Prof. Dr.-Ing. Johannes Schildgen
johannes.schildgen@oth-regensburg.de

Datenbanken

Kapitel 5: Mehrbenutzerbetrieb

OTH REGENSBURG

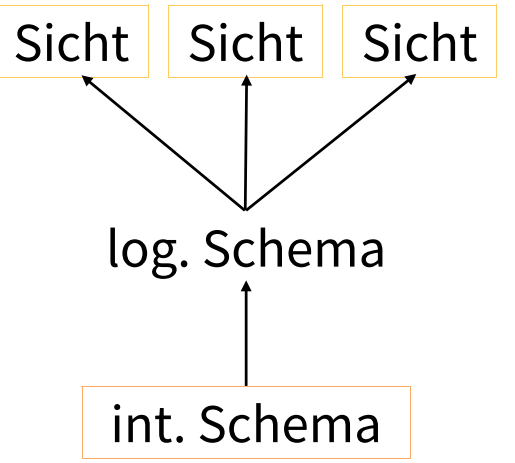
In diesem Kapitel...

- ... definieren wir Sichten,
- ... lernen, wie man DB-Benutzer und Rollen verwaltet,
- ... schauen wir uns das Transaktionenkonzept genauer an,
- ... erfahren wir, wie man Mehrbenutzeranomalien vermeidet
- ... und wie Recovery-Mechanismen funktionieren.

Datenabstraktion

Externe Ebene (Sichten / Views)

Sicht auf eine Teilmenge des logischen Schemas für eine bestimmte Benutzergruppe



Logische Ebene

DB-Gesamt-Schema

Physische Ebene

Internes Schema / Speicherung der Daten

Datenabstraktion (Bsp.)

Externe Ebene (View für Webshop-Anwendung)

Produktnr	**Bezeichnung**	**Preis**	**Bewertung**
17	Schokoriegel	0.89	4.5
29	Spülmaschinentabs	3.99	2.0

Logische Ebene

Produktnr	**Bezeichnung**	**Preis**
17	Schokoriegel	0.89
29	Spülmaschinentabs	3.99

Kundennr	**Produktnr**	**Bew**
5	17	4
8	17	5
5	29	2

Physische Ebene: 00101100000001001011110101010...

Sichten / Views in SQL

Produktnr	Bezeichnung	Preis	Hersteller	Bewertung
17	Schokoriegel	0.89	Monsterfood	4.5
18	Müsliriegel	1.19	Monsterfood	-
29	Spülmaschinentabs	3.99	Calgonte	2.0

```
CREATE VIEW produkte_view AS
SELECT P.produktnummer, P.bezeichnung, P.preis, P.hersteller,
       AVG(B.sterne) AS bewertung
FROM produkte P LEFT JOIN bewertungen B
     ON P.produktnummer = B.produktnummer
GROUP BY P.produktnummer, P.bezeichnung, P.preis, P.hersteller
```

```
SELECT * FROM produkte_view
```

Eine View ist eine Art virtuelle Tabelle. Sie wird definiert über einen Namen und eine SELECT-Anfrage. Sie kann genau wie Tabellen in der FROM-Klausel in SELECT-Anfragen abgefragt werden. In der hier dargestellten Anfrage wird ein Left-Join verwendet, damit auch Produkte, die nie bewertet wurden, im Ergebnis auftauchen.

Views / Sichten in SQL

```
CREATE VIEW teure_produkte AS SELECT * FROM produkte
                              WHERE preis > 10;
```

```
SELECT * FROM teure_produkte WHERE hersteller = 'Monsterfood'
```

wird ausgeführt als:

```
SELECT * FROM (SELECT * FROM produkte WHERE preis > 10)
WHERE hersteller = 'Monsterfood'
```

Erfolgt eine Abfrage auf eine View, wird der Name der View durch die Anfrage ersetzt, über die die View definiert ist. Danach erfolgen Optimierungen, sodass die hier dargestellt Anfrage äquivalent ist zu `SELECT * FROM produkte WHERE preis > 10 AND hersteller = 'Monsterfood'`. Anfragen auf Sichten sind nicht langsamer als die hinter der Sicht stehende Abfrage direkt auszuführen, sie sind aber auch nicht schneller.

Einsatzszenarios von Views

Vereinfachung / Speicherung von Anfragen

```
CREATE VIEW produkte_view AS ...
```

Datenunabhängigkeit

Anwendungen greifen über View auf Daten zu; unabhängig, wie die tatsächlichen Tabelle aussehen.

Datenschutz

Benutzern wird nur der Zugriff auf die View gewährt, nicht auf die zugrundeliegenden Tabellen.

Wenn Operationen wie Joins oder Aggregationen immer wieder in Anfragen vorkommen, kann man diese in eine View platzieren und damit den Zugriff vereinfachen. So sind auch Denormalisierungen von normalisierten Tabellen mittels Sichten möglich.

