

DATENBANKANWENDUNG

Wintersemester 2013/2014

Holger Schwarz
Universität Stuttgart, IPVS
holger.schwarz@ipvs.uni-stuttgart.de

Beginn: 23.10.2013
Mittwochs: 11.45 – 15.15 Uhr, Raum 46-268 (Pause 13.00 – 13.30)
Donnerstags: 10.00 – 11.30 Uhr, Raum 46-268
11.45 – 13.15 Uhr, Raum 46-260

<http://www.lgis.informatik.uni-kl.de/cms/courses/datenbankanwendung/>

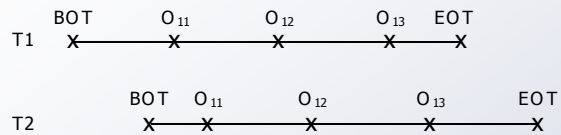


5. Transaktionsverwaltung

- **Transaktionsparadigma**
 - ACID-Eigenschaften
 - Programmierschnittstelle
 - Mögliche Ausgänge einer TA
- **Gefährdung der DB-Konsistenz**
- **Transaktionsverwaltung und Transaktionsablauf**
 - Architektur
 - SQL-Operationen: COMMIT WORK, ROLLBACK WORK
 - Zustände und Zustandsübergänge
 - Verarbeitung in verteilten Systemen
- **Commit-Protokolle**
 - Einsatz von Commit-Protokollen (zentralisierter TA-Ablauf)
 - **2PC (Zweiphasen-Commit-Protokoll)**
 - verteilter TA-Ablauf
 - Fehleraspekte
 - Kostenbetrachtungen
 - 2PC-Protokoll in Bäumen
 - Bestimmen eines Koordinators
 - Hierarchisches 2PC
 - TA-Verarbeitung in offenen Systemen
- **BASE (basically available, soft state, eventually consistent)**
 - eine ACID-Alternative
 - CAP-Theorem

Transaktionsparadigma

■ Ablaufkontrollstruktur: Transaktion



■ Transaktionsparadigma

- führt ein neues Verarbeitungsparadigma ein
- ist Voraussetzung für die Abwicklung betrieblicher Anwendungen (mission-critical applications)
- erlaubt „Vertragsrecht“ in rechnergestützten IS zu implementieren

5-3

Transaktionsparadigma (2)

■ Welche Eigenschaften von Transaktionen sind zu garantieren?

- **Atomicity (Atomarität)**
 - TA ist kleinste, nicht mehr weiter zerlegbare Einheit
 - Entweder werden alle Änderungen der TA festgeschrieben oder gar keine („alles-oder-nichts“-Prinzip)
- **Consistency**
 - TA hinterlässt einen konsistenten DB-Zustand, sonst wird sie komplett zurückgesetzt (siehe Atomarität)
 - Zwischenzustände während der TA-Bearbeitung dürfen inkonsistent sein
 - Endzustand muß die Integritätsbedingungen des DB-Schemas erfüllen
- **Isolation**
 - Nebenläufig (parallel, gleichzeitig) ausgeführte TA dürfen sich nicht gegenseitig beeinflussen
 - Parallele TA bzw. deren Effekte sind nicht sichtbar
- **Durability (Dauerhaftigkeit)**
 - Wirkung erfolgreich abgeschlossener TA bleibt dauerhaft in der DB
 - TA-Verwaltung muß sicherstellen, dass dies auch nach einem Systemfehler (HW- oder System-SW) gewährleistet ist
 - Wirkung einer erfolgreich abgeschlossenen TA kann nur durch eine sog. kompensierende TA aufgehoben werden

5-4

Transaktionsparadigma (3)

■ DB-bezogene Definition der Transaktion

Eine TA ist eine ununterbrechbare Folge v on DML-Befehlen, welche die Datenbank v on einem logisch konsistenten Zustand in einen neuen logisch konsistenten Zustand überführt.

⇒ Diese Definition eignet sich insbesondere für relativ kurze TA, die auch als **A CID-Transaktionen** bezeichnet werden.

■ Transaktionsprogramm – Beispiel:

```

BOT
UPDATE Konto
...
UPDATE Schalter
...
UPDATE Zweigstelle
...
INSERT INTO Ablage (...)
COMMIT;
    
```

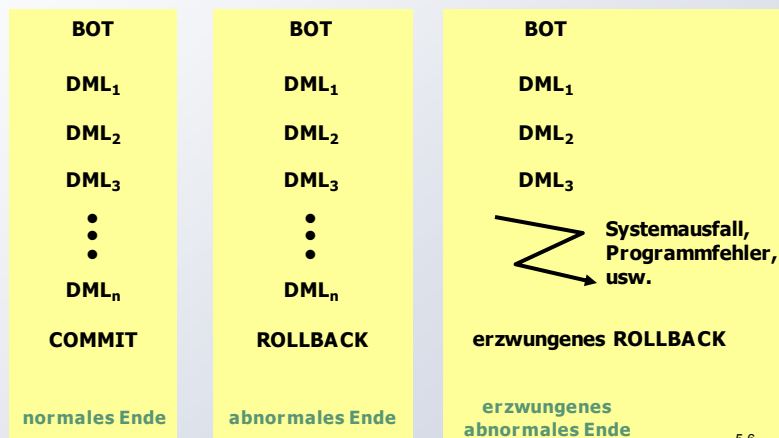
5-5

Transaktionsparadigma (4)

■ Programmierschnittstelle für Transaktionen:

- begin of transaction (BOT)
- commit transaction ("commit work" in SQL)
- rollback transaction ("rollback work" in SQL)

■ Mögliche Ausgänge einer Transaktion



5-6

Gefährdung der DB-Konsistenz

TA-Paradigma

DB-Konsistenz

TA-Verw.

Commit-
Protokolle

BASE



	Korrektheit der Abbildungshierarchie	Übereinstimmung zwischen DB und Miniwelt
durch das Anwendungs- programm	Mehrbenutzer-Anomalien Synchronisation	unzulässige Änderungen Integritätsüberwachung des DBVS TA-orientierte Verarbeitung
durch das DBVS und die Betriebsumgebung	Fehler auf den Externspeichern, Inkonsistenzen in den Zugriffs- pfaden Fehlertolerante Implementierung, Archiv kopien (Backup)	undefinierter DB-Zustand nach einem Systemausfall Transaktionsorientierte Fehler- behandlung (Recovery)

5-7

Transaktionsverwaltung

TA-Paradigma

DB-Konsistenz

TA-Verw.

