Verteilte Datenbanken

Einführung

- Daten sind auf mehreren Knoten *(Sites)* gespeichert, jeweils verwaltet durch ein DBMS, das unabhängig auf den einzelnen Knoten läuft
  - z.B. ein Teil der Daten auf einem PC unter MS SQL-Server, ein anderer Teil auf einem Solaris-Rechner unter Oracle
- Verteilte Datenunabhängigkeit: Benutzer brauchen nicht zu wissen, wo sich die Daten wirklich befinden (Erweiterung des Prinzips der physischen und logischen Datenunabhängigkeit bei Datenbanksystemen). Dabei kann es sich um Fragmente oder Kopien der Daten handeln. Diese Eigenschaft wird auch als Transparenz bezeichnet.
- Verteilte Transaktionsatomarität: Benutzer müssen in der Lage sein, Transaktionen zu schreiben, die auf mehreren Knoten laufen, sich aber wie lokale Transaktionen verhalten (ACID-Eigenschaften).
- Sehr hoher administrativer Overhead, um diese Eigenschaften zu unterstützen, insbes. auf global verteilten Knoten.

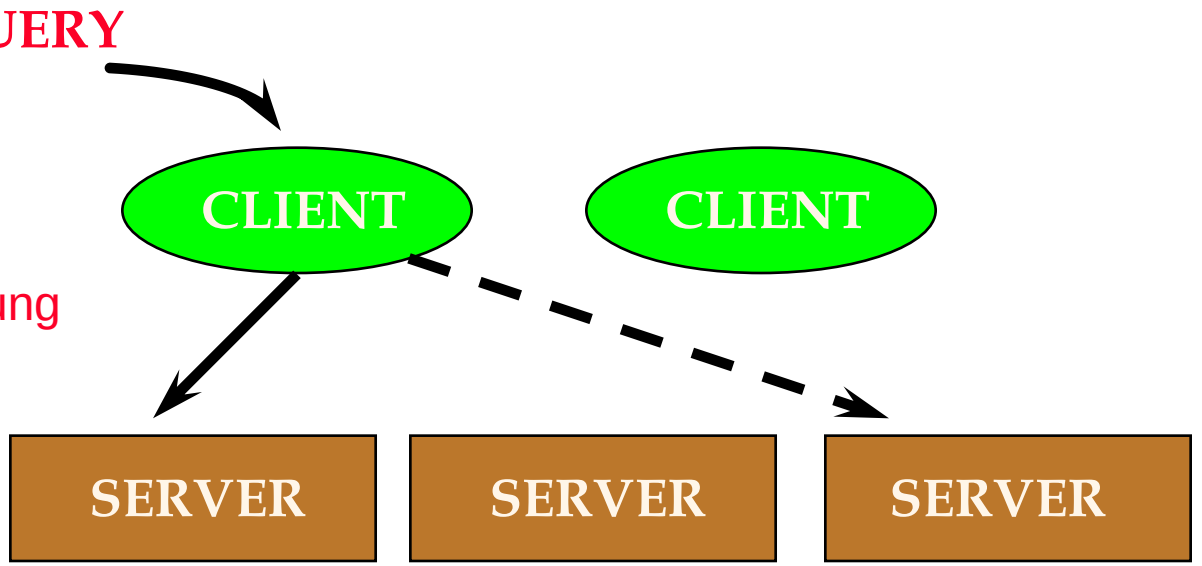
Typen Verteilter Datenbanken

- Homogen: Auf jedem Knoten läuft der gleiche Typ des DBMS.
- Heterogeneous: Auf verschiedenen Knoten laufen verschiedene DBMS (verschiedene relationale oder auch nicht-relationale DBMS)

Architekturen Verteilter DBMSe

- Client-Server

  Client sendet eine Query an einen einzigen Knoten. Gesamte Query-Verarbeitung findet auf Server statt.
  - *Thin* vs. *Fat* Clients.
  - Mengen-orientierte Kommunikation, client-seitiges Caching.

