14 Concurrency

- Synchronisation paralleler Transaktionen

14.1 Problemstellung

- In einem Multi-User-Betrieb (Multi-Tasking-Betrieb) müssen parallele Zugriffe verschiedener Anwender (Tasks) auf Daten synchronisiert werden (Concurrency-Control)

- Lost Update Problem (Problem der verlorenen Änderung)

Transaktion A	Zeit	Transaktion B
-	|	-
FETCH R	t1	-
-	|	-
-	t2	FETCH R
-	|	-
UPDATE R	t3	-
-	|	-
-	t4	UPDATE R
	↓	

Änderung der Transaktion A (t3) geht verloren (t4).

- Inconsistent Analysis Problem (Incorrect Summary Problem, Problem der inkonsistenten Sicht)

3 Zeilen mit Kontoständen: R1.stand=40, R2.stand=50, R3.stand=30
Transaktion A: Summiert die Kontostände auf
Transaktion B: Bucht 10 Einheiten von R3.stand (-10) auf R1.stand (+10) um

Transaktion A	Zeit	Transaktion B
-	|	-
FETCH R1	t1	-
sum=sum+stand (sum: 40)	|	-
-	|	-
FETCH R2	t2	-
sum=sum+stand (sum: 90)	|	-
-	|	-
-	t3	FETCH R3
-	|	stand=stand-10
-	t4	UPDATE R3 (stand: 20)
-	|	-
-	t5	FETCH R1
-	|	stand=stand+10
-	t6	UPDATE R1 (stand: 50)
-	|	-
-	t7	COMMIT
-	|	-
FETCH R3	t8	-
sum=sum+stand (sum: 110!)	|	-
	↓	

Transaktion A ermittelt eine falsche Summe (t8), richtig wäre 120.

Hinweis: Die folgende Situation stellt kein Inconsistent Analysis Problem dar: Transaktion B liest zwar den alten Wert, der stammt aber von einem konsistenten Datenzustand.

Transaktion A	Zeit	Transaktion B
-	|	-
FETCH R	t1	-
stand=stand+10	|	-
-	|	-
-	t2	FETCH R
-	|	-
UPDATE R	t3	-
	↓	

- Uncommited Dependency Problem (Temporary Update Problem, Problem der temporären Änderung)

Transaktion A	Zeit	Transaktion B
-	|	-
-	t1	UPDATE R
-	|	-
FETCH R	t2	-
-	|	-
-	t3	ROLLBACK
	↓	

Transaktion A liest Daten (t2), die logisch nie existiert haben (t3).
Einen solchen Lesevorgang bezeichnet man als Dirty Read.

Transaktion A	Zeit	Transaktion B
-	|	-
-	t1	UPDATE R
-	|	-
UPDATE R	t2	-
-	|	-
-	t3	ROLLBACK
	↓	

Zweifaches Problem:
- Transaktion A ändert Daten (t2), ausgehend von einem nicht existierenden Stand
- Änderungen werden wieder zurückgenommen (t3). Variante des Lost Update Problems

14.2 Serialisierbarkeit

- Serialisierbarkeit von Transaktionen: Der Zustand der Datenbank nach einem konkurrierenden Ablauf von Transaktionen und die Ergebnisse von Lesezugriffen während eines konkurrierenden Ablaufes von Transaktionen müssen gleich sein wie nach/bei einem beliebigen seriellen (sequentiellen) Ablauf der Transaktionen (umfangreiches theoretisches Gebiet).

- Schedule (History, Ausführung, Ausführungsplan)
  Ein konkreter (auch zeitlich verzahnter) Ablauf von Datenbank-Abfragen und -Änderungen mehrerer Transaktionen
  Transaktionen: T1, T2, T3, …, Tn
  Operationen: read, write, commit, abort
  Daten: A, B, C, …
  Schedule: S1 = r1(A); r2(A); w1(A); w2(A)
  S2 = r1(A); w1(A); r2(A); w2(A)
  …
  Die Anzahl der Operationen der Transaktion Ti sein Ni,
  dann kann die Anzahl der Schedules berechnet werden als:

  (N1 + N2 + … + Nn)! / (N1! * N2! * … * Nn!)

  Übung: Wieviele Schedules gibt es beim Lost Update Problem und beim Inconsistent Analysis Problem?

- Serieller (Serial) Schedule
  Schedule, bei dem die Transaktionen hintereinander ausgeführt werden (Reihenfolge der Transaktionen beliebig)

  Übung: Wieviele serielle Schedules gibt es beim Lost Update Problem?
  Übung: Wieviele serielle Schedules gibt es bei n Transaktionen?

