SELECT 1 + (random()*5)::int FROM
 generate_series(1,6) ORDER BY 1); --{1,3,5,5,6,6}

SELECT ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] --true
SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6]; -- true is
 contained by
SELECT ARRAY[1,4,3] && ARRAY[2,1] --true, @> bed.
 contains und && bed. overlap und = bed. equal
select * from tictactoe where 'z2 k2' = any(board); --
 Wenn ein Element vorhanden ist.
select ARRAY (select 1 + (random()*5)::int FROM
 generate_series(1,6));

SELECT * FROM planet_osm_point WHERE tags ? 'parking';
 --Alle Zeilen die den Schlüssel parking enthalten.

Geht nicht '{{"infsys", "dbs1"}, {"prog2", "dbs2", "vss
 "}}', weil nicht gleiche Dimension

select
ang.persnr,
min(ang.name),
array_agg(trim(proj.bezeichnung)) --Macht array
from angestellter ang
left outer join projektzuteilung pz on ang.persnr=pz.
 persnr
left outer join projekt proj on pz.projnr=proj.projnr
group by ang.persnr
order by ang.persnr;
```

6 Dictionary(hstore)

```
CREATE TABLE kvp_table(id serial, kvp hstore);
SELECT 'a=>1,a=>2'::hstore;
SELECT hstore(t) FROM test AS t; --Spalten sind jeweils
 key
SELECT akeys(mykvpfield) FROM ... --Alle Keys in einem
 array
SELECT each(mykvpfield) FROM ... --Alle Key-Value-Pairs
 als einzelne set
SELECT (each(h)).key, (each(h)).value INTO stat FROM
 testhstore;
SELECT mykvpfield->'name' FROM ... --Get Value of Key '
 name' as text
WHERE mykvpfield @> 'tourism=>zoo'; -- or hstore('
 tourism', 'zoo') - Vergleicht zwei hstore ob left is
 contained in right
hstore_to_array(tags)
```

7 JSON

JSON ist text in Db, also immer reparsen. JSONB ist gepartes Binärformat. Input lansamer, schneller Index.

```
'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2 ergibt
 {"c":"baz"} start bei 0, negativ von Ende. Also json
 object
'{"a":1,"b":2}'::json->>'b' ergibt 2 als text
'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb Does the
 left JSON value contain the right JSON path/value
 entries at the top level?
'{"a":1, "b":2}'::jsonb ? 'b' - Does the string exist as
  a top-level key within the JSON value?
'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}' Get JSON
 object at specified path = {"c": "foo"} oder mit #>> as
  text
to_json(anyelement) to_json('Fred said "Hi."'::text)
to_jsonb(anyelement) to_jsonb('Fred said "Hi."'::text)
row_to_json(record [, pretty_bool])
jsonb_each(jsonb) - select * from json_each('{"a":"foo",
  "b":"bar"}') ergibt spalte key, value

'{"a": {"b":"foo"}}'::json->'a' --gets json object by
 key 'a' = {"b":"foo"}
'{"a":1,"b":2}'::json->>'b' --gets text by key 'b' = 2
'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb --Does the
 left JSON value contain the right JSON path/value
 entries at the top level?
'{"a":1, "b":2}'::jsonb ? 'b' --true or false - check
 string exist as a top-level key within column?
'["a", "b"]'::jsonb || '["c", "d"]'::jsonb --concat two
 jsonb value
'{"a": "b"}'::jsonb - 'a' --Delete key/value pair or
 string
```

```
where angwithproj->>'persnr' = 1001::text;
where (angwithproj->>'name') like 'Marxer%'
where angwithproj->'projects' @> to_jsonb('Uranus'::text
 )

from angprojj, jsonb_array_elements_text(angwithproj->'
 projects');
cross join lateral jsonb_array_elements_text(angwithproj
 ->'projects');

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 proj on pz.projnr=proj.projnr
group by ang.persnr
order by ang.persnr;

ergibt {"name": "Marxer, Markus", "persnr": 1001, "
 projects": ["Mars", "Uranus", "Jupiter"]} wie unten

select jsonb_pretty(jsonb_agg(tmp))
from (
  select
    ang.persnr as persnr,
    min(ang.name) as name,
    jsonb_agg(trim(proj.bezeichnung)) as projects
    from angestellter ang
    left outer join projektzuteilung pz on ang.persnr=pz.
     persnr
    left outer join projekt proj on pz.projnr=proj.projnr
    group by ang.persnr
    order by ang.persnr
) tmp;

--Fügen Sie ein outer-level Tag „"carrier_hub mit dem
 Wert „"Swiss (in einem Array, damit weitere
 Fluggesellschaften hinzugefügt werden können.
UPDATE airports
SET airport = airport || '{"carrier_hub": ["Swiss"]}'
MUSTERLÖSUNG
WHERE airport ->> 'ident' IN (
'LSZH', -- Zürich Airport
'KMDW', -- Chicago Midway
'KLAS', -- Las Vegas
'KDAL'); -- Love Field, Dallas;

--Fügen Sie dem Flughafen Zürich Edelweiss zu
 carrier_hub hinzu ( jsonb_set ).
UPDATE airports
SET airport = JSONB_SET(
airport,
'{carrier_hub}',
(SELECT (airport -> 'carrier_hub') || TO_JSONB('
 Edelweiss'::TEXT)
FROM airports WHERE airport ->> 'ident' = 'LSZH'),
false)
WHERE airport ->> 'ident' = 'LSZH';

--ergibt ["Swiss", "Edelweiss"]

--entfernt ...wikipedia vom airport
UPDATE airports
SET airport = (airport - 'airport_wikipedia' - '
 region_wikipedia');
```

8 XML

Postgresql kann nur XPATH, Oracle, MS Server auch XQuery. XQuery ist XPATH + SQL-Like Syntax für joins, wie for, let, where, order by, return.

XNODE(XMLPARSE(DOCUMENT '<unit>value</unit>'));
XMLPARSE(DOCUMENT '<unit>value</unit>'));
XPATH('fn:name(/*)

9 Table

```
TRUNCATE table1; --löscht Inhalt von der Table
SET CONSTRAINTS ALL DEFERRED; führt dazu, dass die
 Constraints für die aktuelle Transaktion nicht geprüft
 werden.
```

10 Cursor

Cursor erlaubt den sequentiellen Zugriff auf die einzelnen Tuples des Result Set. Handle or name for a 'private SQL area' - an area in memory inside DB server.

