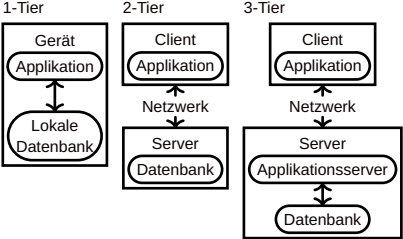


Glossar

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.
Objektorientiert	Kombiniert objektorientierte + relationale Prinzipien
Relational	Speichert Daten in Tabellen (Relationen) und verwaltet Beziehungen durch Schlüssel



DataBase System (DBS)

Besteht aus DBMS und Datenbasen

DataBase Management System (DBMS)

- (A) Transaktionen

– (C) Konsistenz

– (I) Mehrbenutzerbetrieb

– (D) Grosse Datenmengen

– (D) Sicherheit
- Datentypen

– Abfragesprache

– Backup & Recovery

– Redundanzfreiheit

– Kapselung

ANSI Modell

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 i2,
u VARCHAR(9) DEFAULT CURRENT_USER,
);

Unified Modeling Language (UML)

Eins	+	Mehrere	➤
Eins (zwingend)	++	Mehrere (mind. 1)	➤
0 oder eins	0..1	0 oder mehrere	0..➤
Assoziation	◀	Komposition	◆
Aggregation	◊	Vererbung	◄

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
1	Fugazi: Song #1

 ⇒

id	interpret	titel
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

track	cd_id	titel
1	1	Turnover
2	1	Song #1

cd

id	album
1	Repeater

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

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

⇒ cd

id	album	interpret	id	name	land
1	Repeater	1	1	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 → B**)

Teilweise funkt. 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 → B ∧ B → C ⇒ A → C**)

Denormalisierung: In geringere NF zurückführen (Verbessert Performance und reduziert Joins-Komplexität)

Anomalien

Einfügeanomalie, Löschanomalie, Änderungsanomalie

Data Control Language (DCL)

```
<role> ::= 'ALTER ROLE' <name> <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' )  
<actions> ::= '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 TIMESTAMPT DEFAULT CURRENT_TIMESTAMP,  
u VARCHAR(9) DEFAULT CURRENT_USER,  
CHECK (grade between 1 and 6)  
);
```

```
ALTER TABLE t2 ADD CONSTRAINT c PRIMARY KEY (a, b);  
TRUNCATE/DROP TABLE t;
```

Vererbung

Tabelle pro Sub- und Superklasse:

```
CREATE TABLE sup ( -- 3.a  
id SERIAL PRIMARY KEY,  
name TEXT UNIQUE  
);  
CREATE TABLE sub1 (  
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. Subklassenattribute

```
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)  
);
```

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_platze ≥ 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_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]
```

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(5 AS float8) = 5::float8  
SELECT 'ABCDEF6'::NUMERIC; -- error  
SELECT SAFE_CAST('ABCDEF6' 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> ] }  
[ ',' ... ] ]  
{ 'FROM' <from_item> [ '(' ... ) ]  
[ '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> | 'ALL' } ]  
[ '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> 'NATURAL' <join_type> <from_item>  
<from_item> 'CROSS JOIN' <from_item>
```

```
<with_query> ::= <name> [ '(' <col_name> [ ',' ... ] ) ]  
'AS' { <select> | <values> | <insert> | <update> | <delete> |  
<merge> }  
[ 'USING' <cycle_path_col_name> ]
```

users (*u*) actions (*a*)

id	name	id	uid	action
1	Alice	7	1	LOGIN
2	Bob	8	2	VIEW
		9	4	LOGIN

INFO: FK *uid* in den Query-Resultaten unten aus Platzgründen ausgelassen

Georgiy Shevoroshkin

Inner Join

Zeilen, die in beiden Tabellen matchen

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

Equi Join

Wie Inner Join

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

Natural Join

Wie Inner Join aber ohne Duplikate

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

Semi Join

Nur Zeilen aus a, wobei b matchen muss

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

Anti Join

Nur Zeilen aus a, wobei b nicht matchen darf

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

Left outer Join

Alle Zeilen beider Tabellen, NULL für b falls kein match

```
SELECT u.*, a.* FROM u LEFT JOIN a ON u.id = a.uid;
```

Right outer Join

Alle Zeilen beider Tabellen, NULL für a falls kein match

```
SELECT u.*, a.* FROM u RIGHT JOIN a ON u.id = a.uid;
```

Full outer Join

Alle Zeilen beider Tabellen, NULL falls kein match

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

Cross Join

Liefert alle möglichen Kombinationen zweier Tabellen.

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

Union

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

```
SELECT name FROM u UNION SELECT action FROM a;
```

Lateral Join

Erlaubt Subqueries mit Referenzen zu den anderen Tabellen

