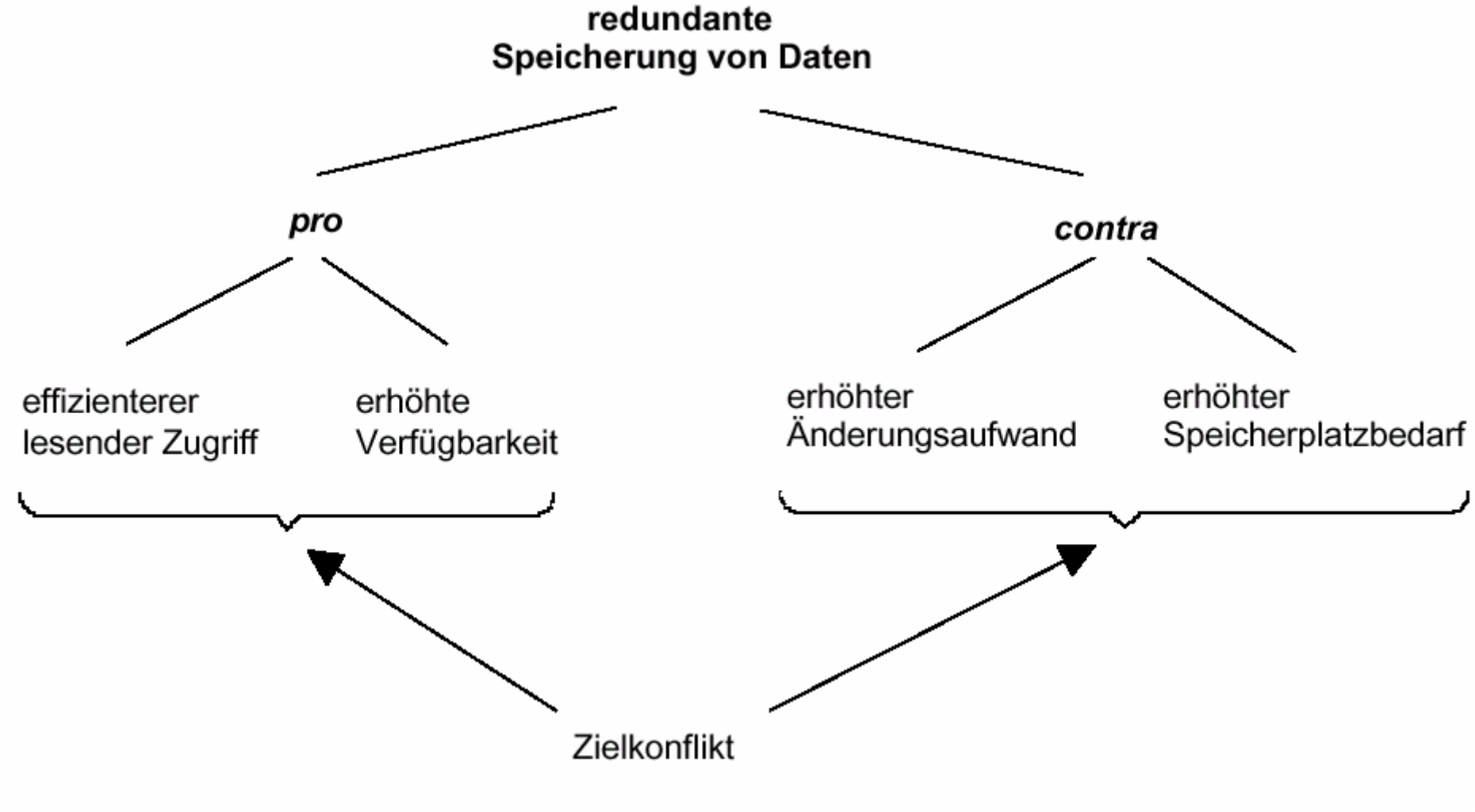
Replizierte Daten

Mehrbenutzersynchronisation,
Fehlertoleranz,
Änderungspropagation,
Anwendungen

Zielkonflikt bei der redundanten Speicherung von Information

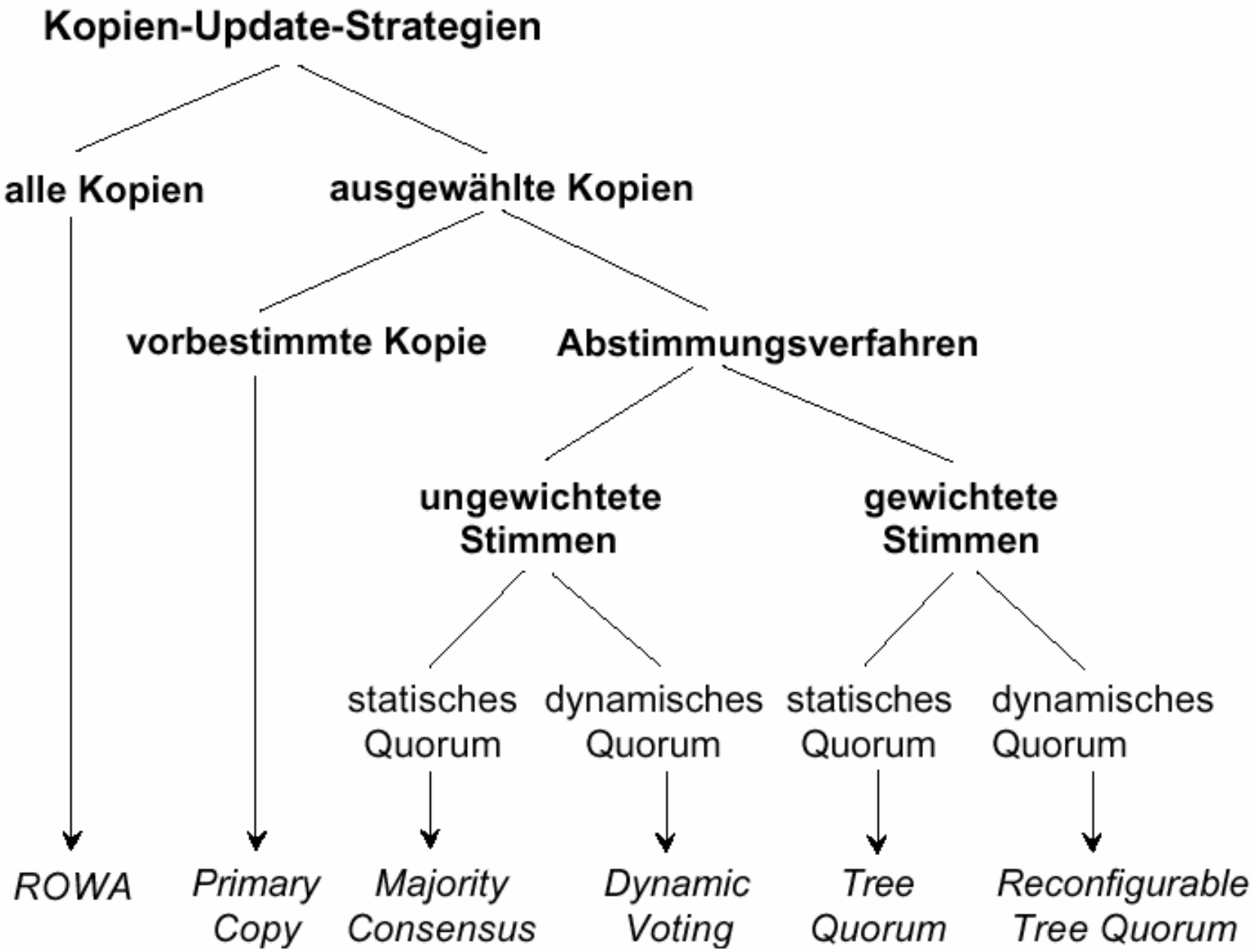


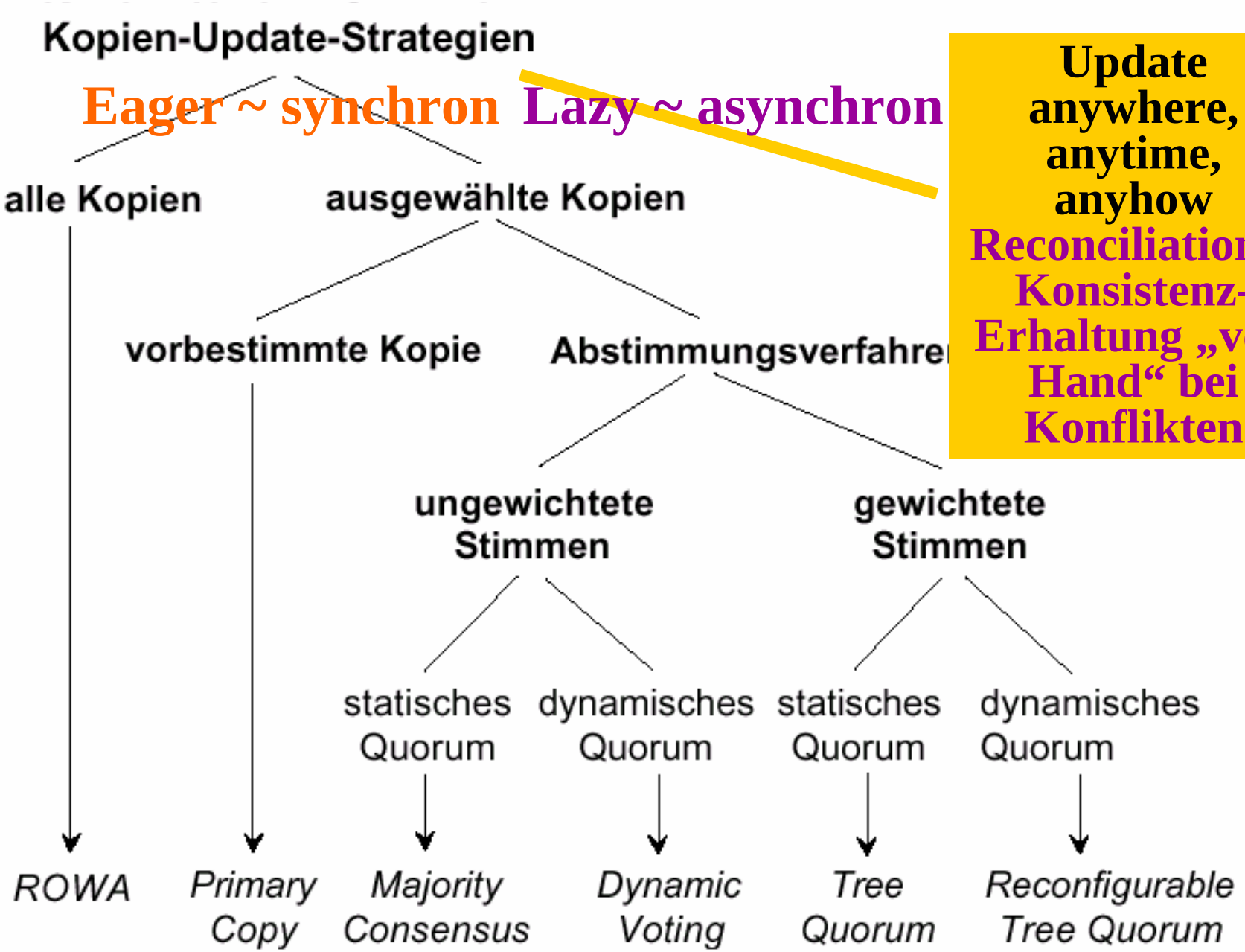
Warum Daten(banken) replizieren?

- Gebe Benutzern lokale Kopien wegen
  - Performance

  - Verfügbarkeit

  - Mobilität (disconnected users)
- ABER … Was passiert wenn sie Updates durchführen?
- Updates müssen zu anderen Kopien propagiert werden


Kopien-Update-Strategien
alle Kopien
ausgewählte Kopien
vorbestimmte Kopie
Abstimmungsverfahren
ungewichtete Stimmen
gewichtete Stimmen
statisches Quorum
dynamisches Quorum
statisches Quorum
dynamisches Quorum
ROWA
Primary Copy
Majority Consensus
Dynamic Voting
Tree Quorum
Reconfigurable Tree Quorum



Update anywhere, anytime, anyhow
Reconciliation ~ Konsistenz-Erhaltung „von Hand“ bei Konflikten

