4. Transaktionsverwaltung

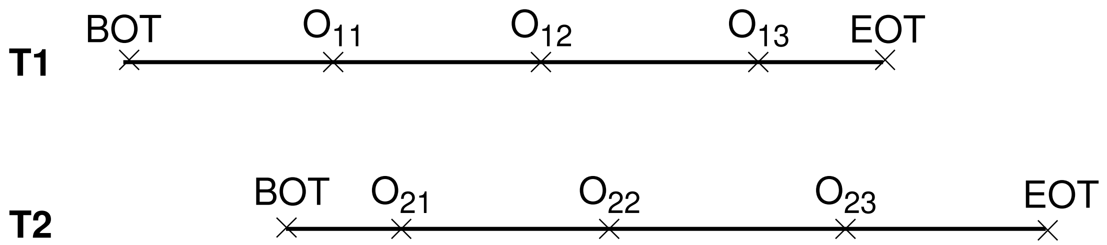
- **Transaktionskonzept**
  - führt ein neues Verarbeitungsparadigma ein
  - ist Voraussetzung für die Abwicklung betrieblicher Anwendungen (*mission-critical applications*)
  - erlaubt „Vertragsrecht" in rechnergestützten IS zu implementieren
- **Gefährdung der DB-Konsistenz**
- **Transaktionsverwaltung**
  - ACID-Eigenschaften
  - Architektur
  - Integration heterogener Komponenten
- **Transaktionsablauf**
  - SQL-Operationen: COMMIT WORK, ROLLBACK WORK
    (Beginn einer Transaktion implizit)
  - Zustände und Zustandsübergänge
- **Wie erreicht man Atomarität?**
  - Einsatz von Commit-Protokollen
    (zentralisierter TA-Ablauf)
  - 2PC (Zweiphasen-Commit-Protokoll)
    - verteilter TA-Ablauf
    - Fehleraspekte
    - Kostenbetrachtungen
  - Hierarchisches 2PC

Gefährdung der DB-Konsistenz

	Korrektheit der Abbildungshierarchie	Übereinstimmung zwischen DB und Miniwelt
durch das Anwendungsprogramm	Mehrbenutzer-Anomalien **Synchronisation**	unzulässige Änderungen **Integritätsüberwachung des DBVS** **TA-orientierte Verarbeitung**
durch das DBVS und die Betriebsumgebung	Fehler auf den Externspeichern, Inkonsistenzen in den Zugriffspfaden **Fehlertolerante Implementierung** **Archivkopien (Backup)**	undefinierter DB-Zustand nach einem Systemausfall **Transaktionsorientierte Fehlerbehandlung (Recovery)**

Transaktionsverwaltung

- **Ablaufkontrollstruktur: Transaktion**



- **Welche Eigenschaften von Transaktionen sind zu garantieren? (zur Erinnerung)**

  - **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 (siehe Atomarität) zurückgesetzt
    - 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

Transaktionsverwaltung (2)

- **DB-bezogene Definition der Transaktion:**
  Eine TA ist eine ununterbrechbare Folge von DML-Befehlen,
  welche die Datenbank von einem logisch konsistenten Zustand in einen
  neuen logisch konsistenten Zustand überführt.

➥ Diese Definition eignet sich insbesondere für relativ kurze TA,
die auch als ACID-Transaktionen bezeichnet werden.

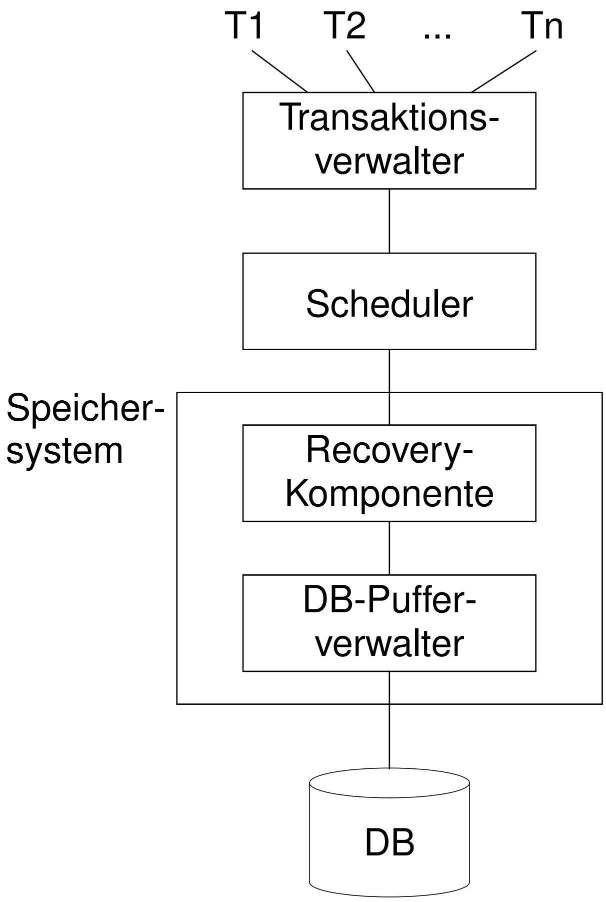
- **Wesentliche Abstraktionen aus Sicht der DB-Anwendung**
  - Alle Auswirkungen auftretender Fehler bleiben der Anwendung verborgen (*failure transparency*)
  - Es sind keine anwendungsseitigen Vorkehrungen zu treffen, um Effekte der Nebenläufigkeit beim DB-Zugriff auszuschließen (*concurrency transparency*)

