



**Georgiy Shevoroshkin****Window Functions**

```
SELECT id, RANK() OVER
  (ORDER BY grade DESC) as r FROM t;
SELECT id, u, LAG(name) 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='` WHERE id = 1;
```

**Subqueries**

```
SELECT * FROM t WHERE grade > ANY (SELECT g FROM t2);
SELECT * FROM t WHERE EXISTS (SELECT g FROM t2);
```

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

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

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

**Cross Join**

Liefert alle möglichen Kombinationen zweier Tabellen.

1	1	Alice	7	LOGIN
2	1	Alice	8	VIEW
3	1	Alice	9	LOGIN
4	2	Bob	7	LOGIN
5	2	Bob	8	VIEW
6	2	Bob	9	LOGIN

**Union**

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

1	Alice
2	Bob
3	LOGIN
4	VIEW

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

**Lateral Join**

Erlaubt Subqueries mit Referenzen zu den anderen Tabellen

1	LOGIN
---	-------

```
SELECT u.* , x.action FROM u JOIN LATERAL
  (SELECT * FROM a WHERE a.uid = u.id)
  AS x 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
```

**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)

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

$A=B$  Dreiwertige Logik (cursed)

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

**INDEX**

	B-Tree	Hash	BRIN	ISAM
Gleichheitsabfragen	✓	✓	✗	✓
Range Queries	✓	✗	✓	✗
Sortierte Daten	✓	✗	✓	✓
Große Tabellen	*	bei =	✓	✓
Häufige abfragen	✓	*	✓	✗
Direkte Zugriff über PK	✓	✓	✗	*
Überlaufsuchen	✓	✓	✗	✓

```
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 Uncommitted	Read Committed	Repeatable Read	Serizable
Dirty Write	*	*	*	✗
Dirty Read	✓	✗	✗	✗
Lost Update	✓	✓	✗	✗
Fuzzy Read	✓	✓	✗	✗
Phantom Read	✓	✓	✓	✗
Read Skew	✓	✓	✗	✗
Write Skew	✓	✓	✓	*
Dauerhaftigkeit	✓	✓	✗	✗
Atomizität	✗	✗	✓	✓

\* Nur in SQL92 möglich, PSQL  $\geq 9.1$  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

Serialisierbar	Deadlocks	Cascading Rollb.	Konflikt-Rollb.	Hohe Parallelität	Realistisch
Two-Phase Locking	✓	✓	*	*	✗
Strict 2PL	✓	✗	✗	✗	✓
Precalming 2PL	✓	✗	✗	✗	✗
Validation-based	✓	✗	✓	✓	✓
Timestamp-based	✓	*	✓	✓	✓
Snapshot Isolation	*	*	✗	✓	✓
SSI	✓	*	✗	✓	✓

\* 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

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

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

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

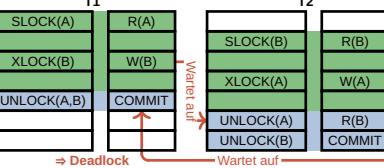
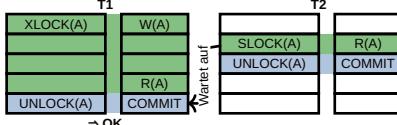
**Starvation:** T erhält aufgrund von Sperren niemals die Möglichkeit, ihre Arbeit abzuschließen, da immer blockiert wird

**Optimistic Lockverfahren**

T operieren ohne anfängliche Sperren. Überprüfen am Ende falls Konflikte auftreten  $\rightarrow$  Änderungen zurücksetzen.

**Pessimistisches Lockverfahren** (Precalming 2PL)

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

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

**Serieller Schedule:** Führt Transaktionen am Stück aus  
Nicht serialisierbar:

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

**Konfliktpaar:**

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

**Konflikt-Serialisierbar:**

$r1(b)2(b)w2(b)r2(c)r2(d)w3(a)r4(d)r3(b)w4(d)r5(c)r5(a)w4(c)$

**Konflikt-Äquivalenter serieller Schedule:**

$r1(b)2(b)w2(b)r2(c)r2(d)w3(a)r3(b)r5(c)r5(a)r4(d)w4(d)w4(c)$

**Begriff**

**Seriell:** Alle T in einem Schedule sind geordnet

**Konfliktäquivalent:** Reihenfolge aller Paare von konfigurierenden Aktionen ist in beiden Schedules gleich

**Konfliktserialisierbar:** Ein S Backslash a regestix 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)**

Ermöglicht es, mehrere T gleichzeitig zu laufen. Bei jeder Änderung wird eine neue Version der Daten erstellt. Leser sehen die älteren Versionen, während Schreiber die neueren 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 staelle 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 staelle s
```

```
JOIN pferde p ON p.zuechternr = s.zuechternr
```

```
JOIN pferde p2 ON p2.pnr = p.vaternr
```

```
WHERE 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**