Transaktionsverarbeitung

Lehrveranstaltung Datenbanktechnologien

Prof. Dr. Ingo Claßen Prof. Dr. Martin Kempa

Hochschule für Technik und Wirtschaft Berlin

Transaktionen

- Folge von Datenbankanweisungen, die als Einheit betrachtet werden
- Beispiel: Geld von einem Unterkonto auf ein anderes überweisen

```
Kontonummern auswählen: uk1Nr, uk2Nr
Betrag eingeben: betrag

// Überprüfung ob Unterkonto 1 gedeckt ist
if ok then
  update Konto
  set Kontostand = Kontostand - :betrag
  where KontoNr = :uk1Nr

  update Konto
  set Kontostand = Kontostand + :betrag
  where KontoNr = :uk2Nr
end
```

ACID-Eigenschaften

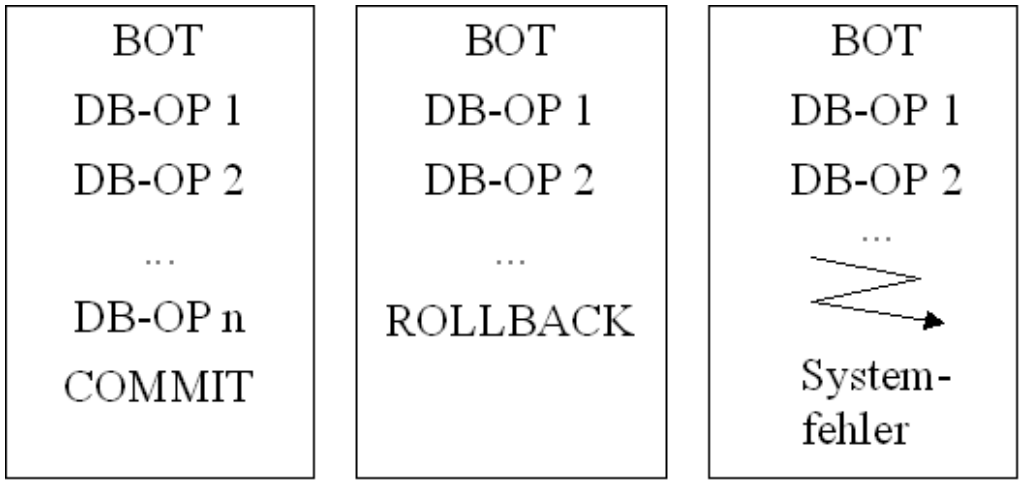
- Atomarität (atomicity)
  - Ausführung aller Aktionen oder keiner
- Konsistenz (consistency)
  - Überführung der Datenbank von einem konsistenten in einen konsistenten Zustand
- Isolation (isolation)
  - Keine Beeinflussung parallel ablaufender Transaktionen
- Dauerhaftigkeit (durability)
  - Bestätigte Veränderungen dürfen nicht verloren gehen

Mechanismen zur Sicherstellung von ACID

- Einbettung von Transaktionen in Anwendungsprogramme
  - Programmtechnische Transaktionssteuerung
  - Schnittstelle zwischen Anwendungsprogramm und Transaktionsmanager
- Transaktionssysteme
  - Transaktionsmanager
  - Transaktionsmonitore
- Steuerung der Nebenläufigkeit (concurrency control)
  - Synchronisation von Zugriffen auf gemeinsame Daten
- Wiederherstellung (recovery)
  - Herstellung eines konsistenten Zustands nach Fehlern

Transaktionssteuerung

- BOT
  - Beginn der Transaktion (Begin of Transaction)
- COMMIT
  - Erfolgreiches Ende
- ROLLBACK
  - Abbruch

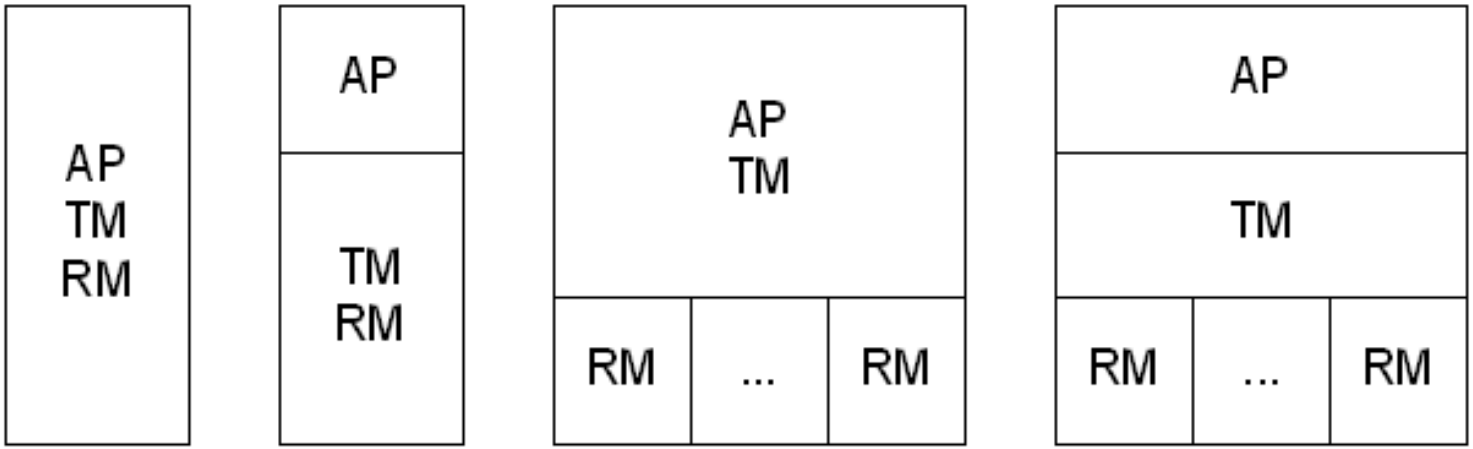


Kontrollfluss zwischen Datenbanksystem und Anwendungsprogramm

Anwendungsprogramm		**Datenbanksystem**
Transaktionsbeginn	$\longleftrightarrow$	Sicherstellen der Rücksetzbarkeit von Änderungsoperationen
$\downarrow$		
Aufruf einer Datenbankoperation	$\longleftrightarrow$	Änderungen ausführen Überprüfen von unverzögerten Integritätsbedingungen
$\downarrow$		
Transaktionsende	$\longrightarrow$	Überprüfen der verzögerten Integritätsbedingungen Sicherstellen der Wiederholbarkeit aller Änderungen
		$\downarrow$
Weiterarbeit Anwendungsprogramm	$\longleftarrow$	Sichtbarmachen von Änderungen Bestätigung über erfolgreiches Ende

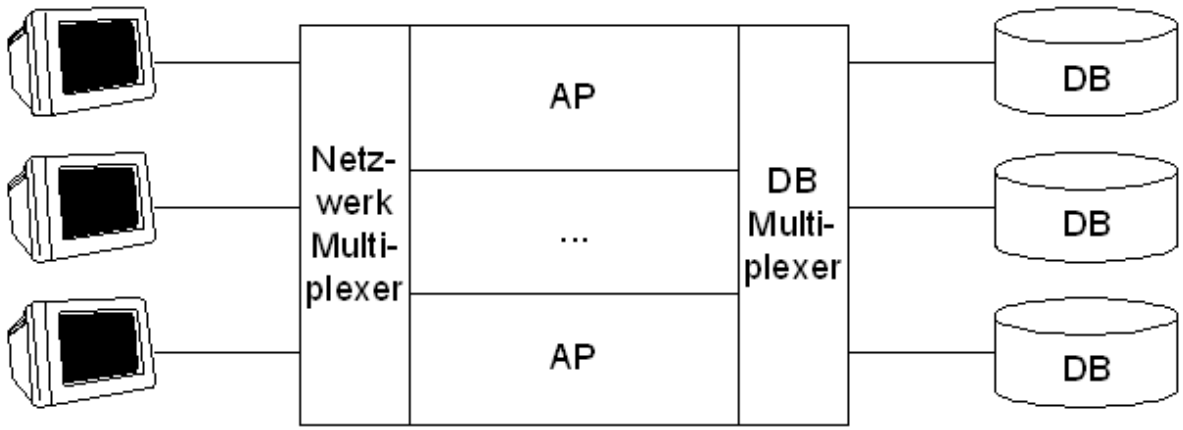
Architekturvarianten

- AP: Anwendungsprogramm
- TM: Transaktionsmanager
- RM: Ressourcenmanager



Transaction-Processing-Monitor (TP-Monitor)

