Datenbanken und SQL

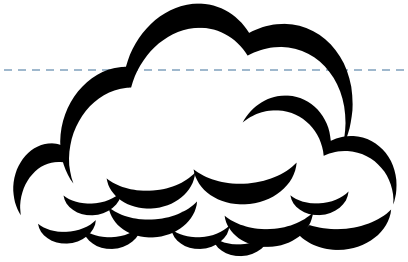
Kapitel 9

Moderne Datenbankkonzepte

Moderne Datenbankkonzepte

- Verteilte Datenbanken
  - Vorteile der verteilten Datenbanken
  - Die 12 Regeln von Date zu verteilten Datenbanken
  - Das CAP-Theorem
  - Vergleich der Systeme CA, CP, PA
  - Anwendungsfälle für CA, CP, PA
  - BASE versus ACID
  - Überblick über moderne verteilte Datenbanken
  - Zwei-Phasen-Commit

Übersicht



- Neue wichtige Begriffe:
  - Cloud-Computing
    - Umfassender Begriff dafür, dass Daten im Netz gehalten werden
    - Der exakte Speicherort ist in der Regel nicht festgelegt und nicht bekannt
    - Beispiele:
      - Drop Box, OneDrive, IMAP (Email) usw.
      - Verteilte Datenbankserver von Amazon, Google, Facebook usw.
  - NoSQL
    - Begriff steht für: Not Only SQL
    - Erweiterung der Sprache SQL für neue Datenbankkonzepte
    - SQL ist sehr weit verbreitet, optimiert für relationale Datenbanken
    - Für nicht relationale Datenbanken sind Erweiterungen erforderlich

Vorteile verteilter Datenbanken

- Schnelle Verfügbarkeit
  - da Daten parallel zugreifbar
  - da Daten eventuell mehrfach gehalten werden
  - da deshalb Daten eventuell lokal direkt verfügbar
- Ausfallsicherheit
  - Wenn ein Knoten ausfällt, sind die anderen noch verfügbar
- Im Extremfall
  - gibt es tausende von Knoten
  - Daten werden redundant gehalten
- Aber: Synchronisierung geänderter Daten ist komplex

Fundamentales Prinzip

- **Fundamentales Prinzip verteilter Datenbanken:**

  Ein verteiltes System sollte sich dem Anwender gegenüber genauso wie ein zentrales verhalten.

- **Dies heißt:**
  - Der Anwender bemerkt bei seinen Zugriffen keinen Unterschied, ob er zentral auf eine Datenbank oder auf viele verteilte Daten zugreift
  - Dies gilt auch für den Anwendungsprogrammierer!
- **Die folgenden 12 Regeln von Date basieren auf diesem Prinzip**

Die 12 Regeln von Daten

- J.F. Date stellte 12 Regeln zu verteilten Datenbanken auf:

1.	Lokale Eigenständigkeit jedes Rechners
2.	Keine zentrale Verwaltungsinstanz
3.	Ständige Verfügbarkeit
4.	Lokale Unabhängigkeit
5.	Unabhängigkeit gegenüber Fragmentierung
6.	Unabhängigkeit gegenüber Datenreplikation
7.	Optimierte verteilte Zugriffe
8.	Verteilte Transaktionsverwaltung
9.	Unabhängigkeit von der Hardware
10.	Unabhängigkeit von Betriebssystemen
11.	Unabhängigkeit vom Netz
12.	Unabhängigkeit von den Datenverwaltungssystemen

Regeln 1 und 2

- Lokale Eigenständigkeit jedes einzelnen Rechners
  - Jeder einzelne Rechner arbeitet möglichst autonom
  - Dies garantiert eine hohe Ausfallsicherheit
  - Dies erfordert einen hohen internen Kommunikationsaufwand
- Keine zentrale Instanz, die das System verwaltet
  - Fast eine Folgerung aus Regel 1: Wenn jede Instanz eigenständig ist, benötigen wir keine zentrale Instanz
  - Damit müssen sich alle einzelnen Rechner gegenseitig verwalten
  - Dies führt zu einem hohen internen Kommunikationsaufwand

Regeln 3, 4 und 5

- Ständige Verfügbarkeit
  - Das gesamte System sollte nie abgeschaltet werden
- Lokale Unabhängigkeit
  - Jeder Zugriff ist unabhängig davon, wo sich die gewünschten Daten derzeit befinden, ob lokal oder entfernt
- Unabhängigkeit gegenüber Fragmentierung
  - Fragmentierung: Relationen werden auf mehrere Rechner verteilt (auch sharding genannt)
  - Der Anwender greift unabhängig von der Fragmentierung zu
  - Die Fragmentierung kann dynamisch sein: Daten werden meist dort gespeichert, wo sie häufig zugegriffen werden (Regionalisierung)

Regeln 6, 7 und 8

- Unabhängigkeit gegenüber Datenreplikation
  - Die Zugriffe ändern sich nicht, falls Replikate existieren
  - Die Verwaltung der Replikate und die Konsistenz der Daten übernimmt das verteilte System
- Optimierung verteilter Zugriffe
  - Das Suchen der Daten im verteilten System und das Lesen und Schreiben werden intern optimiert
- Verteilte Transaktionsverwaltung
  - Transaktionen werden als atomare Einheiten voll unterstützt
  - Recovery und Concurrency werden voll unterstützt

Regeln 9 bis 12

- Unabhängigkeit von der verwendeten Hardware
  - PCs, Großrechner, unterschiedliche Netze werden unterstützt
- Unabhängigkeit von den verwendeten Betriebssystemen
  - Windows, MacOS, Unix, usw.
- Unabhängigkeit vom verwendeten Netzwerk
  - Unterstützung aller wichtigen Protokolle, z.B. TCP/IP
- Unabhängigkeit vom verwendeten DBMS
  - Verwendung gemeinsamer Zugriffssprachen wie JSON, PDO
  - Unterstützung von SQL und NoSQL