CHECK OPTION

INSERT, UPDATE und DELETE sind auf simplen Projektion/Selektion-Views erlaubt.

```
CREATE VIEW teure_produkte AS SELECT * FROM produkte
                              WHERE preis > 10;
```

INSERT / UPDATE funktioniert ohne Überprüfung des WHERE-Prädikats.

WITH CHECK OPTION

```
CREATE VIEW teure_produkte AS SELECT * FROM produkte
                              WHERE preis > 10 WITH CHECK OPTION;
```

Das einzufügende oder zu ändernde Tupel muss das WHERE-Prädikat erfüllen.

Ein `INSERT INTO teure_produkte` ist hier möglich. Im unteren Fall jedoch nur, wenn das neue Produkt auch einen Preis von mehr als 10 EUR hat. `WITH CHECK OPTION` ist eine Kurzschreibweise für `WITH LOCAL CHECK OPTION`. Wenn eine View mit dieser Option eine andere View in der FROM-Klausel hat, wird nicht das Prädikat der anderen View überprüft, außer: Für diese ist ebenfalls eine CHECK OPTION gesetzt, oder wir spezifizieren für die View: `WITH CASCADED CHECK OPTION`.

Materialisierte Sichten

Views sind nur virtuell, sie speichern nicht wirklich Daten.

Materialisierte Sichten jedoch speichern physisch das Ergebnis der dahinterstehenden Anfrage.

```
CREATE MATERIALIZED VIEW anz_produkte AS
SELECT COUNT(*) as anz FROM produkte;
```

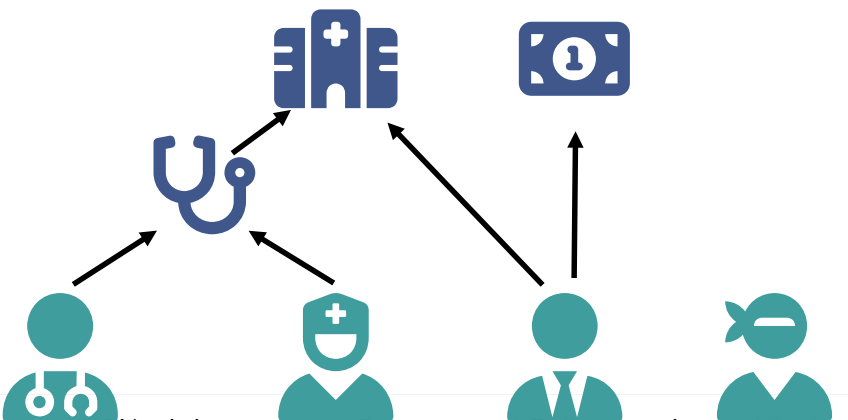
REFRESH MATERIALIZED VIEW

```
SELECT * FROM anz_produkte; -- zeigt an: 8
INSERT INTO produkte (produktnr, bezeichnung) values (123, 'X');
SELECT * FROM anz_produkte; -- zeigt immer noch 8
REFRESH MATERIALIZED VIEW anz_produkte;
SELECT * FROM anz_produkte; -- zeigt nun 9
```

Beim Erstellen einer materialisierten Sicht, wird diese initial mit dem Anfrageergebnis befüllt. Es gibt Datenbankmanagementsysteme, die ein automatisches Auffrischen unterstützen. Dies erfolgt, wenn sich die Daten in den Basistabellen ändern. In PostgreSQL ist beinhaltet die materialisierte Sich in diesen Fällen veraltete Daten. Mit `REFRESH MATERIALIZED VIEW` kann ein Auffrischen angestoßen werden.

Benutzer und Rollen

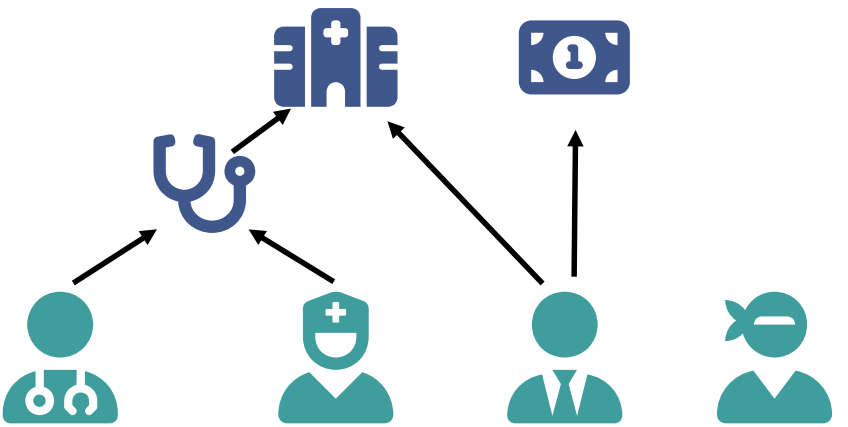
```
CREATE USER doktor_x WITH PASSWORD 'geheimespasswort1';
CREATE USER schwester_y WITH PASSWORD 'geheimespasswort2';
CREATE USER big_boss WITH PASSWORD 'geheimespasswort3';
CREATE USER ninja WITH PASSWORD 'geheimespasswort4';
```



