Datenbank-Programmierung

Chapter 4: Mehrbenutzer-Synchronisation

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2018/19

http://www.informatik.uni-halle.de/~brass/dbp19/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Erklären, was geschieht, wenn mehrere Benutzer gleichzeitig auf die Datenbank zugreifen.

  Problem-Typen aufzählen, zu jedem ein Beispiel machen.

  Sperren (inkl. Deadlocks) und "Multi Version Concurrency Control" erklären.

- Mehrbenutzer-Sicherheit von Programmen bewerten.

  Wann muss man "`FOR UPDATE`" zu einer Anfrage hinzufügen?

- "Lost Updates" schon beim Entwurf von Anwendungen (z.B. Web-Formularen) vermeiden.

- Gegebene Schedules auf "Konflikt-Serialisierbarkeit" prüfen.

Inhalt

Ziel: Isolation (1)

- Jeder Benutzer soll den Eindruck haben, dass er/sie für die ganze Dauer der Transaktion exklusiven Zugriff auf die Datenbank hat.
- Alle anderen Transaktionen müssen daher so erscheinen, als wären sie
  - vor der eigenen Transaktion vollständig abgeschlossen, oder
  - erst nach dem Ende der eigenen Transaktion begonnen.

Ziel: Isolation (2)

- Was Benutzer sehen (als Ergebnisse von Anfragen) und die Änderungen, die sie in der DB hinterlassen, müssen äquivalent zu einem seriellen Schedule sein.

  Ein Schedule legt die Verschachtelung der Ausführung von Befehlen verschiedener Benutzer (genauer: Transaktionen) fest. Die Komponente "Scheduler" des DBMS bestimmt, wer "als nächstes drankommt". Ein Schedule heißt seriell, wenn er immer eine Transaktion vollständig abarbeitet, bevor er mit der nächsten beginnt. Ein Schedule, der äquivalent zu einem seriellen Schedule ist, heißt serialisierbar.

- Theoretisch soll es für jeden Benutzer so aussehen, als hätte man den "Ein-Terminal-Betrieb".

  Auf die Datenbank kann nur über ein einziges Terminal zugegriffen werden, dahinter reihen sich alle Benutzer in einer Warteschlange auf.

Ziel: Leistung

- Während eine Transaktion auf eine Platte oder Benutzereingaben wartet, sollte das DBMS eine andere Transaktion bearbeiten (statt nichts tun).
- Eine lange Transaktion muss von Zeit zu Zeit unterbrochen werden, um kurze Transaktionen zwischendurch abzuarbeiten.

  Dies verbessert die durchschittliche Antwortzeit deutlich: Sonst würde sich hinter der langen Transaktion eine lange Warteschlange mit kurzen Transaktionen aufbauen.

- Gleichzeitige Transaktionen können parallele Hardware gut ausnutzen.

  Schon Notebooks haben heute 2–4 Kerne. Wir haben 2016 einen Server-Rechner für ca. 4 200 Euro gekauft mit zwei CPUs "Intel Xeon E5-2630 v4 @ 2.20GH" mit jeweils 10 Kernen, die durch Hyperthreading verdoppelt werden.
  (d.h. insgesamt 40 Threads hardware-unterstützt gleichzeitig in Ausführung).

Probleme (1)

- Die beiden Ziele stehen im Konflikt mit einander: 100% Isolation bedeutet sehr wenig Parallelität — häufig müssen ganze Tabellen gesperrt werden.
- SQL hat erst seit SQL-99 ein "`START TRANSACTION`" Kommando (optional, existiert nur in manchen DBMS). Bei einer langen Folge von Anfragen ist nicht klar,
  - ob sie wirklich alle zusammen eine Transaktion bilden sollen,
  - oder jede für sich eine eigene Transaktion.

    Eigentlich müßte man dafür nach jeder Abfrage `COMMIT`/`ROLLBACK` eingeben, aber das ist unüblich. Für das DBMS sind viele kurze Transaktionen einfacher als eine lange, auch bei Abfragen. Abfrageergebnisse fließen manchmal in ein folgendes Update ein.

Probleme (2)

- DBMS garantieren daher “etwas Isolation” und bieten Mechanismen an, um die vollständige Isolation zu erreichen.
- Aber sie brauchen dazu Hilfe vom Programmierer.
- Meistens braucht sich der Programmierer keine Gedanken über die Möglichkeit paralleler Transaktionen machen.

  Das vereinfacht natürlich die Anwendungsentwicklung.

- Er muss sich aber der wenigen Fälle bewusst sein, in denen spezielle Befehle benutzt werden müssen.

Probleme (3)

- Fehler aufgrund störender gleichzeitiger Transaktionen sind besonders unangenehm/schwierig:
  - Sie werden beim Testen nicht gefunden.

    Normalerweise testet nur ein Entwickler gleichzeitig. Es braucht aber die reale Systemlast und selbst dann kann es Monate dauern, bis die kritische Verschachtelung der Transaktionen auftritt.
  - Sie sind nicht einfach reproduzierbar.
- Daher ist es wichtig, sie theoretisch auszuschließen (durch Nachdenken/Planung, nicht durch Hoffen und Testen).

  Am besten ist natürlich eine Lösung, in der das DBMS sich alleine darum kümmert, und zum Teil ist das ja auch realisiert.

Parallele Sitzungen testen

- Die Mehrbenutzer-Fähigkeiten eines DBMS können ausprobiert werden, indem man den SQL Interpreter mehrfach in verschiedenen Fenstern startet.
- Man hat dann mehrere parallele Sitzungen

  Unter dem gleichen Benutzernamen, d.h. mit Zugriff auf das gleiche Datenbank-Schema. Es ist in der Praxis nicht untypisch, dass verschiedene Personen über Anwendungsprogramme unter dem gleichen DB-Account arbeiten. Natürlich ist es auch möglich, dass verschiedene Datenbank-Benutzer auf die gleichen Tabellen Zugriff haben. Für die Mehrbenutzer-Synchronization macht das keinen Unterschied.
- In den Beispielen wird folgende Tabelle benutzt: `KONTO(NR, STAND)`.