- Betriebssystem für Transaktionsverarbeitung
- Reduktion der Anwendungsentwicklungs-Komplexität
- Effiziente Ressourcen-Nutzung
- Hoher Durchsatz



Vermeidung von Anomalien

- Anomalien
  - Nebeneffekte durch verzahnten Ablauf von Transaktionen
- Sperren
  - Erreichung der Isolation durch Kontrolle der Nebenläufigkeit
- Konsistenzstufen
  - Umsetzung der Isolation in SQL

Anomalie: Lost Update

- a = 80 freie Plätze
- 5 Neubuchungen, 4 Stornierungen
- Erwartetes Ergebnis für a: 79 freie Plätze
- Tatsächliches (falsches) Ergebnis für a: 84 freie Plätze

T1: Neubuchungen	**T2: Stornierungen**
`read a`	
`a = a - 5`	
	`read a`
	`a = a + 4`
`write a`	
	`write a`

Anomalie: Dirty Read

- a = 1 freier Platz
- Eine Neubuchung → keine freien Plätze mehr
- T2 führt keine Aktion aus, da keine freien Plätze mehr
- T1 bricht ab. T2 hätte buchen können, hat aber einen nicht bestätigten Wert gelesen

T1: Reservierung letzter Platz	**T2: Versuch einer Reservierung**
`read a`	
`a = a - 1`	
`write a`	
	`read a`
	Anscheinend keine Plätze frei
	Keine weitere Aktion
Reservierung wird nicht bestätigt	
Abbruch der Transaktion	

Anomalie: Non-repeatable Read

- Wiederholtes Lesen von a in T2 führt zu unterschiedlichen Werten
- Phantome sind eine spezielle Form des non-repeatable read. Neue Datensätze erscheinen im Laufe der Bearbeitung

T1: Neubuchungen	**T2: Wiederholtes Lesen**
	`read a`
`read a`	
`a = a - 5`	
`write a`	
	`read a`

Isolation durch Setzen von Sperren

- read, write
  Lesen und Schreiben von Daten. Entspricht select und update von SQL
- slock (shared lock)
  Sperren auf Daten, die von mehreren Transaktionen gemeinsam gesetzt werden können
- xlock (exclusive lock)
  Exklusive Sperre, kann nur von einer Transaktion gesetzt werden
- unlock
  Aufhebung von Sperren
- bot, commit, rollback
  Starten einer Transaktionen, erfolgreiches Abschließen bzw. Verwerfen der durchgeführten Operationen

Transaktionen mit Sperren

- Transaktionen sind Listen von Aktionen, die mit `begin` starten und mit `commit` oder `rollback` enden

T1	**T2**
`bot`	`bot`
`slock a`	`slock a`
`xlock b`	`read a`
`read a`	`xlock b`
`write b`	`write b`
`commit`	`rollback`

Ersetzung von `bot`, `commit` und `rollback`

- Weglassen von `bot`
- Endet eine Transaktion mit `commit`, so wird dieses durch eine Liste von `unlock`-Operationen ersetzt. Für jedes `slock a` oder `xlock a` wird ein `unlock a` erzeugt
- Endet eine Transaktion mit `rollback`, so wird dieses durch eine Liste von `write`- und `unlock`-Operationen ersetzt. Die `unlock`-Operationen werden wie im `commit`-Fall erzeugt. Die `write`-Operationen schreiben wieder die alten Werte zurück und bewirken damit ein Rückgängigmachen der ausgeführten Operationen

T1	**T2**
`slock a`	`slock a`
`xlock b`	`read a`
`read a`	`xlock b`
`write b`	`write b`
`unlock a`	`write b (undo)`
`unlock b`	`unlock a`
	`unlock b`

Verträglichkeit von Sperren

		Erteilter Modus	
		`slock`	`xlock`
Angeforderter	`slock`	$+$[1]	$-$
Modus	`xlock`	$-$[2]	$-$

[1] $+$: verträglich
[2] $-$: nicht verträglich

Sperrprotokolle

- Varianten für das Setzen von Sperren
  - Keine Sperren
  - Setzen von `slock` vor dem Lesen von Daten
  - Setzen von `xlock` vor dem Schreiben von Daten
- Sperrdauer
  - Kurze Sperren: direkt nach dem Benutzen wieder freigeben (vor Transaktionsende)
  - Lange Sperren: erst am Transaktionsende freigeben

Lösung für Lost Update mit Sperren

T1	**T2**
`xlock a`	
`read a`	`xlock a` (blockiert, da nicht verträglich)
`a = a - 5`	
`write a`	
`unlock a`	
	`read a` (Sperre kann jetzt erteilt werden)
	`a = a + 4`
	`write a`
	`unlock a`

Hierachische Sperren

- Sperrgranularität/-hierachie
  - Spalte, Datensatz, Seite, Tabelle, Datei, Datenbank
- Erweiterte Sperrmodi
  - IX: Vorhaben eine gemeinsame oder exklusive Sperre auf einer niedrigeren Ebene zu setzen (intend exclusive)
  - IS: Vorhaben eine gemeinsame Sperre auf einer niedrigeren Ebene zu setzen (intend share)
  - SIX: Setzen einer gemeinsamen Sperre auf höherer Ebene und Vorhaben eine exklusive Sperre auf einer niedrigeren Ebene zu setzen (shared and intend exclusive)
  - U: Setzen einer gemeinsamen Sperre, die ggf. in eine exklusive Sperre umgewandelt werden kann

Kompatibilitätsmatrix

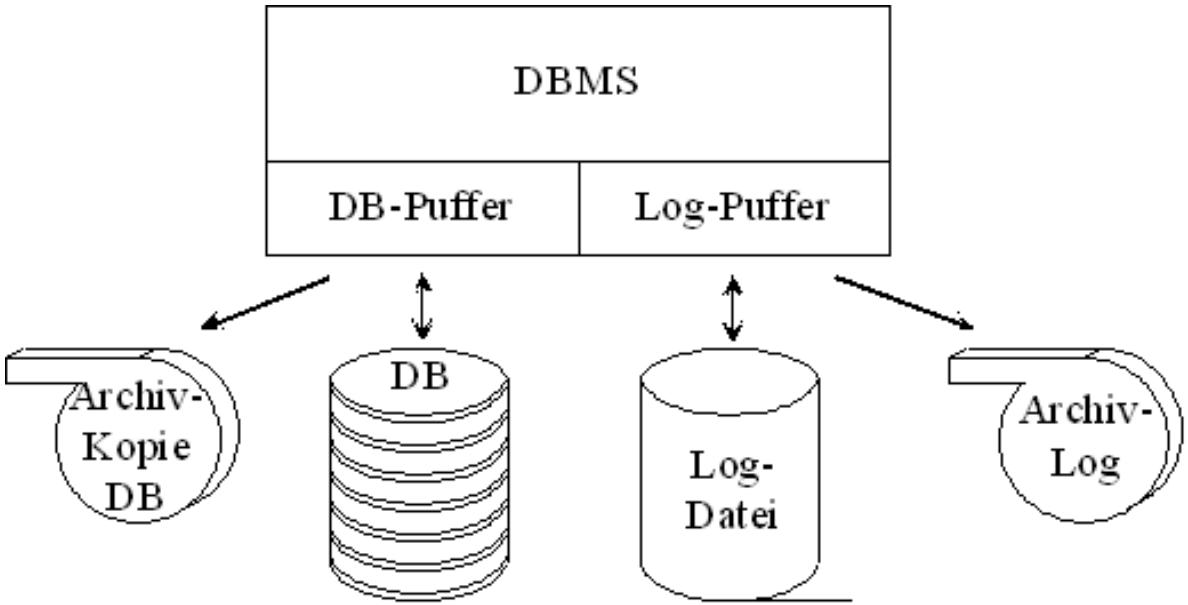
		Erteilter Modus					
		IS	IX	S	SIX	U	X
	IS	+	+	+	+	−	−
Ange-	IX	+	+	−	−	−	−
forder-	S	+	−	+	−	−	−
ter	SIX	+	−	−	−	−	−
Modus	U	−	−	+	−	−	−
	X	−	−	−	−	−	−