Zusammenfassung zu den 12 Regeln

- Regel 9 bis 11 (unabhängig von HW, OS, Netz):
  - Heute weitgehend erfüllt
- Regel 1 bis 6 (eigenständig, verfügbar, fragmentiert, repliziert):
  - Ehrgeizige Ziele, die teilweise schon erfüllt werden
- Regel 7, 8 und 12 (optimiert, Transaktion, DBMS-unabhängig):
  - Nicht widerspruchsfrei
  - Regel 7 und 12 erwarten einheitliche Schnittstellen und einen Transaktionsbetrieb
  - Regel 8 fordert verteilte Transaktionen, mit SQL allein kaum zu erfüllen

Das CAP-Theorem

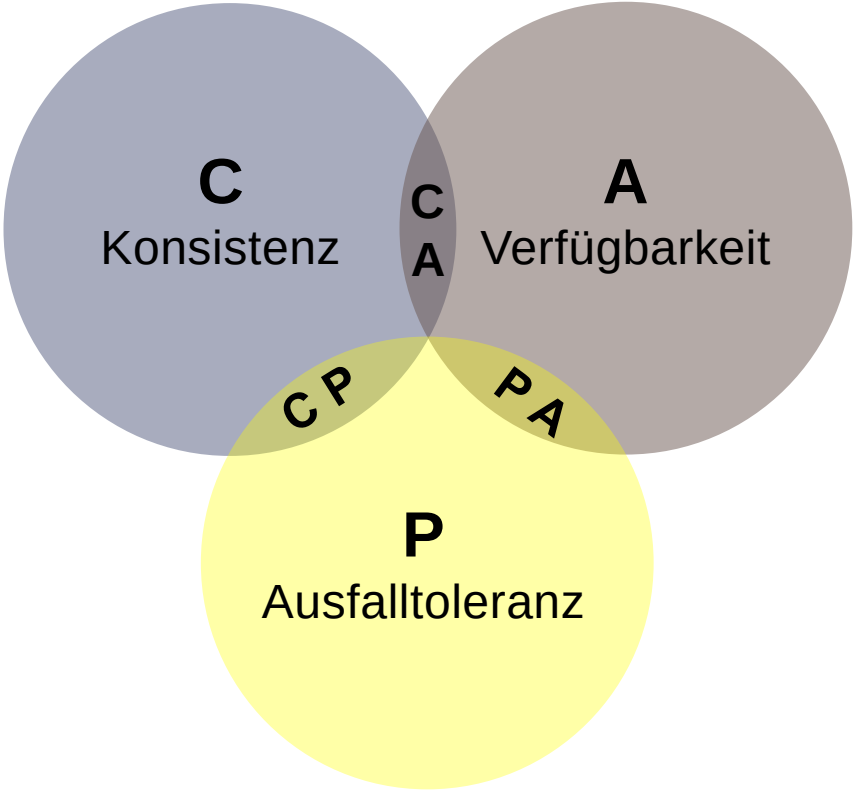
- Konsistenz (**C**onsistence),
  Verfügbarkeit (**A**vailability) und
  Ausfalltoleranz (Tolerance of Network **P**artitions)
  können in verteilten Datenbanken nicht gleichzeitig erfüllt werden.

  - Hier wird sehr gezielt auf das Problem des Widerspruchs der 12 Regeln von Date eingegangen.
  - Nur jeweils 2 der obigen Eigenschaften C, A und P können gleichzeitig vollständig erfüllt werden.

C – A – P

- **C**onsistence (Konsistenz)
  - → Transaktionsbetrieb, ACID, Regel 8
  - Erfordert die Atomarität, Konsistenz bei redundanten Daten
- **A**vailability (Verfügbarkeit)
  - → Regel 3 und 7
  - Ein Teilausfall des Systems sollte nicht zum Gesamtausfall führen
- **P**artition-Toleranz (Ausfalltoleranz)
  - → in mehreren Regeln enthalten
  - Ein Ausfall von Netzkomponenten führt nicht zum Gesamtausfall
  - Ein verspätetes Zustellen ist zu tolerieren

