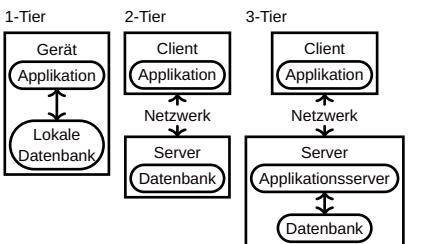
**Glossar**

Begriff	Bedeutung
Impedance-Mismatch	Diskrepanz zwischen Datenstrukturen auf Applikations- und Datenbankebene
System-/Datenkatalog	Enthält Metadaten über die Datenbankobjekte, z.B. Tabellen und Schemata.
Datenbankschema	Struktur einer Datenbank, die die Organisation der Daten und Beziehungen beschreibt.
Datenbasis	Der physische Speicherort
Surrogate Key	Künstlich generierter PK
Referentielle Integrität	Fremdschlüssel muss zu einem Wert der referenzierten Tabelle oder NULL zeigen
Datenunabhängigkeit	Daten in einer DB ändern können, ohne dass Anwendungen geändert werden müssen
Data Pages	Kleinste Speicher-Dateneinheiten einer DB
Heaps	Unsortierte Datenorganisation
Semantische Integrität	Daten sind nicht nur syntaktisch, sondern auch inhaltlich korrekt, insbesondere nach T
Data dictionary	Zentrale Sammlung von Metadaten über die Daten im DBMS

**Datenbankmodelle**

Begriff	Bedeutung
Hierarchisch	Daten sind in einer baumartigen Struktur geordnet
Netzwerk	Flexiblere Struktur als hierarchisch, erlaubt mehrere Pfade zwischen Entitäten
Objektorientiert	Speichert Daten und ihr Verhalten in Form von Obj.
Objektrelational	Kombiniert objektorientierte + relationale Prinzipien
Relational	Speichert Daten in Tabellen (Relationen) und verwendet Beziehungen durch Schlüssel

**DataBase System (DBS)**

Besteht aus DBMS und Datenbasen

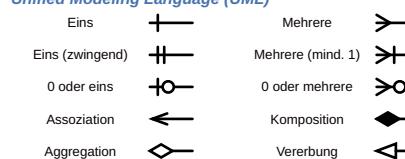
 **DataBase Management System (DBMS)**

- (A) Transaktionen
- Datentypen
- (C) Konsistenz
- Abfragesprache
- (I) Mehrbenutzerbetrieb
- Backup & Recovery
- (D) Große Datens Mengen
- Redundanzfreiheit
- (D) Sicherheit
- Kapselung

**ANSI Modell****Logische Ebene:** Logische Struktur der Daten**Interne Ebene:** Speicherstrukturen, Definition durch interne Schema (Beziehungen, Tabellen etc.)**Externe Ebene:** Sicht einer Benutzerklasse auf Teilmenge der DB, Definition durch externes Schema**Mapping:** Zwischen den Ebenen ist eine mehr oder weniger komplexe Abbildung notwendig**Relationales Modell**

PK sind unterstrichen, FK sind kursiv tabellenname (

- id SERIAL PRIMARY KEY,
- grade DECIMAL(2,1) NOT NULL,
- fk INT FOREIGN KEY REFERENCES t2,
- u VARCHAR(9) DEFAULT CURRENT\_USER,

)
**Unified Modeling Language (UML)****Complete:** Alle Subklassen sind definiert**Incomplete:** Zusätzliche Subklassen sind erlaubt**Disjoint:** Ist Instanz von genau einer Unterklasse**Overlapping:** Kann Instanz von mehreren überlappenden Unterklassen sein**Normalisierung****1NF:** Atomare Attributwerte: track aufteilen

id	track	→	id	interpret	titel
1	Fugazi: Song #1	→	1	Fugazi	Song #1

**2NF:** Nichtschlüsselattr. voll vom Schlüssel abhängig. Ist PK atomar, dann 2NF gegeben. Im Beispiel sind nicht alle Attribute des PK notwendig, um album eindeutig zu identifizieren

