Teil 12: Updates in SQL

Literatur:

- Elmasri/Navathe:Fundamentals of Database Systems, 3rd Edition, 1999. Chap. 8, "SQL — The Relational Database Standard"
- Kemper/Eickler: Datenbanksysteme (in German), 4th Ed., Oldenbourg, 1997. Chapter 4: Relationale Anfragesprachen (Relational Query Languages).
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Date/Darwen: A Guide to the SQL Standard, Fourth Edition, Addison-Wesley, 1997.
- van der Lans: SQL, Der ISO-Standard (in German), Hanser, 1990.
- 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.
- Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft SQL Server Books Online: Accessing and Changing Data.
- 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.
- P.C. Zikopoulos, J. Gibbs, R.B. Melnyk: DB2 Fundamentals Certification for Dummies, 2001.
- R. Sanders: DB2 V8.1 Family Fundamentals Certification Prep, Part 6: Data Concurrency, IBM developerWorks, 2003.

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- die SQL-Kommandos `INSERT`, `UPDATE`, `DELETE`, sowie `COMMIT` und `ROLLBACK` verwenden.
- das Transaktions-Konzept erklären.

  Typisches Beispiel nennen, ACID-Eigenschaften erklären.
- erklären, was geschieht, wenn mehrere Benutzer gleichzeitig auf die Datenbank zugreifen.

  Problem-Typen aufzählen, zu jedem ein Beispiel machen. Sperren (ink. Deadlocks) und "Multi Version Concurrency Control" erklären.
- Mehrbenutzer-Sicherheit von Programmen bewerten.

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

Inhalt

Updates in SQL: Übersicht

- SQL-Befehle zur Änderung des DB-Zustands:
  - ◇ `INSERT`: Einfügung neuer Zeilen in eine Tabelle.
  - ◇ `DELETE`: Löschung von Zeilen aus einer Tabelle.
  - ◇ `UPDATE`: Änderung von Tabelleneinträgen (Werten in existierenden Zeilen).
- SQL-Befehle zur Beendigung von Transaktionen:
  - ◇ `COMMIT`: Erfolgreiches Ende der Transaktion, Änderungen des DB-Zustands werden dauerhaft.
  - ◇ `ROLLBACK`: Transaktion fehlgeschlagen, alle Änderungen rückgängig machen.

Beispiel-Datenbank

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	···
102	Michael	Grau	NULL
103	Daniel	Sommer	···
104	Iris	Winter	···

AUFGABEN			
ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

BEWERTUNGEN			
SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
102	H	2	9
102	Z	1	10
103	H	1	5
103	Z	1	7

INSERT: Übersicht

- Der `INSERT`-Befehl hat zwei Formen:
  - ◇ Einfügung einer einzelnen Zeile mit neuen Daten.
  - ◇ Einfügung des Ergebnisses einer Anfrage.
- Die zweite Form kann z.B. benutzt werden, um eine Tabelle zu kopieren.

  Die neue Tabelle (Kopie) muß allerdings vorher mit “`CREATE TABLE`” erstellt werden. Oracle erlaubt “`CREATE TABLE ... AS SELECT ...`”.
- In SQL-92 gibt es nur einen allgemeinen `INSERT`-Befehl: “`VALUES`” (erste Form) und “`SELECT`” (zweite Form) sind beides Tabellenausdrücke.

INSERT: Neue Werte (1)

- Beispiel:

```
INSERT INTO STUDENTEN
VALUES (105, 'Nina', 'Brass', NULL);
```

- Mögliche Werte für die `VALUES`-Klausel sind:
  - ◇ Terme, also insbesondere Konstanten, aber auch z.B. “`100+5`”, “`SYSDATE`”.

    Die Terme können natürlich keine Attributreferenzen enthalten, da hier keine Tupelvariablen deklariert sind.
  - ◇ Schlüsselworte “`NULL`”, “`DEFAULT`”.

    Nach dem SQL-Standard ist “`NULL`” kein Term, deswegen muß es hier getrennt aufgeführt werden. “`DEFAULT`” meint den in der “`CREATE TABLE`”-Anweisung deklarierten Defaultwert für die Spalte.

INSERT: Neue Werte (2)

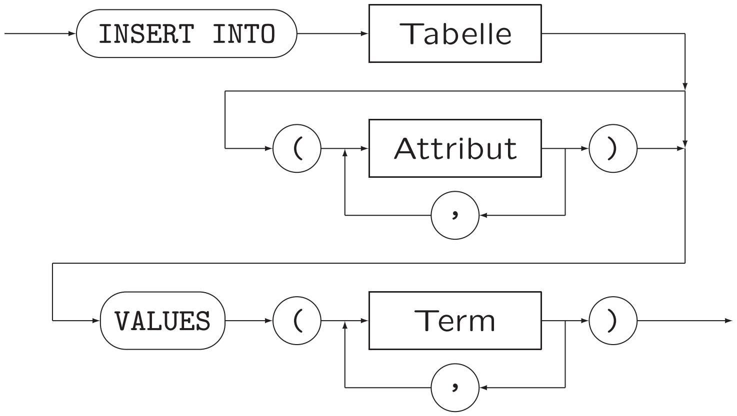
- Man kann Werte auch nur für eine Teilmenge der Spalten angeben:

```
INSERT INTO STUDENTEN(SID, VORNAME, NACHNAME)
VALUES (105, 'Nina', 'Brass')
```

- In die übrigen Spalten (hier “`EMAIL`”) wird der jeweilige Default-Wert eingetragen (hier Null).
- Die Spalten müssen nicht in der gleichen Reihenfolge wie in der Tabelle angegeben werden.