CAP-Eigenschaften



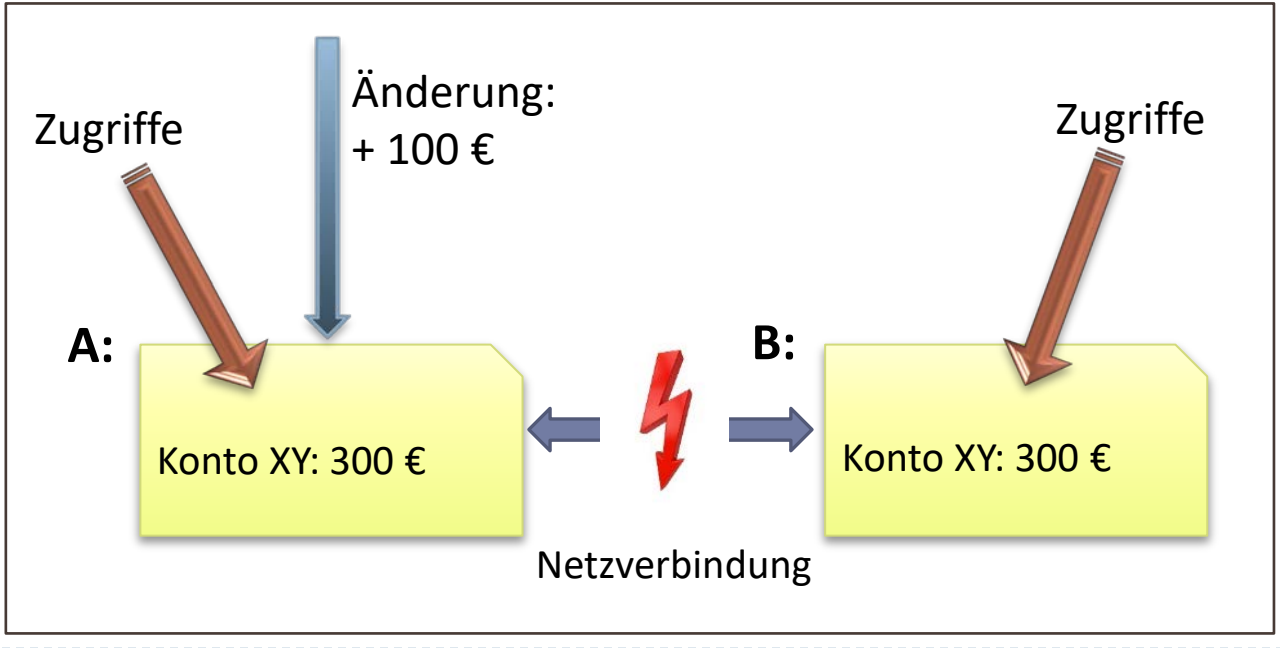
- CA-Systeme:
  - Klassischer Bereich der relationalen Datenbanken
- CP-Systeme:
  - Auf einzelne Knoten kann aus Konsistenzgründen nicht mehr zugegriffen werden.
- PA-Systeme:
  - Hochverfügbares System auf Kosten der sofortigen Konsistenz

Beispiel: Konto XY existiert auf 2 Knoten

- Beispiel:
  - Konto XY sei auf zwei Knoten A und B vorhanden
  - Anwender ist mit Knoten A verbunden
  - Anwender überweist einen Betrag von 100 €
  - Alle anderen Anwender sind entweder mit A oder B verbunden
- Szenario:
  - Die Verbindung zwischen Knoten A und B ist gestört
  - Wie verhält sich jetzt das System?
- Antwort:
  - Abhängig vom System: CA, CP, AP

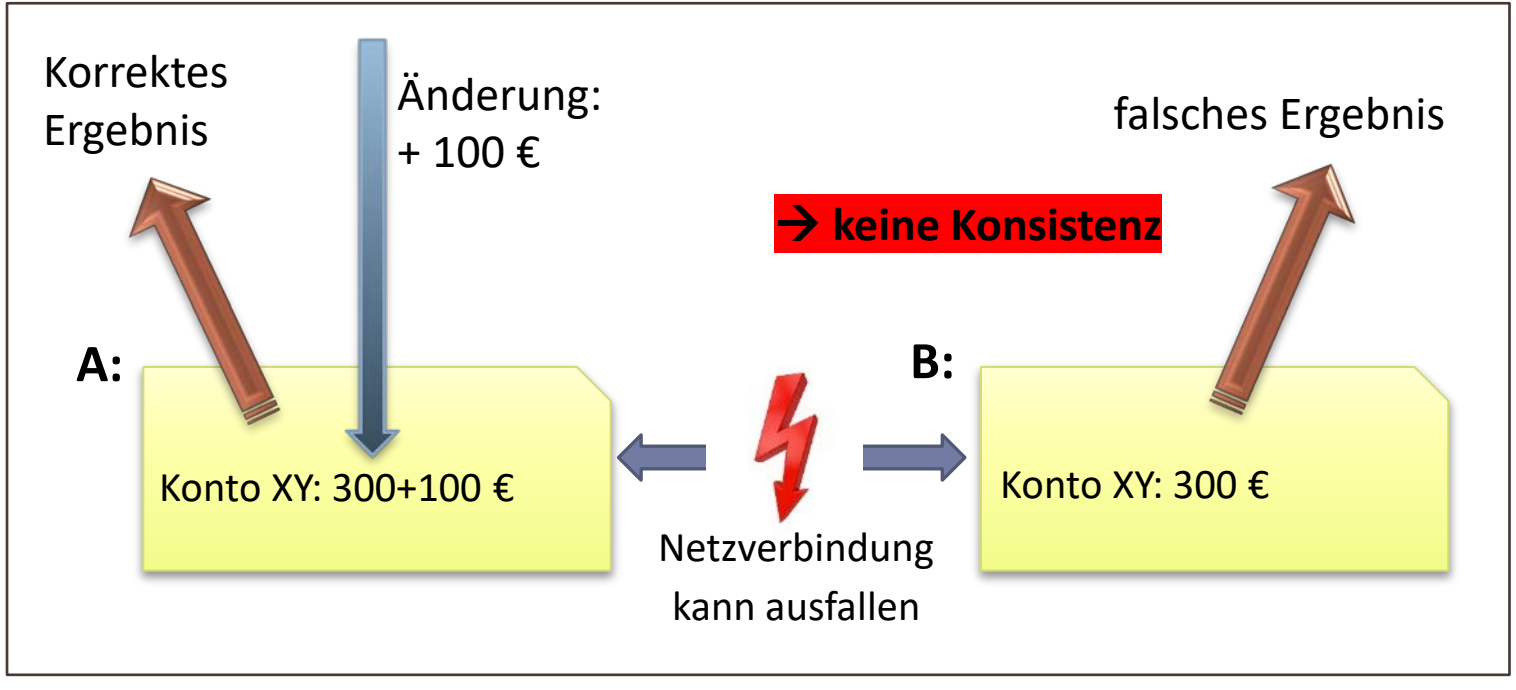
Beispiel: Veranschaulichung am Bild

- Konto XY ist auf den Knoten A und B verfügbar
- A und B sind über das Netz verbunden
- Am Knoten A wird Konto um 100 Euro erhöht