Commit-
Protokolle

BASE



■ Wesentliche Abstraktionen aus Sicht der DB-Anwendung

- Alle Auswirkungen auftretender Fehler bleiben der Anwendung verborgen (*failure transparency*)
- Es sind keine anwendungsseitigen Vorkehrungen zu treffen, um Effekte der Nebenläufigkeit beim DB-Zugriff auszuschließen (*concurrency transparency*)

⇒ Gewährleistung einer fehlerfreien Sicht auf die Datenbank im logischen Einbenutzerbetrieb

■ Transaktionsverwaltung

- koordiniert alle DBS-seitigen Maßnahmen, um ACID zu garantieren
- besitzt drei wesentliche Komponenten zur/zum
 - Synchronisation
 - Logging und Recovery
 - Konsistenzsicherung/Regelverarbeitung
- kann zentralisiert oder verteilt (z.B. bei VDBS) realisiert sein
- soll Transaktionsschutz für heterogene Komponenten bieten

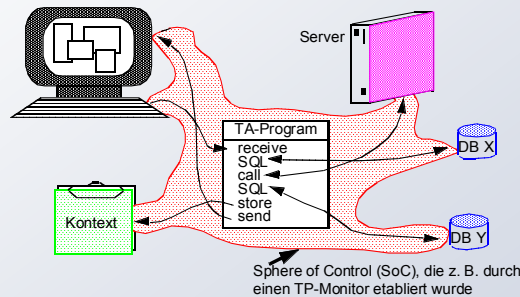
5-8

Transaktionsverwaltung (2)

■ Einsatz kooperierender Ressourcen-Manager (RM)

- RM sind Systemkomponenten, die **Transaktionsschutz für ihre gemeinsam nutzbaren Betriebsmittel (BM)** bieten
- RM gestatten die externe Koordination v von BM-Aktualisierungen durch spezielle Commit-Protokolle
- ⇒ Gewährleistung v **ACID für DB-Daten und auch für andere BM** (persistente Warteschlangen, Nachrichten, Objekte v von persistenten Programmiersprachen)

■ Ziel: TA-orientierte Verarbeitung in heterogenen Systemen



⇒ Die gesamte verteilte Verarbeitung in einer SoC ist eine ACID-TA

- alle Komponenten werden durch die TA-Dienste integriert
- für die Kooperation ist eine Grundmenge von Protokollen erforderlich

5-9

Transaktionsverwaltung (3)

■ Abstraktes Architekturmodell für die Transaktionsverwaltung (für das Read/Write-Modell auf Seitenbasis)

■ Transaktionsverwalter

- Verteilung der DB-Operationen in VDBS und Weiterreichen an den Scheduler
- zeitweise Deaktivierung v von TA (bei Überlast)
- Koordination der Abort- und Commit-Behandlung

■ Scheduler (Synchronisation)

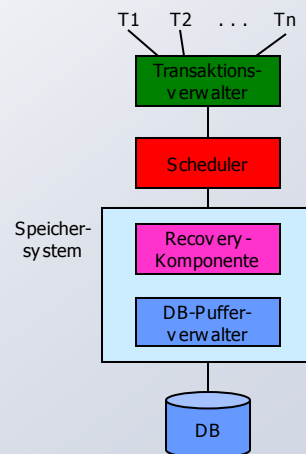
- kontrolliert die Abwicklung der um DB-Daten konkurrierenden TA

■ Recovery-Komponente

- sorgt für die Rücksetzbarkeit/Wiederholbarkeit der Effekte v von TA

■ DB-Pufferverwalter

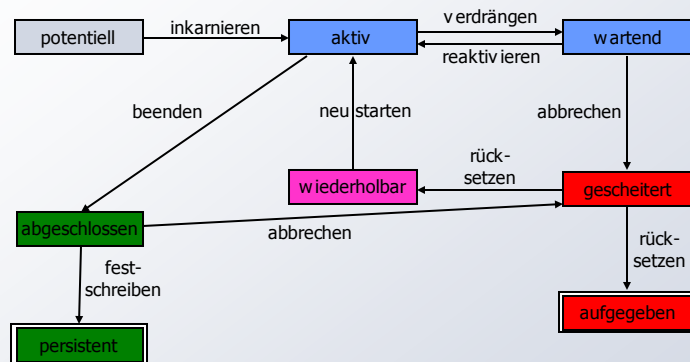
- stellt DB-Seiten bereit und gewährleistet persistente Seitenänderungen



5-10

Zustände einer Transaktion

■ Zustandsübergangs-Diagramm



■ Transaktionsverwaltung

- muß die möglichen Zustände einer TA kennen und
- ihre Zustandsübergänge kontrollieren/auslösen

5-11

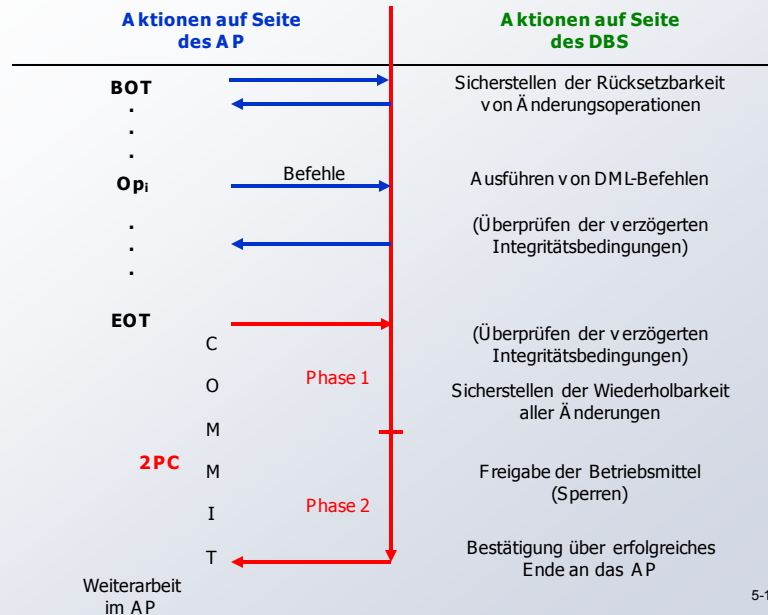
Zustände einer Transaktion (2)

■ TA-Zustände

- **potentiell**
 - TAP wartet auf Ausführung
 - Beim Start werden, falls erforderlich, aktuelle Parameter übergeben
- **aktiv**
 - TA konkurriert um Betriebsmittel und führt Operationen aus
- **wartend**
 - Deaktivierung bei Überlast
 - Blockierung z.B. durch Sperren
- **abgeschlossen**
 - TA kann sich (einseitig) nicht mehr zurücksetzen
 - TA kann jedoch noch scheitern (z.B. bei Konsistenzverletzung)
- **persistent** (Endzustand)
 - Wirkung aller DB-Änderungen werden dauerhaft garantiert
- **gescheitert**
 - Vielfältige Ereignisse können zum Scheitern ein TA führen (siehe Fehlermodell, Verklemmung usw.)
- **wiederholbar**
 - Gescheiterte TA kann ggf. (mit demselben Eingabewerten) erneut ausgeführt werden
- **aufgegeben** (Endzustand)

