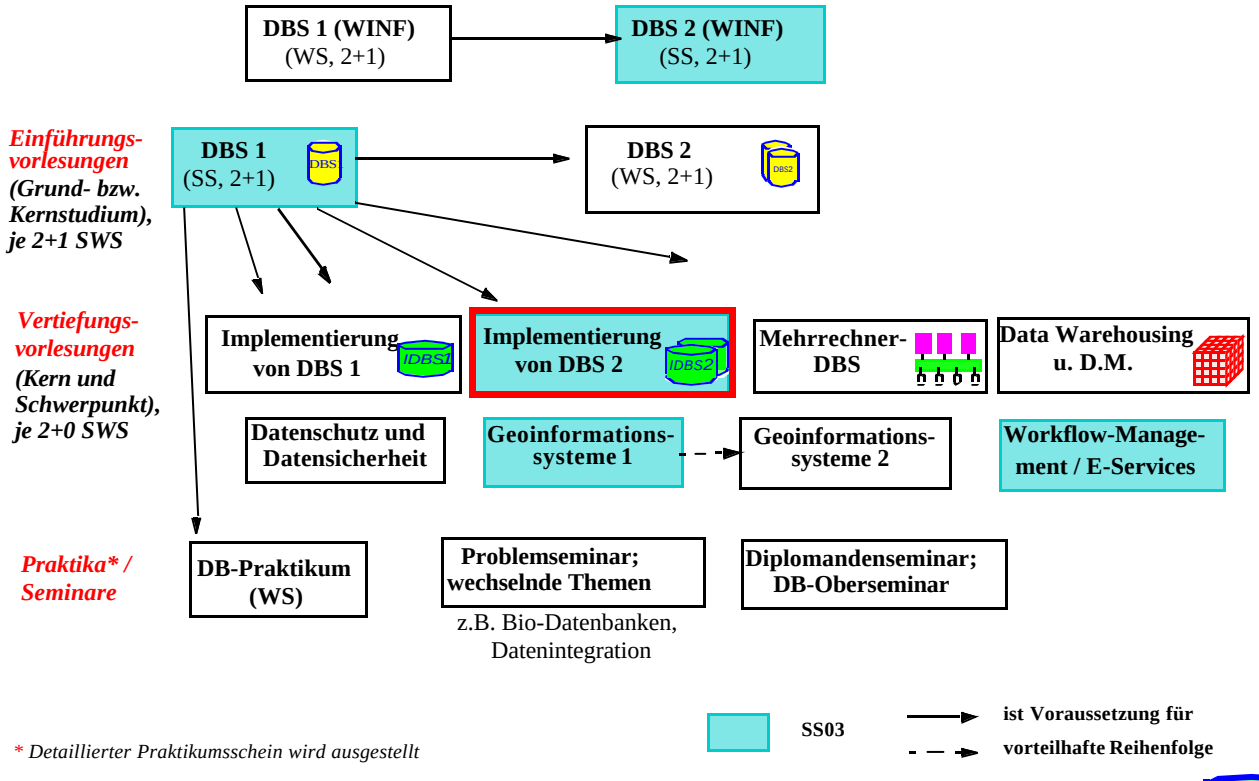
Implementierung von Datenbanksystemen 2

Sommersemester 2003

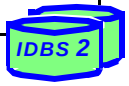
Prof. Dr. E. Rahm
Universität Leipzig
Institut für Informatik

http://dbs.uni-leipzig.de

Lehrveranstaltungen zu "Datenbanken" (SS03)



* *Detaillierter Praktikumsschein wird ausgestellt*

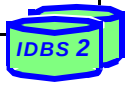


Scheinvergabe / Klausuren

- Vorlesung kann angerechnet werden
  - als Kernfachvorlesung im Diplom- / Master- / Bachelor-Studiengang Informatik
  - im Schwerpunkt „Praktische Informatik“
- Prüfungsvarianten
  - Modulklausur zu IDBS2 im Juli für Kerngebiet Praktische Informatik (2 SWS)
  - kombinierte Modulklausur IDBS1 + IDBS2 (4 SWS) im Juli 2003
  - Teil der mündlichen Schwerpunktprüfung
- geprüft werden konzeptionelles Wissen + Anwendungsfälle

Lernziele der Vorlesung IDBS

- fundierte Kenntnisse der Funktionsweise von Datenbanksystemen
- Implementierungstechniken zur Sicherstellung einer hohen Performanz der Datenverarbeitung sowie zur Datensicherheit
- IDBS2: Verfahren zur Transaktionsverwaltung: Synchronisation (Concurrency Control), Logging/Archivierung, Recovery ...
- tiefergehende Kenntnisse wichtig für Datenbank-Administratoren sowie generell für anspruchsvolle DB-Nutzung
- sachkundige Beurteilung von kommerziell verfügbaren DBS
- Verfahren nicht nur für Datenbanksysteme relevant (-> Betriebssysteme, Web-/ Applikations-Server, ...)

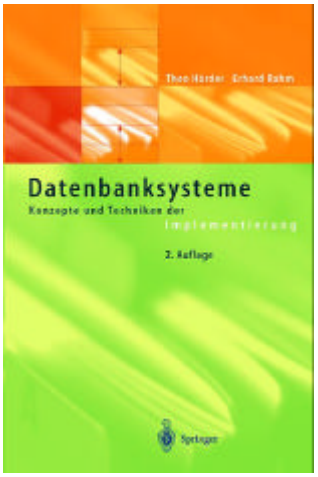


Vorläufige Vorlesungsübersicht

1. **Einführung: Transaktionsverwaltung, Integritätskontrolle**
2. **Synchronisation**
   - Mehrbenutzer-Anomalien
   - Serialisierbarkeit
   - Sperrverfahren, Deadlock-Behandlung
   - Optimistische Verfahren
   - Mehrversionen- und Zeitstempel-Verfahren
3. **Logging und Recovery**
   - Begriffe und Annahmen, Fehlermodell
   - Logging-Verfahren
   - Crash-Recovery
   - Platten-Recovery
4. **Transaktionskonzept: Weiterentwicklungen**
   - Geschachtelte Transaktionen
   - Mehrebenen-Transaktionen
   - Transaktionsketten (Sagas)
5. **Leistungsbewertung von DBS / Benchmarks**

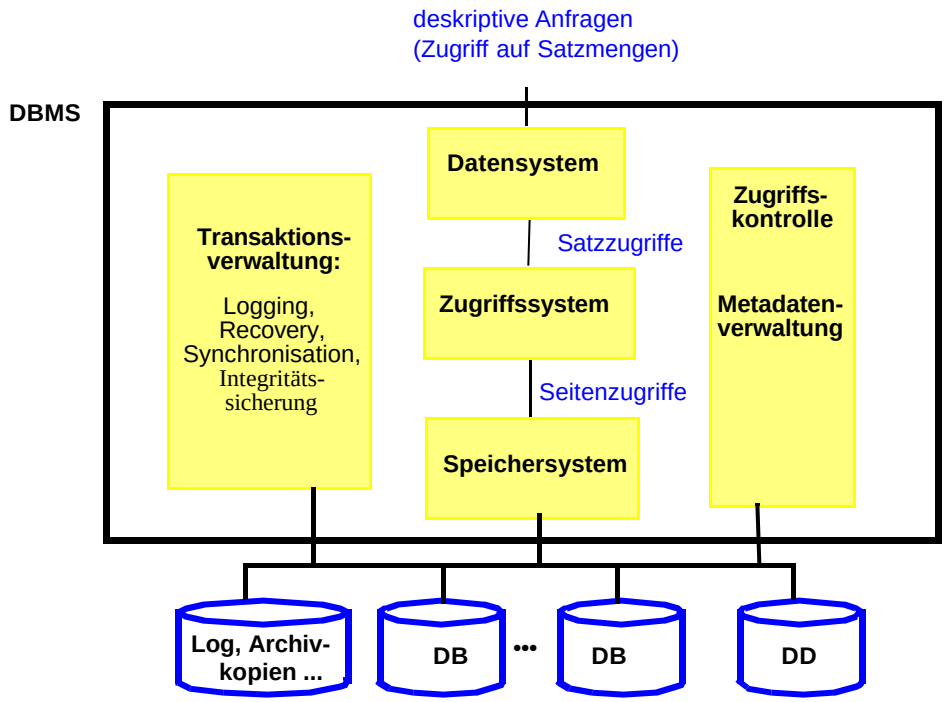
Literatur

- Härder, T., Rahm, E.: Datenbanksysteme - Konzepte und Techniken der Implementierung. Springer-Verlag, 2. Auflage 2001



- Weitere Lehrbücher
  - Weikum, G., Vossen, G.: Transactional Information Systems, Morgan Kaufmann, 2002
  - Saake, G., Heuer, A.: Datenbanken: Implementierungstechniken, MITP-Verlag, 1999
  - Gray, J., Reuter, A.: Transaction Processing. Morgan Kaufmann, 1993
- Forschungsergebnisse
  - Tagungsbände: VLDB (jährliche Konferenz "Very Large Data Bases"), SIGMOD (Konferenz der ACM Special Interest Group on Management of Data), Data Engineering (jährliche Konferenz der IEEE), EDBT, BTW ...
  - Zeitschriften: ACM TODS (Transactions on Database Systems), VLDB Journal (Very Large Data Bases), Datenbank-Spektrum ...
- DBLP-Portal: http://dblp.uni-trier.de
  (>350.000 Referenzen, viele Links auf Volltexte, Homepages etc.)

Grobaufbau eines DBS



Das Transaktionsparadigma

Definition der Transaktion:

Eine Transaktion ist eine Folge von DB-Operationen (DML-Befehlen), welche die Datenbank von einem logisch konsistenten Zustand in einen neuen logisch konsistenten Zustand überführt. Das DBS gewährleistet für Transaktionen die sogenannten ACID-Eigenschaften.

ACID-Prinzip

- **A**tomicity: 'Alles oder Nichts'-Eigenschaft (Fehlerisolierung)
- **C**onsistency: eine erfolgreiche Transaktion erhält die DB-Konsistenz (Menge der definierten Integritätsbedingungen)
- **I**solation: alle Aktionen innerhalb einer Transaktion müssen vor parallel ablaufenden Transaktionen verborgen werden (logischer Einbenutzerbetrieb)
- **D**urability: Überleben von Änderungen erfolgreich beendeter Transaktionen trotz beliebiger (erwarteter) Fehler garantieren (*Persistenz*).

Transaktionsparadigma (2)

- Programmierschnittstelle für Transaktionen
  - begin of transaction (BOT)
  - commit transaction („commit work“ in SQL)
  - rollback transaction („rollback work“ in SQL)
- Mögliche Ausgänge einer Transaktion

BOT	BOT	BOT
DML1	DML1	DML1
DML2	DML2	DML2
• • •	• • •	
DMLn	DMLn	
COMMIT WORK	ROLLBACK WORK	erzwungenes ROLLBACK
		Systemausfall, Programm-fehler usw.
normales Ende	**abnormales Ende**	**abnormales Ende**

- ACID vereinfacht DB-Anwendungsprogrammierung erheblich
  - Fehlertransparenz (failure transparency)
  - Transparenz der Nebenläufigkeit (concurrency transparency)
  - erlaubt also eine fehlerfreie Sicht auf die Datenbank im logischen Einbenutzerbetrieb

Transaktionsbeispiel: Debit/Credit

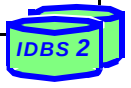
```
void main ( ) {
  EXEC SQL        BEGIN DECLARE SECTION
    int b /*balance*/, a /*accountid*/, amount;
  EXEC SQL        END DECLARE SECTION;
  /* read user input */
  scanf („%d %d“, &a, &amount);
  /* read account balance */
  EXEC SQL        Select Balance into :b From Account
                  Where Account_Id = :a;
/* add amount (positive for debit, negative for credit) */
  b = b + amount;
/* write account balance back into database */
  EXEC SQL        Update Account
                  Set Balance = :b Where Account_Id = :a;
  EXEC SQL        Commit Work;
}
```

Transaktionsverwaltung

- Mechanismen zur Einhaltung der ACID-Eigenschaften
  - Synchronisation (Concurrency Control)
  - Logging, Recovery, Commit-Behandlung
  - Integritätskontrolle
- Enge Abhängigkeiten untereinander sowie zu anderen Systemfunktionen (Pufferverwaltung, etc.)
- Das ACID-Paradigma eignet sich vor allem für relativ kurze Transaktionen, die in den meisten Anwendungen vorherrschen

Beispiel paralleler Ausführung (Synchronisationsproblem)

P1	Time	P2
Select Balance Into :b1		
From Account	**1**	
Where Account_Id = :a		
/* b1=0, a.Balance=100, b2=0 */		
		Select Balance Into :b2
	2	**From Account**
		Where Account_Id = :a
/* b1=100, a.Balance=100, b2=100 */		
b1 = b1-50	**3**	
/* b1=50, a.Balance=100, b2=100 */		
	4	**b2 = b2 +100**
/* b1=50, a.Balance=100, b2=200 */		
Update Account		
Set Balance = :b1	**5**	
Where Account_Id = :a		
/* b1=50, a.Balance=50, b2=200 */		
		Update Account
	6	**Set Balance = :b2**
		Where Account_Id = :a
/* b1=50, a.Balance=200, b2=200 */		



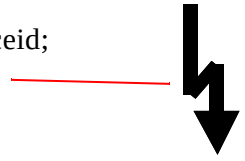
Synchronisation

- DBS müssen Mehrbenutzerbetrieb unterstützen
- ohne Synchronisation kommt es zu sogenannten Mehrbenutzer-Anomalien
  - Verlorengegangene Änderungen (lost updates)
  - Abhängigkeiten von nicht freigegeben Änderungen (dirty read, dirty overwrite)
  - inkonsistente Analyse (non-repeatable read)
  - Phantom-Probleme
- Anomalien sind nur durch Änderungen verursacht
- Synchronisation erfolgt automatisch durch das DBS
- zu klärende Fragen
  - Korrektheitskriterium ?
  - Realisierung ?
  - Leistungsfähigkeit ?

Atomaritätsproblem: Beispiel

- Unterbrechung während einer Überweisung

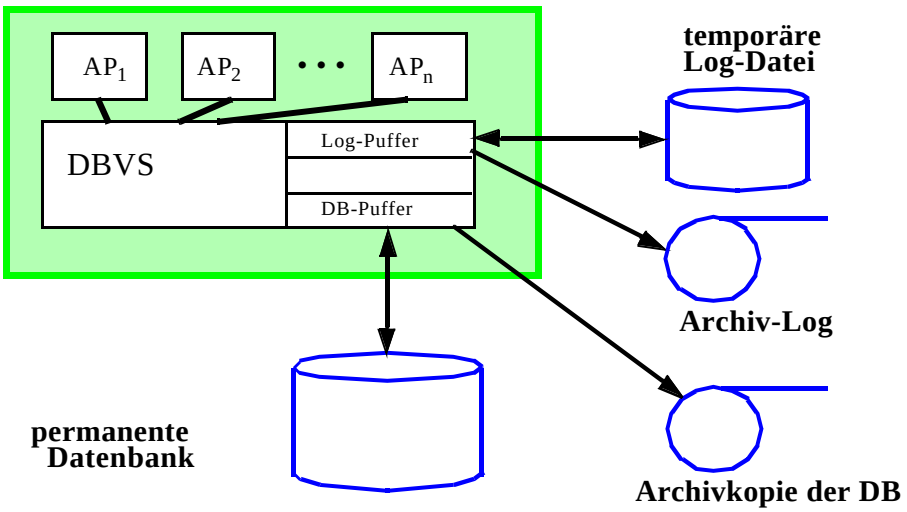
```
void main ( ) {
  /* read user input */
  scanf („%d %d %d“, &sourceid, &targetid, &amount);

  /* subtract amount from source account */
  EXEC SQL Update Account
    Set Balance = Balance - :amount Where Account_Id = :sourceid;

  /* add amount to target account */
  EXEC SQL Update Account
    Set Balance = Balance + :amount Where Account_Id = :targetid;
  EXEC SQL Commit Work; }
```



Recovery-Unterstützung

- automatische Behandlung aller erwarteten Fehler durch das DBVS
- Voraussetzung: Sammeln redundanter Informationen während Normalbetrieb (Logging)
- Transaktionsparadigma verlangt:
  - Alles-oder-Nichts-Eigenschaft von Transaktionen
  - Dauerhaftigkeit erfolgreicher Änderungen
- Zielzustand nach Recovery: jüngster, transaktionskonsistenter Zustand vor Erkennen des Fehlers
- Fehlerarten:
  - *Transaktionsfehler:* vollständiges Zurücksetzen auf Transaktionsbeginn (Undo)
  - *Systemfehler* (Rechnerausfall, DBVS-Absturz)
    - REDO für erfolgreiche Transaktionen (Wiederholung verlorengegangener Änderungen)
    - UNDO aller durch Ausfall unterbrochenen Transaktionen (Entfernen derer Änderungen aus der permanenten DB)
  - *Gerätefehler* (Plattenausfall):
    - vollständiges Wiederholen (REDO) aller Änderungen auf einer Archivkopie
    - oder: Spiegelplatten bzw. RAID-Disk-Arrays

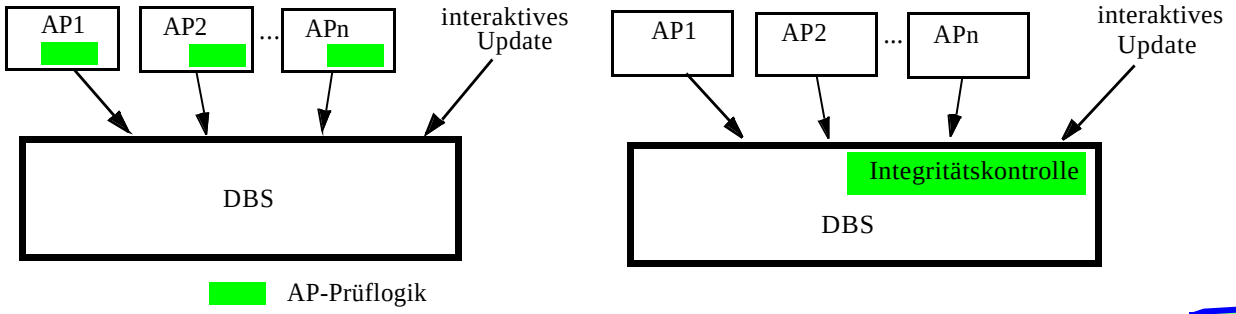
Systemkomponenten



- Pufferung von Log-Daten im Hauptspeicher (Log-Puffer)
  - Ausschreiben spätestens am Transaktionsende ("Commit")
- Temporäre Log-Datei zur Behandlung von Transaktions- und Systemfehler
- Behandlung von Gerätefehlern: Archivkopie + Archiv-Log

Integritätskontrolle

- Wahrung der logischen DB-Konsistenz
- Überwachung von semantischen Integritätsbedingungen durch Anwendungen oder durch DBS
- DBS-basierte Integritätskontrolle
  - größere Sicherheit
  - vereinfachte Anwendungserstellung
  - Unterstützung von interaktiven sowie programmierten DB-Änderungen
  - leichtere Änderbarkeit von Integritätsbedingungen
  - ggf. Leistungsvorteile



Arten von Integritätsbedingungen

- Modellinhärente vs. sonstige (modellunabhängige) Integritätsbedingungen
  - *Modellinhärente Bedingungen* des RM: Primärschlüsseleigenschaft; referentielle Integrität für Fremdschlüssel; Definitionsbereiche (Domains) für Attribute
- Reichweite der Bedingung
  - *Attributwert-Bedingungen* (z.B. Geburtsjahr > 1900)
  - *Satzbedingungen* (z.B. Geburtsdatum < Einstellungsdatum)
  - *Satztyp-Bedingungen* (z.B. Eindeutigkeit von Attributwerten)
  - *satztypübergreifende Bedingungen* (z.B. referentielle Integrität zwischen verschiedenen Tabellen)
- Statische vs. dynamische Bedingungen
  - *Statische Bedingungen* (Zustandsbedingungen) beschränken zulässige DB-Zustände (z.B. Gehalt < 500000)
  - *dynamische Integritätsbedingungen* (Übergangsbedingungen): zulässige Zustandsübergänge (z.B. Gehalt darf nicht kleiner werden)
  - Variante dynamischer IB: *temporale Integritätsbedingungen* für längerfristige Abläufe (z.B. Gehalt darf innerhalb von 3 Jahren nicht um mehr als 25% steigen).
- Zeitpunkt der Überprüfbarkeit: *unverzögerte* vs. *verzögerte Integritätsbedingungen*

Integritätsbedingungen in SQL92

- *Primary Key*- und *Foreign Key-Klauseln*
  - referentielle Integrität: deklarative Spezifikation unterschiedlicher Reaktionsmöglichkeiten für Wegfall (Löschung, Änderung) eines referenzierten Satzes bzw. Primärschlüssels (CASCADE, SET NULL, SET DEFAULT, NO ACTION)
- Festlegung von Wertebereichen für Attribute durch Angabe eines Datentyps bzw. Domains
  - optional: Angabe von Default-Werten, Eindeutigkeit (UNIQUE), Nullwerte-Ausschluß (NOT NULL)
  - allgemeine Wertebereichsbeschränkungen über CHECK-Klausel
- Spezifikation allgemeiner, z. B. tabellenübergreifender Bedingungen durch *CREATE ASSERTION*
- direkte (IMMEDIATE) oder verzögerte (DEFERRED) Überwachung spezifizierbar

```
CREATE TABLE  PERS
 (  PNR INT PRIMARY KEY,
    GEHALT INT  CHECK (VALUE < 500000),
    ANR   INT   NOT NULL
        FOREIGN KEY  REFERENCES  ABT
        ON DELETE CASCADE
              ... );
```

```
CREATE ASSERTION A1
  CHECK (NOT EXISTS (SELECT *
        FROM ABT
         WHERE ANR NOT IN
        (SELECT ANR FROM PERS)))
  DEFERRED;
```


Integritätsregeln

- Standardreaktion auf verletzte Integritätsbedingung: ROLLBACK
- Integritätsregeln erlauben Spezifikation von Folgeaktionen, z.B. um Einhaltung von IB zu erreichen
  - SQL92: deklarative Festlegung referentieller Folgeaktionen (CASCADE, SET NULL, ...)
  - SQL99: Trigger
  - Verallgemeinerung von Triggern: ECA-Regeln
- Trigger bzw. ECA-Regeln teilweise prozedural, jedoch sehr flexibel und mächtig
  - Realisierungsmöglichkeit für nahezu alle Integritätsbedingungen, u.a. dynamische IB
  - Zeitpunkte, Verwendung alter/neuer Werte, Aktionsteil im Detail festzulegen
  - viele Einsatzformen über Integritätskontrolle hinaus

```
CREATE TRIGGER GEHALTSTEST
AFTER UPDATE OF GEHALT ON PERS
REFERENCING OLD AS AltesGehalt,
NEW AS NeuesGehalt
WHEN (NeuesGehalt < AltesGehalt)
ROLLBACK;

CREATE TRIGGER MITARBEITERLÖSCHEN
AFTER DELETE ON ABT
REFERENCING OLD AS A
DELETE FROM PERS P WHERE P.ANR =
A.ANR;
```

- Probleme von Triggern
  - Trigger i.a. beschränkt auf Änderungsoperationen einer Tabelle (UPDATE, INSERT, DELETE)
  - derzeit i.a. keine verzögerte Auswertung von Triggern
  - Gefahr zyklischer, nicht-terminierender Aktivierungen
  - Korrektheit des DB-/Trigger-Entwurfes (Regelabhängigkeiten, parallele Regelausführung, ...)

Implementierungsaspekte der Integritätskontrolle

- IB-Überprüfung verlangt vom DBS Entscheidungen
  - für welche DB-Operationen welche Überprüfungen zusätzlich vorzunehmen sind
  - wann Überprüfungen durchzuführen sind (direkt, verzögert)
  - wie Überprüfungen vorzunehmen sind (Ausführungsplan)
- Behandlung zur Übersetzungszeit (falls Namen der Tabellen, Attribute... bekannt) oder zur Laufzeit
- in einfachen Fällen können IB über Anfragemodifikation (query modification) behandelt werden
  - Transformation von Änderungsoperation durch Hinzunahme einzuhaltender IB-Prädikate
  - verhindert Ausführung integritätsverletzender Änderungen

```
UPDATE PERS
SET GEHALT = GEHALT * 1.05
WHERE PNR = 4711
```

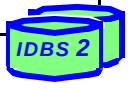
Integritätsbedingung GEHALT < 500000

- Integritätskontrolle über allgemeines Regelsystem
  - interne Verwendung von Triggern bzw. ECA-Regeln auch bei deklarativer Spezifikation von IB
  - dynamische Überwachung regelauslösender Ereignisse sowie der ausgelösten Ausführungen

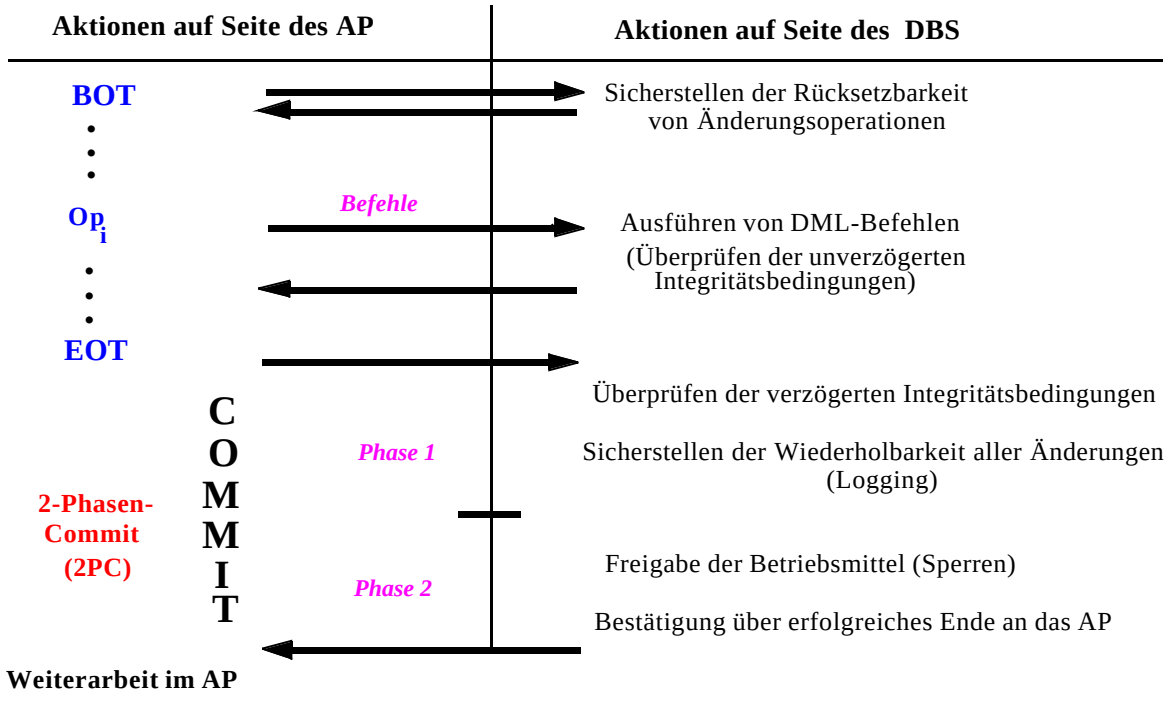
Regelverarbeitung im Starburst-Prototyp

- mengenorientierte Regelauswertung am Ende von Transaktionen
- pro Transaktion werden für jede geänderte Tabelle vier temporäre Relationen (transition tables) mit den von Änderungen betroffenen Sätzen geführt
  - deleted, inserted, old-updated, new-updated
- Begrenzung der Regelauswertung auf minimale Menge relevanter Daten
- Nutzung der Tabellen innerhalb von Regeln zur Integrationskontrolle (i.a. automatisch vom Compiler erzeugt)
- Beispiel: Wahrung der Integritätsbedingung GEHALT < 500000

```
CREATE RULE  GehaltsCheck ON PERS
WHEN  INSERTED,  UPDATED (GEHALT)  // zusammengesetztes Event (Disjunktion)
      IF        EXISTS (SELECT *     // Bedingung (Condition)
     FROM inserted UNION new-updated
    WHERE  GEHALT >= 500000)
     THEN ROLLBACK // Aktion
```



Die Transaktion als Schnittstelle zwischen Anwendungsprogramm und DBS



Zusammenfassung

- Transaktionskonzept vereinfacht Datenbank-Programmierung und -Nutzung: Transparenz gegenüber Fehlern und Mehrbenutzerbetrieb
- Transaktionsverwaltung zur Sicherung der ACID-Eigenschaften
  - Synchronisation
  - Logging / Recovery
  - Integritätskontrolle
- Integritätskontrolle möglichst weitgehend im DBS
- Zwei-Phasen-Commit: erfolgreicher Transaktionsabschluss (Commit) mit Gültigstellung von Änderungen erst nach Sicherstellung aller Integritätsbedingungen sowie der Wiederholbarkeit von Änderungen
- ACID v.a. für kurze Transaktionen geeignet
  -> Bedarf für erweiterte Transaktionskonzepte

2. Synchronisation in DBS: Grundlagen, Sperrverfahren

- Anomalien im Mehrbenutzerbetrieb
- Serialisierbarkeit
- Zweiphasen-Sperrprotokolle
- Hierarchische Sperrverfahren
- Konsistenzstufen von Transaktionen
- Deadlock-Behandlung
  - Timeout
  - Wait/Die, Wound/Wait, WDL
  - Erkennung)
- Implementierung der Datenstrukturen (Sperrtabelle)
- *Übungen*