track	cd_id	album	titel
1	1	Repeater	Turnover
2	1	Repeater	Song #1

track	cd	cd
1	1	Turnover
2	1	Song #1

**3NF:** Keine transitiven Abhängigkeiten: land ist abhängig von interpret

id	album	interpret	land
1	Repeater	Fugazi	USA
2	Red Medicine	Fugazi	USA

**BCNF:** Nur Abhängigkeiten vom Schlüssel**(Voll-)funktionale Abhängigkeit:** B hängt von A ab, zu jedem Wert von A gibt es genau einen Wert von B ( $A \rightarrow B$ )**Teilweise funk. Abh.:** B hängt von A ab, aber auch von einem Teil eines zusammengesetzten Schlüssels.**Transitive Abhängigkeit:** B hängt vom Attribut A ab, C hängt von B ab ( $A \rightarrow B \wedge B \rightarrow C \Rightarrow A \rightarrow C$ )**Denormalisierung:** In geringere NF zurückführen (verbessert Performance und reduziert Joins-Komplexität)**Anomalien:** Einfügeanomalie, Löschanomalie, Änderungsanomalie**Data Control Language (DCL)**

```

<role> ::= 'ALTER ROLE' <rolename> '<priv> { ',' <priv> }'
<grant> ::= 'GRANT' <actions> 'ON' <object> 'TO' <grantees>
            [ 'WITH GRANT OPTION' ]
<revoke> ::= 'REVOKE' [ 'GRANT OPTION FOR' ] <actions> 'ON'
            <object> [ 'FROM' <grantees> ] [ 'CASCADE' | 'RESTRICT' ]
<priv> ::= [ 'NO' ] [ 'CREATEDB' | 'CREATEROLE', 'INHERIT' ]
<action> ::= [ 'ALL' | 'SELECT' | 'DELETE' | 'TRIGGER' ]
            [ 'INSERT' | 'UPDATE' | 'REFERENCES' ] [ '(' <columns> ')' ]
<actions> ::= <action> { ',' <action> }
<grantees> ::= <role_name> { ',' <role_name> }
<object> ::= 'TABLE' | 'COLUMN' | 'VIEW' | 'SEQUENCE' |
            'DATABASE' | 'FUNCTION' | 'SCHEMA'
    
```

Falls WITH GRANT OPTION: Der Berechtigte kann den Zugriff anderen Usern verteilen. → REVOKE ... CASCADE;

```

CREATE ROLE u WITH LOGIN PASSWORD '' ; -- user
GRANT INSERT ON TABLE t TO u WITH GRANT OPTION;
ALTER ROLE u CREATEROLE, CREATEDB, INHERIT;
CREATE ROLE r ; -- group
GRANT r TO u ; -- put user u in group r
REVOKE CREATE ON SCHEMA s FROM r;
CREATE ROLE u PASSWORD '' IN ROLE r ; -- equivalent
    
```

**Read-only user**

```

-- creating
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
CREATE ROLE u WITH LOGIN ENCRYPTED PASSWORD ''
    NOINHERIT; -- don't inherit privileges
GRANT SELECT ON ALL TABLES IN SCHEMA public TO u;
-- read all new tables (also created by others):
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT
    SELECT ON TABLES TO u;
-- deleting
REVOKE SELECT ON ALL TABLES IN SCHEMA public FROM u;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    REVOKE SELECT ON TABLES FROM u;
DROP USER u;
    
```

**Row-Level Security (RLS)**

```

CREATE TABLE exams (
    id SERIAL, -- other fields...
    teacher VARCHAR(60) DEFAULT current_user
);
CREATE POLICY teachers_see_own_exams ON exams
    FOR ALL TO PUBLIC USING (teacher = current_user);
ALTER TABLE exams ENABLE ROW LEVEL SECURITY;
    
```

**Data Definition Language (DDL)**

Wichtig: NOT NULL wo notwendig nicht vergessen

```

CREATE SCHEMA s;
CREATE TABLE t (
    id SERIAL PRIMARY KEY,
    name TEXT UNIQUE,
    grade DECIMAL(2,1) NOT NULL,
    fk INT FOREIGN KEY REFERENCES t2.id
        ON DELETE CASCADE,
    added TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    u VARCHAR(9) DEFAULT CURRENT_USER,
    CHECK (grade between 1 and 6)
);
    
```