5-12

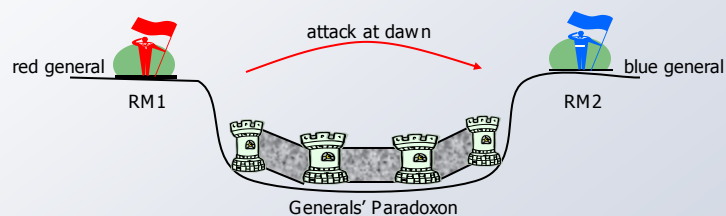
Schnittstelle zwischen AP und DBS – transaktionsbezogene Aspekte



5-13

Verarbeitung in Verteilten Systemen

- Ein **verteiltes System** besteht aus autonomen Subsystemen, die koordiniert zusammenarbeiten, um eine gemeinsame Aufgabe zu erfüllen
 - Client/Server-Systeme (C/S), Doppelrolle von Servern: Sie können wiederum als Clients anderer Server auftreten!
 - Mehrrechner-DBS, ...
- Beispiel:** The „Coordinated Attack“ Problem



Wie wird die Eindeutigkeit einer Entscheidung bei Unsicherheit in einer verteilten Situation erreicht?

5-14

Verarbeitung in Verteilten Systemen (2)

■ Grundproblem verteilter Systeme

Das für verteilte Systeme charakteristische Kernproblem ist der Mangel an globalem (zentralisiertem) Wissen

- ⇒ **symmetrische Kontrollalgorithmen sind oft zu teuer oder zu ineffektiv**
- ⇒ **fallweise Zuordnung der Kontrolle**

■ Annahmen im allgemeinen Fall

- nicht nur Verlust (omission) von Nachrichten (Bote/Kurier wird gefangen)
- sondern auch ihre bösartige Verfälschung (commission failures)

- ⇒ **dann komplexere Protokolle erforderlich:**
Distributed consensus (bekannt als Byzantine agreement)

5-15

Verarbeitung in Verteilten Systemen (3)

■ Fehlermodell

- **allgemein:**
 - Transaktionsfehler
 - Crash (i. Allg. einiger Rechner)
 - Gerätefehler
- **speziell** (nur omission failures):
 - Nachrichtenverluste
 - Nachrichten-Duplikate: dieselbe Nachricht erreicht ihr Ziel mehrmals, möglicherweise verschachtelt mit anderen Nachrichten
 - transiente Prozessfehler (soft crashes):
Restart erforderlich, aber kein Datenverlust auf Externspeicher
 - Achtung: Wir schließen hier bösartige Verfälschungen von Nachrichten aus!

- ⇒ **keine speziellen Annahmen:**
Kommunikationssystem arbeitet mit Datagrammen als einfachstem Typ von unbestätigten, sitzungsfreien Nachrichten

- ⇒ **schwierige Fehlererkennung, z. B. oft über Timeout**

5-16

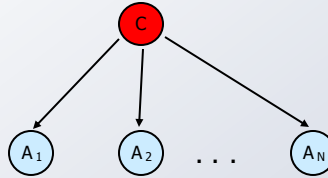
Verarbeitung in Verteilten Systemen (4)

■ Erweitertes Transaktionsmodell

verteilte Transaktionsbearbeitung (Primär-, Teiltransaktionen) –
zentralisierte Steuerung des Commit-Protokolls (vergleiche „Heirat“)

1 Koordinator

**N Teiltransaktionen
(Agenten)**



⇒ *rechnerübergreifendes Mehrphasen-Commit-Protokoll notwendig, um Atomicität einer globalen Transaktion sicherzustellen*

5-17

Commit-Protokolle

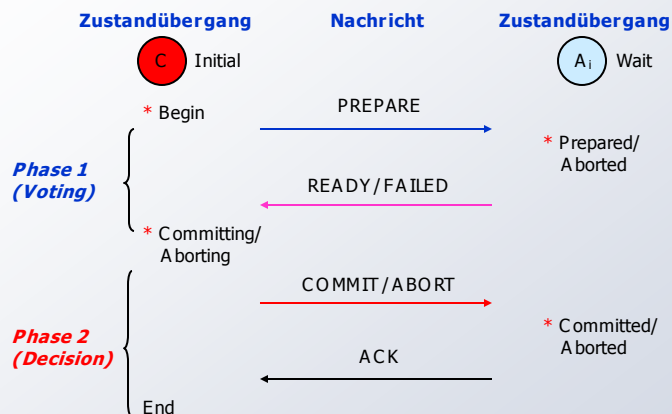
■ Allgemeine Anforderungen an geeignetes Commit-Protokoll:

- Geringer Aufwand (#Nachrichten, #Log-Ausgaben)
- Minimale Antwortzeitverlängerung (Nutzung von Parallelität)
- Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern

⇒ *Zentralisiertes Zweiphasen-Commit-Protokoll stellt geeignete Lösung dar*

5-18

Zentralisiertes Zweiphasen-Commit



* synchrone Log-Ausgabe (log record is force-written)
"End" ist wichtig für Garbage Collection im Log von C

5-19

Zentralisiertes Zweiphasen-Commit (2)

■ Protokoll (Basic 2PC)

- Folge von Zustandsänderungen für Koordinator und für Agenten
 - Protokollzustand auch nach Crash eindeutig: synchrones Logging
 - Sobald C ein NO VOTE (FAILED) erhält, entscheidet er ABORT
 - Sobald C die ACK-Nachricht von allen Agenten bekommen hat, weiß er, dass alle Agenten den Ausgang der TA kennen

⇒ C kann diese TA vergessen, d. h. ihre Log-Einträge im Log löschen!