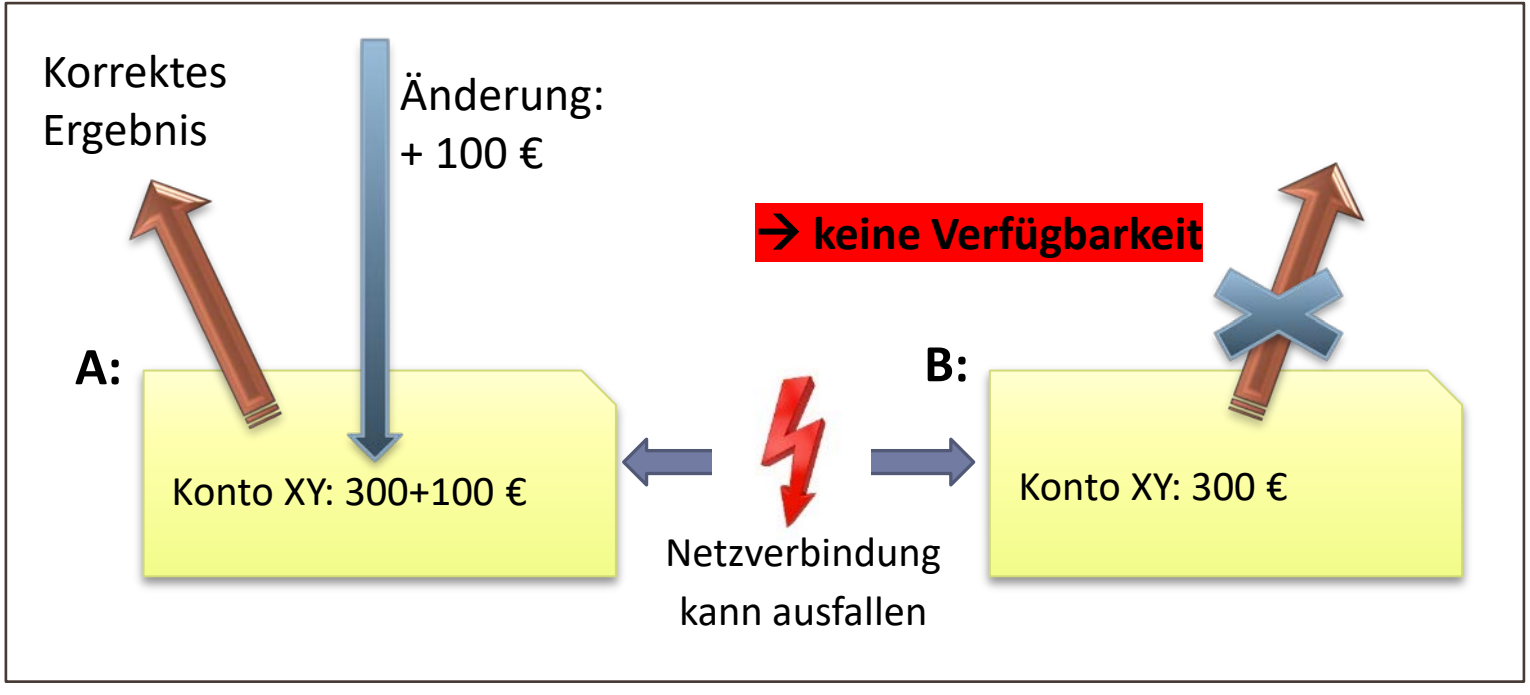
Beispiel: Realisierung als PA-System

- Netzausfall wird kurzfristig toleriert
- Hochverfügbar, aber zwischenzeitlich keine Konsistenz