**Updatable View**

Views sind updatable wenn diese Kriterien erfüllt sind:

- Eine einzige <base table>
- Keine aggregate, DISTINCT, GROUP BY, oder HAVING
- Alle Spalten müssen zur originalen Tabelle direkt gemappt werden können

**Materialized View**

Speichert Resultat auf Disk

```

CREATE MATERIALIZED VIEW mv AS SELECT * FROM t;
REFRESH MATERIALIZED VIEW mv; -- refresh results
    
```

**Temporäre Tabellen**

```

CREATE TEMPORARY TABLE temp_products (
    id SERIAL PRIMARY KEY,
    product_name TEXT
);
    
```

```

INSERT INTO temp_products (product_name) VALUES
    ('Product A'), ('Product B'), ('Product C');
    
```

```

SELECT ts.product_name, ts.quantity FROM
    temp_sales ts JOIN temp_products tp ON
    ts.product_name = tp.product_name;
    
```

**Datentypen**

```

CREATE TYPE grade AS ENUM('A','B','C','D','E','F');
NUMERIC(4, 2) /* 99.99 */ / NUMERIC(2, 1) /* 9.9 */;
VARCHAR(5) /* abcde */ / CHAR(5) /* abcde */;
    
```

**Dreiwertige Logik (cursed)**

```

SELECT NULL IS NULL; -- true
SELECT NULL = NULL; -- [unknown]
    
```

**Tabelle pro Subklasse:** Enthält jeweil. Subklassattribute

```

CREATE TABLE sub1 ( -- 3.b
    id SERIAL PRIMARY KEY,
    name TEXT UNIQUE,
    age INT
);
CREATE TABLE sub2 ( -- 3.c
    id SERIAL PRIMARY KEY,
    name TEXT UNIQUE
);
    
```

**Einzige Tabelle für Superklasse:** Enthält alle Attribute

```

CREATE TABLE sup ( -- 3.c
    id SERIAL PRIMARY KEY,
    name TEXT UNIQUE,
    age INT
);
    
```

**Junction Tabellen**

```

CREATE TABLE a_b(
    a INTEGER REFERENCES a(id),
    b INTEGER REFERENCES b(id),
    PRIMARY KEY(a, b)
);
    
```

**Views**

Resultate werden jedes mal dynamisch queriert

```

CREATE VIEW v (id, u) AS SELECT id, u FROM t;
-- complex query
CREATE VIEW cheap_restaurant_view AS
    WITH big_restaurant AS (
        SELECT * FROM restaurant
        WHERE anzahl_plaetze >= 20
    )
    SELECT r.name AS restaurant_name, s.name,
        MIN(g.preis) AS cheap_gericht
    FROM big_restaurant r
    LEFT JOIN skiegibet s ON (s.id = r.skigebiet_id)
    LEFT JOIN menukarte m ON (r.id = m.restaurant_id)
    LEFT JOIN menu_gericht mg ON (m.id = mg.menu_id)
    LEFT JOIN gericht g ON (g.id = mg.gericht_id)
    WHERE ist_tagesmenu = true
    GROUP BY r.id, s.id, restaurant_name
    HAVING MIN(g.preis) > 3
    ORDER BY cheap_gericht;
    
```

**Updatable View**

Views sind updatable wenn diese Kriterien erfüllt sind:

- Eine einzige <base table>
- Keine aggregate, DISTINCT, GROUP BY, oder HAVING
- Alle Spalten müssen zur originalen Tabelle direkt gemappt werden können

**Materialized View**

Speichert Resultat auf Disk

```

CREATE MATERIALIZED VIEW mv AS SELECT * FROM t;
REFRESH MATERIALIZED VIEW mv; -- refresh results
    
```

**Temporäre Tabellen**

```

CREATE TEMPORARY TABLE temp_products (
    id SERIAL PRIMARY KEY,
    product_name TEXT
);
    
```

```

INSERT INTO temp_products (product_name) VALUES
    ('Product A'), ('Product B'), ('Product C');
    
```

```

SELECT ts.product_name, ts.quantity FROM
    temp_sales ts JOIN temp_products tp ON
    ts.product_name = tp.product_name;
    
```