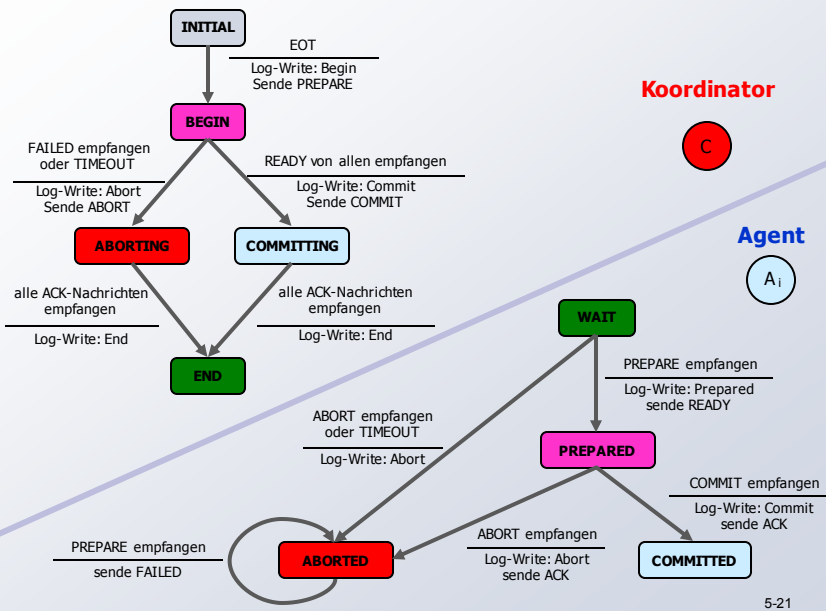
- Warum ist das 2PC-Protokoll blockierend?

■ Aufwand im Erfolgsfall:

- Nachrichten:
- Log-Ausgaben (forced log writes):

5-20

2PC: Zustandsübergänge



2PC: Fehlerbehandlung

Timeout-Bedingungen für Koordinator:

- BEGIN** ⇒ setze Transaktion zurück; versicke ABORT-Nachricht
- ABORTING** ⇒ vermerke Agenten, für die ACK noch aussteht
- COMMITTING**

Timeout-Bedingungen für Agenten:

- WAIT** ⇒ setze Teiltransaktion zurück (unilateral ABORT)
- PREPARED** ⇒ erfrage Transaktionsausgang bei Koordinator (bzw. bei anderen Rechnern)

Ausfall des Koordinator-knotens: Vermerkter Zustand auf Log

- END** ⇒ UNDO bzw. REDO-Recovery, je nach Transaktionsausgang keine "offene" Teiltransaktionen möglich
- ABORTING** ⇒ UNDO-Recovery
ABORT-Nachricht an Rechner, von denen ACK noch aussteht
- COMMITTING** ⇒ REDO-Recovery
COMMIT-Nachricht an Rechner, von denen ACK aussteht
- Sonst ⇒ UNDO-Recovery

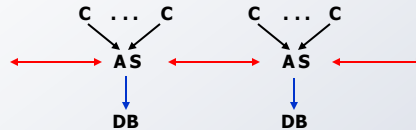
Rechnerausfall für Agenten: Vermerkter Zustand auf Log

- COMMITTED** ⇒ REDO-Recovery
- ABORTED** bzw. kein 2PC-Log-Satz vorhanden ⇒ UNDO-Recovery
- PREPARED** ⇒ Anfrage an Koordinator-Knoten, wie TA beendet wurde (Koordinator hält Information, da noch kein ACK erfolgte)

2PC-Protokoll in TA-Bäumen

■ Typischer AW-Kontext: C/S-Umgebung

- Beteiligte: Clients (C), Applikations-Server (AS), DB-Server (DB)
- unterschiedliche *Quality of Service* bei Zuverlässigkeit, Erreichbarkeit, Leistung, ..



5-23

2PC-Protokoll in TA-Bäumen (2)

■ Wer übernimmt die Koordinatorrolle? – wichtige Aspekte

- TA-Initiator
- Zuverlässigkeit und Geschwindigkeit der Teilnehmer
 - Anzahl der Teilnehmer
 - momentane Last
 - Geschwindigkeit der Netzwerkverbindung
- Kommunikationstopologie und -protokoll
 - sichere/schnelle Erreichbarkeit
 - im LAN: Netzwerkadressen aller Server sind bekannt; jeder kann jeden mit Datagrammen oder neu eingerichteten Sitzungen erreichen
 - im WAN: „transitive“ Erreichbarkeit; mehrere Hops erforderlich; Initiator kennt nicht unbedingt alle (dynamisch hinzukommenden) Teilnehmer (z. B. im Internet)

■ Im einfachsten Fall: TA-Initiator übernimmt Koordinatorrolle

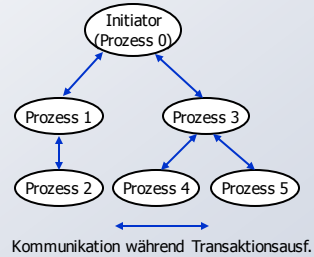
⇒ sinnvoll, wenn Initiator ein zuverlässiger, schneller und gut angebundener Applikations-Server ist!

5-24

2PC-Protokoll in TA-Bäumen (3)

■ Drei wichtige Beobachtungen

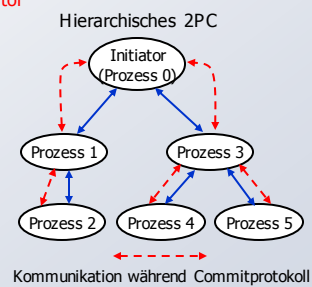
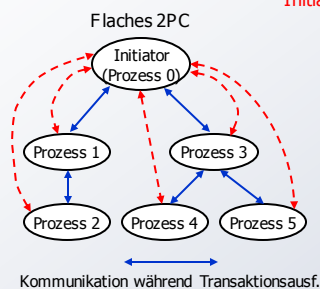
- Während der TA-Verarbeitung formen die involvierten Prozesse einen (unbalancierten) Baum mit dem Initiator als Wurzel. Jede Kante entspricht einer dynamisch eingerichteten Kommunikationsverbindung
- Für die Commit-Verarbeitung kann der Baum „flachgemacht“ werden
 - Der Initiator kennt die Netzwerkadressen aller Teilnehmerprozesse bei Commit (durch „piggy backing“ bei vorherigen Aufrufen)
 - Nicht möglich, wenn die Server die Information, welche Server sie aufgerufen haben, kapseln
- „Flachmachen“ kann als spezieller Fall der Restrukturierung der C/S-Aufrufhierarchie gesehen werden
 - Es könnte auch ein zuverlässiger innerer Knoten als Koordinator ausgewählt werden
 - „Rotation“ der Aufrufhierarchie um den neu gewählten Koordinatorknoten



5-25

2PC-Protokoll in TA-Bäumen (4)