➥ Gewährleistung einer fehlerfreien Sicht auf die Datenbank
im logischen Einbenutzerbetrieb

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

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 von TA (bei Überlast)
  - Koordination der Abort- und Commit-Behandlung

- **Scheduler** (Synchonisation)
  kontrolliert die Abwicklung der um DB-Daten konkurrierenden TA

- **Recovery-Komponente**
  sorgt für die Rücksetzbarkeit/Wiederholbarkeit der Effekte von TA

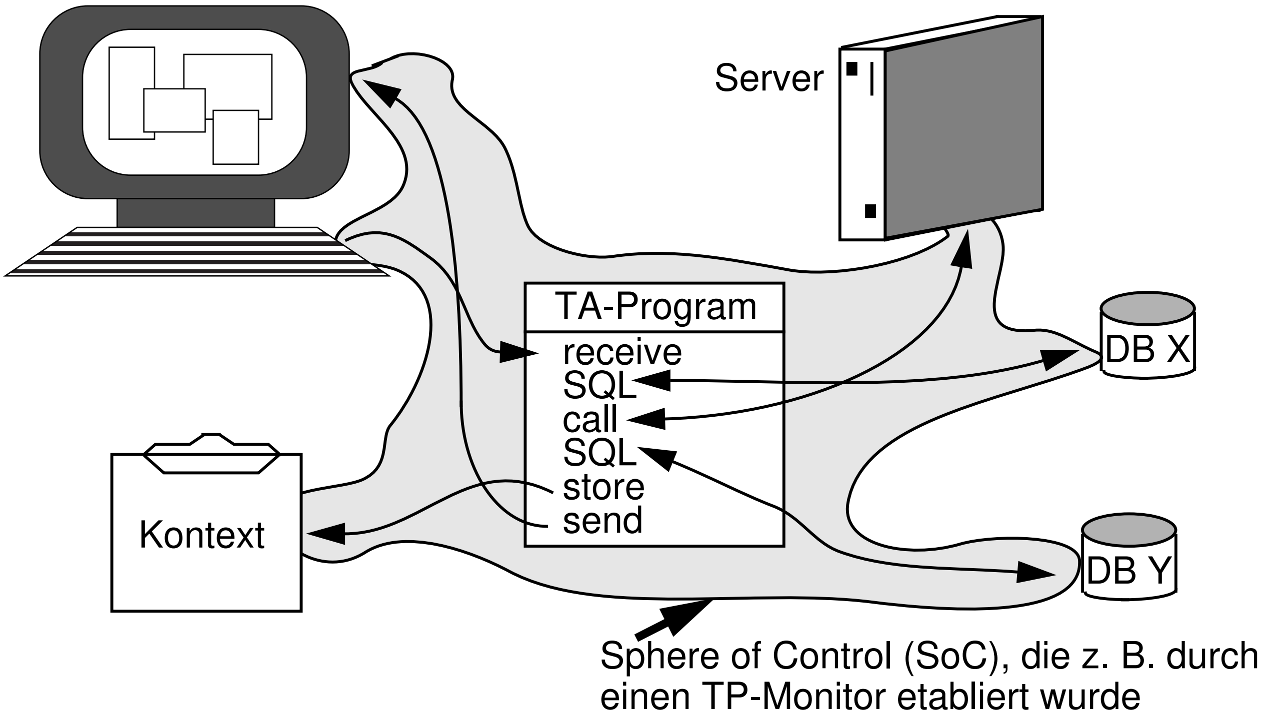
- **DB-Pufferverwalter**
  stellt DB-Seiten bereit und gewährleistet persistente Seitenänderungen

Transaktionsverwaltung (4)

- **Einsatz kooperierender Ressourcen-Manager (RM)**
  - RM sind Systemkomponenten, die **Transaktionsschutz für ihre gemeinsam nutzbaren Betriebsmittel** (BM) bieten
  - RM gestatten die externe Koordination von BM-Aktualisierungen durch spezielle Commit-Protokolle

  ➥ Gewährleistung von ACID für DB-Daten und auch für andere BM (persistente Warteschlangen, Nachrichten, Objekte 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

