6. Transaktionskonzept: Weiterentwicklungen

- Beschränkungen flacher Transaktionen
- Rücksetzpunkte (Savepoints)
- (Geschlossen) Geschachtelte Transaktionen
  - Konzept
  - Sperrverfahren
  - Freiheitsgrade im Modell
- Offen geschachtelte Transaktionen
  - Transaktionsketten (Sagas)
  - ConTracts
- Lange Entwurfstransaktionen

Beschränkungen flacher Transaktionen: Anwendungsbeispiele

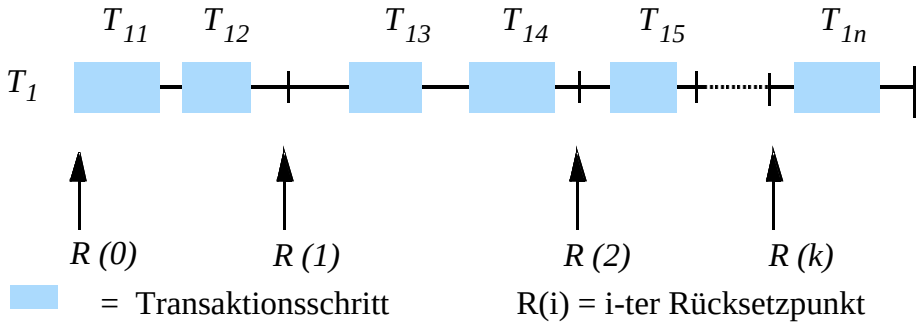
- ACID auf kurze Transaktionen zugeschnitten,
  Probleme mit "lang-lebigen" Aktivitäten (long-lived transactions)
- lange Batch-Vorgänge (Bsp.: Zinsberechnung)
  - Alles-oder-Nichts führt zu hohem Verlust an Arbeit
  - Einsatz vieler unabhängiger Transaktionen verlangt manuelle Recovery-Maßnahmen nach Systemfehler
- Workflows
  - Bsp.: mehrere Reservierungen für Dienstreise
  - lange Sperrdauer führt zu katastrophalem Leistungsverhalten (Sperrkonflikte, Deadlocks)
  - Rücksetzen der gesamten Aktivität im Fehlerfall i.a. nicht akzeptabel
- Entwurfsvorgänge (CAD, CASE, ...)
  - lange Dauer von Entwurfsvorgängen (Wochen/Monate)
  - kontrollierte Kooperation zwischen mehreren Entwerfern
  - Unterstützung von Versionen

Beschränkungen flacher Transaktionen

- Alles-oder-Nichts-Eigenschaft oft inakzeptabel: hoher Arbeitsverlust
- höhere Wahrscheinlichkeit, durch Systemfehler zurückgesetzt zu werden
- Isolation
  - Leistungsprobleme durch "lange" Sperren
  - Sperren vieler Objekte → Erhöhung der Blockierungsrate und Konfliktrate
  - höhere Rücksetzrate (Deadlockhäufigkeit stark abhängig von der Größe der Transaktion)
  - fehlende Unterstützung zur Kooperation
- keine Binnenstruktur
  - fehlende Kapselung und Zerlegbarkeit von Teilabläufen
  - keine abgestufte Kontrolle für Synchronisation und Recovery
  - keine Unterstützung zur Parallelisierung
- fehlende Benutzerkontrolle

Partielles Zurücksetzen von Transaktionen

- Voraussetzung: private Rücksetzpunkte (Savepoints) innerhalb einer Transaktion



- Operationen: SAVEPOINT R(i)
  ROLLBACK TO SAVEPOINT R(j)
- Protokollierung aller Änderungen, Sperren, Cursor-Positionen etc. notwendig
- Partielle UNDO-Operation bis R(i) in LIFO-Reihenfolge
- Problem: Savepoints werden vom Laufzeitsystem der Programmiersprachen nicht unterstützt