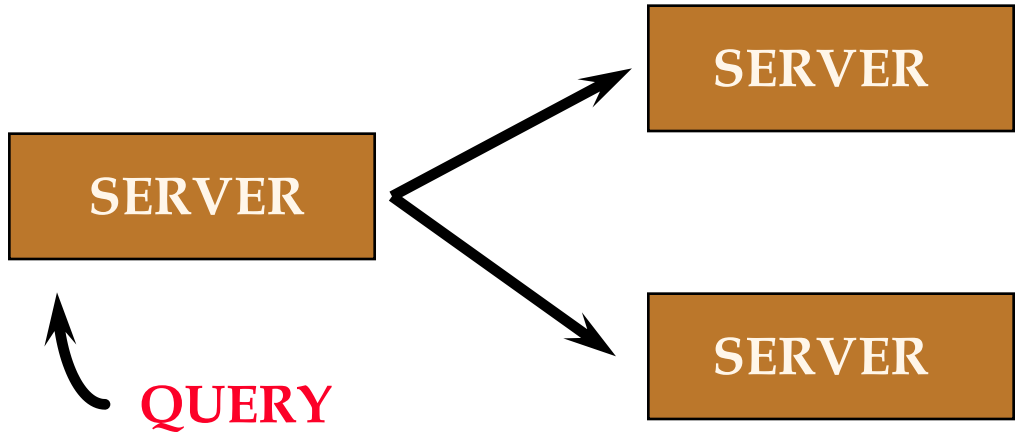


- Collaborating-Server

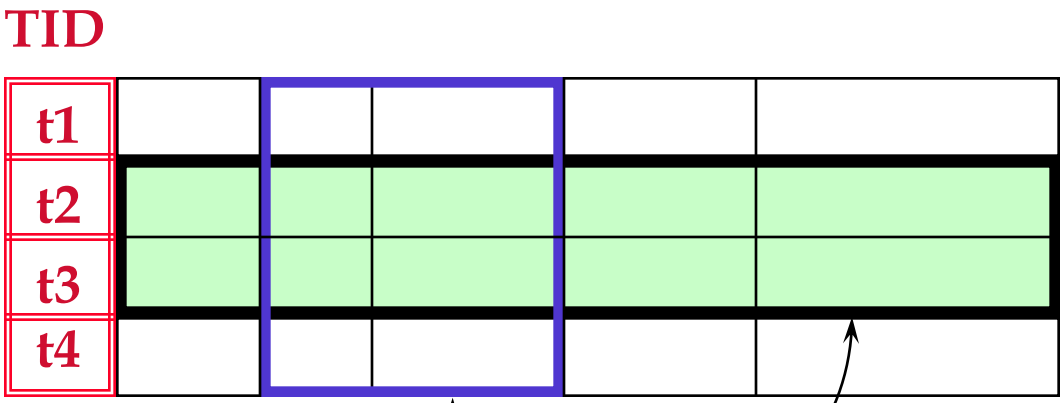
  Query kann mehrere Knoten umfassen

- Middleware

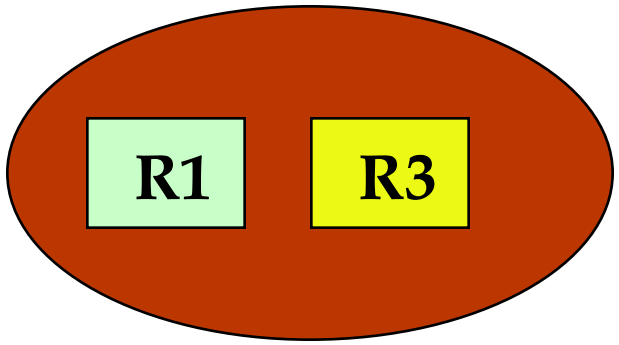
  Spezielle Softwareschicht zur Koordination von Queries / Transaktionen zwischen mehreren DB-Servern



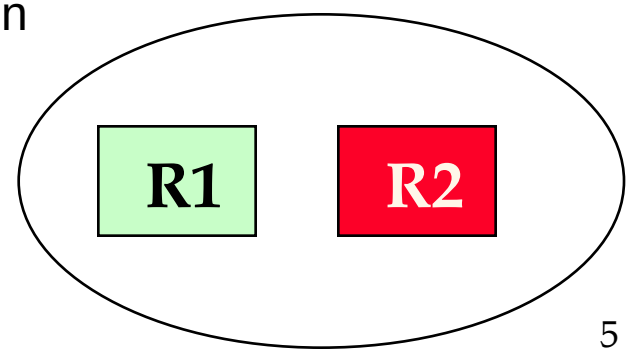
Storing Data



- Fragmentierung
  - Horizontal: Gewöhnlich disjunkt
  - Vertikal: Zerlegung in verlustfreie Join-Relationen; Tids (TupelIdentifier).
- Replikation
  - Schafft erhöhte Verfügbarkeit
  - Schnellere Query-Verarbeitung
  - Synchron vs. Asynchron:
    - Unterscheidet, wie die Kopien aktuell gehalten werden bei Änderung der Relation



SITE A



Verteilte Katalog-Verwaltung

- Kontrolle darüber notwendig, wie die Daten über die einzelnen Knoten hinweg verteilt werden
- Es muß möglich sein, jedes Replikat von jedem Fragment zu identifizieren. Zur Bewahrung der lokalen Autonomie:
  - <**local-name**, **birth-site**>