```
DECLARE
AngRec RECORD;
curs1 REFCURSOR; -- unbound
curs2 CURSOR FOR select * from abteilung;
curs3 CURSOR (id integer) for select * from abteilung
 where abtnr = id;
AngCursor CURSOR FOR select * from b FOR UPDATE;
id int := 1;
rowvar abteilung%rowtyp;
name text;
BEGIN
-- open
open curs1 for execute 'select * from abteilung where
 abtnr = $1' using id;
OPEN curs1 FOR select * from angestellter ang where ang.
 AbtNr = Abteilungsnummer;
open curs2;
open curs3(2);

OPEN AngCursor; /*SQL-Abfrage starten und Resultat in
 Puffer speichern*/
LOOP /*Iteration ueber Resulatmenge*/
 FETCH AngCursor INTO AngSalaer, AngPersNr;
 --oder FETCH AngCursor INTO AngRec;
 EXIT WHEN NOT FOUND;
 SalSumme := SalSumme + AngSalaer;
 RAISE notice 'Angstellter PersNr: % Salaer %',
  AngPersNr, AngSalaer;
END LOOP;

OPEN AngCursor;
LOOP
 FETCH AngCursor INTO AngRec; --aktuelles Tupel wird
  gesperrt.
 EXIT WHEN NOT FOUND;
 UPDATE Angestellter SET Salaer = MinSalaer
 WHERE CURRENT OF AngCursor;
END LOOP;

CLOSE AngCursor;
CLOSE curs1; close curs2; close curs3;
END;
$$ language plpgsql;
```

11 Trigger

Für Implementation von komplexen Konsistenzbedingungen. Berechnung abgeleiteten Attributen, Logdaten und für Updateable Views.

INSTEAD OF trigger should either return NULL to indicate that it did not modify any data from the view's underlying base tables, or it should return the view row that was passed in (the NEW row for INSERT and UPDATE operations, or the OLD row for DELETE operations)

A row-level trigger fired before an operation has the following choices: it can return NULL to skip the operation for the current row. For row-level INSERT and UPDATE triggers only, the returned row becomes the row that will be inserted or will replace the row being updated.

The return value is ignored for row-level triggers fired after an operation, and so they can return NULL.

For a row-level trigger, the input data also includes the NEW row for INSERT and UPDATE triggers, and/or the OLD row for UPDATE and DELETE triggers.

```
CREATE TRIGGER mytrigger
{BEFORE | AFTER | INSTEAD OF} {event [OR ...]} --event=
 INSERT,UPDATE,DELETE, INSTEAD OF nur für Views
FOR EACH {ROW | STATEMENT}
EXECUTE PROCEDURE mytriggerfun();
```

```
CREATE OR REPLACE FUNCTION dt_trigger_func()
RETURNS TRIGGER AS $$
BEGIN
 --Bei Update oder Delete auch OLD verfuegbar.
 IF (TG_OP = 'INSERT') THEN
  NEW.creation_date := now();
 ELSIF (TG_OP = 'UPDATE') THEN
  NEW.modification_date := now();
 END IF;
 --INSERT INTO ang_audit SELECT 'U', now(), user, NEW.
  name, NEW.salaer;
 coalesce (SalaerSumme ,0) + v_salaer --wegen NULL + 1 =
   NULL
 RETURN NEW; --wenn return null dann nichts eingefügt.
  Bei delete old zurück.
END
$$ LANGUAGE plpgsql;
```

12 Updateable View

Eine View ist automatisch aktualisierbar (updatable), wenn...:

- Die View hat genau einen Eintrag in der FROM-Klausel, der eine Tabelle oder eine andere updatable View sein muss.
  - Die View darf keine WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET enthalten.
  - Die View darf kein UNION, INTERSECT, EXCEPT enthalten.
  - Die SELECT-Liste der View darf keine Aggregation, Window-Funktion oder SET-returning-Funktion enthalten.
- Eine Kolonne ist aktualisierbar, wenn sie eine einfache Referenz auf eine aktualisierbare Kolonne der darunterliegenden Relation ist.
- Eine autom. aktualiserbare View kann ein Mix von aktualiserbaren und nicht-aktualisierbaren Kolonnen enthalten.

```
CREATE OR REPLACE VIEW abtleiterinfo (
abtnr, abtname, abtchef)
AS
SELECT abt.abtnr, abt.name, al.abtchef
FROM abteilung abt
INNER JOIN abtleitung al
ON abt.abtnr=al.abtnr
INNER JOIN angestellter ang
ON ang.persnr=al.abtchef;
SELECT * FROM abtleiterinfo;

CREATE TRIGGER abtleiterinfo_update_abtchef
INSTEAD OF UPDATE ON abtleiterinfo
FOR EACH ROW
EXECUTE PROCEDURE abtleiterinfo_update_abtchef_fn();
```

13 Materialized Views

Werden zwischengespeichert. Wie normale View. Kann man mit Trigger, Cron-Job oder manuell refreshen.

```
CREATE MATERIALIZED VIEW aam AS SELECT * FROM aa WHERE a
  <= 500000;
REFRESH MATERIALIZED VIEW aam; --weil nicht aktualisiert
 .
```

14 Temporäre Tabelle

Erzeugt eine Tabelle, die automatisch gelöscht wird am Ende der Transaktion oder der Session.

```
CREATE TEMPORARY TABLE tmp AS SELECT generate_series
 (1,100000) AS a;
```

15 Zugriffsrechte

View kann man horizontal und vertikal schützen. Prozeduren laufen mit Berechtigung des Erstellers.