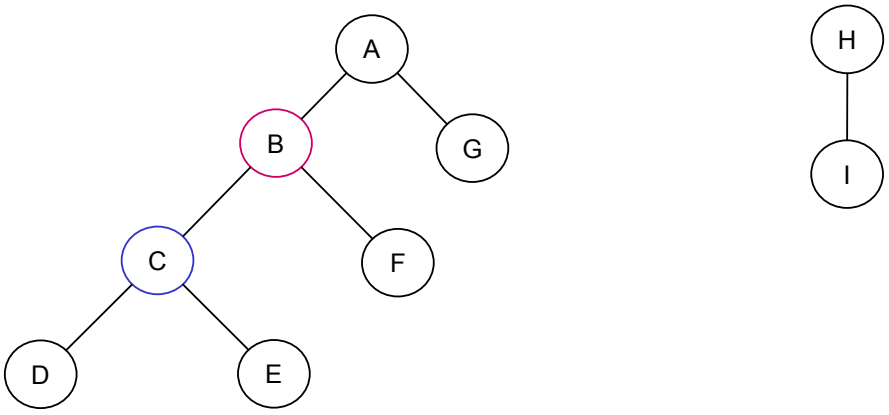
Savepoints in SQL:1999

- SQL-Transaktionsanweisungen
  - START TRANSACTION [READ { ONLY | WRITE }]
    [ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }]
  - SET TRANSACTION [READ { ONLY | WRITE }] [ISOLATION LEVEL { ... }]
  - SET CONSTRAINTS { ALL | <Liste von Int.beding.>} {IMMEDIATE | DEFERRED}
  - COMMIT [WORK] [AND [NO] CHAIN]
  - SAVEPOINT <Rücksetzpunktname>
  - RELEASE SAVEPOINT <Rücksetzpunktname>
  - ROLLBACK [WORK] [AND [NO] CHAIN] [TO SAVEPOINT <Rücksetzpunktname>]
- Beispiel

```
INSERT INTO Pers (PNR, Name, Gehalt) VALUES (1234, 'Schulz', 40000);
INSERT INTO Pers (PNR, Name, Gehalt) VALUES (1235, 'Schneider', 38000);
SELECT SUM (Gehalt) INTO Summe FROM Pers;
IF Summe > 1000000 THEN ROLLBACK; ELSE SAVEPOINT R1;
INSERT INTO Pers (PNR, Name, Gehalt) VALUES (1300, 'Weber', 39000);   ...
IF ... THEN ROLLBACK TO SAVEPOINT R1;
```

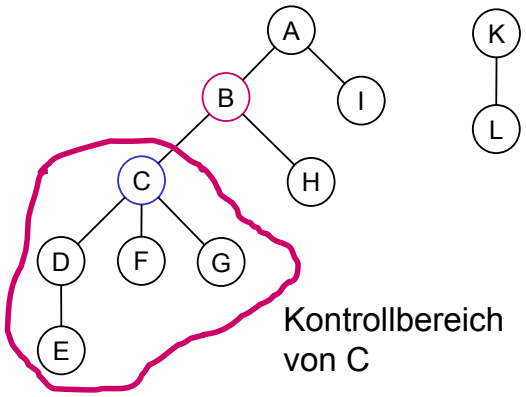

Geschachtelte Transaktionen (nested transactions)

- Zerlegung einer Transaktion in eine Hierarchie von Sub-Transaktionen
  - Zerlegung erfolgt anwendungsbezogen, z.B. gemäß Modularisierung von Anwendungsfunktionen
  - Transaktionsbaum verdeutlicht statische Aufrufhierarchie



- ausgezeichnete Transaktion = Top-Level Transaction (TL)
  - Bewahrung der ACID-Eigenschaften für TL-Transaktion
- Welche Eigenschaften gelten für Sub-Transaktionen?

Transaktionseigenschaften



- *Commit-Regel*:
  - Das (lokale) Commit einer Sub-Transaktion macht ihre Ergebnisse nur der Vater-Transaktion zugänglich. Das endgültige Commit der Sub-Transaktion erfolgt dann und nur dann, wenn für alle Vorfahren bis zur TL-Transaktion das endgültige Commit erfolgreich verläuft.
- *Rücksetzregel*:
  - Wenn eine (Sub-) Transaktion auf irgendeiner Schachtelungsebene zurückgesetzt wird, werden alle ihre Sub-Transaktionen, unabhängig von ihrem lokalen Commit-Status ebenso zurückgesetzt. Diese Regel wird rekursiv angewendet.
- *Sichtbarkeits-Regel*:
  - Änderungen einer Sub-Transaktion werden bei ihrem Commit für die Vater-Transaktion sichtbar. Objekte, die eine Vater-Transaktion hält, können Sub-Transaktionen zugänglich gemacht werden. Änderungen einer Sub-Transaktion sind für Geschwister-Transaktionen nicht sichtbar.