  Daher ist diese Syntax auch bequem, wenn man zwar für alle Spalten einen Wert hat, aber sich nicht an die Reihenfolge der Spalten in der Tabelle erinnert. In einem Programm sollte man diese Syntax wählen, um Probleme durch später hinzugefügte Spalten zu vermeiden.

INSERT: Neue Werte (3)



INSERT: Mit Anfrage (1)

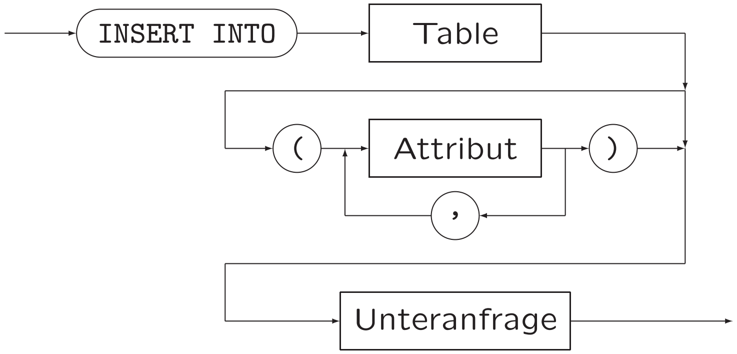
- Beispiel:

```
INSERT INTO KLAUSUREN(BEZ, ANR, THEMA, PROZENT)
SELECT 'DB-2005-E', A.ANO, A.THEMA,
       AVG(B.PUNKTE/A.MAXPT)*100
FROM   AUFGABEN A, BEWERTUNGEN B
WHERE  A.ATYP='E' AND B.ATYP='E' AND A.ANO=B.ANO
GROUP  BY A.ANO, A.THEMA
```

- Die Unteranfrage wird vollständig ausgewertet bevor die Ergebnistupel eingefügt werden.

  Daher gibt es auch dann ein definiertes Ergebnis (und niemals Endlosschleifen), wenn die Tabelle, in die eingefügt wird, in der Unteranfrage selbst verwendet wird.

INSERT: Mit Anfrage (2)



Beachte: Hier steht die Unteranfrage ausnahmsweise nicht in (...).

DELETE (1)

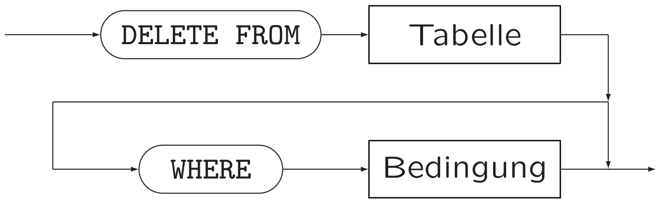
- Beispiel: Lösche alle Bewertungen für Lisa Weiss:

```
DELETE FROM BEWERTUNGEN
WHERE  SID IN (SELECT SID
               FROM   STUDENTEN
               WHERE  VORNAME  = 'Lisa'
               AND    NACHNAME = 'Weiss')
```

- Achtung: Wenn man die `WHERE`-Bedingung weglässt, werden alle Tupel gelöscht!

  Es ist eventuell möglich, "`ROLLBACK`" zu verwenden, wenn etwas schief gelaufen ist. Dafür muß man den Fehler aber bemerken, bevor die Transaktion beendet wird. Man sollte sich die Tabelle also nochmal anschauen. Manche SQL-Schnittstellen bestätigen jede Änderung sofort (`autocommit`), dann gibt es keine Möglichkeit mehr für ein Undo.

DELETE (2)



TRUNCATE (1)

- Oracle, SQL Server, und MySQL (aber nicht DB2 und Access) haben ein Kommando

  `TRUNCATE TABLE ⟨Tabellenname⟩`

  Dies löscht alle Zeilen aus der Tabelle und gibt den von der Tabelle belegten Speicherplatz frei.
- Es ist ähnlich zu einem `DROP TABLE`, aber die Tabellendefinition (Schemainformation) bleibt erhalten, nur die Tabellendaten werden gelöscht.

  Daher sind auch Verweise auf die Tabelle in Zugriffsrechten, Sichten, Triggern, gespeicherten Prozeduren, etc. nicht betroffen.

TRUNCATE (2)

- Im Gegensatz zu `DELETE`, kann `TRUNCATE` nicht mit `ROLLBACK` zurückgenommen werden.
- Dafür ist es viel schneller.

  Zumindest in Oracle würde `DELETE` auch nicht wirklich Speicherplatz frei geben, er bleibt für die Tabelle reserviert.
  Ein anderes Problem ist, daß wenn man alle Zeilen einer großen Tabelle mit `DELETE` löschen will, das Rollback Segment (Speicherplatz für Undo-Information in Oracle) eventuell zu klein ist. Dann liefert "`DELETE FROM ⟨Table Name⟩`" eine Fehlermeldung und nichts wird gelöscht. Bei `TRUNCATE` kann das nicht passieren.
- `TRUNCATE` gehört nicht zum SQL-92 Standard.

  Aber es tritt in der Oracle Zertifizierungs-Prüfung auf.

UPDATE (1)

- Das `UPDATE`-Kommando dient zur Änderung von Attributwerten ausgewählter Tupel.
- Z.B. soll ein Zusatzpunkt für alle Lösungen von Aufgabe 1 in der Zwischenklausur vergeben werden:

```
UPDATE BEWERTUNGEN
SET    PUNKTE = PUNKTE + 1
WHERE  ATYP = 'Z' AND ANO = 1
```

- Die rechte Seite der Zuweisung kann die alten Werte aller Attribute des aktuellen Tupels verwenden.

  Die `WHERE`-Bedingung und die Terme auf der rechten Seite der Zuweisung werden ausgewertet, bevor ein Update wirklich durchgeführt wird. Als neuer Attributwert ist auch `NULL` erlaubt.

