Vorlesung Implementierung von Datenbanksystemen

12. Synchronisation

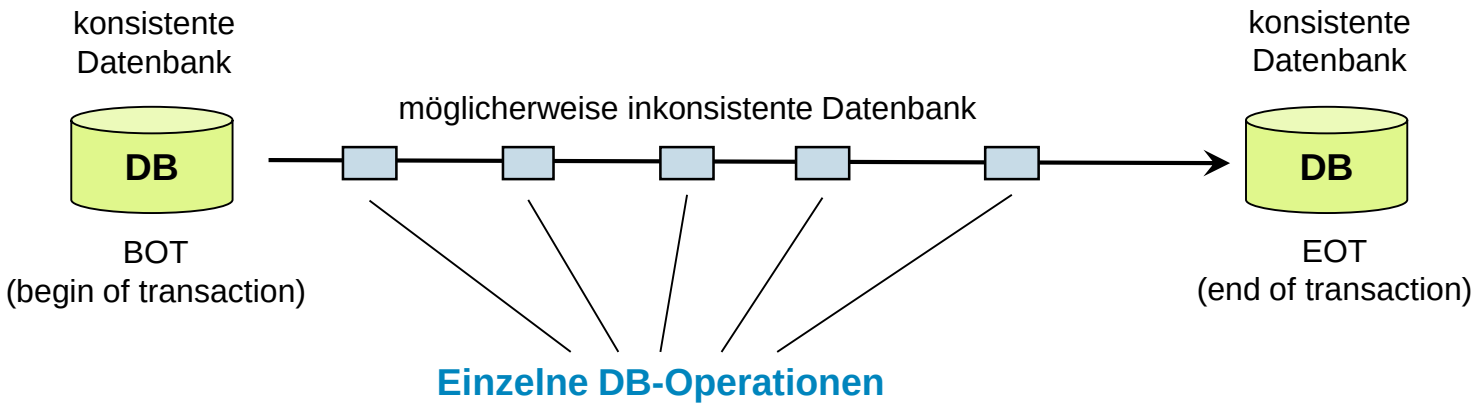
Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Implementierung von Transaktionen

- **Transaktion (Wiederholung)**
  - Zusammenfassung von aufeinander folgenden DB-Operationen, die eine Datenbank von einem konsistenten Zustand in einen wiederum konsistenten Zustand überführen



- **Beachte:**
  - Eine Transaktion muss nicht notwendigerweise in einem *anderen* konsistenten Zustand enden – es kann auch derselbe wie zu Anfang sein.
  - Transaktionen werden immer beendet:
    - Normal (**commit**): Änderungen sind permanent in der DB.
    - Anormal (**abort** / **rollback**): Bereits durchgeführte Änderungen werden zurückgenommen.

Anmerkungen zur Konsistenz in DBS

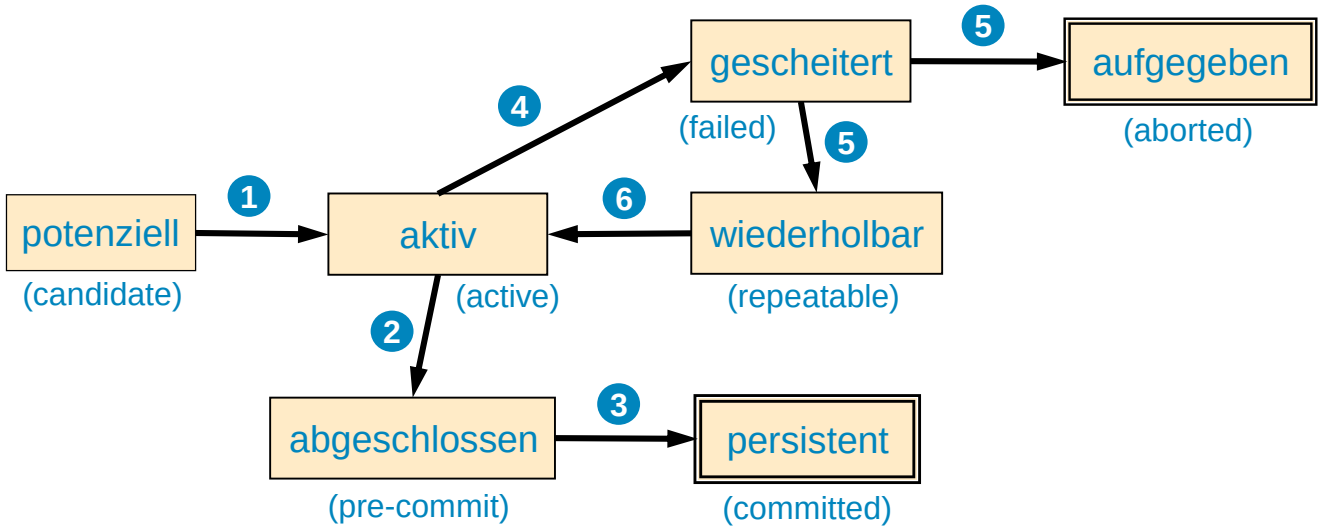
- **Beispiel**
  - Vorgegebene Konsistenzbedingung: Es muss immer x = y gelten.

```
BOT;                // Begin of Transaction; DB ist konsistent
read(x);
read(y);
x := x + 1;
write(x);           // DB ist zeitweilig inkonsistent
y := y + 1;
write(y);
EOT;                // End of Transaction; DB ist wieder konsistent
```

- **Arten von Konsistenz**
  - Datenbankkonsistenz
    - Alle (auf der DB definierten) Konsistenzbedingungen sind erfüllt.
  - **Transaktionskonsistenz** (operationelle Integrität)
    - Der nebenläufige Ablauf der Transaktionen ist korrekt.