Mögliche Ausgänge einer Transaktion

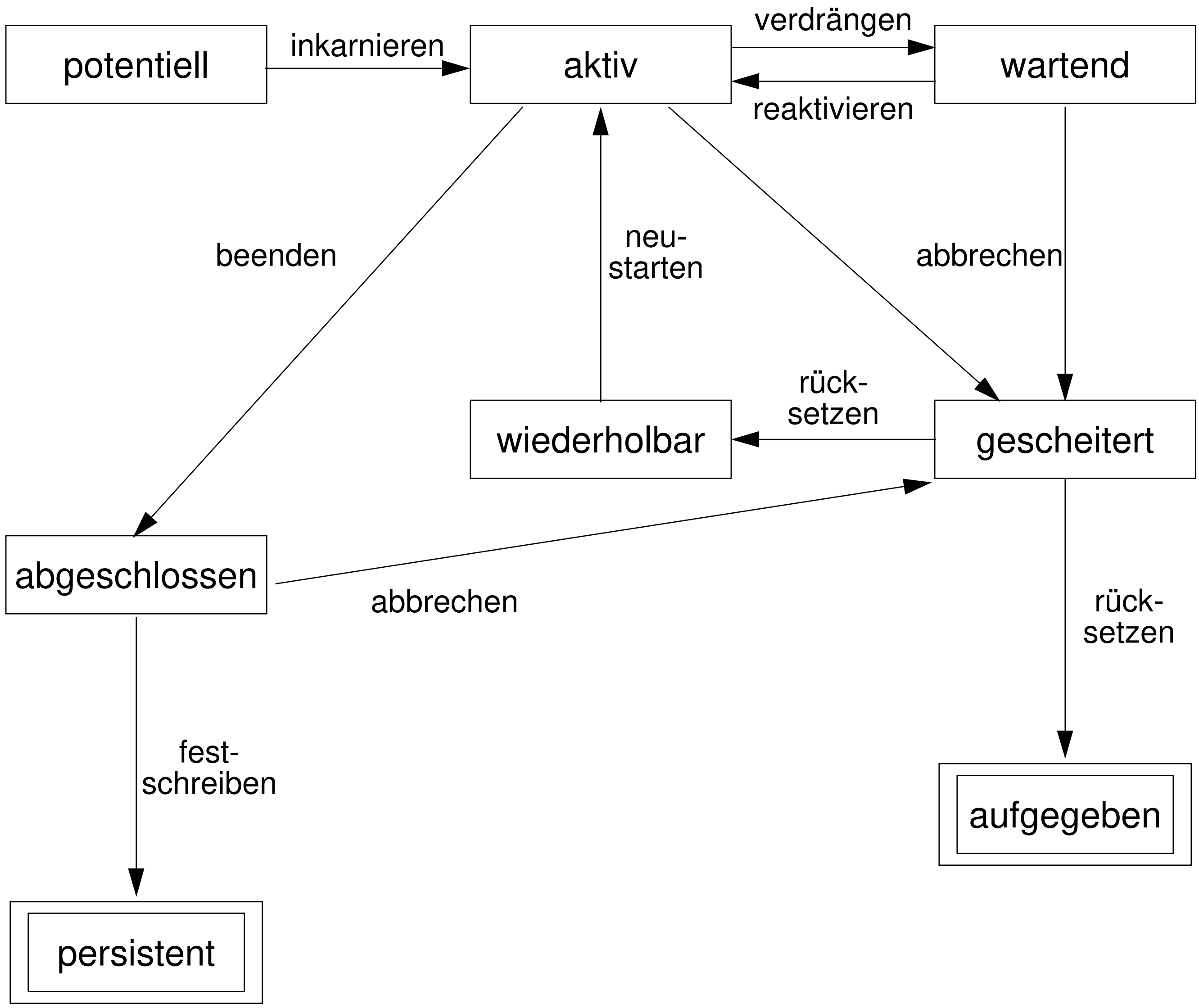
BOT	BOT	BOT
DML1	DML1	DML1
DML2	DML2	DML2
DML3	DML3	DML3
• • •	• • •	Systemausfall, Programm-fehler usw.
DMLn	DMLn	
COMMIT	ROLLBACK	erzwungenes ROLLBACK
normales Ende	abnormales Ende	erzwungenes abnormales Ende

