**TODO: glossar****DataBase System (DBS)**

Besteht aus DBMS und Datenbasen

 **DataBase Management System (DBMS)**

- (A) Transaktionen
- (C) Konsistenz
- (I) Mehrbenutzerbetrieb
- (D) Grosse Datensmengen
- (S) Sicherheit
- Datentypen
- Abfragesprache
- Backup & Recovery
- Redundanzfreiheit
- Kapselung

**ANSI Model**

Logische Ebene: Logische Struktur der Daten

Interne Ebene: Speicherstrukturen, Definition durch internes Schema (Beziehungen, Tabellen etc.)

Externe Ebene: Sicht einer Benutzerklasse auf Teilmenge der DB, Definition durch externes Schema

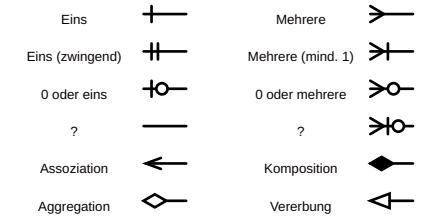
Mapping: Zwischen den Ebenen ist eine mehr oder weniger komplexe Abbildung notwendig

**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

**TODO: better examples**

<i>id</i>	<i>full_name</i>	⇒	<i>id</i>	<i>first</i>	<i>last</i>
1	First Last		1	First	Last

2NF: Nichtschlüsselattr. voll vom Schlüssel abhängig

<i>track</i>	<i>title</i>	<i>cd_id</i>	<i>album</i>
1	Turnover	1	Repeater
2	Repeater	1	Repeater

⇒ track			cd	
<i>track</i>	<i>cd_id</i>	<i>title</i>	<i>id</i>	<i>album</i>
1	1	Turnover	1	Repeater
2	1	Repeater	1	Repeater

3NF: Keine transitiven Abhängigkeiten

```
);
CREATE TABLE sub2 (
    id SERIAL PRIMARY KEY
);
ALTER TABLE sub1 ADD CONSTRAINT id FOREIGN KEY
    REFERENCES sup (id);
ALTER TABLE sub2 ADD CONSTRAINT id FOREIGN KEY
    REFERENCES sup (id);
```

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

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

**Datentypen**

```
SMALLINT INT INTEGER BIGINT REAL FLOAT
DOUBLE NUMERIC(precision,scale) DECIMAL(p,s)
VARCHAR(size) TEXT CHAR(size) -- fixed size
DATETIME DATE INTERVAL TIME BINARY
CLOB /*Char Large Objects*/ BLOB VARBINARY
```

**Casting**

```
Explicit
CAST(5 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
```

**Views**

Resultate werden jedes mal dynamisch queried

```
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 skigebiet 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_tagesmenue = 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:

- Single base table

- Keine aggregate, DISTINCT, GROUP BY, oder HAVING Klauseln  
- 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;
```

**Data Manipulation Language (DML)**

```
FROM -> JOIN -> WHERE -> GROUP BY -> HAVING ->
    SELECT (WINDOW FUNCTIONS) -> ORDER BY -> LIMIT
```

**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 tmp;
-- recursive
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;
```

**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()
```

**INSERT**

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

**UPDATE**

```
UPDATE t SET grade = grade+1, name=''
```

**Subqueries**

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

**Inner Join**

Zeilen, die in beiden Tabellen matchen

```
SELECT a.* , b.* FROM a INNER JOIN b ON a.id = b.id;
```

**Equi Join**

Wie Inner Join

```
SELECT a.* , b.* FROM a JOIN b ON a.id = b.id;
```

**Natural Join**

Wie Inner Join aber ohne Duplikate

```
SELECT a.* , b.* FROM a NATURAL JOIN b ON a.id =
b.id;
```

**Semi Join**

Nur Zeilen aus a, wobei b matchen muss

```
SELECT a.* FROM a WHERE EXISTS
    (SELECT 1 FROM b WHERE a.id = b.id);
```

**Anti Join**

Nur Zeilen aus a, wobei b nicht matchen darf

```
SELECT a.* FROM a WHERE NOT EXISTS
    (SELECT 1 FROM b WHERE a.id = b.id);
```

**Left outer Join**

Alle Zeilen beider Tabellen, NULL für b falls kein match

```
SELECT a.* , b.* FROM a LEFT OUTER JOIN b ON
a.id=b.id;
```

**Right outer Join**

Alle Zeilen beider Tabellen, NULL für a falls kein match

```
SELECT a.* , b.* FROM a RIGHT OUTER JOIN b ON
a.id=b.id;
```

**Full outer Join**

Alle Zeilen beider Tabellen, NULL falls kein match

```
SELECT a.* , b.* FROM a FULL OUTER JOIN b ON
a.id=b.id;
```

**Lateral Join**

Join, der Subqueries erlaubt

```
SELECT x.* , y.* FROM a AS x JOIN LATERAL
    (SELECT * FROM b WHERE b.id = y.id) AS y ON TRUE;
```