SQL-Konsistenzstufen

- Read Uncommitted
  Keine `slock`, Daten ändern nicht erlaubt
- Read Committed
  Kurze `slock`, lange `xlock`
- Repeatable Read
  Lange `slock`, lange `xlock`
- Serializable
  Lange `slock`, lange `xlock`, `xlock` auf höherer Ebene

		Anomalie		
		Dirty Read	Non-repeatable Read	Phantome
Konsistenzstufe	Read Uncommitted	+	+	+
	Read Committed	−	+	+
	Repeatable Read	−	−	+
	Serializable	−	−	−

Systemkomponenten



- Fehlerarten
  - Transaktionsfehler
  - Systemfehler
  - Gerätefehler

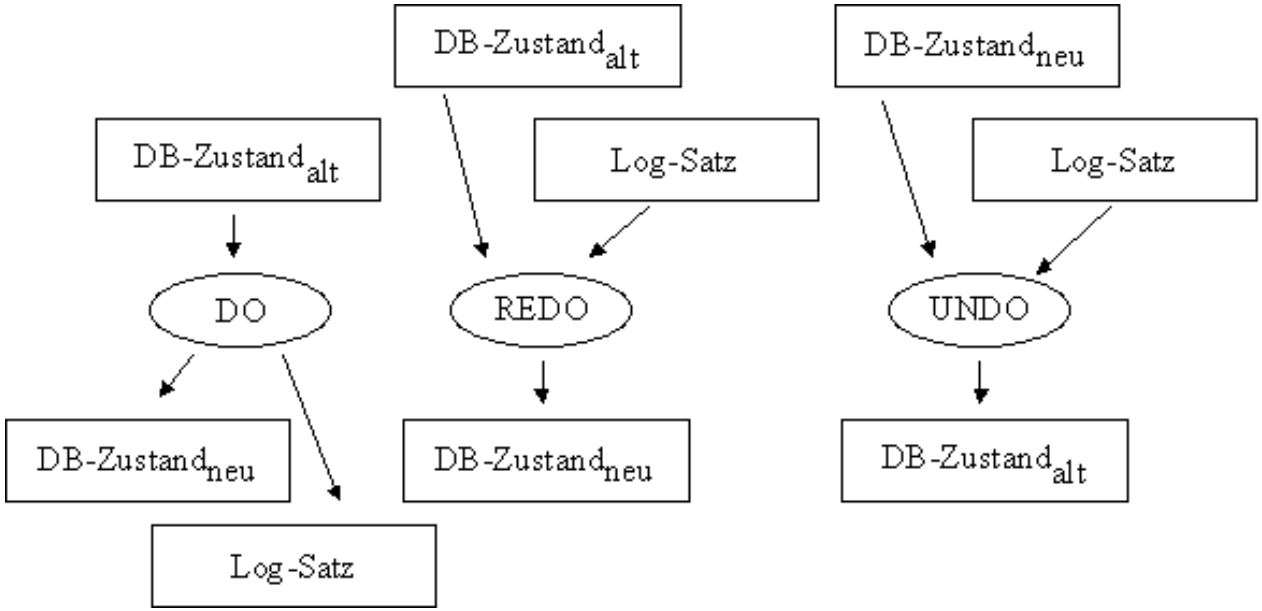
Ausschreibungsstrategien

- Änderungen vor Commit
  - Nosteal: Keine Ausschreibung „schmutziger“ Änderungen
  - Steal: Ausschreibung „schmutziger“ Änderungen
- Beim Commit
  - Force: Ausschreibung von Änderungen vor Abschluss Commit
  - Noforce: Kein Zwang zur Ausschreibung von Änderungen vor Abschluss Commit

	Steal	**NoSteal**
Force	Undo-Recovery Kein Redo-Recovery	nicht möglich
NoForce	Undo-Recovery Redo-Recovery	Kein Undo-Recovery Redo-Recovery

Protokollierung (Logging)

- Do-Redo-Undo-Prinzip



- Prinzipieller Aufbau von Protokolleinträgen
  - T = Transaktions-ID
  - [start transaction, T]
  - [write, T, x, Wert vorher, Wert nacher]
  - [commit, T]

Protokolierungsregeln

- Write-Ahead-Log-Prinzip
  - Erst Undo-Informationen in Log schreiben
  - Auf Festplatte sichern
  - Dann „schmutzige“ Änderungen in die DB einbringen
- Commit-Regel
  - Erst Redo-Informationen in Log schreiben
  - Auf Festplatte sichern
  - Dann Commit durchführen
- Wiederherstellung nach Systemfehler
  - Undo aller Änderungen ohne Commit im Log
  - Redo aller Änderungen mit Commit Log

Gesamtsicherung und Wiederherstellung (Backup)

- Speicherung auf einem externen Medium (z. B. Band)
  - Gesamte DB
  - Log-Dateien
- Zeitlicher Rahmen
  - Gesamte DB selten, dauert lange, kein normaler Betrieb
  - Log-Dateien häufig
- Recovery
  - Gesamte DB einspielen
  - Log abarbeiten: Redo aller Transaktionen mit Commit

Beispiele mit Konsistenzstufen

- Tabelle

Plaetze		
VeranstaltungsNr	Bezeichnung	FreiePlaetze
99841	DMDB	37
6121810	DBTECH	1
6122812	REALDBAS	21

Konsistenzstufe: `Read Committed`

- Transaktion 1

```
update PLAETZE
set FREIE_PLAETZE = FREIE_PLAETZE - 5
where VERANSTALTUNGSNR = 99841
```

- Transaktion 2

```
select * from PLAETZE
```

- Ergebnis
  - Transaktion 2 bleibt stehen, da Transaktion 1 eine Sperre auf Zeile 99841 setzt.
  - Damit wird das Lesen nicht bestätigter Änderungen verhindert.

Konsistenzstufe: `Read Committed`

- Transaktion 1

```
select * from PLAETZE
```

- Transaktion 2

```
update PLAETZE
set FREIE_PLAETZE = FREIE_PLAETZE - 5
where VERANSTALTUNGSNR = 99841
```

- Ergebnis
  - Transaktion 2 läuft durch, da sie Ihre Sperren trotz des Lesens von Transaktion 1 setzen kann.

Konsistenzstufe: `Read Committed`

- Transaktion 1

```
update PLAETZE
set FREIE_PLAETZE = FREIE_PLAETZE - 5
where VERANSTALTUNGSNR = 99841
```

- Transaktion 2

```
update PLAETZE
set FREIE_PLAETZE = FREIE_PLAETZE - 3
where VERANSTALTUNGSNR = 6122812
```

- Ergebnis
  - Beide Transaktionen laufen durch, da die Sperren auf verschiedene Zeilen gesetzt werden.

Konsistenzstufe: `Read Uncommitted`

- Transaktion 1

```
update PLAETZE
set FREIE_PLAETZE = FREIE_PLAETZE - 5
where VERANSTALTUNGSNR = 99841
```

- Transaktion 2

```
select * from PLAETZE
```

- Ergebnis
  - Transaktion 2 läuft durch und berücksichtigt auf Grund der Konsistenzstufe Read-Uncommitted die Sperren nicht.
  - Damit wird das Lesen nicht bestätigter Änderungen zugelassen.

Konsistenzstufe: `Repeatable Read`

- Transaktion 1

```
select * from PLAETZE
```

- Transaktion 2

```
update PLAETZE
set FREIE_PLAETZE = FREIE_PLAETZE - 5
where VERANSTALTUNGSNR = 99841
```

- Ergebnis
  - Transaktion 2 bleibt stehen.
  - So wird verhindert, das Transaktion 1 beim nochmaligen Lesen der Daten durch Transaktion 2 geänderte Werte erhält.

Konsistenzstufe: `Repeatable Read`

- Transaktion 1
  ```
  select * from PLAETZE
  ```
- Transaktion 2
  ```
  insert into PLAETZE
  values (6122814,'DB4',10)
  ```
- Ergebnis
  - Transaktion 2 läuft durch, da Repeatable-Read Phantome zulässt.
  - Ein nochmaliges Lesen innerhalb der noch offenen Transaktion 1 würde die eingefügte Zeile anzeigen.

Konsistenzstufe: `Serializable`

- Transaktion 1
  ```
  select * from PLAETZE
  ```
- Transaktion 2
  ```
  insert into PLAETZE
  values (6122814,'DB4',10)
  ```
- Ergebnis
  - Transaktion 2 bleibt stehen, da Serializable keine Phantome zulässt.
  - Um dieses Verhalten zu erreichen muss eine Sperre auf die gesamte Plätze-Tabelle gesetzt werden.

