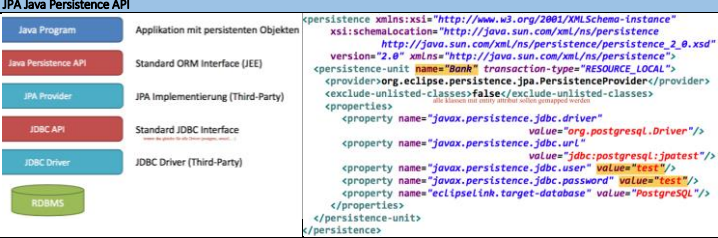


Dbs2 ACID: Atomicity, Consistency, Isolation, Durability



**Entitäten**

```
@Entity @Inheritance(strategy = InheritanceType.JOINED /*zusätzl Tabelle für spez felder*/ )
    SINGLE_TABLE /*(1 Tabelle, NULL-Werte)*/ | TABLE_PER_CLASS

@DiscriminatorColumn(name = "type")
public class BankCustomer {
    @OneToOne(optional=true) @JoinColumn(name="Customer_AddressId") private Address address;
    @ManyToMany(mappedBy="customers", fetch=FetchType.EAGER) private Collection<BankManager> managers = new ArrayList<>();
    @OneToOne @JoinColumn(name="Account_CustomerId", referencedColumnName="CustomerId")
    private Collection<BankAccount> accounts = new ArrayList<>();
}
```

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

@Entity
public class Address {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private long addressId;
    @OneToOne(mappedBy="address") private BankCustomer customer;
}
```

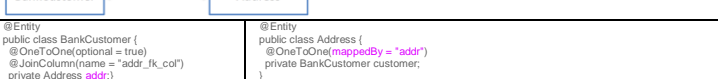
```
@Entity public class BankManager {
    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(name = "CustomerManager", joinColumns = {
        @JoinColumn(name = "ManagerId"), inverseJoinColumns = {
            @JoinColumn(name = "CustomerId")
        }
    })
    private Collection<BankCustomer> customers = new ArrayList<>();
}
```

```
@Entity public class BankAccount {
    @Enumerated(EnumType.STRING) private Currency currency;
    @OneToOne @JoinColumn(name="Account_CustomerId") private BankCustomer customer;
}
```

**Lazy Loading** @OneToOne(fetch = FetchType.LAZY) Default bei OneToOne und ManyToOne  
**Eager Loading** @ManyToMany(fetch = FetchType.EAGER) Default bei OneToMany und ManyToMany

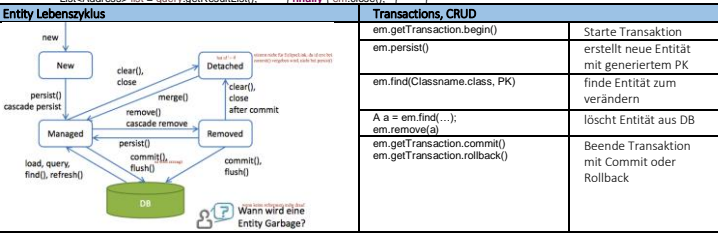
**Enumeration** @Enumerated(EnumType.STRING) private Currency currency; Primary Key ID @Id @GeneratedValue(strategy = GenerationType.IDENTITY)

**Dates** @Temporal(TemporalType.TIMESTAMP) private Calendar date; Custom Column @Column(name = "description") private String explanation;



**Queries** // Query Parameter verhindern SQL Injection

```
public class TheBank {
    private static EntityManagerFactory factory;
    factory = Persistence.createEntityManagerFactory("Bank");
    EntityManager em = factory.createEntityManager(); try {
        Query query = em.createQuery("select a from Address a where aid = ?1 and a.street like :street");
        query.setParameter(1,12); query.setParameter("street", "Hausptst");
        List<Address> list = query.getResultList(); // finally (em.close());
    }
```



**Transactions, CRUD**

```
em.getTransaction.begin() // Starte Transaktion
em.persist(a) // erstellt neue Entität mit generiertem PK
em.find(Classname.class, PK) // finde Entität zum verändern
A a = em.find(...); // löscht Entität aus DB
em.remove(a)
em.getTransaction.commit() // Beende Transaktion mit Commit oder Rollback
em.getTransaction.rollback()
```

**Stored Procedures** IF ... THEN ...; ELSEIF ... THEN ...; ELSE ...; END IF; FOR ... IN ... LOOP ...; END LOOP;

User-definierte Subroutine die auf DB läuft und nahe der Daten gestored ist. Erlauben SELECT, INSERT, DELETE, UPDATE innerhalb Funktion. Statements vorkompliert und wiederverwendet. Code wird beim Aufruf geparkt + als Pseudocode gespeichert, erst bei Ausführung voll Syntaxcheck. SP machen nur Sinn, wenn auch DB-Zugriff stattfindet.

**Vorteile** - Performance - Code Wiederverwendbarkeit - Security Massnahmen (durch Zugriffsschutz (GRANT)) - Log Auditing

**Nachteile** - Entwicklungsprozess komplexer - an Hersteller gebunden (2T größere Unterschiede)

**Vergleichen mit Funktionen** - SP nicht in Expression verwendbar - kein Wert zurückgeben - kann values als out param zurückgeben - mehrere Resultsets zurückgeben möglich

PL/pgSQL (PostgreSQL) Syntax:

```
create (or replace) function
name ([argname] anytype [...])
RETURNS rettype | RETURNS TABLE ( colname coltype [...])