```
GRANT SELECT ON Angestellter_V TO PUBLIC; --alle dürfen
GRANT EXEC ON SalaerErhoehung TO PersonalChef_R; --nur
 Gruppe PersonalChef
WHERE al.AbtChef = login --Oracle login = username
```

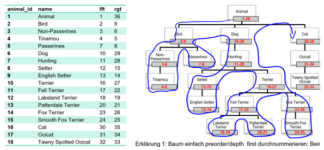
16 Graph und Tree

```
WITH cte_name(
    CTE_query_definition -- non-recursive term
    UNION [ALL] --all ist mit duplicates
    CTE_query definion -- recursive term
) SELECT * FROM cte_name;
```

```
WITH RECURSIVE
graph_cte (node1, node2) AS (
 SELECT node1, node2 from graph
 UNION ALL
 SELECT node2 AS "node1", node1 AS "node2" FROM graph
),
paths (node1,node2,path) AS (
 SELECT node1, node2, ARRAY[node1] AS "path"
 FROM graph_cte b1
 WHERE b1.node1 = 2 --<<< Start Node >>>
 UNION ALL
 SELECT b2.node1,b2.node2,p.path || b2.node1
 FROM graph_cte b2
 JOIN paths p ON (p.node2 = b2.node1 AND b2.node1 <> ALL
   (p.path[2:array_upper(p.path,1)]) /* Prevent
  retracing*/)
)
SELECT path || node2 AS "path"
FROM paths
WHERE node2 = 6 --<<< End node >>>
--AND ARRAY[2,3] <@ path -- <<< via... >>> - <@ means '
 is contained by'
ORDER BY array_length(path,1), path;
```

```
WITH RECURSIVE subordinates AS (
 SELECT employee_id, manager_id, full_name
 FROM employees
 WHERE employee_id = 2
 UNION
 SELECT e.employee_id, e.manager_id, e.full_name
 FROM employees e
 INNER JOIN subordinates s ON s.employee_id = e.
  manager_id
) SELECT * FROM subordinates;
```

3 Arten von Graphen. **Adjazenzliste** mit ParentId ist schnell mit insert, deletes und updates aber aufwändiger für Queries(Nachfolge, Vorgänger usw.). **Nestetd-Set-Modell** mit lft und rgt wobei lft erste Besuch und rgt zweiter Besuch ist. Aufwändig für insert, moves, deletes aber einfach für queries, da man einfach between 1 and 3 (= a >= x AND a <= y) machen kann. **Materialized Path** mit 1.1.1 oder Buchstaben. ltree @> ltree left ein Vorfahre von rechts oder gleich. left <@ right is left a descendant of right (or equal). Wobei jede ebene mit Punkt getrennt ist.



Notieren Sie das Resultat der folgenden Query unter der Annahme, dass die Tabelle „animals" die Struktur und den Inhalt der Aufgabe **Nested Set** enthält?

