

# 6. Transaktionskonzept: Weiterentwicklungen

- Beschränkungen flacher Transaktionen
- Rücksetzpunkte (Savepoints)
- (geschlossen) geschachtelte Transaktionen
  - Konzept
  - Sperrverfahren
- offen geschachtelte Transaktionen
  - Transaktionsketten (Sagas)
- Transaktionen in Workflows / Geschäftsprozessen
  - ConTracts
  - Webservice Transaktionen (Biztalk / WS-Transactions)
- 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
  - Zerlegung in mehrere Transaktionen verlangt manuelle Recovery-Maßnahmen nach Systemfehler
- Workflows (Bsp.: mehrere Buchungen für Dienstreise)
  - lange Sperrdauer würde zu katastrophalem Leistungsverhalten (Sperrkonflikte, Deadlocks) führen
  - Rücksetzen der gesamten Aktivität im Fehlerfall i.a. nicht akzeptabel
- Entwurfsvorgänge (CAD, CASE, Dokumentenerstellung...)
  - lange Dauer von Entwurfsvorgängen (Wochen/Monate)
  - kontrollierte Kooperation zwischen mehreren Entwerfern
  - Unterstützung von Versionen



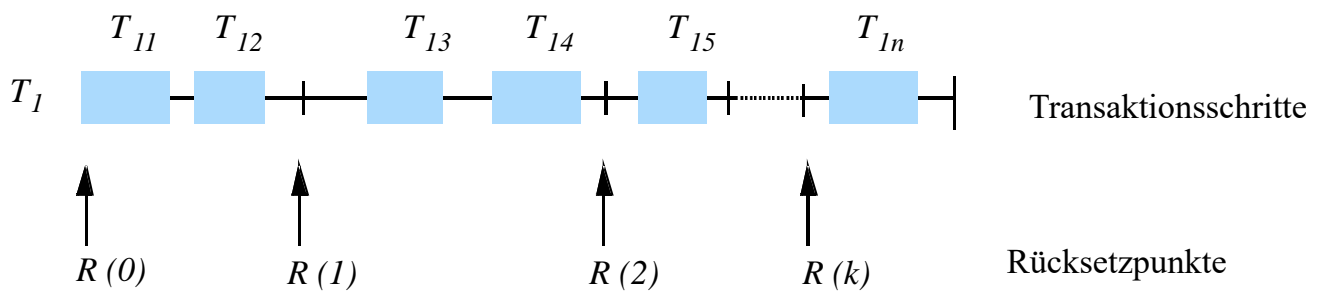
# Beschränkungen flacher Transaktionen

- 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 (Deadlock-Häufigkeit 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: **Rücksetzpunkte (Savepoints)** innerhalb 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
  - nur für Transaktionsfehler, aber nicht für Systemfehler nutzbar
- **Anmerkung: Savepoints sind keine Checkpoints**



# 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 > 1.000.000 THEN ROLLBACK; ELSE SAVEPOINT R1;
INSERT INTO Pers (PNR, Name, Gehalt) VALUES (1300, 'Weber', 39000); ...
IF ... THEN ROLLBACK TO SAVEPOINT R1;
```



# Savepoints in JDBC

## ■ seit JDBC 3.0

## ■ Methoden (Klasse Connection)

- setSavepoint (savepointName)
- releaseSavepoint (savepoint)
- rollback (savepoint)

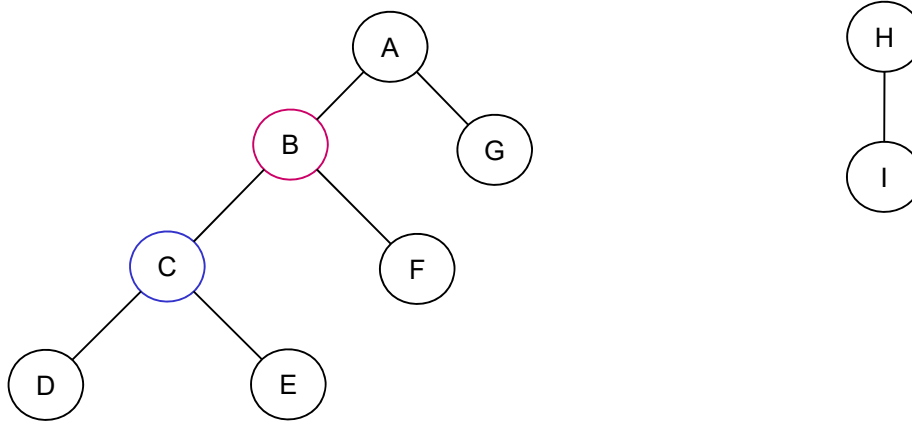
## ■ Beispiel

```
try { // Verbindungsobjekt conn sei gegeben
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    String SQL = "INSERT INTO Pers VALUES (106, 20, 'Rita', 'Tez)";
    stmt.executeUpdate(SQL);
    Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
    String SQL = "INSERTED IN Pers VALUES (107, 22, 'Sita', 'Tez)"; //fehlerhaftes SQL
    stmt.executeUpdate(SQL);
    conn.commit();
} catch(SQLException se) {
    conn.rollback (savepoint1);
}
```



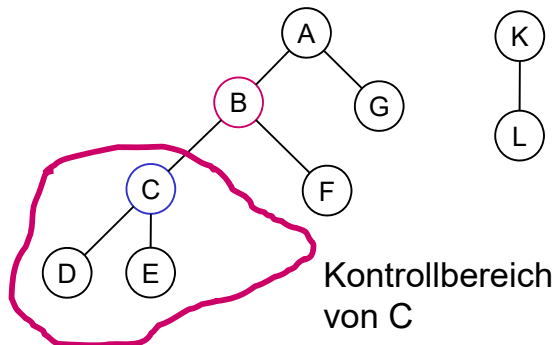
# Geschachtelte Transaktionen (nested transactions)

- Zerlegung einer Transaktion in 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?