Zustände von Transaktionen

- **Zustandsübergangsdiagramm**



1. Inkarnieren: TA ist angemeldet und wechselt in den Zustand aktiv
2. Beenden: TA ist beendet, aber Änderungen sind noch nicht permanent eingebracht
3. Festschreiben: Änderungen werden eingebracht
4. Abbrechen: TA ist fehlgeschlagen, aber noch nicht zurückgesetzt
5. Zurücksetzen: Änderungen werden rückgängig gemacht
6. Neustarten: TA wird wiederholt

ACID-Eigenschaften von Transaktionen

- **Atomarität (atomicity)**
  - Unteilbarkeit gemäß Transaktionsdefinition (Begin – End)
  - Alles-oder-Nichts-Prinzip, d.h. das DBS garantiert
    - entweder die vollständige Ausführung einer Transaktion
    - oder die Wirkungslosigkeit der Transaktion (und damit aller beteiligten Operationen).
- **Konsistenzerhaltung (consistency)**
  - Eine erfolgreiche Transaktion garantiert, dass alle Konsistenzbedingungen (Integritätsbedingungen) eingehalten worden sind.
- **Isolation (isolation)**
  - Mehrere Transaktionen laufen voneinander isoliert ab und benutzen keine (inkonsistenten) Zwischenergebnisse anderer Transaktionen.
- **Dauerhaftigkeit (durability)**
  - Alle Ergebnisse erfolgreicher Transaktionen müssen persistent gemacht werden (worden sein).

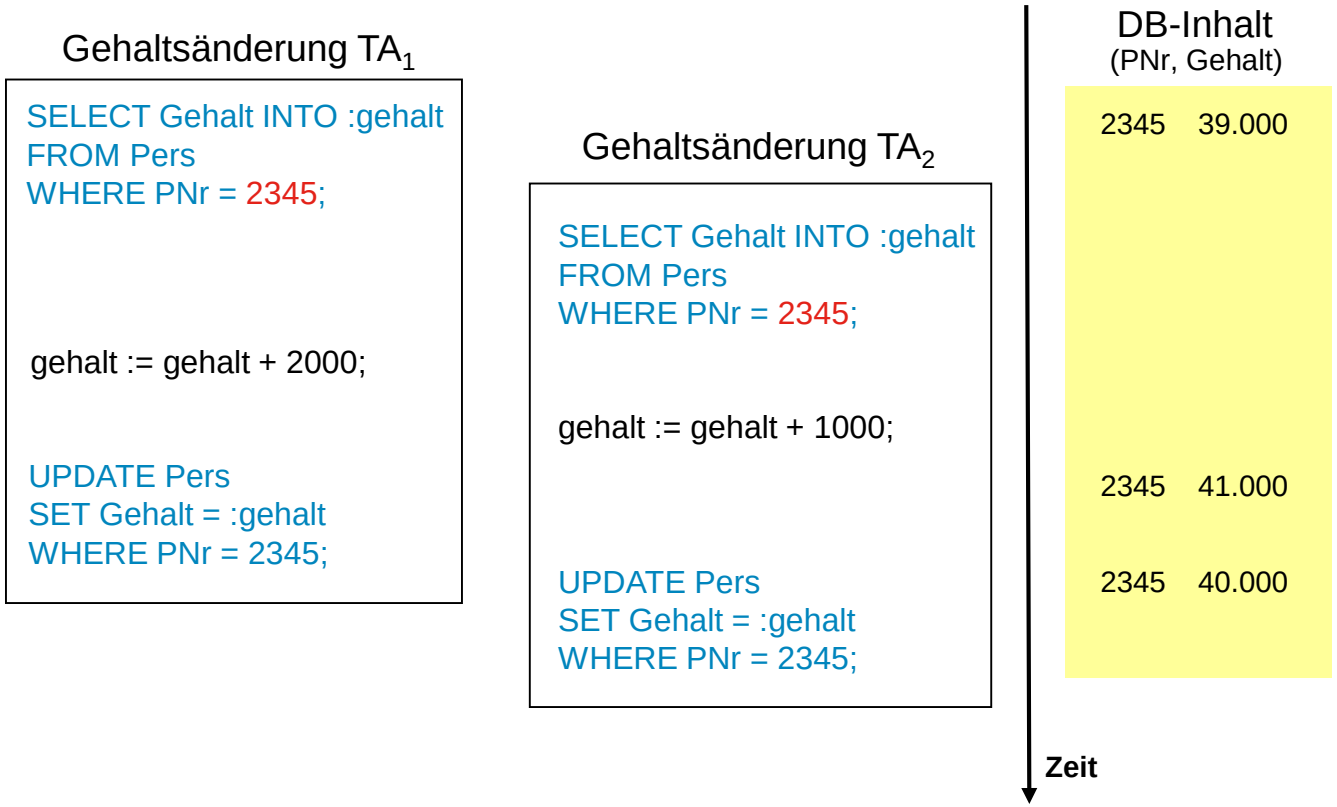
12.1 Synchronisation allgemein

- **Ziel**
  - Erhaltung der **Transaktionskonsistenz** (= operationelle Integrität) im Mehrbenutzerbetrieb
- **Gründe für den Mehrbenutzerbetrieb**
  - Verteilung der Clients generell (z.B. Geldausgabeautomat)
  - Prozessornutzung auch während systembedingter und benutzungsbedingter Transaktionsunterbrechungen (Wartezeiten, z.B. Ein-/Ausgabe)
  - Kommunikationsvorgänge in verteilten Systemen
- **Gegenstand der Synchronisation**
  - Vermeidung der gegenseitigen Beeinflussung von Lese- und Schreiboperationen
  - Verhinderung von **Anomalien** im Mehrbenutzerbetrieb