```
WITH p AS (
 SELECT lft, rgt FROM animals where name='Accipitrimorphae'
)
SELECT a.id, a.name
FROM animals a, p
WHERE a.lft BETWEEN p.lft AND p.rgt
ORDER BY a.lft;
```

id	Name
10	Accipitrimorphae
11	Greifvögel

17 Interne Ebene

Heap = Collection von Datapages (DP). DP enthält Rows in unsortierter Reihenfolge ca. 8kB gross. B-tree Knoten sind eine Page gross und enthalten Records. Baum ist selbstbalancierend und mind. 50% Auslasung mit log(n) Zugriffszeit. Ein Index wird erstellt mit: "create index customers_lastname_firstname on customers(lastname,firstname);"

Alle Blattknoten sind auf gleicher Höhe. Jeder Knoten hat mind. k max. 2k Einträge (Baum mit Grad k). Alle Knoten mit n Einträgen haben n+1 Kinder. Wenn Split dann mittleres Element eine Ebene höher schreiben und falls nicht geht dann da ein Split.

Dichte ist durchschnittlicher prozentualer Anteil von Duplikaten also $\frac{\text{Anzahl distinct}}{\text{Anzahl Tupel}}$

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

Query-Engine parst SQL-Anweisung und wandelt sie in Query-Tree. Dann allenfalls optimiert. Dann kommt Optimizer und wählt einen Execution Plan, basierend auf Statistiken über die Datenverteilung.

PG unterstützt keine clustered indexes.

17.1 Kosten-Modell

- P = Anzahl Data-Pages
- R = Anzahl Records per Page
- F = Fanout = durchschn. anzahl children in Nonleaf-Node
- Pl = Anzahl Pages im Leaf-Level des Index
- n = Anzahl Tupels, welche die Gleichheitsbedingung erfüllen.

Index:

- **Table/Heap Scan** = Scanning aller Pages einer Tabelle. Wird gemacht, wenn man mehr als 80% der Einträge braucht. Scan = P, equality-search = 1/2 P, range search = P
- **Clustered Index Scan**: scan = P, equality-search $Log_F(P) = Log(P)/Log(F)$ oder Anzahl Kinder ungf. Anzahl Einträge und so dann die Ebenen zählen = Anzahl Suchblöcke. Wenn Index Block F Einträge ermöglicht dann mit 2 IO schon F * F (*F für 3. Ebene) Einträge. Wenn Zahl das erste mal grösser als P dann korrekte IO. range-selektion = $log_F(P) + aufrunden(n/R)$ wobei R = Anzahl Records pro Page ist und F der Fanout
- **unclustered index**: scan = P * R/F+P*R, equality search = $Log_F(P*R)$, range search = $log_F(P*R)+n+\frac{n}{F}$ wobei n die erwarteten gleichen Records sind. Wenn Suche mehr als 10% Ausbeute, dann Table-Scan mit ev. Sort besser.

Join:

- **Nested Loop Join** ist Anzahl Records von A * Blöcke von B + Blöcke von A
- **Block Nested Loop Join**: P(R) + P(S) * $\frac{P(R)}{B-2}$ wobei P(R) Anzahl Pages (logisch optimiert die kleinere P(R)) ist und B Anzahl Hauptspeicherblöcke (Input-Output-Page 2).
- **Indexed Nested Loop Join**: P(R) + c * P(S) * N(R) wobei N(R) Anzahl Tupel von R und c Kosten für Indextraversierung
- **Hashjoin**: 3*(P(R)+P(S)) und wenn Inmemory Platz dann P(R) + P(S)
- **Merge-Join**: ExternalSort(R) + ExternalSort(S) + P(R)+ P(S). Sort ist 0, wenn Index besteht.

Beim clustered index kann man bei den Blättern wie bei einer linked list hüpfen. Bei einem Hash-Join hilft der Index auf Join-Condition nicht, da ganze kleinere Relation in Hashmap geladen werden muss, jedoch unabh. Spalten z.B. in Where hilft er. Hashjoin kann nur equi-joins.

Falls kein Index, Hash-Join in der Regel. Falls Index: Index-Nested-Loop effizienter als Hash-Join, falls Anzahl distincter Werte vom Index in etwa gleich der Anzahl Rows von S ist. Ist häufig, wegen Joins auf Foreign-Keys. Wenn Covering-Index, dann Index-Nested-Loop join schneller. Clustered index scan auch schneller, gibt es aber in postgresql nicht.

Index auf kleine Tabellen (< 200 Tupels) werden in der Regel nicht verwendet, wenn die Tabelle in eine Page passt (Index wäre dann +1 IO).

Logische Optimierung ist algebraische Umformung, sodass mehr Tupel eleminiert werden vor dem join zum Beispiel. kostenbasierte Optimierung basiert auf Statistiken, um den optimalen Ausführungsplan zu finden.

Statistik beinhaltet für jedes Attribut Anzahl Tuples, Anzahl unterschiedlicher Werte, Min und Max und Histogramm.

- Seq Scan: scans the entire relation (table) as stored on disk
- Index Scan: performs a B-tree traversal, walks through the leaf nodes to find all matching entries, and fetches the corresponding table data.
- Index Only Scan: like Index Scan but no table access because index has all columns to satisfy the query
- Bitmap Index Scan / Bitmap Heap Scan / Recheck Cond
- Sort
- HashAggregate: baut In-Memory Hashtable zum Aggregieren von Tupels
- GroupAggregate: aggregiert basierend auf vorsortierten Tuples

17.2 Histogram

- von 1 bis 10 jeweils 10 %, die andere Spalte zeigt den Max-Key. Wenn key drin, dann nehmen.
- Abschätzung Selektivität in % ohne Histogramm bei Wertebereich 0-353 und x < 70 dann = 70/(353 - 0) (* Anzahl Tupel, falls Resultat gewünscht)
- Bei x > 200 dann (353-200)/353 * Anzahl Tupels
- 237 distincte Werte. Wenn x = 210: Anzahl Tupels der Tabelle / 237 (Anzahl untersch. Werte).

18 Verteilte DBMS

- Fragmentierung horizontal = Records sind verteilt in versch. Knoten.

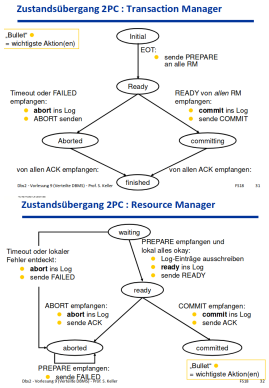
- Fragmentierung vertikal = Spalten sind verteilt in versch. Knoten.
- Relation ist die ganze Sicht.
- Sharding führt zu einer besseren Skalierbarkeit durch Aufteilung der Records auf ein oder mehrere Nodes (Shard). Wobei die Aufteilung der Records gleichbedeutend zu horizontaler Fragmentierung ist.
- Im **Homogene verteiltes DBMS** haben allen Knoten identische Software. Ist für User wie ein System. Bei **Heterogenen Systemen** können untersch. Schema oder Software aufweisen und haben dann Probleme mit Ausführung verteilter Queries (anderes Schema o.ä.) oder Transaktion(wegen untersch. Software).

Nachteile von Replikation sind die höheren Updatekosten und die komplexe Synchronisation, dafür aber schnellere Query oder Daten direkt vor Ort und höhere Verfügbarkeit.

Transparenz: Der Benutzer sieht eine globale Sicht, er kennt die Replikation der Fragmente nicht Queries werden auf der Relation definiert, nicht auf den Fragmenten. **2. wichtige Eigenschaft** auch Atomarität von verteilten Transaktionen.

18.1 Zwei-Phasen-Commit

- 1. Prepare to commit: TM fragt alle RM, ob bereit für Commit, falls gehen alle RM auf Zustand prepared und antworten mit READY.
- 2. Commit. RM antworten mit ACK.



Der TM schreibt Start-Transaktion und End-Transaktion ins Log sowie auch die Antworten aller RMs. Wenn TM nach Recovery commit oder abort findet, dann abort. Ansonsten dann abort senden. Bei RM wenn im Ready dann warten (oder Komm. mit anderen RMs) und abort nach Timeout schicken. Wenn RM in abort oder commit dann undo bzw. redo.

2PC ist bewährt aber skaliert eher schlecht, da blockierend. Serialisierung von Transaktionen muss global geschehen, da sonst allenfalls deadlocks.

Query Processing Strategien "Ship-Whole" macht eine grosse message und joint es dann auf einem Node. Bei Fetch-as-needed werden messages pro Tupel geschickt also viele kleine messages.