AS $$
[DECLARE]
var1 int;
var2 int;
RETURN QUERY
SELECT abt.name, arg name FROM abteilung abt join
angestellter ang on ang.abtnr=abt.abtnr where abt.abtnr=
end; $$ LANGUAGE plpgsql;
select * from get_AbtM2(x);
```

**Triggers (Nur Row Trigger unterstützen NEW und OLD)**

Werden meist zur Wahrung der Datenintegrität genutzt. Sind normale DB Objekte, die immer **einer Tabelle** zugeordnet werden. Werden in stored procedures programmiert. Haben keine Parameter. Werden automatisch beim Eintreten bestimmter Events ausgeführt. Triggers können wiederum andere Triggers ansossen. Trigger machen DB tendenziell langsamer und schwerer wartbar

**Events:** INSERT, UPDATE [OF <columnname>], DELETE

**Ausführung:** BEFORE, AFTER, INSTEAD OF Anzahl: ROW (für jeden Tupel) oder STATEMENT (1 Mal)

**Datenzugriff:** NEW (für INSERT und UPDATE), OLD (für UPDATE, DELETE)

**Returnwerte:** BEFORE Trigger: INSERT/UPDATE → RETURN NEW, DELETE → RETURN OLD; (RETURN NULL → Abbruch der Op)

**Auführungsreihenfolge:** 1. alle before statement triggers (alphabetische Reihenfolge) 2. für jedes betroffene tupel 2a. before row trigger 2b. tupel bearbeiten 2c. alle after row trigger 3. alle after statement trigger

Trigger Variablen		hängt Strings zusammen		Ausgabe: RAISE NOTICE '%', <varname>	
TG_NAME	Name des Triggers (TG)	TG_WHEN	BEFORE oder AFTER		
TG_LEVEL	ROW od. STATEMENT	TG_OP	INSERT, UPDATE, DELETE		
TG_RELID	OID der Tabelle	TG_TABLE_NAME	Name der Tabelle		
TG_TABLE_SCHEMA	Schema der Tabelle				
user	aktuell eingeloggter User	now()	aktuelle Zeit		

**Trigger Beispiel**

```
CREATE TRIGGER <triggername>
AFTER <events> OF <cols> ON <tablename>
DECLARE
FOR EACH [ROW|STATEMENT]
EXECUTE PROCEDURE <triggerfuncn>;
DROP TRIGGER <triggername> ON <tablename>;

CREATE TRIGGER trigger_projektauslastung
BEFORE INSERT ON Projektzeile
FOR EACH ROW
EXECUTE PROCEDURE func_projektauslastung();

CREATE OR REPLACE FUNCTION func_projektauslastung()
RETURNS TRIGGER AS $$
BEGIN
IF (SELECT SUM(zeitanteil) FROM Projektzeile WHERE persnr = NEW.persnr)
> 100 - NEW.zeitanteil THEN
RAISE EXCEPTION 'Zeitanteil über 100';
END IF; RETURN NEW; END; $$ LANGUAGE plpgsql;
```

**Cursors** (Ablauf: 1.Declare → 2.Open → 3.Found? → 4.Fetch (Back to 3) → 5.Close)

Anstatt eine Query auf einmal auszuführen, kann man mit einem Cursor einzelne Tupel des Result Sets sequentiell abarbeiten. Somit verhindert man einen Memory Overrun bei vielen Resultaten (wird im Hintergrund so oder so gemacht). Ein Cursor ist ein Pointer in ein Query Result Set, welchen man wie ein Iterator nach vorne bewegen kann. Ein Cursor muss immer geschlossen werden.

```
DECLARE
curs1 refcursor; --unbound(any query)
curs2 cursor (id integer) for select * from abteilung where abtnr = id; --rowvar abteilung%rowtype;
BEGIN
open curs1 for execute 'select * from abteilung where abtnr = $1' using id; open curs2; open curs3(2);
LOOP
FETCH curs1 INTO <rowvar>; EXIT WHEN NOT FOUND;
UPDATE abteilung SET abtnr = rowvar.abtnr + 1 WHERE CURRENT OF curs1;
END LOOP; close curs1; close curs2; close curs3; END;
```

**Verteilte DBMS** (Wichtigste Anforderungen: **Transparenz** und **Atomarität** von verteilten Transaktionen (2PC))

**Homogen:** 1 einheitliches Verteiltes System. **Heterogen:** verschiedene SW und Schemas, Knoten können nichts voneinander wissen

Vorteile Replikation	Verfügbarkeit, Parallelität=Speed, Reduzierte Datentransfers bei lokal replizierten Daten
Nachteile Replikation	Höhere Update-Kosten, Komplexere Synchronisation, Deadlocks, Timeout Handling
Fragmentierung	Horizontal: (Sharding) Trennung zwischen Datensätzen als ganzes in Fragmente (Shards) → oft bei NoSQL DBMS Vertikal: Zerlegung eines Datensatzes über mehrere Nodes Verteilt nach Spalten

**Two Phase Commit Protokoll 2PC**

Sichert Atomarität und Konsistenz von verteilten Transaktionen. Hat zentralen Transaktionsmanager. Wird verwendet sobald mehrere RM zum Einsatz kommen. Bei DBMS: sobald sich die Transaktionen über mehrere Session erstrecken.

**Transaktion Manager TM:** Steuert abgeglichene Committen auf allen Nodes. **Resource Manager RM** = DBMS

**Phase 1:** Alle RM mit «Prepare to Commit» und «Ready»/«Not Ready»-Rückmeldung vorbereiten

**Phase 2:** «Commits auf allen RM (rsp. «Rollbacks» und warten auf «Ack» von allen RM.

Falls einer Abbricht, wird überall abgebrochen. Steuerung durch TM.