Inhalt

Sperren (1)

- Die meisten Systeme benutzen Sperren ("Locks") für die Mehrbenutzer-Synchronization.

  Sperren können auf Objekten verschiedener Granularität genutzt werden: Tabellen, Plattenblöcken, Tupeln, Tabelleneinträgen.

- Wenn eine Transaktion A ein Objekt (z.B. ein Tupel) gesperrt hat, und Transaktion B möchte das Objekt auch sperren, so muss B warten.

  B bekommt in der Zwischenzeit keine CPU-Zyklen mehr (wird "schlafen gelegt"). Der "Lock Manager" im DBMS bzw. im Betriebssystem hat für jede Sperre eine Liste aller wartenden Transaktionen/Threads. Wenn Transaktion A die Sperre freigibt, weckt der "Lock Manager" B wieder auf.

Sperren (2)

Transaktion A	Transaktion B

`UPDATE KONTO`
`SET STAND = STAND + 10`
`WHERE NR = 1001`
$\longrightarrow$ 1 row updated.	

	`UPDATE KONTO`
	`SET STAND = STAND + 20`
	`WHERE NR = 1001`
	$\longrightarrow$ (keine Reaktion)

`COMMIT`	
	$\longrightarrow$ 1 row updated. `COMMIT`

Sperren (3)

- Warum kann Transaktion B nicht sofort ausgeführt werden?
  - Die Erhöhung des Kontostands wird als Lesezugriff gefolgt von einem Schreibzugriff behandelt.

    Es wäre auch möglich, "Increment" als Basisoperation zu betrachten. Dann müßte man nicht unbedingt abwarten, bis Transaktion A beendet ist. Dies geht aber nur in Spezialsystemen.
  - Der Lesezugriff hat kein eindeutiges Ergebnis, solange Transaktion A noch läuft.
  - Transaktion A könnte ja z.B. noch mit `ROLLBACK` abgebrochen werden.

Sperren (4)

- Warum bekommt Transaktion B keinen Hinweis?
  - Dann müsste der Fall “Tupel gesperrt” im Anwendungsprogramm speziell behandelt werden.
  - So braucht der Datenbank-Aufruf, der normalerweise vielleicht 10 ms braucht, ausnahmsweise einmal etwas länger (z.B. einige Sekunden).
  - Die Logik des Anwendungsprogramms ist davon überhaupt nicht betroffen.

    Wenn man aber wünscht, kann man Optionen setzen, so dass man statt der Verzögerung eine Fehlermeldung erhält.

Typen von Sperren (1)

- Die meisten DBMS haben (mindestens) zwei Arten von Sperren:
  - Schreibsperren ("exclusive locks", "X-locks") werden vor einem Schreibzugriff gesetzt.

    Sie schließen jeden anderen Zugriff aus (Lesen oder Schreiben).
  - Lesesperren ("shared locks", "S-locks") werden vor einem Lesezugriff gesetzt.

    Sie schließen Schreibzugriffe aus, aber erlauben Lesezugriffe von anderen Transaktionen (Lesesperren sind Sperren zum Zwecke des Lesens, nicht Sperren, die Lesezugriffe verbieten!).

Typen von Sperren (2)

- Die Wirkungsweise der verschiedenen Sperrentypen wird in einer Kompatibilitätsmatrix veranschaulicht:

Angeforderte Sperre	Existierende Sperre		
	Keine	S	X
S	+	+	−
X	+	−	−

Deadlocks (1)

- Hier warten zwei Transaktionen auf Sperren, die die jeweils andere Transaktion hält:

Transaktion A	Transaktion B
`UPDATE KONTO ...` `WHERE NR = 1001`	
	`UPDATE KONTO ...` `WHERE NR = 2345`
	`UPDATE KONTO ...` `WHERE NR = 1001`
`UPDATE KONTO ...` `WHERE NR = 2345`	

Deadlocks (2)

- In diesem Fall muss eine der am Deadlock beteiligten Transaktionen abgebrochen werden (`ROLLBACK`).

  Dabei werden die von dieser Transaktion gehaltenen Sperren freigegeben, so dass die andere Transaktion forgesetzt werden kann. Oracle führt das Rollback nicht automatisch aus, sondern liefert einer der beiden Transaktionen für das `UPDATE` eine Fehlermeldung. Das Anwendungsprogramm sollte dann `ROLLBACK` aufrufen. Dies zeigt, dass man immer auf Fehler gefasst sein muss, selbst wenn man "alles richtig gemacht hat" und beim Testen nie ein Fehler aufgetreten ist.

- Natürlich ist ein Deadlock auch mit mehr als zwei Transaktionen möglich (zyklisches Warten).

Deadlocks (3)

- Der Deadlock-Test ist ziemlich aufwendig, deswegen führen ihn manche Systeme nur von Zeit zu Zeit aus (oder erst nachdem eine Transaktion etwas länger auf eine Sperre gewartet hat).
- Deadlocks könnten vermieden werden, wenn Sperren immer in einer bestimmten Reihenfolge angefordert würden.

  Z.B. könnte man bei Überweisungen immer auf die kleinere Kontonummer zuerst zugreifen (anstatt immer die Abbuchung zuerst ausführen).

Inhalt

Dirty Read Problem (1)

- Transaktion A setzt den Kontostand auf 1 000 000, und erkennt dann den Fehler. B berechnet Zinsen.

Transaktion A	Transaktion B

`UPDATE KONTO`
`SET STAND = 1 000 000`
`WHERE NR = 1001`	

	`SELECT STAND`
	`FROM KONTO`
	`WHERE NR = 1001`
	$\longrightarrow$ 1 000 000

`ROLLBACK`	**(Passiert so nicht)**

Dirty Read Problem (2)

- In obigem Schedule sieht B Daten, die eigentlich niemals offiziell existierten.