Eigenschaften von Sub-Transaktionen

- **A**: erforderlich wegen Zerlegbarkeit, isoliertes Rücksetzen, usw.
- **C**: zu strikt; Vater-Transaktion (spätestens TL-Transaktion) kann Konsistenz wiederherstellen
- **I**: erforderlich wegen isolierter Rücksetzbarkeit usw.
- **D**: nicht möglich, da Rücksetzen eines äußeren Kontrollbereichs das Rücksetzen aller inneren impliziert

Geschachtelte Transaktionen: Sperrverfahren

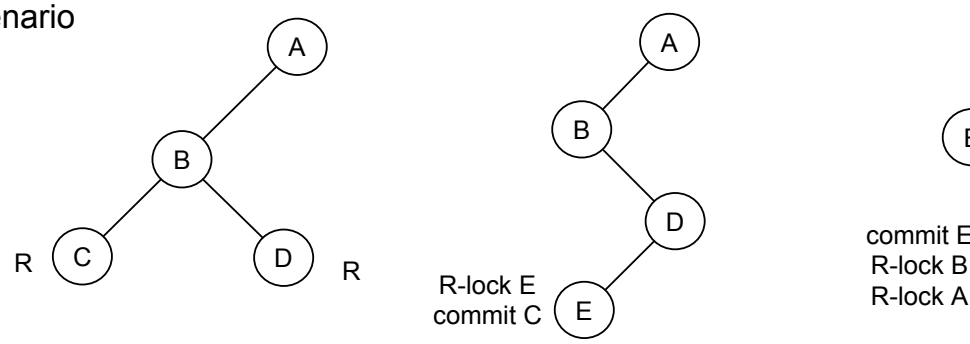
- Sperren bei flachen Transaktionen:
  - Erwerb gemäß Kompatibilitätsmatrix (z.B. Halten von R- und X-Sperren)
  - Freigabe bei Commit
- Unterscheidung zwischen gehaltenen (X- und R-) Sperren und von Sub-Transaktionen geerbten Platzhalter-Sperren (*retained locks*) r-X und r-R
  - *r-X*: nur Nachfahren im Transaktionsbaum (und Transaktion selbst) können Sperren erwerben
  - *r-R*: keine X-Sperre für Vorfahren im Transaktionsbaum sowie andere (unabhängige) Transaktionen

Regeln zum Sperren geschachtelter Transaktionen

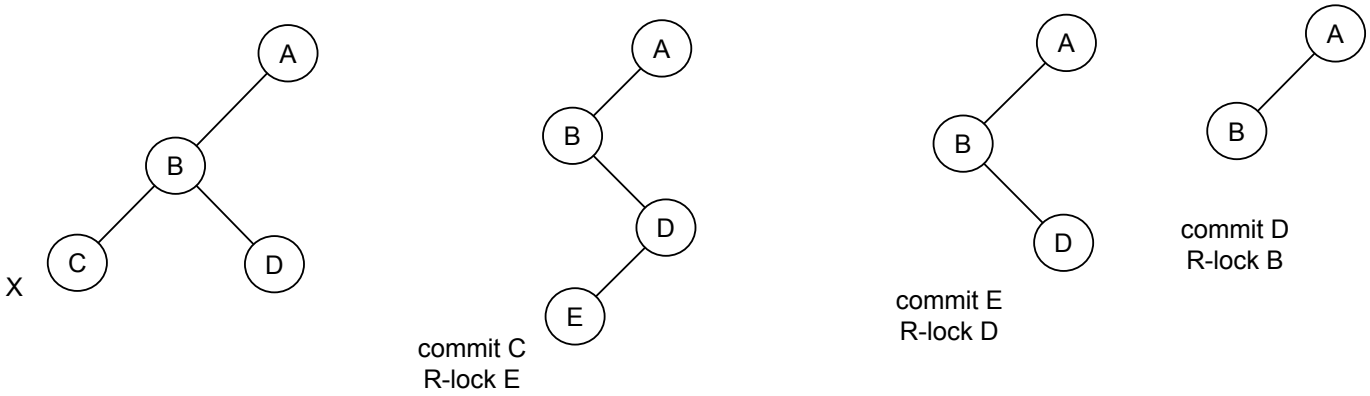
- **R1:** Transaktion T kann X-Sperre erwerben falls
  - keine andere Transaktion eine X- oder R-Sperre auf dem Objekt hält, sowie
  - alle Transaktionen, welche eine r-X oder r-R-Sperre besitzen, Vorfahren von T sind (bzw. T selbst)
- **R2:** Transaktion T kann R-Sperre erwerben falls
  - keine andere Transaktion eine X-Sperre hält, sowie
  - alle Transaktionen, welche eine r-X besitzen, Vorfahren von T sind (bzw. T selbst)
- **R3:** Beim Commit von Sub-Transaktion T erbt Vater von T alle Sperren von T (reguläre + retained-Sperren). Für reguläre Sperren von T werden beim Vater die entsprechenden retained-Sperren gesetzt
- **R4:** Beim Abbruch einer Transaktion T werden alle regulären und Platzhalter-Sperren von T freigegeben. Sperren der Vorfahren bleiben davon unberührt.