Lokale Transaktionen

- Methoden zur Transaktionssteuerung in der Schnittstelle `Connection`
- Standardmäßig arbeiten Verbindungen im AutoCommit-Modus, d. h. jede einzelne SQL-Anweisung wird als eigenständige Transaktion betrachtet ohne `commit` bzw. `rollback` aufzurufen.
- Transaktionen werden nicht explizit gestartet. Der Start erfolgt implizit durch die erste SQL-Anweisung bzw. durch die erste SQL-Anweisung nach Aufruf des letzten `commit` bzw. `rollback`.

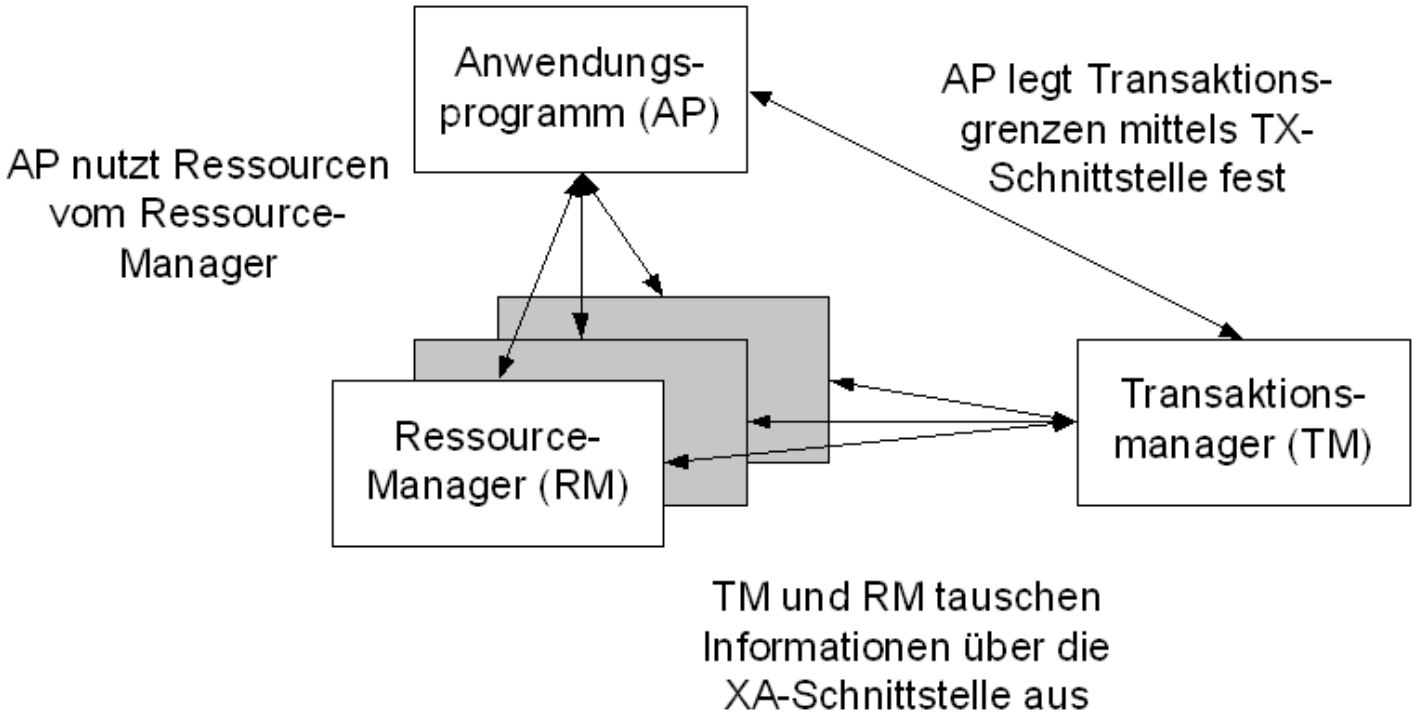
```
public interface Connection {
  public static final int TRANSACTION_READ_UNCOMMITTED;
  public static final int TRANSACTION_READ_COMMITTED;
  public static final int TRANSACTION_REPEATABLE_READ;
  public static final int TRANSACTION_SERIALIZABLE;
   ...
  public void setAutoCommit(boolean autoCommit);
  public void setTransactionIsolation(int level);
  public void commit();
  public void rollback();
```

Ausführung einer lokalen Transaktion

```
try {
  // AutoCommit ausschalten, damit mehrere SQL-Anweisungen zu einer
  // Transaktion zusammengefasst werden können
  connection.setAutoCommit(false);
  // Ggf. SQL-Konsistenzstufe setzen
  connection. setTransactionIsolation(...);
  // Zwei Datenbankoperationen ausführen
  Statement statement = connection.createStatement();
  statement.executeUpdate(...);
  ...
  statement.executeUpdate(...);
  ...
  // Transaktion erfolgreich beenden
  connection.commit();
} catch (SQLException e) {
  ...
  connection.rollback();
} finally {
  // Ressourcen freigeben
}
```

Verteilte Transaktionen

- Transaktionen, die mehr als eine Ressource umfassen
- Typische Ressource-Manager sind Datenbank- oder Messaging-Systeme
- Beteiligte



Anforderungen

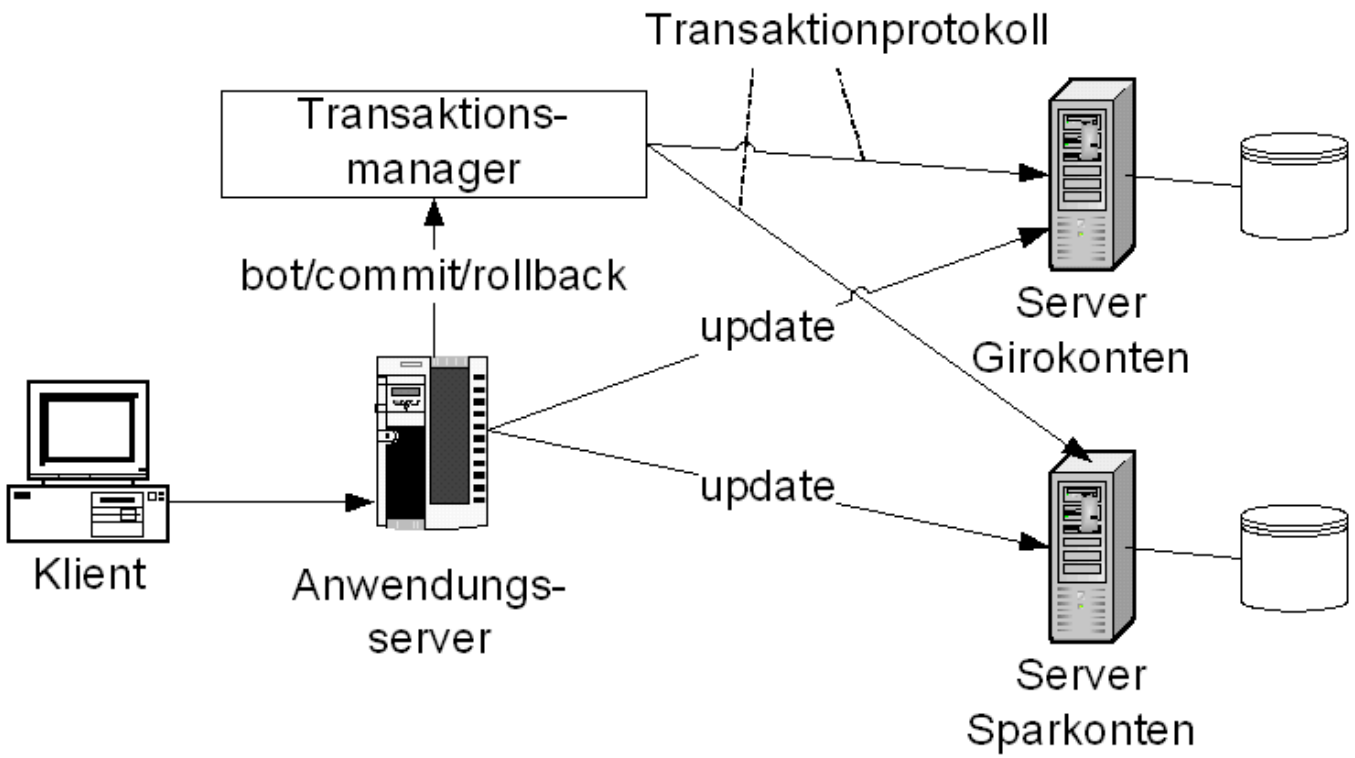
- ACID muss wie bei nicht verteilten Transaktionen gewahrt werden.
- Insbesondere müssen für die Atomarität alle RM entweder die Transaktion erfolgreich beenden oder alle ihre Änderungen rückgängig machen.
- Dafür ist ein spezielles Protokoll erforderlich, das sogenannte Zwei-Phasen-Commit-Protokoll, das die beteiligten RM steuert.
- Dieses Protokoll muss insbesondere mit Systemfehlern umgehen:
  - Kommunikationsfehler, wenn die beteiligten RM auf verschiedenen Servern laufen, die über ein Netzwerk miteinander verbunden sind.
  - TM stürtzt ab, RM stürzt ab.
  - Der Wiederherstellungsprozess muss die notwendigen Aktionen zur Sicherstellung der Atomarität durchführen.