  Transaktionen werden ganz oder gar nicht ausgeführt. Das `ROLLBACK` soll jede Spur der Transaktion beseitigen.

- Keine Transaktion sollte einen Zwischenzustand einer anderen Transaktion sehen.

  Es ist auch ein Dirty Read, wenn Transaktion A den Konstostand später erneut ändert (mit `UPDATE` korrigiert) und dann `COMMIT` aufruft.

- Transaktionen sollten einen Zustand sehen, der das Ergebnis einer Folge von mit `COMMIT` bestätigten Transaktionen ist (plus die eigenen Änderungen).

Dirty Read Problem (3)

- Es ist nicht schwierig, Dirty Reads auszuschließen, und die meisten DBMS machen das auch.
- Der Schedule auf Folie 22 kann in modernen DBMS nicht vorkommen.

  Der Programmierer braucht sich über Dirty Reads keine Gedanken zu machen.
- Es gibt im wesentlichen zwei Lösungen für das Dirty Read Problem, die je nach DBMS benutzt werden:
  - Schreibsperren auf veränderte Tupel.
  - "Multi-Version concurrency control".

Dirty Read Problem (4)

Lösung mit Sperren:

- Das System setzt Schreibsperren auf die von einer Transaktion geänderten Tupel und hält sie bis zum Transaktionsende.

  Die Sperren werden vor der Änderung gesetzt und erst nach dem `COMMIT` entfernt. Daher sind nicht mit `COMMIT` bestätigte Daten für andere Transaktionen nicht zugreifbar.

- Eine Transaktion, die ein Tupel lesen will, fordert dafür eine Lesesperre an. Dies geht nur, wenn es für das Tupel keine Schreibsperre gibt.

  Wenn man nur Dirty Reads ausschliessen will, kann man die Lesesperre sofort wieder löschen, nachdem man das Tupel gelesen hat.

Dirty Read Problem (5)

"Multi Version Concurrency Control" (Oracle, PostgreSQL):

- Bei beiden Systemen bezieht sich eine Anfrage auf den Zustand, der genau die beim Start der Anfrage mit `COMMIT` bestätigten Änderungen enthält.

  Für Lesezugriffe stellt Oracle alte Versionen der Daten wieder her, die dem Zustand nach der letzten mit `COMMIT` bestätigten Transaktion entsprechen.
  PostgreSQL bewahrt alte Versionen von Tupeln auf (nichts wird direkt überschrieben, Updates erzeugen eine neue Version des Tupels).
  Beim "Vacuum Cleaning" wird Speicher von nicht mehr benötigten Tupeln freigegeben (Hintergrundprozess, auch explizit möglich).

- So ist ein konsistenter Zustand garantiert, selbst für Anfragen, die lange laufen.

  Von einer Anfrage bis zur nächsten kann sich der Zustand dagegen ändern ("non-repeatable read problem").

Dirty Read Problem (6)

Transaktion A	Transaktion B
	`SELECT STAND`
	`FROM KONTO`
	`WHERE NR = 1001`
	$\longrightarrow$ 50
`UPDATE KONTO`	
`SET STAND = STAND + 30`	
`WHERE NR = 1001`	
`SELECT STAND ...` $\longrightarrow$ 80	
	`SELECT STAND ...` $\longrightarrow$ 50
`COMMIT`	
	`SELECT STAND ...` $\longrightarrow$ 80

Lost Update Problem (1)

- Angenommen, die folgenden Updates laufen parallel, und der Kontostand ist vorher 100:

Transaction A	Transaction B

`UPDATE KONTO`
`SET STAND = STAND + 20`
`WHERE NR = 1001`
	`UPDATE KONTO`
	`SET STAND = STAND - 50`
	`WHERE NR = 1001`

- Der Kontostand hinterher muss 70 sein.
- Intern muss das DBMS die Daten von der Platte in den Hauptspeicher lesen, dort ändern, und dann zurückschreiben. Dabei muss man aufpassen.

Lost Update Problem (2)

- Folgende Abfolge muss ausgeschlossen werden:

Transaktion A	Transaktion B
	`read(X, 'KONTO ...');`
	$\longrightarrow$ X=100
`read(Y, 'KONTO ...');`	
$\longrightarrow$ Y=100	
`Y := Y + 20;`	
`write(Y, 'KONTO ...');`	
	`X := X - 50;`
	`write(X, 'KONTO ...');`

- Der zweite Schreibzugriff überschreibt das Ergebnis des ersten, damit ist der Kontostand am Ende 50.

Lost Update Problem (3)

- Solche "Lost Updates" werden von heutigen DBMS ausgeschlossen.
- Z.B. wird das Tupel für Konto 1001 exklusiv gesperrt, bevor der Wert gelesen wird.

  Manche DBMS haben spezielle Update-Sperren. Zuerst eine Lesesperre anzufordern, und diese später zu einer Schreibsperre zu verstärken, wäre schlecht, da das leicht zu Deadlocks führt (auch im Beispiel).

- Wenn also Transaktion B die Sperre zuerst bekommt, müßte Transaktion A auch mit dem Lesen warten, bis B fertig ist (`COMMIT` ausgeführt hat).

  Die Sperre wird bis zum `COMMIT` gehalten, um Dirty Reads zu vermeiden.

Lost Update Problem (4)

- Lost Updates werden nur dann automatisch verhindert, wenn `UPDATE` wie oben gezeigt verwendet wird (Lesen und Schreiben in einem Kommando).
- Wenn man für eine komplexere Berechnung zuerst den alten Wert mit `SELECT` liest, und dann den neuen Wert mit `UPDATE` zurückschreibt, können Lost Updates vorkommen.

  Das Problem ist, dass `SELECT` die gelesenen Tupel normalerweise nicht sperrt (oder die Sperre nach dem `SELECT` gleich freigibt). Sperren auf gelesenen Tupeln immer bis zum Ende der Transaktion zu halten, würde die Parallelität zu stark beschränken (und ist oft nicht nötig).