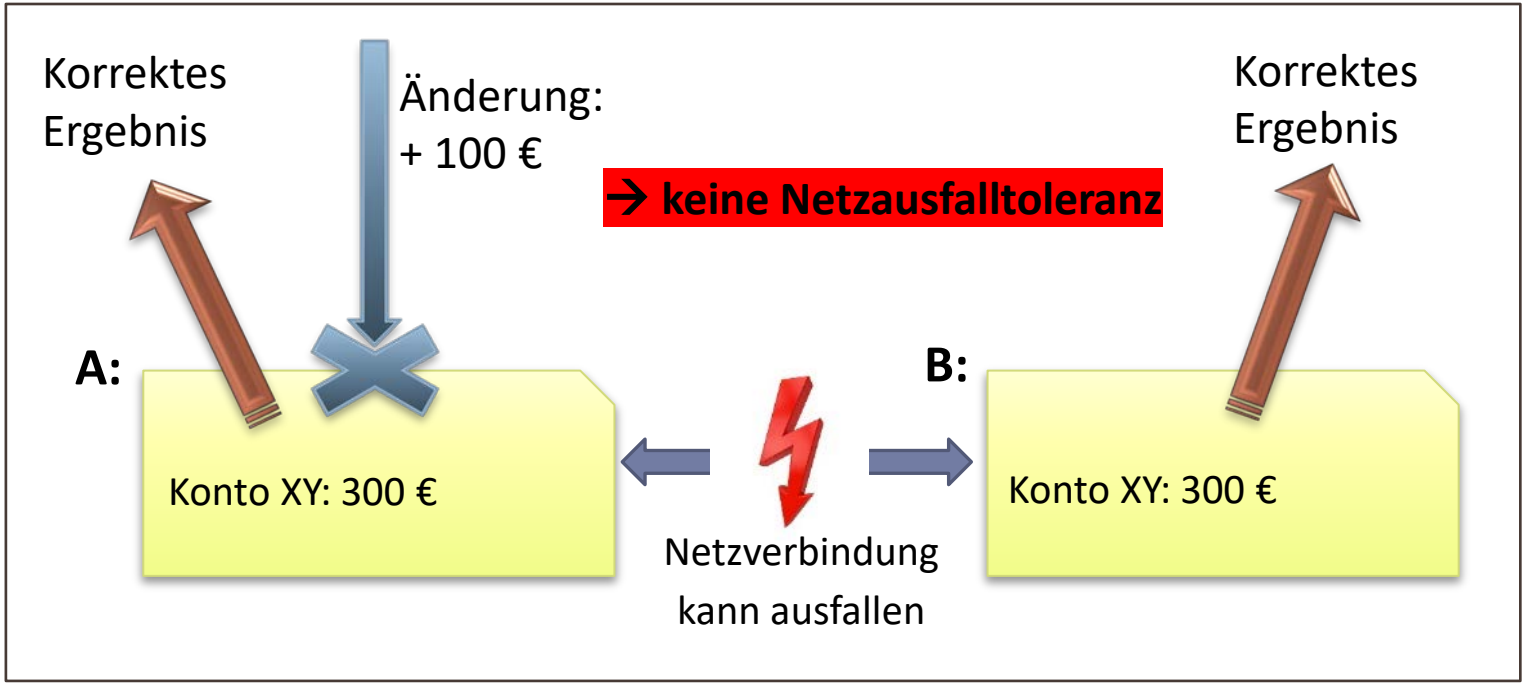
Beispiel: Realisierung als CP-System

- Netzausfall führt zum Abschalten des Knoten B
- Konsistenz ist gesichert auf Kosten der Verfügbarkeit



Beispiel: Realisierung als CA-System

- Netzausfall verhindert Änderungen an den Knoten
- Konsistenz auf Kosten der Netzausfalltoleranz



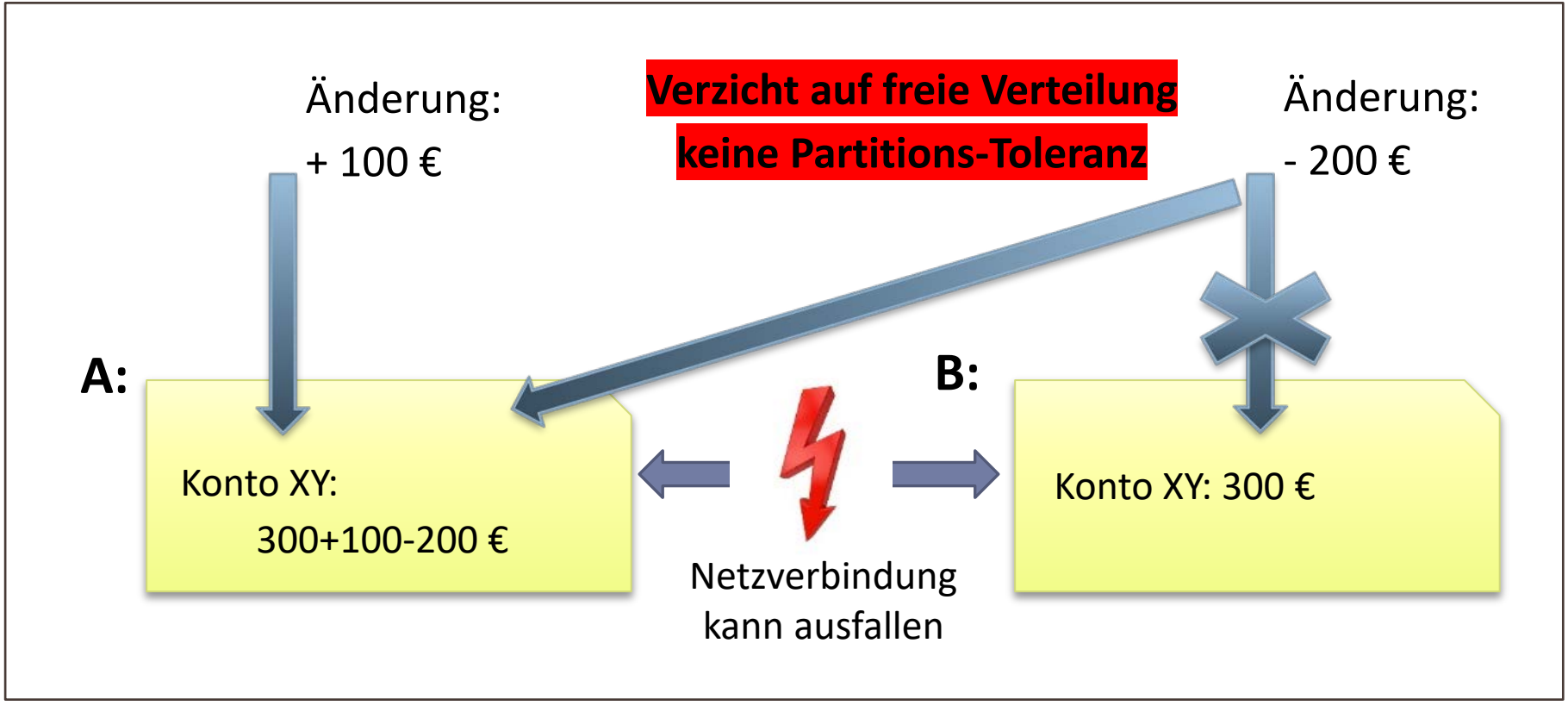
Szenario Banking

- Extrem wichtig: Konsistenz ➔ C
- Konsistenz:
  - Alle Banktransaktionen müssen überall sofort aktualisiert sein
- Verfügbarkeit:
  - Ausfälle von Teilen des Netzes soll der User nicht bemerken
  - Daten müssen aktuell sein ➔ **A**
- Fehlertoleranz:
  - Fehler im Netz sind nicht akzeptabel, damit kein P
- Also: **CA**

Szenario Banking: Vorgehensweise

- Bei Netzausfall eines Bankterminals:
  - Aktualisierungen von außerhalb sind nicht mehr zustellbar!
  - Kontenbewegungen werden außerhalb nicht erkannt!
- Folgerung:
  - Wir greifen nicht mehr auf andere Knoten zu, sondern schalten gegebenenfalls auf den Ausgangsknoten um (Availability!)
  - Damit sind wir nicht mehr netztolerant (kein **P**)!