Vorgehensweise

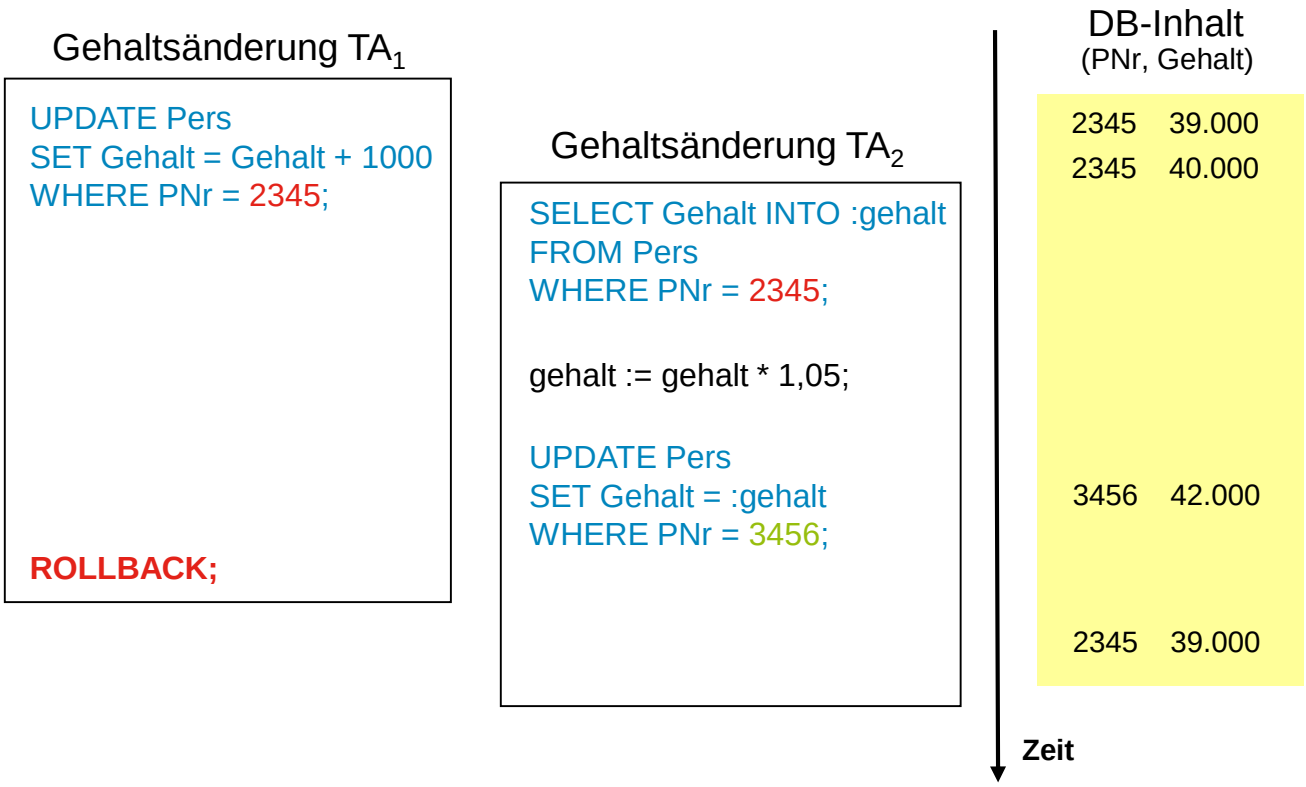
- **Mögliche Anomalien bei fehlender Synchronisation betrachten**
  - Verlorengegangene Änderung (lost update, dirty write)
  - Abhängigkeit von nicht freigegebener Änderung (dirty read)
  - Inkonsistente Analyse (non-repeatable read)
  - (Phantom-Problem)
- **Unbefriedigende Lösung: Serialisierung**
  - Alle Transaktionen (TAen) nacheinander ausführen (Einbenutzerbetrieb)
  - Aber: führt bei langen TAen zu großen Wartezeiten für die anderen TAen (Scheduling-Fairness)
- **Statt dessen:**
  - Sicherstellen der operationellen Integrität
    durch eine "virtuelle" serielle Ausführung der TAen
    ("Logischer" Einbenutzerbetrieb! Als ob jede TA ganz allein auf der DB wäre)
- **Unterschiedliche Sichtweisen**
  - A posteriori: **Serialisierbarkeitstheorie**
  - A priori: **Sperrverfahren**

Beispiel für verlorengegangene Änderung



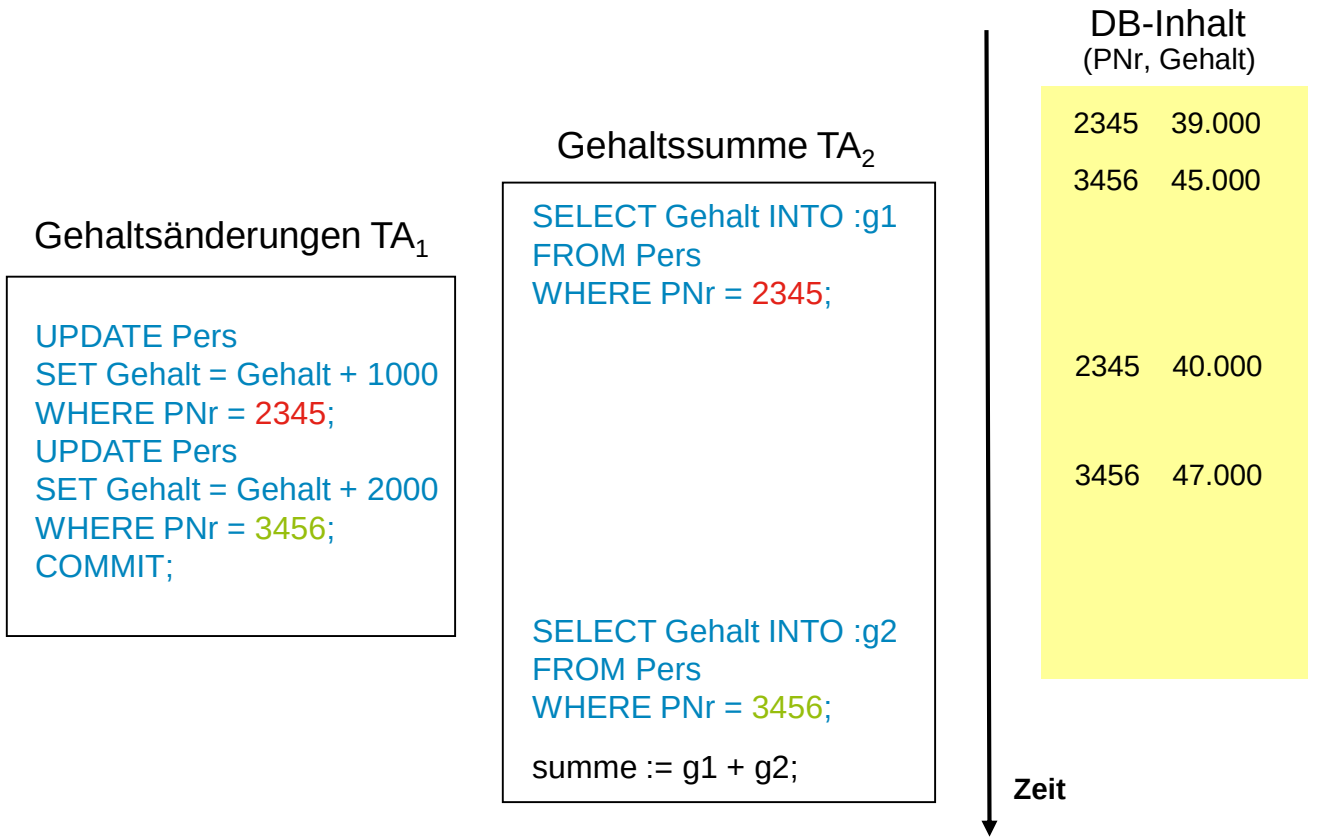
- Konkurrierendes Verändern desselben (!) Datenelements
- Write-write dependency
- Lösung: exklusives Änderungsrecht für Schreiber

Beispiel für "Dirty Read"



- Abhängigkeit von nicht freigegebener Änderung (eigentlich: rekursives Zurücksetzen)
- Write-read dependency
- Lösung:
  - Ermöglichen des isolierten Zurücksetzens
  - Lesen geänderter Daten immer erst dann, wenn sie freigegeben sind !