Anforderungen an die Replikationsverfahren

- Aktualisierung der Kopien durch das vDBMS
- vollständige Transparenz der Replikation für die Anwendungsprogramme (bzw. Anwendungsprogrammierer)

```
update Flüge
    set Preis = Preis * 1.3
where Ziel = `Hannover´
```

- Gleiches Verhalten wie nicht-replizierte Datenbank
  - **1-Kopie-Serialisierbarkeit** (das vDBMS soll sich bzgl. Korrektheit/Serialisierbarkeit nicht anders verhalten als ein vDBMS ohne Kopien)
  - wechselseitige Konsistenz der Kopien dazu nötig
  - ein Schedule heißt dann und nur dann *1-Kopie-serialisierbar* wenn es eine serielle Ausführung der TAs gibt, welche angewandt auf denselben Ausgangszustand diegleiche Ausgabe sowie denselben Endzustand erzeugt.

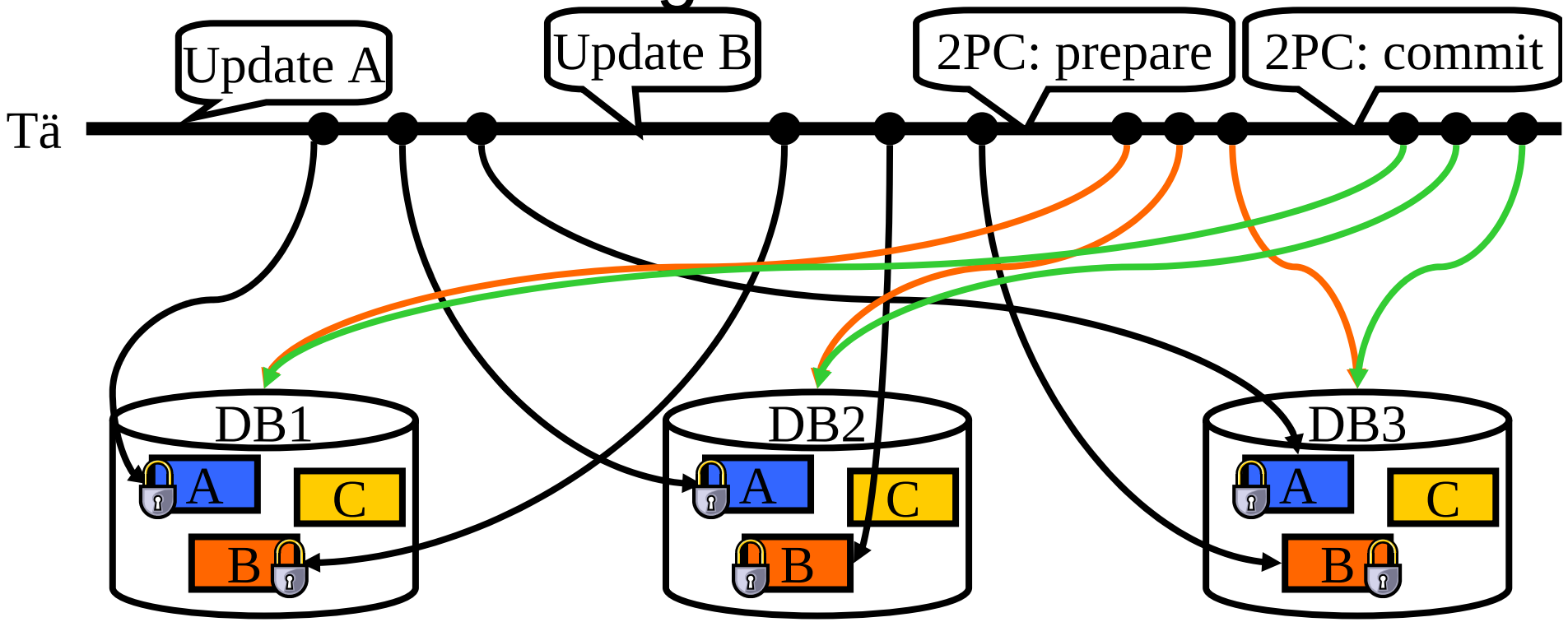
ROWA: Read One / Write All

- Eine logische Leseoperation wird auf einer einzigen beliebigen Kopie durchgeführt
  - die am nächsten im Netzwerk verfügbare Kopie, beispielsweise
- Eine logische Schreiboperation wird zu physischen Schreiboperationen auf allen Kopien des replizierten Datums
  - synchrones Schreiben aller Kopien in derselben Transaktion
  - beim 2PL-Verfahren werden in derselben TA alle Kopien des Datums gesperrt
  - es wird das 2PC-Verfahren angewendet um die Schreiboperation atomar auf allen Kopien „zu commit-ten“
- sehr einfach zu implementieren, da das Verfahren nahtlos in das 2PL-Synchronisationsprotokoll und das 2PC-Commit-Protokoll „passt“