- Serialisierbarer (serializable, konfliktserialisierbarer) Schedule
  Wenn er äquivalent zu irgend einem Seriellen Schedule ist

- Zwei äquivalente (equivalent, konfliktäquivalente) Schedules
  führen zwei in Konflikt stehende Operationen jeweils in derselben Reihenfolge aus

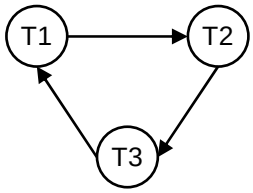
- Konfliktoperationen
  - gehören zu unterschiedlichen Transaktionen
  - greifen auf dasselbe Datenobjekt zu
  - mindestens eine Operation ist eine Schreiboperation

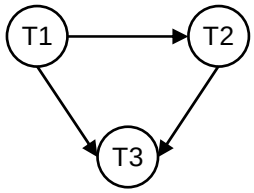
- Algorithmus zur Ermittlung der Serialisierbarkeit eines Schedules mittels
  Präzedenzgraph (Serialisierbarkeitsgraph, Konfliktgraph, Precedence Graph)
  - Knoten: alle Transaktionen
  - Kante: von Ti nach Tj, wenn es zwei Konfliktoperationen OPi $\in$ Ti und OPj $\in$ Tj gibt, wobei OPi zeitlich vor OPj liegt (müssen nicht unmittelbar hintereinander sein)
  - Wenn der Graph azyklisch ist (keine Kreise enthält), dann ist der Schedule serialisierbar (eine Angabe des / der äquivalenten seriellen Schedule ist möglich: Ti vor Tj, wenn Kante von Ti zu Tj führt – topologische Sortierung)

- Beispiel:

  T1: r(A); w(B) T2: w(A) T3: r(A); r(B)

  S1: r1(A); w2(A); r3(A); r3(B); w1(B) S2: r1(A); w2(A); r3(A); w1(B); r3(B)





  S1 ist nicht konfliktserialisierbar

  S2 ist konfliktserialisierbar
  Äquivalenter serieller Schedule: T1, T2, T3

  Bemerkung: S1 unterscheidet sich von S2 nur durch die Reihenfolge der beiden letzten Operationen

- Übung 1):

	T1	T2	T3
1	r(A)	-	-
2	-	w(A)	-
3	w(A)	-	-
4	-	-	w(A)

- Übung 2):

	T1	T2	T3
1	r(A)	-	-
2	-	w(B)	-
3	-	w(C)	-
4	w(B)	-	-
5	-	-	r(C)
6	w(A)	-	-
7	-	-	w(C)

- Übung 3): wie 2), zusätzlich

	T1	T2	T3	T4
0	-	-	-	w(A)

- Übung 4): wie 2), zusätzlich

	T1	T2	T3	T4
8	-	-	-	w(A)

- Vorgangsweise: Einzelne Logs für die beteiligten Objekte aufstellen

- Wenn der Schedule serialisierbar ist, geben sie alle äquivalenten Seriellen Schedules an

- Übung 5): Geben Sie den Präzedenzgraphen für das Lost Update Problem und das Inconsistent Analysis Problem an.

- Übung 6): Gegeben ist folgender Schedule:

  r1(D); r2(B); r3(A), w2(B); w2(A); w3(D); r4(B); r1(C); r4(C); r3(C); w4(D)

  Erstellen Sie den Präzedenzgraph und beurteilen Sie, ob der Schedule serialisierbar ist. Wenn Serialisierbarkeit vorliegt, dann geben Sie alle äquivalenten seriellen Schedules an.