**Datentypen**

```

CREATE TYPE grade AS ENUM('A','B','C','D','E','F');
NUMERIC(4, 2) /* 99.99 */ / NUMERIC(2, 1) /* 9.9 */;
VARCHAR(5) /* abcde */ / CHAR(5) /* abcde */;
    
```

**Dreiwertige Logik (cursed)**

```

SELECT NULL IS NULL; -- true
SELECT NULL = NULL; -- [unknown]
    
```

**users (u)****actions (a)****INFO: FK uid in den Query-Resultaten unten aus Platzgründen ausgelassen**

Typ	Beschreibung
INTEGER/INT	Integer (4 bytes)
BIGINT	Large integer (8 bytes)
SMALLINT	Small integer (2 bytes)
REAL	Single precision float (4 bytes)
NUMERIC(precision, scale)	Exact numeric of selectable precision Alias for DECIMAL(precision, scale)
DOUBLE PRECISION	Double precision float (8 bytes)
SERIAL	Auto-incrementing integer (4 bytes)
BIGSERIAL	Auto-incrementing large integer (8 bytes)
SMALLSERIAL	Auto-incrementing small integer (2 bytes)
CHARACTER/CHAR(size)	Fixed-length, blank-padded string
VARCHAR(size)	Variable-length, non-blank-padded string
TEXT	Variable-length character string
BOOLEAN	Logical Boolean (true/false)
DATE	Calendar date (year, month, day)
TIME	Time of day (no time zone)
TIMESTAMP	Date and time (no time zone)
TIMESTAMP WITH TIME ZONE	Date and time with time zone
INTERVAL	Time interval
JSON	JSON data
UUID	Universally unique identifier
ARRAY OF base_type	Array of values

Casting	Explizit
CAST(s AS float8) = 5::float8	
SELECT 'ABCDEF'::NUMERIC; -- error	
SELECT SAFE_CAST('ABCDEF' AS NUMERIC); -- NULL	
Implizit	
SELECT 5 + 3.2; -- 5 is cast to 5.0 (numeric)	
SELECT 'Number'    42; -- 42 is cast to '42'	
SELECT true AND 1; -- 1 is treated as true	
SELECT CURRENT_TIMESTAMP + INTERVAL '1 day'; -- CURRENT_TIMESTAMP to date ^	
SELECT '100'::text + 1; -- '100' is cast to 100	

Data Manipulation Language (DML)	
<select> ::= [ 'WITH' [ 'RECURSIVE' ] <with_query> [ ',' ... ] ]	
'SELECT' [ 'ALL'   'DISTINCT' [ 'ON' ( <expression> [ ',' ... ] ) ] ]	
[ '*'   <expression> [ [ 'AS' ] <output_name> ] ]	
[ 'WHERE' [ <condition> ] ]	
[ 'GROUP BY' [ 'ALL'   'DISTINCT' ] <grouping_elem> [ ',' ... ] ]	
[ 'HAVING' <condition> ]	
[ 'WINDOW' <window_name> 'AS' ( <>window_def> [ ',' ... ] ) ]	
[ { 'UNION'   'INTERSECT'   'EXCEPT' } [ 'ALL'   'DISTINCT' ] <select> ]	
[ 'ORDER BY' <expression> [ 'ASC'   'DESC'   'USING' <op> ] ]	
[ 'NULLS' { 'FIRST'   'LAST' } [ ',' ... ] ]	
[ 'LIMIT' <count> [ 'ROWS'   'OFFSET' <start> [ 'ROW'   'ROWS' ] ] ]	
<from_item> ::= <table> [ '*'   [ 'AS' ] <alias> [ ',' ... ] ]	
[ <col_alias> [ ',' ... ] ]	
[ 'LATERAL' ] [ <select> ] [ [ 'AS' ] <alias> [ ( <col_alias> [ ',' ... ] ) ] ]	
[ <with_query_name> [ [ 'AS' ] <alias> [ ( <col_alias> [ ',' ... ] ) ] ] ]	
<from_item> <join_type> <from_item> { 'ON' <join_condition>   'USING' ( <join_column> [ ',' ... ] )   'AS' <join_using_alias> } [ <from_item> <join_type> <from_item> ]	
[ <from_item> 'CROSS JOIN' <from_item> ]	
<with_query> ::= <name> [ ( <col_name> [ ',' ... ] ) ]	
'AS' ( <select>   <values>   <insert>   <update>   <delete> )	
[ 'USING' <cycle_path_col_name> ]	