Mehrbenutzerbetrieb

- zahlreiche Benutzer führen zu Mehrbenutzerbetrieb mit paralleler Ausführung unabhängiger Transaktionen auf derselben Datenbank
- serielle (sequentielle) Ausführung von Transaktionen inakzeptabel
  - sehr schlechte CPU-Nutzung aufgrund zahlreicher Transaktionsunterbrechungen: E/A, Kommunikationsvorgänge, Benutzer-Interaktionen („Denkzeiten“)
  - lange Transaktionen/Anfragen verursachen große Wartezeiten für andere
- Anomalien im Mehrbenutzerbetrieb ohne Synchronisation
  1. Verlorengegangene Änderungen *(lost updates)*
  2. Abhängigkeiten von nicht freigegebenen Änderungen *(dirty read, dirty overwrite)*
  3. Inkonsistente Analyse *(non-repeatable read)*
  4. Phantom-Problem

  ⇒ nur durch Änderungstransaktionen verursacht

Verlorengegange Änderung (Lost Update)

Gehaltsänderung T_1

```
SELECT  GEHALT  INTO  :gehalt
FROM  PERS
WHERE PNR = 2345

gehalt := gehalt + 2000;

UPDATE PERS
SET GEHALT = :gehalt
WHERE PNR = 2345
```

Gehaltsänderung T_2

```
SELECT  GEHALT INTO  :pgehalt
FROM  PERS
WHERE PNR = 2345

pgehalt := pgehalt + 1000;

UPDATE PERS
SET GEHALT = :pgehalt
WHERE PNR = 2345
```

DB-Inhalt
(PNR, GEHALT)

2345 39.000

2345

2345

Zeit

Schmutziges Lesen (Dirty Read)

Gehaltsänderung T_1

```
UPDATE PERS
SET GEHALT = GEHALT + 1000
WHERE PNR = 2345

...

ROLLBACK
```

Gehaltsänderung T_2

```
SELECT  GEHALT
      INTO  :gehalt
FROM  PERS
WHERE PNR = 2345

gehalt := gehalt * 1.05;

UPDATE PERS
SET GEHALT = :gehalt
WHERE PNR = 3456

COMMIT
```

DB-Inhalt
(PNR, GEHALT)

2345 39.000

2345

3456

2345

Zeit

Inkonsistente Analyse (Non-repeatable Read)

Lesetransaktion (Gehaltssumme berechnen)	**Änderungstransaktion**	**DB-Inhalt** (PNR, GEHALT)
		2345 39.000 3456 48.000
SELECT GEHALT INTO :gehalt FROM PERS WHERE PNR=2345 summe := summe + gehalt		
	UPDATE PERS SET GEHALT = GEHALT + 1000 WHERE PNR=2345	2345 40.000
	UPDATE PERS SET GEHALT = GEHALT + 2000 WHERE PNR=3456	3456 50.000
	COMMIT WORK	
SELECT GEHALT INTO :gehalt FROM PERS WHERE PNR=3456 summe := summe + gehalt		

Zeit

Phantom-Problem

Lesetransaktion (Gehaltssumme überprüfen)	**Änderungstransaktion** (Einfügen eines neuen Angestellten)
SELECT SUM (GEHALT) INTO :summe FROM PERS WHERE ANR=17	
	INSERT INTO PERS (PNR, ANR, GEHALT) VALUES (4567, 17, 55.000)
	COMMIT WORK
SELECT GEHALTSUMME INTO :summe2 FROM ABT WHERE ANR= 17	
IF summe <> summe2 THEN <Fehlerbehandlung>	

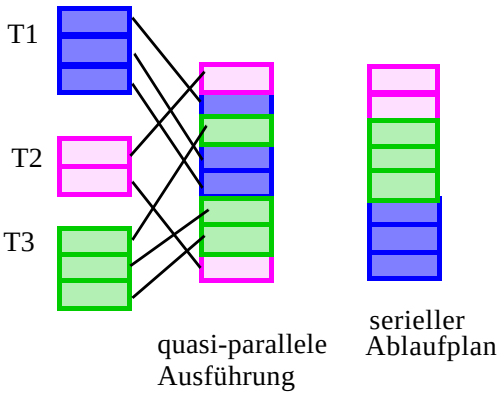
Zeit

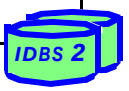
Synchronisation von Transaktionen: Modellannahmen

- Transaktion:
  Programm T mit DB-Anweisungen, so daß: Wenn T allein auf einer konsistenten DB ausgeführt wird, dann terminiert T (irgendwann) und hinterläßt DB in einem konsistenten Zustand. Während der Transaktionsverarbeitung werden keine Konsistenzgarantien eingehalten.
- Wenn Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten.
- DB-Anweisungen lassen sich nachbilden durch READ- und WRITE-Operationen
  Transaktion: BOT, Folge von READ- und WRITE-Anweisungen auf Objekte, EOT
- Die Ablauffolge von Transaktionen mit ihren Operationen kann durch einen *Schedule* beschrieben werden:
  Beispiel: $r_1(x), r_2(x), r_3(y), w_1(x), w_3(y), r_1(y), c_1, r_3(x), w_2(x), a_2, w_3(x), c_3, ...$
  Beispiel eines *seriellen Schedules*:
  $r_1(x), w_1(x), r_1(y), c_1, r_3(y), w_3(y), r_3(x), c_3, r_2(x), w_2(x), c_2,...$
  BOT ist implizit, EOT wird durch c_i (Commit) oder a_i (Abort / Rollback) dargestellt

Korrektheitskriterium der Synchronisation: Serialisierbarkeit

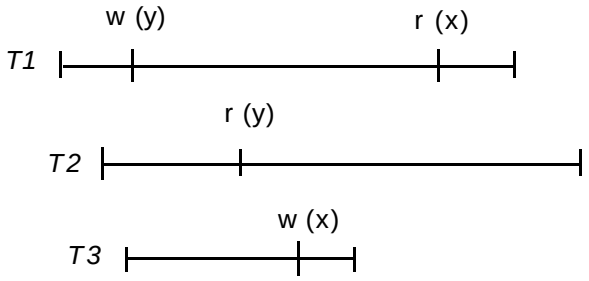
- Ziel der Synchronisation: logischer Einbenutzerbetrieb, d.h. Vermeidung aller Mehrbenutzeranomalien
- Gleichbedeutend mit formalem Korrektheitskriterium der *Serialisierbarkeit*:
  *Die parallele Ausführung einer Menge von n Transaktionen ist **serialisierbar**, wenn es eine serielle Ausführung derselben Transaktionen gibt, die den gleichen DB-Zustand und die gleichen Ausgabewerte wie die ursprüngliche Ausführung erzielt.*
- Hintergrund:
  - serielle Ablaufpläne sind korrekt
  - jeder Ablaufplan, der denselben Effekt wie ein serieller erzielt, ist akzeptierbar





Nachweis der Serialisierbarkeit

- Führen von zeitlichen Abhängigkeiten zwischen Transaktionen in einem *Abhängigkeitsgraphen (Konfliktgraphen)*
- Abhängigkeit (Konflikt) besteht, wenn zwei Transaktionen auf dasselbe Objekt mit nicht reihenfolgeunabhängigen Operationen zugreifen
- Konfliktarten:
  - Schreib-/Lese-Konflikt
  - Lese-/Schreib-Konflikt
  - Schreib-/Schreib-Konflikt

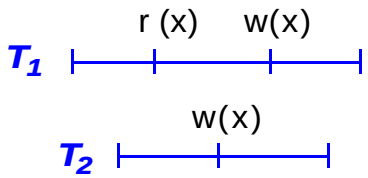


- Serialisierbarkeit liegt vor, wenn der Abhängigkeitsgraph keine Zyklen enthält

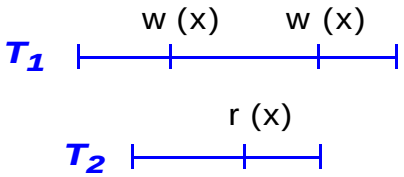
  => Abhängigkeitsgraph beschreibt partielle Ordnung zwischen Transaktionen, die sich zu einer vollständigen erweitern läßt (Serialisierungsreihenfolge)

Anomalien im Schreib/Lese-Modell

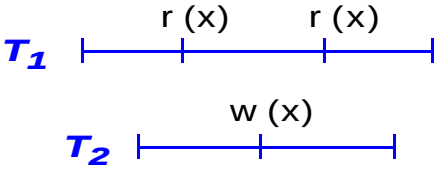
Lost Update



Dirty Read



Non-repeatable Read

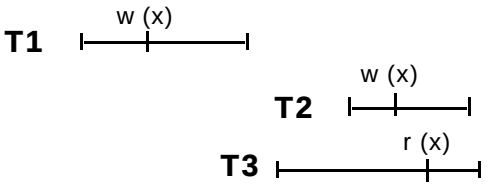


Konsistenzerhaltende Ablaufpläne

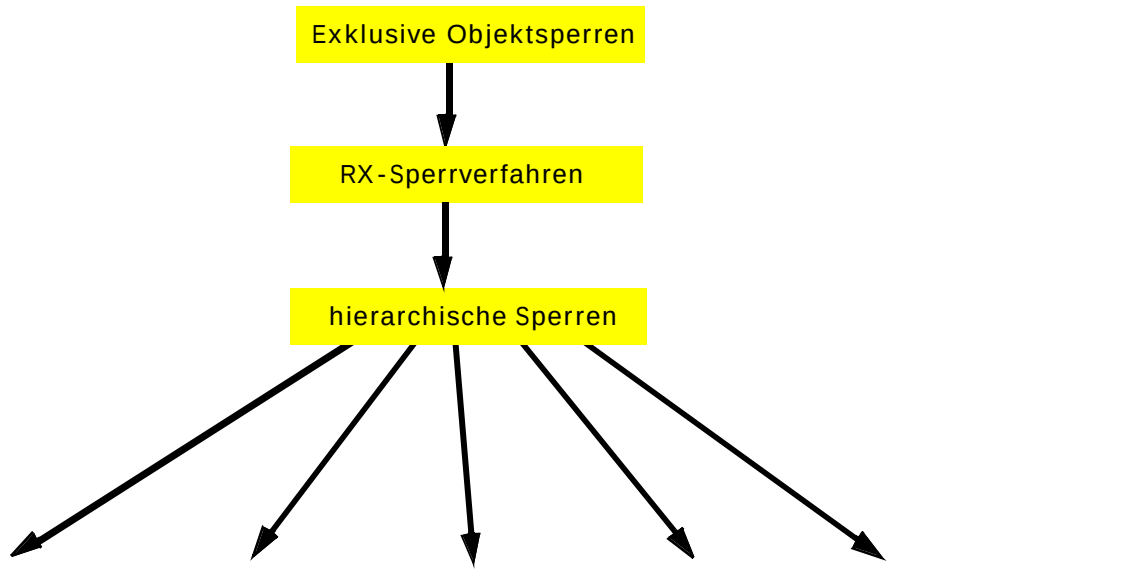
- Die Transaktionen T1 bis T3 müssen so synchronisiert werden, daß der resultierende DB-Zustand gleich dem ist, der bei der seriellen Ausführung in einer der folgenden Sequenzen zustande gekommen wäre.

T1, T2, T3	T2, T1, T3	T3, T1, T2
T1, T3, T2	T2, T3, T1	T3, T2, T1

- bei n Transaktionen bestehen n! mögliche serielle Schedules
- Sinnvolle Einschränkungen:
  - *Reihenfolgeerhaltende Serialisierbarkeit:* jede Transaktion sollte wenigstens alle Änderungen sehen, die bei ihrem Start (BOT) bereits beendet waren
  - *Chronologieerhaltende Serialisierbarkeit:* jede Transaktion sollte stets die aktuellste Objektversion sehen
- Beispiel



Historische Entwicklung von Synchronisationsverfahren

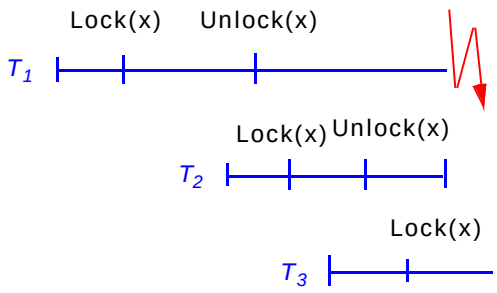


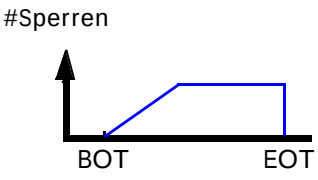
Zweiphasen-Sperrprotokolle (2 Phase Locking)

- Einhaltung folgender Regeln gewährleistet Serialisierbarkeit:
  1. vor jedem Objektzugriff muß Sperre mit ausreichendem Modus angefordert werden
  2. gesetzte Sperren anderer Transaktionen sind zu beachten
  3. eine Transaktion darf nicht mehrere Sperren für ein Objekt anfordern
  4. Zweiphasigkeit:
     - Anfordern von Sperren erfolgt in einer *Wachstumsphase*
     - Freigabe der Sperren in *Schrumpfungsphase*
     - Sperrfreigabe kann erst beginnen, wenn alle Sperren gehalten werden
  5. Spätestens bei EOT sind alle Sperren freizugeben

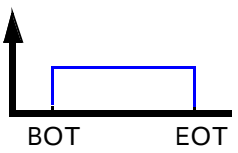
Striktes Zwei-Phasen-Sperren

- 2PL garantiert Serialisierbarkeit lediglich in einer fehlerfreien Umgebung
- Fehler während Schrumpfungsphase können zu "Dirty Read" etc. führen
- Lösungsalternativen
  - Lesen schmuziger Daten und Abhängigkeiten bei Commit überprüfen (Problem: kaskadierende Rollbacks)
  - Besser: strikte Zwei-Phasen-Sperrverfahren mit Sperrfreigabe nach Commit





strikt zweiphasiges Sperren



Preclaiming

RX-Sperrverfahren

- Sperranforderung einer Transaktion: R (Read) oder X (eXclusive bzw. Write)
- Gewährter Sperrmodus des Objektes: NL, R, X
- Kompatibilitätsmatrix:

angeforderter Modus \ aktueller Modus	NL	R	X
R			
X			

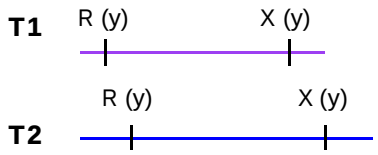
(NL (no lock) wird meist weggelassen)

\+ verträglich (kompatibel)

\- unverträglich

- unverträgliche Sperranforderung (Sperrkonflikt) führt zur Blockierung

Problem von Sperrkonversionen



- Sperrkonversionen führen oft zu Deadlocks
- Erweitertes Sperrverfahren
  - Ziel: Verhinderung von Konversions-Deadlocks
  - U-Sperre (Update) für Lesen mit Änderungsabsicht
  - bei Änderung Konversion U → X, andernfalls U → R (Downgrading)

angeforderter Modus \ aktueller Modus	R	U	X
R	+	-	-
U	+	-	-
X	-	-	-

  - u.a. in DB2 eingesetzt
  - das Verfahren ist unsymmetrisch - was würde eine Symmetrie bei U bewirken?

Konsistenzstufen von Transaktionen

- Usprüngliche Definition von Gray et al. (1976)
- *Konsistenzstufe 0:* Die Transaktionen halten kurze Schreibsperren auf den Objekten, die sie ändern
- *Konsistenzstufe 1:* Transaktionen halten lange Schreibsperren auf den Objekten, die sie ändern
- *Konsistenzstufe 2:* Transaktionen halten lange Schreibsperren auf den Objekten, die sie ändern, sowie kurze Lesesperren auf Objekten, die sie lesen
- *Konsistenzstufe 3:* Transaktionen halten lange Schreibsperren auf den Objekten, die sie ändern, sowie lange Lesesperren auf Objekten, die sie lesen.

Cursor Stability

- Kurze Lesesperren innerhalb von Änderungstransaktionen (Konsistenzstufe 2) können zu Lost Updates führen
- Cursor Stability: (kurze) Lesesperre bleibt gesetzt bis Cursor auf nächsten Satz wechselt
  - Verlust cursorbasierter Änderungen wird umgangen
  - Mitverantwortung des Programmierers zur Korrektheit der Synchronisation

```
exec sql SELECT  GEHALT INTO :gehalt
         FROM  PERS WHERE PNR=:pnr;
gehalt = gehalt + 1000;
exec sql   UPDATE PERS
           SET GEHALT = :gehalt
           WHERE PNR=:pnr;
```

```
exec sql    DECLARE CURSOR C FOR
            SELECT  GEHALT
            FROM  PERS WHERE PNR=:pnr;
exec sql    OPEN C;
exec sql    FETCH C INTO :gehalt;
gehalt = gehalt + 1000;
exec sql    UPDATE PERS
            SET GEHALT = :gehalt
            WHERE CURRENT OF CURSOR;
exec sql CLOSE C;
```

Konsistenzebenen in SQL92

- SQL92: vier Konsistenzebenen (Isolation Level) bzgl. Synchronisation
  - Konsistenzebenen sind durch die Anomalien bestimmt, die jeweils in Kauf genommen werden
  - Lost-Update muß generell vermieden werden
  - Default ist Serialisierbarkeit (serializable)

Konsistenzebene	Anomalie		
	Dirty Read	Non-Repeatable Read	Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

- SQL-Anweisung zum Setzen der Konsistenzebene:

```
SET  TRANSACTION <tx mode>, ISOLATION LEVEL <level>
```

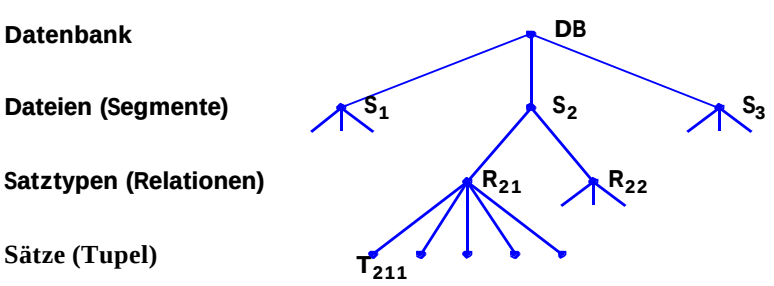
  - tx mode: READ WRITE (Default) bzw. READ ONLY

Beispiel: `SET TRANSACTION READ ONLY, ISOLATION LEVEL READ COMMITTED`

- READ UNCOMMITTED für Änderungstransaktionen unzulässig

Hierarchische Sperrverfahren

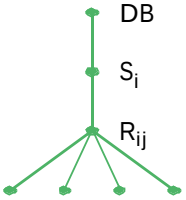
- Sperrgranulat bestimmt Parallelität/Aufwand
  - feines Granulat reduziert Sperrkonflikte
  - jedoch sind viele Sperren anzufordern und zu verwalten
- Hierarchische Verfahren erlauben Flexibilität bei Wahl des Granulates ('multigranularity locking')
  - z.B. lange Transaktionen (Anfragen) auf Relationenebene und kurze Transaktionen auf Satzebene synchronisieren
  - kommerzielle DBS unterstützen zumeist mindestens 2-stufige Objekthierarchie, z.B. Segment-Seite bzw. Satztyp (Relation) - Satz (Tupel)

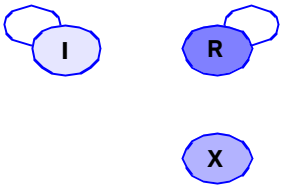


Hierarchische Sperrverfahren: Anwartschaftssperren

- mit R- und X-Sperre werden alle Nachfolgerknoten implizit mitgesperrt => *Einsparungen möglich*
- alle Vorgängerknoten sind ebenfalls zu sperren, um Unverträglichkeiten zu vermeiden => Verwendung von Anwartschaftssperren ('intention locks')
- einfachste Lösung: Nutzung eines Sperrtyps (I-Sperre)

	I	R	X
I	+	-	-
R	-	+	-
X	-	-	-



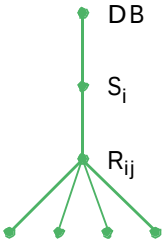


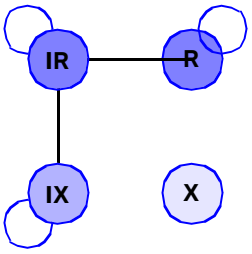
- Unverträglichkeit von I- und R-Sperren zu restriktiv => zwei Arten von Anwartschaftssperren (IR und IX)

Anwartschaftssperren (2)

- Anwartschaftssperren für Leser und Schreiber

	IR	IX	R	X
IR	+	+	+	-
IX	+	+	-	-
R	+	-	+	-
X	-	-	-	-





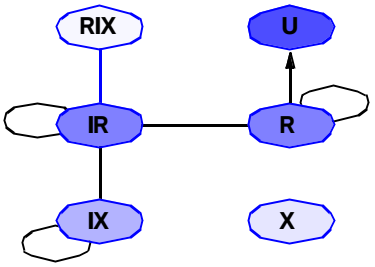
- IR- Sperre (intent read), falls auf untergeordneten Objekten nur lesend zugegriffen wird, sonst IX-Sperre
- Weitere Verfeinerung sinnvoll, um den Fall zu unterstützen, wo alle Tupel eines Satztyps gelesen und nur einige davon geändert werden sollen
  - X-Sperre auf Satztyp sehr restriktiv
  - IX-auf Satztyp verlangt Sperren jedes Tupels

  => neuer Typ von Anwartschaftssperre: RIX = R + IX
  - nur für zu ändernde Sätze muß (X-)Sperre auf Tupelebene angefordert werden

Anwartschaftssperren (3)

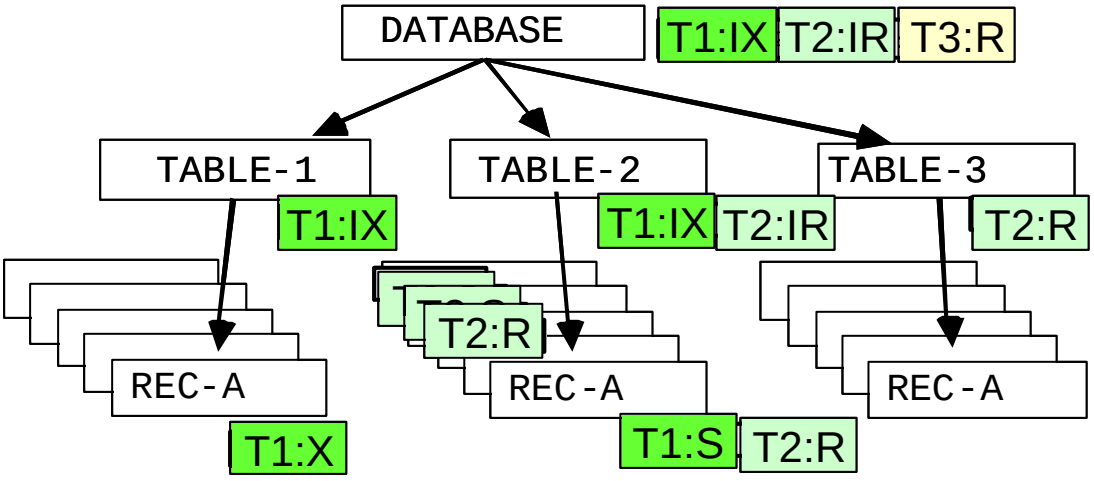
- Vollständiges Protokoll der Anwartschaftssperren
  - RIX gibt ein Leserecht auf den Knoten und seine Nachfolger. Weiterhin ist damit das Recht verbunden, auf Nachfolger-Knoten IX, U und X-Sperren anzufordern.
  - U gewährt Leserecht auf den Knoten und seine Nachfolger - repräsentiert die Absicht, den Knoten in der Zukunft zu verändern. Bei Änderung Konversion U → X, sonst U → R.

	IR	IX	R	RIX	U	X
IR	+	+	+	+	-	-
IX	+	+	-	-	-	-
R	+	-	+	-	-	-
RIX	+	-	-	-	-	-
U	-	-	+	-	-	-
X	-	-	-	-	-	-



  - Sperranforderungen von der Wurzel zu den Blättern
  - Bei R- oder IR-Anforderung müssen für alle Vorgänger IX- oder IR-Sperren erworben werden
  - Bei X-, U-, RIX- o. IX-Anforderung müssen alle Vorgänger in RIX oder IX gehalten werden
  - Sperrfreigaben von den Blättern zu der Wurzel
  - Bei EOT sind alle Sperren freizugeben

Hierarchische Sperren: Beispiel



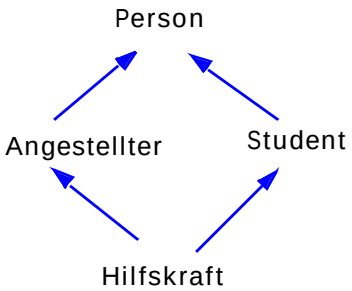
  - T3 wartet
  - T2 hat Lesesperre auf der gesamten Tabelle 3
  - Lesesperren auf Satzebene in Tabelle 2
  - T1 hat Satzsperren in Tabelle 1 und 2

Hierarchische Sperren: (De-) Escala-tion

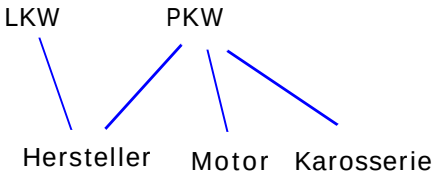
- Lock Escalation
  - Falls Transaktion sehr viele Sperren benötigt -> dynamisches Umschalten auf gröberes Granulat
  - Bsp.: nach 1000 Satzsperren auf einer Tabelle -> 1 Tabellensperre erwerben
  - Schranke ist typischer Tuning-Parameter
- Manchmal kann *Lock De-Escalation* sinnvoll sein
  - Erwerbe grob-granulare Sperre (z.B. für Tabelle)
  - vermerke referenzierte Objekte auf fein-granularer Ebene (z.B. Sätze)
  - bei Konflikt(en): Umschalten auf feine Sperren

Hierarchische Sperren in OODBS

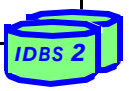
- Reduzierung des Sperraufwandes innerhalb von Generalisierungs- und Aggregationshierarchien
- Generalisierung: Probleme durch Mehrfachvererbung
- Aggregation: Probleme durch gemeinsam genutzte Komponentenobjekte
  - explizites Sperren aller Teilobjekte sehr aufwendig



a) Generalisierungshierarchie



b) Aggregationshierarchie

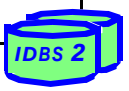


Deadlock-Behandlung

- 5 Voraussetzungen für Deadlock
  - paralleler Objektzugriff
  - exklusive Zugriffsanforderungen
  - anfordernde Transaktion besitzt bereits Objekte/Sperren
  - keine vorzeitige Freigabe von Objekten/Sperren (non-preemption)
  - zyklische Wartebeziehung zwischen zwei oder mehr Transaktionen
- Beispiel
- Datenbanksysteme: Behandlung erfordert Rücksetzung (Rollback) von Transaktionen

Lösungsmöglichkeiten zur Deadlock-Behandlung

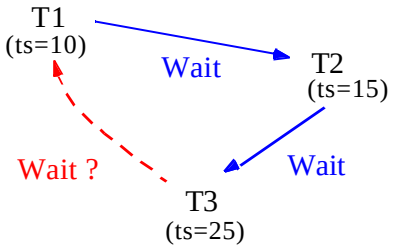
- 1. Timeout-Verfahren
  - Transaktion wird nach festgelegter Wartezeit auf Sperre zurückgesetzt
  - problematische Bestimmung des Timeout-Wertes
  - viele unnötige Rücksetzungen bei kleinem Timeout-Wert
  - lange Blockaden bei großem Timeout-Wert
- 2. Deadlock-Verhütung (Prevention)
  - *keine Laufzeitunterstützung* zur Deadlock-Behandlung erforderlich
  - Bsp.: Preclaiming (in DBS i.a. nicht praktikabel)
- 3. Deadlock-Vermeidung (Avoidance)
  - potentielle Deadlocks werden durch entsprechende Maßnahmen vermieden
  - Laufzeitunterstützung nötig
- 4. Deadlock-Erkennung (Detection)
  - Zyklensuche innerhalb eines Wartegraphen
  - gestattet minimale Anzahl von Rücksetzungen



Deadlock-Vermeidung

- Zuweisung einer eindeutigen *Transaktionszeitmarke* bei BOT
- im Konfliktfall darf nur ältere (bzw. jüngere) Transaktion warten => kein Zyklus möglich
  - ursprünglich für Verteilte DBS vorgeschlagen: Behandlung globaler Deadlocks ohne Kommunikation
- WAIT/DIE-Verfahren
  - anfordernde Transaktion wird zurückgesetzt, falls sie jünger als Sperrbesitzer ist
  - ältere Transaktionen warten auf jüngere

T_i fordert Sperre, Konflikt mit T_j:
  if *ts* (T_i) < *ts* (T_j) { T_i älter als T_j }
  then WAIT (T_i)
  else ROLLBACK (T_i) { "Die" }



Deadlock-Vermeidung (2)

- WOUND / WAIT-Verfahren:
  - Sperrbesitzer wird zurückgesetzt, wenn er jünger als anfordernde Transaktion ist
  - jüngere Transaktionen warten auf ältere
  - preemptiver Ansatz

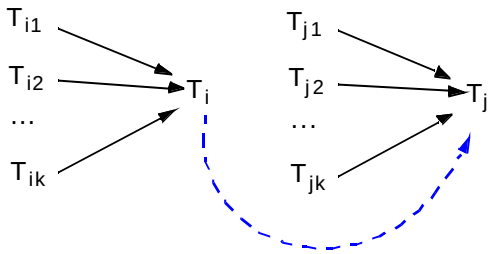
T_i fordert Sperre, Konflikt mit T_j:
  if *ts* (T_i) < *ts* (T_j) { T_i älter als T_j }
  then ROLLBACK (T_j){ "Wound" }
  else WAIT (T_i)