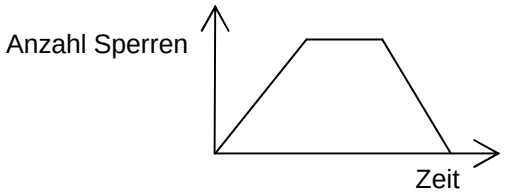
14.3 Lösungsmöglichkeiten

- Methoden zur Lösung von Concurrency-Problemen:
  - Sperr-Verfahren (Locking Techniques, Pessimistische Sperr-Verfahren, Pessimistic Concurrency Control)
    Annahme, dass Konflikte aufteten. Daher werden die Objekte von den Transaktionen gesperrt, um damit den Zugriff anderer Transaktionen auf diese Objekte für eine bestimmte (möglichst kurze) Zeit einzuschränken oder zu verhindern.
  - Optimistische Sperr-Verfahren (Optimistic Concurrency Control):
    Annahme, dass keine Konflikte auftreten. Drei Phasen: Lesephase – Validierungsphase – Schreibphase. Die Objekte werden möglichst lange nicht gesperrt, scheitert allerdings die Validierung, muss reagiert werden.
  - Zeitstempel-Verfahren (Timestamp Techniques, Timestamp Ordering):
    Jede Transaktion erhält eine Zeitmarke (Startzeitpunkt). Synchronisiert wird so, dass die Abarbeitung äquivalent zu einem seriellen Schedule ist. Ein Konflikt tritt auf, falls eine Transaktion z.B. eine Leseanforderung auf ein Objekt absetzt, das bereits von einer 'jüngeren' Transaktion verändert wurde. Konflikte werden durch einen Neustart einer betroffenen Transaktion gelöst.
    Diese Methode hat im Zusammenhang mit optimistischen Sperrverfahren und verteilten Datenbanken an Bedeutung gewonnen.

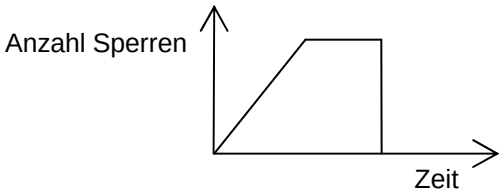
- Two-Phase Locking (2PL, Zwei-Phasen Sperrprotokoll):
  Für jede Transaktion muss gelten
  - Bevor auf ein Objekt zugegriffen wird, muss auf das Objekt eine (entsprechende) Sperre angelegt werden.
  - Nachdem eine Sperre aufgehoben wurde, kann keine neue mehr angelegt werden.

  Die Sperrstrategie besteht somit aus zwei Phasen
  - Growing Phase (Wachstumsphase): Alle nötigen Sperren werden gesetzt, ohne eine Sperre wieder aufzuheben. Wenn alle Sperren gesetzt sind, ist der Locked Point erreicht.
  - Shrinking Phase (Schrumpfungsphase): Alle Sperren werden aufgehoben, keine neuen gesetzt.

  Es kann gezeigt werden, dass Two-Phase Locking Serialisierbarkeit garantiert (allerdings Deadlocks nicht verhindert).



- Strict Two-Phase Locking (S2PL, Striktes Zwei-Phasen Sperrprotokoll):
  Alle Sperren werden erst am Ende der Transaktionen freigegeben (strenger als 2PL)



- Sperrobjekte (Sperrgranulat, Granularity):
  - Hierarchie:
    - Datenbank
    - Tabelle
    - physischer Block / Seite
    - Zeile
  - Vorteil feiner Sperreinheiten (Fine Granularity): Höhere Parallelität zwischen den Transaktionen ist möglich.
  - Vorteil grober Sperreinheiten (Coarse Granularity): Geringerer Verwaltungsaufwand für die Sperren ist notwendig.
  - In der Praxis stellt Sperren auf Zeilenebene (Row-Level Locking) die feinste Sperrform dar.

- Sperrmodi:
  - X-Lock (Exclusive Lock, Schreibsperre, Exklusive Sperre): Wenn eine Transaktion A das Objekt R mit X-Lock gesperrt hat, kann eine andere Transaktion B das Objekt R weder mit X-Lock noch mit S-Lock sperren; in diesem Fall muss Transaktion B solange warten, bis Transaktion A das Objekt R freigegeben hat.
  - S-Lock (Shared Lock, Lesesperre, Gemeinsame Sperre): Wenn eine Transaktion A das Objekt R mit S-Lock gesperrt hat, kann eine andere Transaktion B dasselbe Objekt zwar zusätzlich ebenfalls mit S-Lock, nicht aber mit X-Lock sperren; im zweiten Fall muss Transaktion B solange warten, bis Transaktion A das Objekt R freigegeben hat.
    Transaktion A kann ein Objekt R, das ausschließlich von ihr mit S-Lock belegt ist, sehr wohl mit X-Lock sperren (Upgrade, Sperrkonversion).
  - Kompatibilitätsmatrix (Compatibility Matrix)

		Transaktion A hat Objekt gesperrt mit		
		X-Lock	S-Lock	kein Lock
Transaktion B kann Objekt sperren mit	X-Lock	nein	nein	ja
	S-Lock	nein	ja	ja
	kein Lock	ja	ja	ja

  - Konkrete Systeme arbeiten mit:
    - nur einem Sperrmodus (X-Lock, binärem Sperren): geringerer Verwaltungsaufwand
    - beiden Sperrmodi (S-Lock und X-Lock): höhere Parallelität

- Für jedes System muss der Sperrmechanismus (Sperrstrategie, Locking-Mechanism) definiert werden: Wodurch (explizit oder implizit) bei welchem Objekt welche Sperre angebracht wird und wodurch diese wieder aufgehoben wird.