## Eigenschaften geschachtelter Transaktionen



- **Commit-Regel:**
  - Das (lokale) Commit einer Sub-Transaktion macht ihre Ergebnisse nur der Vorgänger-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.
  - Vorgänger-Transaktionen entscheidet bei Fehler einer Sub-Transaktion über Fortgang (Abbruch, Retry, Ignore, Exception Handling)
- **Sichtbarkeits-Regel:**
  - Änderungen einer Sub-Transaktion werden bei ihrem Commit für die Vorgänger-Transaktion sichtbar. Objekte, die eine Vorgänger-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; Vorgänger-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



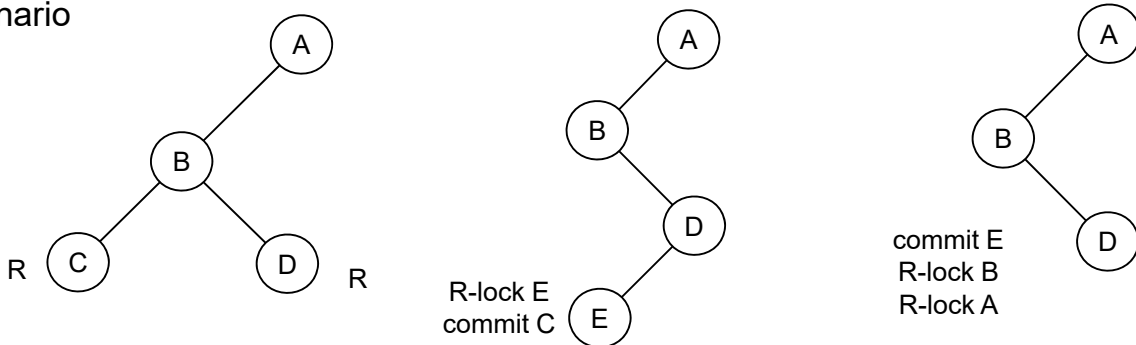
# Regeln zum Sperren geschachtelter Transaktionen

- **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: Commit**
  - beim Commit von Sub-Transaktion T erbt Vorgänger von T alle Sperren von T (reguläre + retained-Sperren)
  - für reguläre Sperren von T werden beim Vorgänger die entsprechenden retained-Sperren gesetzt
- **R4: Rollback**
  - beim Abbruch einer Transaktion T werden alle regulären und Platzhalter-Sperren von T freigegeben.