- Verbesserung für Wait/Die und Wound/Wait:
  - statt BOT-Zeitmarke Zuweisung der Transaktionszeitmarke erst bei erstem Sperrkonflikt ("dynamische Zeitmarken")
  - erster Sperrkonflikt kann stets ohne Rücksetzung behandelt werden

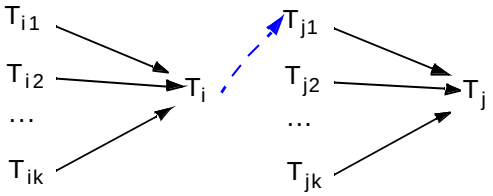
Deadlock-Vermeidung: Wait Depth Limited

- "Wartetiefe" (wait depth):
  - eine nicht-blockierte Transaktion hat Wartetiefe 0
  - blockierte Transaktion T hat Wartetiefe i+1,falls i die maximale Wartetiefe derjenigen Transaktionen ist, die T blockieren
- Wait Depth Limited (WDL): Begrenzung der maximalen Wartetiefe d
  - d=0: kein Warten (Immediate Restart) -> keine Deadlocks möglich
  - d=1: Warten erfolgt nur auf nicht-blockierte (laufende) Transaktionen -> keine Deadlocks möglich
- Problemfälle mit drohender Wartetiefe > 1

Fall 1:



Fall 2:

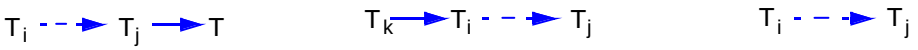


Wait-Depth Limited (2)

- WDL1-Variante "Running Priority"

T_i fordert Sperre, Konflikt mit T_j:

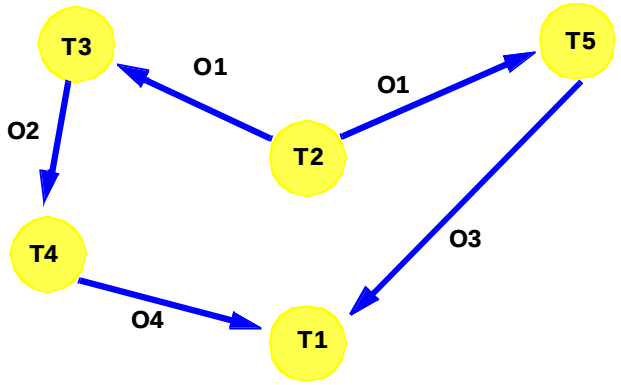
if (T_j blockiert) then KILL (T_j) {Bevorzugung der laufenden Transaktion}
else if (T_k wartet auf T_i) {es existiert ein T_k, die auf T_i wartet }
then ROLLBACK (T_i)
else WAIT (T_i)



- bei hohen Konfliktraten zeigte WDL in Simulationen besseres Leistungsverhalten als 2PL

Deadlock-Erkennung

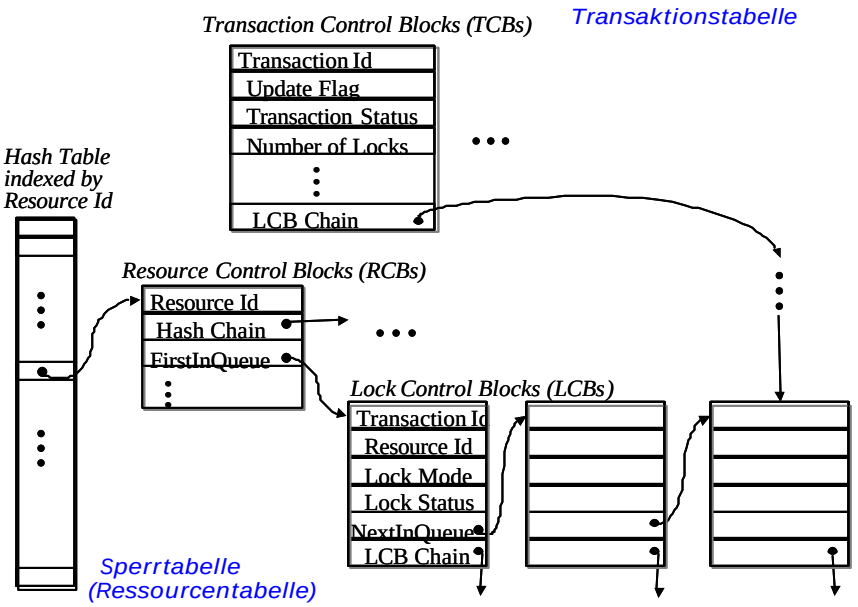
- Explizites Führen eines *Wartegraphen* (wait-for graph) und Zyklensuche zur Erkennung von Verklemmungen



- Deadlock-Auflösung durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter Transaktion (z.B. Verursacher oder 'billigste' Transaktion)
- Zyklensuche entweder
  - bei jedem Sperrkonflikt bzw.
  - verzögert (z.B. über Timeout gesteuert)

Implementierungsaspekte: Datenstrukturen

- Hash-Tabelle zur Realisierung der Sperrtabelle
  - schneller Zugriff auf Objekt/Ressourcenkontrollblöcke für Lock-Aufrufe
- Kurzzeitsperren („Latch") für Zugriffe auf Sperrtabelle
  - Semaphor pro Hash-Klasse reduziert Konflikt-Gefahr
- Matrixorganisation Objekt-/Transaktionstabelle
  - schnelle Bestimmung freizugebender Sperren bei Commit



Sperrverfahren in Datenbanksystemen

- Sperrverfahren: Vermeidung von Anomalien, indem
  - zu ändernde Objekte dem Zugriff aller anderen Transaktionen entzogen werden,
  - zu lesende Objekte vor Änderungen geschützt werden
- Standardverfahren: Hierarchisches Zweiphasen-Sperrprotokoll
  - mehrere Sperrgranulate
  - Verringerung der Anzahl der Sperranforderungen
- Probleme bei der Implementierung von Sperren
  - kleine Sperreinheiten (wünschenswert) erfordern hohen Aufwand
  - Sperranforderung und -freigabe sollten sehr schnell erfolgen, da sie sehr häufig benötigt werden
  - explizite Satzsperren führen u.U. zu umfangreichen Sperrtabellen und großem Zusatzaufwand
  - Zweiphasigkeit der Sperren führt häufig zu langen Wartezeiten (starke Serialisierung)
  - häufig benutzte Indexstrukturen können zu Engpässen werden
  - Eigenschaften des Schemas können "hot spots" erzeugen
- Optimierungen:
  - Änderungen auf privaten Objektkopien (verkürzte Dauer exklusiver Sperren)
  - Nutzung mehrerer Objektversionen
  - spezialisierte Sperren (Nutzung der Semantik von Änderungsoperationen)

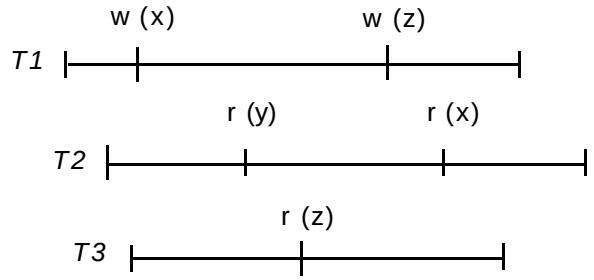
Übungsbeispiel 1

Lesetransaktion (Gehaltssumme berechnen)	**Änderungstransaktion**	**DB-Inhalt** (PNR, GEHALT)
		2345 39.000 3456 48.000
	UPDATE PERS SET GEHALT = GEHALT + 1000 WHERE PNR=2345	
SELECT SUM (GEHALT) INTO :gehalt FROM PERS WHERE PNR IN (2345, 3456)		
	UPDATE PERS SET GEHALT = GEHALT + 2000 WHERE PNR=3456	
	COMMIT WORK	

Zeit

- Welche Mehrbenutzer-Anomalien liegen vor, falls keine Synchronisation erfolgt?
- Wie würde sich eine Synchronisation mit einem RX-Sperrverfahren auswirken?

Übungsbeispiel 2



- Serialisierbarkeit?
- RX:
- RX + Wait-Die:
- RX + Wound-Wait:
- RX + Running Priority:

Übungsfragen

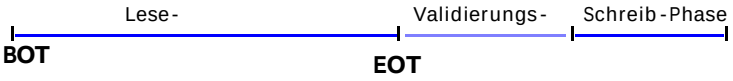
- Auch bei Sperrverfahren mit Deadlock-Erkennung ist ein „Verhungern“ von Transaktionen möglich, falls diese im Falle eines Deadlocks stets als Opfer ausgewählt und zurückgesetzt werden. Welche Möglichkeiten bestehen, jeder Transaktion letztlich ein Durchkommen zu garantieren?
- Zeigen Sie einen Beispiel-Schedule, bei dem „Running Priority“ zu einer Transaktionsrücksetzung führt, ohne daß ein Deadlock vorliegt.
- Überprüfen Sie (z.B. anhand von Online-Manualen), welche der kommerziellen DBS Oracle, IBM DB2 und Microsoft SQL-Server das Konzept der „Cursor Stability“ kennen und unterstützen.

3. Synchronisation: Weitere Verfahren, Leistungsbewertung

- Optimistische Synchronisation
  - BOCC, FOCC
  - Kombination von OCC und Sperrverfahren
- Zeitstempel-Verfahren
- Mehrversionen-Synchronisation
- Prädikatsperren
- Synchronisation bei "High Traffic"-Elementen
- Sperren von B*-Bäumen
- Leistungsbewertung, Lastkontrolle

Optimistische Synchronisation (OCC)

- 3-phasige Verarbeitung:



- Lesephase
  - eigentliche Transaktionsverarbeitung
  - Änderungen einer Transaktion werden in einem privaten Puffer durchgeführt
- Validierungsphase
  - Überprüfung, ob ein Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer der parallel ablaufenden Transaktionen passiert ist
  - Konfliktauflösung durch Zurücksetzen von Transaktionen
- Schreibphase
  - nur bei positiver Validierung
  - Lese-Transaktion ist ohne Zusatzaufwand beendet
  - Schreib-Transaktion schreibt hinreichende Log-Information und propagiert ihre Änderungen
  - Grundannahme: geringe Konfliktwahrscheinlichkeit

Optimistische Synchronisation (2)

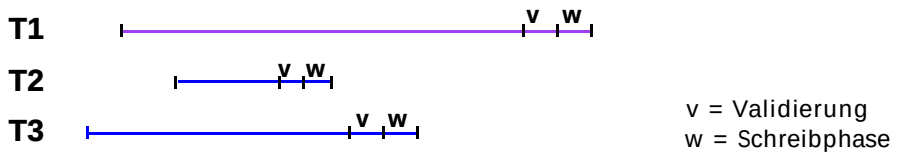
- Allgemeine Eigenschaften von OCC:
  - \+ einfache Transaktionsrücksetzung
  - \+ keine Deadlocks
  - \- mehr Rücksetzungen als bei Sperrverfahren
  - \- Gefahr des „Verhungerns“ (starvation) von Transaktionen
- zur Durchführung der Validierungen werden pro Transaktion der Read-Set (RS) und Write-Set (WS) geführt
- Forderung:
  - eine Transaktion kann nur erfolgreich validieren, wenn sie alle Änderungen von zuvor validierten Transaktionen gesehen hat
  - Validierungsreihenfolge bestimmt Serialisierungsreihenfolge
- generelle Validierungsstrategien:
  - **Backward Oriented (BOCC):** Validierung gegenüber bereits beendeten (Änderungs-) Transaktionen
  - **Forward Oriented (FOCC):** Validierung gegenüber laufenden Transaktionen

BOCC

- ursprünglich vorgeschlagenes Verfahren zur optimistischen Synchronisation [Kung&Robinson, ACM TODS 1981]
- Validierung von Transaktion T:

BOCC-Test gegenüber allen Änderungstransaktionen T_j, die seit BOT von T erfolgreich validiert haben:

IF $RS(T) \cap WS(T_j) \neq \emptyset$ *THEN ROLLBACK (T) ELSE SCHREIBPHASE*

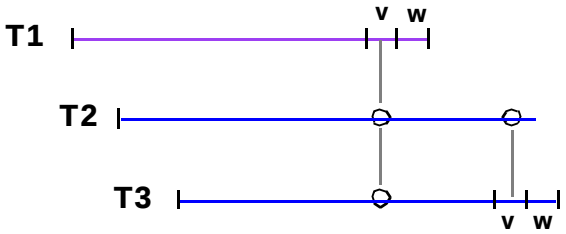


- Nachteile/Probleme:
  - unnötige Rücksetzungen wegen ungenauer Konfliktanalyse
  - Aufbewahren der Write-Sets beendeter Transaktionen erforderlich
  - hohe Anzahl von Vergleichen bei Validierung
  - Rücksetzung erst bei EOT => viel unnötige Arbeit
  - es kann nur die validierende Transaktion zurückgesetzt werden => Gefahr von ’starvation’
  - hohes Rücksetzrisiko für lange Transaktionen und bei Hot-Spots

FOCC

- nur Update-Transaktionen validieren gegenüber laufenden Transaktionen T_i

  Validierungstest:
  $WS(T) \cap RS(T_i) = \varnothing$



- Vorteile:
  - Wahlmöglichkeit des Opfers (Kill, Rollback, Prioritäten, ...)
  - keine unnötigen Rücksetzungen
  - frühzeitige Rücksetzung möglich => Einsparen unnötiger Arbeit
  - keine Aufbewahrung von Write-Sets, geringerer Validierungsaufwand als bei BOCC
- Probleme:
  - Während Validierungs- und Schreibphase muß WS (T) 'gesperrt' werden, damit sich die RS (T_i) nicht ändern (keine Deadlocks damit möglich)
  - immer noch hohe Rücksetzrate möglich
  - es kann immer nur einer Transaktion Durchkommen definitiv zugesichert werden

BOCC+

- Idee
  - Konflikterkennung über Zeitstempel (Änderungszähler) statt Mengenvergleich
  - erfolgreich validierte Transaktionen erhalten eindeutige, monoton wachsende Transaktionsnummer
  - geänderte Objekte erhalten Transaktionsnummer der ändernden Transaktion als Zeitstempel TS zugeordnet
  - beim Lesen eines Objektes wird Zeitstempel ts der gesehenen Version im Read-Set vermerkt
  - Validierung überprüft, ob gesehene Objektversionen zum Validierungszeitpunkt noch aktuell sind:

```
VALID := true
<< for all r in RS (T) do;
       if ts (r,T) < TS (r) then VALID := false;
   end;
   if VALID then do;
      TNC := TNC + 1; {ergibt Transaktionsnummer für T}
      for all w in WS (T) do;
          TS (w) := TNC;
          setze laufende Transaktionen mit w in RS zurück;
      end; >>
      Schreibphase für T;
   end;
   else (setze T zurück);
```

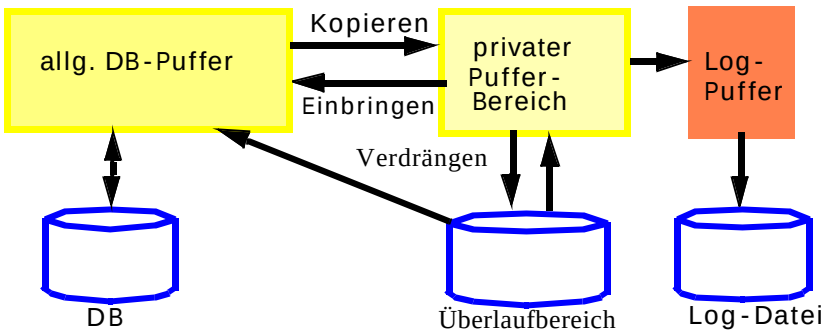
BOCC+ (2)

- Zum Scheitern verurteilte Transaktionen können sofort zurückgesetzt werden
- Zeitstempel TS für geänderte Objekte können zur Durchführung der Validierungen in *Objekttabelle* geführt werden
- Vorteile BOCC+
  - keine unnötigen Rücksetzungen
  - sehr schnelle Validierung
  - frühzeitiges Abbrechen zum Scheitern verurteilter Transaktionen
- Probleme:
  - wie bei BOCC können einzelne Transaktionen verhungern
  - potentiell hohe Rücksetzrate
- Verbesserungsmöglichkeiten:
  - Kombination mit Sperrverfahren
  - Reduzierung der Konfliktwahrscheinlichkeit, z.B. durch
    - geringere Konsistenzebene (Lesetransaktionen werden bei Validierung nicht mehr berücksichtigt)
    - Mehrversionen-Konzept

Kombination von OCC + Sperren

- Ziel: Vorteile beider Verfahrensklassen kombinieren
  - geringe Rücksetzhäufigkeit von Sperrverfahren
  - hohe Parallelität (weniger Sperrwartezeiten) von OCC
- Kombination auf Transaktionsebene
  - optimistisch und pessimistisch synchronisierte Transaktionen
  - für lange und bereits gescheiterte Transaktionen pessimistische Synchronisation => kein Verhungern
- Kombination auf Objekt-Ebene
  - optimistisch und pessimistisch synchronisierte Datenobjekte
  - pessimistische Synchronisation für Hot-Spot-Objekte
- erhöhte Verfahrenskomplexität
  - auch bei pessimistischer Synchronisation Änderungen in privatem Transaktionspuffer
  - erweiterte Validierung: Transaktion scheitert, falls unverträgliche Sperre gesetzt ist
  - (teilweise) pessimistisch synchronisierte Transaktionen:
    - bei EOT optimistische Transaktionen zurücksetzen, die auf X-gesperrte Objekte zugegriffen haben
    - Schreibphase mit anschließender Sperrfreigabe

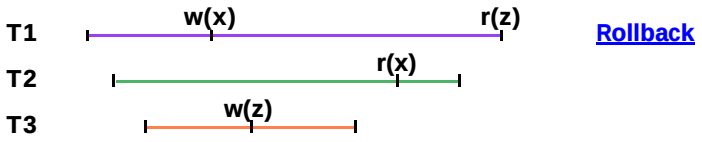
Pufferverwaltung bei OCC



- Änderungen laufender Transaktionen werden nicht in DB verdrängt
  - separater Überlaufbereich auf Externspeicher
  - Transaktionsabbruch erfordert keine Undo-Recovery auf DB
- Einbringen vom Überlaufbereich aufwendig
- analoge Pufferverwaltung bei anderen Synchronisationsverfahren mit Änderungen auf privaten Kopien
- Einbringen verlangt i.a. Synchronisation auf Seitenebene
- optimistische Synchronisation auf Zugriffspfaden i.a. nicht praktikabel

Zeitstempel-Verfahren

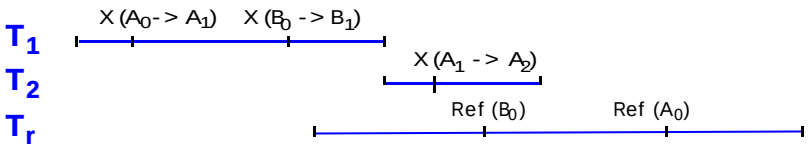
- Grundsätzliche Idee
  - Transaktion bekommt bei BOT einen systemweit eindeutigen Zeitstempel
  - Transaktion hinterläßt den Wert ihres Zeitstempels (als Lese- oder Schreibstempel RTS bzw. WTS) bei jedem Objekt O_i, auf das sie zugreift
  - Prüfung der Serialisierbarkeit ist sehr einfach (Zeitstempelvergleich):
    Transaktion T wird zurückgesetzt, falls
    bei Lesezugriff $ts(T) < WTS$
    bzw. bei Schreibzugriff $ts(T) < max(RTS, WTS)$ *gilt*



- Eigenschaften
  - Serialisierungsreihenfolge einer Transaktion wird bei BOT festgelegt
  - Deadlocks sind ausgeschlossen
  - aber: (viel) höhere Rücksetzraten als pessimistische Verfahren
  - ungelöste Probleme, z.B. wiederholter ROLLBACK einer Transaktion
- Einsetzbarkeit in Verteilten DBS
  - lokale Prüfung der Serialisierbarkeit direkt am Objekt O_i (geringer Kommunikationsaufwand)

Mehrversionen-Konzept

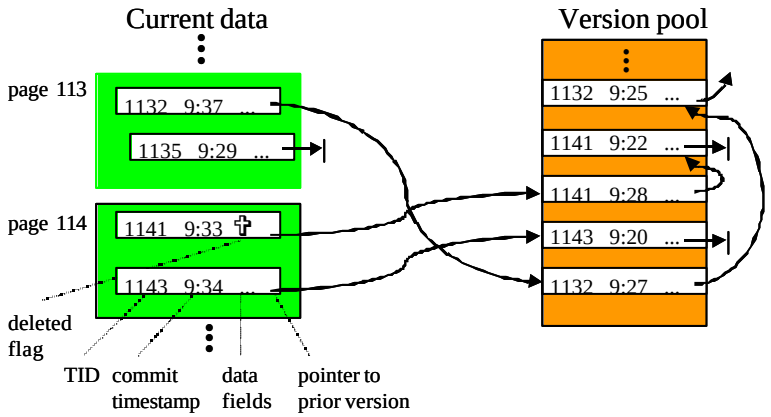
- jede Änderung erzeugt neue Objektversion
- Lesetransaktionen sehen den bei ihrem BOT gültigen DB-Zustand => reihenfolgeerhaltende Serialisierbarkeit
- Lese-Transaktionen brauchen bei Synchronisation nicht mehr berücksichtigt zu werden



  - keine Blockierungen und Rücksetzungen für Lesetransaktionen, dafür ggf. Zugriff auf veraltete Objektversionen
- Änderungstransaktionen werden untereinander über ein allgemeines Synchronisationsverfahren (Sperren, OCC, ...) synchronisiert
- erheblich weniger Synchronisationskonflikte
- zusätzlicher Speicher- und Wartungsaufwand
  - Versionenpool-Verwaltung
  - Auffinden von Versionen
  - Garbage Collection
- Nutzung in kommerziellen DBS (z.B. Oracle)

Versionenpool-Verwaltung

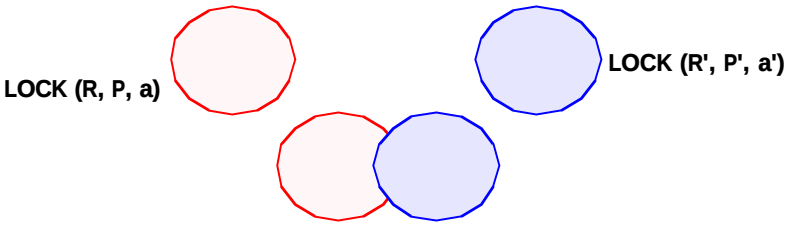
- Realisierungsmöglichkeit
  - Änderungstransaktionen kennzeichnen geänderte Objekte mit Commit-Zeitstempel
  - alte Objektversionen kommen in Versionenpool; Verkettung der Versionen eines Objektes
  - Lesetransaktionen greifen auf jüngste Objektversionen mit Zeitstempel kleiner dem Transaktionsbeginn zu
  - Objektversion V kann freigegeben werden, sobald eine jüngere Version V_j existiert mit Zeitstempel kleiner dem Beginn der ältesten Lesetransaktion im System
  - Reihenfolge der Objektversionen im Pool entspricht den Nachfolger-Zeitstempeln



Logische Sperren (Prädikate)

- Verhütung des Phantomproblems ⇒ Prädikatssperren
- Form: LOCK(R, P, a), UNLOCK (R, P)

  R Relationenname, P Prädikat, a ∈ {read, write}
- Wie kann Konflikt zwischen Prädikaten festgestellt werden?



1. Wenn R ≠ R', kein Konflikt
2. Wenn a = read und a' = read, kein Konflikt
3. Wenn P(t) ∧ P'(t) = TRUE für irgendein t, dann besteht ein Konflikt

- im allgemeinen Fall unentscheidbar, selbst mit eingeschränkten arithmetischen Operatoren

Bsp.: T1: LOCK (PERS, ALTER > 50, read)
T2: LOCK (PERS, PNR = 4711, write)

Prädikatsperren (2)

- pessimistische Entscheidungen => Einschränkung d. Parallelität
  - Sonderfall: P = TRUE entspricht einer Relationensperre
- sehr aufwendige Entscheidungsprozedur mit vielen Prädikaten
- effizientere Implementierung: Präzisionssperren
- nur gelesene Daten werden durch Prädikat gesperrt, Schreibsperren werden für Tupel gesetzt
  - kein Disjunktheitstest für Prädikate mehr, sondern lediglich Test, ob Tupel ein Prädikat erfüllt
- Datenstrukturen:
  - Prädikatliste: pro Relation werden Lesesperren laufender Transaktionen durch Prädikate beschrieben
  - Update-Liste: geänderte Tupel laufender Transaktionen
- Leseanforderung (Prädikat P):
  - für jedes Tupel der Update-Liste ist zu prüfen, ob es P erfüllt
  - wenn ja -> Sperrkonflikt
- Schreibanforderung (Tupel T):
  - Schreibsperre wird gewährt, wenn T keines der Leseprädikate erfüllt

Synchronisation von High-Traffic-Objekten

- High-Traffic-Objekte: meist numerische Felder mit aggregierten Informationen
  - z.B. Anzahl freier Plätze, Summe aller Kontostände
- einfachste Lösung der Sperrprobleme: Vermeidung solcher Felder beim DB-Entwurf
- Alternative: Nutzung von semantischem Wissen zur Synchronisation wie Kommutativität von Änderungsoperationen auf solchen Feldern
- Bsp.: Inkrement-/Dekrement-Operation

	R	X	Inc/Dec
R			
X			
Inc/Dec			

- Problem: Inkrementieren/Dekrementierung erfordert i.a. vorheriges Lesen des Objektes (R-Sperre jedoch inkompatibel mit Inc/Dec-Sperre)

IMS Fast Path - Ansatz

- Spezielle Operationen für High-Traffic-Objecte:

  V E R I F Y *#Plätze >= Anforderung*

  M O D I F Y *#Plätze := #Plätze - Anforderung*

- quasi-optimistische Synchronisation:
  - zunächst werden keine Sperren gesetzt
  - Änderungen werden nicht direkt vorgenommen, sondern nur in 'work-to-do-list' vermerkt
  - bei EOT Validierung- und Schreibphase: Überprüfung, ob VERIFY-Prädikate noch erfüllt sind (geringe Rücksetzwahrscheinlichkeit) sowie Inkrement/Dekrement vornehmen
- Sperren werden nur für Dauer der EOT-Behandlung gehalten
- weit geringere Konfliktgefahr als bei normalen Schreibsperren

Escrow-Ansatz

- Deklaration von High-Traffic-Attributen als Escrow-Felder
- Benutzung spezieller Operationen auf Escrow-Feldern
  - Anforderung einer bestimmten Wertemenge:

    *IF **ESCROW (field=F1, quantity=C1, test=(condition))***
    THEN 'continue with normal processing'
    ELSE 'perform exception handling'
  - Benutzung der reservierten Wertmengen: ***USE (field=F1, quantity=C2)***
- optionale Spezifizierung eines Bereichtests bei Escrow-Anforderung
- wenn Anforderung erfolgreich ist, wird garantiert, daß Prädikat nachträglich nicht mehr invalidiert wird (keine spätere Validierung/Zurücksetzung)
- aktueller Wert eines Escrow-Feldes ist unbekannt, wenn laufende Transaktionen Reservierungen angemeldet haben

=> Führen eines Werteintervalls, das alle möglichen Werte nach Abschluß der laufenden Transaktionen umfaßt

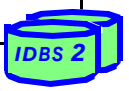
- für Wert Q_k des Escrow-Feldes k gilt: $LO_k <= INF_k <= Q_k <= SUP_k <= HI_k$
- Anpassung von INF, Q, SUP bei Anforderung, Commit und Rollback einer Transaktion

Escrow-Ansatz (2)

- Beispiel: Zugriffe auf Feld mit LO=0, HI=500 (Anzahl freier Plätze)

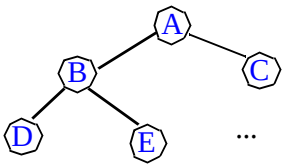
Anforderungen/Rückgaben				Werteintervall		
T1	T2	T3	T4	INF	Q	SUP
				15	15	15
-5						
	-8					
		+4				
			-3			
commit						
		commit				
	rollback					

- Durchführung von Bereichstests bezüglich des Werteintervalls
- Minimal-/Maximalwerte (LO, HI) dürfen nicht überschritten werden
- hohe Parallelität ändernder Zugriffe möglich
- Nachteile:
  - spezielle Programmierschnittstelle
  - tatsächlicher Wert ggf. nicht abrufbar



Sperren in B*-Bäumen

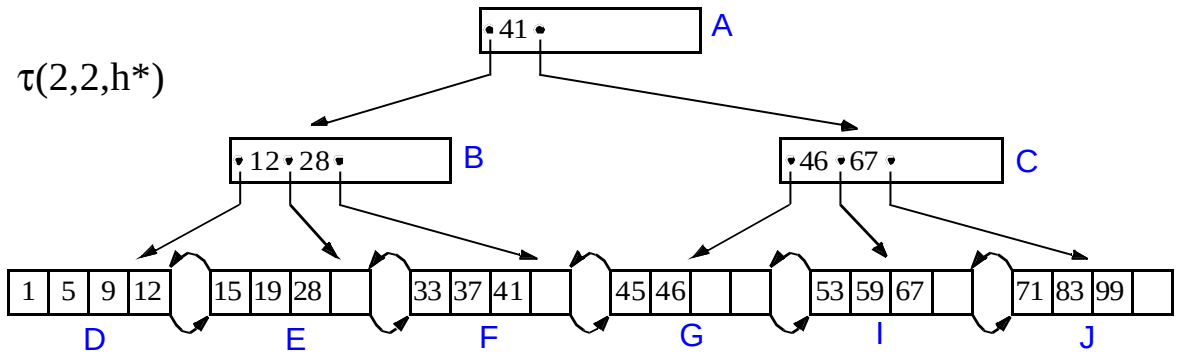
- (lange) Seitensperren führen zu inakzeptablen Behinderungen
  - v.a. Wurzelknoten und Zwischenknoten werden zu Engpässen
  - zahlreiche Einträge auch pro Blatt
- Nutzung der Baum-Charakteristika
  - alle Operationen starten bei Wurzel
  - Zwischenknoten im Baum steuern nur die Suche nach Blattknoten
  - nur Inserts/Deletes die Baumstruktur ändern, erfordern exklusive Sperren des betroffenen Pfades von der Wurzel
- Verwendung eines Baumsperrprotokolls
  - Seite (außer der Wurzel) kann erst gesperrt werden, wenn Vorgänger gesperrt ist („lock coupling“)
  - Sperre auf Vorgänger kann freigegeben werden, wenn seine spätere Änderung - z.B. wegen Split-Vorgängen - ausgeschlossen werden kann
  - garantiert Serialisierbarkeit trotz fehlender Zweiphasigkeit



lock (A), lock (B), unlock (A), lock(D)