UPDATE (2)

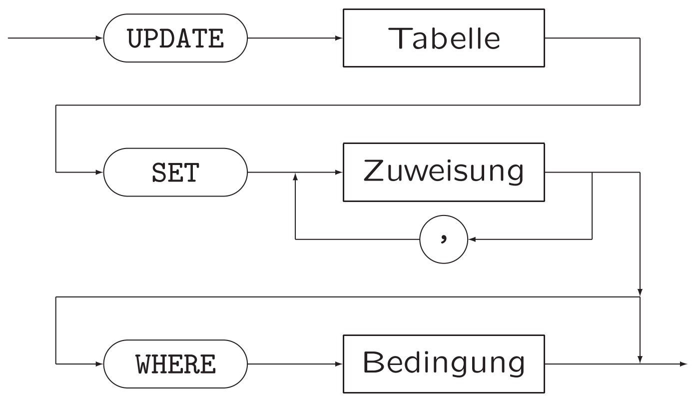
- In SQL-92, Oracle, DB2, SQL Server (aber nicht in SQL-86, MySQL, Access), kann der neue Attributwert mit einer Unteranfrage berechnet werden.

  Die Unteranfrage darf nicht mehr als eine Zeile liefern (mit einer Spalte). Falls sie keine Zeile liefert, wird ein Nullwert verwendet.

- Man kann in einer `UPDATE`-Anweisung auch mehrere Attribute ändern:

```
UPDATE AUFGABEN
SET    THEMA = 'Einfaches SQL',
       MAXPT = 8
WHERE  ATYP = 'H' AND ANO = 1
```

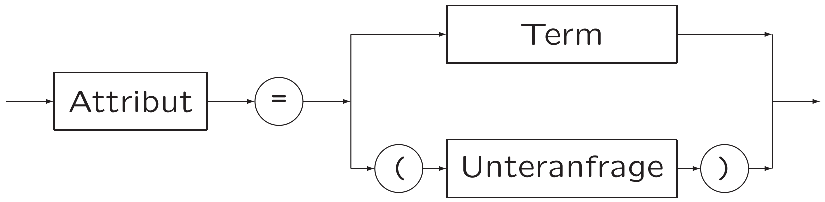

UPDATE (3)



- In MySQL werden Zuweisungen in der gegebenen Reihenfolge abgearbeitet: Neue Werte weiter links stehender Zuweisungen sind schon sichtbar.

UPDATE (4)

Zuweisung:



- SQL-86, MySQL, und Access erlauben keine Unteranfragen auf der rechten Seite. MySQL erlaubt mehrere Tabellen nach `UPDATE` (auch einen Join), im SQL-2003 Standard ist das nicht vorgesehen.
- In SQL-92, DB2 und SQL-Server kann eine Unteranfrage ohnehin als Term genutzt werden, daher ist der zweite Fall eigentlich ein Spezialfall des ersten. Nur für Oracle 8 muß die Unteranfrage explizit genannt werden.

MERGE (1)

- Im SQL-2003 Standard wurde eine neue Anweisung `MERGE` eingeführt, die `INSERT` und `UPDATE` kombiniert.

  Es gibt die `MERGE`-Anweisung z.B. in Oracle 10g, DB2 Ver. 9, SQL Server 2008. Jedes System hat andere Erweiterungen zum Standard.

- Sie wird meistens verwendet, wenn man eine Menge von Änderungen zu einer Tabelle in einer anderen Tabelle gesammelt hat.
- Damit kann man z.B. eine Tabelle mit den Daten in einer anderen Tabelle synchronisieren, wie es besonders im Data Warehouse Bereich benötigt wird.

MERGE (2)

- Beispiel: Ein Tutor trägt seine Bewertungen für Hausaufgaben in folgende Tabelle ein:

  `GRUPPE1(ANR, SID, PUNKTE)`

- Von Zeit zu Zeit müssen die Punkte in die Haupt-Tabelle `BEWERTUNGEN` übernommen werden.
- Es ist auch möglich, dass der Tutor in seiner Tabelle Punkte nachträglich verändert hat (nachdem die Punkte bereits einmal übernommen wurden).

  Wenn Sie eine Bewertung nicht verstehen, oder für falsch halten, fragen Sie nach. Auch Tutoren, Mitarbeiter und Professoren machen gelegentlich Fehler.

MERGE (3)

- Befehl zur Übernahme der Daten:

```
MERGE INTO BEWERTUNGEN B
      USING GRUPPE1 G
      ON (B.SID = G.SID AND
          B.ANR = G.ANR AND B.ATYP ='H')
      WHEN MATCHED THEN
          UPDATE SET B.PUNKTE = G.PUNKTE
      WHEN NOT MATCHED THEN
          INSERT(SID, ATYP, ANR, PUNKTE)
          VALUES(G.SID, 'H', G.ANR, G.PUNKTE)
```

Eine Zeile der Zieltabelle (`BEWERTUNGEN`) kann höchstens einen Join-Partner in der Quelltabelle (`GRUPPE1`) haben. Sonst wäre der Update nicht eindeutig definiert und das `MERGE` scheitert.