Beispiel für inkonsistente Analyse



- Während der Verarbeitung von TA_2 wird der Datenbestand durch TA_1 verändert.
- TA_2 sieht konsistente Zustände, jedoch unterschiedliche !
- Read-write dependency
- Lösung: Auch Leser müssen ihre Daten schützen – vor den Änderungen anderer

Abläufe (Schedules, Histories)

- **Kompakte Schreibweise für das, was passiert ist**
  - Leseoperation einer Transaktion *i* auf Datenobjekt *A*: $r_i[A]$
  - Schreiboperationen einer Transaktion *i* auf Objekt *A*: $w_i[A]$
  - Erfolgreicher Abschluss (commit) einer Transaktion: c_i
  - Abbruch (abort, rollback) einer Transaktion: a_i

- **Beispiele von oben in dieser Notation:**
  - Verlorengegangene Änderung:
    - r_1[PNr = 2345], r_2[PNr = 2345], $\mathbf{w_1[PNr = 2345]}$, $\mathbf{w_2[PNr = 2345]}$, c_1, c_2
  - "Dirty Read":
    - $\mathbf{w_1[PNr = 2345]}$, r_2[PNr = 2345], $\mathbf{w_2[PNr = 3456]}$, a_1, c_2
  - Inkonsistente Analyse:
    - r_2[PNr = 2345], $\mathbf{w_1[PNr = 2345]}$, $\mathbf{w_1[PNr = 3456]}$, r_2[PNr = 3456], c_1, c_2

12.2 Serialisierbarkeit

- **Ziel**
  - Garantie, dass beim verzahnten Ablauf mehrerer Transaktionen die Datenbank konsistent bleibt und jede Transaktion einen konsistenten Zustand sieht
- **Eigenschaft einer Transaktion (das C von ACID)**
  - Wenn eine Transaktion TA *allein* auf einer konsistenten DB ausgeführt wird, dann terminiert die TA (irgendwann) und hinterlässt die DB in einem konsistenten Zustand.
  - **... folglich:**

    Wenn alle Transaktionen *seriell* (hintereinander) ausgeführt werden, dann bleibt die Konsistenz der Datenbank erhalten.

- **Beispiel**

```
TA1: read(A);
     A := A - 50;
     write(A);
     read(B);
     B := B + 50;
     write(B);
```

```
TA2: read(A);
     temp := A * 0.1;
     A := A - temp;
     write(A);
     read(B);
     B := B + temp;
     write(B);
```

Beispiel zur Serialisierbarkeit

- **Zwei serielle Abläufe denkbar**
  - Anfangswerte A = 1000 und B = 2000
  - **Ablauf 1**: TA1 ; TA2 (Semikolon = "vor")
    - TA1: A = **950** B = **2050**
    - TA2: A = **855** B = **2145**
  - **Ablauf 2**: TA2 ; TA1
    - TA2: A = **900** B = **2100**
    - TA1: A = **850** B = **2150**
- **Merke:**
  - Beide Abläufe sind korrekt,
    obwohl sie unterschiedliche (aber jeweils konsistente) Ergebnisse liefern.
  - Bei n Transaktionen sind **n!** serielle Abläufe möglich.
- **Problem**
  - Nicht alle verzahnten Abläufe sind korrekt – manche aber schon.

Beispiel für nicht korrekten parallelen Ablauf

- **Beispiel für verzahnten, jedoch nicht korrekten Ablauf**
  - **Ablauf 3**:

```
TA1: read(A);
     A := A - 50;
                    TA2: read(A);
                         temp := A * 0.1;
                         A := A - temp;
                         write(A);
                         read(B);
     write(A);
     read(B);
     B := B + 50;
     write(B);
                         B := B + temp;
                         write(B);
```

  - liefert A = **950** B = **2100**
    - "Lost Update" (sogar zweimal) !!!
  - Kompakt: $r_1[A]$, $r_2[A]$, $\mathbf{w_2[A]}$, $r_2[B]$, $\mathbf{w_1[A]}$, $r_1[B]$, $\mathbf{w_1[B]}$, $\mathbf{w_2[B]}$, $\mathbf{c_1}$, $\mathbf{c_2}$

Serialisierbare Abläufe

- **Definition: serialisierbarer Ablauf**
  - Ein Ablauf von Transaktionen ist **serialisierbar**, wenn er zu irgendeinem seriellen Ablauf der in ihm enthaltenen Transaktionen äquivalent ist.
- **Definition: Äquivalenz von Abläufen**
  - Seien TA_i und TA_j beliebige erfolgreiche Transaktionen und H und G zwei dazugehörige Abläufe.
    H und G sind **äquivalent**, wenn für alle Operationen auf einem beliebigen Datenobjekt A folgende Bedingungen gelten:

$$r_i[A] <_H w_j[A] \Leftrightarrow r_i[A] <_G w_j[A]$$
$$w_i[A] <_H r_j[A] \Leftrightarrow w_i[A] <_G r_j[A]$$
$$w_i[A] <_H w_j[A] \Leftrightarrow w_i[A] <_G w_j[A]$$

  - D.h. bei solchen Operations-Paaren muss die Reihenfolge in beiden Abläufen gleich sein.
    - Man sagt auch: Sie stehen in **Konflikt** miteinander.
  - Durch Vertauschen von zwei benachbarten Operationen, die nicht in Konflikt zueinander stehen, kann man einen äquivalenten Ablauf erzeugen.
  - (Etwas ungenau: Endzustand der DB evtl. unterschiedlich ...)