Lost Update Problem (5)

Transaktion A	Transaktion B

	`SELECT STAND`
	`FROM KONTO`
	`WHERE KONTO = 1001`
	$\longrightarrow$ 100

`SELECT ...` $\longrightarrow$ 100	

`UPDATE KONTO`
`SET STAND = 120`
`WHERE NR = 1001`	

`COMMIT`	

	`UPDATE KONTO`
	`SET STAND = 50`
	`WHERE NR = 1001`

	`COMMIT`

Lost Update Problem (6)

- Der obige Schedule mit einem Lost Update ist in Oracle und anderen DBMS nicht ausgeschlossen.
- Um ihn zu vermeiden, muss man "`FOR UPDATE`" zu allen Anfragen hinzufügen, deren Ergebnis eventuell hinterher in ein Update eingeht:

```
SELECT STAND
FROM   KONTO
WHERE  NR = 1001
FOR UPDATE
```

- Dadurch werden alle Tupel gesperrt, die die `WHERE`-Bedingung zum Zeitpunkt der Anfrage erfüllen.

Lost Update Problem (7)

- Wie beim richtigen Update werden "`FOR UPDATE`" Sperren bis zum Transaktionsende gehalten.
- `FOR UPDATE` ist nur bei einfachen Anfragen erlaubt.

  Das DBMS muss in der Lage sein, festzustellen, welche Tupel gesperrt werden sollen. Oracle erlaubt bestimmte Verbunde, aber keine Aggregationen, `DISTINCT`, `UNION`. Im allgemeinen kann `FOR UPDATE` benutzt werden, wenn die Anfrage eine updatebare Sicht definieren würde.
- Man kann beim "`FOR UPDATE`" ein Attribut angeben:

  ```
  FOR UPDATE OF STAND
  ```

  Dies ist für DBMS gedacht, die einzelne Tabelleneinträge sperren. In Oracle, das Verbunde in den Anfragen erlaubt, definiert das Attribut, von welcher Tabelle Tupel gesperrt werden sollen.

Lost Update Problem (8)

- "Lost Updates" können z.B. auch auftreten, wenn
  - man einem Benutzer Daten aus der Datenbank anzeigt (etwa in einem Web-Formular),
  - ihn/sie die Daten ändern läßt, und dann
  - die neuen Daten ohne Prüfung zurückschreibt.
- Sperren sind hier ungünstig, da nicht klar ist, ob / wann der Benutzer veränderte Daten zurückschickt.
- Z.B. merkt man sich die alten Werte (in versteckten Feldern) und prüft dann beim Speichern, ob die Werte in der DB noch unverändert sind.

Lost Update Problem (9)

- Hier werden auch Daten überschrieben, aber das DBMS ist unschuldig: Der Schedule ist seriell.
  $\Rightarrow$ **Kein Lost Update Problem!**

Transaktion A	Transaktion B

`UPDATE KONTO`
`SET STAND = 100`
`WHERE NR = 1001`
`COMMIT`
	`UPDATE KONTO`
	`SET STAND = 0`
	`WHERE NR = 1001`
	`COMMIT`

Lost Update Problem (10)

Aufgabe:

- Angenommen, die folgenden Update-Aufträge erreichen das DBMS mehr oder weniger gleichzeitig:

Transaktion A	Transaktion B

`UPDATE KONTO`
`SET STAND = STAND+20`
`WHERE NR = 1001`
`COMMIT`
	`UPDATE KONTO`
	`SET STAND = STAND*1.05`
	
	`COMMIT`

- Angenommen, der Kontostand ist vorher 100 Euro.
- Was wäre ein korrektes Verhalten des DBMS?

Nonrepeatable Read (1)

Transaktion A	Transaktion B

`SELECT STAND`
`FROM KONTO`
`WHERE NR = 1001`
$\longrightarrow$ 100	

	`UPDATE KONTO`
	`SET STAND = STAND + 50`
	`WHERE NR = 1001`
	`COMMIT`

`SELECT STAND`
`FROM KONTO`
`WHERE NR = 1001`
$\longrightarrow$ 150	

Nonrepeatable Read (2)

- In vielen DBMS (z.B. Oracle, PostgreSQL) ist es möglich, dass man verschiedene Antworten bekommt, wenn man die gleichen Daten zweimal abfragt.
- Dieses Verhalten kann in einem seriellen Schedule nicht vorkommen, verletzt also die Isolation.

  Das gleiche Problem ist eigentlich die Ursache für den Lost Update, wenn man den Update in ein `SELECT` und ein `UPDATE` aufspaltet (siehe oben). Natürlich ist es unwahrscheinlich, dass ein Benutzer genau die gleiche Anfrage innerhalb einer Transaktion zweimal stellt. Aber er/sie könnte auf überlappende Tupelmengen zugreifen.

Nonrepeatable Read (3)

- Das DBMS kann dieses Problem vermeiden, indem es die Lesesperren auf den zugegriffenen Tupeln bis zum Ende der Transaktion hält.

  Normalerweise werden sie direkt nach dem Lesen wieder freigegeben, um mehr Parallelität zu ermöglichen.

- Als Benutzer kann man
  - die "`FOR UPDATE`"-Klausel dafür verwenden, oder
  - die Isolationsstufe hochsetzen (s.u., Folie 49).

Inconsistent Analysis (1)

- Angenommen, die Bank speichert aus Leistungsgründen die Summe aller Konten redundant in einer Tabelle `GELDBESTAND(BETRAG)` (mit nur einer Zeile).
- Dann sollten die folgenden Anfragen immer das gleiche Ergebnis liefern:
  - `SELECT SUM(STAND) FROM KONTO`
  - `SELECT BETRAG FROM GELDBESTAND`
- Wenn aber beide Anfragen nacheinander ausgeführt werden, gibt es keine Garantie, dass die Ergebnisse sich wirklich auf den gleichen Zustand beziehen.