users (u)	actions (a)
id	uid
1	1
2	2
3	3
4	4
5	5

**Inner Join**

Zeilen, die in beiden Tabellen  
matchen

1	1	Alice	7	LOGIN
---	---	-------	---	-------

**Subqueries**

```
SELECT * FROM t WHERE grade > ANY
  (SELECT g FROM t2);
SELECT * FROM t WHERE EXISTS (SELECT g FROM t2);
-- ALL, ANY, IN, EXISTS, =
```

**GROUP BY**

```
SELECT id, COUNT(*) FROM t
  GROUP BY grade, id HAVING COUNT(*) > 2;
```

**WHERE**

```
BETWEEN 1 AND 5; LIKE '__%'; AND; IS (NOT) NULL
IN (1, 5); LIKE '%asd'; OR;
```

**Natural Join**

Wie Inner Join aber ohne Duplikate

1	1	Alice	7	LOGIN
---	---	-------	---	-------

**Aggregatfunktionen**

```
COUNT; SUM; MIN; MAX; AVG
```

**Semi Join**

Nur Zeilen aus a, wobei b matchen muss

1	1	Alice
---	---	-------

**Weitere Funktionen**

```
COALESCE(a1, a2, ...); -- returns first non-null arg
```

**Common Table Expressions (CTE)**

- Erlauben die zeilenweise Ausgabe
- Erlauben Abfragen quasi als Parameter
- Können rekursiv sein

-- normal

```
WITH cte AS (SELECT * FROM t) SELECT * FROM cte;
WITH tmp(id, name) AS (SELECT id, name FROM t)
  SELECT id, name FROM tmptable;
```

-- rekursiv

```
WITH RECURSIVE q AS (
  SELECT * FROM t WHERE grade > 1
  UNION ALL SELECT * FROM t INNER JOIN q ON
    q.u = t.name
) SELECT id as 'ID' FROM q;
```

**Left outer Join**

Alle Zeilen beider Tabellen, NULL  
für b falls kein match

1	1	Alice	7	LOGIN
---	---	-------	---	-------

**Window Functions**

```
SELECT id, RANK() OVER
  (ORDER BY grade DESC) as r FROM t;
SELECT id, u, LAG(name, 1) OVER
  (PARTITION BY fk ORDER BY id DESC) FROM t;
-- PERCENT/DENSE_RANK(), FIRST_VALUE(v),
LAST_VALUE(n)
-- NTH_VALUE(v,n), NTILE(n), LEAD(v,o), ROW_NUMBER()
```

**Full outer Join**

Alle Zeilen beider Tabellen, NULL  
falls kein match

1	1	Alice	7	LOGIN
---	---	-------	---	-------

**SELECT u.\* FROM u FULL**

OUTER JOIN a ON u.id = a.uid;

2	2	Bob
---	---	-----

**Cross Join**

Liefert alle möglichen Kombinationen zweier Tabellen.

```
SELECT * FROM u CROSS JOIN a;
INDEX
```

1	1	Alice	7	LOGIN
---	---	-------	---	-------

**Union**

«Verbindet» zwei SELECT's ohne Duplikate.  
Voraussetzung: Spalten müssen ähnliche  
Datentypen beinhalten. **Union all** ist wie Union,  
nur mit duplikaten → Rekursive CTEs

2	2	Bob
---	---	-----

SELECT name FROM u UNION SELECT action FROM a;

**Lateral Join**

Erlaubt Subqueries mit Referenzen zu den  
anderen Tabellen

1	1	Alice	7	LOGIN
---	---	-------	---	-------

Note: In postgres gibt es keine geschachtelten T.