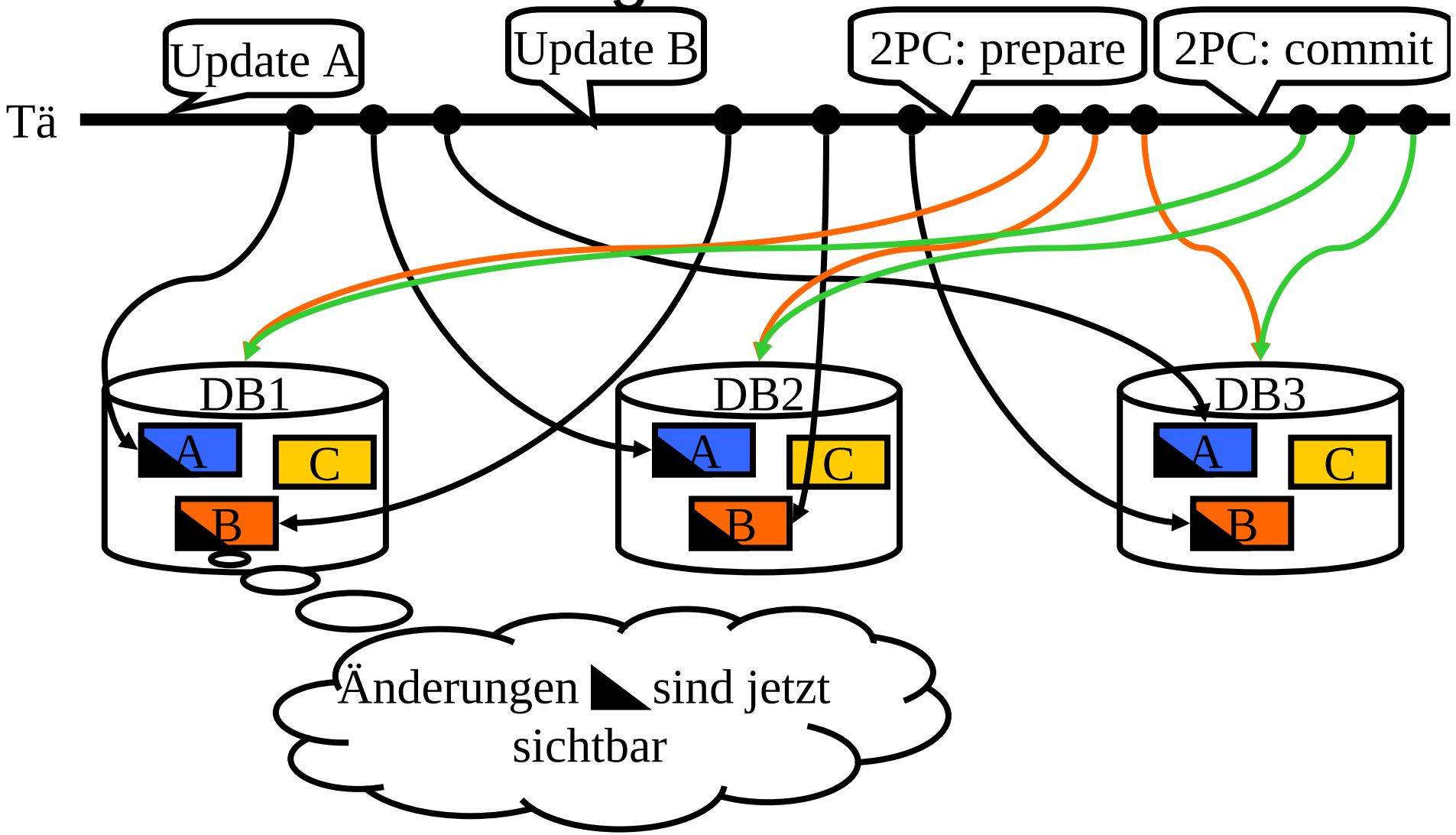
ROWA-Vorteile/Nachteile

- ☺ Einfach zu implementieren
- ☺ alle Kopien stets auf demgleichen aktuellen Stand
- ☺ effizientes lokales Lesen
  - ☺ Ausfall einer/mehrerer Knoten ist für Lesetransaktionen leicht zu verkraften (Ausweichen auf andere Knoten)
- ☺ Konform mit 2PL und 2PC
- ☹ starke Abhängigkeit einer Äanderungs-TA von der Verfügbarkeite **aller kopienhaltenden** DB-Knoten
- ☹ daher relativ geringe Verfügbarkeit des Gesamtsystems
- ☹ längere Laufzeiten von Änderungsfaktoren
  - ☹ bei Replikationsgrad N werden Äanderungs-TA N-mal so lang
  - ☹ daraus resultiert sehr hohe Deadlock-Rate