Spezifikationen

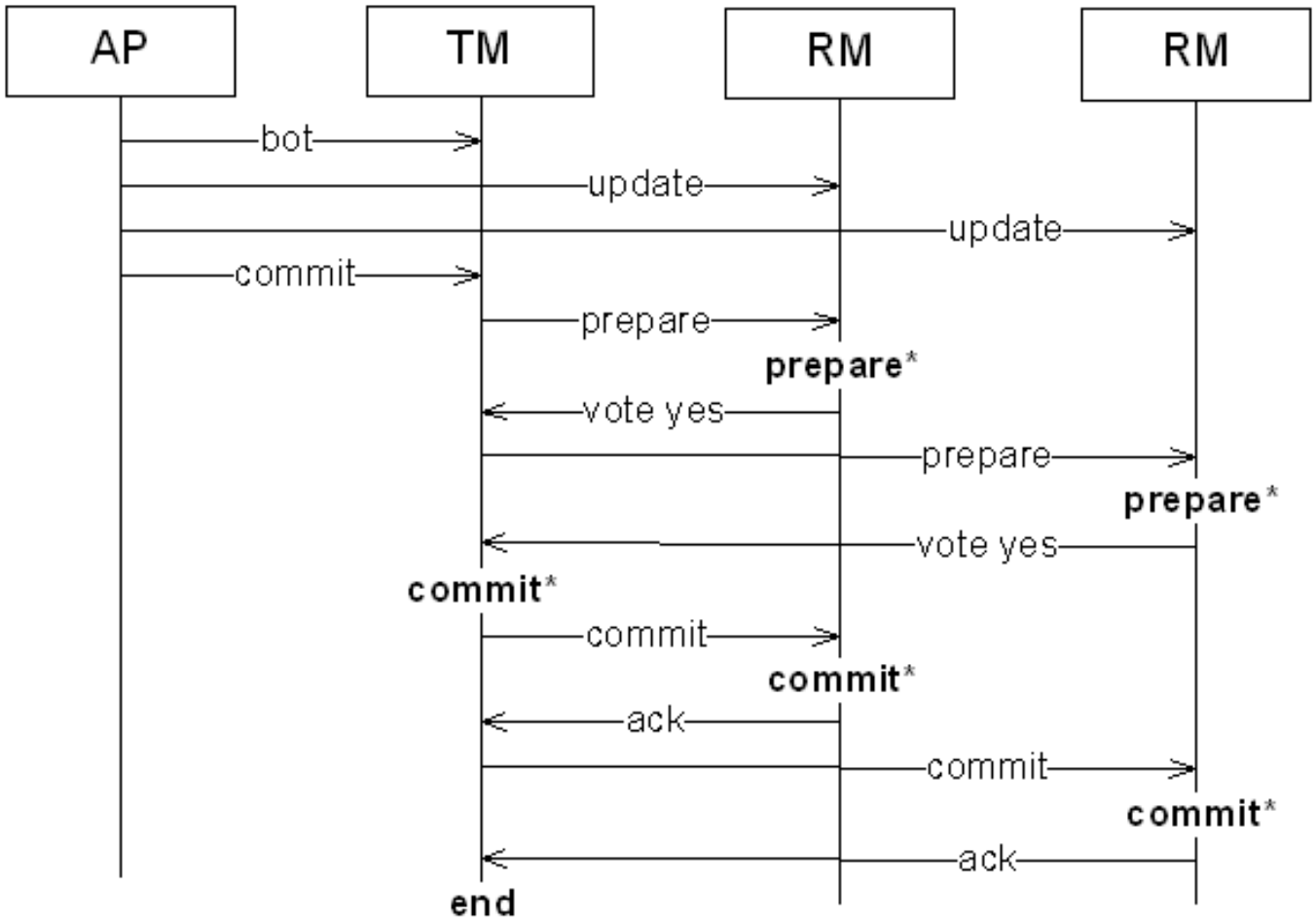
- Grundlage bildet das Distributed-Transaction-Processing-Modell (DTP)
- TX-Spezifikation
  - Beschreibt die Schnittstelle zwischen dem AP und dem TM.
  - Das AP steuert die Transaktion durch Anweisungen zum Starten (bot), erfolgreichem Beenden (commit) und Abbruch (rollback).
- XA-Spezifikation
  - Beschreibt die Schnittstelle zwischen dem TM und den RM.
  - Mit Funktionen dieser Schnittstelle können Operationen im RM mit einer Transaktion assoziiert werden.
  - Stellt Funktionen zur Abbildung des Zwei-Phasen-Commit-Protokolls bereit.

Beispiel: Geld zwischen Giro- und Sparkonto überweisen

- Annahme: Giro- und Sparkonto werden von verschiedenen Unternehmenseinheiten verwaltet, die jeweils eine eigene IT-Infrastruktur haben