- Erweiterung: Verwendung langer Sperren nur für Schlüssel (-intervalle) + kurze Seitensperren

Sperren in B*-Bäumen: Beispiel



1. Suche 37

2. Insert 27

3. Insert 10

Methoden zur Leistungsbewertung von DBS

1.) Kostenformeln
- nur für grobe Abschätzungen brauchbar (back-of-the-envelope calculations)

2.) Analytische Modelle
- z.B. Warteschlangenmodelle
- oft starke Vereinfachungen erforderlich

3.) Simulationen
- sehr geeignet zum Vergleich verschiedener Realisierungsalternativen (Verfahren können direkt implementiert werden)
- synthetische Lasten vs. Trace-getrieben (reale Lasten)

4.) Benchmarks
- Messungen mit DBS-Prototyp bzw. realem DBS
- Bewertung / Vergleich bestimmter Systeme

Wahrscheinlichkeit von Sperrkonflikten und Deadlocks

- Annahmen
  - m Datenbankobjekte
  - n parallele Transaktionen (Multiprogramming Level)
  - k Objektzugriffe pro Transaktion
  - nur exklusive Sperren
  - Gleichverteilung der DB-Zugriffe

- mittlere Anzahl von Sperren pro Transaktion: $\bar{L} \approx \frac{k}{2}$

- Wahrscheinlichkeit eines Sperrkonfliktes für den i-ten Objektzugriff einer Transaktion T:

$$P_C = \frac{\text{\#Sperren anderer Transaktionen}}{\text{\#DB-Objekte, die nicht von T gesperrt}} =$$

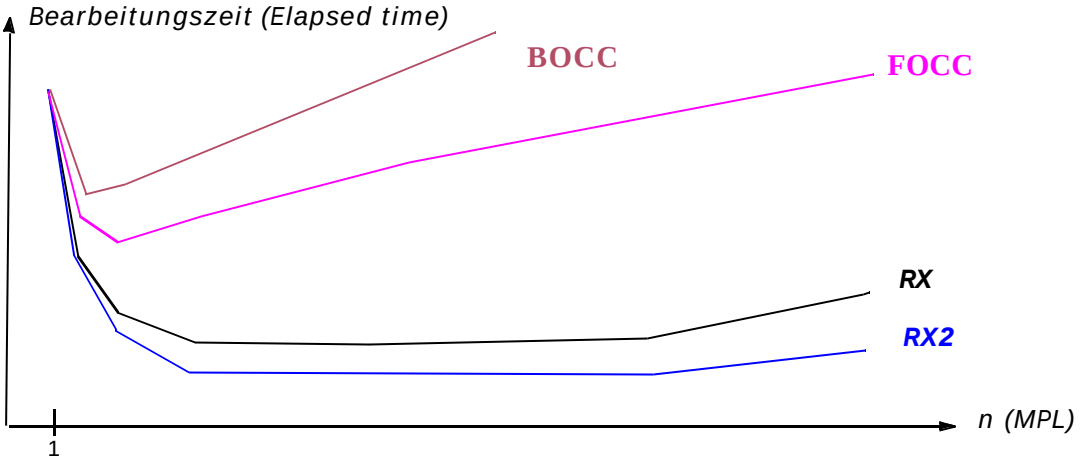
- Wahrscheinlichkeit, daß eine Transaktion einen Sperrkonflikt erfährt:

$$P_W \approx k \cdot P_C \approx$$

- Deadlock-Wahrscheinlichkeit zwischen zwei Transaktionen T1 und T2:

Pr [T1 -> T2) • Pr [T2 -> T1] • #Kandidaten für T2 =

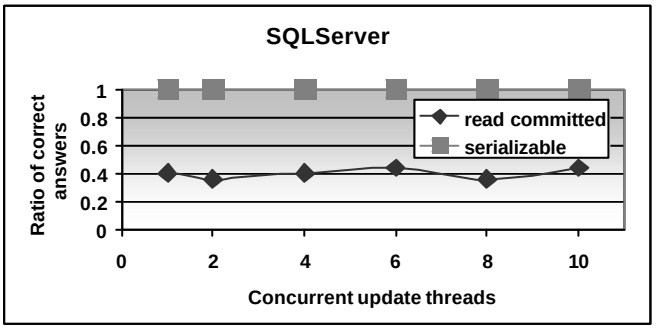
Bewertung von Synchronisationsverfahren mit Trace-basierter Simulation

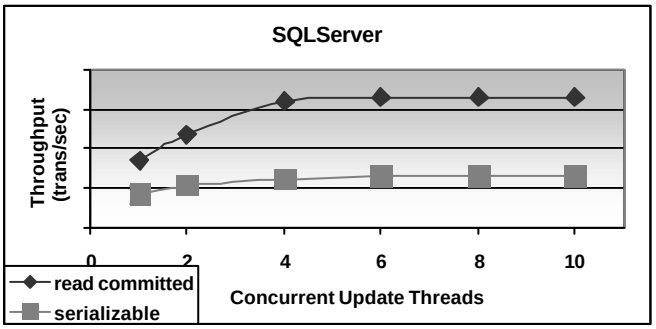


- sehr geringe Parallelität → keine effektive Nutzung der Ressourcen
- geringe Parallelität → bester Durchsatz, nicht notwendigerweise kürzeste Antwortzeiten
- pessimistische Methoden gewinnen: Blockierung vermeidet häufig Deadlocks
- optimistische Methoden geraten leicht in ein Thrashing-Verhalten
- RX2 (kurze Lesesperren) reduziert effektiv den Wettbewerb um gemeinsam genutzte Daten

Benchmark-Ergebnisse (Shasha, 2002)

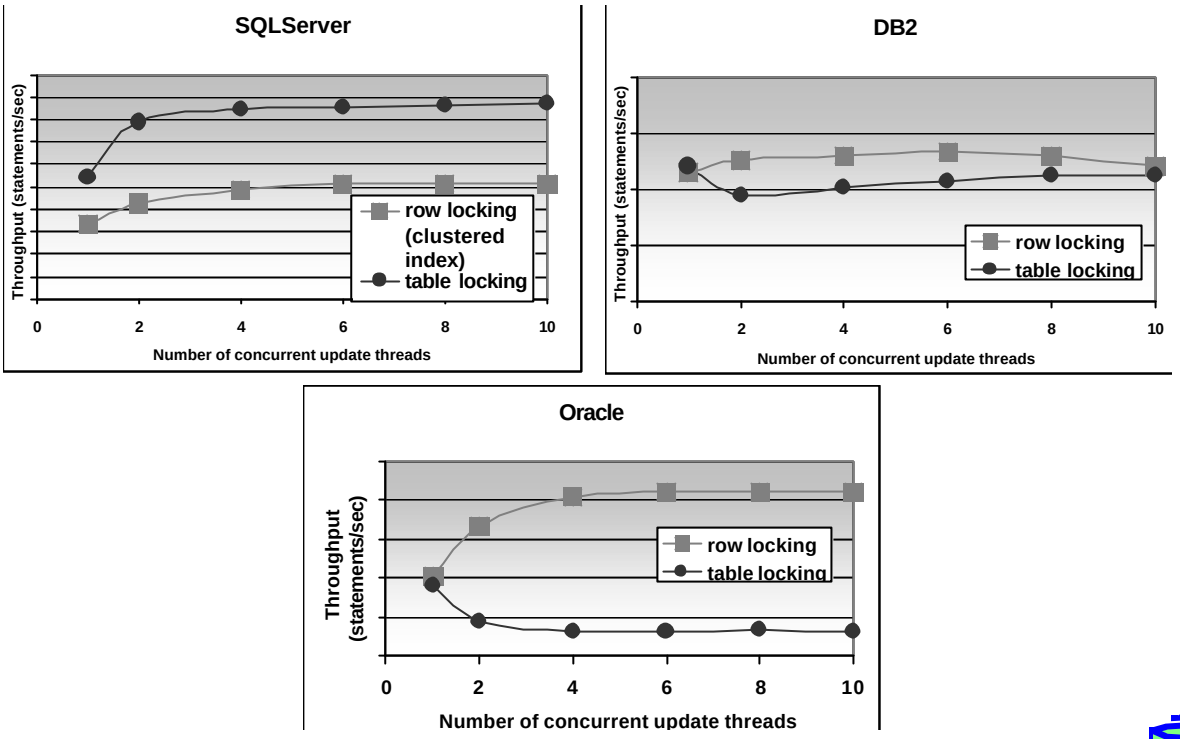
- Serializable vs. Read Committed (RX2)



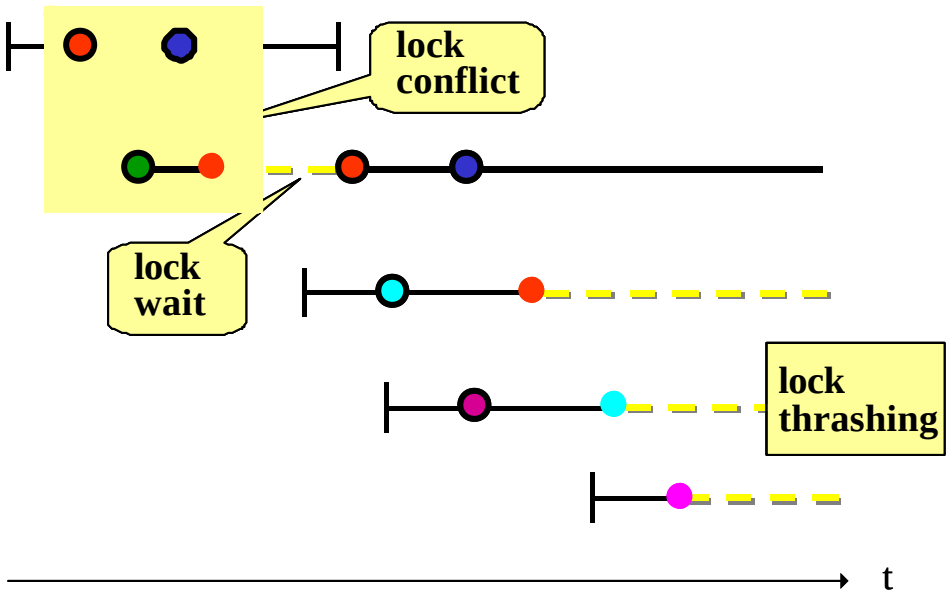


Benchmark-Ergebnisse (2)

- Table vs. row (record) locking

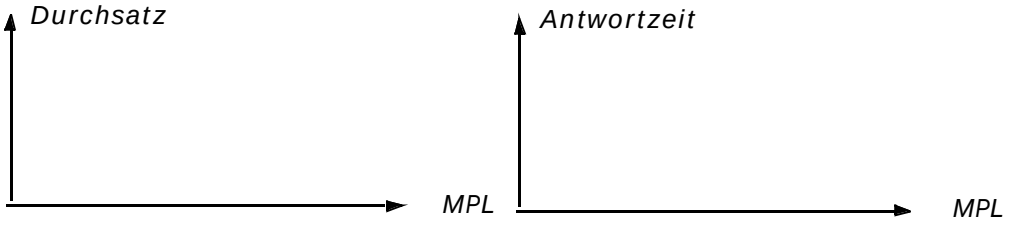


Lock Thrashing



Dynamische MPL-Kontrolle

- Parallelitätsgrad (MPL) hat wesentlichen Einfluß auf Leistungsverhalten, insbesondere Umfang an Konflikten bzw. Rücksetzungen
  - Gefahr von "lock thrashing" bei Überschreiten eines kritischen MPL-Wertes



- statische MPL-Einstellung unzureichend: wechselnde Lastsituationen, mehrere Transaktionstypen
- Idee: dynamische Einstellung des MPL zur Vermeidung von "lock thrashing"
- ein möglicher Ansatz: Nutzung einer Konfliktrate:

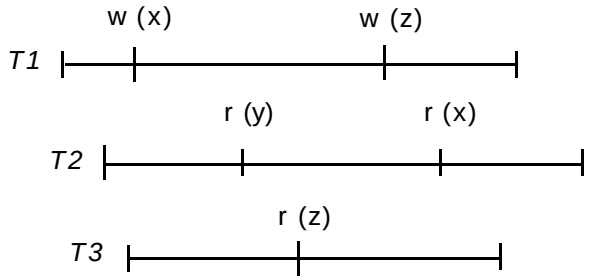
  Konfliktrate = # gehaltener Sperren / #Sperren nicht-blockierter Transaktionen

  kritischer Wert: ca. 1,3 (experimentell bestimmt)
  - Zulassung neuer Transaktionen nur, wenn kritische Wert noch nicht erreicht ist
  - bei Überschreiten erfolgt Abbrechen von Transaktionen

Zusammenfassung

- Korrektheitskriterium der Synchronisation: Serialisierbarkeit
- Sperrverfahren am universellsten einsetzbar
  - Zweiphasen-Sperrprotokolle
  - reine OCC- und Zeitstempelverfahren erzeugen i.a. zu viele Rücksetzungen
  - Hierarchische Sperrverfahren erlauben Begrenzung des Verwaltungs-Overheads
- generelle Optimierungen:
  - reduzierte Konsistenzebene
  - Mehrversionen-Ansatz
- „harte“ Synchronisationsprobleme durch Hot Spots und lange (Änderungs-) Transaktionen
  - möglichst Vermeidungsstrategie anwenden
  - Spezialprotokolle nutzen semantisches Wissen über Operationen/Objekte zur Reduzierung von Synchronisationskonfliken
  - allerdings: ggf. Erweiterung der Programmierschnittstelle, begrenzte Einsetzbarkeit, Zusatzaufwand
- dynamische MPL-Kontrolle kann Lock Thrashing vermeiden

Übungsbeispiel



- RX: Wait (T1, z), Wait (T2, x): T3 < T1 < T2
- BOCC:
- FOCC:
- Zeitmarken:
- Mehrversionen-RX:

Übungen

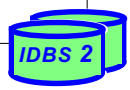
- Escrow-Beispiel:
  Ein Zähler-Objekt x mit Initialwert x=100 habe ein zulässiges Minimum LO=0 und keine obere Beschränkung. Geben Sie die Entwicklung des Wertebereichs (Inf, Q, Sup) für folgenden Schedule der 3 Transaktionen T1, T2, T3 an:
  decrement (T1, x, 60), increment (T2, x, 10), increment (T1, x, 20), decrement (T3, x, 50), increment (T2, x, 10), rollback (T2), commit (T1), commit (T3)
- Mit welchen Synchronisationsverfahren kann das Phantom-Problem gelöst werden?
- Wie kann der Umfang des Versionenpools bei Mehrversionen-Synchronisation begrenzt werden?
- Welchen Einfluß hat die Transaktionslänge auf
  - die Dauer einer Sperrblockierung
  - die Wahrscheinlichkeit von Sperrkonflikten
  - die Wahrscheinlichkeit von Deadlocks
  - die Wahrscheinlichkeit von Rücksetzungen bei FOCC?
- Wie kann die Dauer von Sperren reduziert werden?
- Welche Synchronisationsverfahren erlauben das gleichzeitige (erfolgreiche) Ändern eines Objektes durch mehrere Transaktionen?

4. Logging und Recovery: Grundlagen

Einführung
- Fehlermodell
- Recovery-Arten

Logging-Strategien
- physisches/logisches und Zustands-/Übergangs-Logging
- Eintrags- vs. Seiten-Logging
- Aufbau der Log-Datei

Klassifikation von Recovery-Verfahren
- Einbringstrategie
- Zusammenspiel mit der DBS-Pufferverwaltung (Seitenersetzung, Ausschreibstrategie)
- Commit-Behandlung, Gruppen-Commit
- Sicherungspunkte (Checkpoints)



DB-Recovery

automatische Behandlung aller erwarteten Fehler durch das DBVS

Voraussetzung: Sammeln redundanter Informationen während des normalen Betriebes (Logging)

Zielzustand nach erfolgreicher Recovery:

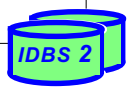
Durch die Recovery-Aktionen ist der jüngste Zustand vor Erkennen des Fehlers wiederherzustellen, der allen semantischen Integritätsbedingungen entspricht, der also ein möglichst aktuelles, exaktes Bild der Miniwelt darstellt

Transaktionsparadigma verlangt:
- Alles-oder-Nichts-Eigenschaft von Transaktionen
- Dauerhaftigkeit erfolgreicher Änderungen

Forward-Recovery i.a. nicht anwendbar
- Fehlerursache häufig falsche Programme, Eingabefehler u.ä.
- durch Fehler unterbrochene Transaktionen sind zurückzusetzen (Backward Recovery)

Fehlerarten

Auswirkung eines Fehlers auf	Fehlertyp	Fehlerklassifikation
eine Transaktion	- Verletzung von Systemrestriktionen • Verstoß gegen Sicherheitsbestimmungen • übermäßige Betriebsmittelanforderungen - anwendungsbedingte Fehler • z.B. falsche Operationen und Werte	*Transaktionsfehler*
mehrere Transaktionen	- geplante Systemschließung - Schwierigkeiten bei der Betriebsmittelvergabe • Überlast des Systems • Verklemmung mehrerer Transaktionen	
alle Transaktionen (das gesamte Systemverhalten)	- Systemzusammenbruch mit Verlust der Hauptspeicherinhalte • Hardware-Fehler • falsche Werte in kritischen Tabellen	*Systemfehler*
	- Zerstörung von Sekundärspeichern	*Gerätefehler*
	- Zerstörung des Rechenzentrums	*Katastrophen*



Recovery-Arten

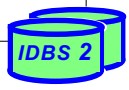
1. Zurücksetzen (Undo) einzelner Transaktionen im laufenden Betrieb (Transaktionsfehler, Deadlock, etc.)
   - vollständiges Zurücksetzen auf Transaktionsbeginn (Standard) bzw.
   - partielles Zurücksetzen auf Rücksetzpunkt (Savepoint) innerhalb der Transaktion

2. Crash-Recovery nach Systemfehler

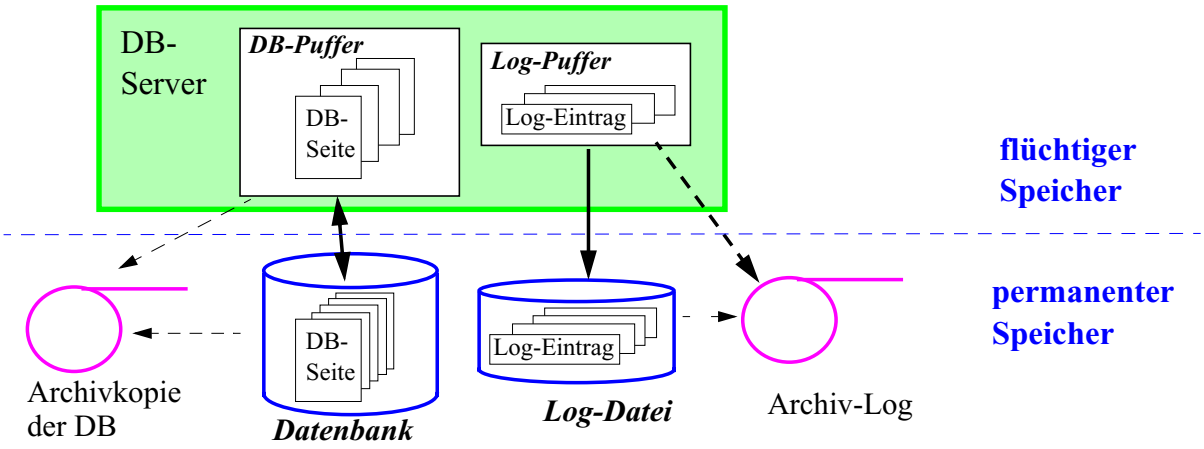
   Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustandes:
   - Redo für erfolgreiche Transaktionen (Wiederholung verlorengegangener Änderungen)
   - Undo aller durch Ausfall unterbrochenen Transaktionen (Entfernen derer Änderungen aus der permanenten DB)

3. Platten-Recovery nach Gerätefehler
   - Spiegelplatten bzw.
   - vollständiges Wiederholen (Redo) aller Änderungen auf einer Archivkopie

4. Katastrophen-Recovery
   - stark verzögerte Fortsetzung der Verarbeitung an repariertem/neuem System mit Archivkopie (Datenverlust!) oder
   - Nutzung einer aktuellen DB-Kopie an einem geographisch separierten System



Systemkomponenten



Pufferung von Log-Daten im Hauptspeicher (Log-Puffer)

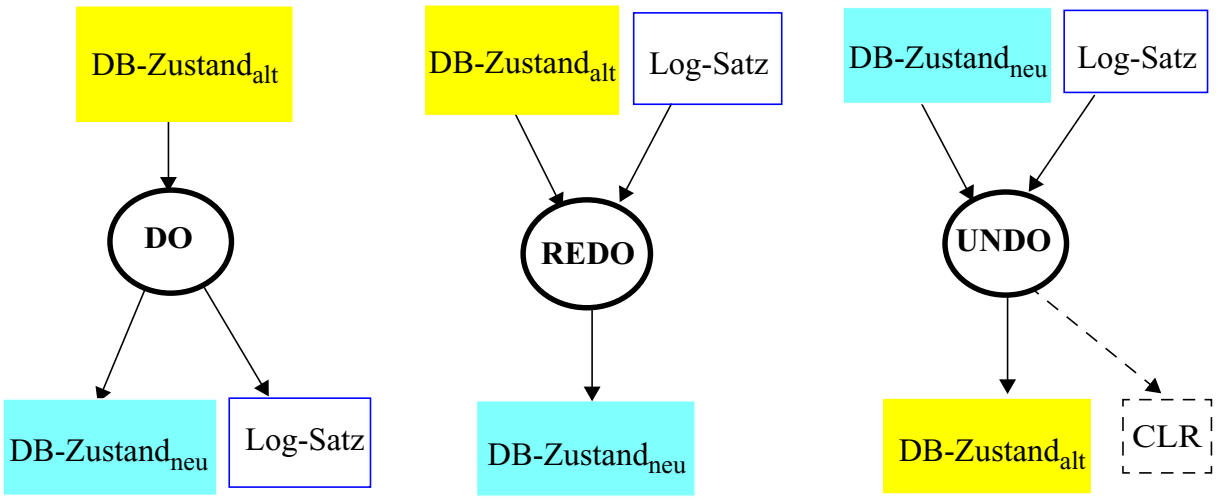
- Ausschreiben spätestens am Transaktionsende ("Commit")

(temporäre) Log-Datei zur Behandlung von Transaktions- und Systemfehler: DB + Log => DB

Behandlung von Gerätefehlern: Archivkopie + Archiv-Log => DB

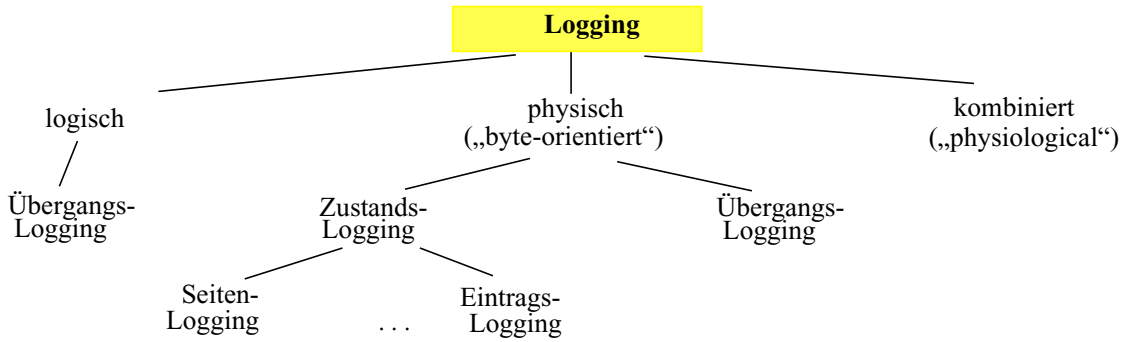
Do-Redo-Undo-Prinzip

Logging (Protokollierung von Änderungen) im Normalbetrieb Voraussetzung für Recovery



Compensation
Log Record

Logging

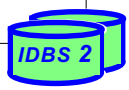


Logisches Logging

- Protokollierung der ändernden DML-Befehle mit ihren Parametern
- Voraussetzung: nach einem Systemausfall müssen auf der permanenten Datenbank DML-Operationen ausführbar sein, d.h. sie muß wenigstens speicherkonsistent sein (Aktionskonsistenz)
  => *indirekte Seitenzuordnung erforderlich*

Physisches Logging

- Log-Granulat: Seite vs. Eintrag/Satz
- Zustands-Logging: alte Zustände (Before-Images) und neue Zustände (After-Images) geänderter Objekte werden auf die Protokolldatei geschrieben
- Übergangs-Logging: Protokollierung der Differenz zwischen Before- und After-Image
- physisches Logging ist bei direkten und indirekten Einbringstrategien anwendbar



Logging: Anwendungsbeispiel

Änderungen bezüglich einer Seite A:

1. Ein Objekt a wird in Seite A eingefügt
2. In A wird ein bestehendes Objekt b_{alt} nach b_{neu} geändert

Zustandsübergänge von A: $A_1 \xrightarrow{1.} A_2 \xrightarrow{2.} A_3$

	logisch	*physisch*
Zustände		Protokollierung der Before- und After-Images 1. 2.
Übergänge	Protokollierung der Operationen mit Parameter 1. 2.	Differenzen-Logging 1. 2.

Rekonstruktion von Seiten beim Differenzen-Logging:

A_1 als Anfangs- oder A_3 als Endzustand seien verfügbar. Es gilt:

A_2 =	A_2 =
A_3 =	A_1 =
Redo-Recovery	Undo-Recovery

Physiologisches Logging

Probleme logischer und physischer Logging-Verfahren

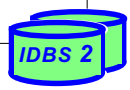
- logisches Logging: für Update-in-Place nicht anwendbar
- physisches, "byte-orientiertes" Logging: aufwendig und unnötig starr v.a. bezüglich Lösch- und Einfügeoperationen

Kombination physische/logische Protokollierung: Physical-to-a-page, Logical-within-a-page

- Protokollierung von elementaren Operationen innerhalb einer Seite
- jeder Log-Satz bezieht sich auf 1 Seite
- mit Update-in-Place verträglich

Beispiel

logisches Logging	*physisches Eintrags-Logging*	*physiologisches Logging*
	Datenseite D	
insert R: a1, a2, ...	Indexseite I1	
	Indexseite I2	



Bewertung der Logging-Strategien

	Logging-Aufwand	Restart-Aufwand
Seitenzustands-Logging		
Seitenübergangs-Logging		
Eintrags-Logging / physiologisches Logging		
logisches Logging		

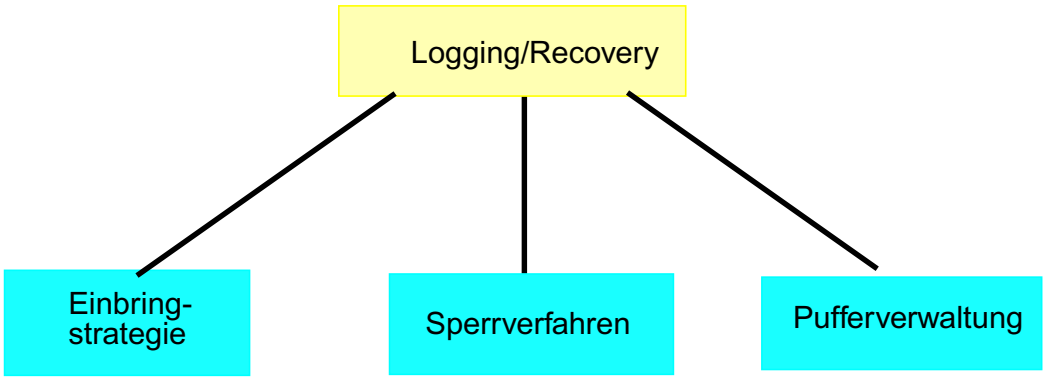
-- sehr hoch - hoch + gering ++ sehr gering

Vorteile von *Eintrags-Logging* gegenüber Seiten-Logging:

- geringerer Platzbedarf
- weniger Log-E/As
- erlaubt bessere Pufferung von Log-Daten (Gruppen-Commit)
- unterstützt feine Synchronisationsgranulate (Seiten-Logging => Synchronisation auf Seitenebene)

jedoch: Recovery komplexer als mit Seiten-Logging

Abhängigkeiten zwischen Systemkomponenten



Logging/Recovery <-> Sperrverwaltung

- Log-Granulat muß i.a. kleiner oder gleich dem Sperrgranulat sein!