  - Sperren der Vorfahren bleiben davon unberührt

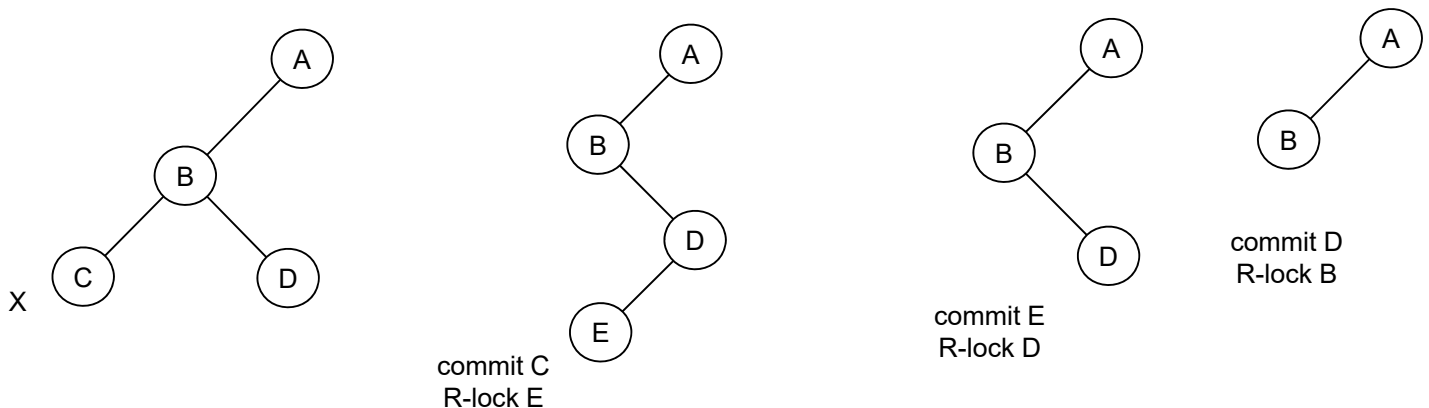


## Geschachtelte Transaktionen: Sperrverfahren (2)

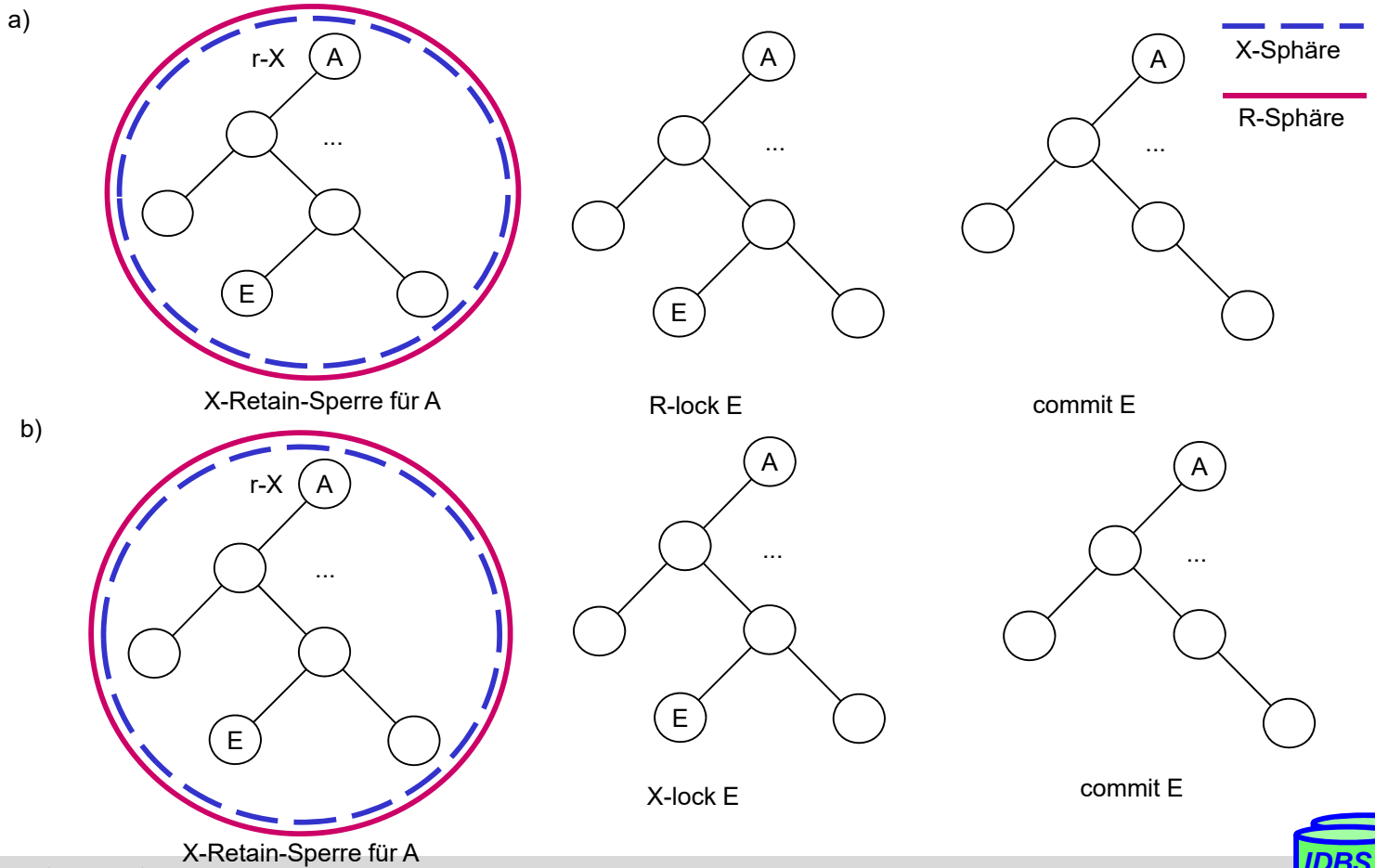
### a) Lese-Szenario



### b) Änderungs-Szenario



# Geschachtelte Transaktionen: Sperrverfahren (3)



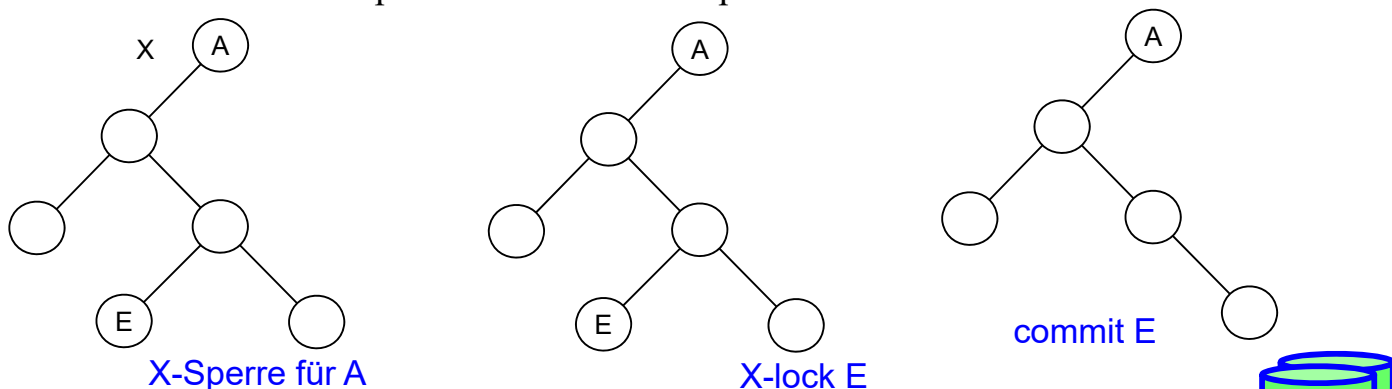
# Geschachtelte Transaktionen: Sperrverfahren (4)

## ■ Beschränkungen des vorgestellten Sperrverfahrens

- Sub-Transaktionen können keine Objekte lesen oder ändern, die von einem Vorfahren geändert werden
- Sub-Transaktionen können keine Objekte ändern, die von einem Vorfahren gelesen werden

## ■ 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

## Geschachtelte Transaktionen in SQL-Server

### Nesting Transactions



SQL Server 2008 R2 | [Other Versions](#) ▾

Explicit transactions can be nested. This is primarily intended to support transactions in stored procedures that can be called either from a process already in a transaction or from processes that have no active transaction.