Szenario Banking: CA-System



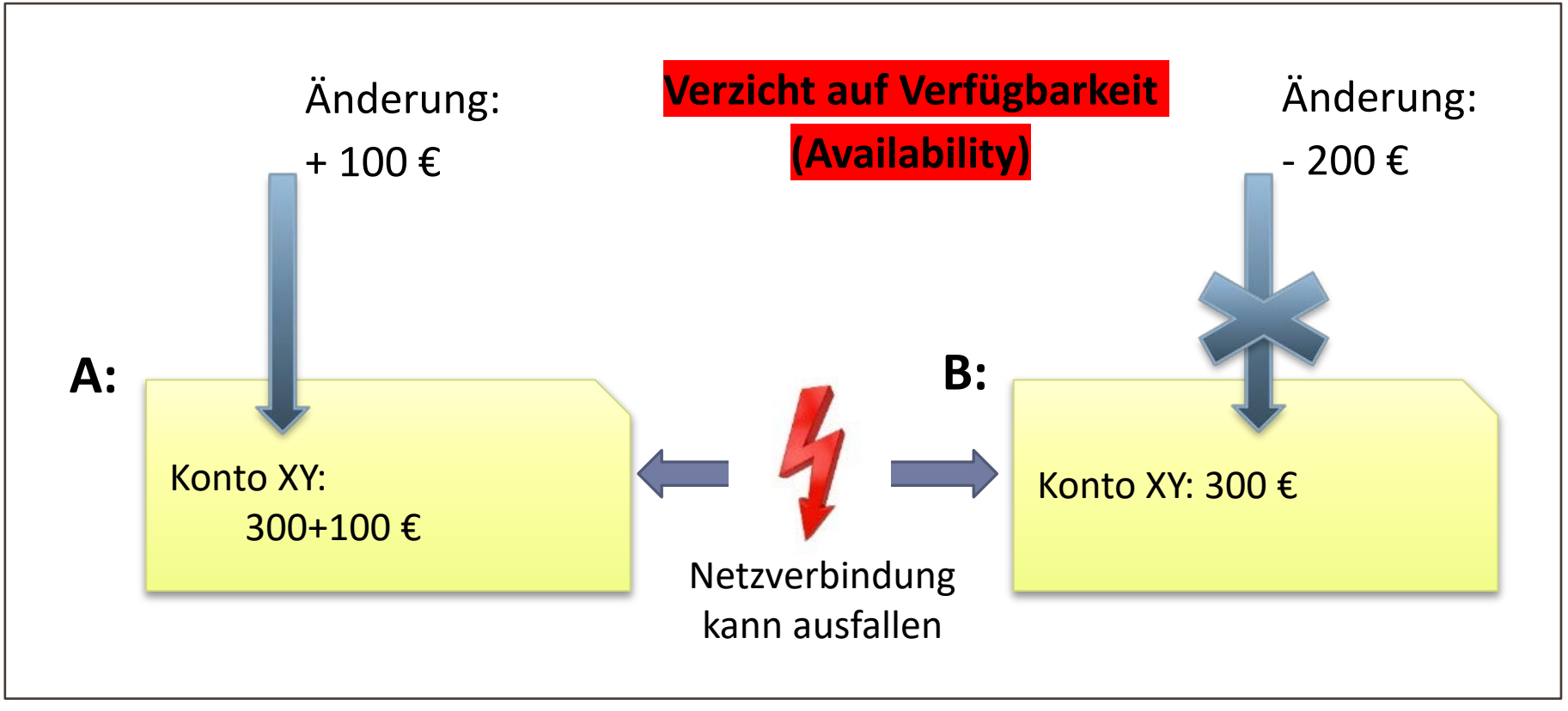
Szenario Banking: Alternative

- Extrem wichtig: Konsistenz ➔ **C**
- Konsistenz:
  - Alle Banktransaktionen müssen überall sofort aktualisiert sein
- Fehlertoleranz:
  - Bei Netzausfall werden Knoten abgeschaltet, die nicht mehr mit dem Hauptknoten verbunden werden können, so dass auf diese Knoten nicht mehr zugegriffen werden kann.
  - Wenn Netz wieder verfügbar: Daten werden ausgetauscht, Knoten wird wieder aktiviert. Also: verzögertes Zustellen ➔ **P**
- Verfügbarkeit:
  - Jeder Ausfall ist ärgerlich, wird aber hingenommen, also kein **A**
- Also: **CP**

Szenario Banking CP: Vorgehensweise

- Hat Knoten keine Verbindung mehr (Netzausfall)
  - Kein Zugriff mehr auf die Daten dieses Knotens (kein **A**)
- Folgerung:
  - Auf alle anderen Konten kann weiter zugegriffen werden ➔ **P**
  - Wenn Netz wieder verfügbar, dann Datenabgleich ➔ **C**