- Die obigen Beispiele werden nun unter Verwendung des folgenden Sperrmechanismus 'SPERR1' behandelt:
  - Sperrobjekte: Zeilen
  - Sperrmodi: Vor FETCH wird ein S-Lock abgesetzt, vor UPDATE wird ein X-Lock durchgeführt
  - Alle Sperren werden bei Transaktionsende aufgehoben

- Lost Update Problem

Transaktion A	Zeit	Transaktion B
-	|	-
FETCH R	t1	-
(S-Lock auf R)	|	-
-	t2	FETCH R
-	|	(S-Lock auf R)
UPDATE R	t3	-
(Request: X-Lock auf R)	|	-
warten	t4	UPDATE R
warten	|	(Request: X-Lock auf R)
warten	|	warten
warten	↓	warten

Das verwendete Sperrverfahren führt bei diesem zeitlichen Ablauf zu einem Deadlock.

- Inconsistent Analysis Problem

Transaktion A	Zeit	Transaktion B
-	|	-
FETCH R1	t1	-
(S-Lock auf R1)	|	-
sum=sum+stand (sum: 40)	|	-
FETCH R2	t2	-
(S-Lock auf R2)	|	-
sum=sum+stand (sum: 90)	|	-
-	|	-
-	t3	FETCH R3
-	|	(S-Lock auf R3)
-	|	stand=stand-10
-	t4	UPDATE R3 (stand: 20)
-	|	(X-Lock auf R3, Upgrade)
-	t5	FETCH R1
-	|	(S-Lock auf R1)
-	|	stand=stand+10
-	t6	UPDATE R1 (stand: 50)
-	|	(Request: X-Lock auf R1)
-	|	warten
FETCH R3	t7	warten
(Request: S-Lock auf R3)	|	warten
warten	|	warten
warten	↓	warten

Das System befindet sich in einer Deadlock-Situation.

- Uncommited Dependency Problem

Transaktion A	Zeit	Transaktion B
-	|	-
-	t1	UPDATE R
-	|	(X-Lock auf R)
FETCH R	t2	-
(Request: S-Lock auf R)	|	-
warten	t3	ROLLBACK
warten	|	(Release: X-Lock)
(Resume: FETCH R)	|	-
(S-Lock auf R)	|	-
	↓	

Transaktion A liest konsistente Daten nach dem Abschluss der Transaktion B.

- Realisierung von Locking:
  In einer Tabelle werden pro Transaktion alle Sperren und Sperranforderungen gespeichert
  - Identifikation der Transaktion
  - Identifikation des Sperrobjekts (z.B. Tabellenname, Zeilennummer)
  - Art der Sperre (S, X)
  - Kennzeichen, ob Sperre oder Sperranforderung

14.4 Deadlocks

- Deadlock (Verklemmung, Deadly Embrace): Wechselseitges Aufeinanderwarten (geschlossene Kette Aufeinanderwartens)

- DieTransaktion kann von sich aus nicht erkennen, ob sie in eine Deadlock-Situation kommt oder sich in einer Deadlock-Situation befindet:

Transaktion A	Zeit	Transaktion B
-		-
FETCH R	t1	-
(S-Lock auf R)		-
-	t2	FETCH R
-		(S-Lock auf R)
UPDATE R	t3	-
(Request: X-Lock auf R)		-
warten		DISPLAY R
warten		DISPLAY 'Weiter ja/nein?'
warten		READ Antwort
Warten	t4	COMMIT
Warten		(Release: S-Lock auf R)
(Resume: UPDATE R)		-
(X-Lock auf R)		-
	↓	

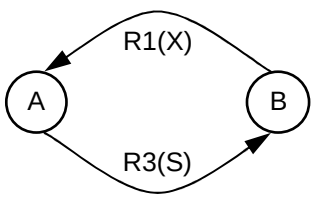
  Beim Warten nach der UPDATE-Anforderung (t3) ist für Transaktion A nicht ersichtlich, ob ein Deadlock vorliegt (beim Lost Update Problem) oder nicht (in diesem Beispiel)

- Die Entscheidung, ob eine Deadlock-Situation vorliegt, muss ein übergeordnetes System (Datenbanksystem, Transaktionsmonitor) treffen.