**Atomicity:** Vollständig oder gar nicht

**Consistency:** Konsistenter Zustand bleibt erhalten

**Isolation:** Transaktion ist von anderen T isoliert

**Durability:** Änderungen sind persistent

```
BEGIN; SAVEPOINT s;
COMMIT; ROLLBACK /*TO SAVEPOINT s*/;
```

**INSERT**

```
INSERT INTO t (added, grade)
VALUES ('2002-10-10', 1) RETURNING id;
```

**UPDATE**

```
UPDATE t SET grade = grade+1, name='`' WHERE id = 1;
```

**Isolation**

```
SET TRANSACTION ISOLATION LEVEL ...; -- transaction
SET SESSION CHARACTERISTICS AS TRANSACTION
ISOLATION LEVEL ...; -- session
```

**READ UNCOMMITTED:** Lesezugriffe nicht synchronisiert  
(keine Read-lock), Read ignoriert jegliche Sperren

**READ COMMITTED:** Lesezugriffe nur kurz/temporär syn-  
chronisiert (default), setzt für gesamte T Write-Lock, Read-  
lock nur kurzfristig

**REPEATABLE READ:** Einzelne Zugriffe ROWS sind syn-  
chronisiert, Read und Write Lock für die gesamte T

**SERIALIZABLE:** Vollständige Isolation nach ACID

	Read Un-committed	Read Committed	Repeata-ble Read	Seria-lizable
Dirty Write	.	.	.	X
Dirty Read	✓	X	X	X
Lost Update	✓	✓	X	X
Fuzzy Read	✓	✓	X	X
Phantom Read	✓	✓	✓	X
Read Skew	✓	✓	X	X
Write Skew	✓	✓	✓	.
Dauerhaftigkeit	✓	✓	X	X
Atomizität	X	X	✓	✓

\* Nur in SQL92 möglich, PSQL >= 9.1 verhindert dies

**Dirty Read:** Lese Daten von nicht committed T's

**Fuzzy Read:** Versch. Werte beim mehrmaligen Lesen gleicher Daten (da durch andere T geändert)

**Phantom Read:** Neue/Gelöschte Rows einer anderen T

**Read Skew:** Daten lesen, die sich während der T ändern

**Write Skew:** Mehrere T lesen Daten und Ändern sie

**Deadline:** Mehrere T blockieren sich, da sie auf die gleiche Ressource warten

**Cascading Rollback:** T schlägt fehl und alle davon abhängigen T müssen ebenfalls zurückgerollt werden

**Two-Phase Locking (2PL)**

Stellt Isolation der T sicher

1) Growing Phase: Die T kann neue Locks erwerben, jedoch keine freigeben

2) Shrinking Phase: Locks können freigegeben werden, aber keine neuen mehr erworben werden

**Strict 2PL:** T geben locks erst nach commit frei

**Precaliming 2PL:** Alle Locks werden zu Beginn der T erstellt

**Shared Lock:** Lesezugriffe (mehrere Transaktionen)

**Exclusive Lock:** Schreib- & Lesezugriffe (eine Transaktion)

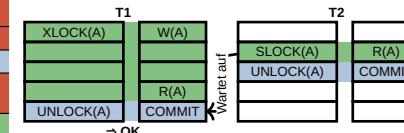
**Starvation:** T erhält aufgrund von Sperren niemals die Mög-  
lichkeit, ihre Arbeit abzuschließen, da immer blockiert wird

**Optimistic Lockverfahren**

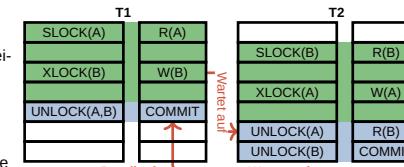
T operieren ohne anfängliche Sperren. Überprüfen am Ende falls Konflikte auftreten → Änderungen zurücksetzen.

**Pessimistisches Lockverfahren (Precaliming 2PL)**

T fordert sofort Sperren an, damit andere T nicht gleichzeitig auf dieselben Daten zugreifen oder diese ändern.

**Growing phase****Shrinking phase**

⇒ OK



⇒ Deadlock

Warte auf  
Warte auf

W