Die Data Control Language (DCL) wird genutzt, um Benutzer und Rollen anzulegen. Hier legen wir vier Benutzer an. Diese können sich mit dem angegebenen Nutzernamen und Passwort mit der Datenbank verbinden. In PostgreSQL sind Benutzer und Rollen übrigens das gleiche. `CREATE USER` ist lediglich ein Alias für `CREATE ROLE`.

Benutzer und Rollen

```
CREATE ROLE klinikpersonal;
GRANT klinikpersonal TO doktor_x, schwester_y;
CREATE ROLE mitarbeiter;
GRANT mitarbeiter TO big_boss, klinikpersonal;
CREATE ROLE finanzen;
GRANT finanzen TO big_boss;
```



GRANT: Berechtigung erteilen

GRANT <Recht> ON <Objekt> TO <Benutzer oder Rolle>

```
--Mitarbeiter dürfen sich mit der Datenbank verbinden
GRANT CONNECT ON klinik_db TO mitarbeiter;
```

```
--Die Finanzabteilung darf die Tabelle Reisekosten lesen
GRANT SELECT ON klinik.reisekosten TO finanzen;
```

```
--Doktor X darf alles auf der Patienten-Tabelle machen
GRANT ALL PRIVILEGES ON klinik.patienten TO doktor_x;
```

```
--Schwester Y darf Patienten einsehen, erstellen und ändern
GRANT SELECT, INSERT, UPDATE ON klinik.patienten TO schwester_y;
```

```
GRANT SELECT ON ALL TABLES IN SCHEMA klinik TO big_boss;
```

```
GRANT SELECT ON klinik_raeume TO PUBLIC; -- Das darf jeder
```

Mit dem GRANT-Befehl wird festgelegt, **wer** auf **welchem** Objekt **was** machen darf. Hinter dem Stichwort ON steht das Objekt. Für das CONNECT-Recht ist dies eine Datenbank. Bei den anderen hier angegeben Privilegien steht hinter dem Stichwort ON eine oder mehrere Tabellen, oder alle Tabellen eines Schemas. In der Rolle PUBLIC sind alle Benutzer.

REVOKE: Berechtigung entziehen

```
GRANT SELECT ON klinik.raeume TO PUBLIC;
```

```
CREATE VIEW klinik.raeume_view AS
SELECT * FROM klinik.raeume WHERE gebaeude IN (10,12,13,15);
```

REVOKE <Recht> ON <Objekt> FROM <Benutzer oder Rolle>

```
REVOKE SELECT ON klinik.raeume FROM PUBLIC;
```

```
GRANT SELECT ON klinik.raeume_view TO PUBLIC;
```

Wichtig: Man kann nur Rechte entziehen, die man genau so auch erteilt hat. Man kann im gezeigten Beispiel nicht das Recht nur gewissen Benutzern der Rolle PUBLIC entziehen (nur allen). Genau so kann nach einem GRANT ALL PRIVILEGES nicht lediglich das INSERT-Recht o. ä. entzogen werden. REVOKE macht also ein GRANT wieder vollständig rückgängig. Im hier gezeigten Beispiel erstellen wir eine Sicht aus Gründen der **Row-Level-Security**. Wir möchten den Benutzern der Rolle PUBLIC nicht erlauben, alle Räume zu sehen, sondern lediglich Räume in den Gebäuden 10, 12, 13 und 15. Analog lässt sich auch **Column-Level-Security** realisieren. Man lässt einfach in der View-Definition Spalten weg, um den Zugriff nur auf bestimmte Spalten zu beschränken.

GRANT OPTION

Das Privileg, ein Privileg weiterzugeben.

```
GRANT SELECT ON ALL TABLES IN SCHEMA klinik TO big_boss
WITH GRANT OPTION;
```

```
REVOKE GRANT OPTION FOR SELECT ON ALL TABLES IN SCHEMA klinik
FROM big_boss; -- nur GRANT OPTION entziehen, nicht das Recht
```

REVOKE ... CASCADE

```
-- Funktioniert nur, wenn das Recht nicht weitergegeben wurde
REVOKE SELECT ON ALL TABLES IN SCHEMA klinik FROM big_boss;
```

```
-- CASCADE: Entzieht das Recht auch allen, denen es weitergegeben wurde
REVOKE SELECT ON ALL TABLES IN SCHEMA klinik FROM big_boss CASCADE;
```

Der Benutzer big_boss hat das Recht, SELECT-Anfragen auf allen Tabellen und Views im Klinik-Schema auszuführen. Außerdem hat er das Recht, dieses Recht auch anderen Benutzern und Rollen zu geben. Intern wird gespeichert, wer wann wem welches Recht gegeben hat. Mit diesen Infos können dann z. B. Rechte wieder kaskadiert entzogen werden.