- Das Deadlock-Problem wird systemmäßig im Wesentlichen auf zwei Arten behandelt:

  - Deadlock-Vermeidung (avoidance):
    Besteht bei einer Sperranforderung die Gefahr, dass ein Deadlock auftreten kann, wird die Transaktion abgebrochen (prevention).
    Alle Objekte, die die Transaktion verwenden will, werden zu Beginn der Transaktion gesperrt (preclaiming).
    Mit dem Zeitstempel-Verfahren ist ebenfalls eine Deadlock-Vermeidung möglich (timestamping).

  - Deadlock-Erkennung (detection):
    Erkennung:
    - Zur Deadlock-Erkennung wird ein (gerichteter) Wartegraph (Wait-For Graph) geführt:
      - Knoten: Im System vorhandene Transaktionen
      - Kanten: Wenn Transaktion A ein Objekt sperren will (Request), Transaktion B aber eine entsprechende Sperre auf dieses Objekt aufrecht hält und Transaktion A daher warten muss, wird eine Kante von A nach B gezeichnet. Zur besseren Übersicht können die Kanten auch mit dem Namen des Objekts und der Art der gewünschten Sperre bezeichnet werden, z.B. R1(X), R3(S)
      - Eine Deadlock-Situation liegt genau dann vor, wenn im Wartegraph Zyklen (Kreise) enthalten sind. Der Wartegraph muss von Zeit zu Zeit (wann?) auf das Vorhandensein von Zyklen geprüft werden.
      - Beispiel: Wartegraph im Inconsistent Analysis Problem



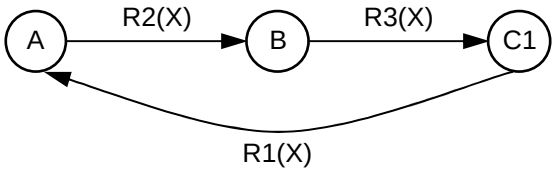
Beseitigung:
- Eine an der Deadlocksituation beteiligte Transaktion ist als 'Opfer' auszuwählen und zurückzusetzen (diejenige mit den geringsten Rücksetzkosten)

- Timeout: Sperranforderungen werden nach einer gewissen Zeitspanne für unerfüllbar erklärt oder jede Transaktion hat eine bestimmte Zeitdauer vorgegeben, nach deren Ablauf sie abgebrochen wird. Deadlocksituationen werden dadurch zwar aufgelöst, jedoch erfolgen auch Abbrüche von Transaktionen, die in einer normalen Wartesituation sind oder aus irgend einem anderen Grund länger dauern.

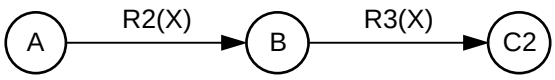
- Ein Deadlock kann auch zyklisch mehrere Transaktionen betreffen:

Zeit	Transaktion A	Transaktion B	Transaktion C1	Transaktion C2
↓	-	-	-	-
	FETCH R1	-	-	-
	-	FETCH R2	-	-
	-	-	FETCH R3	FETCH R3
	UPDATE R2	-	-	-
	warten	UPDATE R3	-	-
	warten	warten	UPDATE R1	UPDATE R3
	warten	warten	warten	-

  - Im Fall C1 liegt ein Deadlock vor, Wartegraph:



  - Im Fall C2 liegt kein Deadlock vor, Wartegraph:



- Eine Deadlock-Situation kann auch auftreten, wenn nur X-Locks angewendet werden:

Transaktion A	Zeit	Transaktion B
-	↓	-
FETCH R1	t1	-
(X-Lock auf R1)	↓	-
-	t2	FETCH R2
-	↓	(X-Lock auf R2)
UPDATE R2	t3	-
(Request: X-Lock auf R2)	↓	-
warten	t4	UPDATE R1
warten	↓	(Request: X-Lock auf R1)
warten	↓	warten
warten	↓	warten

- Die meisten Systeme erlauben eine programmgesteuerte Abfrage, ob ein Objekt (Zeile) gesperrt ist. Daher soll zur Deadlock-Vermeidung in der Logik des Anwenderprogramms bei einer Sperranforderung geprüft werden, ob das Objekt nicht bereits gesperrt ist und in diesem Fall die Transaktion programmgesteuert reagieren (z.B. Entsprechende Meldung geben und abbrechen, Rückfragen ob und wielange gewartet werden soll)

- Durch Angabe entsprechender Optionen beim Datenzugriff kann in den meisten Systemen der Standard-Sperrmechanismus programmspezifisch individuell verändert werden.
  Beispiele: Lesen mit No-Lock (Dirty Read), Lesen mit X-Lock, etc.

- Übung 1): Wie stellen sich die drei, in der Problemstellung beschriebenen, Situationen dar, wenn nur der Sperrmodus X-Lock (vor FETCH und UPDATE) verwendetet wird (Sperrmechanismus 'SPERR2')?