Zustände einer Transaktion

- **Transaktionsprogramm - Beispiel:**

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

- **Zustandsübergangs-Diagramm**

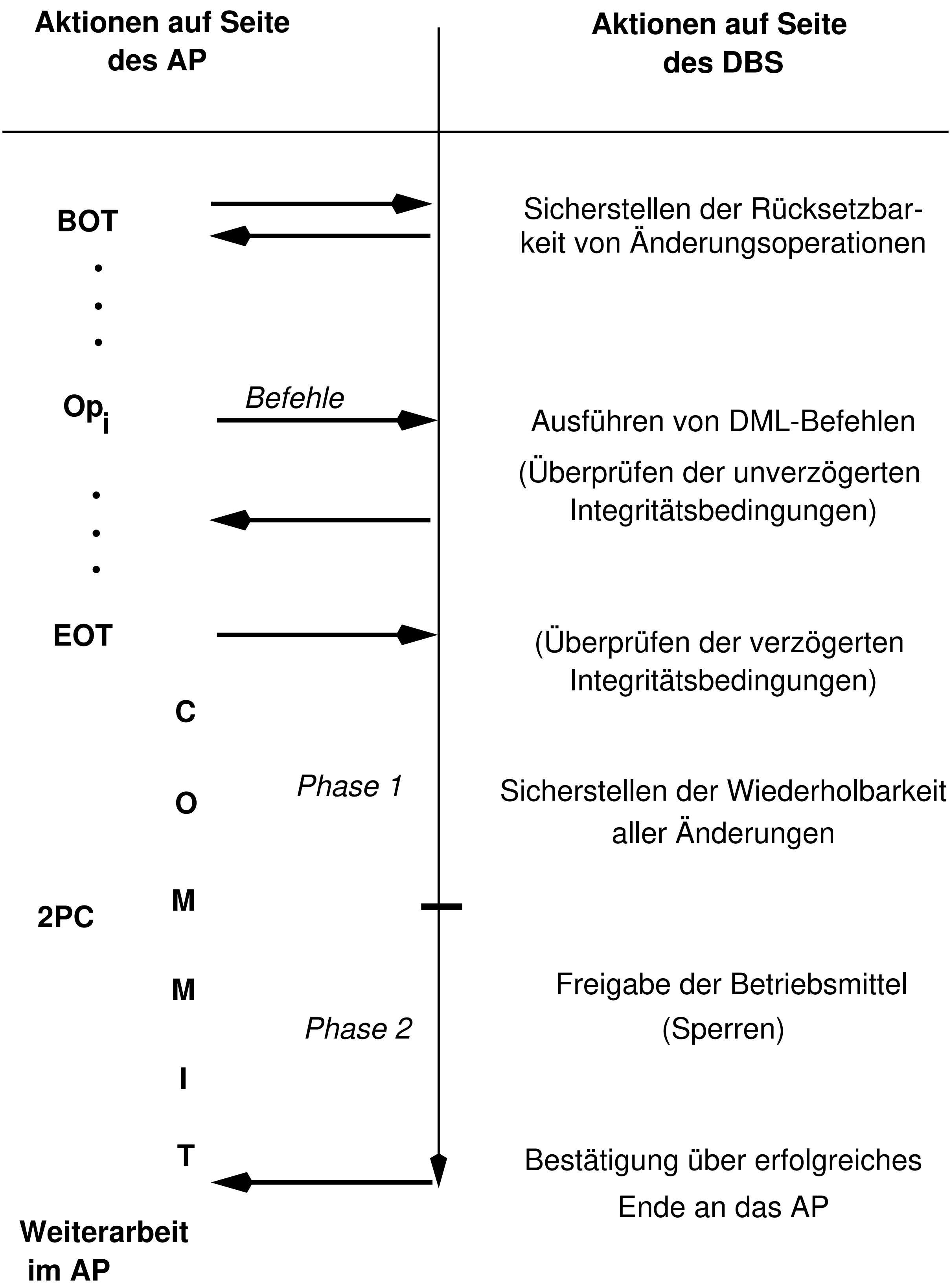


Zustände einer Transaktion (2)

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

- **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)

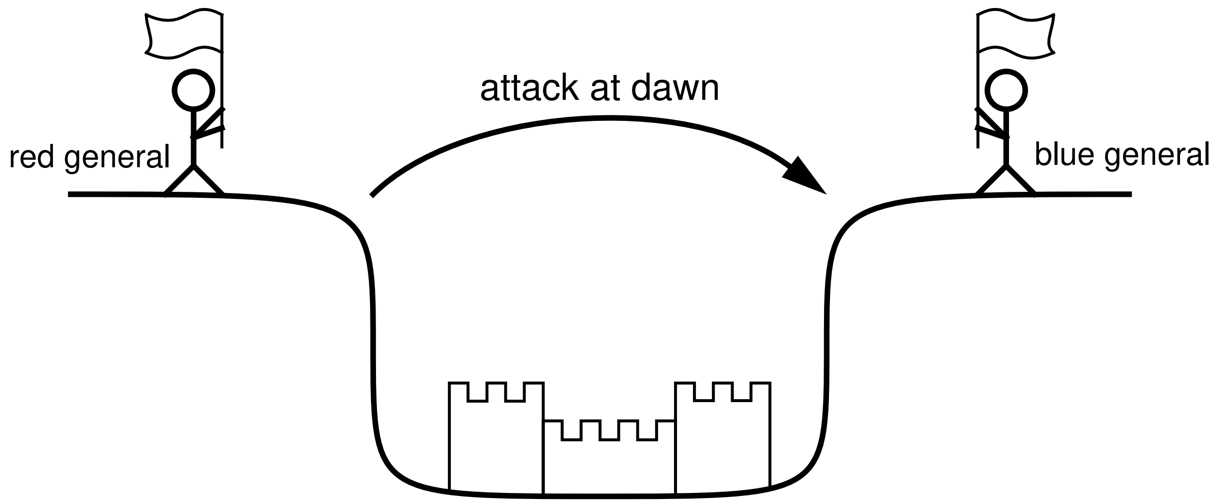
Schnittstelle zwischen AP und DBS - transaktionsbezogene Aspekte



Verarbeitung in Verteilten Systemen

- Ein ***verteiltes System*** besteht aus autonomen Subsystemen, die koordiniert zusammenarbeiten, um eine gemeinsame Aufgabe zu erfüllen
  - Client/Server-Systeme
  - Mehrrechner-DBS, . . .