Inconsistent Analysis (2)

Transaktion A	Transaktion B

`SELECT SUM(STAND)`
`FROM KONTO`
$\longrightarrow$ 1000	

	`UPDATE KONTO`
	`SET STAND = STAND+50`
	`WHERE NR = 1001`

	`UPDATE GELDBESTAND`
	`SET BETRAG = BETRAG+50`
	`COMMIT`

`SELECT BETRAG`
`FROM GELDBESTAND`
$\longrightarrow$ 1050	

Inconsistent Analysis (3)

- "Inconsistent Analysis" und "Nonrepeatable Read" sind recht ähnlich: In beiden Fällen ändert sich der Zustand zwischen zwei Anfragen.
- Beim "Inconsistent Analysis" Problem wird aber in den Anfragen auf verschiedene Tupel zugegriffen.

  Im Beispiel greift die Transaktion A, die die Konsistenz der redundanten Daten überwachen soll, auf kein Tupel zwei Mal zu.
- Auch hier reicht es aus, Sperren auf allen gelesenen Tupeln bis zum Transaktionsende zu halten.

  B könnte sich nicht dazwischen schieben. Eine Einfügung (Kontoeröffnung) wäre möglich, aber das wäre schon das Phantom-Problem.
  Der SQL-Standard betrachtet "Inconsistent Analysis" nicht getrennt.

Inconsistent Analysis (4)

- Da (mindestens bei Oracle, PostgreSQL) garantiert ist, dass eine Anfrage immer bezüglich eines festen Zustands ausgewertet wird, könnte man die beiden Anfragen kombinieren:

```
SELECT SUM(KONTO) AS BETRAG, 'Summe' AS TEIL
FROM   KONTO
UNION  ALL
SELECT BETRAG, 'Geldbestand' AS TEIL
FROM   GELDBESTAND
```

- Ansonsten muss man die beiden Tabellen vor der Analyse manuell sperren (s.u., Folie 48).

Phantom Problem (1)

- Angenommen, die Bank hat 1000 Euro übrig, die sie als Bonus-Zahlung gleichmäßig über alle Konten verteilen möchte.

Transaction A	Transaction B

`SELECT COUNT(*)`
`FROM KONTO`
$\longrightarrow$ 200	

	`INSERT INTO KONTO` `VALUES (...)`
	`COMMIT`
`UPDATE KONTO` `SET STAND = STAND + 5`	

Phantom Problem (2)

- Der neue Kunde profitiert gleich von der Bonus-Zahlung, so dass am Ende 1005 Euro ausgegeben wurden (der Bank fehlen 5 Euro).
- Es würde hier nichts helfen, zunächst alle Tupel zu sperren:

```
SELECT * FROM KONTO
FOR UPDATE
```

- Obwohl alle existierenden Zeilen gesperrt sind, ist die Einfügung einer neuen Zeile weiter möglich.

Sperren auf einzelnen Zeilen können ein `INSERT` nicht verhindern.

Phantom Problem (3)

- Wenn die Anfrage eine Bedingung enthalten würde (z.B. Bonus-Zahlung nur bei `KREDITRAHMEN >= 1000`), könnte auch ein Update ein Phantom-Problem erzeugen.
- Um das Phantom-Problem zu verhindern, braucht man, dass die Menge der Tupel, die eine Bedingung erfüllen, konstant bleiben.
- In der Theorie wurden Prädikat-Sperren vorgeschlagen, aber der Konflikt-Test ist zu aufwendig, so dass sie sich nicht durchsetzen konnten.
- Da es immer möglich ist, mehr Tupel als nötig zu sperren, kann man Prädikat-Sperren mit Sperren in Indexen approximieren (z.B. in B-Baum Index Intervall sperren).

LOCK TABLE

- Man kann das Phantom-Problem ausschließen, indem man die ganze Tabelle manuell sperrt:

  ```
  LOCK TABLE KONTO IN EXCLUSIVE MODE
  ```

  Bis zum `COMMIT`/`ROLLBACK` kann kein anderer auf die Tabelle zugreifen.

- Dieser Befehl ist nicht im SQL-92 Standard enthalten (Sperren zur Synchronisation zu verwenden, ist systemspezifisch, es gibt auch andere Verfahren).

  "`LOCK TABLE`" funktioniert z.B. in Oracle, PostgreSQL und DB2. MySQL verwendet eine andere Syntax: `LOCK TABLES` T_1 `WRITE,` T_2 `READ` (dieses Kommando gibt alle früheren Sperren frei, so dass Deadlocks vermieden werden). "`LOCK TABLE`" funktioniert nicht in SQL Server and Access.

Isolationsstufen (1)

- Anstelle von Sperren erlaubt SQL-92, eine Isolationsstufe mit folgenden Kommando zu wählen:

  `SET TRANSACTION ISOLATION LEVEL` ⟨`Level`⟩

- Der SQL Standard kennt vier Isolationsstufen:
  - `READ UNCOMMITTED`: Die Transaktion kann den DB-Zustand lesen, ohne auf Sperren zu warten.

    Um z.B. Datenverteilungen für den Optimierer zu berechnen, braucht man nur ungefähre Werte. Das "Dirty Read" Problem, das hier auftreten kann, ist für diese Anwendung nicht schädlich.
  - `READ COMMITTED`: Standardfall, wie oben erklärt.

    Lesesperren werden nach dem Lesezugriff wieder freigegeben.

Isolationsstufen (2)

- Isolationsstufen, Fortsetzung:
  - `REPEATABLE READ`: Hier werden auch die Lesesperren erst am Transaktionsende freigegeben.

    Dies schützt nicht vor dem Phantom-Problem und auch nicht vor dem "Inconsistent Analysis" Problem.
  - `SERIALIZABLE`: Das theoretische Ideal vollständiger Isolation. Dies schließt insbesondere auch das Phantom-Problem aus.
- Oracle unterstützt nur
  - "`READ COMMITTED`" (dies ist der Default) und
  - "`SERIALIZABLE`".