Geschachtelte Transaktionen: Sperrverfahren (2)

a) Lese-Szenario

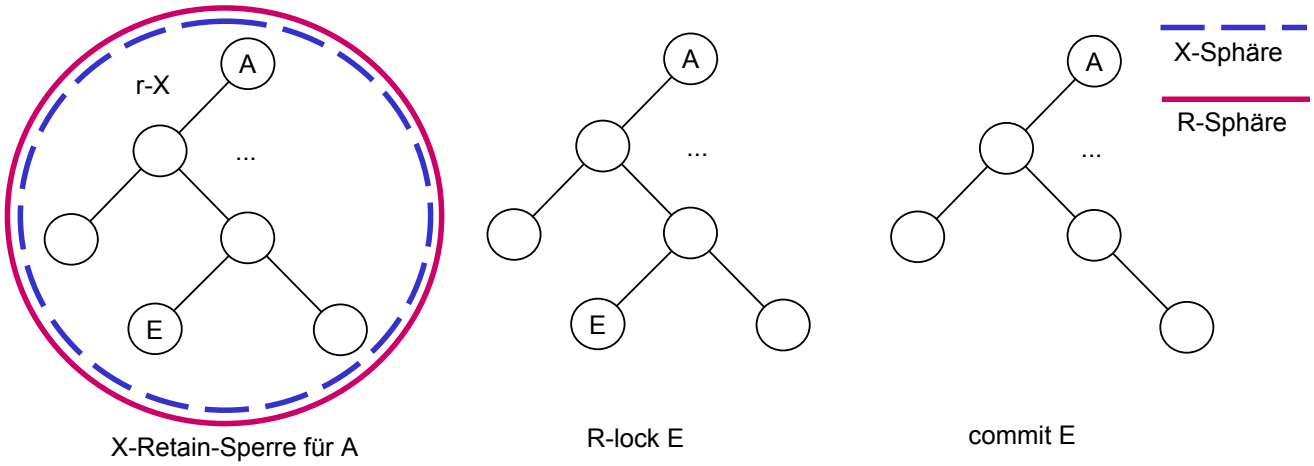


b) Änderungs-Szenario

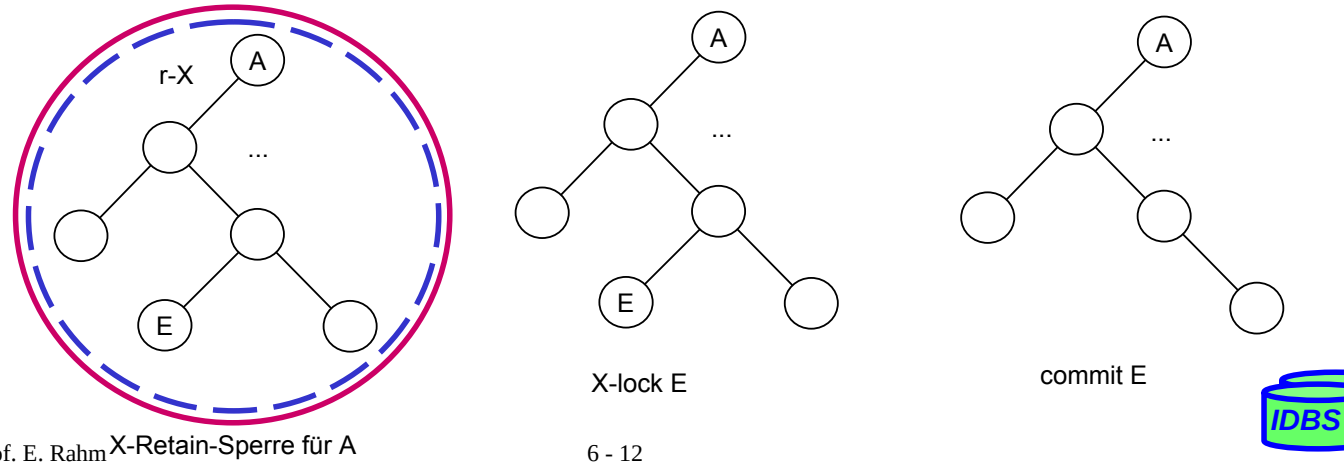


Geschachtelte Transaktionen: Sperrverfahren (3)

a)