Initiator = Koordinator

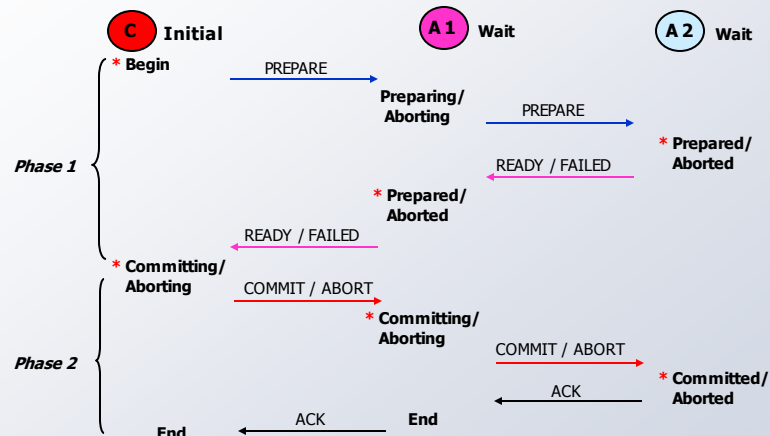


5-26

Hierarchisches 2PC

■ Allgemeineres Ausführungsmodell

- beliebige Schachtelungstiefe, angepasst an C/S-Modell
- Modifikation des Protokolls für "Zwischenknoten"



* synchrone Log-Ausgabe (log record is force-written)
 "End" ist wichtig für Garbage Collection im Log

5-27

Hierarchisches 2PC (2)

■ Aufwand im Erfolgsfall:

- Nachrichten:
- Log-Ausgaben (forced log writes):
- Antwortzeit steigt mit Schachtelungstiefe

■ Problem Koordinator-/Zwischenknotenausfall ⇒ Blockierung möglich!

5-28

Commit: Optimierungen

■ Beobachtung:

Commit-Protokoll

- hat erheblichen Einfluss auf den TA-Durchsatz
 - substanzieller Anteil an der gesamten TA-Zeit bei kurzen TAs (Reservierungen, Buchungen, Kreditkartenvvalidierung usw.)
 - Commit-Anteil bei TAs, die einen Satz ändern, etwa 1/3 der TA-Zeit
- Schnelleres Commit-Protokoll kann Durchsatz verbessern
 - Reduktion der Commit-Dauer jeder TA
 - frühere Freigabe von Sperren, was die Wartezeit anderer TA verkürzt
 - Verkleinerung des Zeitfensters für die Blockierungsgefahr (Crash oder Nicht-Erreichbarkeit von C)

5-29

Commit: Optimierungen (2)

■ Wünschenswerte Charakteristika eines Commit-Protokolls

1. **A von ACID**: stets garantierte TA-Atomarität¹
2. Fähigkeit, den Ausgang der Commit-Verarbeitung einer TA „nach einer Weile“ **vergessen** zu können
 - begrenzte Log-Kapazität von C
 - C muss sicher sein, dass alle Agenten den Commit-Ausgang für T_i kennen oder
 - C nimmt einen bestimmten Ausgang an, wenn er keine Information über T_i mehr besitzt
3. **Minimaler Aufwand** für Nachrichten und synchrones Logging
4. Optimierte **Leistungsfähigkeit im fehlerfreien Fall** (no-failure case)
 - ist wesentlich häufiger zu erwarten: optimistisches Verhalten
 - vor allem Minimierung von synchronen Log-Ausgaben (forced log write)
 - dafür zusätzlicher Recovery-Aufwand, um verlorengegangene Information im Fehlerfall wieder zu beschaffen
5. Spezielle **Optimierung für Leser**
(ihre Commit-Behandlung ist unabhängig vom Ausgang der TA)

1. May all your transactions commit and never leave you in doubt! (Jim Gray)

5-30

Commit: Kostenbetrachtungen

■ Vollständiges 2PC-Protokoll ($N = \# \text{Teil-TA}$, davon $M = \# \text{Leser}$)

- Nachrichten:
- Log-Ausgaben:
- Antwortzeit:
längste Runde in Phase 1 (kritisch, weil Betriebsmittel blockiert)
+ längste Runde in Phase 2

■ Spezielle Optimierung für Leser

- Teil-TA, die nur Leseoperationen ausgeführt haben, benötigen keine Recovery!
 - C und D von ACID sind nicht betroffen
 - für A und I genügt das Halten der Sperren bis Phase 1
- Lesende Teil-TA brauchen deshalb **nur an Phase 1** teilzunehmen
 - Mitteilung an (Zwischen-)Koordinator: „**read-only vote**“
 - dann Freigabe der Sperren
 - (Zwischen-)Koordinator darf (Teil-)Baum nur freigeben, wenn er ausschließlich *read-only votes* erhält
- Nachrichten:
- Log-Ausgaben:
für $N > M$

1. Leseroptimierung lässt sich mit allen 2PC-Varianten kombinieren, wird jedoch nachfolgend nicht berücksichtigt

5-31

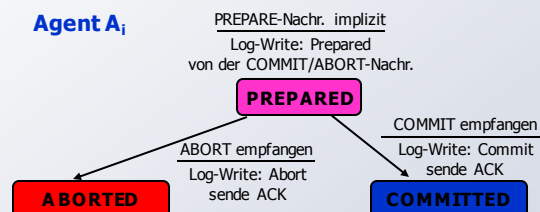
Commit: Kostenbetrachtungen (2)

■ 1PC-Protokolle

- Bei Aufruf der Commit-Operation durch die TA sind bereits alle Agenten im PREPARED-Zustand
 - „implicit yes-vote“ (IYV)
 - Modifikation der API erforderlich

■ Variante 1: A_i geht nach jedem Auftrag in den PREPARED-Zustand

- Jeder Aufruf von A_i : Work & Prepare
- Einschränkungen bei verzögerten Integritätsprüfungen erforderlich (deferred)



- Nachrichten:
- durchschnittlich K Aufträge pro Agent;
Log-Ausgaben:

5-32

Commit: Kostenbetrachtungen (3)

■ Variante 2: A_i geht beim letzten Auftrag in den PREPARED-Zustand

- Normaler Aufruf von A_i : Work
- Letzter Aufruf von A_i : Work&Prepare;
Lässt sich diese Art der Optimierung **immer** erreichen?
- Nachrichten:
- Log-Ausgaben:

■ Weglassen der expliziten ACK-Nachricht

- A_i fragt ggf. nach, falls ihn die Koordinatorenentscheidung nicht erreicht
- C weiß nicht, ob alle A_i die Commit/Abort-Nachricht erhalten haben

⇒ **impliziert "unendlich langes Gedächtnis" von C**

- Nachrichten:
- Log-Ausgaben:

5-33

Commit: Kostenbetrachtungen (4)

■ Spartanisches Protokoll

- A_i geht nach jedem Auftrag in den PREPARED-Zustand;
Weglassen der expliziten ACK-Nachricht
- Nachrichten:
- Log-Ausgaben:
- Nur letzter Aufruf: Work&Prepare;
Log-Ausgaben:

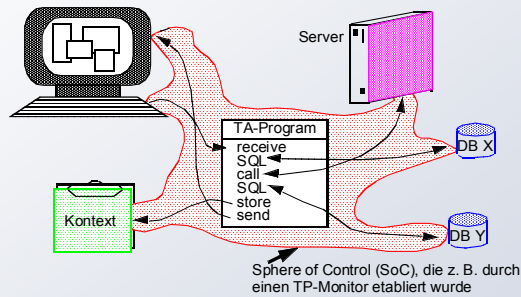
⇒ **Log-Aufwand bleibt gleich (oder erhöht sich drastisch) !**

5-34

TA-Verarbeitung in offenen Systemen

Standards zur TA-Verwaltung

TA-Mgrs benötigen Standards, weil sie letztendlich den „Klebstoff“ verkörpern, der die unterschiedlichen und heterogenen SW-Komponenten zusammenfügt.