```
SELECT u.*, x.action FROM u JOIN LATERAL (SELECT * FROM a WHERE a.uid ≠ u.id) AS x ON TRUE;
```

INSERT

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

UPDATE

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

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 ;
```

Aggregatfunktionen

COUNT ; SUM ; MIN ; MAX ; AVG

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 tmp table; -- 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;
```

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()
```

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)
INDEX

	B-Tree	Hash	BRIN	ISAM
Gleichheitsabfragen	✓	✓	X	✓
Range Queries	✓	X	✓	X
Sortierte Daten	✓	X	✓	✓
Grosse Tabellen	+	bei =	✓	✓
Häufige abfragen	✓	+	✓	X
Direkter zugriff über PK	✓	✓	X	+
Überlaufseiten	✓	✓	X	✓

```
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 synchronisiert, Read und Write Lock für die gesamte T

SERIALIZABLE: Vollständige Isolation nach ACID

	Read Un-committed	Read Com-mitted	Repeata-ble Read	Seriali-zable
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	✓	✓	✓	+
Dauerhaft-igkeit	✓	✓	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

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 RollB.	Kon-flikt-RollB.	Hohe Paral-lelität	Realis-tisch
Two-Phase Locking	✓	✓	✓	X	X	X
Strict 2PL	✓	✓	X	X	X	✓
Preclaiming 2PL	✓	X	X	X	X	X
Validation-based	✓	X	✓	✓	✓	✓
Time-stamp-based	✓	X	✓	✓	✓	✓
Snapshot Isolation	X	+	X	✓	✓	✓
SSI	✓	+	X	✓	✓	✓

* Deadlock in PSQL 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;
```

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

Preclaiming 2PL: Alle Locks werden zu Beginn der T erstellt

Shared Lock: Lesezugriffe (mehrere Transaktionen)

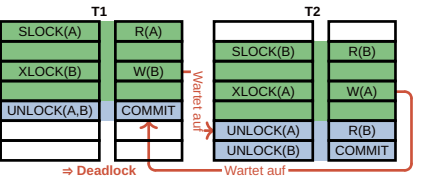
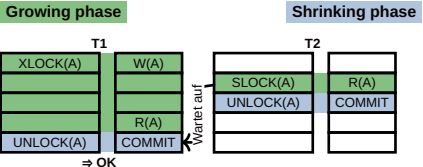
Exclusive Lock: Schreib- & Lesezugriffe (eine Transaktion)

Starvation: T erhält aufgrund von Sperren niemals die Möglichkeit, ihre Arbeit abzuschliessen, da T immer blockiert wird
Optimistisches Lockverfahren

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

Pessimistisches Lockverfahren (Preclaiming 2PL)

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



Serialisierbarkeit

Serieller Schedule: Führt Transaktionen am Stück aus
Nicht serialisierbar:

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

Konfliktpaare:

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

Konflikt-Serialisierbar:

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

Konflikt-Äquivalenter serieller Schedule:

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



Begriff	Bedeutung
Seriell	Alle T in einem Schedule sind geordnet
Konfliktäquivalent	Reihenfolge aller Paare von konfligierenden Aktionen ist in beiden Schedules gleich
Konfliktserialisierbar	Ein S ist konfliktäquivalent zu einem seriellen S

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 mittelgrosse 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)

Ermöglich 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.

Write-Ahead Log (WAL)

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

LSN, TaId, PageId, Redo, Undo, PrevLSN

SQL Beispiele

```
CREATE TABLE pferd ( pnr SERIAL PRIMARY KEY, name TEXT, alter INT, zuechternr INT REFERENCES stall.pk, vaternr 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"

-- Eleganteste anfrage unkorreliert

```
SELECT s.name FROM staele s WHERE s.zuechternr IN ( SELECT p.zuechternr FROM pferde p JOIN pferde p2 ON p2.pnr = p.vaternr WHERE p2.name = 'Hermes' );
```

-- Kürzeste anfrage

```
SELECT DISTINCT s.name FROM staele s JOIN pferde p ON p.zuechternr = s.zuechternr JOIN pferde p2 ON p2.pnr = p.vaternr WHERE p2.name = 'Hermes';
```

```
-- SELECT DISTINCT s.name FROM staele s JOIN pferde p ON p.zuechternr = s.zuechternr WHERE EXISTS ( SELECT vaternr FROM pferde p2 WHERE p2.pnr = p.vaternr AND p2.name = 'Hermes' );
```

-- RECURSIVE CTE

```
WITH RECURSIVE tens AS ( SELECT 1 as n UNION ALL SELECT n+1 FROM tens ) SELECT n FROM tens limit 10; -- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

B-Baum