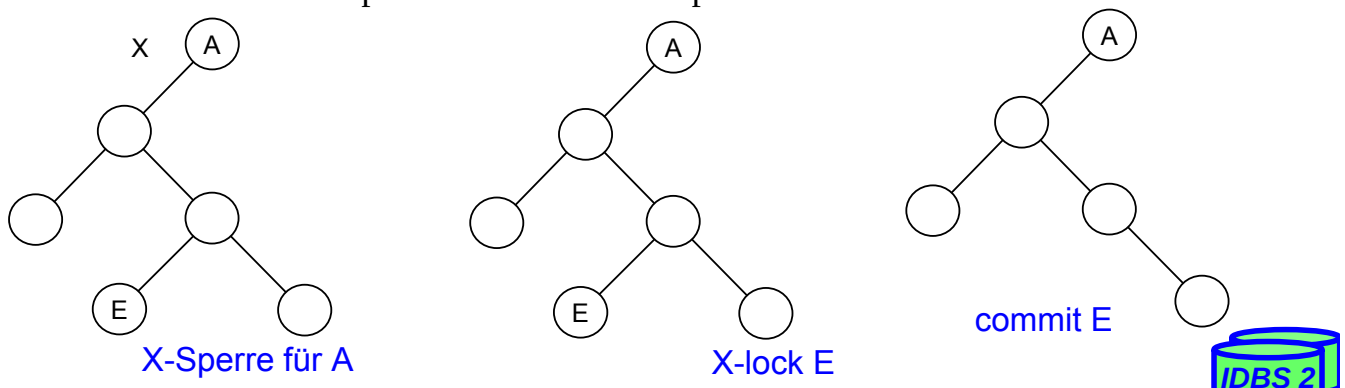


b)



Geschachtelte Transaktionen: Sperrverfahren (4)

- Beschränkungen des vorgestellten Sperrverfahrens
  - Sub-Transaktionen können keine Objekte lesen oder ändern, die von einem Vorfahren geändert wurden
  - Sub-Transaktionen können keine Objekte ändern, die von einem Vorfahren gelesen wurden
- Abhilfe: Unterstützung von Aufwärts- und Abwärts-Vererbung von Sperren
  - bei Sperrkonflikt zwischen Sub-Transaktion und Vorfahr kann Vorfahr Sperre an Sub-Transaktion vererben (downward inheritance)
  - Vorfahr reduziert seine Sperre auf Platzhalter-Sperre



Merkmale geschlossen geschachtelter Transaktionen

- Vorteile
  - explizite Kontrollstruktur innerhalb von Transaktionen
  - Unterstützung von Intra-Transaktionsparallelität
  - Unterstützung verteilter Systemimplementierung
  - feinere Recovery-Kontrolle innerhalb einer Transaktion
  - Modularität des Gesamtsystems
  - einfachere Programmierung paralleler Abläufe
- ACID für Wurzel-Transaktionen lässt Hauptprobleme flacher Transaktionen ungelöst
  - Atomarität gegenüber Systemfehlern
  - Isolation zwischen Transaktionen

Offen geschachtelte Transaktionen (open nested transactions)

- Freigabe von Ressourcen (Sperren) bereits am Ende von Sub-Transaktionen - vor Abschluss der Gesamttransaktion
  - Ziel: Lösung des Isolationsproblems langlebiger Transaktionen
  - verbesserte Inter-Trans.parallelität (neben Intra-Transaktionsparallelität)
  - Probleme hinsichtlich Synchronisation sowie Recovery
- Synchronisationsprobleme
  - Sichtbarwerden "schmutziger" Änderungen verletzt i.a. Serialisierbarkeit
  - dennoch werden oft mit der Realität verträgliche Abläufe erreicht
  - ggf. Einsatz semantischer Synchronisationsverfahren

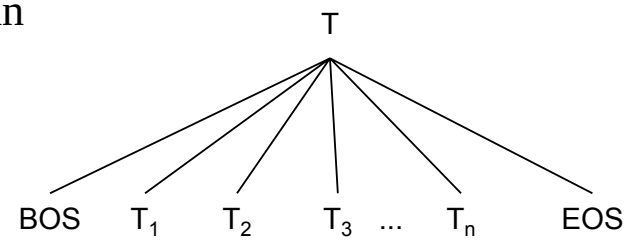
Offene Schachtelung (2)