## ■ Commit/Rollback äußerer Transaktionen wird auf innere Transaktionen propagiert

- Commit innerer Transaktionen wird ignoriert
- *Commit Transaction/ Commit Work* bezieht sich auf letztes *Begin Transaction*

## ■ Rollback in innerer Transaktion führt zum Abbruch der gesamten geschachtelten Transaktion



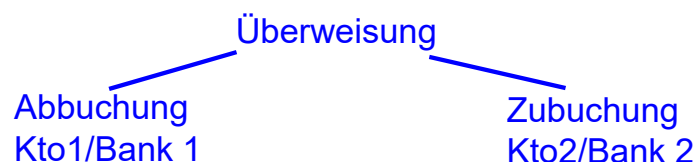
# Beispiel SQL-Server

```
--
CREATE TABLE TestTrans(Cola INT PRIMARY KEY,
                        Colb CHAR(3) NOT NULL);
GO
CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3) AS
BEGIN TRANSACTION InProc
INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol)
COMMIT TRANSACTION InProc;
GO
/* Start a transaction and execute TransProc. */
BEGIN TRANSACTION OutOfProc;
GO
EXEC TransProc 1, 'aaa';
GO
/* Roll back the outer transaction, this will
   roll back TransProc's nested transaction. */
ROLLBACK TRANSACTION OutOfProc;
GO
EXECUTE TransProc 3, 'bbb';
GO
/* The following SELECT statement shows only rows 3 and 4 ar
   still in the table. This indicates that the commit
   of the inner transaction from the first EXECUTE statement
   TransProc was overridden by the subsequent rollback. */
SELECT * FROM TestTrans;
GO
```



## Offen geschachtelte Transaktionen (open nested transactions)

- Freigabe von Ressourcen (Sperrern) bereits am Ende von Sub-Transaktionen - vor Abschluss der Gesamttransaktion
  - Ziel: Lösung des Isolationsproblems langlebiger Transaktionen
  - verbesserte Inter-Transaktionsparallelität (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



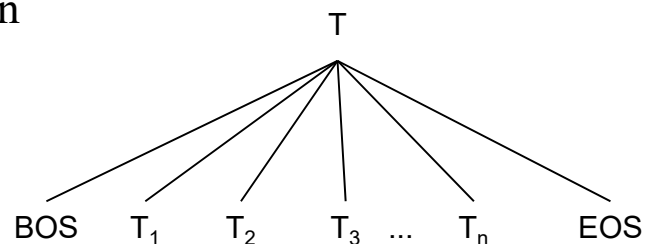
## Offene Schachtelung (2)

- vorzeitige Freigabe von Änderungen erfordert **kompensationsbasierte Undo-Recovery**
  - zustandsorientierte Undo-Recovery nicht möglich -> logische Kompensation („semantisches“ Zurücksetzen)
  - 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  $\equiv$  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)