Beispiel für korrekten parallelen Ablauf

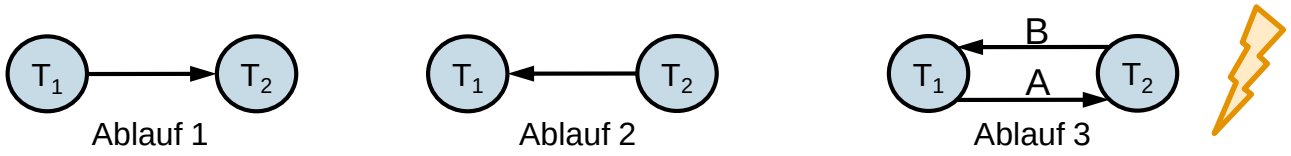
- **Beispiel für verzahnten, jedoch korrekten (serialisierbaren) Ablauf**
  - **Ablauf 4**:

```
TA1: read(A);
     A := A - 50;
     write(A);
                    TA2: read(A);
                         temp := A * 0.1;
                         A := A - temp;
                         write(A);
     read(B);
     B := B + 50;
     write(B);
                         read(B);
                         B := B + temp;
                         write(B);
```

  - liefert A = **855** B = **2145**
    - Also wie (serieller) Ablauf 1: TA1 ; TA2
  - Kompakt: $r_1[A]$, $\mathbf{w_1[A]}$, $r_2[A]$, $\mathbf{w_2[A]}$, $r_1[B]$, $\mathbf{w_1[B]}$, $r_2[B]$, $\mathbf{w_2[B]}$, $\mathbf{c_1}$, $\mathbf{c_2}$

Nachweis der Serialisierbarkeit

- **Abhängigkeitsgraph**
  - **Knoten**: einzelne Transaktionen
  - **Kanten**: Abhängigkeiten (Konflikt) zwischen zwei Transaktionen
    - Zwei TAen greifen auf dasselbe Objekt mit nicht reihenfolgeunabhängigen Operationen zu
      (z.B. Schreib-/Lese-, Lese-/Schreib-, Schreib-/Schreib-Konflikte).
  - Serialisierbarkeit liegt vor, wenn der Abhängigkeitsgraph *keine Zyklen* enthält.
  - Durch topologisches Sortieren ($O(N^2)$) erhält man die möglichen serialisierbaren Abläufe.
- **Beispiel:**



Lies: "T_1 vor T_2"

T_1 liest A, bevor T_2 A schreibt,
T_2 liest B, bevor T_1 B schreibt

  - Mehr dazu, wesentlich ausführlicher (und formeller!) in Vorl. Transaktionssysteme

Zusammenfassung Serialisierbarkeit

- **Zwei Abläufe sind äquivalent, wenn**
  - sie dieselben erfolgreich abgeschlossenen Transaktionen enthalten und
  - ihre Abhängigkeitsgraphen gleich sind.
- **Verhindert werden müssen (Berenson et al., 1995):**
  - Phänomen P0 (Dirty Write):
    - $w_1[x]$ … $w_2[x]$ … ((c_1 oder a_1) und (c_2 oder a_2) in beliebiger Reihenfolge)
  - Phänomen P1 (Dirty Read):
    - $w_1[x]$ … $r_2[x]$ … ((c_1 oder a_1) und (c_2 oder a_2) in beliebiger Reihenfolge)
  - Phänomen P2 (Non-repeatable Read):
    - $r_1[x]$ … $w_2[x]$ … ((c_1 oder a_1) und (c_2 oder a_2) in beliebiger Reihenfolge)
  - Phänomen P3 (Phantom):
    - $r_1[P]$ … $w_2[y \text{ in } P]$ … ((c_1 oder a_1) und (c_2 oder a_2) in bel. Reihenf.)
    - P steht für die Menge der Datenobjekte, die ein Prädikat erfüllen

Klammern bei c_i und a_i bedeuten nur Gruppierung, nicht "optional"

12.3 Implementierungsmethoden

- **Realisierung eines logischen Einbenutzerbetriebs**
  - Einführung von sog. **Sperren** (locks) für Zugriffe auf Datenobjekte
  - Für jedes benutzte Datenobjekt zentral in einer **Sperrtabelle** die Nutzung durch bestimmte Transaktionen registrieren
- **Arten von Sperren**
  - **X** (eXclusive)-Sperre (= Schreibsperre)
  - **S** (shared)-Sperre (= Lesesperre)
  - **Regeln für den Umgang mit Sperren**
    - Jedes Datenobjekt, auf das zugegriffen werden soll, muss *vorher* gesperrt werden.
    - Eine Transaktion fordert eine Sperre, die sie bereits besitzt, nicht noch einmal an.
    - Eine Transaktion muss die von anderen Transaktionen gesetzten Sperren beachten.
    - Am Ende einer Transaktion (und erst dann!) sind alle Sperren wieder freizugeben.

    (Eswaran, Gray, Lorie und Traiger, 1976)
  - **Serialisierbarkeit ist gewährleistet, wenn diese Regeln eingehalten werden!**

Synchronisation mit Sperren

- **Kompatibilitätsmatrix**
  - gibt Auskunft, ob eine Sperranforderung für ein (möglicherweise bereits gesperrtes) Objekt gewährt werden kann

Sperrmodus des Objekts

Sperranforderung der Transaktion

	keine Sperre	**S-Sperre**	**X-Sperre**
S-Sperre	Anforderung wird gewährt +	Anforderung wird gewährt +	Anforderung wird **nicht** gewährt −
X-Sperre	Anforderung wird gewährt +	Anforderung wird **nicht** gewährt −	Anforderung wird **nicht** gewährt −

Synchronisation mit Sperren (2)

- **Wann Sperren erwerben?**
  - **Statisches** Sperren
    - Zu Beginn der Transaktion alle Sperren anfordern ("preclaiming")
    - Nachteil: Man muss alles sperren, was man brauchen *könnte*.
  - **Dynamisches** Sperren
    - Während der Transaktion Sperren nach Bedarf anfordern
    - Nachteil: Verklemmungen (deadlocks) können auftreten.