- **Beispiel: The „Coordinated Attack“ Problem**



Generals-Paradoxon

- **Grundproblem verteilter Systeme**

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

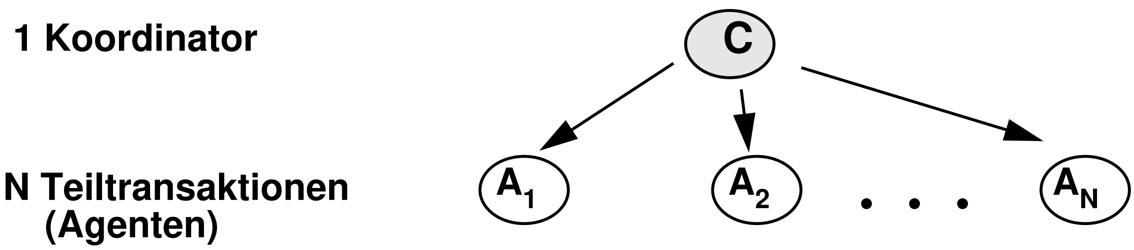
➥ **symmetrische Kontrollalgorithmen sind oft zu teuer oder zu ineffektiv**

➥ **fallweise Zuordnung der Kontrolle**

Verarbeitung in Verteilten Systemen (2)

- **Erweitertes Transaktionsmodell**

  verteilte Transaktionsbearbeitung (Primär-, Teiltransaktionen)



➥ *rechnerübergreifendes Mehrphasen-Commit-Protokoll notwendig, um Atomizität einer globalen Transaktion sicherzustellen*

- **Anforderungen an geeignetes Commit-Protokoll:**
  - Geringer Aufwand (#Nachrichten, #Log-Ausgaben)
  - Minimale Antwortzeitverlängerung (Nutzung von Parallelität)
  - Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern

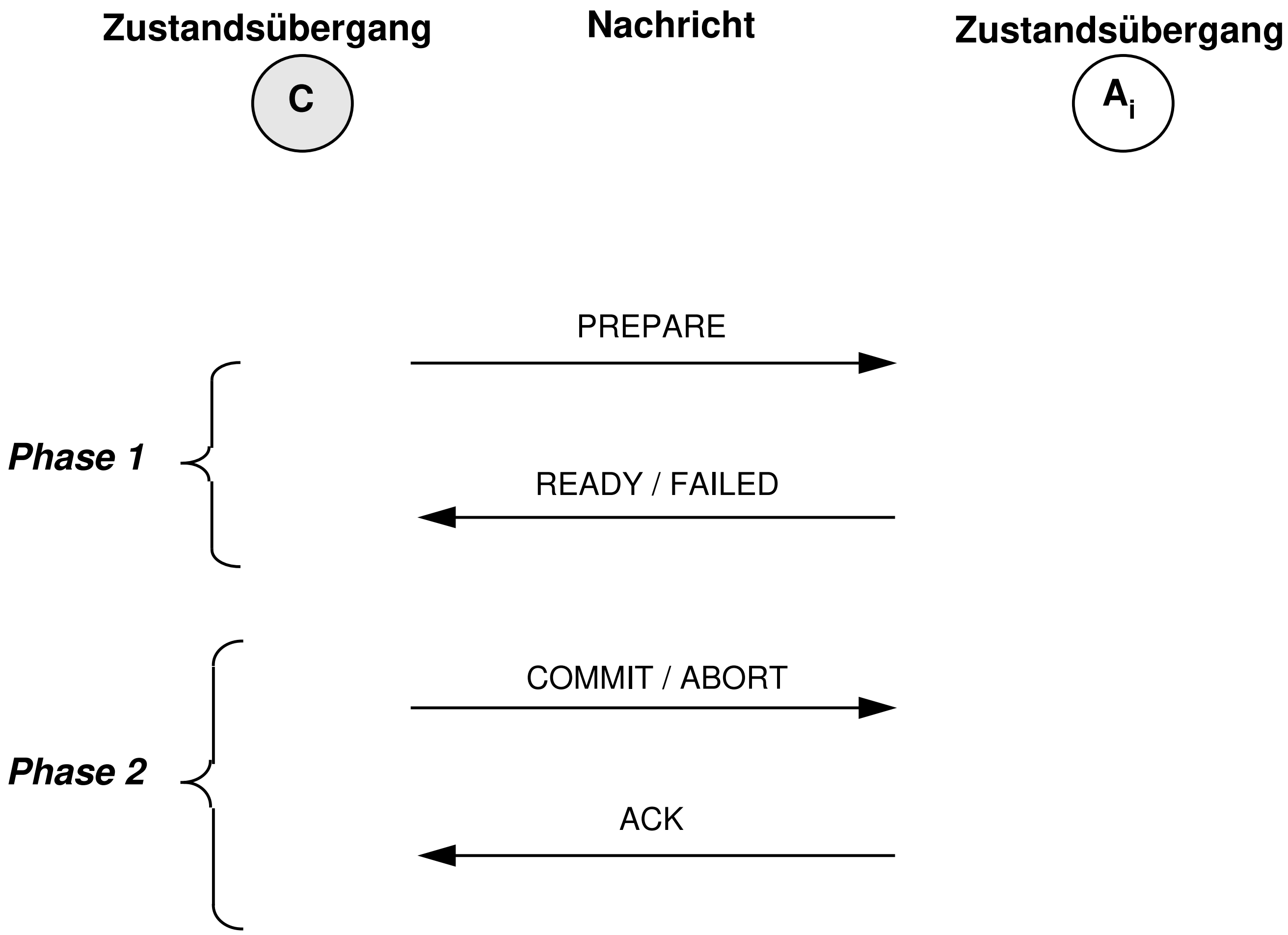
  ➥ Zentralisiertes Zweiphasen-Commit-Protokoll stellt geeignete Lösung dar