**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;
```

**Aggregatfunktionen**

COUNT ; SUM ; MIN ; MAX ; AVG

**Weitere Funktionen**

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

**INDEX**

	B-Tree	Hash	BRIN
Gleichheitsabfragen	✓	✗	✗
Range Queries	✓	✗	✓
Sortierte Daten	✓	✗	✓
Große Tabellen	*	*	✓

\* Hash: Nur bei Gleichheitsabfragen

```
CREATE INDEX i ON t/*USING BTREE*/(grade,upper(u));
CREATE INDEX j ON t (fk) INCLUDE (added) WHERE fk>4;
DROP INDEX i;
```

**Transaktionen**

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\*/;

**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 synchronisiert (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	Serializable
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	✓	✓	✓	*

\* Nur in SQL92 möglich, PostgreSQL >= 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

**Deadlock:** 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

	Seriali-sierbar	Dead-locks	Cas-cading Roll.	Kon-flikt-Roll.	Hohe Paral-lelität	Realis-tisch
Two-Phase Locking	✓	✓	✓	X	X	X
Strict 2PL	✓	✓	X	X	X	✓
Precalim-ing 2PL	✓	X	X	X	X	X
Validation-based	✓	X	✓	✓	✓	✓
Timestamp-based	X	✓	✓	✓	✓	✓
Snapshot Isolation	X	*	X	✓	✓	✓
SSI	✓	*	X	✓	✓	✓

\* Deadlock in PostgreSQL mit Snapshot Isolation

### SQL Beispiel

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

### Relationale Algebra

$\pi_{R1,R4}(R)$  SELECT R1,R4 FROM R; (Projektion)

$\sigma_{R1>30}(R)$  SELECT \* FROM R WHERE R1 > 30; (Selektion)

$\rho_{a \leftarrow R}$  SELECT \* FROM R AS a; (Umbenennung/Alias)

$R \times S$  SELECT \* FROM R,S; (Kartesisches Produkt)

$R \bowtie_{A=B} S$  SELECT \* FROM R JOIN S ON R.A=S.B; (Verbund)

### Serialisierbarkeit

Shared Lock: Schreib- & Lesezugriffe (eine Transaktion)

Exclusive Lock: Lesezugriffe (mehrere Transaktionen)

Serieller Schedule: Führt Transaktionen am Stück aus

Nicht serialisierbar:

$S1=R1(x) R2(x) W1(y) R1(y) W2(x) W1(y)$



### Konfliktpaare:

$R1(x) < W2(x)$        $R2(x) < W1(x)$

### Konflikt-Serialisierbar:

$r1(b)r2(b)r2(c)r2(d)w3(a)r4(d)r3(b)w4(d)r5(c)r5(a)w4(c)$

### Konflikt-Äquivalenter serieller Schedule:

$r1(b)r2(b)w2(b)r2(c)r2(d)w3(a)r3(b)r5(c)r5(a)r4(d)w4(d)w4(c)$



### Vollständiges Backup

Exakte Kopie der ganzen DB

### Inkrementelles Backup

Sichert nur die seit dem letzten Backup geänderten Daten.

### Logisches Backup (SQL Dump)

Blockiert keine T. Für mittelmäßige Datenmengen, interkompatibel mit neuen PG-Versionen und anderen Maschinen.

### Physisches Backup (File System)

Datenbank muss gestoppt werden, schneller als logisches Backup, passt nur zu derselben «Major Version» von PG.

### Multi-Version Concurrency Control (MVCC)

Ermöglicht es, mehreren T gleichzeitig zu laufen. Bei jeder Änderung wird eine neue Version der Daten erstellt. Leser sehen die älteren Versionen, während Schreiber die neuesten Versionen sehen.

### Two-Phase Locking (2PL)

#### TODO: example

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

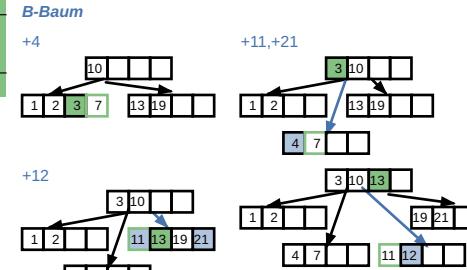
### Write-Ahead Log (WAL)

Schreibt Änderungen der T in Log, dann Commit loggen, dann Updates in DB. Kann bei Absturz replayed werden

### Dreiwertige Logik (cursed)

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

### B-Baum



### SQL Beispiele

```
CREATE TABLE pferd (
    pnr SERIAL PRIMARY KEY,
    name TEXT,
    alter INT,
    zuechternr INT REFERENCES stall.pk,
    vatennr INT REFERENCES pferd.pk
);

CREATE TABLE stall (
    zuechternr SERIAL PRIMARY KEY,
    name TEXT,
    plz INT,
    ort TEXT,
    strasse TEXT
);
```

-- Welche Züchter haben in ihren Ställen mindestens 1 Kind von dem Vater mit Namen "Hermes"