- **Wann Sperren freigeben?**
  - Sperren müssen *bis zum Ende der Transaktion* gehalten werden, um Serialisierbarkeit zu garantieren
  - Abschwächung (früher freigeben) möglich
    - Siehe weiterführende Vorl. oder Literatur

Probleme mit Sperren

- **Sperrgranulat Tupel**
  - Nicht immer effizient
    - Aufwändig bei Transaktionen, die viele (alle) Tupel einer Relation benötigen
    - Große Sperrtabellen, hohe Verwaltungskosten
  - Nicht ausreichend, um alle Fehlerklassen auszuschließen
- **Phantom-Problem**
  - Sperren nur auf *existierende* Tupel
  - Nicht existierende Tupel können jederzeit eingefügt werden – sind dann aber nicht gesperrt (= Phantome).

Lesetransaktion
(Gehaltssumme bestimmen)

```
SELECT SUM(Gehalt)
INTO :summe1
FROM Pers
WHERE  ANr = 17;
        • • •
SELECT SUM(Gehalt)
INTO :summe2
FROM Pers
WHERE  ANr = 17;
if summe1 ≠ summe2
then <Fehler>;
```

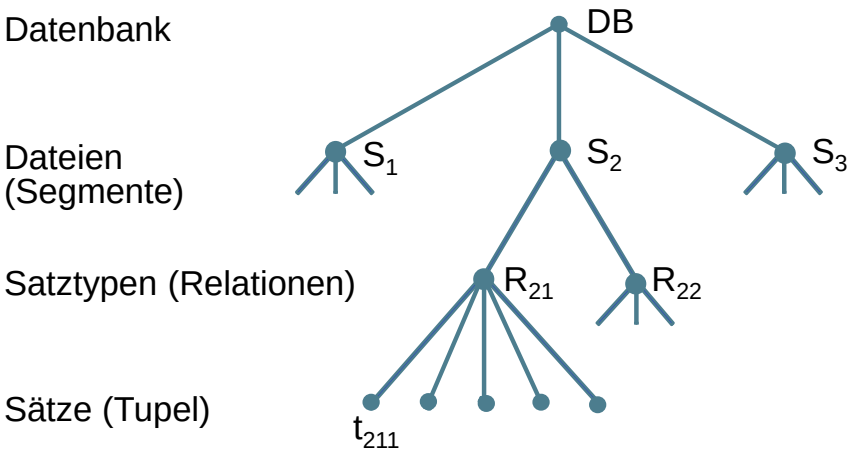
Änderungstransaktion
(Einfügen eines neuen Angestellten)

```
INSERT INTO Pers
  (PNr, ANr, Gehalt)
  VALUES (4567, 17, 55.000);
COMMIT;
```

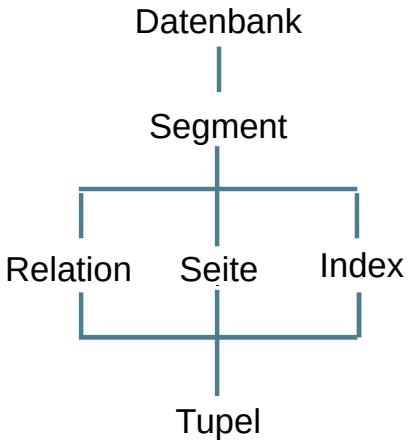
Zeit

Lösungsansatz

- **Hierarchische Schachtelung der Datenobjekte**
  - erlaubt Flexibilität bei der Wahl der Sperrgranulate
    - Synchronisation langer TAen auf Relationenebene
    - Synchronisation kurzer TAen auf Tupelebene