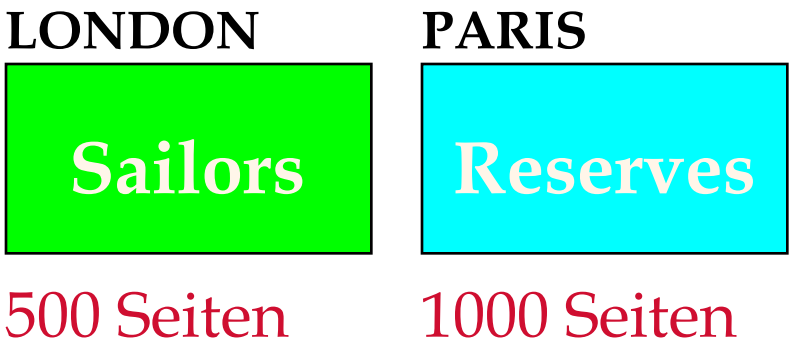
    = Global eindeutige Kombination aus lokalem Namen (zugewiesen am Ursprungsknoten der Relation) und Ursprungsknoten (wo die Relation erzeugt wurde)
- Site Catalog: Beschreibt alle Objekte (Fragmente, Replikate) auf einem Knoten und kontrolliert die Replikate der Relationen, die auf diesem Knoten erzeugt wurden
  - Zum Auffinden einer Relation genügt das Suchen im Katalog seines Ursprungsknotens
  - Ursprungsknoten ändert sich niemals, auch beim Verschieben der Relation zu anderen Knoten

