

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);
```

```
-- ALL, ANY, IN, EXISTS, =
```

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

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

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

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

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

Natural Join

Wie Inner Join aber ohne Duplikate

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

TODO:**Semi Join**

Nur Zeilen aus a, wobei b matchen muss

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

```
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

1	2	Bob
---	---	-----

```
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

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

```
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

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

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

Full outer Join

Alle Zeilen beider Tabellen, NULL
falls kein match

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

```
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;
```

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

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

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
```

Relationale Algebra

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

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

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

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

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

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 =	✓	✓
Häufig abfragen	✓	*	✓	✗
Direkter zugriff über PK	✓	✓	✗	*
Überlaufseiten	✓	✓	✗	✓

```
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	✗	✗	✓	✓

Serieller Schedule: Führt Transaktionen am Stück aus
Nicht serialisierbar:

**Konfliktpaare:**

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

Konflikt-Serialisierbar:

$r1(b)2(b)w2(b)2(c)r2(d)w3(a)r4(d)3(b)w4(d)r5(c)r5(a)w4(c)$

Konflikt-Äquivalenter serieller Schedule:

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

Begriff	Bedeutung
Seriell	Alle T in einem Schedule sind geordnet
Konfliktäquivalent	Reihenfolge aller Paare von konfigurierenden Aktionen ist in beiden Schedules gleich
Konfliktserialisierbar	Ein S Backslash a regxit 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 sich selbst «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 neusten 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

Optimistisches Lockverfahren

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

Pessimistisches Lockverfahren

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

Write-Ahead Log (WAL)

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

LSN, Taid, PagId, Redo, Undo, PrevLSN**SQL Beispiele**

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

 WHERE p2.name = 'Hermes'

)

-- Kürzeste anfrage

SELECT DISTINCT s.name FROM staelle s

JOIN pferde p ON p.zuechternr = s.zuechternr

WHERE EXISTS (

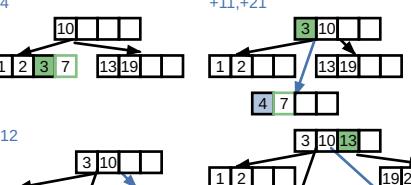
 SELECT veternnr FROM pferde p2

 WHERE p2.pnr = p.veternnr AND p2.name = 'Hermes'

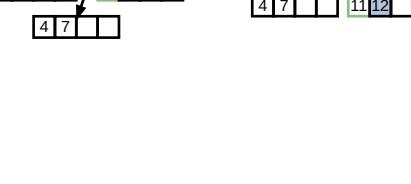
)

B-Baum

+4



+12



+11, +21