19 NoSQL

Oft nicht-Relational, schema-frei, einfaches API. Für grosse Datenvolumen: Scale out mit Daten-Replikation und Partitonierung (Sharding), oft als Cloud-Storage.

- Key/Value Stores - systems basically support get, put, and delete operations based on a primary key. No use if relationship among data. Good for session, shopping-cart.
- Document Stores - systems store structured "documentsßuch as JSON or XML but have no joins (joins must be handled within your application). It's very easy to map data from object-oriented software to these systems.
- Graph Stores
- Column-Family Stores - systems still use tables but have no joins (joins must be handled within your application). Obviously, they store data by column as opposed to traditional row-oriented databases. This makes aggregations much easier. Select wie bei normalen Datenbanken.
- Object oriented DBMS
- In-Memory Column Datenbank **MonetDB** - A database that keeps the whole dataset in RAM, Columns are compressed, each column is mapped onto a file, transaction loggin for persistence or snapshot. Select wie bei normalen Datenbanken. Ist eher für Datawarehouse gemacht, wo nicht viel geschrieben wird.

Es gibt die Replikation Master-Slave (Mongo-Db, asynchron oder synchron). Master schickt updates zu read-only slaves. Wenn synchron, dann blockierend via 2PC, MongoDb asynchron, wegen Majority. Wenn asynchron können Update-, Delete-, Ordering-,Uniqueness-Conflikte entstehen. Master-Master (**Cassandra**, Peer to Peer mit Conflict Resolution last write wins).

Strong consistency after an update is committed, each subsequent access will return the updated value. Weak consistency: the systems does not guarantee that subsequent accesses will return the updated value. Eventual Consistency umfasst Read your writes, Monotonic read, monotonic write.

W+R > N then consistency garantiert. Majority wenn w = N = r. Wenn w = (N+1)/2, dann Partitiontolerant.

Pessimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit. If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks.

Was ist eine In-Memory DB und wie passt MonetDB dazu? An in-memory database is a database that keeps the whole dataset in RAM. It means that each time you query a database or update data in a database, you only access the main memory. So, there's no disk involved into these operations. A good example of such a database is Memcached.

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 Speicherplatz freigeben (virtuelle Speicherverwaltung). In-memory Datenbanken speichern in der Regel die Daten nicht sofort zur Disk. Entweder über Logging (Live-Betrieb) oder über Snapshot (Datenbank angehalten).

19.1 CAP and BASE

Classic distributed system transactions: focused on ACID semantics

- Atomicity - alles oder nichts auf allen Replicas.
- Consistency - after each operation all replicas reach the same state.
- Isolation - no operation can see the data from another operation in an intermediate state.
- Durability - once a write has been successful, that write will persist indefinitely.

Modern Internet system: focused on BASE (Basically Available, Soft-state (or scalable), Eventually consistent

Es ist nicht möglich, dass man consistent (all clients same view of data), availability(every request to a non-failing node must result in a response) und partition-tolerance (No set of failures less than total network failure is allowed to cause invalid response) zur gleichen Zeit erreicht.

Consistent, Available (CA) Systems have trouble with partitions and typically deal with it with replication. Examples of CA systems include: Traditional RDBMSs like Postgres, MySQL, etc (relational)

Consistent, Partition-Tolerant (CP) Systems have trouble with availability while keeping data consistent across partitioned nodes. Examples of CP systems include: MongoDB (document-oriented), Redis (key-value), MemcacheDB (key-value), Neo4J (graph)

Available, Partition-Tolerant (AP) Systems achieve éventual consistency"through replication and verification. Examples of AP systems include: Cassandra (column-oriented/tabular), CouchDB (document-oriented)

Eventual consistency ist etwa ein Gegenteil von strong consistency. Dort jeder nachfolgende access wird den updated value liefern. Eventual Consistency kann man berechnen wann die Daten etwa wieder konsistent sein sollten wegen Netzwerkdelay, systemload oder Anzahl Replicas z.B. bei DNS möglich.

19.2 Neo4j

- MATCH (n:Person)-[:KNOWS]->(m:Person) WHERE n.name = 'Alice'
- MATCH (multitalent:Person)-[a:ACTED_IN]->(m:Movie) WHERE (multitalent)- [:DIRECTED]->(m) RETURN a.roles
- MATCH (audrey:Person {name: Äudrey Tautou"})-[a:ACTED_IN]->(m:Movie) WHERE (audrey)-[:DIRECTED]->(m) RETURN a.roles
- CREATE (n)-[r:KNOWS]->(m) : Create a relationship with the given type and direction; bind a variable to it.
- DELETE n, r : Delete a node and a relationship.
- DETACH DELETE n : Delete a node and all relationships connected to it.
- MATCH p=shortestPath((bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person{name:"Meg Ryan"})) RETURN p
- MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood) RETURN DISTINCT hollywood
  mit 1 bis 4 Hüpfern.

RETURN * Return the value of all variables.

[MATCH WHERE] [OPTIONAL MATCH WHERE] [WITH [ORDER BY] [SKIP] [LIMIT]] (CREATE [UNIQUE] | MERGE)* [SET|DELETE|REMOVE|FOREACH]* [RETURN [ORDER BY] [SKIP] [LIMIT]]

19.3 Mongo-DB

- Table => Collection
- Row => Document
- rowid => _id
- Join => DBRef

Safes data in json or bson. Works with MapReduce und aggregates.

Kennt Range- und Hash-based Sharding, also horizontale Partitionierung.

readConcern: level: <"majority"|"local"|"linearizable">

local is default, instance most recent data. Linearizable waits until all writes finished, that started before query.

Der Client führt einige Schreiboperationen aus. A kommt zu einem späteren Zeitpunkt wieder dazu (d.h alle Knoten A, B und C sind wieder im Netzwerk)? Wie reagiert das System? A wird als Secondary wieder in den Verbund von B und C aufgenommen. A aktualisiert sein Oplog und führt die Operationen nach.

Wie können Sie das Majority-Protokoll in MongoDb umsetzten? Durch setzen des WriteConcerns (standard 1, "majority"möglich), so dass alle Schreiboperationen von einer Majorität der Knoten bestätigt werden müssen (sonst wird die Schreiboperation nicht ausgeführt)

Welcher Mechanismus verwendet MongoDb für das Synchronisieren von Replikas. Ist dieser Mechanismus asynchron oder synchron? Oplog keeps an ordered list of write operations that have occurred. MongoDB applies database operations on the primary and then records the operations on the primary's oplog. The secondary members then copy and apply these operations in an asynchronous process.