Transaktionen

ACID

- Atomarität
- Konsistenz
- Isolation
- Dauerhaftigkeit

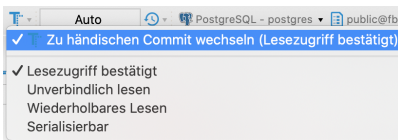
Eine ACID-Transaktion erfüllt die Eigenschaften Atomarität (sie wird vollständig oder gar nicht ausgeführt), Konsistenz (sie hinterlässt die Datenbank in einem konsistenten Zustand), Isolation (sie wird von keinen anderen parallel laufenden Transaktionen beeinflusst) und Dauerhaftigkeit (sie speichert Änderungen persistent).

Commit / Rollback

Mit einer `commit`-Operation beendet man eine Transkation.
`rollback` beendet ebenfalls die Transkation, sie führt jedoch zum Abort, sodass alle Änderungen rückgängig gemacht werden.

Autocommit

DB-Anwendungen setzen Autocommit-Einstellung.
Bei SQL-Clients wie dem DBeaver ist Autocommit standardmäßig an.



Autocommit bedeutet, dass jede SQL-Anfrage automatisch committet wird. Stelle man um auf manuelles Commit, muss man selbst eine `commit`-Operation ausführen, damit Änderungen wirklich in der Datenbank gespeichert werden. Ist Autocommit aus, kann die laufende Transaktion mittels `rollback` auch komplett zurückgerollt werden.

Atomarität

Eine TA wird entweder ganz oder gar nicht ausgeführt.

```
INSERT INTO hersteller VALUES ('Dogdog', 'England');
INSERT INTO produkte VALUES (129, 'Hundefutter', 8.95, 'Dogdog');
COMMIT;
```

```
INSERT INTO hersteller VALUES ('Techbob', 'USA');
INSERT INTO produkte VALUES (129, 'Laptop', 999.99, 'Techbob');
```

SQL-Fehler [23505]: ERROR: duplicate key value violates unique constraint "produkte_pkey" Detail: Key (produktnummer)=(129) already exists.

⇒ Die TA kann komplett zurückgerollt werden, sodass der Hersteller Techbob auch nicht eingefügt wird.

In diesen und in den nun folgenden Beispielen ist Autocommit ausgestellt. Zunächst wird ein Hersteller und ein Produkt eingefügt und diese Transaktion erfolgreich mit einem Commit beendet. Dann folgen zwei INSERTs, wobei das zweite davon fehlschlängt. Nun lässt sich die Transaktion abbrechen, sodass nichts aus der Transaktion ausgeführt wurde.

Konsistenz

Eine TA führt die Datenbank von einem konsistenten Zustand in wieder einen konsistenten Zustand.

```
INSERT INTO produkte VALUES (129, 'Laptop', 999.99, 'Techbob');
```

SQL-Fehler [23505]: ERROR: duplicate key value violates unique constraint "produkte_pkey" Detail: Key (produktnummer)=(129) already exists.

Alle geltenden Integritätsbedingungen müssen zum Ende einer Transaktion erfüllt sein: Primärschlüssel-, Fremdschlüssel-, UNIQUE-, NOT NULL-, CHECK-Constraints, etc.

Es ist erlaubt, dass in Mitten einer Transaktion die Datenbank in einem inkonsistenten Zustand ist. Aber spätestens beim Commit muss dafür gesorgt werden, dass alle Integritätsbedingungen erfüllt sind. Sind sie es nicht, kann die Transaktion nicht erfolgreich abschließen und wird abgebrochen und zurückgesetzt.

Isolation

Parallel laufendende Transaktionen beeinflussen sich nicht.

Mehrbenutzeranomalien

- **Dirty Read**: TA_2 liest noch nicht committete Änderungen von TA_1
- **Lost Update**: TA_1 und TA_2 schreiben gleichzeitig; wer gewinnt?
- **Non-repeatable Read:** TA liest mal veraltete und mal aktualisierte Werte
- ...

Ziel: Gefühlter Einbenutzerbetrieb!