Verteilte Queries

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3
    AND S.rating < 7
```

- Horizontal Fragmentiert: Tuple mit rating < 5 in Shanghai, >= 5 in Tokio.
  - Muß die Funktionen SUM(age), COUNT(age) auf beiden Knoten berechnen
  - Wenn die WHERE-Klausel hieße: S.rating>6, wäre nur ein Knoten beteiligt
- Vertikal Fragmentiert: *sid* und *rating* in Shanghai, *sname* und *age* in Tokyo, *tid* in beiden
  - Relation muß über den Tupel-Identifier *tid* durch Join rekonstruiert werden, danach Auswertung der Query
- Repliziert: Kopien der Relation Sailors auf beiden Seiten
  - Wahl des Knoten basiert auf lokalen Kosten und Transportkosten

Verteilte Joins



- Fetch as Needed, Seitenorientierter Nested Loop, Sailors als äußere Relation:
  - **Kosten:** 500 D + 500 * 1000 (D+S)
  - **D** Kosten zum Lesen/Schreiben einer Seite; **S** sind die Transportkosten für eine Seite
  - Wenn die Query nicht in London gestellt wurde, müssen Transportkosten zum Query-Knoten hinzuaddiert werden
  - Auch möglich ein Index Nested Loop in London, Lesen matchender Tupel nach London bei Bedarf (fetch as needed)
- Transport zu einem Knoten: Transportiere Reserves nach London.
  - Kosten: 1000 S + 4500 D (Sort Merge Join; Kosten = 3*(500+1000))
  - Wenn Resultat sehr groß ist, ist es wohl besser, beide Relationen auf den Ergebnisknoten zu senden und dort den Join auszuführen.

Semijoin

- In London, Projiziere in Sailors die Join-Spalten heraus und transportiere dies nach Paris
- In Paris, joine die Sailors-Projektion mit Reserves.
  - Resultat heißt **Reduktion** von Reserves bezüglich Sailors
- Transportiere die Reduktion von Reserves nach London
- In London, joine Sailors mit der Reduktion von Reserves.
- Idee: Kostenabwägung:
  - Berechnungskosten für Projektion + Transportkosten + Berechnungskosten für Join (Reduktion) + Transportkosten vs. Transportkosten der vollständigen Relation Reserves
- Besonders sinnvoll, wenn es eine Selektionsbedingung auf Sailors gibt und die Antwort in London benötigt wird

Verteilte Query-Optimierung

- Kostenbasierter Ansatz:
  Untersuche alle Pläne; Wähle den billigsten aus; ähnlich der Optimierung in zentralisierten Systemen
  - Unterschied 1: Kommunikationskosten müssen untersucht werden
  - Unterschied 2: Lokale Knotenautonomie muß respektiert werden (z.B. wenn keine Statistiken verfügbar)
  - Unterschied 3: Neue verteilte Join-Methoden (z.B. Semi-Join)
- Anfrageknoten konstruiert einen globalen Plan, mit vorgeschlagenen lokalen Plänen, die die Verarbeitung auf jedem Knoten beschreiben:
  - Optimierung eines vorgeschlagenen lokalen Plans auf einem Knoten möglich

Ändern verteilter Daten

- Synchrone Replikation:
  Alle Kopien einer modifizierten Relation (Fragment) müssen geändert werden vor dem Commit der modifizierenden Relationen
  - Datenverteilung ist transparent für die Benutzer
- Asynchrone Replikation:
  Kopien einer modifizierten Relation werden nur periodisch geändert; verschiedene Kopien können in der Zwischenzeit inkonsistent werden
  - Benutzer müssen sich der Datenverteilung bewußt sein
  - Heutige Produkte folgen diesem Ansatz (kommerziell verfügbare Replication Server)

Synchrone Replikation

- Voting: Transaktion muß eine Mehrheit der Kopien schreiben, um ein Objekt zu modifizieren; Transaktion muß genug Kopien lesen, um sicher zu sein, zumindest eine jüngste Kopie zu sehen:
  - Beispiel: 10 Kopien; 7 geschrieben für Update; muß somit 4 Kopien lesen
  - Jede Kopien hat eine Versionen-Nummer
  - Gewöhnlich nicht attraktiv weil Reads sehr häufig sind
- Read-any Write-all: Writes sind langsamer und Reads sind schneller (da auf lokalen Kopien) - relativ zum Voting