MERGE (4)

- Das entsprechende klassische `INSERT` wäre einfach, aber der `UPDATE`-Teil wäre umständlich:

```
UPDATE BEWERTUNGEN
SET    PUNKTE = (SELECT PUNKTE
                 FROM   GRUPPE1 G
                 WHERE  G.SID = BEWERTUNGEN.SID
                 AND    G.ANR = BEWERTUNGEN.ANR
                 AND    G.ATYP = 'H')
WHERE  EXISTS   (SELECT *
                 FROM   GRUPPE1 G
                 WHERE  G.SID = BEWERTUNGEN.SID
                 AND    G.ANR = BEWERTUNGEN.ANR
                 AND    G.ATYP = 'H')
```


Inhalt

Transaktionen (1)

- Transaktion: Folge von DB-Kommandos, insbesondere Updates, die das DBMS als Einheit behandelt.
- Z.B. besteht eine Überweisung von 50 Euro von Konto 11 auf Konto 23 aus folgenden Schritten:
  - ◇ Prüfung von Kontostand und Kreditrahmen von Konto 11,
  - ◇ Reduktion des Stands von Konto 11 um 50 Euro,
  - ◇ Erhöhung des Stands von Konto 23 um 50 Euro,
  - ◇ Schreiben von Einträgen in die Kontoauszüge beider Konten (plus ggf. Protokoll des Arbeitsplatzes).

Transaktionen (2)

ACID-Merkregel für Eigenschaften von Transaktionen:

- Atomarität ("Atomicity")

  Eine Transaktion wird "ganz oder garnicht" ausgeführt.

- Konsistenz ("Consistency")

  Eine Transaktion führt von einem konsistenten in einen konsistenten DB-Zustand.

- Isolation ("Isolation")

  Transaktionen paralleler Benutzer stören sich nicht gegenseitig.

- Dauerhaftigkeit ("Durability")

  Wenn eine Transaktion erfolgreich mit `COMMIT` abgeschlossen wurde, sind ihre Daten sicher gespeichert.

Transaktionen (3)

Atomarität:

- Moderne DBMS garantieren, daß eine Transaktion
  - ◇ entweder vollständig ausgeführt wird,
  - ◇ oder keinerlei Spuren hinterlässt

  ("alles oder nichts"-Prinzip).
- Kann eine Transaktion nicht zu Ende ausgeführt werden (z.B. wegen Stromausfall), so wird der Zustand vor Beginn der Transaktion beim nächsten Hochfahren des Systems wieder hergestellt.

Transaktionen (4)

- Atomarität gibt eine Undo-Möglichkeit:
  - ◇ Solange die Transaktion nicht als vollständig deklariert wurde (mit `COMMIT`), können alle Änderungen zurückgenommen werden (mit `ROLLBACK`).
  - ◇ In den meisten DBMS kann man aber nur die ganze Transaktion zurücknehmen (nicht nur das letzte Kommando).

    Allerdings kann man z.B. in Oracle und SQL Server "savepoints" inerhalb einer Transaktion setzen, und bei Bedarf auf den so benannten Zustand zurücksetzen.
  - ◇ Nach dem `COMMIT` ist kein Undo mehr möglich.

Transaktionen (5)

Dauerhaftigkeit:

- Wenn das DBMS das erfolgreiche Ende einer Tranaktion bestätigt, sind die Änderungen dauerhaft.
- Die Daten sind dann auf einer Platte gespeichert — sie sind nicht verloren, selbst wenn eine Sekunde später der Strom ausfällt.

  Bei Betriebssystemen weiß man dagegen oft nicht genau ob die Daten schon auf der Platte oder noch in einem Puffer sind.
- Mächtige Backup&Recovery-Mechanismen: selbst wenn eine Platte ausfällt, sind keine Daten verloren.

  Auf Betriebssystem-Ebene dagegen nur ein Backup pro Tag normal.

Transaktionen (6)

- Atomarität und Dauerhaftigkeit zusammen bedeuten, daß es einen Zeitpunkt gibt, an dem alle Änderungen schlagartig dauerhaft werden.

  Stürzt das System vor diesem Zeitpunkt ab, erhält man den alten DB-Zustand (vor der Transaktion). Stürzt es danach ab, erhält man den neuen Zustand (mit allen Änderungen der Transaktion).
- Der Zeitpunkt liegt zwischen
  - ◇ dem Abschicken des `COMMIT`-Kommandos an das DBMS (Benutzer: "Transaktion fertig")
  - ◇ und der Mitteilung des Systems, daß das `COMMIT` erfolgreich ausgeführt wurde.

Transaktionen (7)

Isolation:

- Alle größeren DBMS erlauben gleichzeitige Zugriffe mehrerer Benutzer.
- Ohne Kontrolle könnte dies zum Verlust von Daten führen, und zur Zerstörung der Konsistenz der DB.
- Das DBMS versucht aber die Transaktionen von einander zu isolieren: Jeder Benutzer soll den Eindruck haben, daß seine Transaktion exklusiven Zugriff auf die ganze Datenbank hat.

  Die meisten DBMS verwalten dazu automatisch (intern) Sperren auf DB-Objekten (z.B. Tabellen, Tupeln): s.u.

Transaktionen (8)

Konsistenz:

- Benutzer und System können sicher sein, daß der aktuelle Zustand das Ergebnis einer Folge von vollständig ausgeführten Transaktionen ist.
- Der Benutzer muß sicherstellen, daß jede Transaktion, wenn sie vollständig und isoliert (einzeln) auf einen konsistenten Zustand angewendet wird, auch wieder einen konsistenten Zustand produziert.

  Ein Zustand heißt konsistent, wenn er alle Integritätsbedingungen erfüllt. Moderne DBMS bieten Unterstützung dafür an: Schlüssel, Fremdschlüssel, `NOT NULL` und `CHECK`-Bedingungen können deklarativ spezifiziert werden. Für komplexere Bedingungen gibt es Trigger.

Transaktionen (9)

- Die Konsistenz ist zum Teil eine Folge der anderen drei Eigenschaften und zum Teil etwas, was der Benutzer garantieren muß.
- Konsistenz ist besonders auch für komplexe/redundante Datenstrukturen wichtig.