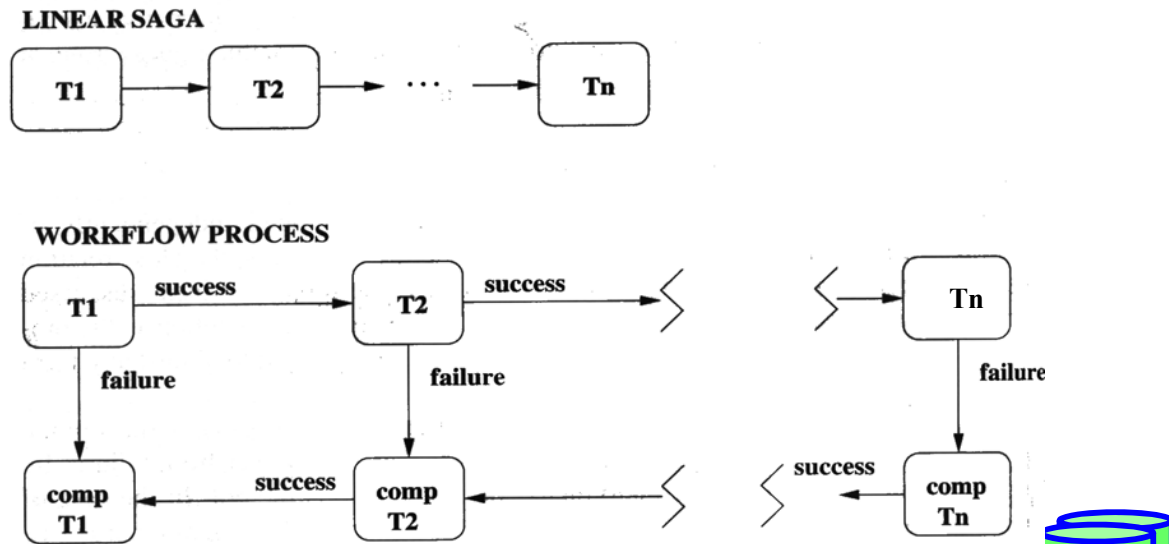
### ■ Zusicherung des DBS

1.  $T_1, T_2, T_3, \dots, T_n$  oder
2.  $T_1, T_2, \dots, T_j, C_j, \dots, C_2, C_1$  für irgendein  $0 \leq j < n$

### ■ Backward Recovery: LIFO-Ausführung der Kompensationen

- Kompensationen dürfen nicht scheitern

Saga-Abbildung  
in Workflow:



## Sagas (3)

### ■ Backward-Recovery vielfach unerwünscht, v.a. nach Systemfehler

- Unterstützung von **Forward-Recovery** durch (persistente) Savepoints
- Partielles Rücksetzen möglich: Kombination von Backward- und Forward-Recovery

### ■ Szenario:

- Savepoint nach  $T_2$
- Crash nach  $T_4$

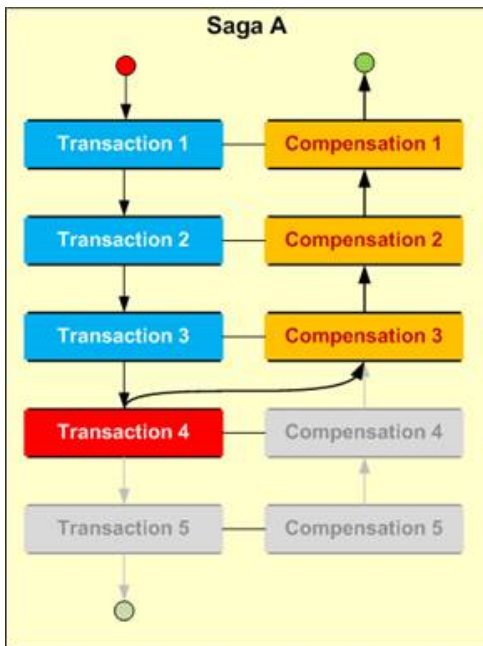
Ablauf: BS,  $T_1, T_2, SP, T_3, T_4$ ,



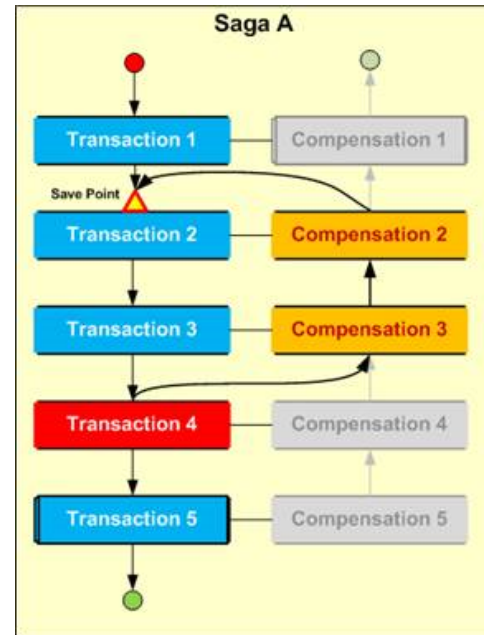
### ■ Zusammenfassung der Eigenschaften:

- A+I für jede Sub-Transaktionen  $T_i$
- A+C+D für Saga (umfassende „Transaktion“ T)

# Sagas: Backward vs. Forward Recovery



<http://geekswithblogs.net/cyoung/articles/100424.aspx>



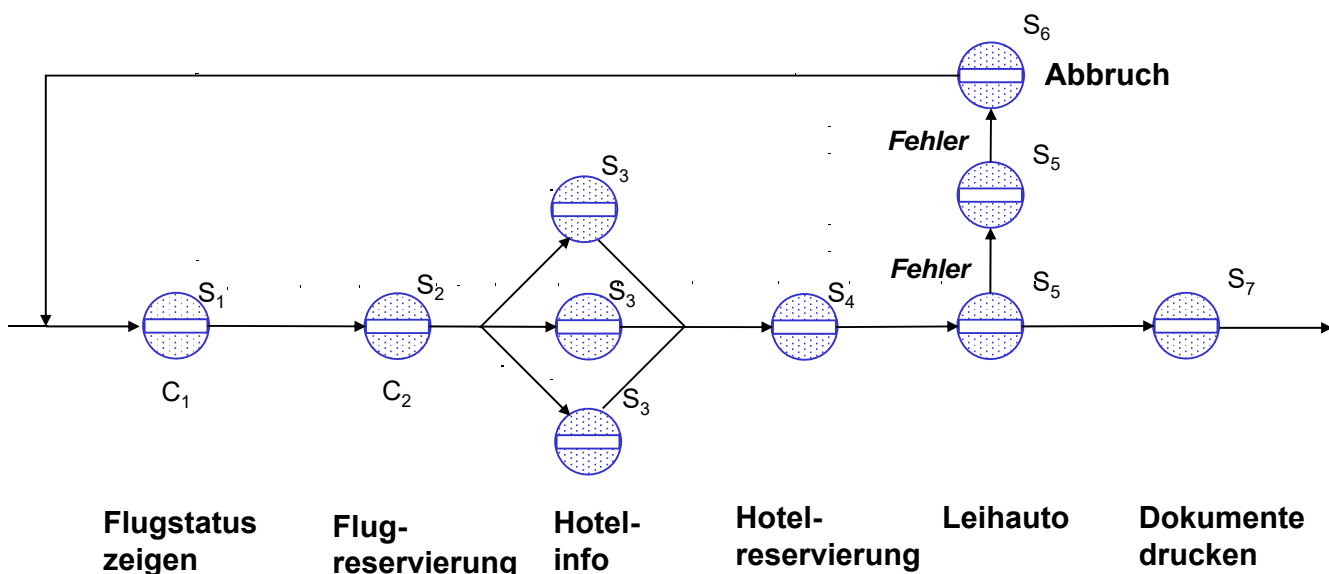
## Komplexere kompensationsbasierte Ansätze

- Saga-Modell zu einfach für reale Workflows / Business-Prozesse
  - komplexe Interaktionen zwischen Teilsystemen (unterschiedliche Datenhaltungssysteme / Organisationen)
  - komplexe Ablauflogik (nicht nur Sequenz)
  - Notwendigkeit Kontext zwischen Schritten persistent zu halten
  - Notwendigkeit einer anwendungsbezogenen Fehler- bzw. Ausnahmebehandlung
- einfachere Erweiterungen, z.B. Contracts
- Transaktionsunterstützung für Business-Prozesse / BPEL
  - Nutzung von Web Services, service-orientierten Architekturen
  - Business Process Execution Language (WS-BPEL)
  - Transaktionsspezifikationen: WS-Transactions
  - BPEL-Unterstützung in zahlreichen service-basierten Applikationsumgebungen: MS Biztalk, IBM Websphere, SAP Netweaver, Oracle, BEA, jBoss ...