  - Verbreitetster Ansatz zur synchronen Replikation
  - heißt auch ROWA (Read-once Write-all)
- Auswahl des Algorithmus hängt vom Lastprofil ab (Frequenz der Reads und Writes)

Synchrone vs. asynchrone Replikation

- Bevor eine Update-Transaktion ein Commit machen kann, muß sie Sperren auf allen modifizierten Kopien erhalten:
  - Sendet Sperranforderungen an die anderen Knoten, wartet auf Antwort, hält die anderen Sperren!
  - Wenn Knoten oder Verbindungen fehlerhaft sind, kann die Transaktion kein Commit machen, bis ein Backup aktiv ist
  - Selbst, wenn kein Fehler auftritt, folgt das Commit einem teuren *Commit-Protokoll* mit vielen Messages
- Daher ist die Alternative *asynchrone Replikation* weit verbreitet:
  - Erlaubt der Update-Transaktion ein Commit, bevor alle Kopien geändert wurden (wobei Leser trotzdem nur eine Kopie sehen)
  - Benutzer müssen wissen, welche Kopie sie lesen und daß Kopien für kurze Zeitabschnitte inkonsistent sein können
- Zwei Ansätze: Primary Site and Peer-to-Peer Replikation
  - Differenz besteht darin, wieviele Kopien änderbar, d.h. *Master-Kopien,* sind

Peer-to-Peer-Replikation

- Mehr als eine der Kopien eines Objekts kann Master sein
- Änderungen auf einer Master-Kopie müssen irgendwie zu anderen Kopien hin propagiert werden
- Wenn zwei Master-Kopien in einer Weise geändert werden, die zum Konflikt führt, muß dies aufgelöst werden, z.B:
  - Knoten 1: Alter von Joe auf 35 geändert
  - Knoten 2: Änderung auf 36
- Am besten einsetzbar, wenn keine Konflikte entstehen können:
  - Beispiel: Jeder Masterknoten besitzt ein disjunktes (horizontales) Fragment (z.B. Änderungen der Daten von deutschen Angestellten in Frankfurt, Änderung bei indischen Angestellten in Madras, obwohl gesamte Datenmenge in beiden Knoten gespeichert)
  - Beispiel: Änderungsrechte liegen zu einem bestimmten Zeitpunkt bei nur einem Master

Primary-Site Replikation

- Genau eine Kopie einer Relation wird bestimmt als Primär- oder Masterkopie. Replikate auf anderen Knoten dürfen nicht direkt geändert werden
  - Die Primärkopie ist öffentlich (publish)
  - Andere Seiten abonnieren diese Relation bzw. Fragemente davon (*subscribe*); diese werden als Sekundärkopien bezeichnet
- Hauptproblem: Wie werden Änderungen auf der Primärkopie zu den Sekundärkopien propagiert?
  - Erfolgt in zwei Schritten:
    1. Erfassen der Änderungen, die von den erfolgreich beendeten Transaktionen gemacht wurden (*Capture*)
    2. Durchführen dieser Änderungen (*Apply*)

Implementierung des 1. Schritts: Capture

- Log-Basierte Erfassung: Der Log (angelegt für Recovery) wird verwendet zur Generierung einer Tabelle der geänderten Daten = Change Data Table (CDT)
  - Beim Schreiben des aktuellen Log-Abschnitts auf Platte muß darauf geachtet werden, daß die Änderungen wieder entfernt werden, die von später abgebrochenen Transaktionen stammen
- Prozedurale Erfassung: Eine Prozedur, die automatisch aufgerufen wird (*Trigger*), macht die Erfassung;
  - Typisch ist die Aufnahme eines Schnappschusses der Primärkopie (Snapshot)
- Log-Basierte Erfassung ist besser
  - Geringerer Mehraufwand und somit billiger
  - Geringere Verzögerung zwischen Änderung der Daten und der Propagierung der Änderungen - somit schneller
  - Nachteil: erfordert vertieftes Verständnis der Struktur der system-spezifischen Logs

Implementierung des 2. Schritts: Apply

- Der Apply-Prozeß auf dem Sekundärknoten bekommt periodisch Snapshots oder Änderungen der Änderungstabelle vom Primärknoten und ändert die Kopie.
  - Periode kann zeitbasiert sein oder definiert durch Benutzer / Applikation