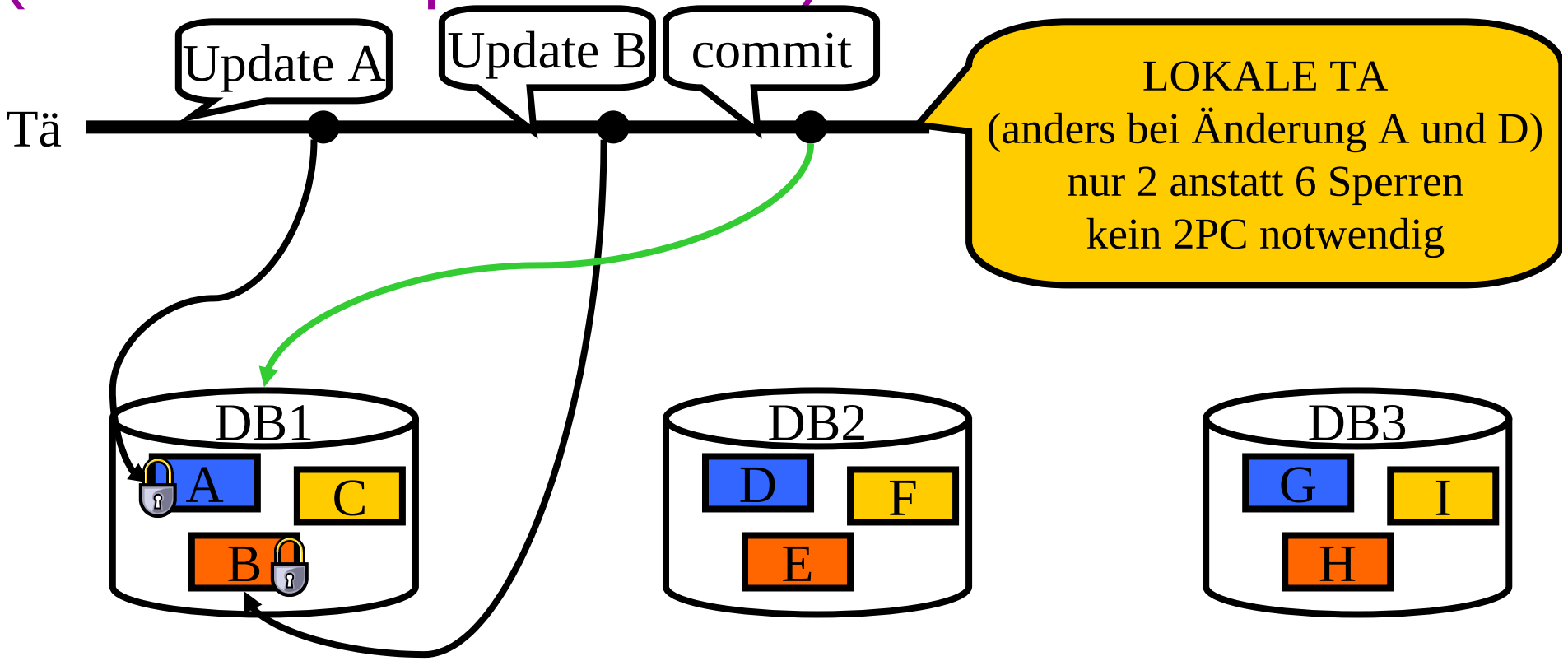
ROWA-Änderungstransaktionen