- **Erwartete Fehlersituationen**
  - Transaktionsfehler
  - Systemfehler (Crash)

    ➥ i. allg. partielle Fehler (Rechner, Verbindungen, ...)
  - Gerätefehler

➥ Fehlererkennung z. B. über Timeout

Zentralisiertes Zweiphasen-Commit



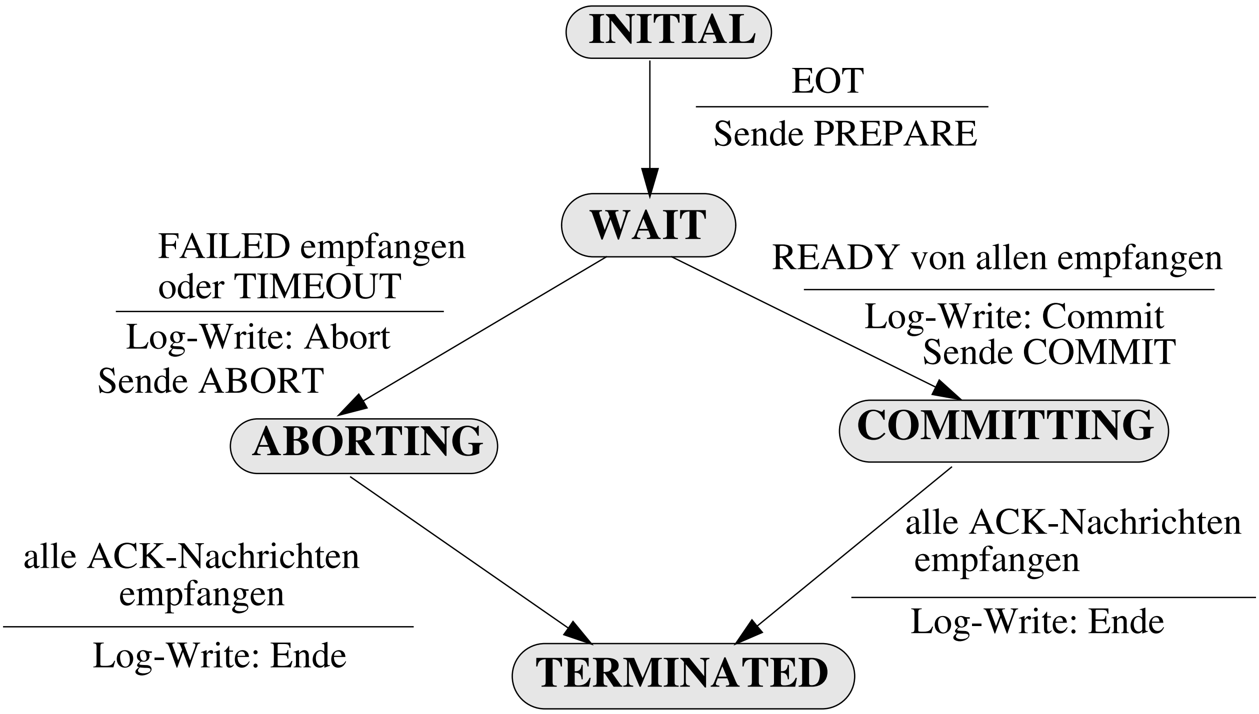
- **Protokoll erfordert Folge von Zustandsübergängen**
  - für Koordinator
  - für jeden Agenten

➥ Zustandsübergänge müssen auf „sicherem Platz“ (Log) vermerkt sein!

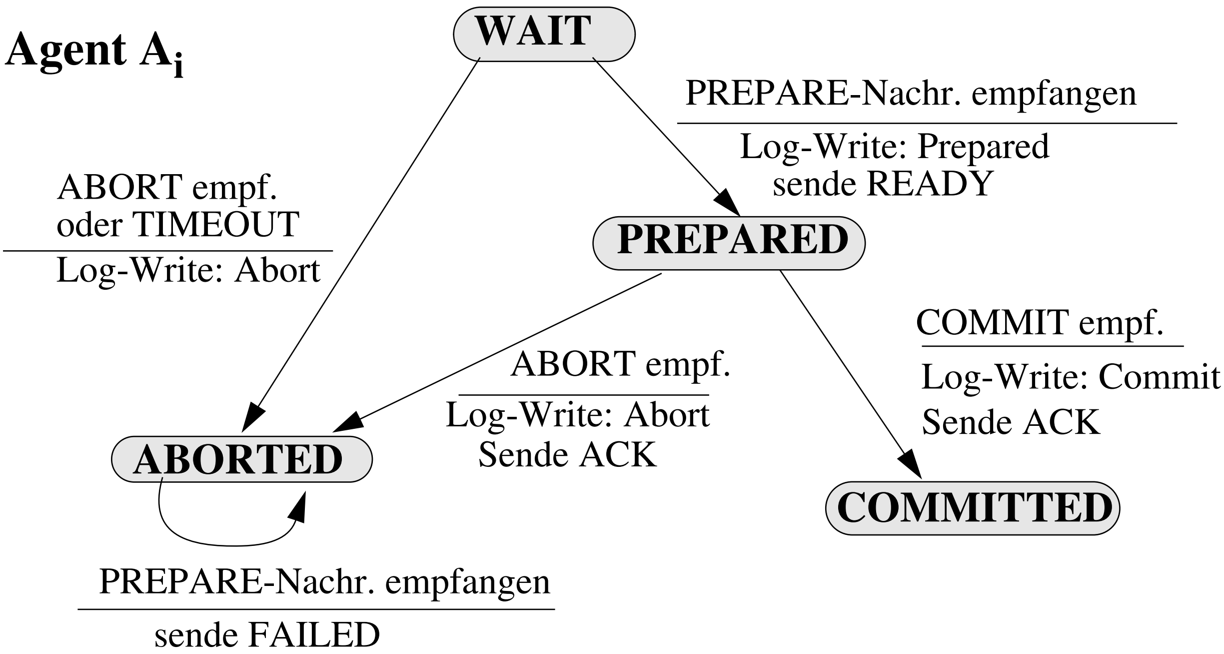
- **Aufwand im Erfolgsfall:**
  - Nachrichten:
  - Log-Ausgaben:

2PC: Zustandsübergänge

Koordinator C



Agent A_i



2PC: Fehlerbehandlung

- **Timeout-Bedingungen für Koordinator:**
  - WAIT ➥ setze Transaktion zurück; verschicke ABORT-Nachr.
  - ABORTING, COMMITTING ➥ vermerke Agenten, für die ACK noch aussteht

- **Timeout-Bedingungen für Agenten:**
  - WAIT ➥ setze Teiltransaktion zurück (unilateral ABORT)
  - PREPARED ➥ erfrage Transaktionsausgang bei Koordinator (bzw. anderen Rechnern)

- **Ausfall des Koordinatorknotens:**
  Vermerkter Zustand auf Log
  - TERMINATED:
    - 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 noch 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)

Commit: Kostenbetrachtungen

- **vollständiges 2PC-Protokoll**
  (N = #Teil-TA, davon M = #Leser)
  - Nachrichten: 4 N
  - Log-Ausgaben: 3 + 2 N
  - Antwortzeit:
    längste Runde in Phase 1 (kritisch, weil Betriebsmittel blockiert)
    + längste Runde in Phase 2

- **Aufwand bei spezieller Optimierung für Leser:**

  Lesende Teil-TA nehmen nur an Phase 1 teil, dann Freigabe der Sperren
  - Nachrichten:
  - Log-Ausgaben:
    für N > M

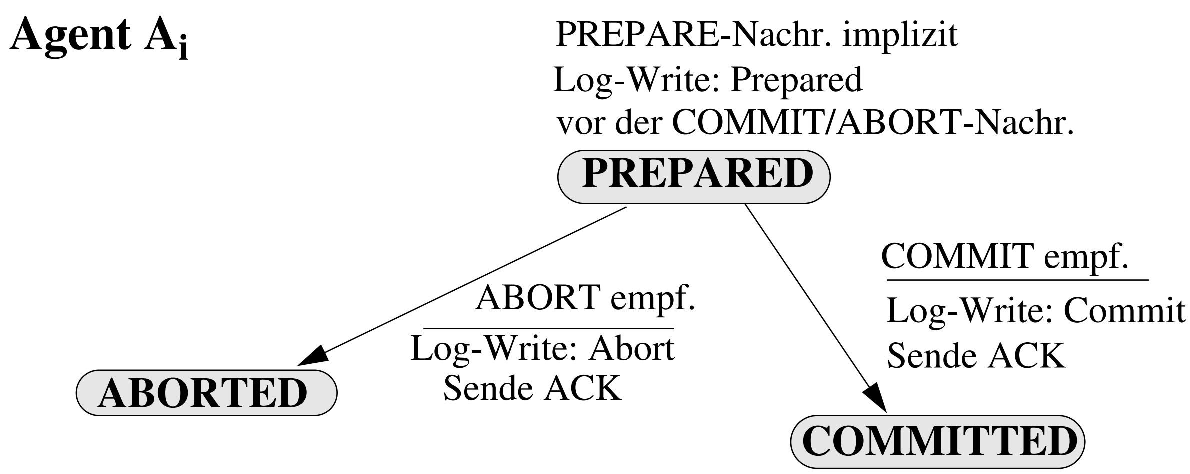
- **Läßt sich das zentralisierte 2PC-Protokoll weiter optimieren?**[1]

1. Leseroptimierung läßt sich mit allen nachfolgenden Varianten kombinieren, wird jedoch nachfolgend nicht berücksichtigt

Commit: Kostenbetrachtungen (2)

- **Weglassen der expliziten Acknowlegde-Nachricht**
  - A_i fragt ggf. nach;
    "unendlich langes Gedächtnis" von C
  - Nachrichten:
  - Log-Ausgaben:

- **A_i geht nach jedem Auftrag in den PREPARED-Zustand**
  - Jeder Aufruf von A_i: Work&Prepare