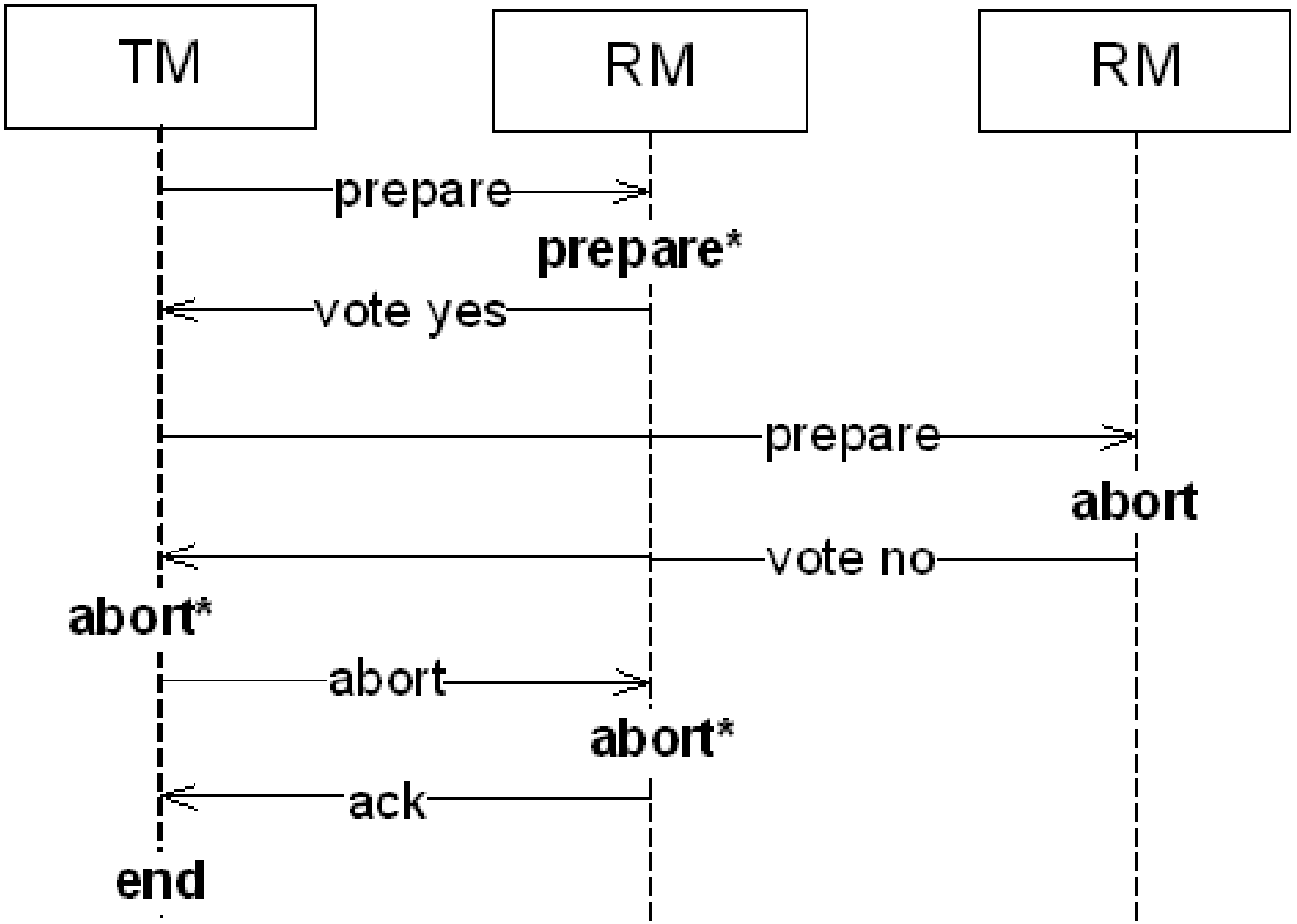


Zwei-Phasen-Commit-Protokoll (erfolgreicher Abschluss)



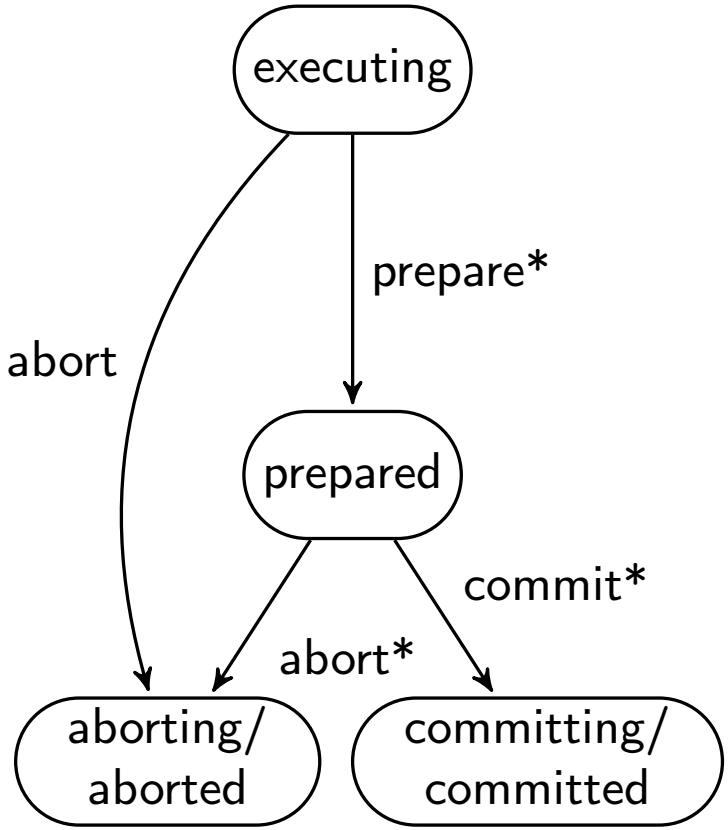
*Speicherung von Protokolleinträgen auf Platte erzwingen (force write)

Zwei-Phasen-Commit-Protokoll (Fehlerabbruch)



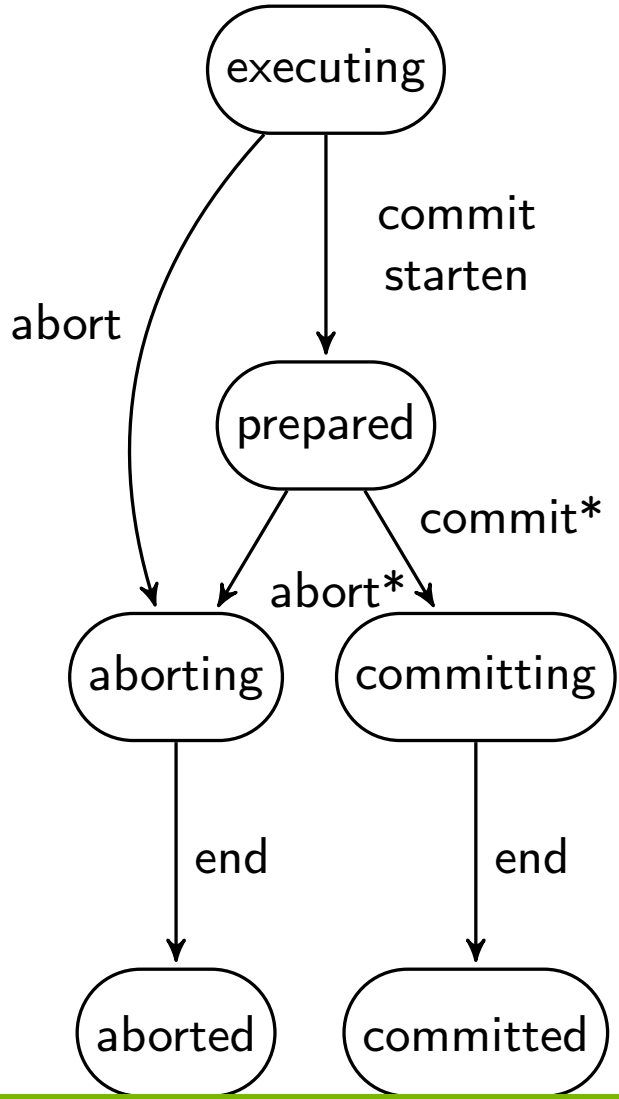
Zustandsübergänge in RM

- executing
  RM führt Operationen durch, z. B. Änderungen an Daten. Bei Fehler kann sofort in den aborting/aborted-Zustand gewechselt werden. Nach der prepare-Aufforderung durch den TM wird bei positivem Ergebnis in den prepared-Zustand gewechselt. Der RM muss auf die Entscheidung durch den TM warten.
- prepared
  Abhängig von der Entscheidung des TM wird die Transaktion erfolgreich beendet oder es findet ein Rückgängigmachen aller Operationen statt.
- aborting/aborted
  Transaktion wurde abgebrochen
- commiting/committed
  Transaktion wurde erfolgreich beendet



Zustandsübergänge in TM

- executing
  - TM startet commit-Bearbeitung. Sendet prepare-Meldung an alle RM.
  - Kann aber Transaktion auch abbrechen.
- prepared
  - TM wartet auf Ergebnisse von den RM.
  - Antworten alle RM positiv geht er in den Zustand committing über.
  - Antwortet nur ein RM negativ geht er in den aborting-Zustand über.
- aborting, commiting
  - TM warted auf Bestätigungen.
- aborted/committed
  - Information zur Transaktion wird in den Programmstrukturen des TM gelöscht, da sie komplett abgeschlossen ist.



Heuristische Entscheidungen

- RM haben dem prepare-Aufruf mit vote yes geantwortet und danach stürzt der TM ab oder kann aus anderen Gründen nicht das Ende der Transaktion einleiten
  - Alle Sperren müssen im RM gehalten werden.
  - Andere Transaktionen auf diesem RM werden möglicherweise blockiert.
- Pragmatische Lösung für diesen Fall: RM können von sich aus, d. h. heuritisch, eine Entscheidung zum Ausgang der Transaktion treffen, entweder commit oder abort.
  - Dadurch wird die Atomarität der Transaktion durchbrochen, da der TM möglicherweise eine andere Entscheidung zum Ausgang getroffen hat.
  - In diesem Fall muss ggf. manuell wieder ein konsistenter Gesamtzustand des Systems hergestellt werden.

Lokale und globale Transaktionen in Java

- Lokale Transaktionen behandeln den nicht verteilten Fall und werden in JDBC komplett über die Schnittstelle `Connection` abgebildet.
- Globale Transaktionen entsprechen verteilten Transaktionen und werden über die Java Transaction API (JTA) gesteuert. Diese stellt Schnittstellen und Klassen zur Verfügung, die das Zwei-Phasen-Commit-Protokoll realisieren.
- JDBC-Verbindungen können innerhalb lokaler und globaler Transaktionen verwendet werden, allerdings immer nur in einer der beiden Transaktionsarten.