**Vorteile:** Bewährt, gute Unterstützung. **Nachteile:** Langsam, Skaliert schlecht (Blockierend → synchron)

**API:** Der TM kommuniziert über das XA-Interface mit den RM und steuert so deren Transaktionen. Die Applikation greift nicht auf die lokalen T-Schnittstellen zu sondern auf die XA-Operationen des TM

**Globale Locks:** oft ein Node, welcher Locks verwaltet: ineffizient, SPOF

**Lokale locks** mit 2PC: Verteilt aber langsam und deadlock-gefährd

Zeitstempel: Synchronisation über Timestamps. Problem: braucht *sehr* genaue Uhren

**Deadlocks**

**Erkennung Lokal:** bei Deadlocks, Transaktion wiederholen.

**Erkennung Global:** zentraler Knoten, welcher die Wartbeziehungen kennt.

**Vermeidung:** Validierung/Rollbacks nach Durchführung, Read/Write-Stempel zwecks Serialisierung

**Verteilte Queries**

Anzahl Queries limitiert von Netzwerk I/O (statt Disk-I/O)

**Join:** Ship Whole(alles auf einem Host zusammenführen), Fetch-As-Needed(Join-Felder werden mitgeteilt)

**Ship Join:** optimiertes Fetch-As-Needed für weniger Pakete

**Query Processing Strategien**

**Semi Whole:** Relation vollständig auf einen Knoten übertragen. Minimale Nachrichtenzahl, hohes Volumen.

**Fetch As Needed:** Nur relevante Join-Attribut-Werte an einen Rechner senden.

Hohe Nachrichtenzahl, dafür evtl. tieferes Volumen (bei hoher Join-Selektivität)

**Kostenabschätzung:** Anzahl # Nachrichten sowie # Attribute («AWs») Für jede Nachfrage bei einem Node gibt es 2 Nachrichten (Request und Reply)

Join-Attributwerte, welche angefragt werden, werden nicht nochmal zurückgeschickt.

Gleiche Werte werden aber mehrmals abgefragt.

**Semi-Join:** Optimierung von Fetch As Needed, damit werden jeweils in einer Nachricht alle duplizierten Verbundattributwerte angefragt und in einer Nachricht alle relevanten Attributwerte (inkl. der Join-Spalte) zurückgesendet.

**Zeitstempel**

System vergibt eindeutiger Zeitstempel zum Begin Of Transaktion ts(T).

**Synchronisation**

**Lesen:** Nach dem Begin der Transaktion darf ein betroffenes Objekt x nicht geändert werden: ts(T) < WTS(x)

**Schreiben:** Neben schreiben (WTS) darf das Objekt auch nicht gelesen worden sein in der Zwischenzeit: ts(T) < Max(RTS(x), WTS(x))

**Replikation:**

Nutzen: höhere Ausfallsicherheit/Verfügbarkeit, schneller Antwortzeiten (Load-Balancing, geografisch), skaliert besser

Asynchroner Multi-Master Replikation über verteilte Standorte, Peer-Peer Verbindungen, mittelschnelle Verbindung mit hoher Verfügbarkeit nötig.

Synchroner Master Slave Readonly Slave für schnelle Antwortzeiten. Master R/W → synchroner Sync (2PC)

Asynchroner Master-Slave Warm Standby Backup Slave. Live Daten werden asynchron backupped. (Inkonsistenzen möglich)

**Strong Consistency:** Alle müssen gleichen Datenstand haben (ACID),

**Weak Consistency:** Unterschiedlicher Lese-Stand möglich (BASE).

**Synchrone Replikation (eager):** Aktive replikation von Transaktionen, Locks über alle Nodes (2PC), ACID. Deshalb tiefere Availability

**Asynchrone Replikation (lazy):** Replikation erst nach abgeschlossenen Transaktionen, periodisch oder nach Commit. Konflikte möglich: Update, Uniqueness, delete, ordering

**Synchronisationsprotokolle**

**Write-All/Read-Any-Strategie** Synchroner Änderung der Replikate am Ende Transaktion → alle Konsistent, daher beliebige, hochverfügbare Lesezugriffe, aber hohe Änderungskosten da z.B. 2PC nötig. Bei P Verfügbarkeit eines Nodes und N Nodes kann mit P<sup>N</sup> geschrieben und 1 → (1 - P)<sup>N</sup> gelesen werden.

**Read One / Write All** Variante: Write-All-Avalanche, Unverfügbare Nodes werden später als Protokoll aktualisiert.

**Majority Protocol**

Lesen & Schreiben verlangt Zugriff auf Mehrheit der Replikate. Gleichzeitig nur 1 Schreib-Transaktion: Mehrheit Replikate wird gelockt, Versionsnummer von Objekt erhöht.

**Vorteile:** Funktioniert bei kleinen Ausfällen;

**Nachteile:** Braucht >2 Knoten damit eine Mehrheit möglich ist

- Hoher Kommunikationsaufwand: n + 2 Msg. für Lock Requests,  $\frac{n}{2} + 1$  für Unlock Requests;
- Potentiell Deadlocks (z.B. 3 Nodes, 3x  $\frac{1}{2}$  Locks)

**Quorum Consensus Protocol**

Jede Kopie erhält Anzahl Votes, Lesen erfordert R Votes, Schreiben R' Votes, V: total Stimmen

$R + W > V$  (Nicht gleichzeitiges Lesen/Schreiben)  $W > \frac{V}{2}$  (Nur von einer Transaktion änderbar)

z.B. <2,1,1,1> → V=5 → z.B. R=3, W=3 oder R=2, W=4 (bevorzugt Lesen)

**Snapshot-Replikation (Materialized View)**

Snapshot = materialisierte View, nur Lesezugriff. Daher oft, einfache, einseitige Synchronisation.

Diverse Aktualisierungsmethoden : on demand, readonly, on commit

**Merge-Replikation**

Publisher und mehrere Subscribers. Publisher macht einen Snapshot und kopiert diesen auf die Subscribers. Publisher und Subscriber machen Änderungen und synchronisieren diese von Zeit zu Zeit. (Anwendung: lokale DB ab mobilen Clients, die auch offline sein können)

Unsynchrone Änderungen mit nachträglicher **Konflikt**: Behandlung für Update, Uniqueness, Delete, Order. **Erkennung** übtl. Timestamp. Konfl.-Behandlung: z.B. Newest, Site Priority, Manuell

**Replikation in kommerziellen Systemen**

**MySQL**

**Snapshot:** Immediate/Queued Updateing. +Gut für wenig Daten, +Keine Schemaänderungen +Funktion mit Snapshots

**Transactional:** Transaktionsweise Log-Daten übertragen. Möglich mit 1 Publisher und N Subscriber (und N P1<>1 S.).

+Push/Pull möglich +Synchroner Modus (Publisher pushed, evtl. 2PC Lock.) +Asynchron (Transaction Queue, Konflikte möglich)

**Merge:** Einsatz von Triggern zum Propagieren, benutzerdefinierte Konfliktauflösung. Meistens unzuverlässige Verbindung oder Offline; →MySQL macht «rowguid» zum eindeutigen Abgleich.