- Ihre AW laufen auf verschiedenartigen Plattformen und
 - haben Zugriff zu verschiedenen DBs und RMs
 - Die AW haben absolut kein Wissen voneinander
- ⇒ Einziger Weg zur Verknüpfung: **„Offene Standards“**
- Bereitstellung von Schnittstellen zwischen TA-Mgr und RM, TA-Mgrs untereinander und TA-Mgr und AW

5-35

TA-Verarbeitung in offenen Systemen (2)

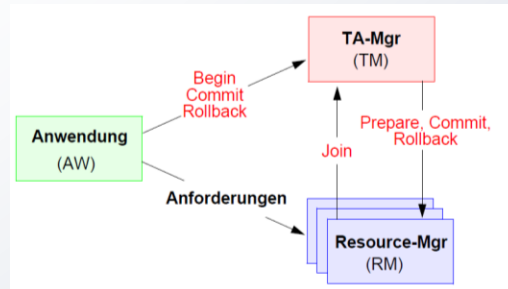
Standards kommen aus zwei Quellen

- **ISO**:
OSI-TP spezifiziert die Nachrichtenprotokolle zur Kooperation von TA-Mgrs
- **X/OPEN** definiert den allgemeinen Rahmen für TA-Verarbeitung:
X/OPEN DTP (distributed transaction processing) stellt eine SW-Architektur dar, die mehreren AWP erlaubt, gemeinsam Betriebsmittel mehrerer RMs zu nutzen und die Arbeit der beteiligten RMs durch globale Transaktionen zusammenzufassen.

5-36

TA-Verarbeitung in offenen Systemen (3)

- **X/Open DTP (1991)** für die lokale Zusammenarbeit von Anwendung (AW), TA-Mgr (TM) und Resource-Mgrs (RMs)



- **Standardisierung**

- Schnittstelle zur Transaktionsverwaltung (TX: AW — TM)
- Schnittstelle zwischen TM und RMs (XA: zur TA-Verwaltung und ACID-Kontrolle)
- zusätzlich: Anforderungsschnittstellen (API, z. B. SQL)

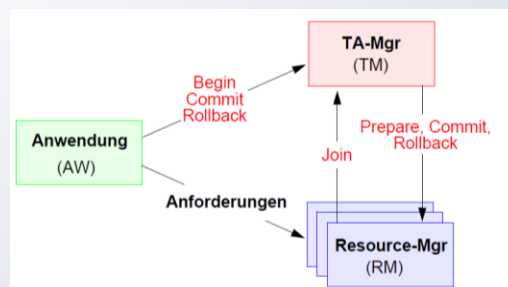
5-37

TA-Verarbeitung in offenen Systemen (4)

- **TA-Ablauf**

- Die AW startet eine TA, die vom lokalen TA-Mgr verwaltet wird
- RMs melden sich bei erster Dienst-Anforderung beim lokalen TA-Mgr an (Join)
- Wenn AW Commit oder Rollback ausführt (oder zwangsweise zurückgesetzt wird), schickt TA-Mgr Ergebnis zu den RMs (über Broadcast)

➔ **hier sorgen die RMs für Synchronisation und Logging** in ihrem Bereich (private Lock-Mgr und Log-Mgr) — Warum?



5-38

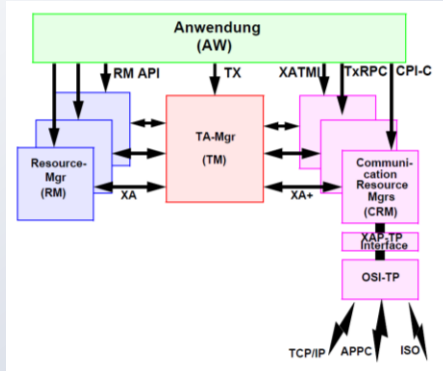
TA-Verarbeitung in offenen Systemen (5)

■ X/Open DTP (1993) standardisiert die verteilte TA-Verarbeitung.

- Es kommt der „Communication Resource Manager“ (CRM) hinzu
- XA + erweitert Schnittstelle XA für den verteilten Fall

■ Definition von Schnittstellen auf der AW-Ebene

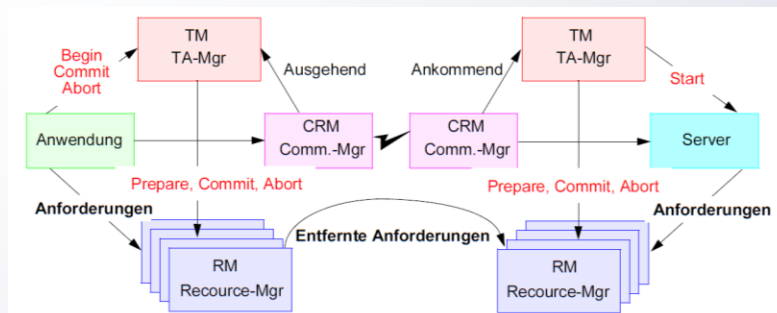
- TxRPC definiert transaktionalen RPC. Seine TA-Eigenschaften können ausgeschaltet werden (optional)
- CPI-C V2 ist Konversationschnittstelle (peer-to-peer), welche die OSI-TP-Semantik unterstützen soll
- XATMI ist Konversationschnittstelle für Client/Server-Beziehungen



- Sind drei Schnittstellen-Standards erforderlich?
Kompromisslösung für drei existierende Protokolle (DCE, CICS, Tuxedo)

5-39

TA-Verarbeitung in offenen Systemen (6)



■ TA-Ablauf

- AW startet TA, die vom lokalen TA-Mgr verwaltet wird
- Wenn die AW oder der RM, der für die AW eine Anforderung bearbeitet, eine entfernte Anforderung durchführen, informieren die CRMs an jedem Knoten ihre lokalen TA-Mgr über die ankommende oder ausgehende TA
- TA-Mgr verwalten an jedem Knoten jeweils die TA-Arbeit am betreffenden Knoten
- Wenn die AW COMMIT oder ROLLBACK durchführt oder scheitert, kooperieren alle beteiligten TA-Mgr, um ein atomares und dauerhaftes Commit zu erzielen.