- In manchen System kann Replikat eine Sicht auf der modifizierten Relation sein!
  - In diesem Fall besteht die Replikation aus einem inkrementellen Update der materialisierten Sicht, wenn sich die Relation ändert
- Log-basierte Erfassung zusammen mit einem kontinuierlichen Apply der Änderungen minimiert die Verzögerung bei der Propagierung der Änderungen.
- Prozedurale Erfassung zusammen mit einem applikations-getriebenen “Apply“ der Änderungen ist die flexibelste Art, Änderungen zu verarbeiten.

Data Warehousing und Replikation

- Aktueller Trend: Aufbau gigantischer *Data Warehouses*, die aus vielen Knoten gespeist werden
  - Erlaubt komplette Anfragen zur Entscheidungsunterstützung (Decision Support) auf Daten über die ganze Organisation hinweg
- Warehouses können als eine Instanz von asynchroner Replikation gesehen werden
  - Source-Daten typischerweise kontrolliert durch verschiedene DBMS
  - Schwerpunkt auf Bereinigung der Daten (*Data Cleaning*) und Beseitigung von syntaktischer und semantischer Heterogenität (z.B. $ vs. €) beim Erzeugen der Replikate.

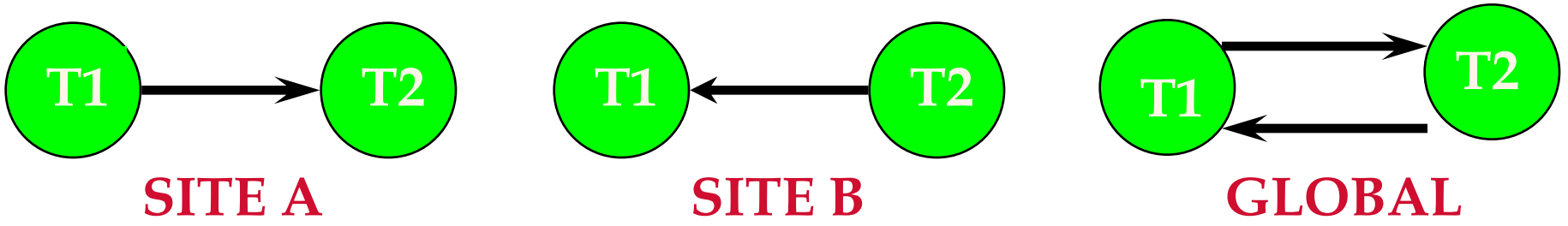
- Schnappschuß-Replikation:
  - Wird durch zunehmende Anzahl kommerzieller Produkte unterstützt (z.B. IBM Data Propagator, Oracle Replication Server)

Verteiltes Sperren

- Wie werden Sperren für Objekte über mehrere Knoten hinweg verwaltet?
  - Zentralisiert: Ein Knoten für Sperren verantwortlich
    - Zentraler Knoten stellt Engpaß für Leistung und Verfügbarkeit dar (anfällig gegenüber *Single Site Failure*)
  - Primärkopie: Alle Sperren für ein Objekt auf dem Primärkopie-Knoten für dieses Objekt verwaltet
    - Lesen erfordert Zugriff auf den sperrenden Knoten und ebenso auf den Knoten, wo das Objekt gespeichert ist.
  - Voll Verteilt: Sperren für eine Kopie werden auf dem Knoten verwaltet, wo die Kopie gespeichert ist
    - Sperren auf allen Knoten beim Schreiben eines Objekts

Verteilte Deadlock-Erkennung

- Jeder Knoten unterhält einen lokalen Wartegraphen
- Ein globaler Deadlock könnte existieren, selbst wenn die lokalen Graphen keine Zyklen enthalten:



Drei Lösungen:

- Zentralisiert: sende alle lokalen Graphen zu einem Knoten
- Hierarchisch: organisiere Knoten in einer Hierarchie und sende lokale Graphen zum Vorgänger in dieser Hierarchie
- Timeout: Abbruch der Transaktion, wenn sie zu lange wartet

Verteilte Recovery

- Zwei mögliche neue Probleme:
  - Neue Fehlerarten: z.B. Fehler in der Kommunikations-verbindung, Fehler auf einem entfernten Knoten, wo eine Subtransaktion läuft
  - Wenn “Sub-Transaktionen” einer Transaktion auf verschiedenen Knoten laufen, müssen alle oder keine ein Commit machen.
    Erfordert ein Commit-Protokoll, um dies zu erreichen
- Ein Log wird auf jedem Knoten unterhalten wie in einem zentralisierten DBMS, und Aktionen des Commit-Protokolls werden zusätzlich protokolliert.
- Meistverbreitetes Protokoll in kommerziellen DBMS: *2-Phase-Commit (2PC)*

Two-Phase Commit (2PC)

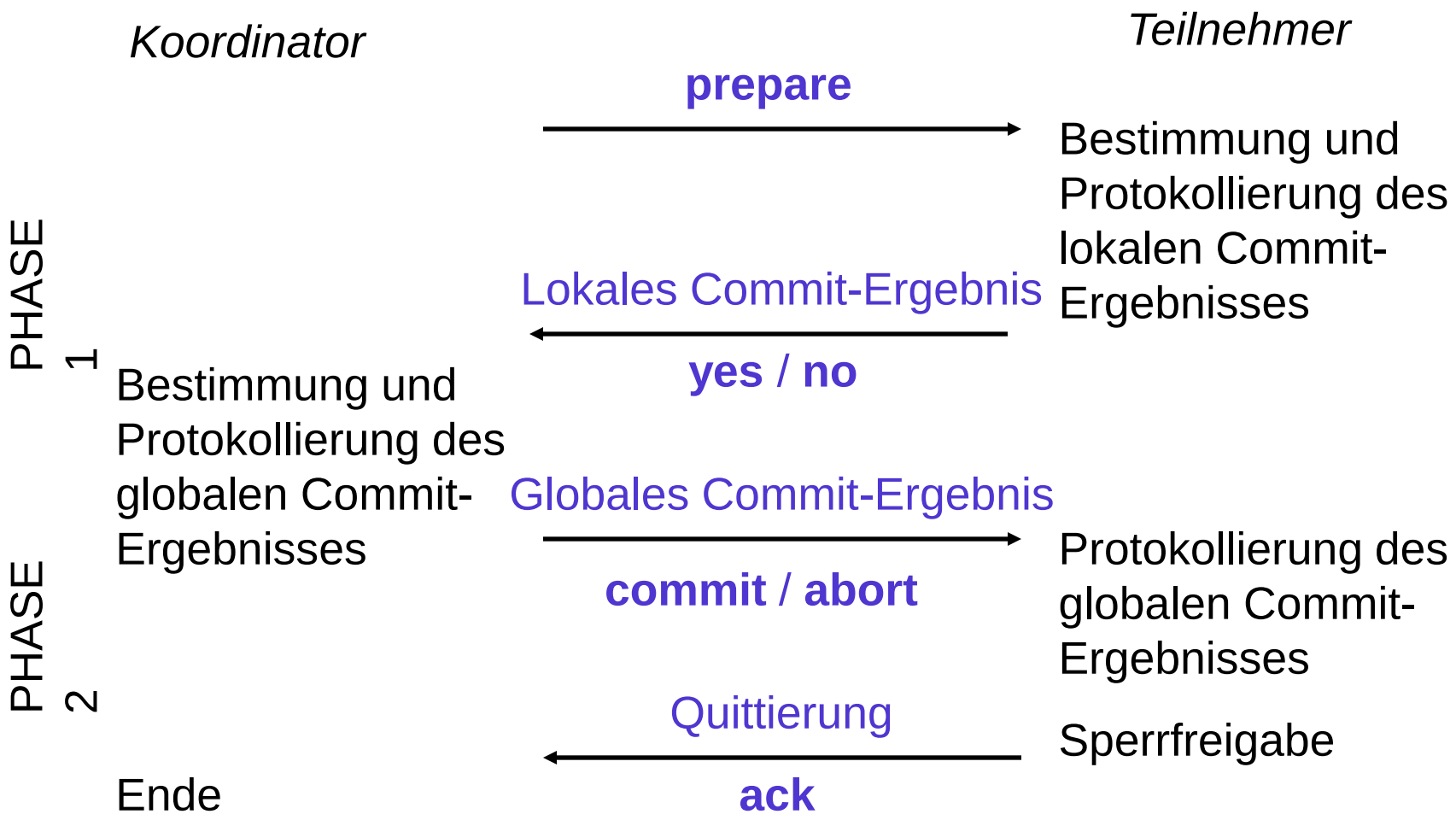
- Knoten, von dem aus die Transaktion beginnt, ist Koordinator; andere Knoten, wo sie ausgeführt wird, sind Teilnehmer
- Wenn eine Transaktion ein Commit machen möchte:
  1. Koordinator sendet eine **prepare** Message an jeden Teilnehmer, um deren lokales Commit-Ergebnis in Erfahrung zu bringen
  2. Nach Empfang der prepare-Nachricht sichert der Teilnehmer bei einer erfolgreich zu Ende gekommenen Subtransaktion durch das Ausschreiben von noch ungesicherten Log-Daten eines **prepare** Satzes. Anschließend sendet der Teilnehmer eine **yes** Nachricht an den Koordinator und wartet auf den Ausgang der globalen Transaktion.