  - Nachrichten:
  - durchschnittlich K Aufträge pro Agent;

    Log-Ausgaben:

Commit: Kostenbetrachtungen (3)

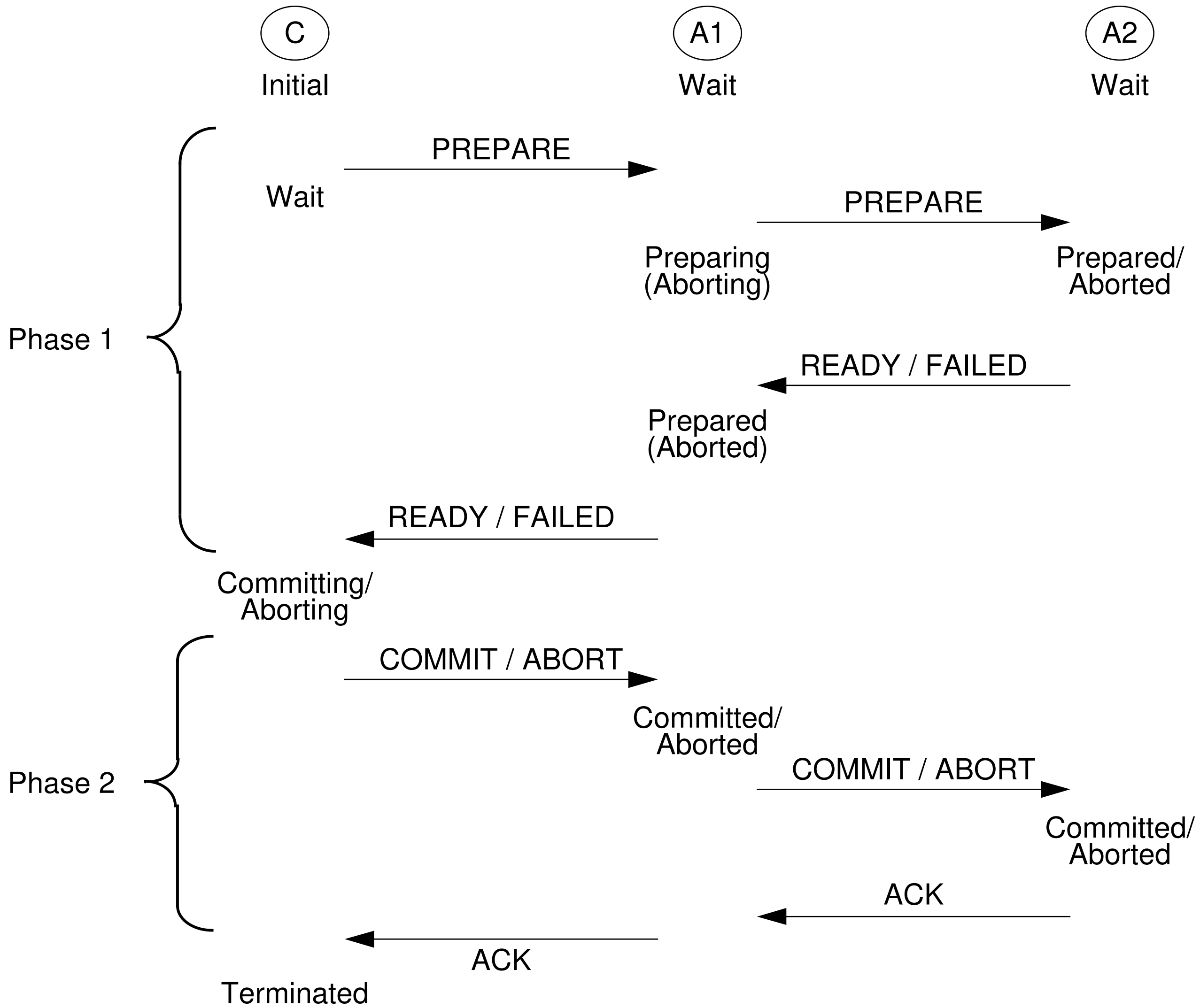
- **A_i geht beim letzten Auftrag in den PREPARED-Zustand**
  - Normaler Aufruf von A_i: Work
  - Letzter Aufruf von A_i: Work&Prepare;
    Läßt sich diese Art der Optimierung **immer** erreichen?
  - Nachrichten:
  - Log-Ausgaben:

- **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) !**

Hierarchisches 2PC

- **Allgemeineres Ausführungsmodell**
  - beliebige Schachtelungstiefe, angepaßt an Client/Server-Modell
  - Modifikation des Protokolls für “Zwischenknoten”



- **Aufwand**
  - Optimierung für lesende Teiltransaktionen
    - kein Logging, Sperrfreigabe in Phase 1
    - Kommunikation für zweite Phase wird vermieden
  - Antwortzeiterhöhung mit steigender Schachtelungstiefe
- **Problem bei 2PC: Koordinatorausfall ➥ Blockierung möglich!**

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:** Ausfall von C

- **Einsatz in allen Systemen!**

- **Varianten des Commit-Protokolls:**
  - Hierarchisches 2PC:
    Verallgemeinerung auf beliebige Schachtelungstiefe
  - 3PC