**Oracle**

**Materialized View** (Master – View): MV enthält Snapshot der Daten. Basiert auf Queries, meistens Read Only.

Replikation mit Triggern, Deferred Transaction Queue und Background-Prozesse mit RPCs zur Übertragung.

**MultiMaster:** Änderungen überall möglich, asynchrone und synchrone Propagation möglich.

**Hybrid** (inkl. zu anderen DB-Vendors); Div. 3<sup>er</sup> Party Produkte

**MongoDB**

**Features:** Dokumentenorientiert (BSON), Embedded documents and arrays (weniger joins), Keine Joins/Multi-Document-Transaktionen, Indexing, Dynamisches Schema (Keine DDL), Query API, Konfigurierbare Atomicity, HA Replikation, GridFS (DFS-API)

**Struktur:** Databases → Collections → Documents → Fields. **Mongo Shell:** use unicorns // erstellen/öffnen Datenbank

db.unicorns.insert({name: 'Horny', dob: new Date(1992,2,13,47), loves: ['carrot', 'papaya'], weight: 600, gender: 'm'}); // überschreibt ganzes Document!

db.unicorns.update({name: 'ABC'}, {\$set: {weight: 590}}) // überschreibt field

db.unicorns.remove({}) // Alle collections löschen

db.unicorns.find(<gender>='f', \$and: [{loves:'apple'},{loves: 'pie'}]) // alle weiblichen Unicorns, die apple&pies mögen.

db.unicorns.find(<gender>='m',weight:{\$lt:900,\$gte:600}) // alle Männchen mit gewicht (600..900)

db.unicorns.find(<gender>='Y'),\_id:0,name:1)) // nur ausgewählte felder anzeigen

**Keywords:** \$regex, \$ne Not equal, \$gt Greater than, \$lte Less than or equal, \$lt Greater than or equal, \$exists a field, \$all Match all elements in an array, \$in Match any elements in an array, \$nin Not match any \$in, \$elemMatch Match all fields in an array of nested documents, \$or, \$nor, \$size Match array of given size, \$mod url, \$type if field is of datatype, \$not Optionen (letzter parameter): {upsert: true} update creates new document if no match, {w: 'majority'} // Write Concern

**Funktionen** am Schluss angehängt: .count() // Resultate Zählen., .pretty() // formatierte Ausgabe, .help() // Kommandos

**Joins:** existieren beim lesen nicht. Müssen beim Schreiben selber verwaltet werden!

db.emps.insert({\_id: ObjectId("1"), name: 'Leto'}, db.emp.insert({\_id: ObjectId("2"), name: 'Duncan', manager: ObjectId("1")}));

db.emp.findOne({\_id: db.emps.findOne(name: 'Duncan').manager}) // finde Manager von 'Duncan'

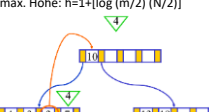
**Mongo Replikation**

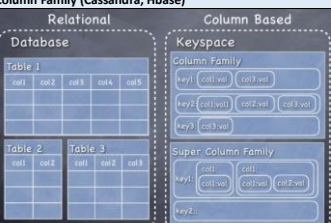
**Elected Master-Slave** (Primary-Secondary); Writes nur auf Primary, nach Schreiben repliziert. Die Majority bestimmt die persistenden Daten. Datensätze werden in Shards aufgeteilt und repliziert. Secondaries erlauben nur Reads mit entsprechender ReadPreference.

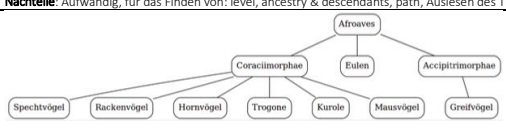
Es gibt aber keine Garantie auf Aktualität (da **async**)

**Aufteilung** in je 1-N Mongod (Datenbank-Instanz), Mongos (Sharding Process, Verteilt requests und fügt Resultat zusammen), Config Servers (Metadeta server, meistens 3), Mongo (Shell Client)

**Optimale Aufteilung:** DataCenterA: 2, DcB: 2, DcC: 1; so kann bei einem Ausfall eine Mehrheit abstimmen (sonst keine primary election möglich).