- Übung 2): Ergeben sich beim Inconsistent Analysis Problem Änderungen im Verhalten, wenn Transaktion B die Lese- / Updateoperationen (R3,R1) in umgekehrter Reihenfolge (R1,R3) durchführt (für SPERR1 und SPERR2)?

- Übung 3): Ergeben sich beim Inconsistent Analysis Problem Änderungen im Verhalten, wenn Transaktion B zuerst beide FETCH-Operationen (R3, R1 oder R1, R3) und dann beide UPDATE-Operationen ausführt (für SPERR1 und SPERR2)?

- Übung 4): Ergeben sich beim Inconsistent Analysis Problem Änderungen im Verhalten, wenn Transaktion B die Lese- / Updateoperationen bereits nach dem FETCH R1 von A durchführt (für SPERR1 und SPERR2)?

- Übung 5)
  Sperrmechanismus:
  - Sperrobjekte sind Zeilen
  - S-Lock vor FETCH
  - X-Lock vor UPDATE
  - Aufheben der Sperren bei Transaktionsende (COMMIT)

Zeitpunkt	Transaktion	Operation
t1	T1	FETCH A
t2	T5	FETCH B
t3	T4	FETCH C
t4	T3	FETCH A
t5	T5	UPDATE B
t6	T4	FETCH A
t7	T1	FETCH C
t8	T2	FETCH B
t9	T1	UPDATE C
t10	T5	COMMIT
t11	T3	UPDATE A
t12	…	…

  Zeitpunkt t12:
  - Welche Transaktionen haben / wollen welche Objekte in welcher Weise gesperrt / sperren?
  - Wie sieht der Wartegraph aus?
  - Liegt ein Deadlock vor?
    ja - welche Transaktion(en) muss (müssen) abgebrochen werden?
    nein - in welcher Reihenfolge können die Transaktionen beendet werden?

- Übung 6)
  Sperrmechanismus wie in Übung 5), folgender Schedule ist gegeben:

  r2(B); r1(A); w2(B); r1(B); r2(A); r3(C); w2(C); w4(B); w3(A);

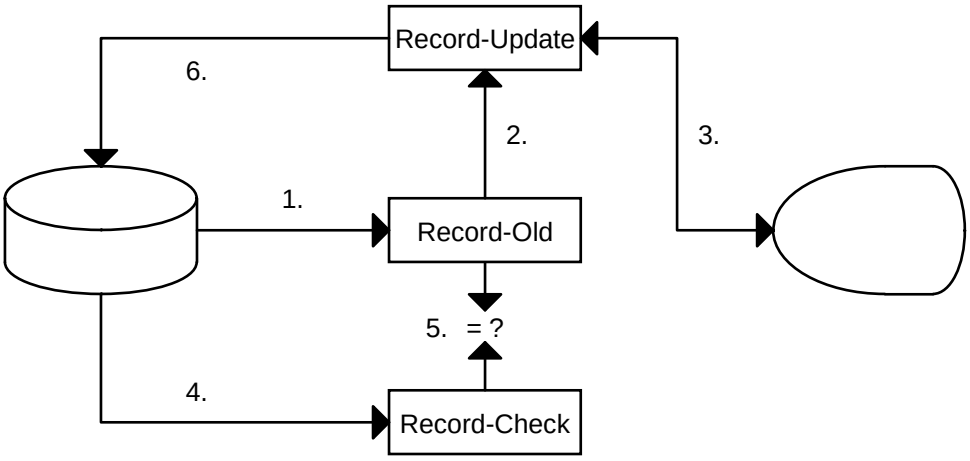
- Übung 7)
  Folgender Schedule von Sperranforderungen ist gegeben:

  s1(A); s3(A); x2(D); x3(C); s1(C); s3(D); x4(A);

14.5 Die Re-Read-Methode

- Am Beispiel der Änderung eines Satzes in einem Multi-Tasking-Betrieb wird das Re-Read-Verfahren (Read-Before-Write-Verfahren) gezeigt, das folgenden Kriterien genügt:
  - Es darf zu keinen Inkonsistenzen kommen
  - Die Objekte dürfen nicht zu lange (im Besonderen über Benutzereingaben hinweg) gesperrt sein (im Besonderen mit X-Lock)
  - Es darf keine Deadlock-Situationen auftreten (wenn das System keinen Mechanismus zur Deadlock-Erkennung hat)

- Graphische Darstellung:



- Beschreibung:
  1. Satz nach Record-Old ohne Sperre einlesen
  2. Satz (oder Teile davon) aus Record-Old nach Record-Update kopieren
  3. Satz Record-Update (oder Teile davon) vom Benutzer ändern lassen
  4. Satz nach Record-Check mit X-Lock noch einmal einlesen (Re-Read)
  5. Ganzen Satz in Record-Old mit ganzem Satz in Record-Check vergleichen:
  6. wenn gleich, dann Satz auf Grund von Record-Update ändern
     wenn ungleich, dann Meldung 'Satz wurde mittlerweile geändert' an den Benutzer

14.6 Zusätzlicher Sperrmodus U-Lock

- Update Lock, Promoteable Lock, Incremental Lock, Aktualisierende Sperre

- Wird bei Lese-Operationen angegeben, wenn danach eine Änderungs-Operation beabsichtigt ist

- Geringere Deadlock-Gefahr als bei Lesen mit S-Lock, höhere Parallelität als bei Lesen mit X-Lock

- Kompatibilitätsmatrix

	X-Lock	U-Lock	S-Lock	kein Lock
X-Lock	nein	nein	nein	ja
U-Lock	nein	nein	ja	ja
S-Lock	nein	ja	ja	ja
kein Lock	ja	ja	ja	ja

- Sperrmechanismus 'SPERR3':
  vor FETCH S-Lock, vor FETCH mit folgendem UPDATE U-Lock, vor UPDATE X-Lock

- Vergleich SPERR1 mit SPERR3
  - Lost Update Problem: kein Deadlock (zweites FETCH muss warten, da zwei U-Locks nicht kompatibel)
  - Inconsistent Analysis Problem: gleiches Verhalten wie bei SPERR1 (Deadlock)
  - Uncommited Dependency Problem: gleiches Verhalten wie bei SPERR1 (kein Deadlock)

- Lesen mit U-Lock bei konkreten Systemen in folgender Form (beispielhaft)
  SELECT ... FOR UPDATE ...
  SQL92-Norm: DECLARE ... CURSOR ... FOR UPDATE
  MS SQLServer: SELECT ... FROM X (UPDLOCK) ...

14.7 Isolation Levels

- Isolation Level: Grad, zu dem Datenkonsistenz (zu Lasten der Parallelität) eingehalten wird (ursprünglich 'Degree of Consistency').
  Definition des Verhaltens einer Transaktion bezüglich Concurrency versus Integrity.
  Grad der gegenseitigen Beinflussung von Transaktionenen (Interference of Transactions).

- Es werden nicht für einzelne Operationen Sperrmodi angegeben, sondern für Transaktionen Isolation Levels definiert, die festlegen welche Serialisierungsprobleme auftreten dürfen und welche nicht