```
var class = {
_id: ObjectId ("509980df3"),
course: {code: "Dbs2", title: "Advanced DB"}, –Subdokument
students: ["Peter", "Manuel",...], –Array of Strings
}

var x = ObjectId()
```

MongoDb-Shell: Wenn man den Punkt-Operator verwendet, muss der key dann auch in "". Also zum Beispiel "Blogpost.stats.visitors" : {$gte: 3}

- db.unicorns.insert({name: "Aurora", gender: "f", weight: 450, loves: ["apple", "grape"], birthday: new ISODate('2013-04-15')}
- db.unicorns.remove({})
  entfernt alle dokumente
- db.unicorns.find()
  zeigt alle an.
- db.employees.findOne({_id: db.employees.findOne({name:'Moneo'}).manager});
  Manueller Join
- db.employees.find({manager:{$in: [db.employees.findOne({name:'Leto'}, {_id:1})._id]},{_id:0})
  Manueller join db.employees.insert({_id: ObjectId("1a"), name: 'Duncan', manager: ObjectId("5c")});
  Find Manager of Duncan
- db.unicorns.find({gender:"f",loves:"apple"})
  für einfache bedingung (loves kann array sein).
- db.unicorns.find({gender:"f", loves:{**$in**:["apple", "carrot"]}})
  gibt alles zurück wenn im array apple or carrot ist.
- db.unicorns.find({gender:"f", loves:{ **$all**:["apple", "carrot"]}})
  Im Array muss Apple und Carrot sein.
- db.unicorns.find({$or:[{vampires: {$exists: false}}, {vampires: {$lte:0}}]})
  wenn property vampires nicht existiert oder less than equals 0 ist.
- db.unicorns.find({gender:"m", **$and**:[{weight: {**$gte**:600}}, {weight:{$lte:900}}]})
  wenn gewicht zwischen 600 und 900 ist.
- db.unicorns.find({gender:"f", **$or**:[{loves:"apple"}, {loves:"carrot"}]}, {**_id:0**, name:1, gender:1, dob:1})
  or anstelle von in und zusätzlich noch Einschränkung welche Spalten kommen mit :0 und :1
- db.unicorns.update({name: "Roooooodles"}, {$set: {weight: 590}}, {upsert: false})
  upsert true macht neues Dokument. Erste {} ist Query, dann Ersatzdokument. $set definiert update-Felder und $unset : {field:""} löscht das feld. Falls upsert dann Dokument aus Query + $set Feldern.
- db.unicorns.update({name: "Roooooodles"}, {$set: {a: 3, b: 10}});
  Fügt a und b hinzu.
- db.unicorns.update({name: "Aurora"}, {$push: {loves: "sugar"}, ...})
  push fügt array element hinzu.
- db.unicorns.update({name: "Pilot"}, {$inc: {vampires: -2}}, {upsert:false})
- db.unicorns.update({name:"Roooooodles"}, {$inc: {vampires: 90, weight: 10}});
- db.unicorns.update({gender:"f", $or:[{loves:"apple"}, {loves:"carrot"}]}, {$push: {loves:"tomato"}}, {upsert:false, multi:true})
  mit Multi werden mehrere Dokumente updated. Push fügt ein Array Element hinzu. Standard ist multi-false.
- db.scores.find({ results: { **$elemMatch**: { $gte: 80, $lt: 85 } } })
  Array Element muss alle Bedingungen erfüllen.

Für Operatoren gilt folgendes { <field1>: { <operator1>: <value1> }, ... } wobei es diese operatoren gibt:

- $eq Matches values that are equal to a specified value.
- $gt Matches values that are greater than a specified value. Mit e am schluss dann equals
- $in Matches any of the values specified in an array.
- $lt Matches values that are less than a specified value.
- $or Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
- { $set/$push: { <field1>: <value1>, ... } }
- $exists Matches documents that have the specified field.
- { <field>: { $elemMatch: { <query1>, <query2>, ... } } } The elemMatch operator matches documents that contain an array field with at least one element that matches all the specified query criteria.

Aggregate Funktionen

- $match, Matches documents equally funktionieren wie find. Subdokumentzugriff via .
- $project, Felder ausblenden.
- $lookup, macht ein Join, from ist die andere aggregation, lokalfield ein Feld der aufr. coll.
- $unwind, macht aus einem Array mehrere dokumente/rows

```
db.address.aggregate([
{$match: {street: "Blumenstrasse 13"}},
{$lookup : {from: "persons", localField: "_id", foreignField: "address", as: "persons"}},
{$project: {_id: 0, street: 1}}]).pretty();

db.orders.aggregate([
{ $match: { status: "A" } },
{ $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
{ $sort: { total: -1 } }
])

db.collection.aggregate( [
{ $group: { _id: null, myCount: { $sum: 1 } } },
{ $project: { _id: 0 } }
] )
```

$size gibt Grösse von array zurück.1

20 Evolutionary DB Design

1. Neue Spalte erstellen, 2. Migrationsskript schreiben. 3. Änderung der Applikation. 4. Änderungen aller DB-Zugriffe im App, View, SP und Trigger usw. 5. Ändern von Indices. 6. Übertragung aller Änderungen mit Versionsverwaltung.

Alle DB-Änderungen sind Migrationen (DB Migrations-Tools wie z.B. Flyway). Alle DB-Artefakte sind versions-kontrolliert mit Applikationscode. SW-Entwickler integrieren kontinuierlich DB-Änderungen. Eine DB besteht aus Schema und Daten DB-Refactorings sind automatisiert (DB-Migrations-Tools) Klare Trennung aller DB-Zugriffe im Applikationscode

21 GraphQL

Sämtliche benötigten Daten werden in einem "Roundtrip" geliefert. Die vom Client definierte Datenstruktur ist deklarativ und typisiert. der Client definiert die verlangten Daten und nicht der Server. Kein N+1 Problem. Nicht normiert.

Restful Resultat enthält z.T. unnötigerweise alle Objekte und de Objekte enthalten immer alle Felder. SQL Direkte Abfragen über HTTP(s) nicht gegeben