Isolationsstufen (3)

Isolationsstufe	Dirty Read	Nonrepeat-able Read	Phantom Problem
`READ UNCOMMITTED`	möglich	möglich	möglich
`READ COMMITTED`	möglich	möglich	—
`REPEATABLE READ`	möglich	—	—
`SERIALIZABLE`	—	—	—

Bei der Isolationsstufe "`SERIALIZABLE`" darf also keins der drei Probleme mehr auftreten. Ein DBMS darf natürlich immer mehr Schutz beim Mehrbenutzerbetrieb liefern als es muss. Z.B. ist der SQL-Standard auch erfüllt, wenn auch bei "`READ UNCOMMITTED`" tatsächlich keine "Dirty Reads" auftreten. Aber man kann sich eben nicht darauf verlassen.

“Serializable” in Oracle8

- In Oracle8 gibt `'SERIALIZABLE'` nur sehr wenig Parallelität und doch nicht die volle Serialisierbarkeit.
- Beispiel: Gegeben zwei Tabellen `R(A)` und `S(A)`, jede mit nur einer Zeile mit dem Wert `'old'`:

Transaktion A	Transaktion B
`SELECT A FROM R`	
$\longrightarrow$ `old`	
`UPDATE S SET A='new'`	
	`SELECT A FROM S`
	$\longrightarrow$ `old`
	`UPDATE R SET A='new'`
	`COMMIT`
`COMMIT`	

Inhalt

Sperren in PostgreSQL (1)

- PostgreSQL erlaubt im `LOCK TABLE` folgende Modi:

  [https://www.postgresql.org/docs/9.4/sql-lock.html]
  [https://www.postgresql.org/docs/9.4/explicit-locking.html]

  - `ACCESS SHARE` (automatisch bei jedem `SELECT`)
  - `ROW SHARE` (automatisch bei `SELECT FOR UPDATE`)
  - `ROW EXCLUSIVE` (automatisch bei `UPDATE` etc.)
  - `SHARE UPDATE EXCLUSIVE` (bei `ALTER TABLE` etc.)
  - `SHARE` (automatisch bei `CREATE INDEX`)
  - `SHARE ROW EXCLUSIVE` (wie `SHARE` plus `ROW EXCLUSIVE`)
  - `EXCLUSIVE`
  - `ACCESS EXCLUSIVE` (automatisch bei `DROP TABLE` etc.)

Sperren in PostgreSQL (2)

Exist. Sperre	Angeforderte Sperre							
	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPD. EX.	SHARE	SHARE ROW EX.	EXCL.	ACCESS EXCL.
ACCESS SHARE	+	+	+	+	+	+	+	−
ROW SHARE	+	+	+	+	+	+	−	−
ROW EXCL.	+	+	+	+	−	−	−	−
SHARE UPD. EX.	+	+	+	−	−	−	−	−
SHARE	+	+	−	−	+	−	−	−
SHARE ROW EX.	+	+	−	−	−	−	−	−
EXCL.	+	−	−	−	−	−	−	−
ACCESS EXCL.	−	−	−	−	−	−	−	−

Sperren in PostgreSQL (3)

- Aufgrund der "Multi-Version Concurrency Control" sperren `SELECT`-Anfragen bei PostgreSQL keine Tupel.
- Sie fordern aber eine `ACCESS SHARE` Sperre auf der Tabelle an.
- Dies ist eine sehr schwache Sperre, die nur mit `ACCESS EXCLUSIVE` (der stärksten Sperre) in Konflikt steht.
- `ACCESS EXCLUSIVE` wird u.a. beim `DROP TABLE` angefordert, und schließt jegliche anderen Zugriffe auf die Tabelle aus.

  Z.B. fordern auch `TRUNCATE` und einige `ALTER TABLE` Kommandos diese Sperre an. Bei allzu dramatischen Änderungen der Tabellenstruktur funktioniert auch der Zugriff auf alte Tupel-Versionen nicht mehr.

Sperren in PostgreSQL (4)

- Wenn man Sperren auf Objekten verschiedener Granularitäten hat (z.B. Tabellen und Tupel), benötigt man zuerst ein "Intent Lock" für die höhere Ebene, bevor man eine Sperre auf der tieferen Ebene anfordern kann.
- Z.B. fordert ein `UPDATE` exklusive Sperren auf den geänderten Tupeln an.
- Es muss verhindert werden, dass jemand anders gleichzeitig die ganze Tabelle im `SHARE` oder `EXCLUSIVE` Modus sperrt.

  Der Konflikt zwischen den Sperren auf Tupelebene und auf Tabellenebene muss erkannt werden.

- Deswegen sperrt das `UPDATE` die Tabelle "`ROW EXCLUSIVE`".

  Zwei "`ROW EXCLUSIVE`" Sperren sind kompatibel, weil die eigentlichen Sperren auf Tupelebene sich auf verschiedene Tupel beziehen können.

Sperren in PostgreSQL (5)

- Da "`SELECT ... FOR UPDATE`" selbst noch nichts ändert, kann das noch mit einer `SHARE` Sperre auf der ganzen Tabelle kompatibel sein. Deswegen gibt es den Sperr-Moduls "`ROW SHARE`" (auch ein "Intent Lock").

  Es ist ja unklar, ob und wann der Benutzer nach einem "`FOR UPDATE`" tatsächlich ein `UPDATE` ausführt. Wenn er das macht, muss er warten, bis die `SHARE`-Sperre des anderen Nutzers auf der Tabelle freigegeben wurde.

- Weil jede Transaktion auf jeder Tabelle nur eine Sperre haben kann, wird ggf. der Sperrmodus auf den stärkeren der beiden Modi gesetzt, wenn sie schon eine Sperre hat und auf der gleichen Tabelle eine Sperre neu anfordert.
- Der einzige Fall, in dem nicht einer von beiden Modi stärker ist als der andere, sind "`SHARE`" und "`ROW EXCLUSIVE`". Deswegen gibt es dafür einen kombinierten Modus.