<b>Write Persistence:</b> Primary Hosts nimmt writes und repliziert diese auf Secondaries <b>Write Concerns:</b> Durch setzen des Write Concerns müssen alle Schreiboperationen von der Mehrheit des Nodes bestätigt werden. <i>Errors ignored</i> (write not acked), <i>unacknowledged</i> (driver/network errors), <i>receipt acknowledged</i> (Primary acks), <i>journalled</i> (Primary writes logs and acks), <i>reply acknowledged</i> (acks from Primary and 1 Secondary (w:2) or majority of nodes (w: "majority")) <b>Vertical Scaling:</b> More CPU and Storage <b>Sharding / Horizontal Scaling:</b> Daten mit einem Partition Key partitionieren und über mehrere Server verteilen. Divide Data over multiple servers (whole documents). A shard is one logical, replicated database on a set of nodes. Balanced Distribution with a consistent HASH(User-defined Shard key (id)) or range based. Shards are split if they exceed 64MB. Deployments mit grossen Datasets und hohem Durchsatz so unterstützt <b>Shard:</b> jeder Shard ist ein einzelnes Replication Set. <i>sh.addShard('localhost:10001'); /*etc...*/</i> ; <i>sh.enableSharding('DB');</i> /*Shardkey:*/ <i>sh.shardCollection('DB.COL', {FIELD:1}; sh.status()); db.printShardingStatus(true)</i> <b>Replication Set:</b> A cluster of N (min: 3,max: 12. more needs Sharding) nodes, Consensus election of 1 primary node, All write operations go to primary, All data replicates from primary to secondary. Automatic failover+recovery <i>rs.initiate([_id: 'firstset', members: [ { _id:1,host:'localhost:10001',priority:2}, { _id:2,host:'localhost:10002',priority:1,}], db.isMaster().ismaster /* true if current node*/; rs.slaveOk(); rs.add(HOST_NAME:PORT)</i> <b>Operation Log:</b> Mongo keeps an ordered list of write operations that have occurred. The secondary members then copy and apply these operations in an <i>asynchronous</i> process		
<b>Interne Ebene (Indexe, Zugriffskosten, Query Execution)</b> Heap besteht aus mehreren Pages. Eine Page enthält mehrer unsortierte Rows. Beim Table Scan werden alle Pages durchsucht. <b>Indexe</b> Indexe machen nur Sinn, wenn ein Subset abgefragt wird. (Bei 80% Ausbeute ist der TableScan ohne Index Overhead schneller) <b>Vorteile:</b> Optimierte Abfragen (optimal bei PK, FK, Range Query Attributen, bei häufigen Sortieranfragen, bei häufige Aggregationen) <b>Nachteile:</b> Update Overhead. Indexe benötigen zusätzlichen Speicherplatz. Reihenfolge der Attribute spielt eine Rolle <code>CREATE INDEX myindex ON myTable(myColumn)</code> <code>DROP INDEX myindex;</code> Auswahlkriterien: Tabellengrösse, Wie viele Inserts/Updates, Welche Queryattribute, Attributreihenfolge beachten Index Kandidaten: Schlüssel, JOIN-Teilnehmer, Gleichheits oder Bereichbedingungen, Sortierungen, Group By Bedingungen <b>Index Varianten</b> (Balanced Trees, Clustered & unclustered Index, HashIndex, Bitmap) <b>Heap</b> Daten liegen unsortiert auf dem Heap Balanced Tree (B-Tree) Geeignet für Sekundärspeicher, Gut bei Range und Equal Search. Gute Effizienz für viele Arten von Abfragen. Schnelle Updates des Trees. Jeder Weg von der Wurzel zum Blatt hat die gleiche Länge. B+ Tree Daten werden nur in den Blätter gehalten. Innere Knoten enthalten nur Zeiger. Blatt Knoten enthalten nur Disk-Referenz auf gespeichertes Tupel. (Clustered enthalten direkt Tupels) Ein Knoten entspricht einer Seite. Baum vom Grad k m: max. Anzahl Element pro Knoten muss >3 sein (min: k, max: 2k) N: Anzahl Knoten, alle internen Knoten haben n+1 Kinder bei n Einträgen pro Knoten <b>Knoten:</b> - Ein Knoten beinhaltet mindestens m/2 Elem (ausser root >=1) - Ein Knoten hat maximal m+1 Unterknoten (sonst rebalancing) - Der Baum ist überall gleich hoch → balanced (sonst rebalancing) max. Höhe: h=1+log <sub>2</sub> (m) (N/2)] 		
Clustered Index	Blätter enthalten Tupels in aufsteigend sortierter Reihenfolge. Sehr schnell bei Range Queries, da nur der Einstiegspunkt gefunden werden muss. (danach ist es sortiert). Durchschnittliche Indexgrösse 5% der Tabellengrösse. Maximal 1 Clustered Index pro Tabelle.	
Unclustered Index	Blätter enthalten Referenzen auf die Daten Records. Da es nur einen Clustered Index geben kann, werden weitere B+ Indexe unclustered definiert. (ACHTUNG: Zusätzlicher I/O Zugriff)	
Hash	Der Key wird durch eine Hash Funktion gelassen. Danach Hash <= Value Mapping. Wichtig bei In-Memory DBMS. Muss reorganisiert werden, wenn zu viel Overflow Chaining. Kein WAL-Log!	
Bitmap	Ordnet Attributwerte als Bitmap. Schnelle Anfragen bei AND und OR. Geeignet bei kleinem Wertebereich. Teure Updates.	
<b>Kostenmodell</b> P = Anzahl Pages/Blöcke mit Fillfaktor = 67%, R = Anzahl Records pro Heap-Page, F = Fanout: Durchschnittliche Anzahl Kinder eines Internen Knoten, Pl = Anzahl Pages im Leaf Level des Indexes, n = Anzahl Tupels, die eine Suchbedingung erfüllen		
<b>Index</b>	<b>Suche</b>	<b>Anzahl I/O (aufrunden)</b>
Heap	Table Scan / Heap Scan:	P (alle Attribute anschauen)
	Equality Search (Unique Attribute)	P/2 (ca. die Hälfte aller Attribute)
Clustered Index	Range Search	P
	Table Scan / Heap Scan:	P
	Equality Search	log(P)
	Range Search	log(P) + n/R
Unclustered Index mit Heap	Index Scan für sortierten Zugriff	P · R / F + P · R
	Equality Search	1 + log(P <sup>2</sup> /R)
	Range Search	log(P <sup>2</sup> /R) + n + n / F
<b>Query Processing</b>		
1. Query Engine parst SQL Statement und konvertiert es in Query Tree 2. Query Tree wird allenfalls transformiert 3.Optimierer generiert Ausführungspläne und wählt den günstigsten aus 4. Zur Laufzeit werden die kompilierten Ausführungspläne mit den aktuellen Variablen gebunden und ausgeführt.		