5-40

BASE – Eine ACID-Alternative?

■ 2PC ist teuer: Wie erreicht man Skalierbarkeit?

- Vertikales Skalieren:
Verschiebe die Anwendung auf das nächst größere Rechensystem
→ aufwändig, teuer, verstärkte Herstellerbindung, ...
- Horizontales Skalieren:
Partitioniere (gruppier) die Daten nach AW-Funktionen und verteile die funktionalen Gruppen auf verschiedene DBs
- **Sharding***:
Weitere Dimension beim horizontalen Skalieren, d.h. horizontale Zerlegung der Daten (Tabellen) innerhalb von funktionalen Gruppen und Zuordnung auf verschiedene DBs
→ Horizontales Skalieren bietet mehr Flexibilität, ist aber auch wesentlich komplexer!

* Nach Wikipedia: Horizontal partitioning is a database design principle whereby rows of a database table are held separately, rather than splitting by columns (as for normalization). Each partition forms part of a shard, which may in turn be located on a separate database server or physical location.

5-41

BASE – Eine ACID-Alternative? (2)

■ Funktionale Partitionierung – wesentlich für hohe Skalierbarkeit

- Beispiel-Schema

transaction(xid, seller_id, buyer_id, amount)

user(id, name, amt_sold, amt_bought)

- Um Constraints (automatisch) einhalten zu können, müssen alle betroffenen Daten (Tabellen) von **einem einzigen** DBMS verwaltet werden
→ horizontales Skalieren wäre hier ausgeschlossen !
- Verwaltung durch verschiedene DBMSs verlangt die Verschiebung von partitionsübergreifenden Constraints in die AW

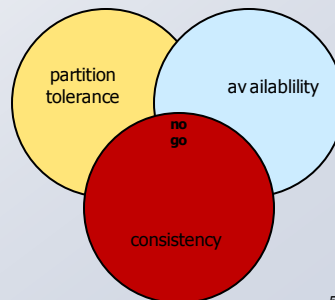
5-42

CAP-Theorem oder Brewer's Theorem

■ CAP-Theorem

- Für ein System zum **verteilten Rechnen** ist es unmöglich, gleichzeitig die folgenden drei Eigenschaften zu garantieren:
 - **Consistency** (Konsistenz):
Aus Benutzersicht findet eine Menge von Operationen auf einmal statt
 - **Availability** (Verfügbarkeit):
Jede Operation muss mit einer bestimmungsgemäßen Antwort beendet werden
 - **Partition tolerance** (Partitionstoleranz):
Operationen kommen zum Ende, selbst wenn individuelle Komponenten nicht verfügbar sind

→ Eine Web-AW kann höchstens zwei von diesen Eigenschaften für jeden beliebigen DB-Entwurf unterstützen!



5-43

CAP-Theorem oder Brewer's Theorem (2)

■ Wie lassen sich ACID-Lösungen annähern?

- Eine horizontale Skalierungsstrategie basiert auf Datenpartitionierung

→ Wir müssen uns zwischen Konsistenz und Verfügbarkeit entscheiden!

■ Aber Konsistenz ist zu garantieren!

→ Wenn Brewer recht hat, muss an der Verfügbarkeit „gedreht“ werden!

■ Auswirkungen reduzierter Verfügbarkeit bei ACID-Lösungen

- angenommene Komponentenv erfügbarkeit von 99,9%
- Gesamtverfügbarkeit als Produkt der Einzelverfügbarkeiten:
2PC-Protokoll bei zwei DBs → 99,8%
(zusätzliche Ausfallzeit von 43 Min./Monat)

→ Reicht das für erhöhte Skalierbarkeit aus?

5-44

BASE (basically available, soft state, eventually consistent)*

■ Eine ACID-Alternative

- ACID ist pessimistisch und erzwingt Konsistenz bei Commit
- BASE ist dazu diametral entgegengesetzt
 - Es ist optimistisch und akzeptiert, dass sich die DB-Konsistenz in einem fließenden Zustand befindet
 - Verfügbarkeit wird dadurch erreicht, dass bestimmte Fehlerzustände akzeptiert werden, ohne dass es zu einem Ausfall des Gesamtsystems kommt
Bsp.: Wenn nur einer von fünf Servern ausfällt, wird im Vergleich zu einem System-Crash eine deutlich höhere Verfügbarkeit wahrgenommen
 - Beim DB-Entwurf sind die **Daten in funktionale Gruppen** zu zerlegen; nach Verkehrsaufkommen werden sie dann auf verschiedene DBMS verteilt

➔ **BASE verlangt eine weitaus detailliertere Analyse der TA-Ops als ACID!**

* In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability (Dan Britchett, eBay)

5-45

BASE (2)

■ Schema

transaction(xid, seller_id, buyer_id, amount)

user(id, name, amt_sold, amt_bought)

■ Konsistenzmuster

- Nach Brewer müssen Wege zur Lockerung der Konsistenzforderung gefunden werden, wenn die Verfügbarkeit in einer partitionierten DB erhalten bleiben soll
- Ein Benutzer kann in mehreren TAs kaufen oder verkaufen
- Müssen die laufenden Gesamtsummen in Tabelle *user* immer aktuell sein?
- Die Tabellen *user* und *transaction* werden hier als verschiedene funktionale Gruppen klassifiziert
- Im Allgemeinen ist die Konsistenz **zwischen** funktionalen Gruppen leichter zu lockern als innerhalb einer funktionalen Gruppe

5-46

BASE (3)

■ Schema

transaction(xid, seller_id, buyer_id, **amount**)

user(id, name, **amt_sold**, **amt_bought**)

■ DB-Ausprägung verteilt

transaction

xid	seller_id	buyer_id	amount
123	4711	0815	20,000
124	0007	4711	10,000

node 2

user

id	name	amt_sold	amt_bought
4711	Coy	200,000	50,000
0815	May	520,000	100,000
0007	Bond	30,000	600,000

node 1

5-47

BASE (4)

■ Eine ACID-Lösung mit SQL

Begin transaction

Insert into transaction(xid, seller_id, buyer_id, **amount**);

Update user set **amt_sold** = amt_sold + \$amount where id = \$seller_id;

Update user set **amt_bought** = amount_bought + \$amount

where id = \$buyer_id;

End transaction

- **2PC** erforderlich
 - ↳ **The Unavailability Problem!** (W. Vogels, Amazon)
- Die Zähler **amount**, **amt_bought** und **amt_sold**, die zu verschiedenen funktionalen Gruppen gehören, werden gemeinsam aktualisiert
 - ↳ **Konsistenz garantiert!**

5-48

BASE (5)

■ Zerlegung in zwei ACID-Transaktionen

Begin transaction

Insert into transaction(id, seller_id, buyer_id, **amount**);

End transaction

Begin transaction

Update user set **amt_sold**=amt_sold+*\$amount*

where id=*\$seller_id*;

Update user set **amt_bought**=amt_bought+*\$amount*

where id=*\$buyer_id*;

End transaction

- Lockerung der Konsistenzanforderung: Die Zähler brauchen nicht sofort das Ergebnis einer Transaktion reflektieren; sie enthalten „Schätzungen“!
- **Konsistenz** zwischen beiden Tabellen wird nicht garantiert!