     Für eine gescheiterte Subtransaktion wird ein **abort** Satz in den lokalen Log geschrieben und eine **no** Message zum Koordinator geschickt. Die Subtransaktion wird vom Teilnehmer zurückgesetzt.
  3. Haben alle Teilnehmer mit **yes** geantwortet, schreibt der Koordinator einen **commit** Satz in die lokale Log-Datei, woraufhin die globale Transaktion als erfolgreich beendet gilt. Danach wird eine **commit** Message gleichzeitig an alle Agenten verschickt.

Two-Phase-Commit (Forts.)

Stimmte mindestens ein Agent mit **no** , so ist auch die globale Transaktion gescheitert. Der Koordinator schreibt einen **abort** Satz in seinen Log und sendet eine **abort** Message an alle Teilnehmer, die mit yes gestimmt haben.

4. Teilnehmer schreiben einen **abort** oder **commit** Satz in die Log-Datei, abhängig von der Antwort des Koordinators. Gehaltene Sperren werden freigegeben.
   Anschließend sendet der Teilnehmer noch eine Quittung (**ack** Message) an den Koordinator.
5. Nach Eintreffen aller **ack** Nachrichten schreibt der Koordinator einen **end** Satz in seine Log-Datei.

Nachrichtenfluß im 2PC-Protokoll



Anmerkungen zum 2PC

- Zwei Kommunikationsphasen, jeweils durch den Koordinator initiiert :
  - erst Abstimmung (Voting);
  - dann Beendigung (Termination)
- Jeder Knoten darf über einen Abbruch der Transaktion entscheiden
- Jede Message reflektiert eine Entscheidung des Absenders; ihr Überleben ist dadurch gesichert, daß die Nachricht zunächst in lokale Log-Datei geschrieben wird
- Offizielles Commit der globalen Transaktion: Koordinator schreibt Commit-Satz ins Log
- All Log-Sätze für Aktionen des Commit-Protokolls für eine Transaktion enthalten:
  - Transaktions-ID und Koordinator-ID
  - Abort/Commit-Satz des Koordinators enthält auch die IDs alles Teilnehmer

Restart nach dem Ausfall eines Knotens

- Wiederanlauf auf einem Knoten (kann vorher Teilnehmer oder Koordinator gewesen sein)
- Wenn es einen **commit** oder **abort** Log-Satz für eine Transaktion T gibt, aber keinen **end**-Satz, muß ein Undo/Redo für die Transaktion durchgeführt werden
  - Wenn dieser Knoten Koordinator für T ist: Erneutes (periodisches) Senden von **commit/abort** Messages bis alle Quittungen (**acks**) eingetroffen sind
- Wenn es einen **prepare** Log-Satz für Transaktion T gibt, aber kein **commit/abort**, ist dieser Knoten ein Teilnehmer für T
  - Wiederholtes Anrufen des Koordinators, um den Status von T zu bestimmen
  - dann Schreiben eines **commit/abort** Logsatzes
  - Durchführung von Redo/Undo von T
  - Schreiben **end** Log-Satz
- Wenn es nicht mal einen **prepare** Log-Satz für T gibt, einseitiger Abort und Rückgängigmachen von T
  - Dieser Knoten kann Koordinator gewesen sein (hat vielleicht eine prepare-Message vor dem Crash noch abgeschickt)
  - Falls Koordinator gewesen, kommen noch Messages von den Teilnehmern an (Notwendige Antwort: Abort T)

Blockieren

- Wenn der Koordinator einer Transaktion T scheitert, können Teilnehmer die mit yes votiert haben, nicht über Commit oder Abort von T entscheiden, bis der Koordinator wiederhergestellt ist:
  - T ist blockiert.
  - Selbst wenn alle Teilnehmer einander kennen (zusätzlicher Mehraufwand in prepare Message), sind sie blockiert, sofern nicht einer von ihnen mit no votiert hat.

Fehler in Verbindungen und auf entfernten Knoten

- Wenn ein entfernter Knoten während des Commit-Protokolls für eine Transaktion T nicht antwortet, hat das 2 Gründe:
  - Fehler auf diesem Knoten
  - Verbindungsfehler
- Regeln:
  - Wenn der aktuelle Knoten der Koordinator für T ist, sollte T abgebrochen werden.