Logging/Recovery <-> Einbringstrategie für Änderungen

- direkt (NonAtomic, Update-in-Place)
- indirekt (Atomic), Bsp.: Schattenspeicherkonzept

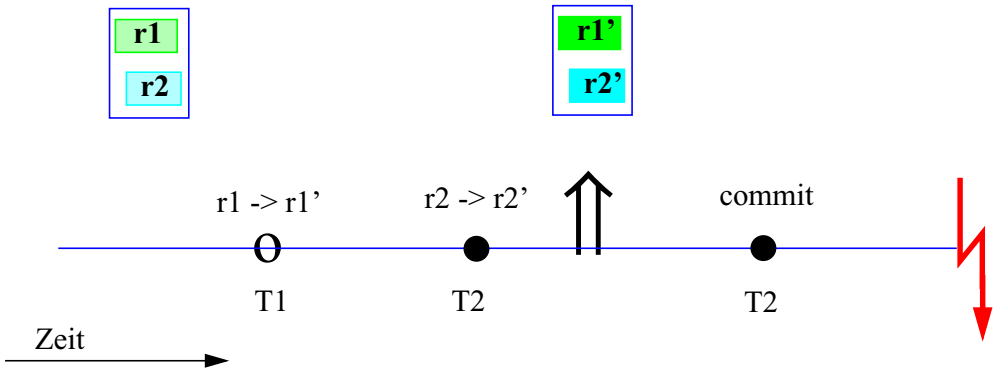
Logging/Recovery <-> Systempufferverwaltung

- Verdrängen 'schmutziger' Seiten (Steal vs. NoSteal)
- Ausschreibstrategie für geänderte Seiten (Force vs. NoForce)

Abhängigkeiten zur Sperrverwaltung

Log-Granulat muß i.a. kleiner oder gleich dem Sperrgranulat sein!

Beispiel: Sperren auf Satzebene, Before- bzw. After-Images auf Seitenebene

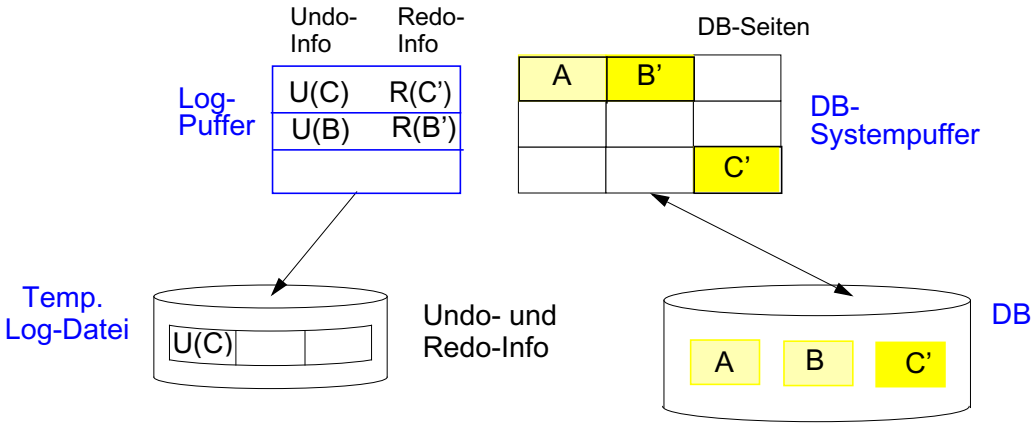


=> Undo (Redo) einer Änderung kann parallel durchgeführte Änderungen derselben Seite überschreiben *(lost update)*

Direkte Einbringstrategien: Update in Place

geänderte Seite wird immer in denselben Block auf Platte zurückgeschrieben

'atomares' Zurückschreiben mehrerer geänderter Seiten ist nicht möglich (NonAtomic)



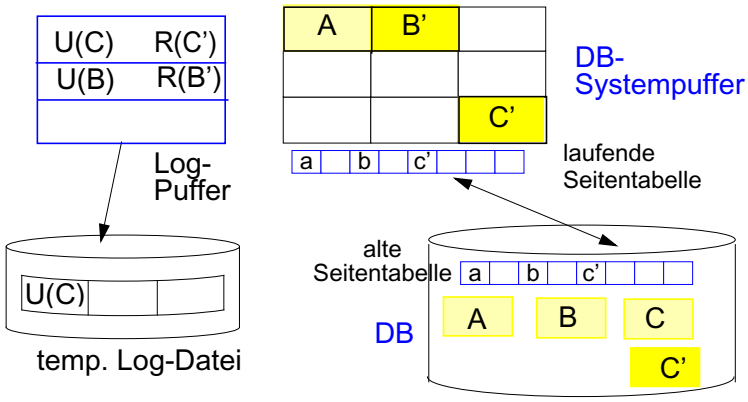
Es sind 2 Prinzipien einzuhalten (Minimalforderung):

1. Undo-Regel: schreibe Undo-Info vor Zurückschreiben unsicherer Änderungen (WAL-Prinzip)
2. Redo-Regel: Ausschreiben der Redo-Info spätestens zu COMMIT

Indirekte Einbringstrategien (Atomic)

Schattenspeicherkonzept (System R, SQL/DS)

- geänderte Seite wird in separaten Block auf Platte geschrieben
- Seitentabelle gibt aktuelle Adresse einer Seite an
- atomares Einbringen mehrerer Änderungen durch Umschalten von Seitentabellen möglich



aktions- oder transaktionskonsistente DB auf Platte (logisches Logging anwendbar)

Schwerwiegende Nachteile:

- aufwendiges Einbringen
- Seitentabelle kann für große DB nicht mehr im Hauptspeicher gehalten werden
- Cluster-Eigenschaften werden zerstört
- Speicherplatzbedarf

Abhängigkeiten zur Pufferverwaltung: Ersetzung 'schmutziger' Seiten

Steal: geänderte Seiten können jederzeit, insbesondere vor Commit der ändernden Transaktion, ersetzt und in permanente DB eingebracht werden

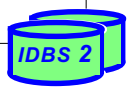
+ große Flexibilität zur Seitenersetzung
- Undo-Recovery vorzusehen (Transaktions-Abbruch, Systemfehler)

Steal erfordert Einhaltung des ***Write-Ahead-Log (WAL)-Prinzips***:

vor dem Einbringen einer schmutzigen Änderung müssen zugehörige Undo-Informationen (z.B. Before-Images) in die Log-Datei geschrieben werden

NoSteal

+ keine Undo-Recovery auf der permanenten DB vorzusehen
- Probleme bei langen Änderungstransaktionen
  (Seiten mit schmutzigen Änderungen dürfen nicht ersetzt werden)



Abhängigkeiten zur Pufferverwaltung: Commit-Behandlung

Force: alle geänderten Seiten werden spätestens beim Commit (vor dem Commit) in die permanente DB durchgeschrieben

+ keine Redo-Recovery nach Rechnerausfall
- hoher Schreibaufwand
- große Systempuffer werden schlecht genutzt
- Antwortzeitverlängerung für Änderungstransaktionen
- Seitensperren

NoForce:

+ kein Durchschreiben der Änderungen bei Commit
+ beim Commit werden lediglich Redo-Informationen in die Log-Datei geschrieben
- Redo-Recovery nach Rechnerausfall

Commit-Regel: bevor das Commit einer Transaktion ausgeführt werden kann, sind für ihre Änderungen ausreichende Redo-Informationen (z.B. After-Images) zu sichern

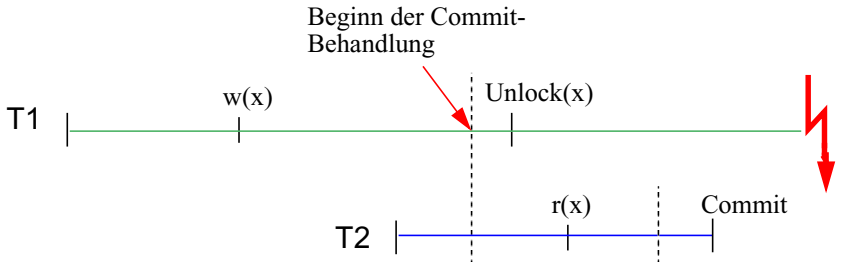
Recovery-Auswirkungen von Ausschreibstrategien

	Steal	NoSteal
Force		
NoForce		

Commit-Behandlung

Änderungen einer Transaktion sind vor Commit zu sichern

- andere Transaktionen dürfen Änderungen erst sehen, wenn Durchkommen der ändernden Transaktion gewährleistet ist (Problem der 'Cascading Aborts')



Zweiphasige Commit-Bearbeitung

- ***Phase 1:*** Wiederholbarkeit der Transaktion sichern:
  Änderungen ggf. noch sichern und Commit-Satz auf Log schreiben
- ***Phase 2:*** Änderungen sichtbarmachen (Freigabe der Sperren)
- Benutzer kann nach Phase 1 vom erfolgreichen Ende der Transaktion informiert werden (Ausgabenachricht)

Bsp.: Commit-Behandlung bei Force/Steal

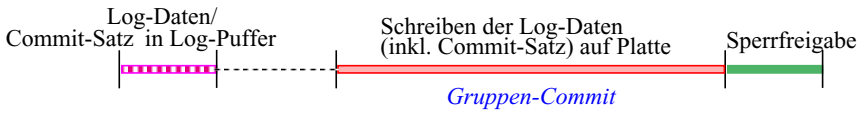
1. Undo-Logging
2. Force (Ausschreiben der geänderten Seiten)
3. Redo-Logging
4. Schreiben Commit-Satz

Gruppen-Commit

Log-Datei potentieller Leistungsengpaß

- pro Änderungstransaktion wenigstens 1 Log-E/A
- max. ca. 60 sequentielle Schreibvorgänge pro Sekunde (1 Platte)

Gruppen-Commit: gemeinsames Schreiben der Log-Daten von mehreren Transaktionen



- Pufferung der Log-Daten in Log-Puffer (1 oder mehrere Seiten)
- Voraussetzung: Eintrags-Logging
- Ausschreiben des Log-Puffers erfolgt, wenn er voll ist bzw. Timer abläuft
- nur geringe Commit-Verzögerung

erlaubt Reduktion auf 0.1 - 0.2 Log-E/As pro Transaktion

=> Durchsatzverbesserung

dynamische Festsetzung des Timer-Wertes durch DBVS wünschenswert

Aufbau der (temporären) Log-Datei

i.a. sequentielle Datei

- Schreiben neuer Protokolldaten an das aktuelle Dateiende
- doppelte Speicherung (Duplex-Logging)

übliche Satzarten:

- Begin-of-Transaction (BOT), Commit-Satz, Rollback-Satz
- Undo-Informationen (z.B. 'Before Images')
- Redo-Informationen (z.B. 'After Images')
- Checkpoint-Sätze

jeder Log-Satz hat eindeutige Adresse: *LSN (Log Sequence Number)*

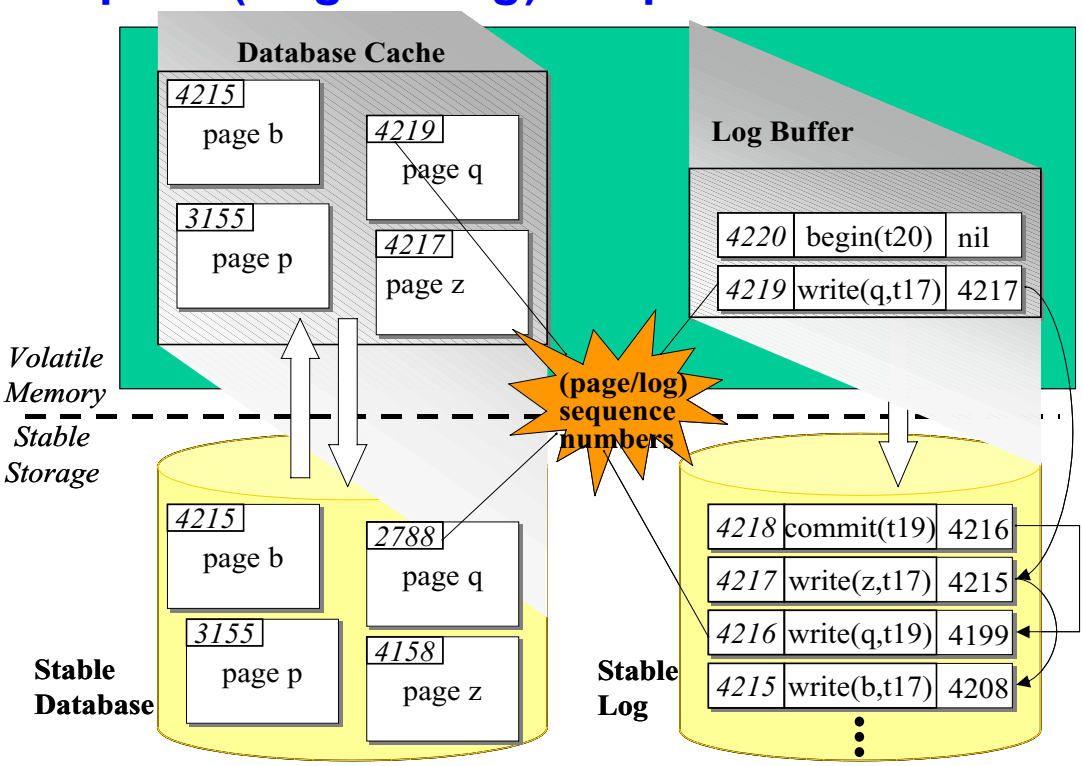
- monoton wachsend !

Log-Sätze einer Transaktion werden rückwärts verkettet (für Transaktions-Undo)

jede DB-Seite hat im Seitenkopf ein Feld *PageLSN* mit der LSN derjenigen Änderung, die zuletzt durchgeführt wurde

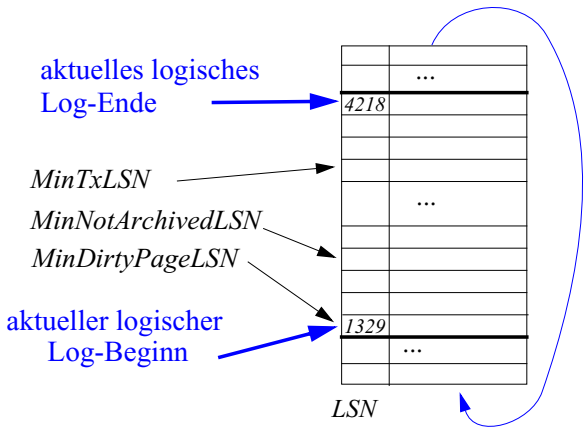
- entspricht monoton wachsender Versionsnummer
- erlaubt Entscheidung darüber, ob ein Log-Satz bei der Recovery anzwenden ist

Beispiel: (Page / Log) Sequence Numbers



Log-Datei (2)

Ringpuffer-Organisation der Log-Datei



Log-Daten sind für Crash-Recovery nur begrenzte Zeit relevant

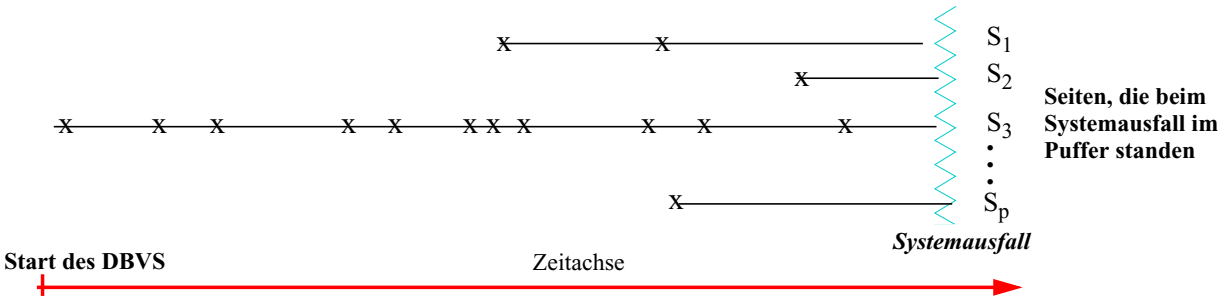
1. Undo-Sätze für erfolgreich beendete Transaktionen werden nicht mehr benötigt
   - pro Transaktion wird LSN ihres BOT-Satzes vermerkt
   - Minimum dieser LSN-Werte laufender Transaktionen (*MinTxLSN*) begrenzt benötigte Undo-Information
2. nach Ausschreiben der Seite in permanente DB wird Redo-Information nicht mehr benötigt
   - pro geänderter Seite im DB-Puffer wird LSN der ersten Änderung nach Lesen von Platte vermerkt
   - Minimum dieser LSN-Werte (*MinDirtyPageLSN*) begrenzt benötigte Redo-Information
3. Protokollsätze für Platten-Recovery werden auf Archiv-Log gehalten
   - LSN der ältesten noch nicht auf Archiv-Log übertragenen Redo-Information: *MinNotArchivedLSN*

Sicherungspunkte (Checkpoints)

Sicherungspunkt: Maßnahme zur Begrenzung des Redo-Aufwandes nach Systemfehlern (nur für NoForce erforderlich)

ohne Sicherungspunkte müßten potentiell alle Änderungen seit Start des DBVS wiederholt werden

besonders kritisch: Hot-Spot-Seiten



Log-Datei:

- BEGIN_CHKPT-Satz
- Checkpoint-Informationen
- END_CHKPT-Satz

Log-Adresse des letzten Checkpoint-Satzes wird in *spezieller Restart-Datei* geführt

Arten von Sicherungspunkten

Direkte Sicherungspunkte

- alle geänderten Seiten im DB-Puffer werden auf die permanente DB ausgeschrieben
- Redo-Recovery beginnt bei letztem Checkpoint
- Nachteil: lange 'Totzeit' des Systems, da während des Sicherungspunktes keine Änderungen durchgeführt werden können
- Problem wird durch große Hauptspeicher verstärkt
- *Transaktionskonsistente* oder *aktionskonsistente* Sicherungspunkte

Indirekte/Unscharfe Sicherungspunkte (Fuzzy Checkpoints)

- kein Hinauszwingen geänderter Seiten
- nur Statusinformationen (Pufferbelegung, Menge aktiver Transaktionen, offene Dateien etc.) werden in Log geschrieben
- sehr geringer Checkpoint-Aufwand
- i.a. Redo-Informationen vor letztem Sicherungspunkt noch zu berücksichtigen
- Sonderbehandlung von Hot-Spot-Seiten erforderlich

Force kann als spezieller direkter Checkpoint-Typ aufgefaßt werden
(nur Seiten einer Transaktion werden ausgeschrieben => transaktionsorientiert)

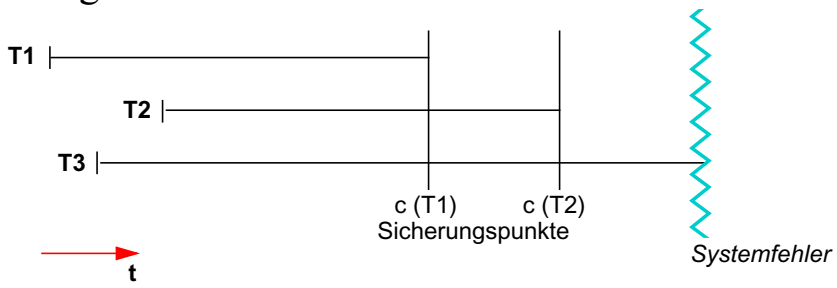
Transaktionsorientierte Sicherungspunkte

TOC: Transaction Oriented Checkpoint ≡ Force

Commit-Behandlung erzwingt Ausschreiben aller geänderten Seiten der Transaktion aus dem Puffer

Übernahme aller Änderungen in die DB

Vermerk in Log-Datei



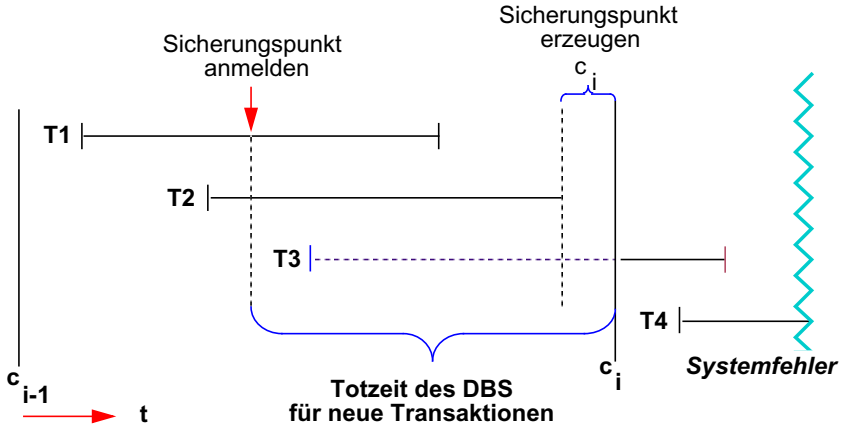
kein atomares Ausschreiben mehrerer Seiten möglich

- zumindestens bei direkter Seitenzuordnung Undo-Recovery vorzusehen (Steal)

Abhängigkeit: NonAtomic, Force => Steal

Transaktionskonsistente Sicherungspunkte

TCC = Transaction Consistent Checkpoints (logisch konsistent)



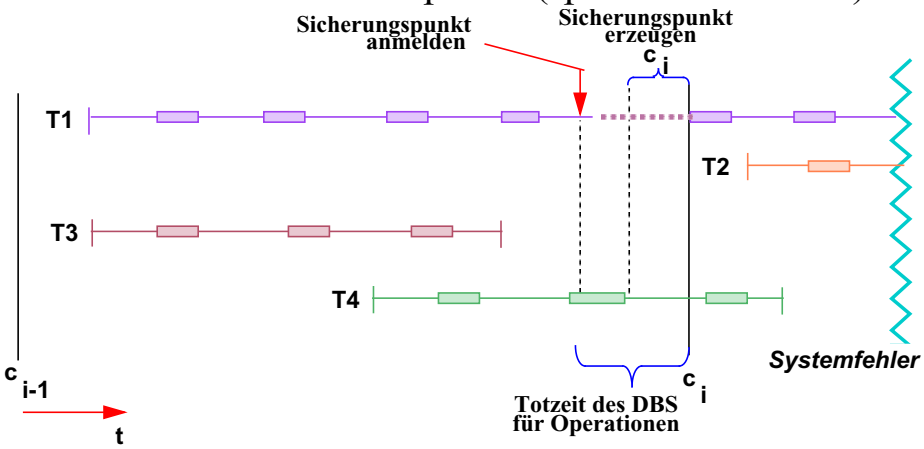
Ausschreiben zu verzögern bis zum Ende aller aktiven Änderungstransaktionen

neue Änderungstransaktionen müssen warten bis Sicherungspunkt beendet ist

Crash-Recovery startet bei letztem Sicherungspunkt

Aktionskonsistente Sicherungspunkte

ACC = Action Consistent Checkpoints (speicherkonsistent)



keine Änderungs-DML-Befehle während Checkpoint

geringere Totzeiten als bei TCC, dafür Verminderung der Qualität der Sicherungspunkte

Crash-Recovery wird nicht durch letzten Sicherungspunkt begrenzt

Abhängigkeit: ACC => Steal

Fuzzy Checkpoints

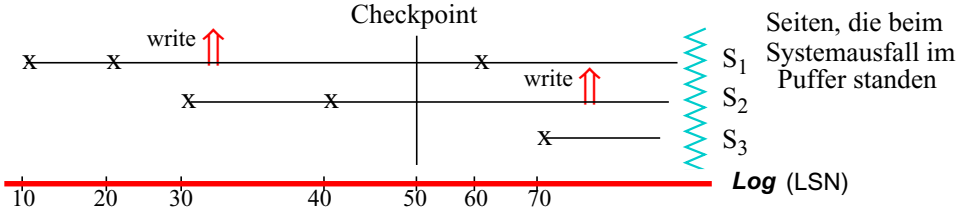
DB auf Platte bleibt 'fuzzy', nicht aktionskonsistent

=> nur bei Update-in-Place (NonAtomic) relevant

Bestimmung der Log-Position, an der Redo-Recovery beginnen muß

- bei Änderung einer Seite im Puffer wird ein Log-Satz erzeugt
- Pufferverwalter vermerkt sich zu jeder geänderten Seite Adresse (LSN) des Log-Satzes der ersten Änderung seit Einlesen von Platte
- Redo-Recovery nach Rechnerausfall beginnt bei *MinDirtyPageLSN*

Startposition wird in Checkpoint-Information vermerkt (daneben laufende Transaktionen, geänderte Seiten, ...)



geänderte Seiten werden asynchron ausgeschrieben

- ggf. Kopie der Seite anlegen (für Hot-Spot-Seiten)
- Seite ausschreiben
- MinDirtyPageLSN anpassen/zurücksetzen

Systembeispiel Oracle†

DBW-Prozeß (database writer) für asynchrone Schreibvorgänge

3 Kontrollparameter beeinflussen Log-Umfang und Redo-Dauer

- FAST_START_IO_TARGET: maximale Anzahl DB-Seiten, für die Redo erforderlich sein soll
- LOG_CHECKPOINT_TIMEOUT: Zeitabstand in s zwischen aktuellem Log-Ende und letztem Checkpoint
- LOG_CHECKPOINT_INTERVAL: max. Anzahl von Redo-Blöcken seit letztem Checkpoint

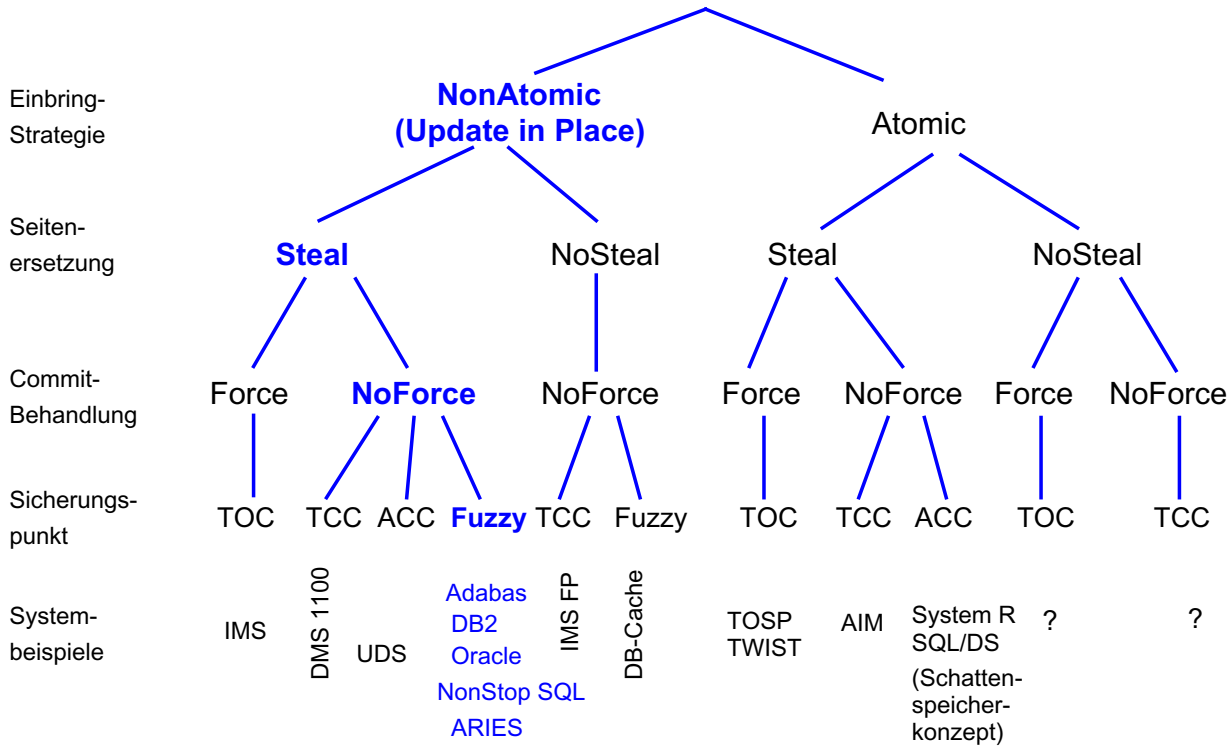
DBW bestimmt periodisch Ziel-LSN (RBA = Redo Byte Address), bis zu der Änderungen geschrieben werden müssen, um Schranken einzuhalten

Messung für OLTP-Last (Puffer: 200.000 4 KB-Seiten)

FAST_START_IO_TARGET	Durchsatz (Transaktionen pro Minute)	Recovery-Dauer (Redo)
disabled	805	4 Min. 34 s
30.000	804	1 Min. 10 s
20.000	798	1 Min. 20 s
10.000	798	49 s
1.000	797	21 s

† T. Lahiri et al.: Fast-Start: Quick Fault Recovery in Oracle. Proc. ACM SIGMOD Conf., May 2001

Klassifikation von DB-Recovery-Verfahren



Zusammenfassung

Fehlerarten: Transaktions-, System- und Gerätefehler

breites Spektrum von Logging- und Recovery-Verfahren

Eintrags-Logging

- ist Seiten-Logging überlegen (geringerer Platzbedarf, weniger E/As, Gruppen-Commit)
- sequentielle Log-Datei (Log-Umfang kann begrenzt werden)

Update-in-Place-Verfahren

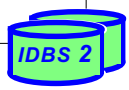
- sind Atomic-Strategien vorzuziehen
- erfordern physisches bzw. physiologisches Logging
- Log-Granulat kleiner oder gleich Sperrgranulat

NoForce-Strategien

- sind Force-Verfahren vorzuziehen
- erfordern den Einsatz von Checkpoint-Maßnahmen zur Begrenzung des Redo-Aufwandes
- 'Fuzzy Checkpoints' erzeugen den geringsten Overhead im Normalbetrieb

Steal-Methoden

- verlangen die Einhaltung des WAL-Prinzips
- erfordern Undo-Aktionen nach einem Rechnerausfall



Übung

Situation im Fehlerfall (Crash)	Datenseite bereits in DB zurückgeschrieben	Log-Satz bereits in Log-Datei geschrieben	Transaktion	
			nicht beendet ggf. Zurücksetzung	abgeschlossen ggf. Wiederholung
1.	Nein	Nein		
2.	Nein	Ja		
3.	Ja	Nein		
4.	Ja	Ja		

Mögliche Antworten:

a) Tue überhaupt nichts

b) Benutze die Undo-Information und setze zurück

c) Benutze die Redo-Information und wiederhole

d) WAL-Prinzip verhindert diese Situation

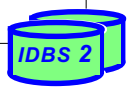
e) Zwei-Phasen-Commit-Protokoll verhindert diese Situation

Übungsfragen

- Welche Möglichkeiten der Fehlerbehandlung bestehen, wenn nach einer Woche festgestellt wird, dass ein neu eingeführtes und bereits hundertfach ausgeführtes Transaktionsprogramm einen logischen Programmfehler enthält?
- Wie könnte ein NoSteal-Verfahren für beliebig lange Transaktionen realisiert werden? Welche Nachteile entstehen? (Hinweis: siehe Implementierung optimistischer Synchronisationsverfahren)
- Zeigen Sie ein Beispiel, das die Probleme einer Force-Schreibstrategie mit Satzsperren verdeutlicht
- Im Beispiel von S. 4-21 soll Seite q verdrängt (ausgeschrieben) werden. Danach bricht Transaktion t17 ab. Welche Logging- und Recovery-Aktionen sind notwendig ?
- Ein OLTP-System verarbeite 1000 Transaktionen pro Sekunde, davon 50% Update-Transaktionen mit durchschnittlich 4 Änderungen.