  Wenn Benutzer redundante Daten speichern, müssen sie diese Daten in derselben Transaktion aktualisieren, die auch die Originaldaten modifiziert. Dann stellt aber das System sicher, daß selbst bei einem Stromausfall zwischen den Befehlen die beiden Kopien niemals auseinander laufen. Dies betrifft auch die internen Datenstrukturen des DBMS, z.B. Indexe (redundante Datenstrukturen, um Zeilen mit gegebenen Attributwerten schnell zu finden). Würden manche Zeilen im Index fehlen, wäre das Systemverhalten unvorhersehbar.

Transaktions-Verwaltung (1)

- SQL hat kein Kommando, um den Beginn einer Transaktion zu markieren.

  Eine Transaktion beginnt automatisch, wenn man sich beim DBMS anmeldet, und jedes Mal, nachdem eine Transaktion beendet wurde.
- Eine Transaktion wird beendet mit
  - `COMMIT [WORK]`: Macht Änderungen dauerhaft.
  - `ROLLBACK [WORK]`: Nimmt Änderungen zurück.
- Manche Kommandos, wie etwa `DROP TABLE`, lösen zumindest in Oracle automatisch ein `COMMIT` aus.

  Solche Kommandos können daher nicht zurückgenommen werden. Dies betrifft auch vorangegangene, noch nicht bestätigte Updates.

Transaktions-Verwaltung (2)

- Manche Systeme haben einen "Autocommit Modus": Dann wird ein `COMMIT` automatisch nach jedem Update durchgeführt (dann gibt es keine Undo-Möglichkeit mehr!).

  In Oracle SQL*Plus kann man diesen Modus mit "`set autocommit on`" auswählen (defaultmäßig ist der Autocommit Modus ausgeschaltet). SQL Server läuft normalerweise im Autocommit Modus, aber das Kommando "`BEGIN TRANSACTION`" schaltet diesen Modus aus. In DB2 funktionieren `COMMIT` und `ROLLBACK` normal. MySQL hat nur den Autocommit Modus, außer wenn man einen speziellen Tabellentyp verwendet, der Transaktionen unterstützt. Access bestätigt ebenfalls alle Änderungen automatisch und versteht die Kommandos `COMMIT` und `ROLLBACK` nicht.

Transaktions-Verwaltung (3)

- Wenn man SQL*Plus (SQL-Interpreter von Oracle) normal verlässt (mit `QUIT` oder `EXIT`), findet automatisch ein `COMMIT` statt.
- Wenn man dagegen einfach das Fenster schließt, oder sich beim Betriebssystem abmeldet, findet ein `ROLLBACK` statt.

  Obwohl man die geänderten Daten vorher gesehen hat, sind sie bei der nächsten Sitzung verschwunden.
- Es ist also besser, explizit `COMMIT` einzugeben.

  Das gilt auch für SQL-Skripte.

Transaktions-Verwaltung (4)

- Wenn man mit der Datenbank für eine längere Zeit arbeitet, sollte man die Änderungen von Zeit zu Zeit mit `COMMIT` bestätigen.

  Falls es zu einem Stromausfall etc. kommen sollte, sind so nur die Änderungen nach dem letzten `COMMIT` verloren. Außerdem sperrt das DBMS typischerweise von der Transaktion veränderte Zeilen, eventuell auch ganze Blöcke auf der Platte. Diese Sperren bleiben bis zum Ende der Transaktion erhalten. Lange Transaktionen können dann andere Benutzer behindern. Schließlich muß das DBMS für die Dauer der Transaktion Undo-Information aufbewahren. Wenn es die Speicherbereiche zyklisch neu verwendet, kann das auch zu Problemen führen. Klassische Datenbanksysteme sind nicht für lange Transaktionen gedacht.

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.

Probleme (1)

- Die beiden Ziele stehen im Konflikt mit einander: 100% Isolation bedeutet sehr wenig Parallelität — häufig müssen ganze Tabellen gesperrt werden.
- SQL hat kein "Begin Transaction" Kommando. 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 (durch Nachdenken/Planung) auszuschließen.

  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)`.

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 hat für jede Sperre eine Liste aller wartenden Transaktionen. 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`
⟶ 1 row updated.	

	`UPDATE KONTO`
	`SET STAND = STAND + 20`
	`WHERE NR = 1001`
	⟶ (keine Reaktion)

`COMMIT`	
	⟶ 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).

Dirty Read Problem (1)

- Transaktion A setzt den Kontostand auf 1000000, und erkennt dann den Fehler. B berechnet Zinsen.

Transaktion A	Transaktion B
UPDATE KONTO SET STAND = 1000000 WHERE NR = 1001	
	SELECT STAND FROM KONTO WHERE NR = 1001 $\longrightarrow$ 1000000
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 12-55 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):

- Für Lesezugriffe stellt Oracle alte Versionen der Daten wieder her, die dem Zustand nach der letzten mit `COMMIT` bestätigten Transaktion entsprechen.
- D.h. für diesen Lesezugriff werden alle noch unbestätigten Änderungen wieder zurückgenommen.

  Genauer werden auch mit `COMMIT` bestätigte Änderungen zurückgenommen, wenn das `COMMIT` nach dem Beginn der Ausführung dieser Anfrage lag. So garantiert Oracle einen konsistenten Zustand 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 ⟶ 50
UPDATE KONTO SET STAND = STAND + 30 WHERE NR = 1001 SELECT STAND ... ⟶ 80	
	SELECT STAND ... ⟶ 50
COMMIT	
	SELECT STAND ... ⟶ 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 System 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.

  ⇒ **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`
⟶ 100	

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

`SELECT STAND`
`FROM KONTO`
`WHERE NR = 1001`
⟶ 150	

Nonrepeatable Read (2)

- In vielen DBMS (z.B. Oracle) 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 12-82).

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`
⟶ 1000	

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

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

`SELECT BETRAG`
`FROM GELDBESTAND`
⟶ 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) garantiert ist, daß eine Anfrage immer bezüglich nur 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 12-81).