- vorzeitige Freigabe von Änderungen erfordert kompensationsbasierte Undo-Recovery
  - zustandsorientierte Undo-Recovery nicht möglich -> logische Kompensation
  - Kompensationen sind auch in der Realität verbreitet (Stornierung, Terminabsage, ...)
- Probleme kompensationsbasierter Recovery
  - Korrektheit der Kompensationsprogramme
  - Kompensationen dürfen nicht scheitern
  - nicht alle Operationen sind kompensierbar (z.B. "real actions" mit irreversiblen Auswirkungen)

Das Konzept der Sagas

- Saga ≡ langlebige „Transaktion“, die in eine Sammlung von Sub-Transaktionen aufgeteilt werden kann

spezielle Art von zweistufigen, offen geschachtelten Transaktionen



- T_i geben Ressourcen vorzeitig frei
  - Verzahnung mit T_j anderer Transaktionen (Sagas)
  - keine Serialisierbarkeit der Gesamt-Transaktion (Saga)
- Rücksetzen von Sub-Transaktionen durch Kompensation
  - alle T_i gehören zusammen; keine teilweise Ausführung von T
  - Bereitstellung von Kompensationstransaktionen C_i für jede T_i

Sagas (2)

- Zusicherung des DBS
  1. $T_1, T_2, T_3, ..., T_n$ oder
  2. $T_1, T_2, ..., T_j, C_j, ..., C_2, C_1$ für irgendein $0 \leq j < n$
- Fehlerfall (Backward Recovery)
  - DBS garantiert LIFO-Ausführung der Kompensationen
  - Kompensationen dürfen nicht scheitern
- Backward-Revovery vielfach unerwünscht, v.a. nach Systemfehler
- Unterstützung von Forward-Recovery durch (persistente) Savepoints
- Partielles Rücksetzen möglich: Kombination von Backward- und Forward-Recovery

Sagas (3)

- Szenario:
  - Savepoint nach T_2
  - Crash nach T_4

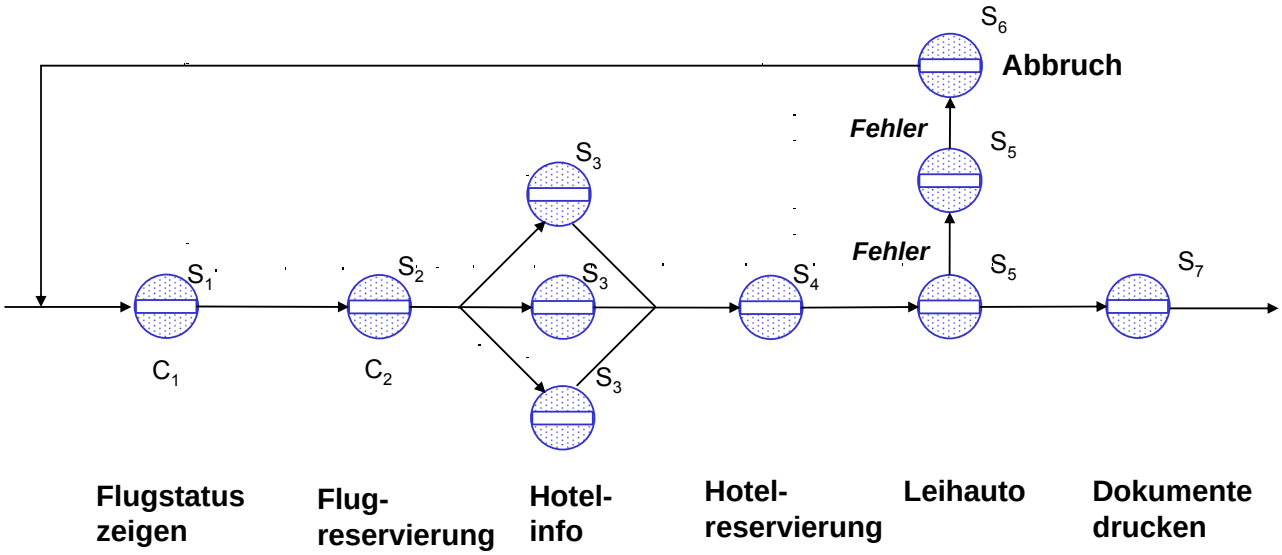
  Ablauf: BS, T_1, T_2, SP, T_3, T_4, ↯ C_4, C_3, T_3, T_4, T_5, T_6, ES

- Zusammenfassung der Eigenschaften:
  - A+I für jede Sub-Transaktionen T_i
  - A+C+D für umfassende „Transaktion“ T