  Welcher Log-Umfang wird pro Stunde generiert (ein Log-Eintrag habe 200 B)?

  Schätzen Sie die Dauer eines direkten Sicherungspunktes ab für einen DB-Puffer von 800 MB und 4 KB-Seiten ab, wenn im Mittel 30% der Seiten im Puffer geändert sind
- Durch welche Maßnahmen läßt sich ein drohender Überlauf der Log-Datei verhindern, ohne den DB-Betrieb aufzuhalten ?



5. Crash- und Medien-Recovery

- Crash-Recovery
  - Restart-Prozedur
  - Redo Recovery
  - Einsatz von Compensation Log Records
  - Beispiel
- Medien-Recovery (Behandlung von Gerätefehlern)
  - Alternativen
  - Inkrementelles Dumping
  - Erstellung transaktionskonsistenter Archivkopien

Crash-Recovery

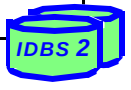
- jüngster transaktionskonsistenter DB-Zustand aus permanenter DB und temporärer Log-Datei herzustellen
- Idempotenz der Recovery: Fehler während Recovery müssen behandelbar sein
- bei Update-in-Place (NonATOMIC):

  Zustand der permanenten DB nach Crash unvorhersehbar (kein logisches Logging anwendbar)

  ein Block der permanenten DB ist entweder
  - aktuell oder
  - veraltet (NOFORCE) ⇒ REDO oder
  - 'schmutzig' (STEAL) ⇒ UNDO
- bei ATOMIC
  - permanente DB entspricht Zustand des letzten erfolgreichen Einbringens
  - zumindest speicherkonsistent ⇒ DML-Befehle ausführbar (logisches Logging)
  - FORCE: kein REDO
  - NOFORCE:

    a) transaktionskonsistentes Einbringen ⇒ REDO, jedoch kein UNDO

    b) speicherkonsistentes Einbringen ⇒ UNDO + REDO



Crash-Recovery für Update-in-Place

(NonAtomic, Steal, Noforce, Checkpoint)

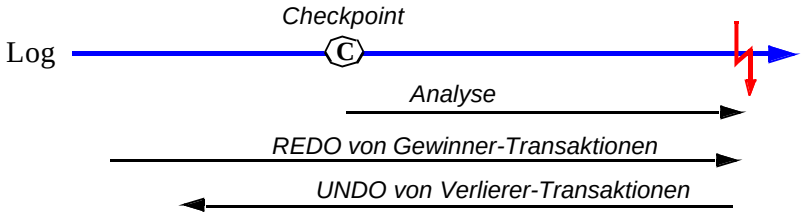
- Temporäre Log-Datei wird 3-mal gelesen:

  1. *Analyse-Lauf (vom letzten Checkpoint bis zum Log-Ende):*
  Bestimmung von ***Gewinner***- und ***Verlierer-Transaktionen*** sowie der Seiten, die von ihnen geändert wurden

  2. *REDO-Lauf:* Vorwärtslesen des Logs (Startpunkt abhängig vom Checkpoint-Typ)

  Wiederholung der Änderungen von Gewinner-Transaktionen (*selektives Redo*) bzw. von allen Transaktionen (*vollständiges Redo*), sofern erforderlich

  3. *UNDO-Lauf:* Rücksetzen der Verlierer-Transaktionen durch Rückwärtslesen des Logs bis zum BOT-Satz der ältesten Verlierer-Transaktion



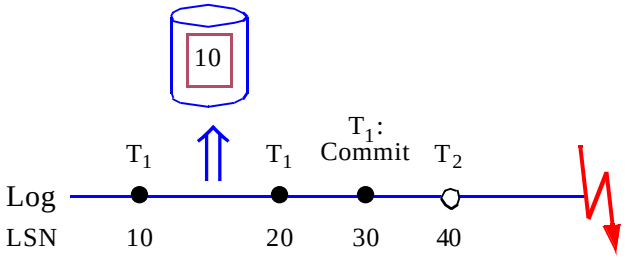
- für Schritte 2 und 3 sind betroffene DB-Seiten einzulesen
- LSN der Seiten zeigen, ob Log-Informationen anzuwenden sind
- am Ende sind alle geänderten Seiten wieder auszuschreiben, bzw. es wird ein Checkpoint erzeugt

Redo-Recovery

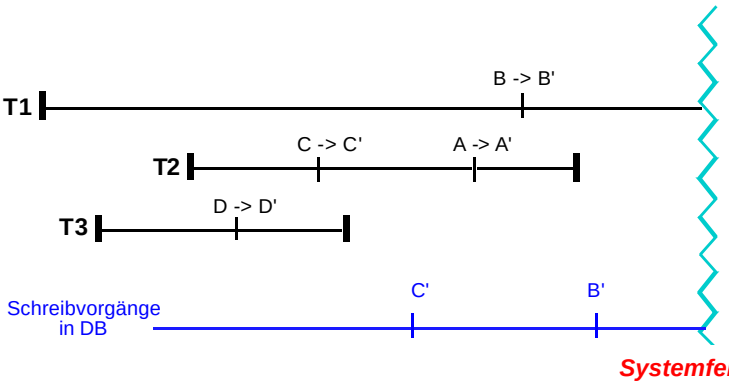
- physiologisches und physisches Logging: Notwendigkeit einer Redo-Aktion für Log-Satz L wird über PageLSN der betroffenen Seite B angezeigt

```
if (B nicht gepuffert) then (lies B in den Hauptspeicher ein);
if   LSN (L) > PageLSN (B)   then do;
        REDO (Änderung aus L);
        PageLSN (B) := LSN (L);
end;
```

- wiederholte Anwendung des Log-Satzes (z.B. nach mehrfachen Fehlern) erhält Korrektheit (Idempotenz der Recovery)



Beispiel



DB-Inhalt

Seite	Page-LSN
A	5
B'	80
C'	60
D	8

Log-Inhalt

LSN	Log-Satz
10	BOT (T1)
20	BOT (T3)
30	BOT (T2)
40	T3, D -> D'
50	Commit (T3)
60	T2, C -> C'
70	T2, A -> A'
80	T3, B -> B'
90	Commit (T2)

Analyselauf:

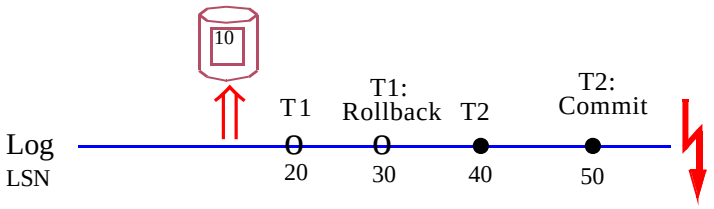
- Verlierer:
- Gewinner:
- relevante Seiten:

Redo-Lauf:

Undo-Lauf:

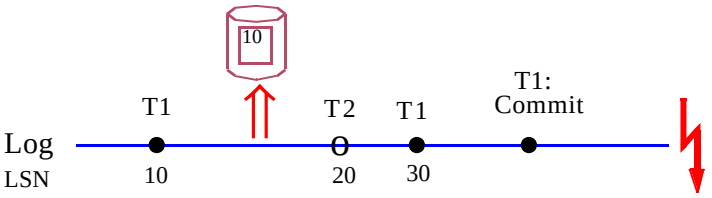
Probleme bei LSN-Verwendung (für Undo)

- UNDO für Verlierer, wenn PageLSN >= LSN (Undo-Log-Satz) ???
- Problem 1: Transaktionsrücksetzungen



Redo-Lauf: Änderung von T2 wird wiederholt (Seiten-LSN := 40)
Undo-Lauf: Änderung 20 von T1 wird zurückgesetzt (da 20 < 40) -> Fehler

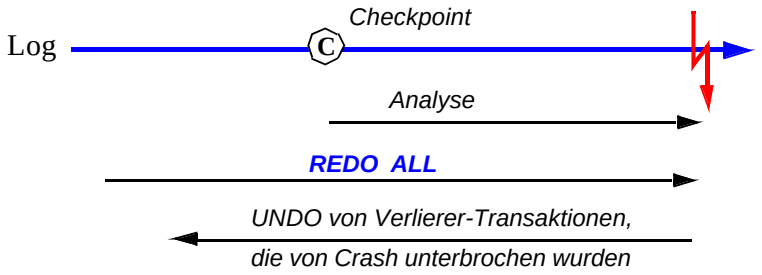
- Problem 2: Satzsperren



REDO von Änderung 30 (Seiten-LSN := 30)
UNDO von Änderung 20, obwohl nicht in der Seite vorhanden

Lösung der Probleme

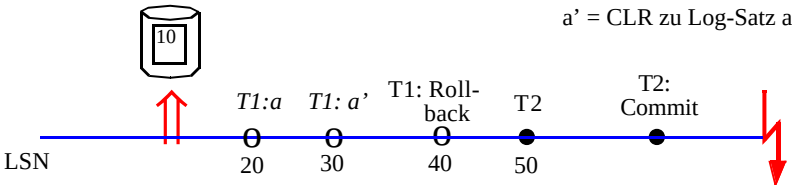
- Protokollieren der Undo-Operationen durch *Compensation Log Records* (CLR)
- Vollständiges Redo oder "*Repeating History*", d.h. im Redo-Lauf werden alle Änderungen (auch von Verlierer-Transaktionen) wiederholt



- Umsetzung durch *ARIES*-Protokoll (*Algorithm for Recovery and Isolation Exploiting Semantics*)
  - entwickelt von Mohan et al. (IBM Research)
  - realisiert in mehreren kommerziellen DBS

Compensation Log Records (CLR)

- *Compensation Log Record* (CLR) = Log-Satz einer Undo-Operation
- CLR wird geschrieben
  - für jede Seitenänderung beim Rollback im Normalbetrieb
  - für jede Undo-Operation während der Crash-Recovery
- Beispiel

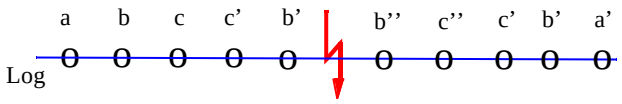


  - selektives Redo:

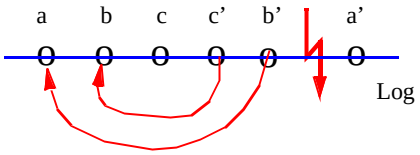
  - vollständiges Redo:

CLR (2)

- CLR-Einsatz bei „Repeating History“ (ARIES)
  - Redo-Lauf : Wiederholung von Rollback-Operationen durch Anwendung der CLR-Sätze
  - Undo-Lauf: nur für Transaktionen, die bei Rechnerausfall aktiv waren
- effiziente Behandlung von Crashs während Recovery
  - jeder CLR-Satz hält Rückwärtsverweis auf transaktionsspezifischen *Vorgänger* des Log-Satzes, für den er die UNDO-Operation repräsentiert
  - durch Crash unterbrochene Rollback-Aktion kann dort fortfahren, wo sie beim Crash-Zeitpunkt angekommen war (keine Kompensation und erneute Durchführung)



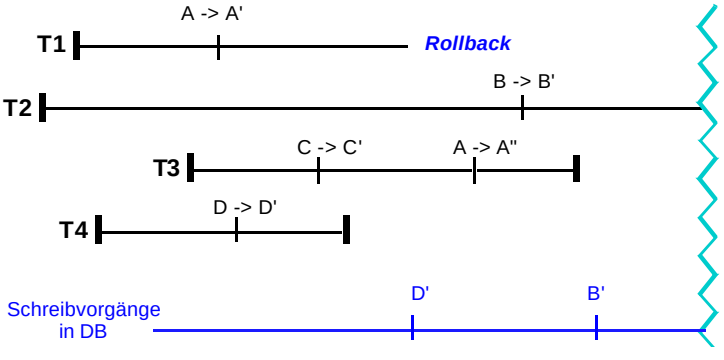
unoptimierter CLR-Einsatz



optimierter CLR-Einsatz (ARIES)

- Rücksetzen während des UNDO-Laufes erfolgt ohne LSN-Vergleich
  - alle noch nicht bearbeiteten UNDO-Sätze von Verlierer-Transaktionen werden angewendet
  - erforderlich, da Redo-Lauf auch Änderungen von Verlierer-Transaktionen wiederholt hatte (’Repeating History’)
  - kein UNDO von CLR-Sätzen

Beispiel



DB-Inhalt

Seite	Page-LSN
A	5
B’	100
C	10
D’	40

Log-Inhalt

LSN	Log-Satz
	...
20	BOT (T3)
30	T1, A -> A'
40	T4, D -> D'
50	Commit (T4)
60	T3, C -> C'
70	T1, A’->A (CLR)
80	Rollback (T1)
90	T3, A -> A"
100	T2, B -> B'
110	Commit (T3)

Analyselauf:

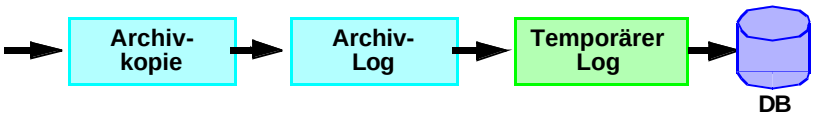
- Verlierer-Transaktionen: T1, T2; davon T2 noch aktiv zum Crash-Zeitpunkt
- Gewinner: T3, T4
- relevante Seiten: A, D, C, B

Vollständiges Redo:

Undo-Lauf: nur T2 betroffen

Platten-Recovery

- Spiegelplatten
  - schellste und einfachste Lösung
  - hohe Speicherkosten
  - Doppelfehler nicht auszuschließen
- Disk-Arrays (RAID-5)
  - geringere Speicherkosten als Spiegelplatten, jedoch auch geringere Fehlertoleranz
  - langsamerer Zugriff für einzelne Blöcke, dafür Unterstützung von E/A-Parallelität
- Alternative: Archivkopie + Archiv-Log
  - Archivkopie + Archiv-Log sind längerfristig verfügbar zu halten (z.B. auf Band)
  - Führen von Generationen der Archivkopie
  - Duplex-Logging für Archiv-Log



- Archiv-Log offline aus temporärer Log-Datei ableitbar
- Erstellung von Archivkopien und Archiv-Log erfolgt segmentorientiert

Erstellung der Archivkopie

- Anhaltung des Änderungsbetriebs zur Erstellung einer DB-Kopie i.a. nicht tolerierbar
- Alternativen:

a) Incremental Dumping

- Ableiten neuer Generationen aus 'Urkopie'
- nur Änderungen seit der letzten Archivkopie werden protokolliert
- Offline-Erstellung einer aktuelleren Kopie

b) Online-Erstellung einer vollständigen Archivkopie (parallel zum Änderungsbetrieb)

- Unterschiedliche Konsistenzgrade:

b1) Fuzzy Dump

- Kopieren der DB im laufenden Betrieb, kurze Lesesperren
- bei Plattenfehler Archiv-Log ab Beginn der Dump-Erstellung anzuwenden

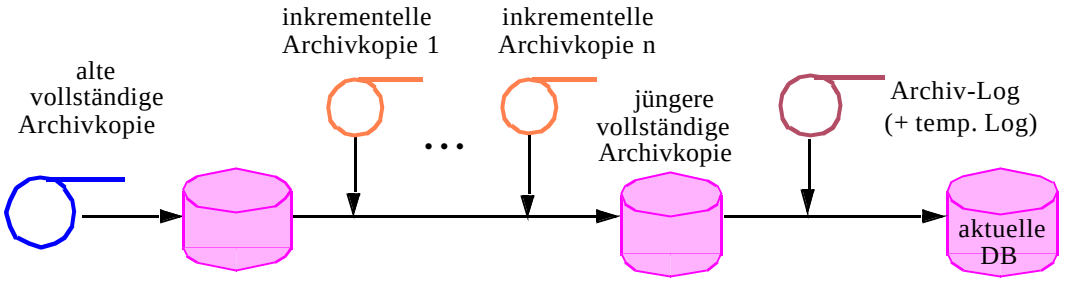
b2) Aktionskonsistente Archivkopie (Voraussetzung bei logischem Operations-Logging)

b3) Transaktionskonsistente Archivkopie (Voraussetzung bei logischem Transaktions-Logging)

- Black-/White-Verfahren
- Copy-on-Update-Verfahren

Inkrementelles Dumping

- nur seit der letzten Archivkopie-Erstellung geänderte DB-Seiten werden archiviert



- Erkennung geänderter Seiten
  - Archivierungs-Bit pro Seite -> alle Seiten für Dump-Erstellung zu lesen
  - sehr hoher E/A-Aufwand
- besser: Verwendung separater Datenstrukturen (Bitlisten)
  - Änderungsbit zeigt Notwendigkeit, Seite in den nächsten Dump zu schreiben
  - Setzen des Änderungsbits falls
    (PageLSN der ungeänderten Seite) < (LSN zu Beginn des letzten Dumps)

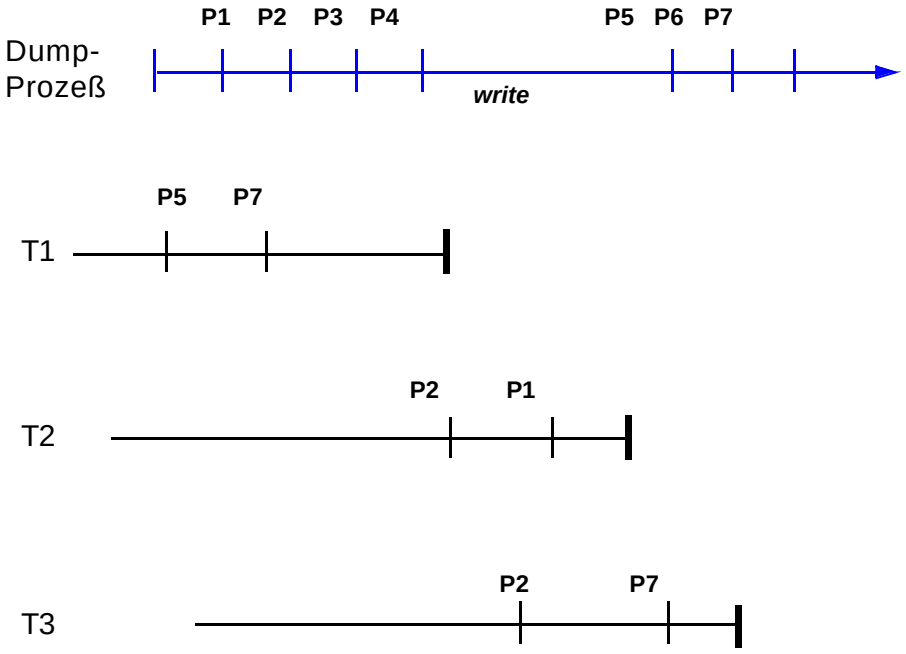
Black-/White-Verfahren

- Erzeugung transaktionskonsistenter Archivkopien
- spezieller Schreibprozeß zur Erstellung der Archivkopie
- Paint-Bit pro Seite:
  - weiß: Seite wurde noch nicht überprüft
  - schwarz: Seite wurde bereits verarbeitet
- Modified-Bit pro Seite zeigt an, ob eine Änderung seit Erstellung der letzten Archivkopie erfolgte
- Schreibprozeß färbt alle weißen Seiten schwarz und schreibt geänderte Seiten in Archivkopie:

```
WHILE there are white pages  DO;    lock any white page;
                                    IF page is modified THEN DO; write page to archive copy;
                                                                 clear modified bit;
                                    END;
                                    change page color;
                                    release page lock;
END;
```

- Transaktionen, die sowohl weiße als auch schwarze Objekte geändert haben (’graue Transaktionen’), werden zurückgesetzt (’Farbtest’ am Transaktionsende)

Black-/White-Verfahren: Beispiel



Black-/White-Verfahren:

Erweiterungen zur Vermeidung von Rücksetzungen

Turn-White-Strategien (Turn gray transactions white)

- für graue Transaktionen werden Änderungen 'schwarzer' Objekte nachträglich in Archivkopie geschrieben
  - Problem: transitive Abhängigkeiten
- Alternative: alle Änderungen schwarzer Objekte seit Dump-Beginn werden noch geschrieben (repaint all)
  - Problem: Archivkopie-Erstellung kommt u.U. nie zu Ende

Turn-Black-Strategien (Turn gray transactions black)

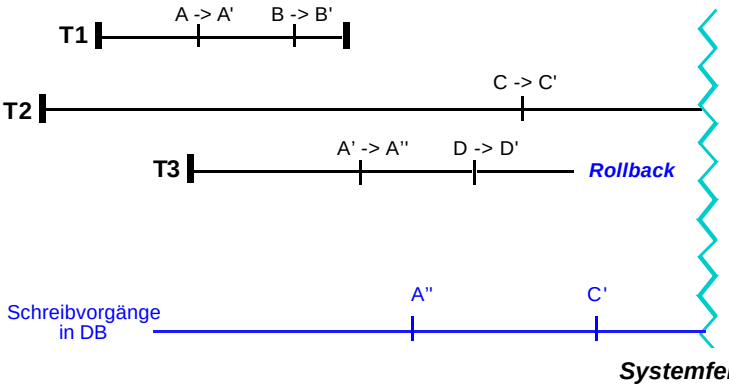
- während der Archivkopie-Erstellung werden keine Zugriffe auf weiße Objekte vorgenommen
  - ggf. zu warten bis Objekt schwarz gefärbt wird
- Alternative: *Copy-on-Update* ("save some")
  - während einer Archivkopie-Erstellung wird bei Änderung eines weißen Objektes Kopie mit Before-Image der Seite angelegt
  - Dump-Prozeß greift auf Before-Images zu
  - Archivkopie entspricht DB-Schnappschuß bei Dump-Beginn

Zusammenfassung

- Crash-Recovery
  - Analyse-, Redo-, Undo-Lauf auf temporärer Log-Datei
  - Redo über Vergleich PageLSN mit LSN der Log-Sätze
  - Notwendigkeit von Compensation Log Records
  - vollständiges Redo (ARIES) unterstützt Satzsperren
  - Idempotenz: Crashs während Recovery können behandelt werden
- Erstellung von Archivkopien der DB (Dumping)
  - inkrementelle vs. vollständige Dumps
  - "Fuzzy Dumps" vs. aktions/transaktionskonsistente Dumps
  - transaktionskonsistente Archivkopie durch "Copy on Update"

Übungsaufgabe

Gegeben sei folgender Transaktionsablauf bis zum Eintreten eines Systemausfalls.



DB-Inhalt

Seite	Page-LSN
A	
B	
C	
D	

Log-Inhalt

LSN	Log-Satz

1. Geben Sie an, welcher Log- und DB-Inhalt zum Crash-Zeitpunkt vorliegt.

2. Beschreiben Sie die gemäß ARIES-Protokoll vorzunehmenden Aktionen während der Crash-Recovery.

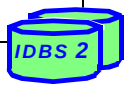
- **Analyse-Lauf:**
- **Redo-Lauf:**
- **Undo-Lauf:**

Übungsfragen

- Was sind CLRs? Wozu werden sie benötigt bzw. welche Vorteile lassen sich mit ihnen erreichen?
- Welche Vorteile/Nachteile hat vollständiges Redo gegenüber selektivem Redo?
- Warum wird durch Spiegelplatten die Notwendigkeit von Archivkopien nicht aufgehoben?
- Schätzen Sie den Zeitbedarf für die wöchentliche Erstellung einer Archivkopie einer 1 TB-Datenbank (Seitengröße 8 KB), wenn pro Woche ca. 10% der Seiten geändert werden und die Archivkopie auf Platten geführt wird
  - für inkrementelles Dumping
  - für Copy-on-Update

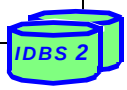
6. Transaktionskonzept: Weiterentwicklungen

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



Beschränkungen flacher Transaktionen: Anwendungsbeispiele

- lange Batch-Vorgänge (Bsp.: Zinsberechnung)
  - Alles-oder-Nichts führt zu hohem Verlust an Arbeit
  - Einsatz vieler unabhängiger Transaktionen verlangt manuelle Recovery-Maßnahmen nach Systemfehler
- Mehrschritt-Transaktionen, lang-lebige Aktivitäten (Bsp.: mehrere Reservierungen pro Transaktion)
  - lange Sperrdauer (Isolation) 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
- Aktive DBS: ECA-Regeln (Event, Condition, Action)
- Realzeit-Anwendungen: zeitbezogene Konsistenzforderungen (Deadlines); oft irreversible Interaktionen mit der Außenwelt

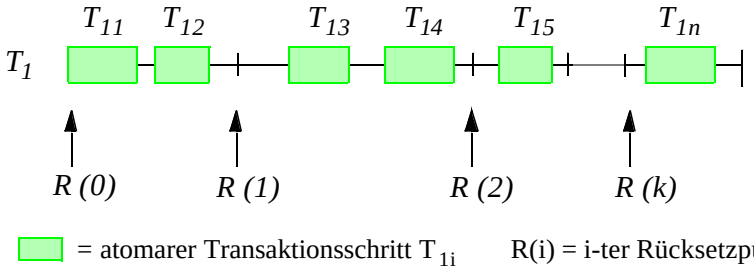


Beschränkungen flacher Transaktionen

- auf kurze Transaktionen zugeschnitten, Probleme mit "lang-lebigen" Aktivitäten
- Alles-oder-Nichts-Eigenschaft oft inakzeptabel: hoher Arbeitsverlust
- Isolation
  - Leistungsprobleme durch "lange" Sperren
  - 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

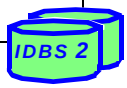


- Operationen: SAVEPOINT R(i)
  ROLLBACK TO SAVEPOINT R(j)
- Protokollierung aller Änderungen, Sperren, Cursor-Positionen etc. notwendig
- Partielle UNDO-Operation bis R(i) in LIFO-Reihenfolge
- Problem: Savepoints werden vom Laufzeitsystem der Programmiersprache nicht unterstützt