Um in Datenbanken eine hohe Performanz zu ermöglichen, ist es erlaubt, dass viele Transaktionen parallel laufen. Die dabei naturgemäß auftretenden Mehrbenutzeranomalien müssen dabei verhindert werden. Für eine Transaktion soll es sich so anfühlen, als ob sie die einzige gerade laufende Transaktion ist.

Dirty Read

TA_1

```
SELECT preis FROM produkte
WHERE produktnr = 17; -- 0.89 EUR

UPDATE produkte
SET preis = 0
WHERE produktnr = 17;

ROLLBACK;
```

TA_2

```
SELECT preis FROM produkte
WHERE produktnr = 17; -- 0 EUR
COMMIT;
```

Es wäre ein Dirty Read, wenn TA_2 die noch nicht freigegebene Preisänderung von TA_1 sehen würde. Das Transaktions-Management-System (TMS) eines DBMS muss gewährleisten, dass Änderungen für andere Transaktionen erst nach dem Commit der Änderung sichtbar werden.

Lost Update

TA_1	TA_2
`SELECT preis FROM produkte WHERE produktnr = 17; -- 0.89 EUR`	`SELECT preis FROM produkte WHERE produktnr = 17; -- 0.89 EUR`
`UPDATE produkte SET preis = preis + 0.1 WHERE produktnr = 17; -- 0.99 EUR`	`UPDATE produkte SET preis = preis + 0.1 WHERE produktnr = 17; -- 0.99 EUR`
`COMMIT;`	`COMMIT;`

Zwei Transkationen möchten den Preis eines Produktes um 10 Cent erhöhen. Als Endresultat sollte das Produkte also 0.89+0.1+0.1 = 1.09 EUR kosten. Da im gezeigten Szenario die Änderungen gleichzeitig und unisoliert passierten, trat ein Lost Update aus. Eine der beiden Änderungsoperationen ist verloren gegangen. Das Transaktions-Management-System muss solche Lost Updates verhindern, sodass der Endzustand der Datenbank äquivalent ist zu dem Endzustand, den die DB hätte, wenn die Transaktionen nicht gleichzeitig, sondern seriell hintereinander gelaufen wären.

Non-repeatable Read

TA_1	TA_2
`SELECT preis FROM produkte WHERE produktnr = 17; -- 0.89 EUR`	`UPDATE produkte SET preis = preis + 0.1 WHERE produktnr = 17; -- 0.99 EUR COMMIT;`
`SELECT preis FROM produkte WHERE produktnr = 17; -- 0.99 EUR COMMIT;`	

Im Einbenutzerbetrieb wäre es unmöglich, dass eine Transaktion zwei verschiedene Ergebnisse für die gleiche Abfrage erhält, außer sie hätte die Daten selbst geändert. Das TMS muss gewährleisten, dass in einem Fall, wie er hier gezeigt wird, wiederholtes Lesen stets das gleiche Ergebnis liefert.

Phantomproblem

TA_1

```
SELECT * FROM produkte
WHERE hersteller = 'Monsterfood';
-- 2 Produkte werden angezeigt



SELECT COUNT(*) FROM produkte
WHERE hersteller = 'Monsterfood';
-- Ergebnis: 3

COMMIT;
```

TA_2

```


INSERT INTO produkte
VALUES (..., 'Monsterfood');
COMMIT;
```

Das Phantomproblem ist ähnlich zum Non-repeatable Read, bezieht sich jedoch auf während einer laufenden Transaktion neu hinzugefügte oder gelöschte Zeilen. TA_1 möchte alle Monsterfood-Produkte anzeigen und zusätzlich deren Anzahl (oder Preis-Summe, o.ä.) berechnen. Das Ergebnis ist inkonsistent, weil zwischenzeitlich TA_2 ein neues Produkt hinzugefügt hat.

Serialisierbarkeit

Jede Transaktion besteht aus Lese- und Schreibaktionen: $r_1(x), r_1(y), w_1(x), c_1$

Serieller Ablauf

Keine verzahnte Ausführung, z. B.
$r_1(x), r_1(y), w_1(x), c_1, r_2(z), r_2(x), c_2, r_3(y), c_3$

Serialisierbarkeit ("Final-State-serialisierbar")

Ablauf ist serialisierbar, wenn Anfragen das gleiche Ergebnis liefern und DB im gleichen Zustand hinterlassen wird, wie bei irgendeinem seriellen Ablauf.

Serialisierbarkeit ("Konflikt-serialisierbar")

Ablauf ist serialisierbar, wenn es keine Zyklen im Serialisierbarkeitsgraphen gibt.

Die Notation $r_1(x)$ bzw. $w_1(x)$ bedeutet, dass eine Transaktion 1 ein Objekt X (z. B. eine Tabelle, eine Zeile oder einen Spaltenwert) liest bzw. schreibt. c_1 ist das Commit von TA_1. Final-State-Serialisierbarkeit ist schwierig zu überprüfen und außerdem kann auch rein zufällig das gleiche herauskommen wie bei einem seriellen Ablauf. Daher wird in der Regel auf Konflikt-Serialisierbarkeit überprüft.