  - Wenn der aktuelle Knoten ein Teilnehmer ist, der nicht mit yes votiert hat, sollte er T abbrechen.
  - Wenn der aktuelle Knoten ein Teilnehmer ist und mit yes votiert hat, ist er blockiert, solange der Koordinator nicht antwortet

Konsistenzkriterien in verteilten Datenbanken

- Konsistenz der Daten bei Datenmanipulation oder wiederholten Anfragen
- DBS verantwortlich für Konsistenz – auch im verteilten System
  - **A**tomicity: Alles oder Nichts
  - **C**onsistency: Daten werden von einem korrekten Zustand in einen neuen korrekten Zustand überführt
  - **I**solation: Keine gegenseitige Beeinflussung nebenläufiger Transaktionen
  - **D**auerhaftigkeit: Persistenz der Daten und Protokollierung der Operationen
- ABER: Keine ausreichende Verfügbarkeit bei Millionen konkurrierender Benutzer
- Frage: Kann C und I zugunsten der stetigen Verfügbarkeit und Performance aufgegeben werden?
- BASE für verteilte Systeme
  - **B**asically **A**vailable
  - **S**oft-state
  - **E**ventual consistency

Vergleich ACID vs. BASE (nach E. Brewer)

ACID

- Konsistenz
- Isolation durch Sperren
- Fokus auf „commit“
- Verschachtelte Transaktionen
- Verfügbarkeit?
- Konservative Strategie
- Erschwerte Evolution von Schema und Daten

BASE

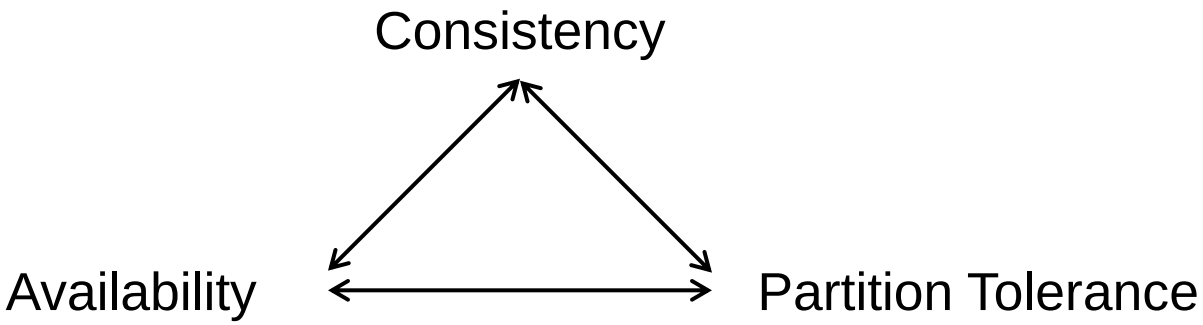
- Vernachlässigte Konsistenz (in gewissen Grenzen)
- Hohe Verfügbarkeit
- Optimistische Strategie
- Einfacher und schneller
- Einfachere Evolution von Daten und Schema

Meistens bedienen sich Datenbanken im Web den Vorzügen aus beiden Ansätzen, so dass ein fließender Übergang entsteht.

CAP-Theorem

Theorem: Es können maximal zwei der Eigenschaften Konsistenz, Verfügbarkeit und Partitionstoleranz gleichzeitig in einem verteilten (Daten) System erreicht werden [Brewer, 2000].

- **Consistency**: Alle Clients sehen zu einem Zeitpunkt denselben Datenbestand.
- **Availability**: Alle Anfragen werden stets beantwortet (Verfügbarkeit).
- **Partitionstoleranz**: Das System arbeitet bei Ausfall einzelner Netzknoten oder Teilnetze weiter.



C, A und P sind graduelle Größen (schnell, langsam, sofort, mit Zeitverzug, …

CAP-Theorem: Beispiele

- Verfügbarkeit – Partitionstoleranz (keine Konsistenz)
  - Domain Name System: Hohe Toleranz gegenüber Ausfall durch hohe Duplizierungsrate, lange Wartezeiten und unterschiedliche Stände bei Propagierung neuer DNS
  - Cloud Computing / Social Networks: Keine Auswirkungen bei Ausfall einzelner Netzteilnehmer und bei Weiterreichung von Nachrichten
- Verfügbarkeit – Konsistenz (keine Partitionstoleranz)
  - Relationale DBMS: Konsistenz! Partitionstoleranz nachrangig durch Zentralisierung
- Konsistenz – Partitionstoleranz (keine Verfügbarkeit)
  - Bankanwendungen: Konsistenz! aber tolerant bei Ausfall einzelner Bankterminals

Eventual Consistency

- Drei Möglichkeiten, Konsistenz wiederherzustellen („Conflict resolution“):
  - **Read repair**: Korrektur, wenn eine Lese-OP eine Inkonsistenz aufdeckt Abschluss der Lese-OP, wenn Konsistenz hergestellt
  - **Write repair**: Korrektur, wenn eine Schreib-OP eine Inkonsistenz entdeckt Abschluss der Schreib-OP, wenn Konsistenz hergestellt
  - **Asynchronous repair**: Konfliktbeseitigung ist Bestandteil von Hintergrundprozessen und damit weder Teil einer Lese- noch Schreib-OP
- Beispiel: Cassandra (Wide Column Store) bietet drei Ebenen zur Konsistenzsicherung
  - ONE: Veränderte Daten werden nach dem Commit an einem Knoten über das Netzwerk verteilt.
  - QUORUM: Veränderte Daten werden zu $\frac{Replikationsfaktor}{2}+1$ Knoten propagiert, bevor der Client informiert wird.
  - ALL: Alle Knoten replizieren die veränderten Daten.