Scenario Banking: CP-System



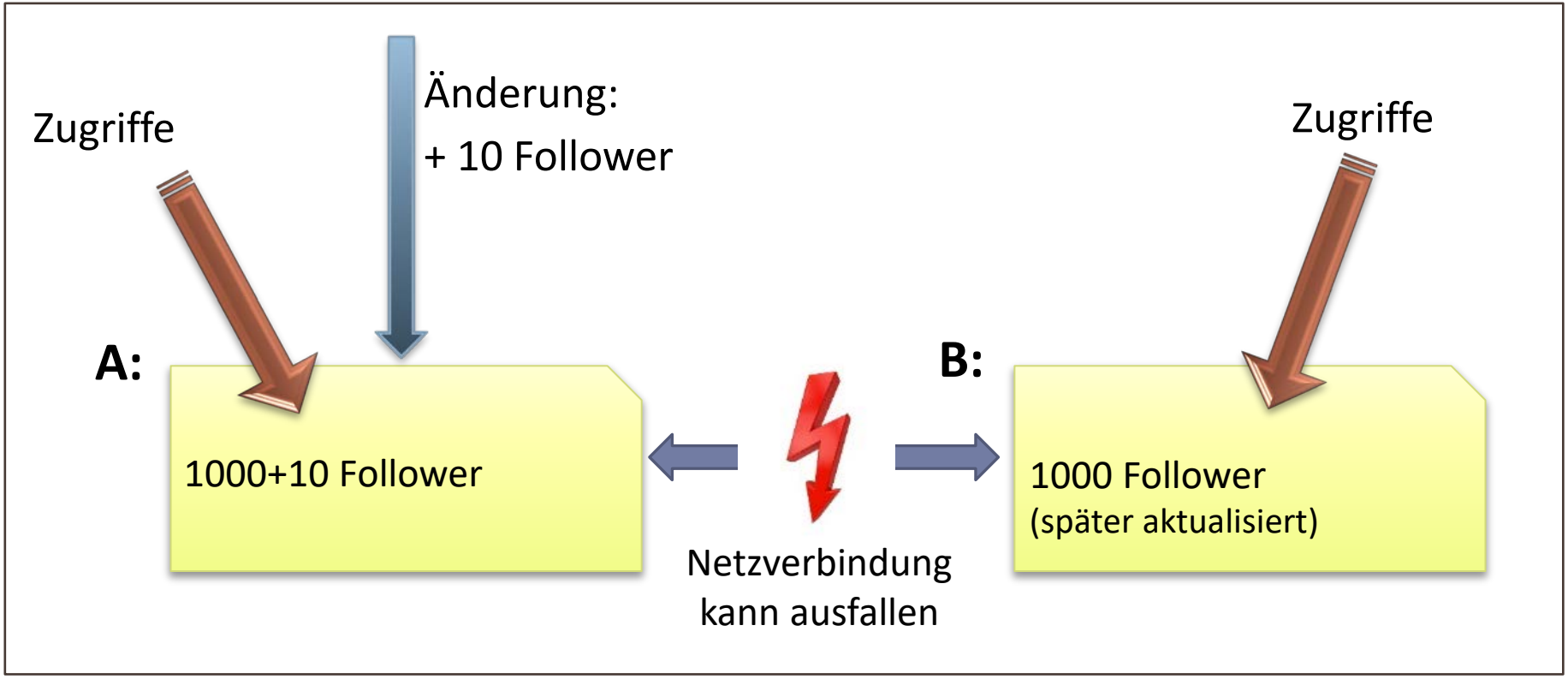
Szenario Soziale Netze

- Wichtig: Verfügbarkeit ➔ A
- Konsistenz:
  - Ist wichtig, muss aber nicht augenblicklich erfolgen
- Fehlertoleranz im Netz:
  - Wünschenswert, da aus Laufzeitgründen die Daten auf möglichst viele Knoten verteilt werden sollen ➔ P
- Also: **PA**

Szenario Soziale Netze: Vorgehensweise

- Beispiel Facebook:
  - Neue Daten müssen nicht sofort überall sichtbar sein
- Folgerung:
  - Alle Knoten arbeiten bei Netzausfällen unabhängig weiter
  - Ist das Netz wieder verfügbar, so werden die Daten peu à peu aktualisiert
  - Nach einer gewissen Zeit ist das System wieder konsistent

Szenario Soziale Netze: PA-System



Das Konsistenzmodell BASE

- **B**asically **A**vailable
  - Die Verfügbarkeit ist wichtiger als die Konsistenz
- **S**oft State
  - Konsistenz wird nach Transaktionsende fließend (soft) erreicht
- **E**ventual Consistency
  - Letztendlich wird die Konsistenz erreicht und garantiert

- Im klassischen Modell steht konträr dazu: **ACID**

Überblick über moderne DB-Systeme (1)

- Key/Value-Datenbanken
  - Daten werden mit dem Schlüssel abgelegt
  - z.B. Amazon System Dynamo, Riak
  - Leicht zu skalieren, in Cloud-Systemen gerne angewendet
- Dokumentenbasierte Datenbanken
  - Als Dokumente abgelegt ohne fest vorgegebene Strukturen
  - Basiert auf Lotus Notes
  - z.B. MongoDB, CouchDB, OrientDB

Überblick über moderne DB-Systeme (2)

- Spaltenorientierte Datenbanken
  - Daten werden spaltenweise verwaltet und gespeichert
  - Meist aber Mischformen mit Key/Value,
  - Anwendung in Big Table Konzept von Google
  - z.B. HBase von Apache, Cassandra von Apache
- Graphen-Datenbanken
  - Grundlage ist die Graphentheorie, basiert auf Graphen
  - Sehr gut für rekursive Suche geeignet, für Navis, für Geodaten-banken, für soziale Netzwerke
  - z.B. GraphDB von Sones, Open Source Neo4J, OrientDB