Serialisierbarkeitsgraph

Konfliktoperationen

- r → w
- w → r
- w → w

Von verschiedenen TAs auf gleichem Objekt.

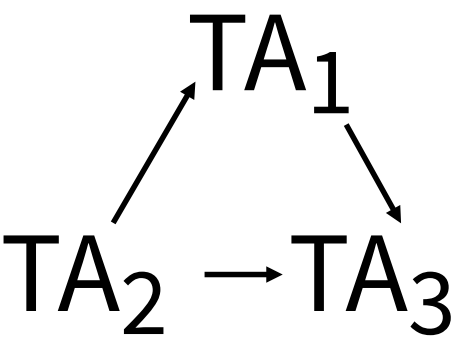
Im Serialisierbarkeitsgraphen sind die TA die Knoten und es gibt eine Kante von einer TA zu einer anderen, wenn Konfliktoperationen zwischen diesen existieren.

Serialisierbarkeitsgraph

Beispiel: $r_1(x), w_1(y), r_2(x), w_1(x), w_3(y), w_3(x), c_1, c_2, c_3$

Konfliktoperationen hier:

- $r_1(x) \rightarrow w_3(x)$
- $w_1(y) \rightarrow w_3(y)$
- $r_2(x) \rightarrow w_1(x)$
- $r_2(x) \rightarrow w_3(x)$
- $w_1(x) \rightarrow w_3(x)$



Kein Zyklus im Serialisierbarkeitsgraphen ⇒ Der Ablauf ist serialisierbar

Äquivalenter serieller Ablauf: TA_2, TA_1, TA_3

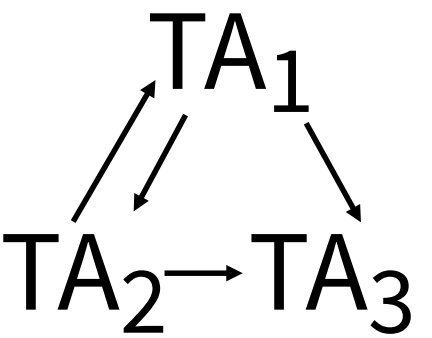
Um den Serialisierbarkeitsgraphen zu erzeugen, geht man wie folgt vor: Für jede Operation prüft man, ob Konfliktoperationen von einer anderen Transaktion auf dem gleichen Objekt zeitlich später folgen. Falls ja, erstellt man, falls noch nicht vorhanden, eine Kante von der ersten zur zweiten Transaktion. Manchmal gibt es mehrere äquivalente serielle Abläufpläne. Alle diese würden zum gleichen Ergebnis führen wie der gegebene verzahnte Ablauf.

Serialisierbarkeitsgraph

Anderes Beispiel: $r_1(x), w_1(x), r_2(x), w_1(x), w_3(y), w_3(x), c_1, c_2, c_3$

Konfliktoperationen hier:

- $r_1(x) \rightarrow w_3(x)$
- $w_1(x) \rightarrow r_2(x)$
- $w_1(x) \rightarrow w_3(x)$
- $r_2(x) \rightarrow w_1(x)$
- $r_2(x) \rightarrow w_3(x)$
- $w_1(x) \rightarrow w_3(x)$



Zyklus im Serialisierbarkeitsgraphen ⇒ Der Ablauf ist nicht serialisierbar

Es gibt keinen äquivalenten seriellen Ablauf, daher ist der hier gegebene Ablauf nicht serialisierbar..

SX-Sperrverfahren

Sperrverfahren werden eingesetzt, um Serialisierbarkeit zu erreichen. Bevor eine TA ein Objekt liest oder schreibt, muss es dies mit einer Sperre versehen.

Sperrmatrix

	S	X
S	✓	-
X	-	-

Sperrverfahren werden eingesetzt, um Isolation zu gewährleisten. Beim SX-Sperrverfahren gibt es Shared-Sperren (S) und eXklusive Sperren (X). Mehrere TAs können eine gemeinsame S-Sperre auf dem gleichen Objekt haben, aber nur eine TA darf eine X-Sperre auf einem Objekt haben. Vor dem Lesen eines Objektes wird die S-Sperre angefordert, vor dem Schreiben eine X-Sperre. Links in der Sperrmatrix steht die aktuell auf einem Objekt existierende Sperre anderer Transaktionen. Oben steht, welche Sperre eine Transkation anfordern möchte. Der Haken gibt an, ob sie die Sperre bekommen kann oder nicht. Im letzteren Fall muss die TA warten, bis die Sperre wieder freigegeben wurde.