- 4 Isolation Levels (Intermediate SQL-92):
  - SERIALIZABLE hoch keine Gefahr von Inkonsistenzen - niedere Parallelität
  - REPEATABLE READ |
  - READ COMMITTED ↓
  - READ UNCOMMITTED nieder Gefahr von Inkonsistenzen - hohe Parallelität

  - ```
    SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED |
                                      READ COMMITTED   |
                                      REPEATABLE READ  |
                                      SERIALIZABLE       }
    ```

  - Standardannahme: SERIALIZEABLE

- 3 Arten von Serialisierungsproblemen:

  - Dirty Read: Transaktion A liest eine Zeile, die von einer anderen Transaktion B geändert wurde. B führt ein Rollback durch und A arbeitet mit einer Zeile, die so nie existiert hat (Uncommited Dependency).

  - Nonrepeatable Read (Fuzzy Read): Transaktion A liest eine Zeile; eine andere Transaktion B ändert diese Zeile. A liest diese Zeile nochmals und erhält eine andere Version derselben Zeile.

  - Phantoms: Transaktion A liest mehrere Zeilen, die einer bestimmten Bedingung genügen (Abfrage); eine andere Transaktion B fügt eine Zeile ein, die auch diese Bedingung erfüllt (oder macht in einer existierenden Zeile eine Änderung, sodass sie diese Bedingung dann auch erfüllt). A wiederholt die Abfrage und erhält ein anderes Ergebnis (mehr Zeilen).

- Folgende Tabelle gibt an, ob bei einem bestimmten Isolation Level ein bestimmtes Problem auftreten kann:

	Dirty Read	Nonrepeatable Read	Phantoms
READ UNCOMMITTED	ja	ja	ja
READ COMMITTED	nein	ja	ja
REPEATABLE READ	nein	nein	ja
SERIALIZABLE	nein	nein	nein

- Realisierung mit Sperren:
  - READ UNCOMMITTED kein Lock beim Lesen (alle Probleme treten auf)
  - READ COMMITTED S-Lock auf Zeile beim Lesen; nicht Two-Phase Locking (z.B. Lost Update tritt auf)
  - REPEATABLE READ S-Lock auf Zeile beim Lesen bis Transaktionsende
  - SERIALIZABLE einfach: S-Lock auf Tabelle(n) beim Lesen bis Transaktionsende
    komplex: Predicate Locking; Zugriffspfad (Access Path) für die angegebene Bedingung sperren (z.B. bestimmte Indexeinträge); Sperren 'nicht existenter' Zeilen

- Varianten der Isolations Levels von SQL-92:
  - DIRTY READ = READ UNCOMMITED (INFORMIX)
  - CURSOR STABILITY (INFORMIX, SQLBase)
    zwischen READ COMMITTED und REPEATABLE READ
  - RELEASE LOCKS, READ ONLY (SQLBase)
  - VERSIONING (ODBC)
    ```

14.8 Hierarchische Sperrverfahren

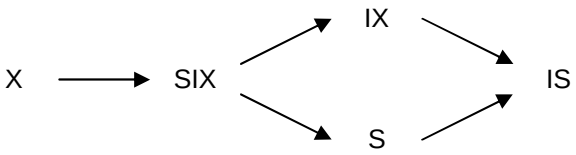
- Sperren von verschiedenen Granulaten / auf verschiedenen Hierarchien
  (Datenbank – Tabelle – (Seite) – Zeile)
  Multiple-Granularity Locking (MGL)

- Problem: Wenn eine Transaktion ein Objekt sperren will, muss für alle untergordenten Objekte überprüft werden, ob ein Sperrkonflikt auftritt

- Beim Sperren eines Objekts muss Top-Down bei allen in der Hierarchie übergeordneten Objekten eine geeignete Sperre erworben werden, das Freigeben erfolgt Bottom-Up

- Einführung von Intent Locks (Anwartschaftssperren)
  (nicht sinnvoll für Objekte auf unterster Hierarchiestufe):
  - IS-Lock (Intent Shared Lock): weiter unten in der Hierarchie sind S-Locks beabsichtigt
  - IX-Lock (Intent Exclusive Lock): weiter unten in der Hierarchie sind X-Locks beabsichtigt
  - SIX-Lock (Shared Intent Exclusive Lock): sperrt Objekt mit S-Lock und siehe IX

- Stärke der Sperrmodi (links ist stärker):

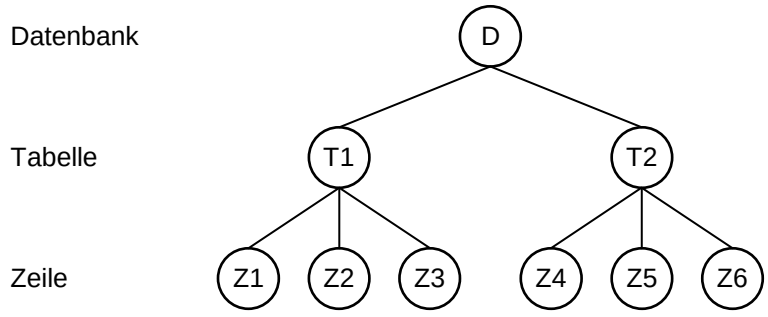


- Sperrstrategie
  - Bevor eine Transaktion ein Objekt mit S-Lock sperren kann, müssen alle seine übergeordneten Objekte Top-Down (von oben nach unten) mit IS-Lock oder stärker gesperrt werden
  - Bevor eine Transaktion ein Objekt mit X-Lock sperren kann, müssen alle seine übergeordneten Objekte Top-Down (von oben nach unten) mit IX-Lock oder stärker gesperrt werden
  - Die Sperren werden Bottom-Up (von unten nach oben) freigegeben, d.h. es wird die Sperre nicht freigegeben, wenn die Transaktion noch untergeordnete Objekte gesperrt hat

- Kompatibilitätsmatrix

	X	SIX	IX	U	S	IS
X	-	-	-	-	-	-
SIX	-	-	-	-	-	+
IX	-	-	+	-	-	+
U	-	-	-	-	+	+
S	-	-	-	+	+	+
IS	-	+	+	+	+	+

- Beispiel



Transaktion1 sperrt Z1 mit X-Lock
Transaktion2 sperrt T2 mit S-Lock
Transaktion2 sperrt Z2 mit S-Lock
Transaktion3 sperrt Z5 mit S-Lock
Transaktion3 sperrt Z6 mit X-Lock

Welche Transaktionen bringen auf welchen Objekten welche Sperren an?