# 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

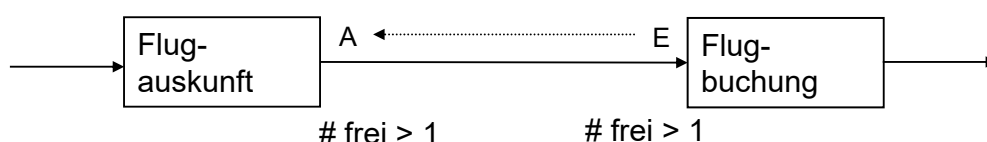


## ConTracts (Forts.)

- 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 Verarbeitung
  - Synchronisation
  - Recovery
  - Kontext

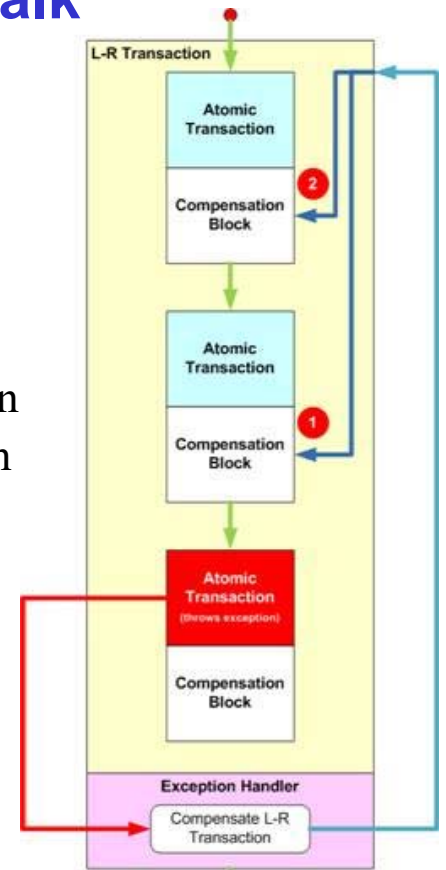
## Synchronisation von Contracts

- 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



# Transaktionen in Biztalk

- Realisierung bereits in XLANG-Sprache (Vorläufer von BPEL)
- GUI zur Definition der Prozesse
- Long Running (L-R) Transaktionen und atomare Transaktionen (Scopes)
  - Compensation Handler pro atomarer Transaktion
  - Anwendung in umfassender L-R-Transaktion im Rahmen des Exception Handling
- Default-Fehlerbehandlung: Backward Recovery über Compensation Handler
- Besonderheiten (Forward Recovery)
  - Retry für Atomic Tx möglich
  - manuelles Zurückgehen auf Persistence Points (Savepoints), die nach Atomic Tx gesetzt werden

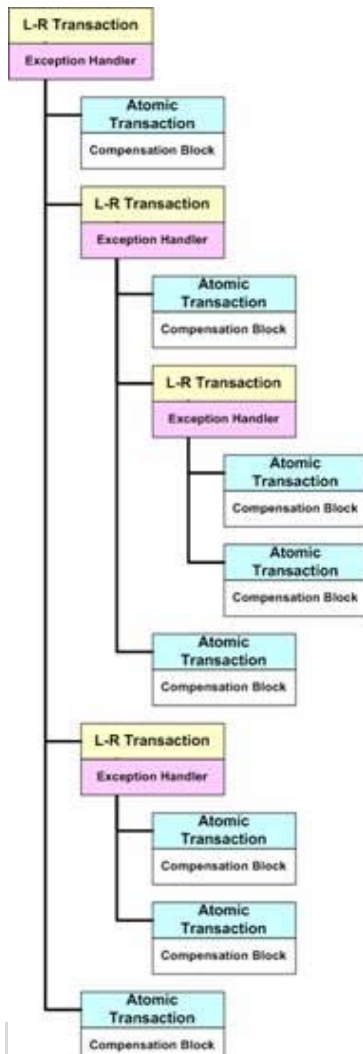


<http://geekswithblogs.net/cyoung/articles/100424.aspx>



## Transaktionen in Biztalk (2)

- Schachtelung von L-R und Atomic Tx
- Atomic Tx können als Blatt-Knoten auftreten