SX-Sperrverfahren

Beispiel: $r_1(x), r_2(x), c_1, c_2$

TA_1	TA_2	Gesetzte Sperren
$r_1(x)$		$S_1(x)$
	$r_2(x)$	$S_{1,2}(x)$
c_1		$S_2(x)$
	c_2	

Damit $_1$ das Objekt x lesen kann, muss es zunächst eine S-Sperre auf x anfordern. Dies funktioniert, da im Moment noch keine Sperre auf x existiert. Auch TA_2 möchte das Objekt x lesen. Aktuell gibt es eine S-Sperre von TA_1 auf x, die sich aber mit der angeforderten S-Sperre verträgt. Es gibt nun also eine S-Sperre von TA_1 und TA_2 auf x. Beim Commit oder Rollback einer Transaktion werden Sperren wieder freigegeben. In unserem Beispiel sehen wir, dass nach dem Commit von TA_1 nur noch die S-Sperre von TA_2 auf x existiert. Im nächsten Schritt, beim Commit von TA_2, wird auch diese aufgelöst, sodass es schließlich keine Sperren mehr gibt.

SX-Sperrverfahren

Beispiel: $r_1(x), w_1(x), r_2(x), w_1(x), w_3(y), w_3(x), c_1, c_2, c_3$

TA_1	TA_2	TA_3	Gesetzte Sperren
$r_1(x)$			$S_1(x)$
$w_1(x)$			$X_1(x)$
	warten		$X_1(x)$
$w_1(x)$	warten		$X_1(x)$
	warten	$w_3(y)$	$X_1(x), X_3(y)$
	warten	warten	$X_1(x), X_3(y)$
c_1	warten	warten	$X_3(y)$
	$r_2(x)$	warten	$S_2(x), X_3(y)$
	c_2	warten	$X_3(y)$
		$w_3(x)$	$X_3(x), X_3(y)$
		c_3	

Deadlocks

Beispiel: $r_1(x), r_2(y), w_2(x), w_1(y), c_1, c_2$

TA_1	TA_2	Gesetzte Sperren
$r_1(x)$		$S_1(x)$
	$r_2(y)$	$S_1(x), S_2(y)$
	warten	$S_1(x), S_2(y)$
warten	warten	$S_1(x), S_2(y)$

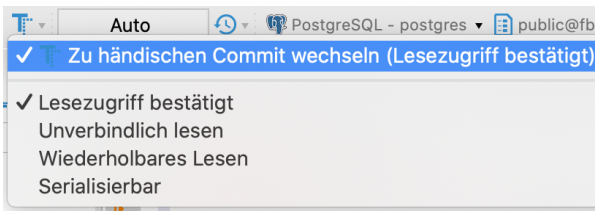
TA_2 wartet auf TA_1 und umgekehrt ⇒ Deadlock

Lösung: Eine der TAs muss vom TMS zurückgesetzt werden.

Ein Transaktions-Management-System muss in der Lage sein, Deadlocks zu erkennen (z. B. durch Erkennung zyklischer Wartebedingungen oder mittels Timeouts) und sie aufzulösen, damit Transaktionen nicht unendlich lange warten. In der Regel wird ein Deadlock aufgelöst, indem eine der TAs zurückgesetzt wird und somit die andere fortfahren kann.

SQL-Isolationslevels

- Read Committed
- Read Uncommitted
- Repeatable Read
- Serializable



	Dirty Read	Non-Repeatable Read	Phantomproblem
Read Uncommitted	möglich	möglich	möglich
Read Committed	nicht mögl.	möglich	möglich
Repeatable Read	nicht mögl.	nicht mögl.	möglich
Serializable	nicht mögl.	nicht mögl.	nicht mögl.

Lost Updates werden stets verhindert, sie sind in allen Iso'levels nicht möglich.

Während einer Transaktion kann eine Anwendung ein Isolationslevel setzen, um dem TMS mitzuteilen, welche Anomalien innerhalb dieser TA verhindert werden müssen. Werden weniger strikte Levels gewählt, ist in der Regel eine bessere Performance möglich.