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` ⟶ 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. nur Kontos mit `KREDITRAHMEN >= 1000` bekommen die Bonus-Zahlung), 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.

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 in Oracle 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`".

"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` ⟶ `old`	
`UPDATE S SET A='new'`	
	`SELECT A FROM S` ⟶ `old`
	`UPDATE R SET A='new'`
	`COMMIT`
`COMMIT`	

Inhalt

Isolationsstufen (1)

- Bei IBM DB2 (Ver. 8) gibt es vier Isolationsstufen:
  - Repeatable Read (`RR`)

    Höchste Isolationsstufe, vermeidet auch das Phantom Problem, wenig Parallelität.
  - Read Stability (`RS`)

    Vermeidet das Nonrepeatable Read Problem, alle Lost Updates.
  - Cursor Stability (`CS`): Default-Wert

    Vermeidet das Dirty Read Problem sowie Lost Updates bei Updates aus einem Befehl bzw. für das aktuelle Tupel eines Cursors.
  - Uncommitted Read (`UR`)

    Erlaubt das Lesen noch nicht mit `COMMIT` bestätigter Daten. Bei `UPDATE`-Befehlen verhält es sich aber wie `CS`.

Isolationsstufen (2)

Isolationsstufe	Dirty Read	Nonrepeatable Read	Phantom Problem
RR	—	—	—
RS	—	—	möglich
CS	—	möglich	möglich
UR	möglich	möglich	möglich

Nach der IBM-Dokumentation sind Lost Updates bei jeder Isolationsstufe ausgeschlossen. Das Beispiel von Folie 12-65 (Lost Update bei Verwendung eines vorher mit `SELECT` gelesenen Wertes) läuft auch bei DB2 mit Isolationsstufe CS durch. Wie oben erläutert, ist dies aber eher ein Fall von Nonrepeatable Read, der dann zu einem Lost Update führt.

Isolationsstufen (3)

- Transaktionen heißen bei IBM auch "Unit of Work" (UOW).
- Wenn man parallele Transaktionen ausprobieren will (z.B. mit zwei "Command Editor" Fenstern), muß man zuerst den Autocommit Modus ausschalten.

  Das geht unter "Tools → Tools Settings → Command Editor". Danach muß man mit "Tool Settings → Exit" die neuen Setzungen abspeichern.
  Wenn man mit dem klassischen "Command Window" (CLP) arbeitet, muß man die Option "`+c`" bei jedem Kommando verwenden, z.B. "`db2 +c insert into ...`".

Isolationsstufen (4)

- Eine Isolationsstufe wird meistens für ein ganzes Anwendungsprogramm beim Aufruf des Precompilers festgelegt.

  Z.B. "db2 prep example.sqc isolation RS". Hier ist "example.sqc" ein C-Programm mit Embedded SQL. Der Aufruf von PRECOMPILE (PREP) übersetzt es in ein reines C-Programm mit Funktionsaufrufen, sowie ein "Package" in der Datenbank, das Auswertungspläne für die SQL-Anweisungen in "example.sqc" enthält. Man kann auch zuerst nur ein "Bindfile" erzeugen, und in einem zweiten Schritt ein Package erzeugen: "db2 prep example.sqc bindfile using example.bnd isolation RS" und dann "db2 bind example.bnd". Mit dieser Methode kann man den "BIND"-Befehl später mit neuen Optionen wiederholen, z.B. um eine neue Isolationsstufe zu setzen: "db2 bind example.bnd isolation RR" (überschreibt Angaben im PREP). Im Data Dictionary kann man in SYSCAT.PACKAGES, Spalte ISOLATION, die Isolationsstufe einsehen.

Isolationsstufen (5)

- Im "Command Editor" und "Command Window" kann man eine Isolationsstufe mit

  change isolation level to ⟨Isolationsstufe⟩

  wählen, allerdings nur vor dem "connect".

  Man kann eine Sitzung mit "terminate" beenden und dann eine neue Sitzung z.B. mit "connect to sample" beginnen.

- Neuerdings kann man Isolationsstufen auch in jeder Anweisung festlegen:

  SELECT * FROM KONTO WITH RR

  Die Isolationsstufe UR (Uncommitted Read) kann man nur für Leseanweisungen wählen. Andernfalls ersetzt sie das DBMS durch CS.

Isolationsstufen (6)

- Für JDBC/SQLJ gibt es in java.sql die Methode setTransactionIsolation().

  Für CLI/ODBC kann man SQLSetConnectAttr-Funktion mit dem Attribut SQL_ATTR_TXN_ISOLATION aufrufen. Außerdem kann man die Isolationsstufe in db2cli.ini setzen, Parameter TXNISOLATION. Diese Einstellung beeinflusst auch JDBC/SQLJ.

- Damit die Isolationsstufe UR (Uncommitted Read) für Cursor wirksam wird, muß es klar sein, daß der Cursor nicht updatebar ist.

  Man kann dazu der SELECT-Anweisung FOR READ ONLY hinzufügen. Alternativ kann man bei PREP/BIND die Option BLOCKING ALL angeben: Damit sichert man zu, daß es keine positionierten UPDATE/DELETE über dynamisches SQL gibt, so daß die Updatebarkeit der Cursor klar wird.

Sperren (1)

- Sperren haben drei wichtige Attribute:
  - ◇ Modus: Legt die erlaubten Zugriffe des Besitzers der Sperre und der anderen Benutzer fest.
  - ◇ Objekt: Eine Tabelle, eine Zeile, etc.
  - ◇ Dauer: Sperren werden spätestens beim Commit bzw. Rollback freigegeben, manche früher.
- Man kann nur Tabellen explizit sperren:

  LOCK TABLE ⟨Tabelle⟩ IN EXCLUSIVE MODE

  Man kann auch "IN SHARE MODE" sperren.