Thread-Bindung von Transaktionen

- Pro Thread kann immer nur eine Transaktion existieren bzw. aktiv sein
- Transaktionen werden in ThreadLocal-Variablen gespeichert
- Der Transaktionsmanager bezieht seine Transaktion über den Thread in dem er aufgerufen wird.

Schnittstelle `TransactionManager`

Diese Schnittstelle dient zur programmtechnischen Steuerung von Transaktionen. Es verwaltet alle Transaktionen.

- `void begin()`
  Erzeugt eine neue Transaktion und verbindet sie mit dem aktuellen Thread.
- `void commit()`
  Schließt die mit dem Thread verbundene Transaktion positiv ab und löst die Verbindung zu diesem.
- `void rollback()`
  Schließt die mit dem Thread verbundene Transaktion negativ ab und löst die Verbindung zu diesem
- `Transaction getTransaction()`
  Liefert das Transaktionsobjekt, das mit dem aktuellen Thread verbunden ist.

Schnittstelle `Transaction`

Diese Schnittstelle beschreibt das Transaktionsobjekt, das pro Thread nur einmal existieren kann.

- `boolean enlistResource(XAResource xaRes)`
  Registriert die Resource in der Transaktion. Damit arbeitet die Resource im Kontext der Transaktion.
- `boolean delistResource(XAResource xaRes, int flag)`
  Entfernt die Registrierung der Resource aus der Transaktion.
- `void commit()`
  Schreibt Änderungen fest und schließt die Transaktion ab.
- `void rollback()`
  Macht Änderungen rückgängig und schließt die Transaktion ab
- `void registerSynchronization(Synchronization sync)`
  - Registriert ein Synchronisationsobjekt, das vor und nach der Abschlussphase der Transaktion informiert wird (callback).
  - Damit können Aktionen wie z. B. Zurückschreiben eines Objektzustandes in die Datenbank (flushing) vor Abschluss der Transaktion angestoßen werden.

Schnittstelle `XADataSource`

Eine XADataSource ist eine DataSource, die die XA-Spezifikation unterstützt, und damit in verteilten Transaktionen verwendet werden kann. Sie ist eine Fabrik für XAConnection-Objekte.

- `XAConnection getXAConnection(String user, String password)`
  Liefert eine XAConnection.

Schnittstelle `XAConnection`

Instanzen, die diese Schnittstelle implementieren, nehmen an verteilten Transaktionen teil. Zu jeder XAConnection gehört eine XAResource, die das Zwei-Phasen-Commit-Protokoll steuert.

- `XAResource getXAResource()`
- `Connection getConnection()`
- `void addConnectionEventListener(ConnectionEventListener listener)`
  Registriert ein Callback-Objekt, das bei Schließen der Verbindung aufgerufen wird.
- `void removeConnectionEventListener(ConnectionEventListener listener)`
  Hebt die Registrierung des Callback-Objekts auf.

Schnittstelle `Xid`

Instanzen, die diese Schnittstelle implementieren, sind Transaktions-Identifizierer. Damit werden Datenbankaktionen im Datenbankprotokoll gekennzeichnet um die Wiederherstellung (undo oder redo) zu ermöglichen. Xids sind strukturiert aufgebaut.

- `byte[ ] getGlobalTransactionId()`
  Der globale Anteil des Transaktions-Identifizierers, der für alle beteiligten Ressourcen in einer verteilten Transaktion gleich ist.
- `byte[ ] getBranchQualifier()`
  Der lokale Anteil des Transaktions-Identifizierers, der innerhalb einer Ressource eindeutig sein muss.
- `int getFormatId()`
  Kennzeichnung des Transaktionsmanagers, damit Transaktions-Identifizierer verschiedener Transaktionsmanager nicht zufälligerweise als gleich betrachtet werden.

Schnittstelle `XAResource`

Java-Abstraktion einer XA-Ressource entsprechend der XA-Spezifikation.

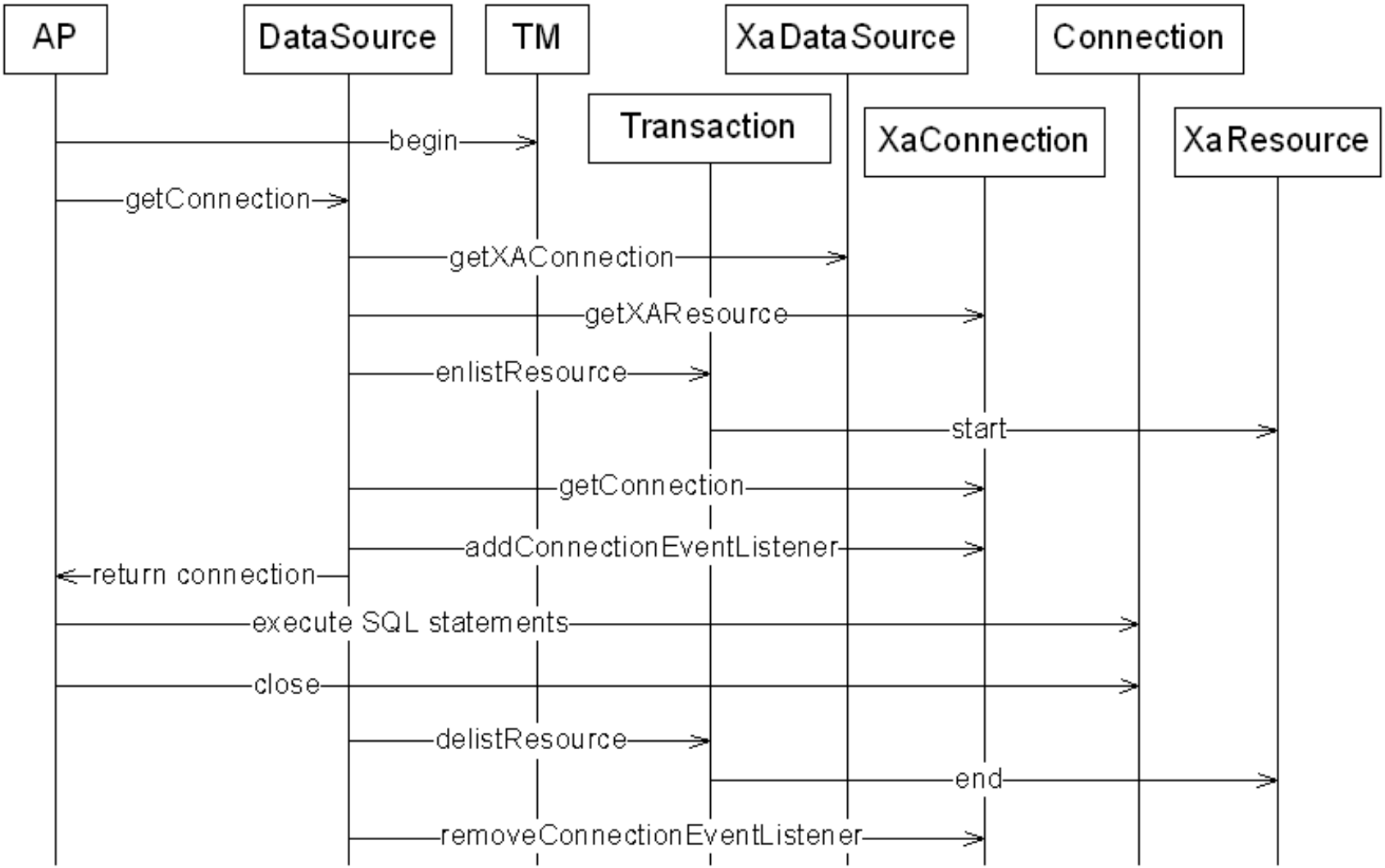
- `void start(Xid xid, int flags)`
  Nach Aufruf dieser Methode arbeitet die Ressource im Kontext der mit xid gekennzeichneten Transaktion.
- `void end(Xid xid, int flags)`
  Mit dieser Methode wird der Bezug zur Transaktion aufgehoben.
- `int prepare(Xid xid)`
  Mit dieser Methode wird die Ressource im Rahmen des Zwei-Phasen-Commit-Protokolls aufgefordert, sich auf den Abschluss der Transaktion vorzubereiten.
- `void commit(Xid xid, boolean onePhase)`
  Festschreiben der Änderungen.
- `void rollback(Xid xid)`
  Rückgängigmachen der Änderungen.