ConTracts

- ConTract-Modell: Mechanismus zur kontrollierten und zuverlässigen Ausführung langlebiger Aktivitäten
- zweistufiges Programmiermodell: Trennung Anwendungsentwicklung von Beschreibung der Ablaufstruktur
  - Skript: Beschreibung der Ablaufstruktur / Kontroll- und Datenfluss (Workflow-Definition)
  - Steps: Programmierung der elementaren Verarbeitungsschritte der Anwendung + Kompensationsaktion
  - Step ist sequentielles Programm, z.B. ACID-Transaktion
- Zentrale Konsistenzeigenschaft: ein ConTract terminiert in endlicher Zeit und in einem korrekten Endzustand
  - auch bei Systemfehler Fortsetzung der Verarbeitung “nach vorne” oder
  - kontrollierte Zurückführung eines ConTracts auf seinen Anfangszustand

Beispiel-Workflow

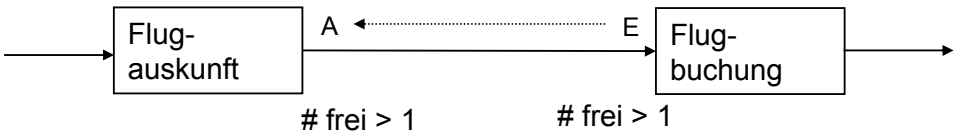


ConTracts (Forts.)

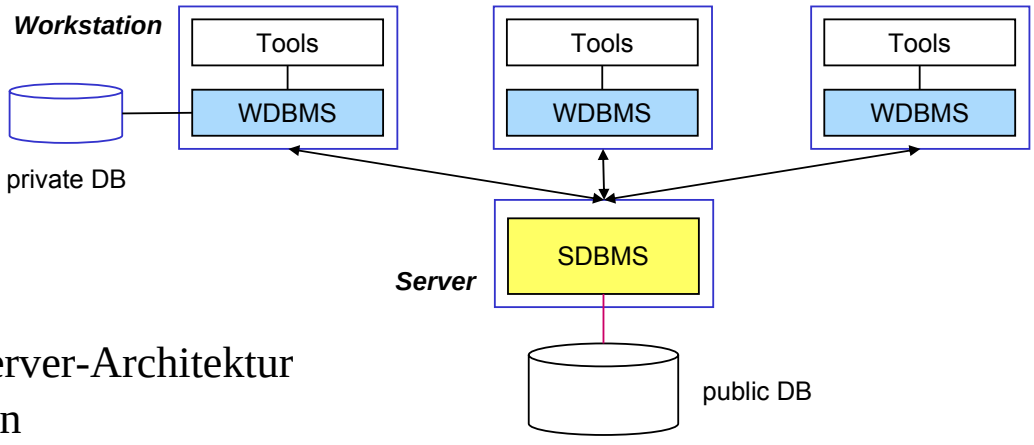
- Erweiterungen gegenüber Saga-Ansatz
  - reichere Kontrollstrukturen (Sequenz, Verzweigung, Parallelität, Schleife, etc.)
  - getrennte Beschreibung von **Steps** und Ablaufkontrolle (**Skript**)
  - Verwaltung eines persistenten **Kontextes** für globale Variablen, Zwischenergebnisse, Bildschirmausgaben, etc.
  - Synchronisation zwischen Steps über Invarianten
  - flexible Konflikt-/Fehlerbehandlung
- Transaktionsübergreifende Kontrolle der Vearbeitung
  - Synchronisation
  - Recovery
  - Kontext

Synchronisation von Contracts

- Synchronisation mit Invarianten: semantische Synchronisationsbedingungen für korrekte Step-Ausführung
  - Wenig Behinderungen: hohe Parallelität
  - Ausschluss von Konsistenzverletzungen trotz frühzeitiger Sperrfreigabe
- Invarianten steuern die Überlappung parallel ablaufender ConTracts bzw. Steps über Prädikate (keine Serialisierbarkeit)
  - Ausgangs-Invarianten charakterisieren den am Ende eines Steps erreichten Zustand der bearbeiteten Objekte
  - Folge-Step kann mit seiner Eingangs-Invarianten überprüfen, ob die Bedingung für seine korrekte Synchronisation noch erfüllt ist
  - Realisierung mit Check/Revalidate-Ansatz
- Real Actions können nicht über Invarianten synchronisiert werden