```
query alleAngestelltenNamen {
    allAngestellters {
        nodes {
            name
        }
    }
}

query alleAngestelltenMitInLuzern {
    allAngestellters(filter: {
        and: [{
            salaer: {greaterThan: "5000"}
        }, {
            salaer: {lessThan: "8000"}
        }],
        wohnort: {equalTo: "Luzern"}
    }, orderBy: SALAER_ASC)
    {
        nodes {
            name
            salaer
        }
    }
}
```

22 Column Family Store Cassandra

keyspace is like Database, column family like table. consistency setting = QUORUM: majority of the nodes are accessed and the column with the newest timestamp is returned. consistency setting = ALL: all nodes will have to respond to reads or writes. consistency setting = ONE (the default): the data from the first replica is returned even if it is stale. consistency setting = QUORUM: the write has to propagate to the majority of the nodes before it is considered successful

Consistent Hashing Algorithm for Partitioning. Gossip protocol is used for cluster membership discover node state for all nodes in a cluster (Heartbeat).

Scales linear, because you can add a node and shard consistent hashing some data in cassandra.

On Keyspace you can define replication factor.

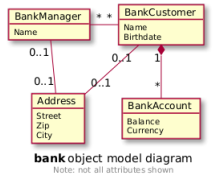
Atomicity at the row level. No traditional transaction.

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';

Not use for ACID, SUM, AVERAGE. Scales on Write so for Event Logging, CMS useful.

23 Übungen

23.1 Ü1




```
CREATE TABLE BankCustomer(
  CustomerId SERIAL NOT NULL PRIMARY KEY,
  Name TEXT NOT NULL,
  Birthdate DATE,
```

```
  Customer_AddressId INTEGER);

CREATE TABLE Address(
  AddressId SERIAL NOT NULL PRIMARY KEY,
  Street TEXT NOT NULL,
  Zip INTEGER,
  City TEXT NOT NULL);

CREATE TABLE BankAccount(
  AccountId SERIAL NOT NULL PRIMARY KEY,
  Account_CustomerId INTEGER NOT NULL,
  Balance DOUBLE PRECISION NOT NULL,
  Currency TEXT NOT NULL DEFAULT 'CHF'
);

CREATE TABLE BankManager(
  ManagerId SERIAL NOT NULL PRIMARY KEY,
  Name TEXT NOT NULL,
  Manager_AddressId INTEGER
);

CREATE TABLE CustomerManager(
  CustomerId INTEGER NOT NULL,
  ManagerId INTEGER NOT NULL,
  PRIMARY KEY(CustomerId, ManagerId)
);
```

```
public enum Currency {
 CHF, EUR, USD, JPY, GBP
}

@Entity
public class Address {
 @Id
 private long addressid;

 @OneToOne(mappedBy="address")
 private BankCustomer customer;

 @OneToOne(mappedBy="address")
 private BankManager manager;
}

@Entity
public class BankAccount {

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

 private double balance;

 @Enumerated(EnumType.STRING)
 private Currency currency;

 @OneToOne
 @JoinColumn(name="Account_CustomerId")
 private BankCustomer customer;
}

@Entity
public class BankCustomer {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private long customerid;

 private Date birthdate;

 @OneToOne(optional=true)
 @JoinColumn(name="Customer_AddressId")
 private Address address;

 @ManyToMany(mappedBy="customers", fetch=FetchType.EAGER
 )
 private Collection<BankManager> managers = new
  ArrayList<>();

 @OneToMany //name bezieht sich auf die Many Seite und
  referencedColumnName auf die eigene.
 @JoinColumn(name="Account_CustomerId",
  referencedColumnName="CustomerId")
 private Collection<BankAccount> accounts = new
  ArrayList<>();

 private String name;

 @Override
 public boolean equals(Object obj) {
  if (!(obj instanceof BankCustomer)) return false;
  return ((BankCustomer)obj).customerid == customerid;
 }
}

@Entity
public class BankManager {

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

 private String name;

 public long getManagerid() {
  return managerid;
 }

 @OneToOne(optional = true)
 @JoinColumn(name = "Manager_AddressId")
 private Address address;

 @ManyToMany
 @JoinTable(name = "CustomerManager", joinColumns = {
  @JoinColumn(name = "ManagerId") }, inverseJoinColumns
  = { @JoinColumn(name = "CustomerId") })
 private Collection<BankCustomer> customers = new
  ArrayList<>();

 public void addCustomer(BankCustomer customer) {
  this.customers.add(customer);
  if(!customer.getManagers().contains(this)) {
   customer.getManagers().add(this);
  }
 }

 public void removeCustomer(BankCustomer customer) {
  this.customers.remove(customer);
  if(customer.getManagers().contains(this)) {
   customer.getManagers().remove(this);
  }
 }
}

public Bank {
 public static void openAccount(String name, Date
  birthDate) {
   EntityManager em = factory.createEntityManager();
   try {
    em.getTransaction().begin();
    BankCustomer customer = new BankCustomer();
    customer.setName(name);
    customer.setBirthdate(birthDate);

    BankAccount account = new BankAccount();
    account.setBalance(0);
    account.setCurrency(Currency.CHF);
    account.setCustomer(customer);
    em.persist(account);

    customer.getAccounts().add(account);
    em.persist(customer);

    System.out.println("ACTION NEW: "+customer);

    em.getTransaction().commit();
   } catch (Exception e) {
    em.getTransaction().rollback();
    System.err.println("Failed to open account with
     message [" + e.getMessage() + "]");
   } finally {
    em.close();
   }
  }
 public static void transfer(long fromAccountId, long
  toAccountId, double amount) {
   EntityManager em = factory.createEntityManager();
   try {
    em.getTransaction().begin();
    BankAccount from = em.find(BankAccount.class,
     fromAccountId);
    BankAccount to = em.find(BankAccount.class,
     toAccountId);
    from.setBalance(from.getBalance() - amount);
    to.setBalance(to.getBalance() + amount);
    System.out.println("ACTION TRANSFER: " + from + " =>
      " + to + ", " + amount);
    em.getTransaction().commit();
   } catch(Exception e) {
    em.getTransaction().rollback();
    System.err.println("Failed to execute transfer with
     message [" + e.getMessage() + "]");
   } finally {
    em.close();
   }
  }
}
```

23.2 Ü3

```
CREATE OR REPLACE FUNCTION public.projektzuteilen(angnr
 integer, projektnr integer, arbzeit integer, startzeit
 date)
 RETURNS integer
 LANGUAGE plpgsql
AS $function$
DECLARE
       carbzeit int := 0;
BEGIN
       IF arbzeit < 10 OR arbzeit > 90 THEN
             return -1;
       END IF;