Savepoints in SQL:1999

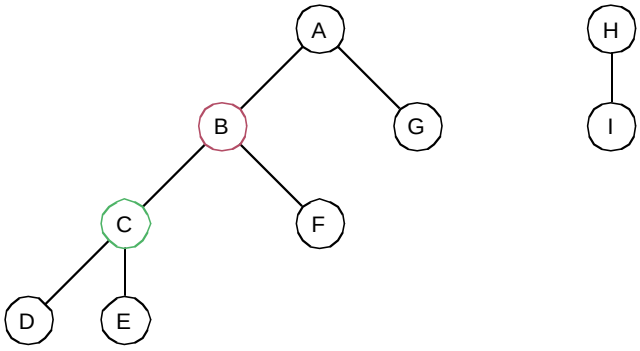
- SQL-Transaktionsanweisungen
  - START TRANSACTION [READ { ONLY | WRITE }]
    [ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED |
    REPEATABLE READ | SERIALIZABLE }]
  - SET TRANSACTION [READ { ONLY | WRITE }] [ISOLATION LEVEL { ... }]
  - SET CONSTRAINTS { ALL | <Liste von Int.bedingungen>} {IMMEDIATE | 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;
```

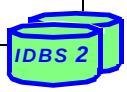


Geschachtelte Transaktionen (nested transactions)

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

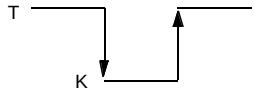


- ausgezeichnete Transaktion = Top-Level Transaction (TL)
  - Wurzel-Transaktion bildet äußersten Kontrollbereich
  - Bewahrung der ACID-Eigenschaften für TL-Transaktion
- Welche Eigenschaften gelten für Sub-Transaktionen?

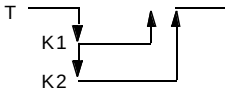


Freiheitsgrade im Modell geschachtelter Transaktionen

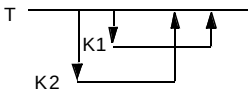
- Repräsentation/Ausführung von (Sub-) Transaktionen
  - in einem oder mehreren Prozessen
  - lokale oder verteilte Anordnung
- Client-Server-Beziehung zwischen Transaktionen
  - synchroner Aufruf: Client ist blockiert, bis Antwort eintrifft
  - asynchroner Aufruf: Client und Server können parallel arbeiten
  - parallele Initiierung mehrerer (synchroner) Aufrufe (PARBEGIN ... PAREND)



a) synchrone Aufrufe, serielle Ausführung



b) synchrone Aufrufe, parallele Ausführung

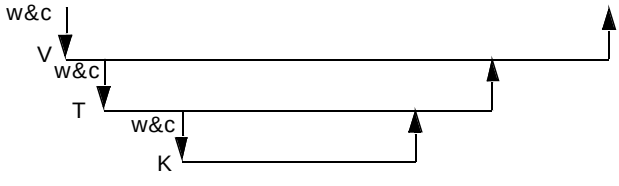


c) asynchrone Aufrufe, parallele Ausführung

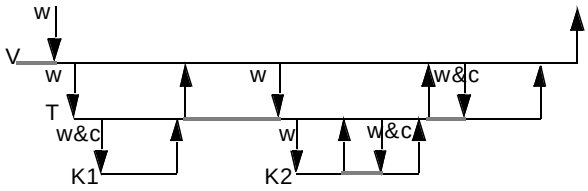
- 4 Arten von “Intra-Transaction”-Parallelität
  - serielle Ausführung von Transaktionen: synchrone Aktivierung (z.B. “remote procedure call”)
  - nur Parallelität zwischen Kind-Transaktionen (sibling parallelism): parallele Aktivierung mehrerer synchroner Aktionen
  - Vater/Kind-Parallelität: nur parallele Transaktionsausführung in 1 hierarchischen Pfad
  - uneingeschränkte Parallelität: asynchrone Transaktionsaktivierungen auf allen Ebenen

Kooperation von Sub-Transaktionen

- “Singe Call”-Schnittstelle: Transaktion erzeugt Sub-Transaktion (und diese ggf. rekursiv weitere); Antwort impliziert Ende der Sub-Transaktion
  - isolierte Rücksetzbarkeit von Sub-Transaktionen
- Konversations-Schnittstelle:
  - Transaktion erzeugt Sub-Transaktion; nach einer Antwort bleibt der Kontext der Sub-Transaktion erhalten; sie kann weitere Anforderungen bearbeiten, bis explizit EOT der Sub-Transaktion ausgeführt wird.



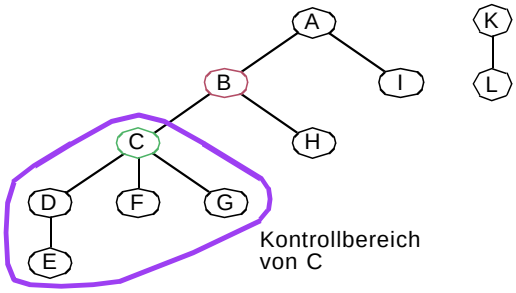
a) Single-Call-Schnittstellen



b) Konversations-Schnittstellen

- Forderung: “Hierarchical Containment” bei Konversationsschnittstelle
  - Vereinfachung der Schachtelungsstruktur
  - Begrenzung der Weitergabe schmutziger Änderungen (Vermeidung von Domino-Effekten)
- Aufrufprimitive
  - work (w)
  - commit (c)
  - abort
  - work & commit (w&c)
  - accept
  - done

Transaktionseigenschaften



- 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

- *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*:
  - Alle Änderungen einer Sub-Transaktion werden bei ihrem Commit für die Vater-Transaktion sichtbar. Alle Objekte, die eine Vater-Transaktion hält, können den Sub-Transaktionen zugänglich gemacht werden. Änderungen einer Sub-Transaktion sind für Geschwister-Transaktionen nicht sichtbar.

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 bei geschachtelten 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

R2: Transaktion T kann R-Sperre erwerben falls

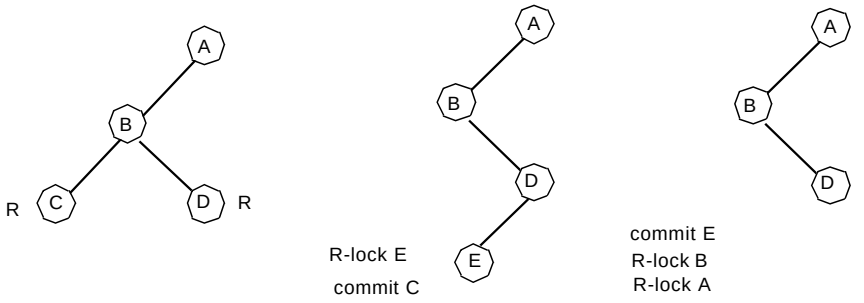
- keine andere Transaktion eine X-Sperre hält, sowie
- alle Transaktionen, welche eine r-X besitzen, Vorfahren von T sind

R3: Beim Commit einer Sub-Transaktion T erbt der Vater von T alle Sperren von T (reguläre sowie retained-Sperren). Für reguläre Sperren von T werden beim Vater die entsprechenden retained-Sperren gesetzt

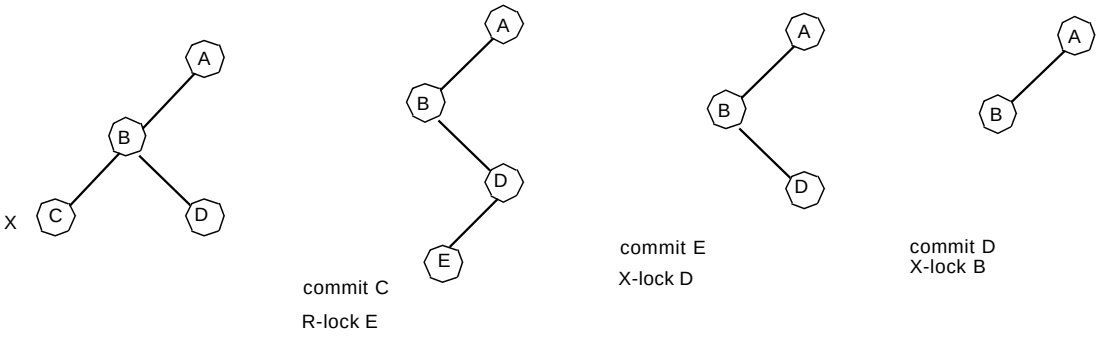
R4: Beim Abbruch einer Transaktion T werden alle regulären und Platzhalter-Sperren von T freigegeben. Sperren der Vorfahren bleiben davon unberührt.

Geschachtelte Transaktionen: Sperrverfahren (2)

a) Lese-Szenario

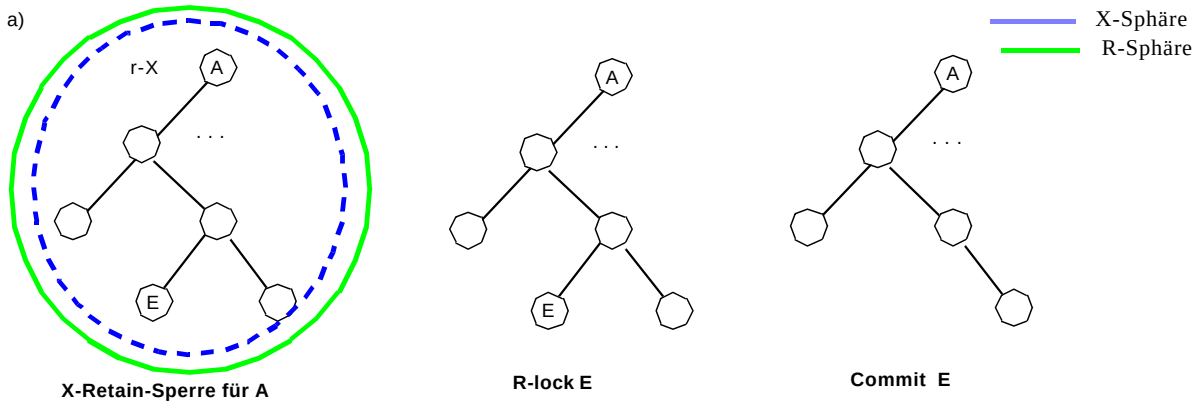


b) Änderungs-Szenario

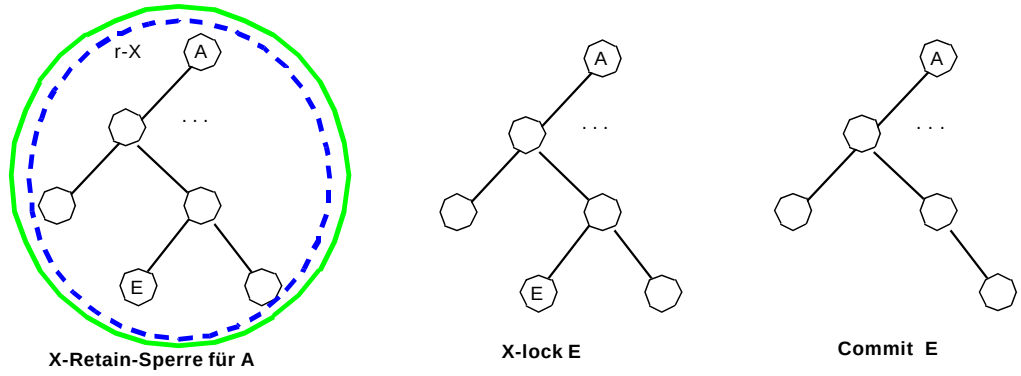


Geschachtelte Transaktionen: Sperrverfahren (3)

a)

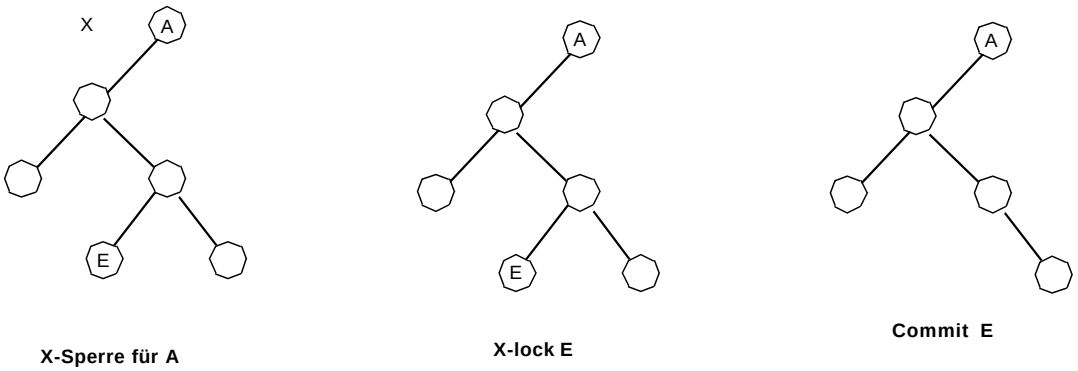


b)



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



X-Sperre für A | X-lock E | Commit E

Merkmale geschlossen geschachtelter Transaktionen

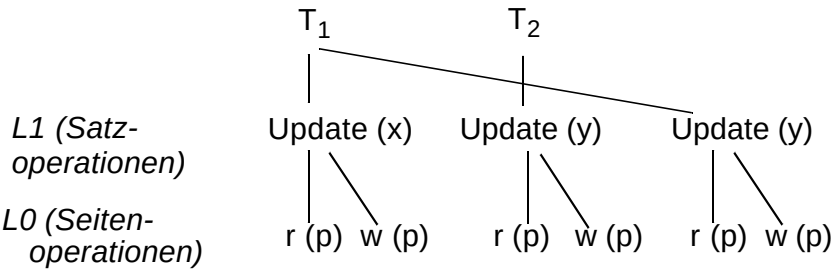
- Vorteile
  - explizite Kontrollstruktur innerhalb von Transaktionen
  - Unterstützung von Intra-Transaktionsparallelität
  - Unterstützung verteilter Systemimplementierung
  - feinere Recovery-Kontrolle innerhalb einer Transaktion
  - Modularität des Gesamtsystems
  - einfachere Programmierung paralleler 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 Unterstützung von Inter-Trans.parallelität (neben Intra-Transaktionsparallelität)
  - Probleme hinsichtlich Korrektheit der Synchronisation sowie der Recovery
- Synchronisationsprobleme
  - Sichtbarwerden "schmutziger" Änderungen verletzt i.a. Serialisierbarkeit
  - dennoch werden oft mit der Realität verträgliche Abläufe erreicht
  - ggf. Einsatz semantischer Synchronisationsverfahren
- 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 kompensierbar (z.B. "real actions" mit irreversiblen Auswirkungen)

Mehrebenen-Transaktionen (multi-level transactions)

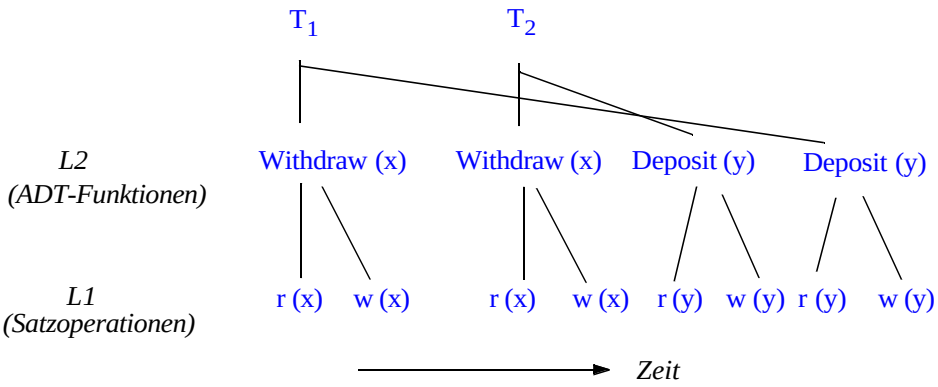
- Schachtelung von Transaktionen längs der Abbildungshierarchie von Schichtenarchitekturen
  - feste Anzahl von Schichten mit bestimmten Operationen
  - Operationen der Ebene i werden jeweils durch Operationen der darunterliegenden Ebene realisiert
  - Transaktionsverwaltung auf jeder Ebene



- vorzeitiges Commit (Freigabe von Änderungen/Sperren) von Sub-Transaktionen
  - offen geschachtelte Transaktionen
  - aber: “Schutzschirm” auf höherer Ebene bleibt erhalten
- reduzierte Konfliktgefahr zwischen Transaktionen unter Wahrung von Serialisierbarkeit
- Transaktionsabbruch erfordert *Kompensation* bereits beendeter Sub-Transaktionen

Mehrebenen-Transaktionen: Merkmale

- Verallgemeinerung auf beliebige Schichten/Operationen möglich
  - Nutzung von Anwendungssemantik zur Synchronisation möglich
  - Basisdienste einer (seitenorientierten) Transaktionsverwaltung des Betriebssystems können genutzt werden



- theoretisch fundierter Ansatz
- Wahl unterschiedlicher Synchronisationstechniken in den verschiedenen Systemschichten (z. B.: optimistische und pessimistische Verfahren)
- potentiell hoher Aufwand zur Transaktionsverwaltung, insbesondere für Logging und Recovery
- für Gesamt-Transaktion gelten weiterhin ACID-Eigenschaften

Langlebige Transaktionen (LLT's)

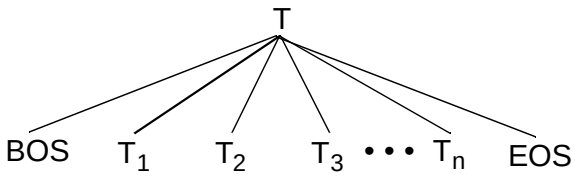
- Bsp.: Bürovorgänge, Entwurfstätigkeiten, ...
- ACID für LLT's
  - sehr lange Sperren
  - Sperren vieler Objekte → Erhöhung der Blockierungsrate und Konfliktrate
  - höhere Rücksetzrate (Deadlockhäufigkeit stark abhängig von der Größe der Transaktion)
  - höhere Chance, durch einen Systemfehler zurückgesetzt zu werden
- Linderung der LLT-Probleme bei speziellen Anwendungen
  - vorzeitige Freigabe von Ressourcen
  - LLT nicht mehr atomar

  => Preisgabe der DB-Konsistenz nicht akzeptabel
- Systemunterstützung erforderlich für
  - vollständige Überwachung eines langlebigen Vorgangs
  - Verwaltung eines konsistenten DB-Zustandes
  - korrekte Behandlung von Fehlern
  - flexibles Ablaufmodell (verteilte und parallele Ausführbarkeit von Teilaktionen etc.)

Das Konzept der Sagas

- Saga ≡ LLT, 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
- Zusicherung des DBS

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

Sagas (2)

- 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)

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

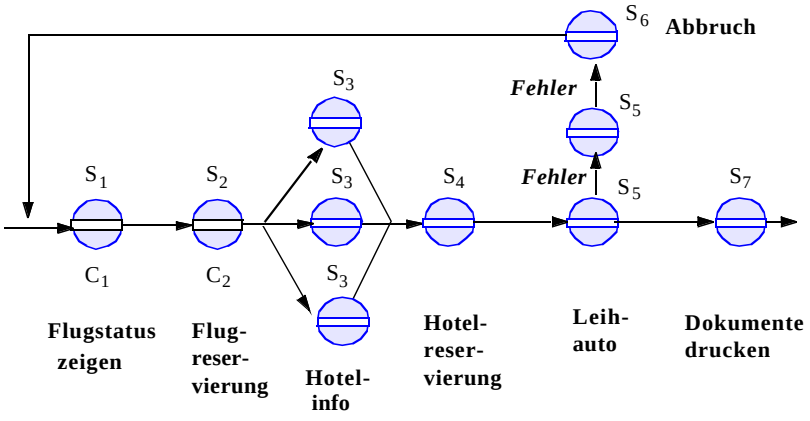
- Zusammenfassung der Eigenschaften:
  - ACID für jede Sub-Transaktionen T_i (D kann durch Kompensation aufgehoben werden)
  - CD für umfassende Transaktion T

ConTracts

- ConTract-Modell: transaktionsübergreifender Mechanismus zur kontrollierten und zuverlässigen Ausführung langlebiger Aktivitäten
- Trennung der Anwendungentwicklung von der Beschreibung der Ablaufstruktur durch zweistufiges Programmiermodell
  - Skript: Beschreibung der Ablaufstruktur / Kontroll- und Datenfluß (Workflow-Definition)
  - Steps: Programmierung der elementaren Verarbeitungsschritte der Anwendung + Kompensationsaktion
  - Step ist sequentielles Programm (ohne Asynchronität, Parallelität usw.), z.B. ACID-Transaktion
- Zentrale Konsistenzeigenschaft: ein ConTract terminiert unter (System-) Garantie in endlicher Zeit und in einem korrekten Endzustand
  - auch bei System-/Knotenfehler Fortsetzung der Verarbeitung "nach vorne" oder
  - kontrollierte Zurückführung eines ConTracts auf seinen Anfangszustand
- Transaktionsübergreifende Verarbeitungskontrolle
  - globale Synchronisation
  - Recovery
  - Sicherung des Verarbeitungszustandes eines ConTracts

ConTracts (2)

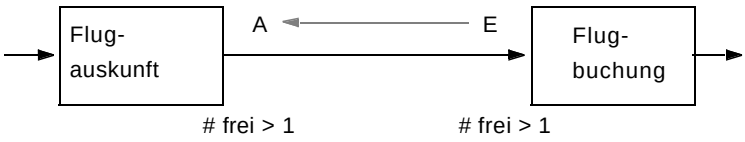
- Erweiterungen gegenüber Saga-Ansatz
  - reichere Kontrollstrukturen (Sequenz, Verzweigung, Parallelität, Schleife, etc.)
  - getrennte Beschreibung von Sub-Transaktionen (**Steps**) und Ablaufkontrolle (**Skript**)
  - Verwaltung eines persistenten **Kontextes** für globale Variablen, Zwischenergebnisse,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



Synchronisation von Contracts

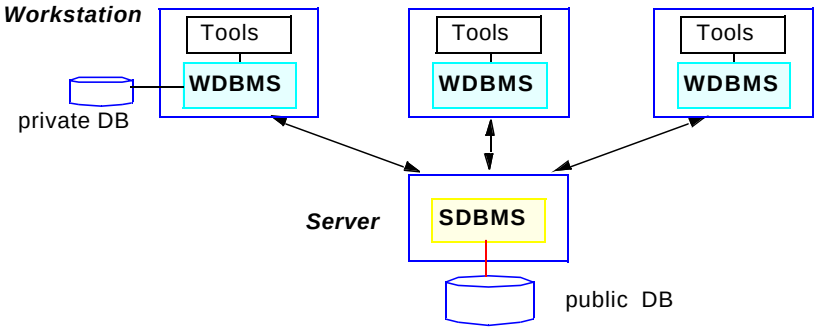
- Synchronisation mit Invarianten: semantische Synchronisationsbedingungen für korrekte Step-Ausführung
  - Ermöglichung eines hohen Parallelitätsgrades
  - Ausschluß von Konsistenzverletzungen trotz frühzeitiger Sperrfreigabe

  Achtung: nicht alle Synchronisationsvorgänge lassen sich so behandeln → z.B. real actions
- 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



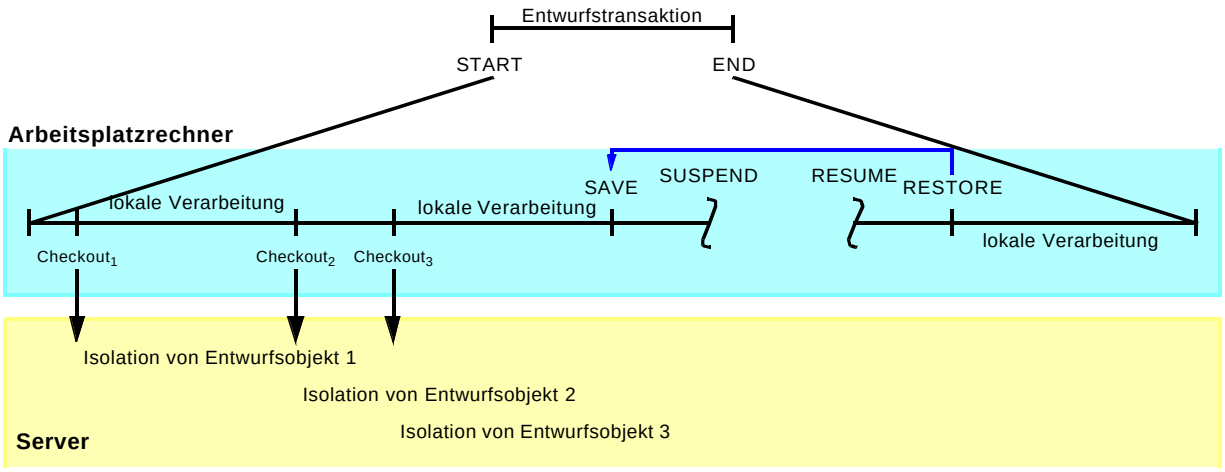
DB-Verarbeitung in Entwurfsumgebungen

- Workstation/Server-Architektur



- 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



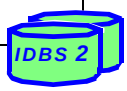
- Charakteristika: 0 .. n Checkout-, 0 .. 1 Checkin-Vorgänge, lange Dauer
- Speicherung von Zwischenzuständen einer Entwurfstransaktion zum:
  - Unterbrechen der Verarbeitung (SUSPEND, RESUME)
  - Rücksetzten auf frühere Verarbeitungszustände (SAVE, RESTORE)

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 in Workstation-/Server-DBS
  - zugeschnittene Verarbeitungsmodelle (Checkout/Checkin)
  - Kooperation innerhalb von Transaktionen
  - Unterstützung von Versionen und Savepoints
- Nachweis der Brauchbarkeit/effizienten Implementierbarkeit der erweiterten Transaktionskonzepte steht noch weitgehend aus

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?



7. Datenbank-Benchmarks

- Einführung
- TPC-C
- TPC-H, TPC-R
- TPC-W
- XML-DB-Benchmarks

Literatur zu DB-Benchmarks:

- J. Gray (ed.): The Benchmark Handbook for Database and Transaction Processing Systems. 2nd edition, Morgan Kaufmann, 1993
  Online-Edition: *www.benchmarkresources.com/handbook/index.html*
- Aktuelle Informationen zu TPC-Benchmarks finden sich unter: *www.tpc.org/*
- XML-DB-Benchmark XMach-1: *dbs.uni-leipzig.de*

Anforderungen an geeignete Benchmarks

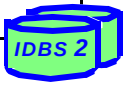
1. Domain-spezifische Benchmarks
   - kein geeignetes Leistungsmaß für alle Anwendungsklassen möglich
   - spezielle Benchmarks für techn./wissenschaftliche Anwendungen, DB-Anwendungen, etc.

2. Relevanz
   - Berücksichtigung “typischer” Operationen des jeweiligen Anwendungsbereichs
   - Messen der maximalen Leistung
   - Berücksichtigung der Systemkosten (Kosteneffektivität)

3. Portierbarkeit
   - Übetragbarkeit auf verschiedene Systemplattformen
   - Vergleichbarkeit

4. Skalierbarkeit
   - Anwendbarkeit auf kleine und große Computersysteme
   - Übertragbarkeit auf verteilte/parallele Systeme

5. Einfachheit / Verständlichkeit



- Herstellergremium zur Standardisierung von DB-Benchmarks

- Gründung 1988

- erste Benchmarks basierend auf Kontenbuchung (“Debit-Credit”): TPC-A (1989) und TPC-B (1990)

- besondere Merkmale
  - Leistung eines Gesamt-Systems wird bewertet
  - Bewertung der Kosteneffektivität (Kosten / Leistung)
  - skalierbare Konfigurationen
  - verbindliche Richtlinien zur Durchführung und Dokumentation (Auditing; Full Disclosure Reports)
  - Ausschluß von “Benchmark Specials” innerhalb von DBMS etc.

- aktuelle Benchmarks für OLTP (TPC-C), Decision Support (TPC-H/R) und Web-Transaktionen (TPC-W)

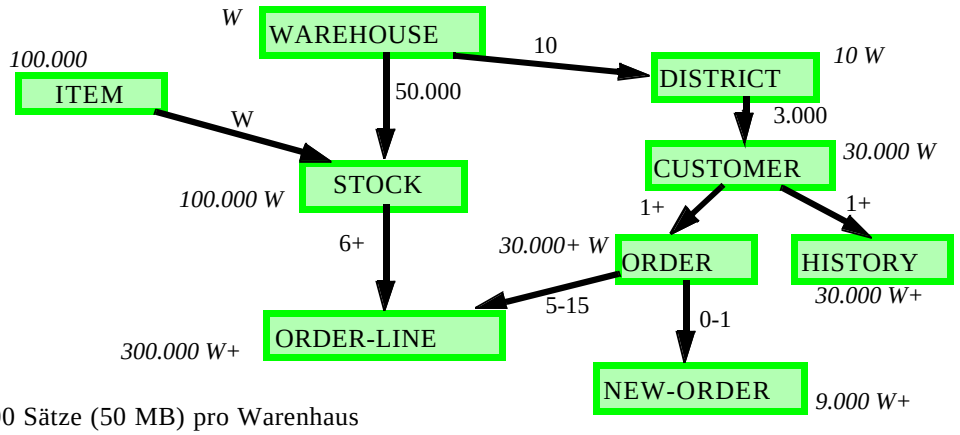
- einige erfolglose Benchmark-Definitionsversuche (TPC-S, TPC-Client/Server, TPC-E)

TPC-C-Benchmark

- Verabschiedung: August 1992

- Anwendung: Bestellverwaltung im Großhandel (order entry)
  - Betrieb umfaßt W Warenhäuser, pro Warenhaus 10 Distrikte, pro Distrikt 3000 Kunden
  - 100.000 Artikel; pro Warenhaus wird Anzahl vorhandener Artikel geführt
  - 1% aller Bestellungen müssen von nicht-lokalem Warenhaus angefordert werden

- 9 Satztypen



über 500.000 Sätze (50 MB) pro Warenhaus

TPC-C (2)

- Haupttransaktionstyp NEW-ORDER