Zuordnung moderner DB-Systeme

	CA-System	CP-System	PA-System
Relationale Datenbanken	Oracle SQL Server MySQL, …		
Key-Value Datenbanken		BerkeleyDB	Dynamo Riak
Dokumentenbasierte Datenbanken		MongoDB	CouchDB
Spaltenorientierte Datenbanken		Big Table HBase	Cassandra

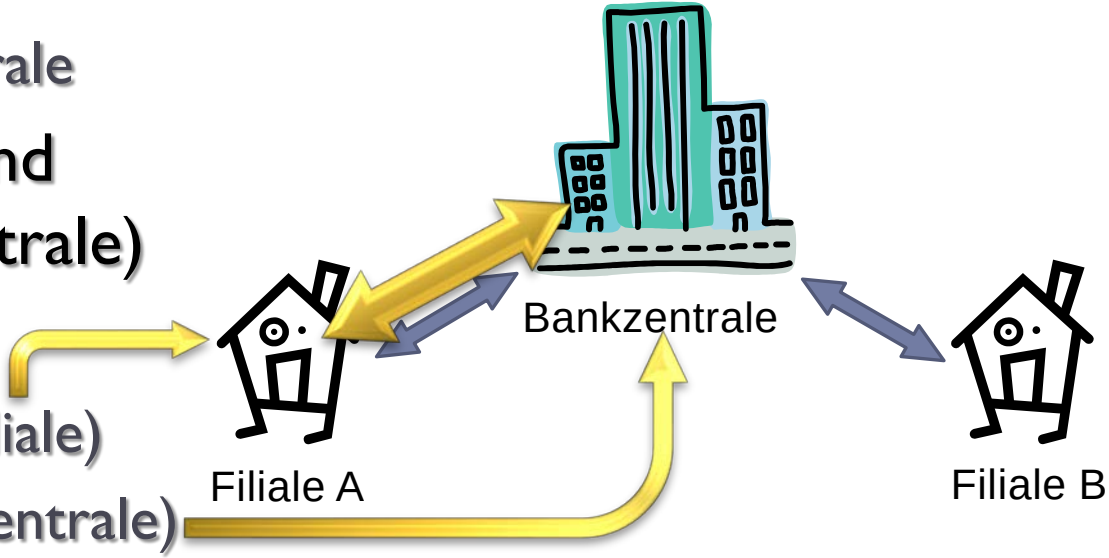
Zwei-Phasen-Commit

- **Gegeben:**
  - Daten sind auf mindestens 2 Datenbanken verteilt
  - CA-System ist erforderlich
  - z.B. Überweisung von einer Bank auf eine andere

- **Idee:**
  - Jede Datenbank führt lokal ein eigenes Transaktionsprotokoll
    - → Phase 1
  - Übergreifend gibt es ein globales Transaktionsprotokoll
    - → Phase 2

Die Idee des Zwei-Phasen-Commits

- Überweisung
  - von Filiale A zur Zentrale
- Zwei Datenbanken sind involviert (Filiale, Zentrale)
- Also:
  - Lokale Transaktion (Filiale)
  - Lokale Transaktion (Zentrale)
  - Koordination (Globale Transaktion)



Das Zwei-Phasen-Commit

1. Jede Datenbank arbeitet getrennt im Transaktionsbetrieb
   - Jede Datenbank macht eigene Recovery (z.B. Logs, Checkpoints)
   - Jede Datenbank beendet mit einem „lokalen“ Commit
2. Eine Datenbank ist zusätzlich der Koordinator
   - Der Koordinator startet eine „globale“ Transaktion
   - Diese überwacht die „lokalen“ Transaktionen
   - Bei Erfolg aller lokalen Transaktionen wird dies an alle zurückgemeldet
   - Die „lokalen“ Commits werden dann in „globale“ umgewandelt

Algorithmus des 2-Phasen-Commits

Lokales Abarbeiten einer Transaktion	Jede betroffene Datenbank startet eine lokale Transaktion: Lokale Änderungen werden in der lokalen Logdatei protokolliert. (Beginn der **Phase 1**)
Melden des Transaktionsendes	Am Ende einer lokalen Transaktion erfolgt eine entsprechende Meldung (Commit bzw. Rollback) an den Koordinator (Ende der **Phase 1**).
Globales Transaktionsende	Der Koordinator sammelt alle lokalen Meldungen. Liegen nur erfolgreiche Rückmeldungen vor, so wird ein globales Commit, ansonsten ein Rollback eingetragen (**Phase 2**).
Endgültiges lokales Transaktionsende	Das Ergebnis der globalen Transaktion wird an alle lokalen Rechner geschickt. Jeder lokale Rechner übernimmt das globale Ergebnis (Commit oder Rollback) als endgültiges. Erst jetzt ist die Transaktion abgeschlossen (Ende der **Phase 2**).

Zusammenfassung: 2-Phasen-Commit

- Der transaktionsübergreifende Commit wird sichergestellt
- Die Konsistenz wird datenbankübergreifend garantiert
- Der 2-Phasen-Commit wird kommerziell angewendet
- Das Protokoll ist extrem aufwendig
  - da der Koordinator ständig die lokalen Transaktionen überwachen muss
  - da auch Netzausfälle mit einkalkuliert werden müssen
  - da Netzverzögerungen nicht sofort zum Abbruch der globalen Transaktionen führen sollen

Zusammenfassung

- Verteilte Datenbanken erhöhen die Zugriffsgeschwindigkeit und die Verfügbarkeit auf Kosten der sofortigen Konsistenz
- ACID wird durch BASE ersetzt!
- Bankanwendungen arbeiten weiter mit CA, eventuell CP
- Soziale Netzwerke arbeiten mit AP
- Zwei-Phasen-Commit garantiert sofortige Konsistenz, ist aber sehr aufwendig