Inhalt

Transaktionen (1)

- Transaktionen können formalisiert werden als Folgen von Operationen der folgenden Typen:
  - `read(`x`)`: Eine Kopie von Objekt x aus der DB wird der Transaktion zur Verfügung gestellt.

    Objekte können z.B. Tabellenzeilen oder Plattenblöcke sein.
  - `write(`x`)`: Ersetze die aktuelle Version von Objekt x in der Datenbank durch eine neue Version.

    Natürlich hat `write` einen zweiten Parameter für den neuen Wert.

    Für diese Theorie ist aber nur wichtig, dass x geschrieben wird.
  - `rollback`: Alle Änderungen zurücknehmen.
  - `commit`: Alle Änderungen dauerhaft machen.

Transaktionen (2)

- Jede Transaktion muss mit einem der Kommandos `commit` oder `rollback` enden, und diese Kommandos sind auch nur als letztes Element der Folge erlaubt.

  Für die Definition der (Konflikt-)Serialisierbarkeit könnte man das Kommando `rollback` durch `write(`x`)` für alle in der Transaktion geschriebenen Objekte ersetzen (es setzt die Werte ja auf die alten Werte zurück). Das Kommando `commit` könnte man ganz weglassen (es ändert die Werte der Objekte nicht). Daher ist es auch möglich, als einzige Operationen in Schedules `read(`x`)` und `write(`x`)` zu betrachten. Allerdings werden die Operationen `commit` und `rollback` in der Praxis verwendet, und wären z.B. für einen Transaktionsmanager, der mit Sperren arbeitet, wichtig. In den hier folgenden Definitionen sind sie dagegen nicht wirklich wichtig.

Transaktionen (3)

- Der Transaktions-Manager im DBMS weiß nicht, wie bei `write(`x`)` der neue Wert von x von der Transaktion berechnet wurde.

  Er kennt den Programmcode nicht, hat keine Formel für diesen Wert.

- Er muss daher annehmen, dass der neue Wert von x möglicherweise von allen Objekten abhängt, die die Transaktion vorher gelesen hat.

Transaktionen (4)

- Dieses formale Modell nimmt an, dass jede Transaktion explizit angibt, welche Objekte (Tupel) sie lesen oder schreiben will.
- Dies ist eine Vereinfachung der Realität: In SQL gibt man eine Bedingung für die zu lesenden oder zu ändernden Tupel an.
- Z.B. kann das Phantom-Problem in diesem Modell gar nicht untersucht werden.

  Natürlich gibt es kompliziertere formale Modelle, die auch Operationen für das Lesen oder Schreiben einer Menge von Objekten haben, die über eine Bedingung spezifiziert wird.

Schedules (1)

- Sei eine endliche Menge $\{T_1, \ldots, T_n\}$ gegeben, deren Elemente Transaktionen heißen, und für jedes T_i eine Folge $c_{i,1} \ldots c_{i,m_i}$ von Kommandos.
- Sei weiter $\mathcal{S}$ die Menge der Tripel $s = (T_i, j, c_{i,j})$ mit $1 \leq i \leq n$ und $1 \leq j \leq m_i$ (auszuführende Schritte).

  Menge der Kommandos aller Transaktionen mit Positionsinformation.

- Die Transaktionen definieren eine partielle Ordnung auf $\mathcal{S}$: $s \prec s'$ gdw. s und s' zu der gleichen Transaktion gehören und s vor s' in der Transaktion kommt, d.h. $s = (T_i, j, c_{i,j})$ und $s' = (T_i, k, c_{i,k})$ mit $j < k$.

Schedules (2)

- Ein Schedule (Historie) dieser Transaktionen ist eine lineare Ordnung $<$ auf $\mathcal{S}$, die mit $\prec$ verträglich ist (d.h. wenn $s \prec s'$, dann $s < s'$).
- Ein Schedule definiert also eine Reihenfolge $s_1 \ldots s_l$ von Schritten, die jedes Element von $\mathcal{S}$ genau einmal enthält, und die Ordnung der Schritte innerhalb einer Transaktion berücksichtigt (wenn $s_i \prec s_j$, dann $i < j$).
- Ein Schedule ist also eine Verschachtelung der einzelnen Schritte der Transaktionen.

Schedules (3)

- Beispiel: Angenommen, T_1 und T_2 wollen beide ein Objekt `A` ändern, also haben sie beide die Folge von Operationen: `read(A)`, `write(A)`, `commit`.
- Dann ist ein Schedule (Beispiel für Lost Update):

T_1: `read(A)`,
T_2: `read(A)`,
T_2: `write(A)`,
T_2: `commit`,
T_1: `write(A)`,
T_1: `commit`.

T_1	T_2
`read(A)`	
	`read(A)`
	`write(A)`
	`commit`
`write(A)`	
`commit`	

Schedules (4)

- Serielle Schedules sind Schedules, die jede Transaktion in einem Stück ausführen, d.h. für alle Schritte $s_1 < s_2 < s_3$ gilt: Wenn s_1 und s_3 zur gleichen Transaktion T_i gehören, dann muss s_2 auch zu T_i gehören.

T_1	T_2
`read(A)`	
`write(A)`	
`commit`	
	`read(A)`
	`write(A)`
	`commit`

T_1	T_2
	`read(A)`
	`write(A)`
	`commit`
`read(A)`	
`write(A)`	
`commit`	

Serialisierbarkeit

- Informell ist ein Schedule serialisierbar gdw. er äquivalent zu einem seriellen Schedule ist.
- Äquivalent bedeutet:
  - Die Leseoperationen aller Transaktionen liefern die gleichen Werte in beiden Schedules.
  - Am Ende wird der gleiche DB-Zustand erreicht (identische Werte für alle Objekte).
- Das DBMS muss obige Serialisierbarkeit garantieren. Da es die Berechnung der neuen Werte nicht kennt, kann es die Schedules weiter einschränken.