a) Beispiel einer Objekthierarchie



b) nicht-hierarchische Granularitäten

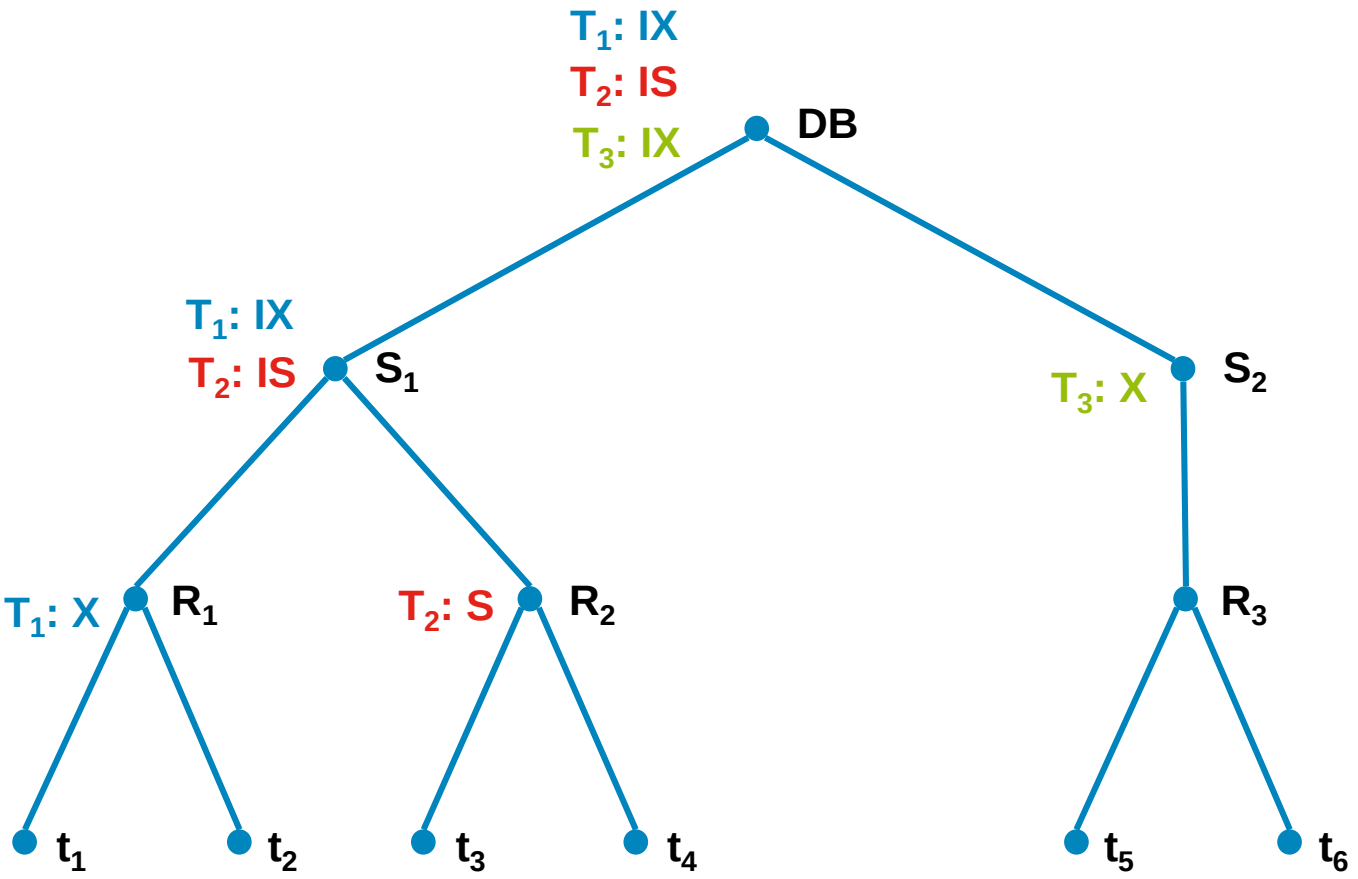
Anwartschaftssperren

- **Nachteil von reinen S- und X-Sperren**
  - Alle Nachfolgeknoten implizit mitgesperrt
  - Alle Vorgängerknoten ebenfalls sperren, um Unverträglichkeiten zu erkennen
    - X-Sperre auf DB: erzwingt Einbenutzerbetrieb
    - S-Sperre auf DB: nur Lese-Transaktionen können parallel laufen
- **Verwendung von Anwartschaftssperren (intention locks)**
  - I-Sperre oder Sperranzeige
    - **IS-Sperre (intention share)**,
      falls auf untergeordnete Objekte nur lesend zugegriffen wird
    - **IX-Sperre (intention exclusive)**,
      falls auf untergeordnete Objekte auch schreibend zugegriffen wird
  - Ersetzt die Sperren für Datenobjekte auf den höheren Hierarchie-Ebenen
  - Nutzung einer Untermenge wird angezeigt, aber in der Untermenge werden noch explizit Sperren gesetzt

Verwendung von Anwartschaftssperren

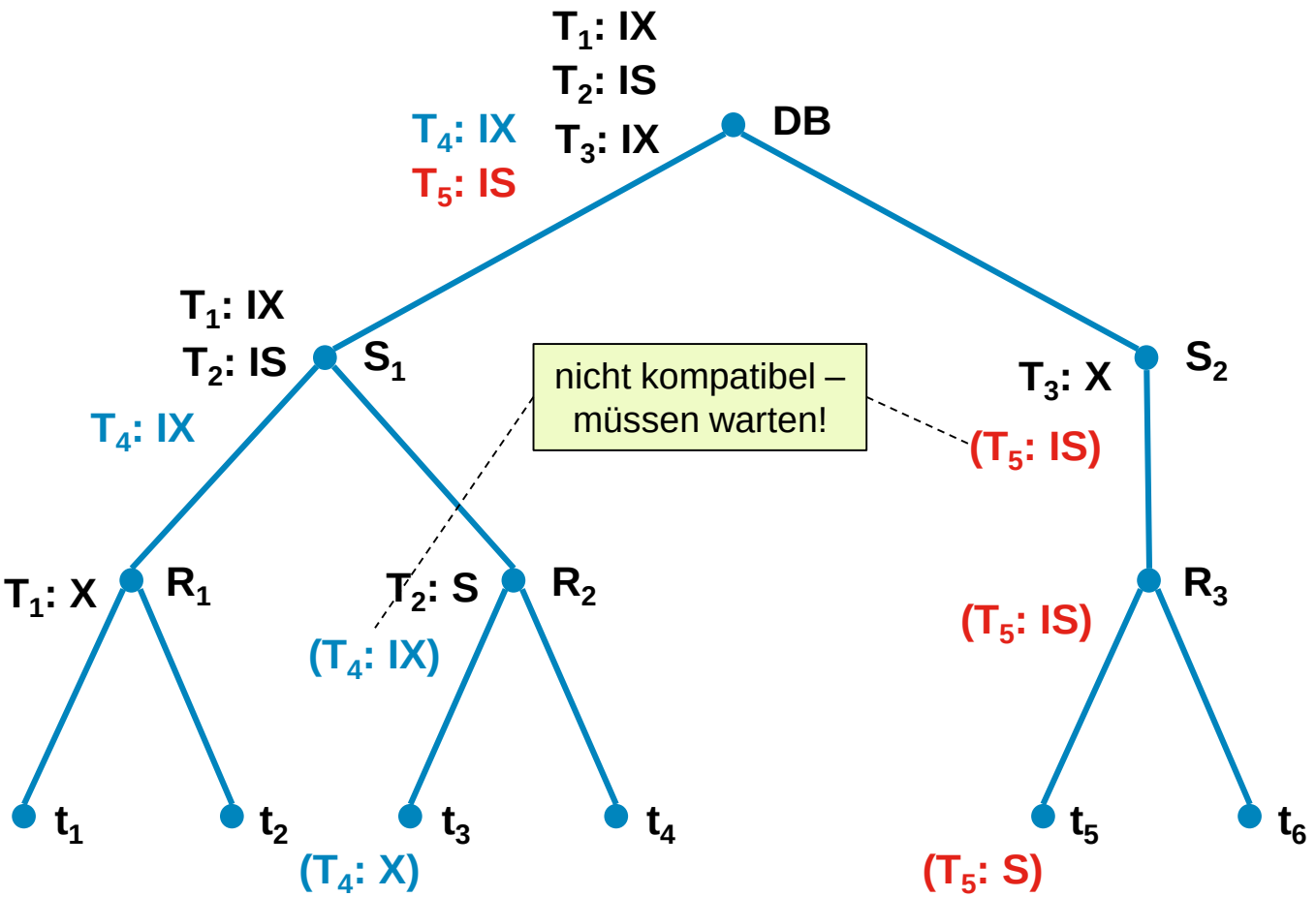
- **Top Down beim Erwerb von Sperren**
  - Bevor ein Knoten mit S oder IS gesperrt werden darf, müssen alle Vorgänger in der Hierarchie im IX- oder im IS-Modus gesperrt worden sein.
  - Bevor ein Knoten mit X oder IX gesperrt werden darf, müssen alle Vorgänger in der Hierarchie im IX-Modus gesperrt worden sein.
- **Bottom Up bei der Freigabe von Sperren**
  - Freigabe der Sperren von unten nach oben
  - Bei keinem Knoten darf die Sperre aufgehoben werden, wenn die betreffende Transaktion noch Nachfolger dieses Knotens gesperrt hat.
- **Beispiel**
  - T_1: benötigt exklusive Sperre von R_1 in S_1
  - T_2: benötigt Lesesperre von R_2 in S_1
  - T_3: benötigt exklusive Sperre von Segment S_2