<b>Kostenmodell Joins</b> $\log(100)=2, \log(x)=\log_2(x), \log_2(2^{10}) \rightarrow$ annähernde Lösung, berechenbar mit binär Potenzen C = Kosten, N=Anz. Tupel einer Relation (S/R), P=Anzahl Pages der Relation B=Puffer Seiten, c=Kosten für Index Traversierung		
<b>Join Variante</b>	<b>Beschreibung</b>	<b>Kosten C</b>
Hash Join	Intern eine Hashtabelle bei der kleineren Relation. (default, wenn kein Index existiert) → Partition und Probe Phase	$C = 3 * (P(R) + P(S))$
Nested Loop Join	Falls ein Index existiert, ist dieser Join häufig schneller wie der Hash Join.	$C = P(R) + P(S) * P(R)$ falls P(R) oder P(S) in B passt: $C = P(R) + P(S)$
Block Nested Loop		$C = P(R) + P(S) * (P(R) / (B-2))$
Indexed Nested Loop		$C = P(R) + c * P(R) * N(R)$
Merge Join	Sortiert beide Relationen und merged Join Columns	$C = P(R) * \log(P(R)) + P(S) * \log(P(S)) + P(R) + P(S)$ $C = P(R) + P(S)$ (falls vorsortiert)

<b>Anfrageoptimierung</b>		
1. Übersetzung der Query 2. Logische Optimierung (Selektion so früh wie möglich) 3. Physische Optimierung (Einbezug von Indizes) 4. Wahl des besten Plans (generiere alle möglichen Pläne und wählen schnellsten)		
<b>Kostenbasierte Opt</b>	Unter Einbezug von Statistiken wählt der Optimizer jene Query mit den tiefsten Kosten	
<b>Statistik</b>	Information über die Anzahl und die Verteilung der Daten	
<b>Selektivität</b>	Selektivität $\leq 0.1 \rightarrow$ DBMS verwendet Index, Selektivität $> 0.1 \rightarrow$ DBMS macht Table Scan	
	Hohe Selektivität bedeutet tiefer Wert bei der Formel $\rightarrow$ Hohe Selektivität = Tiefe Dichte	
	<b>Point Queries:</b> Anz. Qualifizierende Tupel / Gesamt Anz. Tupel	Mit Histogram: 0.1 pro Einheit
<b>Range Queries:</b> (WHERE Range) / Wertebereich des Attribut		
<b>Dichte</b>	Anz. Distinct Tupel / Anz. Tupel $\rightarrow$ Durchschnittlicher prozentualer Anteil an Duplikaten	
<b>NoSQL DBs(Not Only SQL)</b>		
Schemalose Datenbanken, verzichtet auf ACID verwendet stattdessen <b>BASE</b> (Basically Available, Soft state, Eventually consistent). <b>CAP</b> (Consistency (atomicity), Availability (non-failing nodes must respond), Partition tolerance(only total network failure leads to wrong answers))-Theorem: alle drei Eigenschaften zu erfüllen unmöglich. Meistens wählt man zwischen A und C. <b>Map Reduce</b> anstatt Joins, <b>keine Einheitliche Abfragesprache/Standards</b>		
Consistency & Availability	Single-Site + Clustered DBs, cluster-based designs	Eine Transaktionslogik wie 2PC muss existieren
Partition T & Availability	DNS, Web Cache, Distributed Systems for mobile environment(Coda, Bayou)	Typische Features: Optimistisches updaten mit Konfliktauflösung, wie Internet, TTL
<b>Kategorien</b>	<b>Beispiele</b>	<b>Anmerkung</b>
Key Value	Riak Redis, DynamoDB, Voldemort	Einfachste NoSQL Form: Simple Hashtabelle (Key/Value). Oft keine Abfragesprache. Get/put/delete-Operationen, falls bei Put schon existiert -> überschreiben <b>Riak:</b> Schlüssel werden in Buckets gespeichert, P2P, bei Conflict 2 Ansätze: neuester Write oder client entscheidet, nur mit Key queries
Document DB	MongoDB, CouchDB	Erweitertes Key/Value Konzept, wobei mehrere Keys in einem Dokumente gruppiert werden. Dokument in JSON/BSON(binär mit Typinformation) Format gestored. (Siehe $\rightarrow$ MongoDB)
Map Reduce	Map Shuffle Reduce. Daten auf bestimmte Attribute mappen und nach Kriterien reduzieren. var emitCustPrice = function(){emit(this.cust_id, this.price); } //map Funktion var sumUp = function(custid, prices){return Array.sum(prices); } // reduce Funktion db.orders.mapReduce(emitCustPrice, sumUp,{out:'PurchasesPerCustomer'});	
Polyglot Persistence	Man nutzt die Vorteile von verschiedenen Welten und kombiniert diese auf eine optimale Art und Weise. Datenbank, die verschiedene DBMS "spricht". Bsp. E-Commerce Plattform hat Shopping Cart data in Key-Value Store, die vollständigen Bestellungen in RDBMS und Sessioninfos ebenfalls in einem Key-Value Store.	