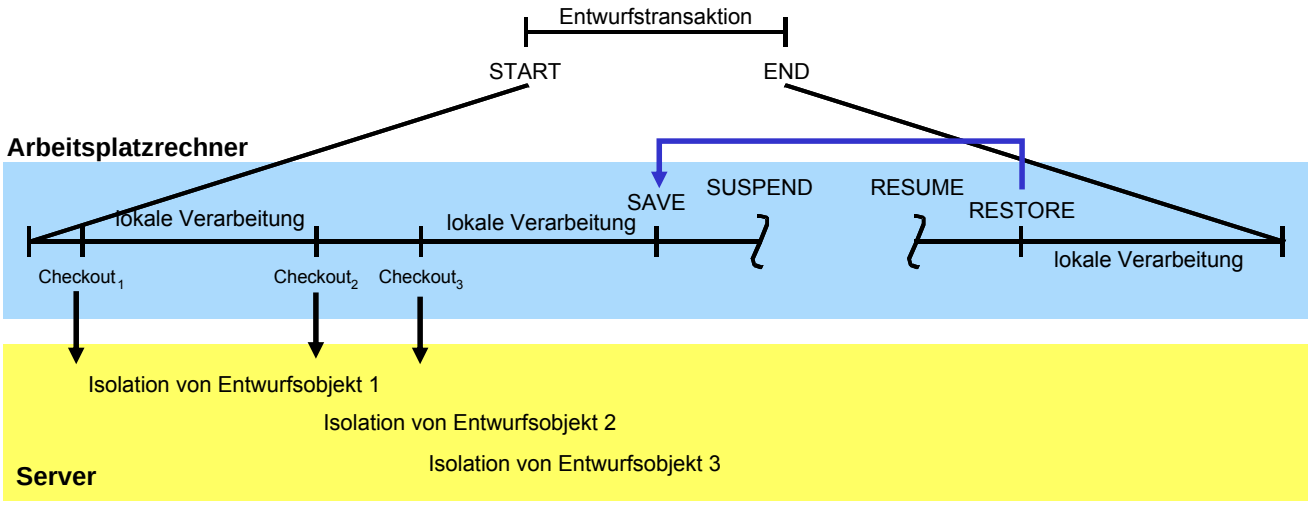


DB-Verarbeitung in Entwurfsumgebungen



- Merkmale
  - Workstation/Server-Architektur
  - lange Dauer von Entwurfsvorgängen (Wochen/Monate)
  - Benutzerkontrolle (nicht-deterministischer Ablauf)
  - kontrollierte Kooperation zwischen mehreren Entwerfern
  - Unterstützung von Versionen
- Lösungsansätze:
  - *Checkout/Checkin-Modell*
  - transaktionsinterne Savepoints
  - vorzeitiger Austausch von Änderungen zwischen Designern

Entwurfstransaktion bei Workstation/ Server-Kooperation



- Charakteristika: 0 .. n Checkout-, 0 .. 1 Checkin-Vorgänge, lange Dauer
- Speicherung von Zwischenzuständen einer Entwurfstransaktion zum:
  - Unterbrechen der Verarbeitung (SUSPEND, RESUME)
  - Rücksetzten auf frühere Verarbeitungszustände (SAVE, RESTORE)

Zusammenfassung

- ACID verbreitet und bewährt, hat jedoch Beschränkungen
- Geschlossen geschachtelte Transaktionen
  - Unterstützung von Intra-Transaktionsparallelität
  - feinere Rücksetzeinheiten
  - v.a. in verteilten Systemen wichtig
- Offen geschachtelte Transaktionen (z.B. Sagas)
  - Unterstützung langlebiger Transaktionen
  - Reduzierung der Konfliktgefahr durch vorzeitige Sperrfreigabe (=> erhöhte Inter-Transaktions-Parallelität)
  - Backward-Recovery durch Kompensation
  - Forward-Recovery erforderlich
- Unterstützung langer Entwurfstransaktionen
  - zugeschnittene Verarbeitungsmodelle (Checkout/Checkin)
  - Kooperation innerhalb von Transaktionen
  - Unterstützung von Versionen und Savepoints

Übungsfragen

- Welche der ACID-Eigenschaften gelten nicht mehr für
  - Transaktionen mit Savepoints
  - geschlossen geschachtelte Transaktionen bzw. deren Subtransaktionen
  - Sagas
  - Entwurfsvorgänge mit Checkout/Checkin-Verarbeitung
- Welche Unterschiede bestehen zwischen offen und geschlossen geschachtelten Transaktionen?
- Welche Probleme bestehen bezüglich Kompensationen?