Beispiel zu Anwartschaftssperren



Beispiel zu Anwartschaftssperren (2)

- T_4: benötigt exklusive Sperre auf Tupel t_3
- T_5: benötigt Lesesperre auf Tupel t_5



Noch eine Anwartschaftssperre

- **Kombination von Sperre und Sperranzeige**

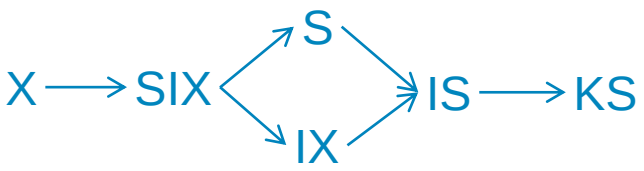
  SIX = S + IX (share and intention exclusive)

  - Sperrt das Objekt in S-Modus
  - Verlangt auf tieferen Hierarchieebenen nur noch IX- oder X-Sperren für zu ändernde Objekte
  - Sinnvoll für den Fall, dass alle Tupel einer Relation gelesen und nur einige davon geändert werden
    - X-Sperre auf Relation zu restriktiv
    - IX-Sperre auf Relation verlangt Sperren jedes Tupels zum Lesen
  - (Die analoge Lösung XIS ist natürlich Unsinn.)

Kompatibilitätsmatrix

	IS	IX	S	SIX	X
IS	+	+	+	+	–
IX	+	+	–	–	–
S	+	–	+	–	–
SIX	+	–	–	–	–
X	–	–	–	–	–

- Darstellung der Sperrmodi in einer Halbordnung (Dominanz-Relation):
  - "$\rightarrow$" bedeutet: Wenn man die linke Sperre hat, kann man mehr machen als mit der rechten.



Weitere Probleme bei der Sperrverwaltung

- **Probleme bei der Implementierung von Sperren**
  - Sperranforderung und -freigabe sollten sehr schnell erfolgen, da sie sehr häufig benötigt werden.
  - Explizite, tupelweise Sperren führen u.U. zu umfangreichen Sperrtabellen und großem Zusatzaufwand.
  - Halten der Sperren bis Transaktionsende führt häufig zu langen Wartezeiten (starke Serialisierung).
  - Häufig berührte Zugriffspfade können zu Engpässen werden.
  - Eigenschaften des Schemas können "hot spots" erzeugen – Datenelemente, auf die fast alle Transaktionen zugreifen müssen.
- **Mögliche Optimierungen**
  - Änderungen auf privaten Objektkopien (verkürzte Dauer exklusiver Sperren)
  - Nutzung mehrerer Objektversionen
  - Spezialisierte Sperren (Nutzung der Semantik von Änderungsoperationen)

Alternative: "Optimistische" Verfahren

- **Erst einmal ohne Sperren einfach zugreifen**
  - Zugriffe aber protokollieren:
    - Menge der gelesenen Datenobjekte
    - Menge der geschriebenen Datenobjekte
- **Dann kurz vor Ende der Transaktion:**
  - Also in der Ausführung des Commit
  - Lese- und Schreibmengen der abschließenden Transaktion mit den Lese- und Schreibmengen der anderen gerade laufenden Transaktionen vergleichen
    - Verschiedene Möglichkeiten des Vergleichs – siehe Literatur
  - Bei Überschneidungen (= Zugriffskonflikten) muss die Transaktion, die gerade abschließen will, zurückgesetzt werden.
    - Deshalb "optimistisch": Ohne Überschneidungen klappt es, und sogar effizienter, weil ohne Sperren (und Wartezeiten).

Entstehung von Verklemmungen (Deadlocks)

- **Beispiel eines elementaren Deadlocks**

```
TA1 hält X-Sperre auf A        TA2 hält X-Sperre auf B
TA1 benötigt B zum Beenden     TA2 benötigt A zum Beenden
```

- **Notwendige Voraussetzungen für einen Deadlock**
  - Gleichzeitiger Zugriff
  - Exklusive Zugriffsanforderungen (X-Sperren)
  - Anfordernde TA besitzt bereits Sperren auf Datenobjekten
  - Keine vorzeitige Freigabe von Sperren auf Datenobjekten (non-preemption)
  - Zyklische Wartebeziehungen zwischen zwei oder mehr Transaktionen

Behandlung von Verklemmungen

- **Lösungsmöglichkeiten**
  - **Timeout**
    - Transaktion nach festgelegter Wartezeit auf eine Sperre zurücksetzen
    - Bestimmung des richtigen Timeout-Werts problematisch
  - **Verhütung** (Prevention)
    - Keine Laufzeitunterstützung zur Deadlock-Behandlung erforderlich
    - Bsp.: Preclaiming (siehe oben, in DBS i. Allg. nicht praktikabel)
  - **Vermeidung** (Avoidance)
    - Potenzielle Deadlocks im voraus (in dem Moment, wo eine TA auf eine andere warten muss) erkennen und durch entsprechende Maßnahmen vermeiden
    - $\Rightarrow$ Laufzeitunterstützung nötig
  - **Erkennung** (Detection)
    - Explizites Führen eines Wartegraphen (wait-for graph) und darin Zyklensuche
    - Auflösung durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter TA (z. B. den Verursacher oder "billigste" TA zurücksetzen)

12.4 Zusammenfassung Synchronisation

- **Serialisierung paralleler Abläufe**
  - Bereitstellung eines logischen Einbenutzerbetriebs
  - Vermeidung von Anomalien
    - Lost updates, dirty reads, inkonsistente Analysen
- **Serialisierbarkeitstheorie**
  - Ein Ablauf ist serialisierbar, wenn er zu einem seriellen Ablauf äquivalent ist.
- **Implementierung: Sperrverfahren**
  - Einfache S- und X-Sperren
  - Zwei-Phasen-Sperrprotokoll
  - Hierarchisches Sperrkonzept
    - Anwartschaftssperren (I-Sperren)
  - Verklemmungen

Literatur

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. In: Michael J. Carey (Hrsg.), Donovan A. Schneider (Hrsg.): *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 22-25, 1995. ACM Press 1995, S. 1-10

Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, Irving L. Traiger: The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* 19(11): 624-633 (1976)