Multi-Version-Concurrency-Control

- Ein Objekt (Tabelle, Zeile, Wert, ...) kann in mehreren Versionen existieren.
- TA schreibt: $w_1(x) \Rightarrow$ neue Version x' anlegen
- TA liest: $r_2(x) \Rightarrow$ alte Version x wird gelesen
- TA committet: $c_1 \Rightarrow$ neue Version wird gültig und sichtbar für andere TAs

Beispiel (nicht Konflikt-serialisierbar): $r_1(x), w_1(x), r_2(x), w_1(x), c_1, c_2$
↖ TA_2 liest alte Version

Das ist äquivalent zu folgendem und somit doch Konflikt-serialisierbar:
$r_1(x), r_2(x), w_1(x), w_1(x), c_1, c_2$

Moderne DBMS wie PostgreSQL, Oracle, DB2, SQL Server verwenden MVCC zur Transaktionssicherung. Jede Aktion einer TA basiert logisch gesehen auf einem Schnappschuss der Datenbank zum Zeitpunkt des TA-Starts. Im gezeigten Beispiel liest TA_2 den Wert von x, den es vor der Änderung von TA_1 hatte. Es ist daher auch kein Dirty Read.
Beim Schreiben eines Objektes wird eine neue Version davon angelegt. Die TA selbst arbeitet natürlich auf seiner geänderten Kopie, andere laufende Transaktion arbeiten auf der alten Version. Selbst wenn die TA committet, muss die alte Version noch beibehalten werden, solange noch laufende TAs diese alte Version benötigen.

Recovery-Mechanismen

Transaktionen müssen auch bei TA-, System- und Hardwarefehlern ACID-konform ausgeführt werden (atomar, ..., dauerhaft).

Aus Performance-Gründen schreiben TAs erst einmal Änderungen nur im RAM; von Zeit zu Zeit werden die geänderten Blöcke dann auf die Festplatte ge-flush-t.

Alle Änderungen einer TA werden aber in ein **Transaktions-Log** eingetragen, welches immer spätestens bei einem Commit auf die Platte geschrieben wird.

LSN	TA	PageID	Undo	Redo	PrevLSN
27	TA_1	x	Undo-Info	Redo-Info	22

Das Transaktions-Log speichert, wer (TA) wann (LSN = Log-Sequenz-Nummer) was (PageID) wie geändert hat (Redo) und wie man diese Änderung zurückgängig machen kann (Undo). Die Undo- und Redo-Infos so etwas wie "Byte 17 bis Byte 20 haben sich verändert in 1A5B2FFF". Die LSN ist eine laufende Nummer, um die Einträge zu ordnen. Die PrevLSN beinhaltet die LSN der vorherigen Aktion der gleichen TA. Mit all diesen Infos können im Fehlerfall Änderungen rückgängig gemacht werden und noch nicht auf die Platte geschriebene Änderungen nachgeholt werden.

Transaktionsfehler

Beispiel: $r_1(x), r_2(y), r_1(z), w_1(x), w_2(y), w_1(z), a_1, c_2$

LSN	TA	PageID	Undo	Redo	PrevLSN
1	TA_1	BOT	-	-	-
2	TA_2	BOT	-	-	-
3	TA_1	x	**Undo-Info**	Redo-Info	1
4	TA_2	y	Undo-Info	Redo-Info	2
5	TA_1	z	**Undo-Info**	Redo-Info	3
6	TA_1	Abort	-	-	5
7	TA_2	Commit	-	-	4

Irgendetwas hat nicht wie gewünscht funktioniert, daher rollt die Anwendung TA_1 zurück. Das DBMS verwendet dazu die hier fett markierten Undo-Infos. Diese können leicht im Log gefunden werden, indem man immer in die PrevLSN-Spalte schaut und dann an die entsprechende Zeile springt. BOT steht für Begin of Transaction.

Systemfehler

Beispiel: $r_1(x), r_2(y), r_1(z), w_1(x), w_2(y), w_1(z), c_1,$ ↯ Crash!

LSN	TA	PageID	Undo	Redo	PrevLSN
1	TA_1	BOT	-	-	-
2	TA_2	BOT	-	-	-
3	TA_1	x	Undo-Info	Redo-Info	1
4	TA_2	y	**Undo-Info**	Redo-Info	2
5	-	Flush	-	-	-
6	TA_1	z	Undo-Info	**Redo-Info**	3
7	TA_1	Commit	-	-	6

Bei einem Systemfehler ist immer die Frage, welche Transaktionen sind Gewinner (haben committed) und welche sind Verlierer (waren noch aktiv). Da das System zu Zeitpunkt 5 zuletzt einen Flush gemacht hat, um DB-Seiten auf die Festplatte zu persistieren, müssen alle davor durchgeführten Aktionen von Verlierern rückgängig gemacht und noch nicht eingebrachte Änderungen von Gewinnern nachgeholt werden.

Kapitelzusammenfassung

- Sichten: CREATE VIEW ... AS
- WITH [LOCAL|CASCADED] CHECK OPTION
- Materialisierte Sichten / REFRESH
- Benutzer und Rollen
- GRANT / REVOKE
- ACID-Transaktionen
- Mehrbenutzeranomalien (Dirty Read, Lost Update, ...)
- Serialisierbarkeit / Serialisierbarkeitsgraph
- SX-Sperrverfahren
- Isolationslevels
- MVCC
- Recovery / Transaktions-Log