Sperren (2)

Modus		Table	Row	Read	Write
`IN`	Intent None	×			
`IS`	Intent Share	×			
`IX`	Intent Exclusive	×			
`SIX`	Share with Int. Excl.	×		×	
`S`	Share	×	×	×	
`U`	Update	×	×	×	
`X`	Exclusive	×	×	×	×
`W`	Weak Exclusive		×	×	×
`Z`	Super Exclusive	×		×	×
`NS`	Next Key Share		×	×	
`NW`	Next Key Weak Excl.		×	×	

Sperren (3)

- Die Intent-Sperren (für Absichtserklärungen) sind nötig, weil es Sperren auf unterschiedlichen Ebenen (Tabellen, Zeilen) gibt.
- Man muß z.B. vermeiden, daß Nutzer A für einige Zeilen der Tabelle `X`-Sperren bekommt, und dann Nutzer B für die ganze Tabelle eine `S`-Sperre.
- Daher sind Sperren auf Zeilen immer mit Intent-Sperren auf der Tabelle gekoppelt: Nutzer A muß zuerst eine `IX`-Sperre auf der Tabelle haben, bevor er `X`-Sperren auf Tupeln bekommen kann.

Sperren (4)

- Wenn der Benutzer B eine `S`-Sperre für die Tabelle anfordert, nachdem A schon eine `IX`-Sperre hat, muß B warten.
- Intent-Sperren sind untereinander kompatibel, Nutzer C könnte auch eine `IX` oder `IS` Sperre bekommen, obwohl A schon eine `IX`-Sperre hat.

  Die Intent-Sperren sind ja nur ein Hinweis darauf, daß echte Sperren auf Tupel-Ebene existieren können.
- Intent-Sperren alleine geben noch keine Lese- oder Schreibrechte. Sie werden im Zusammenhang mit richtigen Sperren auf Tupelebene gebraucht.

Sperren (5)

- "Intent None" (`IN`) wird bei der Isolationsstufe "Uncommitted Read" benutzt, um anzuzeigen, daß ein Prozess die Tabelle ohne Sperren liest.
- Man muß für diese Zeit aber z.B. ein "`ALTER TABLE`" oder "`DROP TABLE`" ausschließen.
- Diese Kommandos würden eine `Z`-Sperre ("Super Exclusive") anfordern, die mit `IN` ("Intent None") nicht kompatibel ist: Sie müßten also warten.

Sperren (6)

- Jeder Prozess kann auf ein Objekt gleichzeitig nur eine Sperre haben.
- Fordert er eine zweite Sperre auf ein Objekt an, auf dem er schon eine Sperre hat, wird der Sperrmodus auf den stärkeren von beiden gesetzt.

  Diesen Vorgang der Änderung eines Sperrmodus nennt man "Lock Conversion".
- Der einzige Fall, bei dem nicht einer von beiden Sperrmodi stärker als der andere ist, sind `S` und `IX`.
- Daher wurde der Modus `SIX` (= `S` + `IX`) eingeführt.

Sperren (7)

- Wenn ein Update ausgeführt werden soll, werden zunächst einige Zeilen gelesen, und dann eine Teilmenge davon geändert.
- Würde man beim Lesen eine `S`-Sperre verwenden, und dann eine "Lock Conversion" auf eine `X`-Sperre machen, kann das leicht zu Deadlocks führen.
- Daher werden in der Lese-Phase `U`-Sperren verwendet: Diese sind kompatibel zu `S`-Sperren (zu diesem Zeitpunkt ist ja noch nichts verändert), aber nicht zu anderen `U`-Sperren (auch nicht zu `X` etc.).

Sperren (8)

- Zur Vermeidung des Phantom-Problems wird beim Einfügen eines Tupels in einen Index das nächste Tupel im Modus `NW` ("Next Key Weak Exclusive") gesperrt.

  Tatsächlich ist es noch etwas komplizierter. Man muß hier auch zwischen Typ-1 und Typ-2 Indexen unterscheiden. Es wird versucht, diese Sperren einzusparen.
- Ein Index Scan im Modus `RR` würde alle gelesenen Tupel, inklusive des ersten Tupels, das die Bedingung nicht mehr erfüllt, im Modus `S` sperren.
- Die Sperrmodi `NW` und `S` sind nicht kompatibel.

Sperren (9)

- Die Einfügung müsste also warten, wenn sie einen Bereich betrifft, für den "Repeatable Read" garantiert werden soll.

  Wie oben schon erläutert, ist das Problem, daß ein nicht existierendes Tupel nicht gesperrt werden kann, daher wäre normalerweise eine Einfügung immer möglich. Der Trick ist nun, das nächste Tupel zu sperren (bzw. das Ende des Indexes, falls es das letzte Tupel war).
- Das eingefügte Tupel selbst wird mit einer `W`-Sperre versehen ("Weak Exclusive").

  Der einzige Unterschied zu einer `X`-Sperre ist, daß `W` kompatibel mit `NW` ist (jemand anders könnte also davor noch ein Tupel einfügen).

Sperren (10)

- Nur `RR`-Transaktionen müssen die Einfügung von Tupeln in dem von ihnen gelesenen Bereich verhindern.
- Daher wird in niedrigeren Isolationsstufen (`CS`, `RS`) die Sperre `NS` ("Next Key Share") anstelle von `S` verwendet: Sie ist kompatibel mit `NW`, ansonsten verhält sie sich wie `S`.

  Der Name ist etwas unglücklich: Es wird garnicht auf das nächste Tupel gesetzt, sondern es wurde eingeführt, um den Effekt der `NW`-Sperre abzumildern, die auf das nächste Tupel nach einem eingefügten Tupel gesetzt wird.