- Fehlerbehandlung jeweils auf nächsthöherer Ebene

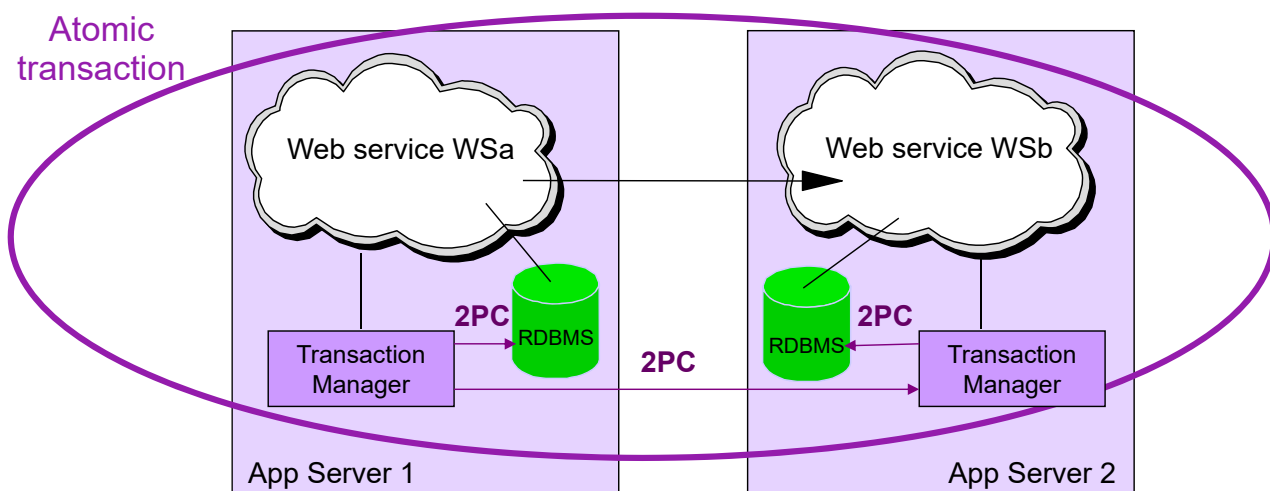


# WS-Transactions

- OASIS-Standard seit 2005 zur Durchführung und Koordination von Transaktionen über verschiedene Webdienste hinweg
  - Realisierung verteilter Geschäftsprozesse mit durch Webdienste realisierten Teiltransaktionen
  - (Teil-)Unterstützung u.a. in Apache Axis2, Oracle WebLogic Server, Microsoft Windows Communication Foundation (WCF)
- drei Teile
  - WS-Coordination +
  - WS-AtomicTransaction (AT)
  - WS-BusinessActivity (BA)
- WS-Coordination (WS-C)
  - Koordinierung der Transaktions/Prozess-Beendigung
  - unterschiedliche Transaktionskonzepte / Commit-Protokolle dazu nutzbar (inkl. selbstdefinierter)
  - Spezifikationen für *Activation Service* (Festlegung Koordinator und Protokoll), *Registration Service* (für neue Teilnehmer), *Completion Service* (Durchführung des gewählten Koordinierungsprotokolls)

## WS-AtomicTransaction

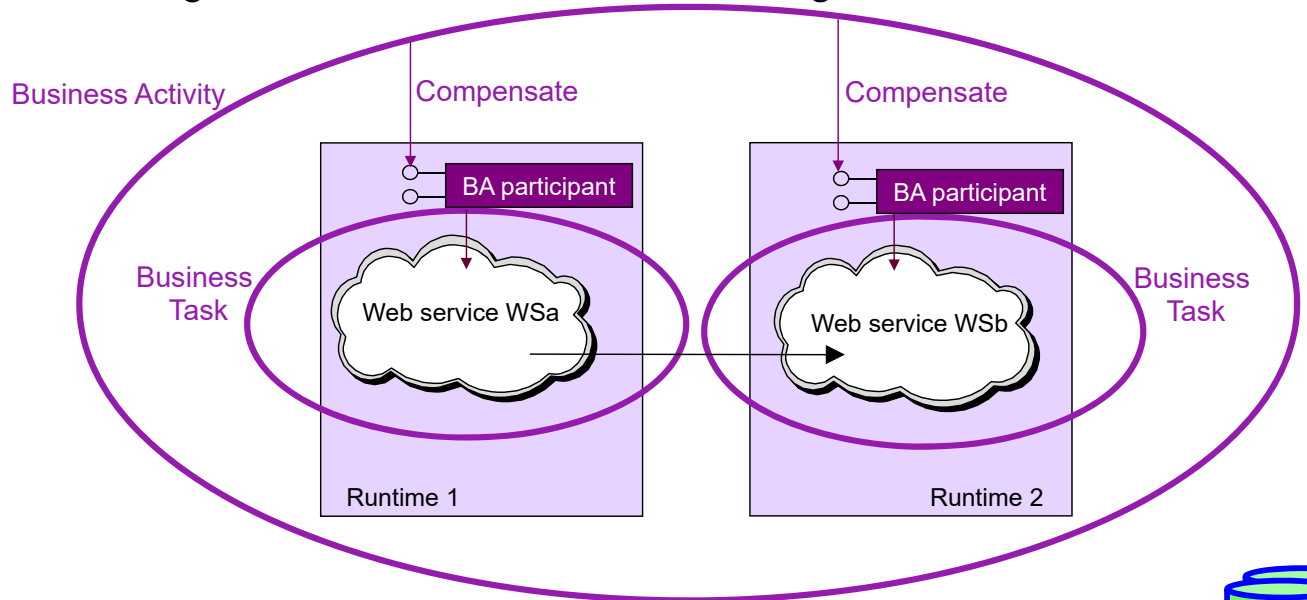
- verteilte ACID-Transaktionen mit mehreren Services
  - verteiltes 2-Phase-Commit (2PC) für Alles-oder-Nichts
  - Serialisierbarkeit (lange Sperren) etc.