Schnittstelle `Synchronization`

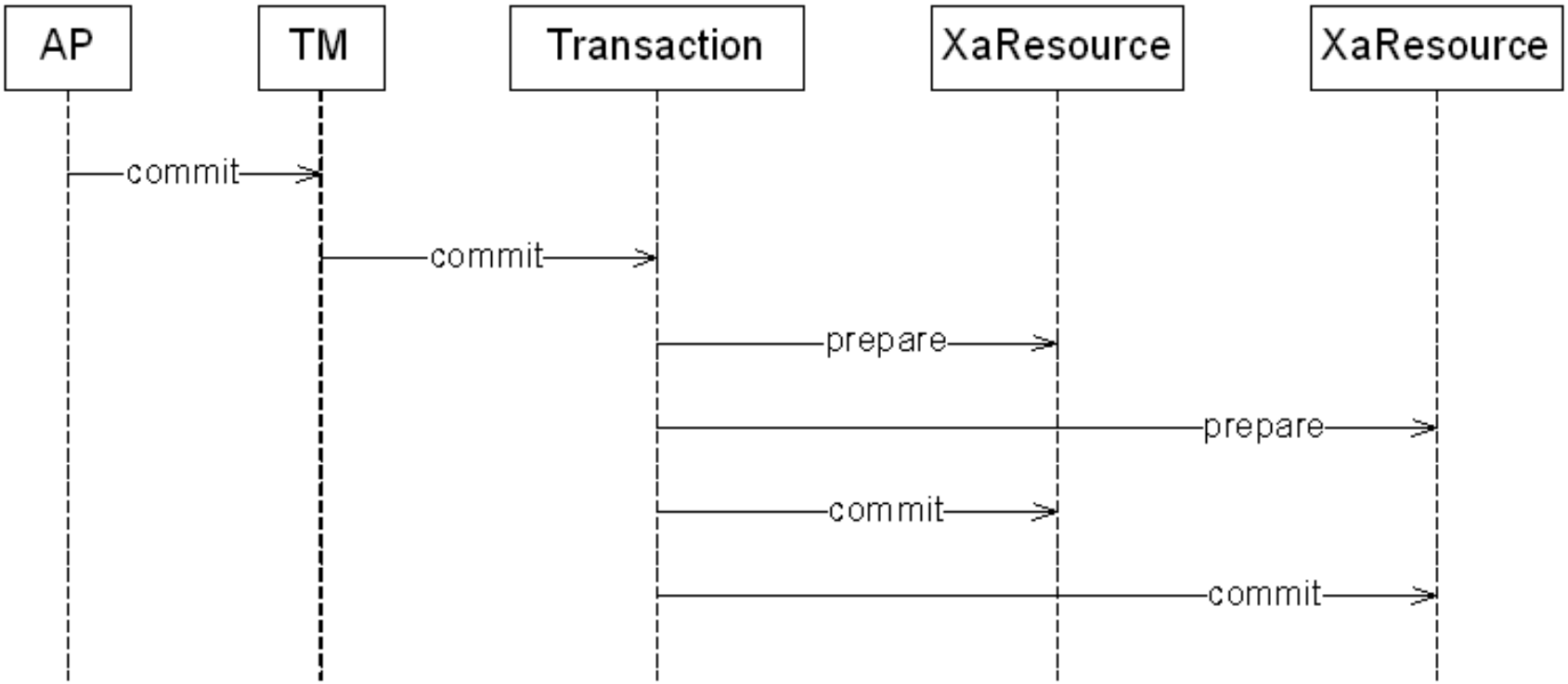
Instanzen, die diese Schnittstelle implementieren, dienen als Callback, um notwendige Aktionen vor Transaktionsabschluss durchführen zu können.

- `void beforeCompletion()`
  Wird vor prepare aufgerufen. Ausnahmen und Fehler in dieser Methode führen zum Abbruch (rollback) der Transaktion.
- `void afterCompletion(int status)`
  Wird nach kompletter Durchführung der Transaktion aufgerufen. Hat keinen Einfluss auf das Transaktionsergebnis mehr. Kann z. B. dazu genutzt werden, die programminternen Datenstrukturen mit der Datenbank abzugleichen.

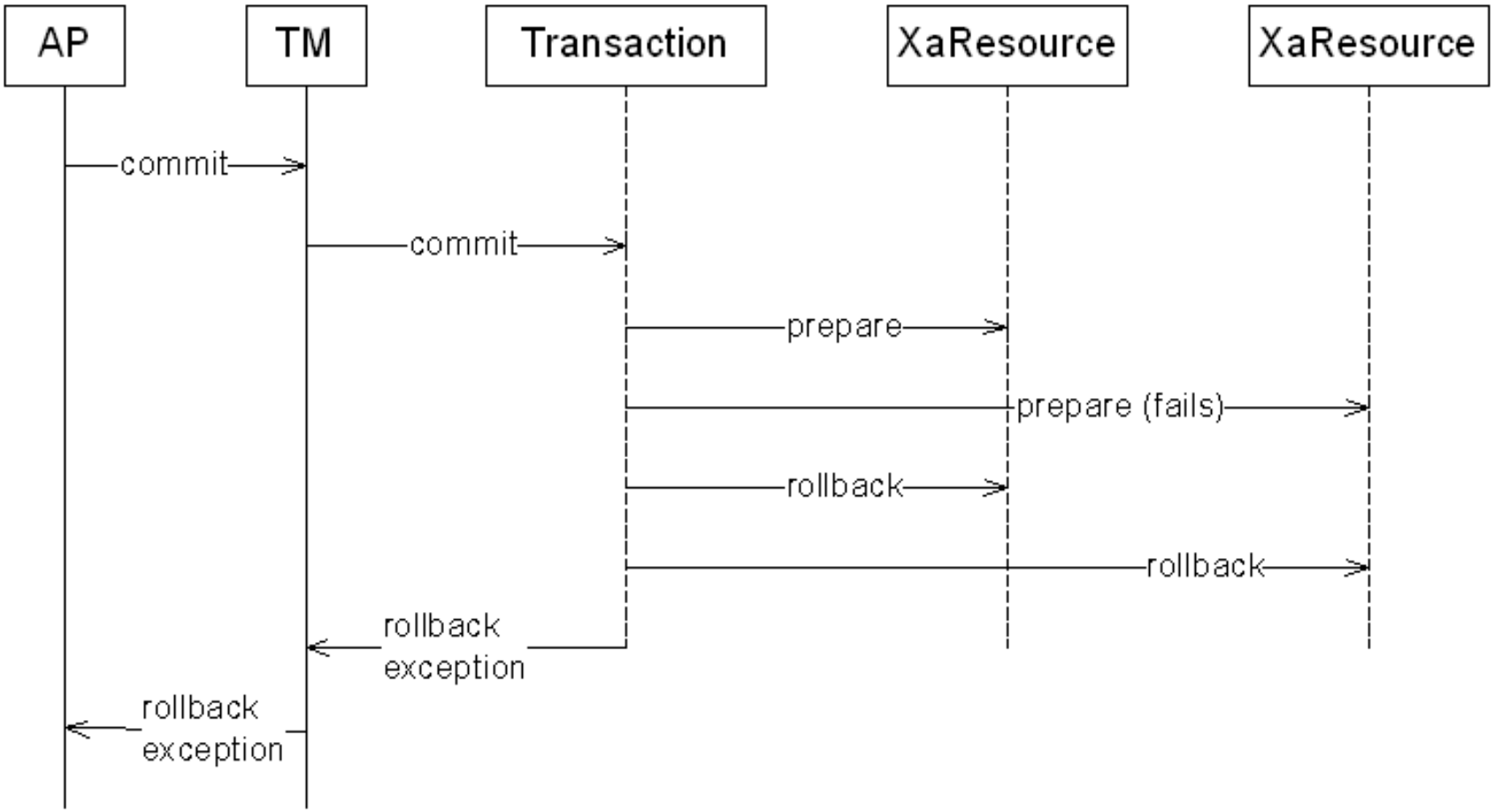
Arbeitsphase einer Transaktion



Abschlussphase commit



Abschlussphase rollback



Heuristische Entscheidung