<b>Graph</b>		
Neo4J, FlockDB, InfiniteGraph. Bestehend aus Knoten und Beziehungen. Joins sind hier implizit, da Abfragen auf Beziehung sind <b>Neo4J Syntax:</b> CREATE (alice:user {username:'Alice'}), (bob:user {username:'Bob'}) //Alice und Bob als User erstellen CREATE (alice)-[:ALIAS_OF]->(bob) // Beziehung namens ALIAS_OF von Alice zu Bob erstellen MATCH (bob:user {username:'Bob'})-[:SENT]-[:email] RETURN email // gibt den Email Knoten zurück, den User mit username 'Bob' gesendet(Beziehung SENT) hat MATCH(ca:Movie {title:'Cloud Atlas'})<-[:DIRECTED]-[:regs:Person] RETURN regs.name // print Name des Regisseurs von Cloud Atlas MATCH p=shortestPath((bacon:Person {name:'Kevin Bacon'})-[*]-[:meg:Person {name:'Meg Ryan'}]) RETURN p // Gibt kürzeste Verbindung zwischen Kevin Bacon und Meg Ryan aus		
<b>Column Family (Cassandra, Hbase)</b>		
		Der <b>Keyspace</b> entspricht der Datenbank im rel. Modell Eine <b>Column Family</b> entspricht einer Tabelle im rel. Modell.  Eine Family besteht aus mehreren Key/Value Paaren. Ein Paar repräsentiert eine <b>Row</b> : $\rightarrow$ Key = RowKey, Value= mehrere zusammengehängte Columns. Mehrere Rows können aus unterschiedlich vielen Columns bestehen. Die Row Keys sind stets sortiert.  Eine <b>Column</b> ist ein Triplet bestehend aus Column Name, Value und einem Timestamp der von der NoSQL DB zur Konfliktbehandlung verwendet wird. Die Columns sind ebenfalls sortiert nach Name
<b>Vorteile:</b> Pro Query müssen nur die relevanten Spalten ins Memory geladen werden! (Relational immer alle) Durch die sortierte Speicherung der Columns kann sehr effizient komprimiert werden. <b>Hbase:</b> Basierend auf bigtable von Google, REST-API, I/O für MapReduce Jobs, Auto Partitioning & Rebalancing <b>Cassandra:</b> Cassandra hat in einem Cluster keinen Master-Node, jeder Node kann Reads/Writes verarbeiten Bei Keyspace-Creation: Replikationsfaktor kann gesetzt werden <b>Write:</b> - Data in Commit log & in-memory struktur memtable geschrieben, defaultmässig reicht das für writesuccess. – Writes periodisch in SSTables geschrieben – man kann consistency setting konfigurieren zB auf QUORUM <b>Read:</b> ConsistencySetting = ONE: Daten des ersten Replikates wird return, auch wenn Daten veraltet ConsistencySetting = QUORUM: Mehrheit der Knoten werden zugegriffen & Column mit neuestem Timestamp returned ConsistencySetting = ALL: alle Knoten müssen auf Read/Writes antworten <b>Transaktionen:</b> gibt keine Transaktionen im traditionellen Sinn, aber auf row-level sind zugriffe atomar. <b>Availability:</b> überprüft mit der (R+W)/N-Formel. R (Minimum Knoten die auf Read-Operationen bestätigen antworten) und W(Minimum Knoten mit erfolgreicher Schreib-Operation)-Werte können auf Replikate geändert werden. <b>Partitioning:</b> In jedem Cassandra Cluster gibt es einen logischen Ring von HashValues. Jedem Knoten im Cluster wird eine Position in diesem Ring zugewelt, wo er für die vorhergehenden Hashes verantwortlich ist. Für Membership & Failure Detection gibt es Gossip-Protokoll. Mit diesem wird von jedem Knoten aus jeweils mit einem Austausch von Statusinformationen mehr Informationen über das Cluster herausgefunden. Status ist in O(log N) auf allen Knoten verteilt. Failure Detection: Bei einem Heartbeat, der nicht beantwortet wird, wird eine Zahl Phi(Verdacht auf Down) verteilt. Falls phi>threshold wird dieser Knoten als offline markiert. Average für Failure Detection 10-15s mit threshold 5		
<b>Indexierung</b> (Auf Column 'city') UPDATE COLUMN FAMILY Customer WITH comparator=UTF8Type AND column_metadata = [{ column_name: city, validation_class:UTF8Type, index_type: KEYS];		<b>Cassandra Query Language (CQL)</b> CREATE COLUMN FAMILY Customer(KEY varchar PRIMARY KEY, name varchar, city varchar, web varchar) INSERT INTO Customer(KEY,name,city,web) VALUES ('mfowler','Martin Fowler','Boston','www.mf.com') SELECT * FROM Customer WHERE city='Boston'
<b>Verwendung:</b> Event logging, CMS/Blogging Platforms, Web App analytics <b>Nicht verwenden bei:</b> Systemen, die ACID für R+W verlangen, Aggregieren von Daten (SUM/AVG), Prototyping(Query Change teurer als SchemaChange!)		

<b>Datenstrukturen</b>																																						
<b>Arrays (nicht relational, geeignet bei wenig Updates/Inserts, Ähnlichkeit zu NoSQL → Array-DB)</b>																																						
einfacher Array: ARRAY['a','b'];	2D Array: ARRAY[[ 'a', 'b'], ['d','c']]																																					
CREATE TABLE sales_emp (name text, schedule text[]);	INSERT INTO sales_emp VALUES('Bill', ARRAY['a', 'b'], ['d','c']);																																					
Array Konstruktoren	SELECT ARRAY[1,2,3,4]; returns: (1,2,7) SELECT ARRAY( SELECT 1 + (random() *5)::int FROM generate_series(1,6) ORDER BY 1 ); returns e.g. : {1,2,3,4,5,5}																																					
Selecting values: Array subscripts and slices	// lowerbound:upperbound pro array dimension SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill'; returns: {[a],[d]} SELECT * FROM sal_emp WHERE 10000 < ANY (pay, by, quarter);																																					
Array Operatoren	Equality: =, „is contained by“: <@ „Overlap“: &&	SELECT ARRAY[1,1,2,1,3,1]::int[] = ARRAY[1,2,3]; SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6]; SELECT ARRAY[1,4,3] && ARRAY[2,1]																																				
Hilfsfunktionen	SELECT cardinality(schedule) FROM sal_emp; returns: 4 // total number of elements of all dim SELECT array_dims(schedule) FROM sal_emp returns: {1:2}[1:2] SELECT array_length(schedule, 1) FROM sal_emp returns: 2 // length of specified dim																																					