Sperren (11)

Angef.	Existierende Sperre											
Sperre	—	IN	IS	IX	SIX	S	U	X	W	Z	NS	NW
IN	+	+	+	+	+	+	+	+	+	–	+	+
IS	+	+	+	+	+	+	+	–	–	–	+	–
IX	+	+	+	+	–	–	–	–	–	–	–	–
SIX	+	+	+	–	–	–	–	–	–	–	–	–
S	+	+	+	–	–	–	+	–	–	–	+	–
U	+	+	+	–	–	–	–	–	–	–	+	–
X	+	+	–	–	–	–	–	–	–	–	–	–
W	+	+	–	–	–	–	–	–	–	–	–	+
Z	+	–	–	–	–	–	–	–	–	–	–	–
NS	+	+	+	–	–	+	+	–	–	–	+	+
NW	+	–	–	–	–	–	–	–	+	–	+	–

Sperren (12)

- Obige Tabelle (die sich so ähnlich in der offiziellen IBM-Dokumentation findet) zeigt Kombinationen, die nicht vorkommen können.
- Man müßte eigentlich eine Kompatibilitätsmatrix für Sperren auf Tabellenebene und eine für Sperren auf Tupelebene aufstellen (Übungsaufgabe).

SQL-Befehle und Sperren (1)

SELECT (Read-Only):

- `RR` (Repeatable Read, höchste Isolationsstufe):
  - ◇ Falls ein Table Scan zur Auswertung benutzt wird: `S`-Sperre auf der Tabelle.

    Dies gilt auch, wenn ein vollständiger Index-Scan genutzt wird.
  - ◇ Bei Zugriff über einen Index: `IS`-Sperre auf der Tabelle und `S`-Sperren auf den über den Index gelesenen Tupeln (plus das nächste Tupel).

    Es werden dabei alle über den Index zugegriffenen Tupel gesperrt, auch solche, die eventuelle weitere Bedingungen in der `WHERE`-Klausel nicht erfüllen. Die Sperren werden bis zum Transaktionsende gehalten.

SQL-Befehle und Sperren (2)

- RS benutzt eine IS-Sperre auf der Tabelle und NS-Sperren auf den gelesenen Tupeln.
  - ◇ Falls ein Tupel die WHERE-Bedingung nicht erfüllt, wird die Sperre gleich wieder freigegeben.
  - ◇ Ansonsten (das Tupel ist Teil des Ergebnisses) wird die Sperre bis Transaktionsende gehalten.
- CS funktioniert ähnlich (IS/NS-Sperren), aber gibt die Sperren immer gleich wieder frei.
- UR verwendet nur eine IN-Sperre auf der Tabelle.

SQL-Befehle und Sperren (3)

UPDATE:

- In diesem Fall werden normalerweise
  - ◇ eine IX-Sperre auf die Tabelle, und
  - ◇ X-Sperren auf die geänderten Tupel

  gesetzt.
- Ausnahmen gibt es nur für die Isolationsstufe RR.

  Hier wird eine X-Sperre auf die ganze Tabelle gesetzt, wenn klar ist, daß ohnehin alle Tupel betroffen sind (UPDATE ohne WHERE).

  Wenn die Bedingung mit einem "Full Table Scan" ausgewertet wird. Es wird eine SIX-Sperre auf die Tabelle gesetzt,

SQL-Befehle und Sperren (4)

Beispiel (Isolationsstufe CS):

- Transaktion A ändert den Stand von Konto 1001 von 100 auf 200 (noch kein COMMIT).
- Transaktion B führt folgende Anfrage aus:

  ```
  SELECT * FROM KONTO WHERE STAND > 500
  ```
- Falls die Anfrage mit "Full Table Scan" ausgeführt wird, trifft sie bei der Ausführung auf das geänderte Tupel und muß warten.
- Wird die Anfrage dagegen mit einem Index ausgeführt, muß sie nicht warten.

SQL-Befehle und Sperren (5)

Bemerkung:

- DB2 bietet (offenbar seit Version 8) eine Option "Evaluate Uncommitted" (für Stufen CS und RS).
- Die WHERE-Bedingung der Anfrage wird dann für die ohne Sperre gelesenen Tupel getestet, und nur Tupel gesperrt, die die Bedingung erfüllen.
- Das gibt natürlich mehr Parallelität.
- Dann werden aber auch Tupel übergangen, die aufgrund eines (später mit ROLLBACK zurückgenommenen) Updates die Bedingung temporär verletzen.

Vergröberung von Sperren (1)

- Wenn es für eine Tabelle viele Sperren auf Tupelebene gibt, kann sich das DBMS entscheiden, auf die Tabellenebene zu wechseln ("Lock Escalation").
- Das kann zu weniger Parallelität und möglicherweise zu Deadlocks führen.
- Aber der Overhead für die Verwaltung der Sperren sinkt.

  Soweit ich weiss, hat die Sperrentabelle eine feste Größe, irgendwann wäre also Schluß.

Vergröberung von Sperren (2)

- Mit folgendem Befehl kann man festlegen, daß für eine bestimmte Tabelle immer Sperren auf Tabellenebene gewählt werden sollen.

  ALTER TABLE ⟨Tabelle⟩ LOCKSIZE TABLE

- Das wäre z.B. für Tabellen nützlich, auf die es (praktisch) nur Lese-Zugriffe gibt.

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 $1 \le i \le n$ und $1 \le j \le m_i$ (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 $s < s' < s''$ gilt: Wenn s und s'' zur gleichen Transaktion T_i gehören, dann muß s' 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 12-118 (mit Lost Update):

$$T_1 \underset{\mathtt{A}}{\overset{\mathtt{A}}{\rightleftarrows}} T_2$$

- 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`	