BASE (6)

■ Ein möglicher Ablauf von xid = 125

transaction

xid	seller_id	buyer_id	amount
123	4711	0815	20,000
124	0007	4711	10,000
125	0815	0007	100,000

node 2

user

id	name	amt_sold	amt_bought
4711	Coy	200,000	50,000
0815	May	520,000	100,000
0007	Bond	30,000	600,000

node 1

BASE (7)

■ Eine einfache BASE-Lösung

- Falls Zähler-Werte als Schätzungen nicht akzeptabel sind und die Konsistenz zumindest zeitverzögert garantiert werden muss?
- Persistente Nachrichtenwarteschlange (PMQ : persistent message queue) löst das Problem

Begin transaction

```
Insert into transaction(id, seller_id, buyer_id, amount);
Queue message "update user(seller_id, "seller", amount)";
Queue message "update user(buyer_id, "buyer", amount)";
```

End transaction

For each message in queue

Begin transaction

Dequeue message

If message.balance == "seller"

```
Update user set amt_sold = amt_sold + message.amount
where id=message.id;
```

Else

```
Update user set amt_bought = amt_bought + message.amount
where id=message.id;
```

End if

End transaction

End for

5-51

BASE (8)

■ Transaktionsablauf transaction

node 2

xid	seller_id	buyer_id	amount
123	4711	0815	20,000
124	0007	4711	10,000
125	0815	0007	100,000

- Erste Transaktion aktualisiert **transaction** und fügt atomar in PMQ ein:
 - update user("seller", 0815, 100,000)
 - update user("buyer", 0007, 100,000)
- PMQ garantiert, dass die Nachrichten in endlicher Zeit atomar verarbeitet werden und die Zähler aktualisieren
 - **eventually consistent!**
- 2PC bei der PMQ-Verarbeitung erforderlich!

5-52

BASE (9)

■ 2PC soll vermieden werden!

- Gewährleistung von **Idempotenz** (bei partiellem Ausfall) !
Welche Zähleraktualisierungen waren erfolgreich und welche nicht?
- Hilfstabelle: **updates_applied**(trans_id, balance, user_id)
- Algorithmus unterstützt partielle Fehler und bietet TA -Garantien (eventually consistent), ohne 2PC auf user und PMQ zu erfordern!

Begin transaction

```
Insert into transaction(id, seller_id, buyer_id, amount);
Queue message "update user(seller_id, "seller", amount)";
Queue message "update user(buyer_id, "buyer", amount)";
```

End transaction

...

5-53

BASE (10)

For each message in **queue**

Peek message

Begin transaction

Select count(*) as processed from **updates_applied**

where trans_id = message.trans_id

and balance = message.balance and user_id = message.user_id

If processed == 0

If message.balance == "seller"

Update user set **amt_sold**=amt_sold + message.amount

where id=message.user_id;

Else

Update user set **amt_bought**=amt_bought + message.amount

where id=message.user_id;

End if

Insert into **updates_applied** (message.trans_id, message.balance, message.user_id);

End if

End transaction

If transaction successful

Remove message from **queue**

End if

End for

- Lösung hängt hier davon ab, dass Nachricht in PMQ aufgesucht (peek) und nach erfolgreicher Verarbeitung aus PMQ entfernt werden kann!
- Queue-Operationen sind vor dem Commit der DB-Transaktion nicht erfolgreich

5-54

BASE (11)

■ Ein möglicher Ablauf von xid = 125

transaction

xid	seller_id	buyer_id	amount
123	4711	0815	20,000
124	0007	4711	10,000
125	0815	0007	100,000

node 2

■ PMQ mit (user_id, balance, amount)

- message.trans_id: update user(0815, "seller", **100,000**)
- message.trans_id: update user(0007, "buyer", **100,000**)
- ...

5-55

BASE (12)

■ Stand von Tabelle user

user

id	name	amt_sold	amt_bought
4711	Coy	200,000	50,000
0815	May	520,000	100,000
0007	Bond	30,000	600,000

node 1

■ Kontrolle erfolgreicher Aktualisierungen von user durch

updates_applied

trans_id	balance	user_id

node 1

5-56

Zusammenfassung

■ Transaktionsparadigma

- Verarbeitungsklammer für die Einhaltung von semantischen Integritätsbedingungen
- Verdeckung der Nebenläufigkeit (*concurrency isolation*)
⇒ Synchronisation
- Verdeckung von (erwarteten) Fehlerfällen (*failure isolation*)
⇒ Logging und Recovery
- im SQL-Standard: COMMIT WORK, ROLLBACK WORK
⇒ Beginn einer Transaktion implizit

■ Zweiphasen-Commit-Protokolle

- Hoher Aufwand an Kommunikation und E/A
- Optimierungsmöglichkeiten sind zu nutzen
- Maßnahmen erforderlich, um Blockierungen zu vermeiden!
⇒ Kritische Stelle: A usfall von C

5-57

Zusammenfassung (2)

■ Einsatz in allen Systemen!

■ Varianten des Commit-Protokolls:

- Hierarchisches 2PC:
Verallgemeinerung auf beliebige Schachtelungstiefe
- Reduzierte Blockierungsgefahr durch 3PC:
Nach Voting-Phase wird in einer zusätzlichen Runde (Dissemination-Phase) allen Agenten der TA-Ausgang mitgeteilt. Erst nachdem C sicher ist, dass alle Agenten das Ergebnis kennen, wird die Decision-Phase initiiert: unabhängige Recovery, aber komplexes Fehlermodell

■ BASE (basically available, soft state, eventually consistent)

- CAP-Theorem: Eine Web-AW kann höchstens zwei der drei Eigenschaften (Konsistenz, Verfügbarkeit, Partitionierungstoleranz) für jeden beliebigen DB-Entwurf unterstützen!
- Lockerung der Konsistenz (zwischen funktionalen Gruppen), wenn die Verfügbarkeit in einer partitionierten DB erhalten bleiben soll

5-58