<b>Dictionaries (Abstrakte Datenstruktur+ Datentyp, Maps unique key to value, Key Value Pairs (KVP)= Entity Attribute Value* (EAV))</b>																																						
used for:	Easy data storage and data capture, Rows with many attributes that are rarely examined, Semi-structured data																																					
SELECT "a">1,a>2::hstore; returns: "a">="1"	CREATE TABLE test (col1 integer, col2 text, col3 text); INSERT INTO test VALUES (123, 'foo', 'bar'); SELECT hstore(t) FROM test AS t; returns: "col1">="123", "col2">="foo", "col3">="bar"	List all keys SELECT akey(myfield) FROM ... Get all K-V pairs SELECT each(myfield) FROM ... Get K V (as text) SELECT myfield->'name' FROM ... contains WHERE myfield @> "a">="foo"; --or hstore("a","zoo")																																				
<b>Trees</b>																																						
<b>Adjazenzliste</b>																																						
<b>Columns:</b> Id, Name, ParentID <b>Vorteile:</b> Schnelle Knoten moves, inserts, und deletes. <b>Nachteile:</b> Aufwändig, für das Finden von: level, ancestry & descendants, path, Auslesen des Trees braucht mehrere Queries																																						
		<table><tr><td>id</td><td>name</td><td>parentid</td></tr><tr><td>1</td><td>Afroaves</td><td>NULL</td></tr><tr><td>2</td><td>Coraciiformae</td><td>1</td></tr><tr><td>3</td><td>Spechtvögel</td><td>2</td></tr><tr><td>4</td><td>Rackenvögel</td><td>2</td></tr><tr><td>5</td><td>Hornvögel</td><td>2</td></tr><tr><td>6</td><td>Trogone</td><td>2</td></tr><tr><td>7</td><td>Kurolo</td><td>2</td></tr><tr><td>8</td><td>Mausvögel</td><td>2</td></tr><tr><td>9</td><td>Greifvögel</td><td>2</td></tr><tr><td>10</td><td>Accipitriformae</td><td>1</td></tr><tr><td>11</td><td>Greifvögel</td><td>10</td></tr></table>	id	name	parentid	1	Afroaves	NULL	2	Coraciiformae	1	3	Spechtvögel	2	4	Rackenvögel	2	5	Hornvögel	2	6	Trogone	2	7	Kurolo	2	8	Mausvögel	2	9	Greifvögel	2	10	Accipitriformae	1	11	Greifvögel	10
id	name	parentid																																				
1	Afroaves	NULL																																				
2	Coraciiformae	1																																				
3	Spechtvögel	2																																				
4	Rackenvögel	2																																				
5	Hornvögel	2																																				
6	Trogone	2																																				
7	Kurolo	2																																				
8	Mausvögel	2																																				
9	Greifvögel	2																																				
10	Accipitriformae	1																																				
11	Greifvögel	10																																				
<b>WITH RECURSIVE</b> a AS ( SELECT id, name, parent_fk FROM animals WHERE name = 'Trogone' <b>UNION ALL</b> SELECT p.id, p.name, p.parent_fk FROM a JOIN animals p ON a.parent_fk = p.id) SELECT * FROM a ORDER BY 1;																																						
<b>Nested Set</b>																																						
<b>Vorgehen:</b> Modified Preorder Tree Traversal: Beim ersten Besuch left aufschreiben, beim zweiten right <b>Columns:</b> Id, Name, Left, Right																																						
<b>Vorteile:</b> Einfach: lvl, ancestry, descendants, Tree mit 1 Query auslesbar																																						
<b>Nachteile:</b> Aufwändiger im vgl. zur Adjazenzliste: moves, inserts, deletes.																																						
WITH p AS (SELECT lft, rgt FROM animals where name='Accipitriformae') SELECT a.id, a.name FROM animals a, p WHERE a.lft BETWEEN p.lft AND p.rgt ORDER BY a.lft;																																						
<b>Result:</b> 10. Accipitriformae 11. Greifvögel																																						
<b>Materialized Path (Label Tree)</b>																																						
<b>Columns:</b> Id, Path <b>Vorgehen:</b> Pfad vom Parent zum gesuchten Knoten notieren [1,2,4] $\rightarrow$ Rackenvögel <b>Contains:</b> SELECT path FROM test WHERE path = * Accipitriformae.* $\rightarrow$ Afroaves, Greifvögel <b>Inheritance:</b> SELECT path FROM test WHERE path <@ ARRAY[1,2]; $\rightarrow$ Alle Children von Coraciiformae																																						
<b>Ausführungspläne</b>																																						
<b>EXPLAIN ANALYZE</b> SELECT Orderid, OrderDate FROM Orders WHERE OrderDate BETWEEN '2004-01-01' AND '2004-02-01'; QUERY PLAN text Seq Scan on orders (cost=0.00..280.00 rows=1007 width=8) (actual time=0.007..1.790 rows=1039 loops=1) Filter: ((orderdate >= '2004-01-01'::date) AND (orderdate <= '2004-02-01'::date)) Rows Removed by Filter: 10961 Planning time: 0.267 ms Execution time: 1.840 ms <b>Analysieren Sie den Ausführungsplan und beantworten Sie die folgenden Fragen: Anzahl Tupels in der Resultatmenge: 1039</b> <b>Anzahl der vom Optimizer geschätzten Tuples: 1007</b> <b>Wie viele Tuples werden gefiltert? 10961</b> <b>Ausführungszeit: 1.84 ms</b>																																						
<b>Die Anfrage wird mit folgendem Index optimiert: create index ix99 on Orders(Orderid,OrderDate);</b> <b>EXPLAIN ANALYZE</b> SELECT Orderid, OrderDate FROM Orders WHERE OrderDate BETWEEN '2004-01-01' AND '2004-02-01'; QUERY PLAN text Index Only Scan using ix99 on orders (cost=0.29..270.36 rows=1007 width=8) (actual time=0.183..1.064 rows=1039 loops=1) Index Cond: ((orderdate >= '2004-01-01'::date) AND (orderdate <= '2004-02-01'::date)) Heap Fetches: 0 Planning time: 0.405 ms Execution time: 1.114 ms <b>Erklären Sie den Query-Plan! (2 P) Execution time 1.114 ms, also etwas schneller</b> <b>Zeile 4 und 5: Index Only Scan: d.h. die ganze Anfrage wird auf dem Index ausgeführt. Keine Heap-Fetches</b>																																						