  - v.a. sinnvoll innerhalb einer Organisation



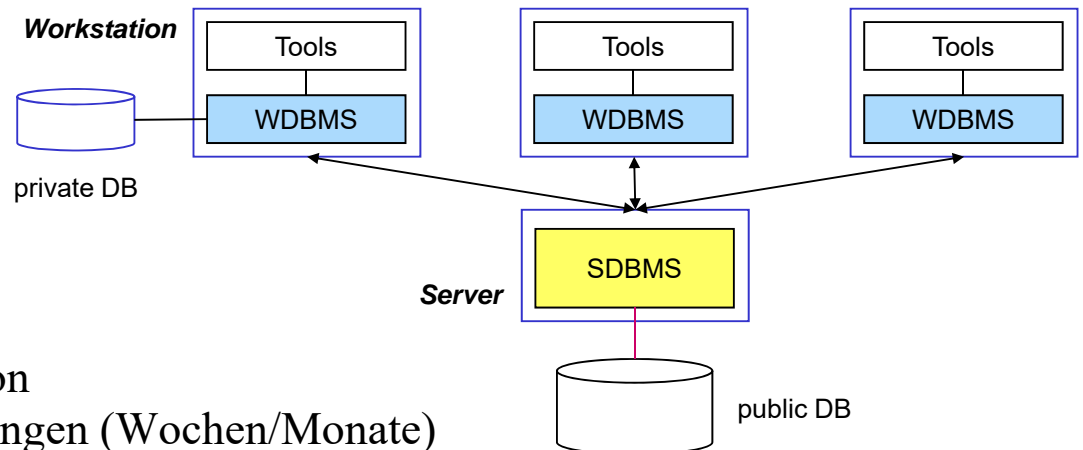


# WS-BusinessActivities

- v.a. für Prozesse zwischen verschiedenen Organisationen
- ähnliche Merkmale wie für L-R-Transaktionen in Biztalk
  - kompensationsbasiertes Zurücksetzen einzelner Services / Aktivitäten
  - keine Sperren über Services hinweg
  - Schachtelung von Aktivitäten (Scopes) möglich
  - Persistierung des Prozess-Zustandes nach Ausführung von Aktivitäten



## DB-Verarbeitung in Entwurfsumgebungen



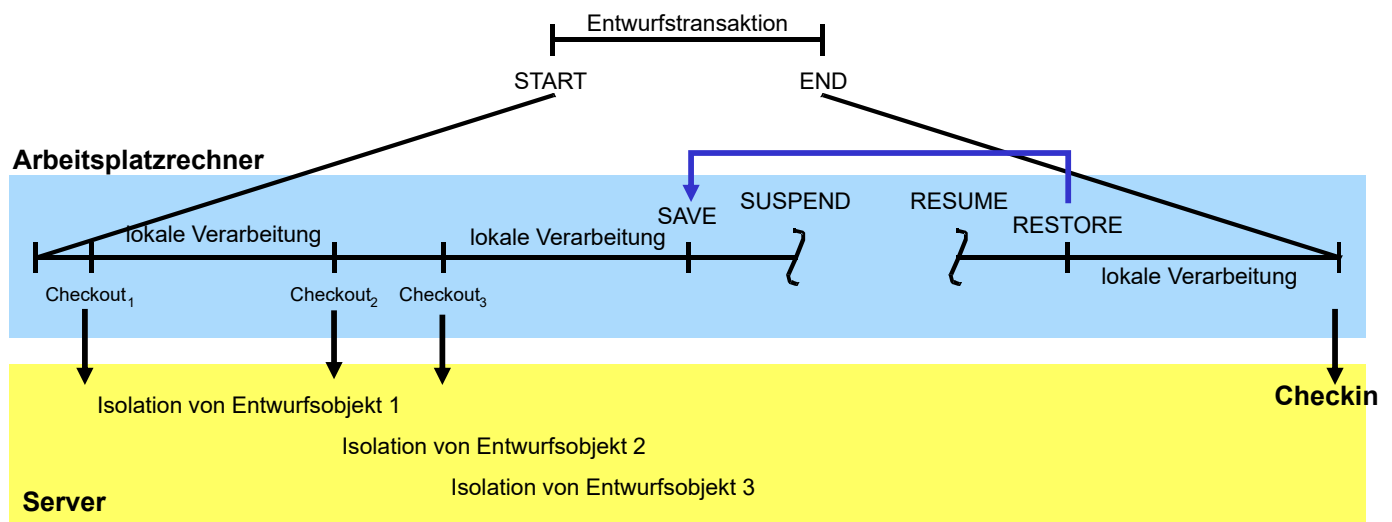
### ■ Merkmale

- 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)
  - Rücksetzen 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
  - Backward-Recovery durch Kompensation
  - Forward-Recovery erforderlich
- service-basierte Business-Prozesse
  - langlebige Transaktionen mit kompensierbaren ACID-Services
  - standardisierte Protokolle (WS-Transactions)
- Unterstützung langer Entwurfstransaktionen
  - zugeschnittene Verarbeitungsmodelle (Checkout/Checkin)
  - Kooperation innerhalb von Transaktionen
  - Unterstützung von Versionen und Savepoints