Konflikt-Serialisierbarkeit (1)

- Man definiert nun eine Konflikt-Relation zwischen Schritten in Transaktionen. Zwei Schritte stehen in Konflikt, wenn die Operationen nicht vertauscht werden können, also ihre Reihenfolge wichtig ist:
  - T_i: `write(`x`)` steht in Konflikt mit T_j: `write(`x`)`,
  - T_i: `write(`x`)` steht in Konflikt mit T_j: `read(`x`)`,
  - T_i: `rollback` steht in Konflikt mit T_j: `read(`x`)` und T_j: `write(`x`)`, wenn `write(`x`)` in T_i enthalten ist.

    D.h. `rollback` schreibt alle Objekte, die in der Transaktion modifiziert wurden: Es muss sie auf auf den alten Wert zurücksetzen.
  - und jeweils auch umgekehrt.

Konflikt-Serialisierbarkeit (2)

- Sei $s_1 \ldots s_k$ ein Schedule. Eine erlaubte elementare Modifikation des Schedules ist es, zwei Schritte zu vertauschen, die direkt aufeinander folgen, also $s_1 \ldots s_i \; s_{i+1} \ldots s_k$ in $s_1 \ldots s_{i+1} \; s_i \ldots s_k$ umzuwandeln, wobei s_i and s_{i+1} nicht in Konflikt stehen.
- Ein Schedule heißt konflikt-serialisierbar gdw. er durch erlaubte elementare Modifikationen in einen seriellen Schedule überführt werden kann.

  Konflikt-Serialisierbarkeit impliziert die Äquivalenz zu einem seriellen Schedule. Das Umgekehrte gilt nicht, weil z.B. zwei Schreiboperationen zufällig den gleichen Wert schreiben können.

Konflikt-Serialisierbarkeit (3)

Beispiel/Aufgabe:

- Der folgende Schedule ist konflikt-serialisierbar:

T_1	T_2
	`read(A)`
`read(A)`	
`read(B)`	
	`write(A)`
`write(B)`	
	`commit`
`commit`	

- Überführen Sie diesen Schedule durch erlaubte elementare Modifikationen in einen seriellen Schedule.

Konflikt-Serialisierbarkeit (4)

- Ein einfacher Test für die Konflikt-Serialisierbarkeit eine Schedules ist es, seinen Konflikt-Graphen zu konstruieren (und diesen auf Zyklen zu testen):
  - Die Knoten des Graphen sind die Transaktionen.
  - Es gibt eine Kante von T_i zu T_j ($i \neq j$) gdw. es im Schedule Schritte s von T_i und s' von T_j mit $s < s'$ gibt, so dass s and s' in Konflikt stehen.

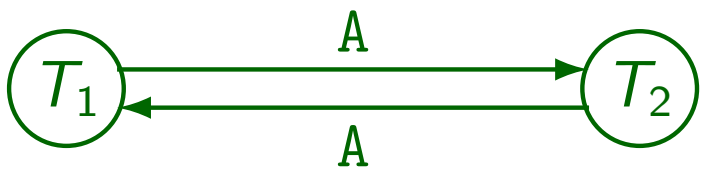
    Man sollte an die Kante die Objekte schreiben, die den Konflikten zugrunde liegen (oder mindestens eines dieser Objekte, um die Kante zu begründen). In Übungsaufgaben und Klausuren ist das oft verlangt, für den eigentlichen Serialisierbarkeitstest ist es nicht wichtig.

Konflikt-Serialisierbarkeit (5)

- Um keinen Konflikt zu übersehen, kann man den Graphen z.B. folgendermaßen aufbauen:
  - Man geht die beteiligten Objekte nacheinander durch, und sucht zuerst die `write`-Anweisungen für das aktuelle Objekt.
  - Für jede `write`-Anweisung erzeugt man Kanten von/zu den anderen Transaktionen, die auf das gleiche Objekt zugreifen.
  - Die Reihenfolge im Schedule bestimmt die Richtung der Kante.

Konflikt-Serialisierbarkeit (6)

- Beispiel: Konfliktgraph für den Schedule auf Folie 66 (mit Lost Update):



- Satz: Ein Schedule ist konflikt-serialisierbar gdw. sein Konfliktgraph keine Zyklen enthält.

  Und topologische Sortierung liefert äquivalente serielle Schedules.

  Beachte: Es ist nicht verlangt, dass das gleiche Objekt an allen Kanten des Zyklus steht (die Kantenbeschriftung ist für diesen Test irrelevant, sie dokumentiert nur den Grund für die Kante). Es ist natürlich auch nicht verlangt, dass der Zyklus alle Knoten des Graphen beinhaltet.

Konflikt-Serialisierbarkeit (7)

Aufgabe:

- Ist dieser Schedule konflikt-serialisierbar?

T_1	T_2	T_3
`read(A)`		
	`read(B)`	
	`write(B)`	
		`read(A)`
		`write(C)`
	`read(C)`	
`read(B)`		
		`write(D)`
		`commit`
`commit`		
	`commit`	

Literatur/Quellen

- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Date/Darwen: A Guide to the SQL Standard, Fourth Edition, Addison-Wesley, 1997.
- Sunderraman: Oracle Programming, A Primer. Addison-Wesley, 1999.
- Oracle8 SQL Reference, Oracle Corporation, 1997, Part No. A58225-01.
- Oracle8 Concepts, Release 8.0, Oracle Corporation, 1997, Part No. A58227-01.
- H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil: A critique of ANSI SQL isolation levels. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1–10, 1995.
- PostgreSQL Documnetation: SQL Commands: `LOCK` [https://www.postgresql.org/docs/9.4/sql-lock.html]
- PostgreSQL Documentation: 13.3 Explicit Locking [https://www.postgresql.org/docs/9.4/explicit-locking.html]
- PostgreSQL Wiki: Lock Monitoring [https://wiki.postgresql.org/wiki/Lock_Monitoring]
- Igor Sarcevic: Selecting for Share and Update in PostgreSQL [http://shiroyasha.io/selecting-for-share-and-update-in-postgresql.html]