       IF startzeit IS NULL THEN
             startzeit := now();
       END IF;

       select SUM(zeitanteil) INTO STRICT carbzeit from
        projektzuteilung where persnr = angnr;
       IF carbzeit + arbzeit > 100 THEN
             RETURN -2;
       END IF;

       IF EXISTS(SELECT 1 FROM projektzuteilung where
        persnr = angnr AND projnr = projektnr) THEN
             RETURN -3;
       END IF;

       IF NOT EXISTS(SELECT 1 FROM angestellter where
        angnr = persnr) OR NOT EXISTS
         (SELECT 1 FROM projekt where projektnr = projnr
         ) THEN
             RETURN -5;
       END IF;
       BEGIN
             INSERT INTO PROJEKTZUTEILUNG (persnr,
              projnr, zeitanteil, startzeit)
                   VALUES (angnr, projektnr, arbzeit,
                    startzeit);

       EXCEPTION
             WHEN UNIQUE_VIOLATION THEN RETURN -6;
             WHEN FOREIGN_KEY_VIOLATION THEN RETURN -7;
       END;

       RETURN 0;
END;
$function$

CREATE OR REPLACE FUNCTION getAllFoo()
RETURNS SETOF foo AS $$
DECLARE
r foo%rowtype;
BEGIN
FOR r IN SELECT * FROM foo WHERE fooid > 0
LOOP
-- do something...
RETURN NEXT r; -- return current row of SELECT
END LOOP;
RETURN;
END
$$ LANGUAGE 'plpgsql';
```

23.3 Ü4

```
DO LANGUAGE plpgsql $$
DECLARE
c1 CURSOR IS
SELECT name, persnr, salaer FROM angestellter
ORDER BY salaer DESC;
-- start with highest-paid angestellter
my_name CHAR(20);
my_persnr NUMERIC;
my_salaer NUMERIC(7,2);
BEGIN
OPEN c1;
truncate Top5;
FOR i IN 1..5 LOOP
FETCH c1 INTO my_name, my_persnr, my_salaer;
EXIT WHEN NOT FOUND;
/* in case the number requested is more than the total
 number of employees
*/
INSERT INTO top5 VALUES (my_name, my_persnr, my_salaer);
END LOOP;
CLOSE c1;
END;
$$
```

```
CREATE SCHEMA angpackage AUTHORIZATION anguser;
CREATE FUNCTION angpackage.AbteilungsAngestellte(
 Abteilungsnummer IN INTEGER)
```

```
DROP FUNCTION IF EXISTS DepartmentSalaries();
DROP TYPE IF EXISTS HOLDER;
CREATE TYPE HOLDER as (abtnr INT, totalsalary NUMERIC
 (7,2));
CREATE OR REPLACE FUNCTION DepartmentSalaries()
RETURNS SETOF holder AS $$
DECLARE
 r holder%rowtype;
BEGIN
 FOR r IN SELECT abtnr, SUM(salaer) AS totalsalary FROM
  GetEmployees() GROUP BY abtnr
 LOOP
  RETURN NEXT r;
 END LOOP;
RETURN;
END;
$$ LANGUAGE 'plpgsql';

SELECT * FROM DepartmentSalaries();
```

24 Begriffe

Polyglot Persistence is a fancy term to mean that when storing data, it is best to use multiple data storage technologies, chosen based upon the way data is being used by individual applications or components of a single application.

Fetch-As-Needed pro Join-Row eine Nachricht mit den Attributen. **Fetch-Whole** ganze Tabelle abz. where.

Funktionen überladen: Mit SET search_path = angpackage, "$user", public; kann Funktionen überladen. Und so das: CREATE SCHEMA angpackage AUTHORIZATION anguser; CREATE FUNCTION angpackage.AbteilungsAngestellte(Abteilungsnummer IN INTEGER)

Heap: filestructure, lists of unordered records <> In-Memory heap!!! Retrieval inefficient as searching is linear

B+-Index: Diskbasierter B-Baum über ein Attribut bzw. Attributkombination. Jeder Knoten enthält nur die At- tributwerte. Blatt-Knoten enthalten die Disk-Referenz auf das gespeicherte Tupel.

Clustered B+-Index: Wie B+-Index aber Blatt-Knoten enthalten die Tupel. D.h. die Tupel sind sortiert nach dem Attribut des Index. Achtung, max. 1 Clustered Index pro Tabelle!

B+ Index basierend auf einem Clustered Index: Da es nur einen Clustered Index proTabelle geben kann, werden weitere B+-Indexe auf anderen Attributen definiert. Im Gegensatz zu den B+-Indizes sind in den Blattknoten die Werte des Clustered Index gespeichert bedingt einen zusätzlichen Zugriff auf den Clustered Index.

Bit-Map-Index Index Update ist teuer. Schnell für OR Abfragen. Für kleinen Wertebereich.

Replizierte Verteilung der Daten=>höhere Verfügbarkeit (geographische Verteilung, bessere Skalierbarkeit, Anwendungsfälle: Loadbalancing, Realtime OLAP, geograph. Verteilte System mit lokaler Teilautonomie

SARG-able Queries: Jede Query wird zuerst analysiert um Suchargumente zu fnden -> Nur Suchargumente können Indexe benutzen, Ein Suchargument ist entweder ein exakt Match oder eine Range, Suchargumente können Listen sein, die mit AND verknüpft sind, Eine Seite des Vergleiches ist Konstante oder aufösbare Variable, Eine Seite des Vergleichs ist Kolonnenname.

Vertikale Fragmentierung: Spalten werden verteilt. Column-family-Store wie Cassandra.

Scaling out: Clustering mit Standard-HW

Scaling up: Schnellerer Server

Replikationsarten: Synchron (Eager) vs. Asynchron (Lazy, mit Versionsnummern)

Quorum Consensus: Wie Majority, aber mehrere Votes pro Node möglich

Replika Sets: Normal 3 Nodes pro Set, 1 Primary Node wird gewählt, W nur an Primary/immer nur 1 Primary, Daten werden nach W repliziert, Load-Balancing für R

25 Prüfungsaufgaben