```
BEGIN WORK    { Beginn der Transaktion }
    SELECT ... FROM CUSTOMER
        WHERE c_w_id = :w_no AND c_d_id = :d_no AND c_id = :cust_no
    SELECT ... FROM WAREHOUSE WHERE w_id = :w_no
    SELECT ... FROM DISTRICT    (* -> next_o_id *)
        WHERE d_w_id = :w_no AND d_id = :d_no
    UPDATE DISTRICT SET d_next_o_id := :next_o_id + 1
        WHERE d_w_id = :w_no AND d_id = :d_no
    INSERT INTO NEW_ORDER ...
    INSERT INTO ORDERS ...
    pro Artikel (im Mittel 10) werden folgende Anweisungen ausgeführt:
        SELECT ... FROM ITEM WHERE ...
        SELECT ... FROM STOCK WHERE ...
        UPDATE STOCK ...
        INSERT INTO ORDER-LINE ...
COMMIT WORK    { Ende der Transaktion }
```

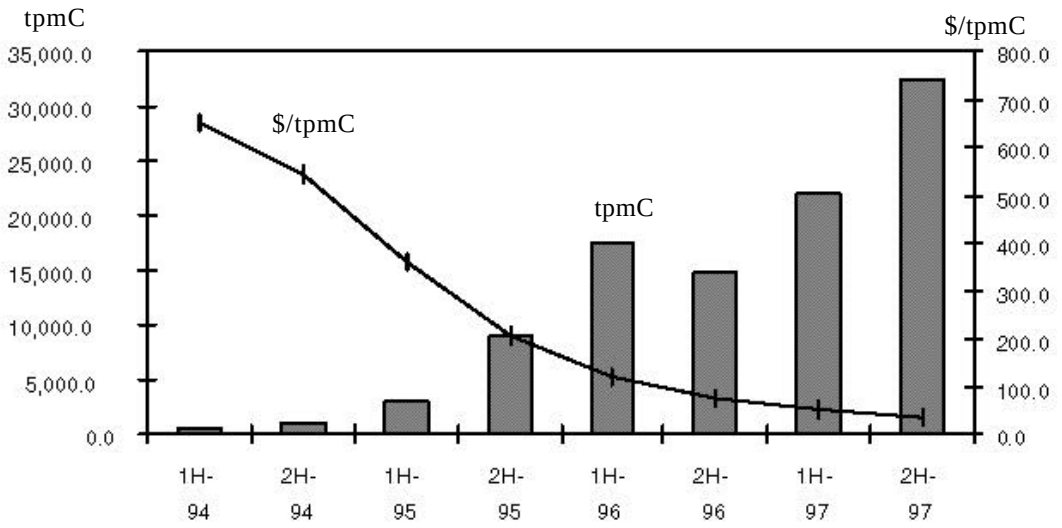
- im Mittel 48 SQL-Anweisungen (BOT, 23 SELECT, 11 UPDATE, 12 INSERT, EOT)
- 1% der Transaktionen sollen zurückgesetzt werden

TPC-C (3)

- 5 Transaktionstypen:
  - *New-Order:* Artikelbestellung (Read-Write)
  - *Payment:* Bezahlung einer Bestellung (Read-Write)
  - *Order-Status*: Status der letzten Bestellung eines Kunden ausgeben (Read-Only)
  - *Delivery:* Verarbeitung von 10 Bestellungen (Read-Write)
  - *Stock-Level:* Anzahl von verkauften Artikeln bestimmen, deren Bestand unter bestimmtem Grenzwert liegt (Read-Only)
- Durchsatzangabe für New-Order-Transaktionen in tpm-C (Transaktionen pro Minute)
- Festlegung des Transaktions-Mixes
  - New-Order-Anteil variabel, jedoch höchstens 45 %
  - Payment-Transaktionen müssen mindestens 43 % der Last ausmachen
  - Order-Status, Delivery und Stock-Level je mindestens 4 %
- pro Transaktionstyp festgelegte Antwortzeitrestriktion (90% unter 5s bzw. 20 s für Stock-Level), mittlere Denkzeiten und Eingabezeiten
- Kostenefffektivität ($/tpm-C) unter Berücksichtigung aller Systemkosten für 5 Jahre (ab V5: 3 Jahre)

Entwicklung der TPC-C-Ergebnisse

- Durchschnittswerte der Top-5-Resultate (tpmC und $/tpmC) von 1994 - 1997:

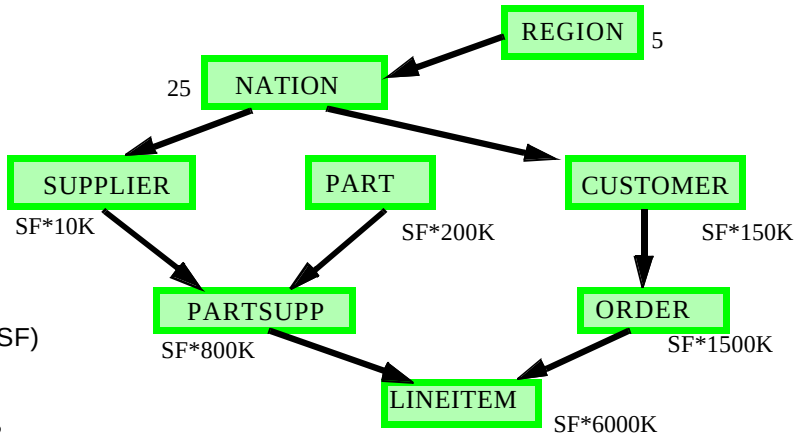


- innerhalb 3,5 Jahre: 60-fache Leistungssteigerung, 20-fach bessere Kosteneffektivität
- erstes TPC-C-Ergebnis (Sep. 92): 54 tpmC, 188 562 $/tpmC
- beste Ergebnisse Juni 2003 (V5): 763 898 tpmC (8,3 $/tpmC) bzw. 2,5 $/tpmC†

† aufgrund von Änderungen in der Benchmark-Definition nicht direkt vergleichbar mit Ergebnissen von 1992

TPC-H, TPC-R

- Fokus auf komplexen Anfragen (Decision Support)
- Vorgänger TPC-D (1995 - 1999)
- TPC-R (Business/decision Reporting) erlaubt Vorberechnungen gegenüber TPC-H (Ad-hoc-Anfragen)
- DB-Aufbau:



acht Skalierungsfaktoren (SF) von 1 bis 10000

-> DB-Größe: 1 GB bis 10 TB

TPC-H/R: Last

- 22 komplexe Anfragetypen (Anfragen nach Sonderangeboten, Lieferzeiten, Kundenzufriedenheit, Marktanteilen etc.) und 2 Refresh-Updates)
- Query 9 Business Question:

  The Product Type Profit Measure Query finds, for each nation and each year, the profit for all parts ordered in that year which contain a specified substring in their names and which were filled by a supplier in that nation. The profit is defined as the sum of [(L_EXTENDEDPRICE*(1-L_DISCOUNT)) - (PS_SUPPLYCOST * L_QUANTITY)] for all line items describing parts in the specified line. The query lists the nations in ascending alphabetical order and, for each nation, the year and profit in descending order by year.

```
SELECT NATION, YEAR, SUM(AMOUNT) AS SUM_PROFIT
FROM   (SELECT     N_NAME AS NATION, EXTRACT (YEAR FROM O_ORDERDATE) AS YEAR,
                   L_EXTENDEDPRICE*(1-L_DISCOUNT)-PS_SUPPLYCOST*L_QUANTITY AS AMOUNT
        FROM PART, SUPPLIER, LINEITEM, PARTSUPP, ORDER, NATION
        WHERE S_SUPPKEY = L_SUPPKEY AND PS_SUPPKEY = L_SUPPKEY AND
                   PS_PARTKEY = L_PARTKEY AND P_PARTKEY = L_PARTKEY AND
                   O_ORDERKEY = L_ORDERKEY AND S_NATIONKEY = N_NATIONKEY AND
                   P_NAME LIKE '%green%' ) AS profit
GROUP BY NATION, YEAR
ORDER BY NATION, YEAR DESC;
```

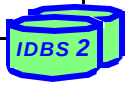
  - 6-Wege-Join ohne Beschränkung der beteiligten Tabellen (außer PART), Gruppierung/Sortierung
  - Subquery in FROM-Klausel

TPC-H, TPC-R (3)

- Leistung für Einbenutzerbetrieb (Power-Metrik) in *QppH@size* bzw. *QppR@size* (z.B. 7500QppH@100GB)
  - Queries pro Stunde bezogen auf bestimmte DB-Größe
  - Berechnung $$QppH@size = \frac{3600}{\sqrt[24]{Q1 \times Q2 \times \ldots \times Q22 \times U1 \times U2}} \times SF$$
- Leistung für Mehrbenutzerbetrieb (Throughput Test) in *QthH@size*

  Berechnung $$QthH@size = \frac{noStreams \times 22 \times 3600}{Zeitdauer} \times SF$$
- kombiniertes Leistungsmaß (query per hour)

  $$QphH@size = \sqrt{QppH@size \times QthH@size}$$
- Kosteneffektivität: price-per-QphH@size = $ / QphH@size
- Beispiel-Ergebnisse (Stand: Juni 2003)
  - TPC-H, 300 GB: 12995 QphH; 203 $/QphH (DB2 UDB7.2 auf Compaq DL760x900-64P)
  - TPC-H, 10 TB: 81501 QphH; 243 $/QphH (Teradata auf NCR 5350)

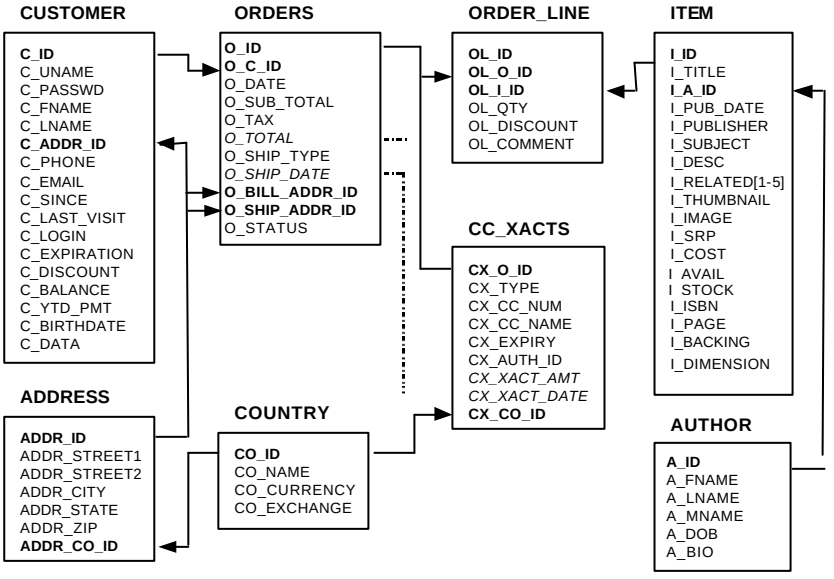


TPC-W

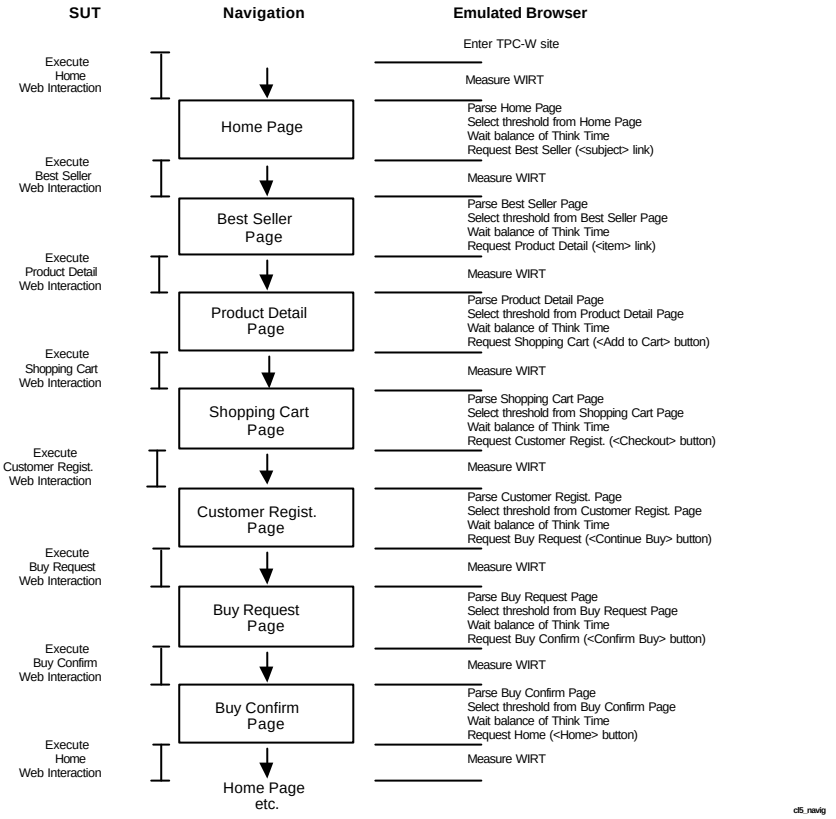
- Nachbildung von E-Commerce-Betrieb (Online-Buchbestellung)
- wurde 2000 verabschiedet
- feste Web-Interaktionsmuster mit unterschiedlichen Transaktionsschritten: Browse, Search, Order, ...
- Berücksichtigung von Benutzerauthentifikation, Datenverschlüsselung, dynamische Seitengenerierung, ...
- Mehrbenutzer-Benchmark mit ACID-Datenbanktransaktionen
  - Web Page Consistency: jede DB-Änderung muss spätestens nach 30 Sekunden in Ergebnissen reflektiert sein (Ausnahme: Search)
  - Durchsatzmessung unter Antwortzeitrestriktionen
  - hauptsächliches Leistungsmaß: Web Interactions per Second (WIPS)
  - Preis-Leistungs-Maß: $ / WIPS

TPC-W Datenbank

- 8 Tabellen
- 2 Bilder pro Item: 5KB - 250 KB (TPC-W Image Generator)
- DB-Skalierung über #Items (1.000 - 10 Millionen) und #EB (emulierte Browser), z.B. #Customer=2880*#EB
- DB-Größe für 1000 EB, 10.000 Items: ca. 5 GB
- Speicherkosten für 180-Tage-Betrieb (DB wächst proportional zur WIPS-Rate)
- Web-Log dauerhaft zu speichern: 14 Tage, davon mind. 8 h am Stück



Web-Interaktionsschema



Simulierte Homepage

Last-Mixes

- 3 Szenarien: Shopping-Szenario (ergibt primäres WIPS-Leistungsmaß) sowie Browsing-Szenario (WIPSb) und Order-Szenario (WIPSo)

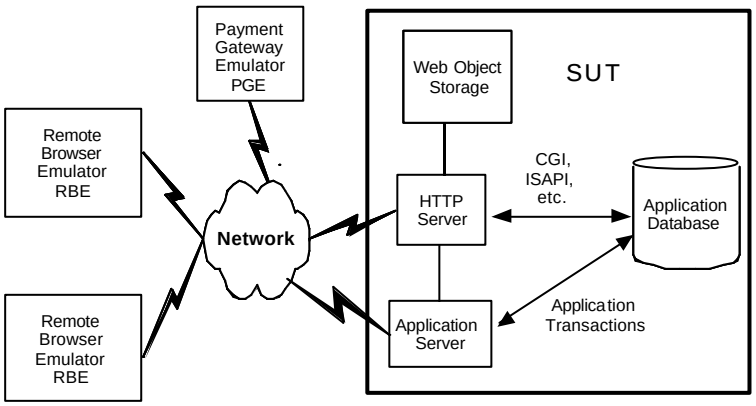
Mix of Web Interactions

Web Interaction	Browsing Mix (WIPSb)	Shopping Mix (WIPS)	Ordering Mix (WIPSo)
Browse	**95 %**	**80 %**	**50 %**
Home	29.00 %	16.00 %	9.12 %
New Products	11.00 %	5.00 %	0.46 %
Best Sellers	11.00 %	5.00 %	0.46 %
Product Detail	21.00 %	17.00 %	12.35 %
Search Request	12.00 %	20.00 %	14.53 %
Search Results	11.00 %	17.00 %	13.08 %
Order	**5 %**	**20 %**	**50 %**
Shopping Cart	2.00 %	11.60 %	13.53 %
Customer Registration	0.82 %	3.00 %	12.86 %
Buy Request	0.75 %	2.60 %	12.73 %
Buy Confirm	0.69 %	1.20 %	10.18 %
Order Inquiry	0.30 %	0.75 %	0.25 %
Order Display	0.25 %	0.66 %	0.22 %
Admin Request	0.10 %	0.10 %	0.12 %
Admin Confirm	0.09 %	0.09 %	0.11 %

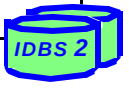
- Antwortzeitschranken von 3 s (Home, Search ...) bis 20 s (Admin confirm) müssen für mind. 90% der Interaktionen eingehalten werden

Messung

- Meßintervall mind. 30 Minuten (erzielte Durchsatzwerte sollen aber über 8 h gehalten werden können)



- Durchsatz muß proportional zu #EB sein: #EB/14 < WIPS < #EB/7
- Systemkosten (SUT) von 3 Jahren für $ / WIPS
- Beispielergebnisse (Juni 2003)
  - 10.000 Items: 21139 WIPS, 33 $/WIPS (MS SQL-Server 2000, IBM eServer 440)
  - 100.000 Items: 10439 WIPS, 107 $/WIPS (MS SQL-Server 2000, Unisys ES 7000 (16 P))



Benchmarks für XML-Datenbanksysteme

Datenbankansätze

- „native“ XML-Datenbanksysteme (z.B. Tamino, eXcelon, Infonyte, X-Hive, eXist ...)
- um XML-Fähigkeiten erweiterte (objekt-) relationale DBS: Oracle, IBM DB2, MS SQLServer

XML-Datenarten

- Dokumente mit oder ohne Schema
- strukturierte, semistrukturierte, gering strukturierte Daten

Speicherungsarten für XML-Dokumente

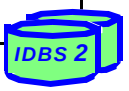
- Speicherung als Ganzes (z.B. als CLOB)
- zerlegte Speicherung: generisch (schema/anwendungsunabhängig)
- zerlegte Speicherung: anwendungsabhängig (z.B. Mapping XML-Elemente -> relationale Tabellen/ Attribute)
- Mischformen

Operationen

- Pfadausdrücke / Navigation
- DB-Operationen (Selektion, Join, Sortierung ...)
- Volltextanfragen ...

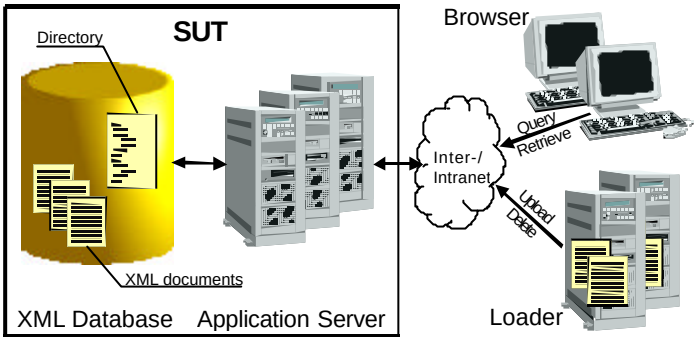
Vergleich von XML-DB-Benchmarks

	XMach-1	XMark	XOO7	XBench	MBench
Anwendungs-domäne	Dokumenten-kollektion	Auktions-verwaltung	Komponenten-verwaltung	mehrere (4)	synthetisch
Schwerpunkt	dokument-orientiert	daten-orientiert	datenorientiert	je nach Domäne	datenorientiert
Skopus	DBMS	Anfrage-prozessor	Anfrage-prozessor	Anfrage-prozessor	Anfrage-prozessor
# Benutzer	Mehrbenutzer	Einbenutzer	Einbenutzer	Einbenutzer	Einbenutzer
# Rechner	≥1	1	1	1	1
# Schemas	#Dok/20	1	1	1-6	1
# Dokumente	10^x (x>2)	1	1	1 / 2,6*10^x / 2600*10^x (1≤x≤4)	1
DB-Größe	16 KB * 10^x	10MB – 10GB	ca. 4 – 1GB	10MB – 10GB	ca. 50MB - 50GB
# Anfragen	8	20	18	20	49
# Update-Op.	3	0	0	0	7



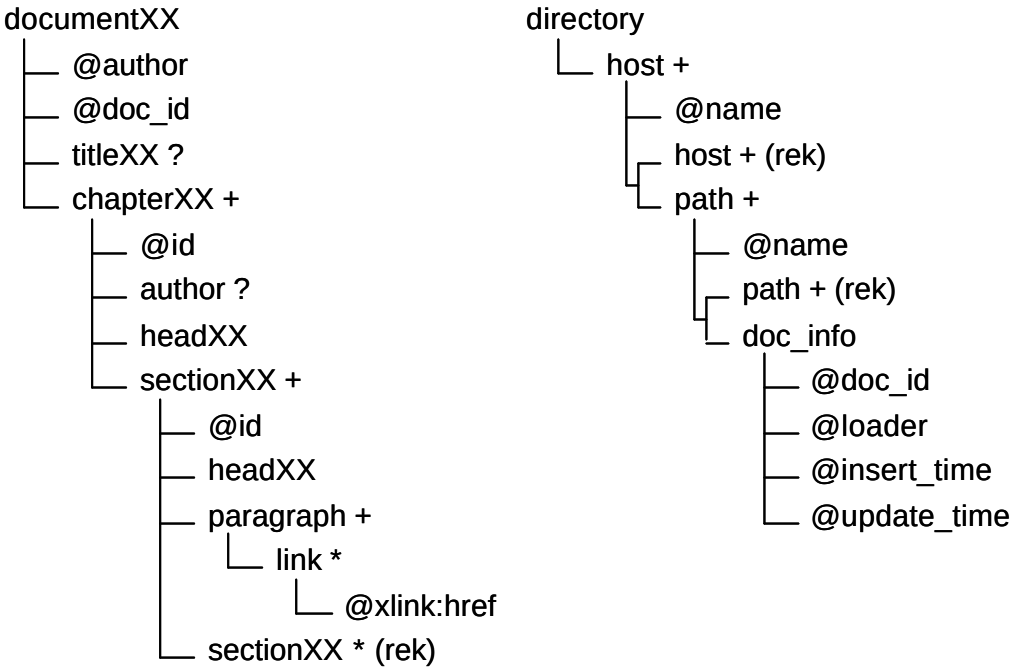
XMACH-1(*X*ML Data *Ma*nagement Ben*ch*mark)

- weltweit erster XML-DB-Benchmark; entwickelt an Univ. Leipzig (2000)
  - Mehrbenutzer-Benchmark
  - synthetisch generierte XML-Dokumente (mit oder ohne Schema) variabler Struktur
  - skalierbar (1000 - 10 Millionen Dokumente)
  - Lastmix mit Antwortzeitrestriktionen (8 Query- und 3 Update-Typen)
- Performance-Metriken: Anzahl XML-Queries pro Sekunde: XQps bzw. XQps-sl (schemalos)



- Beschreibung und Referenzimplementierung s. http://dbs.uni-leipzig.de

X-Mach1: Daten-Layout



X-Mach1 Operationen

■ Anfragen

- Finde Dokument mittels URL
- Finde Dokument-URL von Dokumenten, die eine bestimmte Phrase in paragraph-Elementen haben
- Navigiere über jeweils das erste section-Element ausgehend vom ersten chapter-Element
- Erzeuge Liste aller head-Elemente eines Dokumentes
- Finde alle Dokumentnamen unterhalb einer URL
- Finde Elternelemente von author-Elementen mit bestimmten Inhalt
- Finde Dokumente, die von wenigstens 4 anderen Dokumenten referenziert werden

```
FOR     $refId IN distinct(/*//link/@href)
LET     $refDocs := distinct(/*[.//link/@href=$refID]/@doc_id)
WHERE count($refDocs) >= 4
RETURN <docid>{string($refID)}</docid>
```

- Finde die zuletzt geänderten 100 Dokumente, die ein author-Attribut enthalten

■ Änderungen

- Neues Dokument einfügen
- Dokument löschen
- Setze update_time

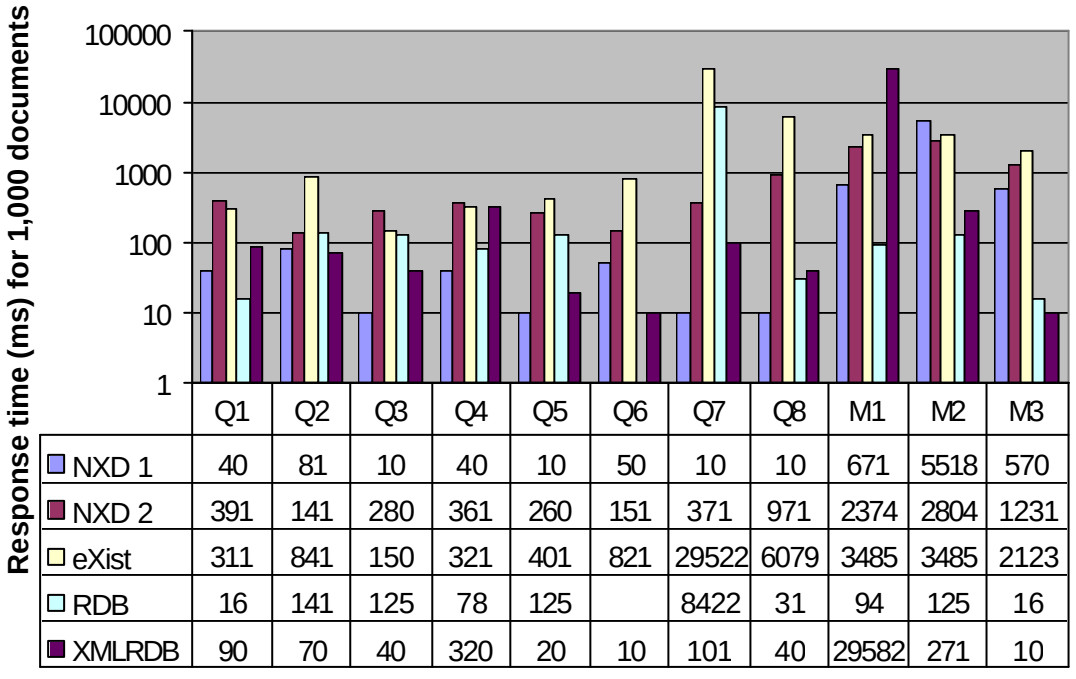
Ergebnisse: DB-Population

- Größe in MB, Zeiten in s

#Dok		NXD1		NXD2		NXD3	eXist		XMLRDB		RDB*	
1,000	Größe ges.	57		206		80	26		71		136	
	Größe Dat. | Idx			145	61		19	7	41	30	129	7
	Ladezeit ges.	734		616		330	130		130		91	
	Zeit Dat. | Idx	579	155	88	528						72	19
10,000	Größe ges.	214+TxtIdx		928		883			834		389	
	Größe Dat. | Idx			561	367				461	373	326	63
	Ladezeit ges.	8.047+TxtIdx		10.803		4.969			2.965		872	
	Zeit Dat. | Idx			971	9.832				1.351	1.614	715	157

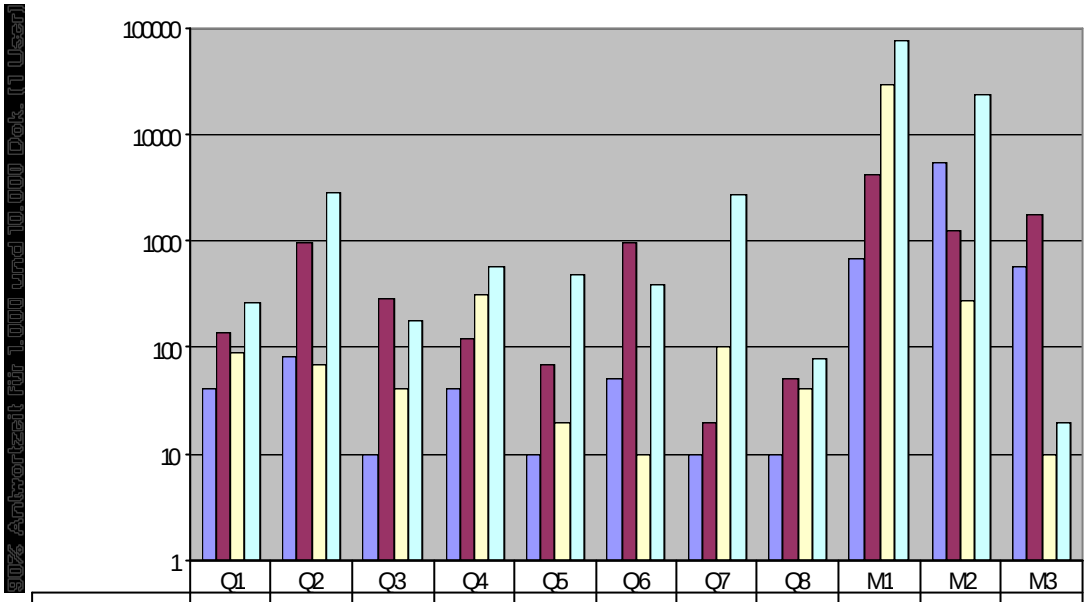
- Rohdatengröße: ca. 15,5 KB / 125 Elemente pro Dokument; 41 / 695 DTDs für 1000/10.000 Dok.
- Server: Intel PIII 800 MHz (512 MB) bzw. * = PIII 2*1GHz, 1GB , jeweils Windows 2000

XMach-1: Antwortzeitvergleich



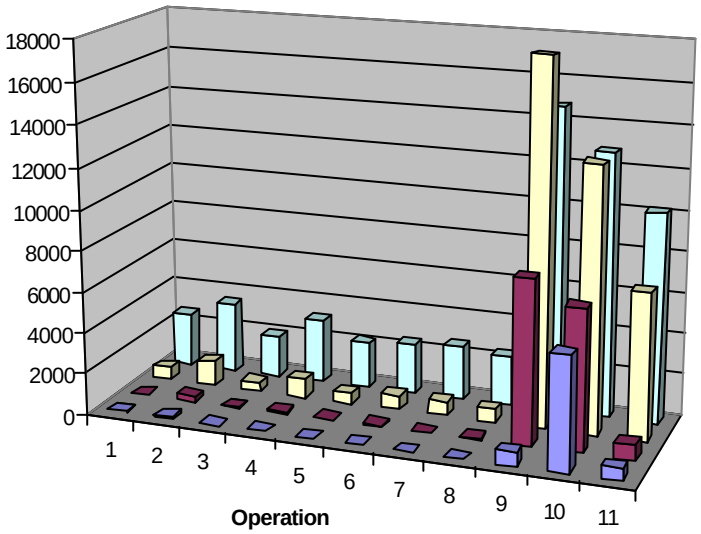
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	M1	M2	M3
NXD 1	40	81	10	40	10	50	10	10	671	5518	570
NXD 2	391	141	280	361	260	151	371	971	2374	2804	1231
eXist	311	841	150	321	401	821	29522	6079	3485	3485	2123
RDB	16	141	125	78	125		8422	31	94	125	16
XMLRDB	90	70	40	320	20	10	101	40	29582	271	10

XMach-1: Einfluß #Dokumente



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	M1	M2	M3
NXD 1 1000	40	81	10	40	10	50	10	10	671	5518	570
NXD1 10000	140	952	290	121	71	951	20	50	4135	1272	1743
XMLRDB 1000	90	70	40	320	20	10	101	40	29582	271	10
XMLRDB 10000	261	2844	180	571	490	380	2714	80	76088	24144	20

XMach-1: Einfluß #Clients



	1	2	3	4	5	6	7	8	9	10	11
1 Client	40	81	10	40	10	50	10	10	671	5518	570
20 Clients	60	260	50	110	40	80	50	50	8032	6870	791
50 Clients	631	1222	471	921	511	671	691	611	17655	12848	7130
80 Clients	2654	3425	2043	3144	2253	2433	2624	2434	14811	12819	10204

XMach-1: Durchsatzergebnisse