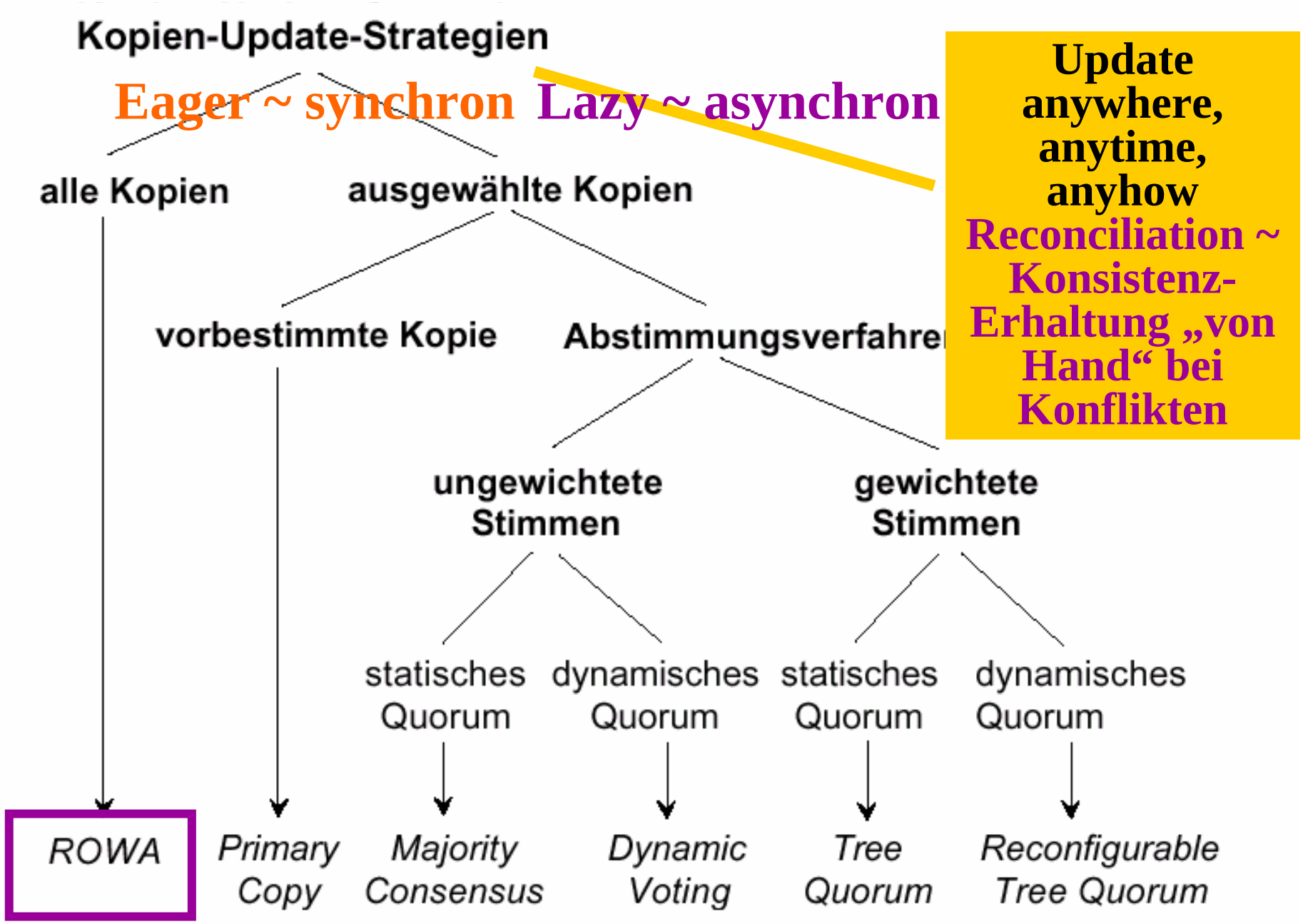


ROWA-Änderungstransaktionen



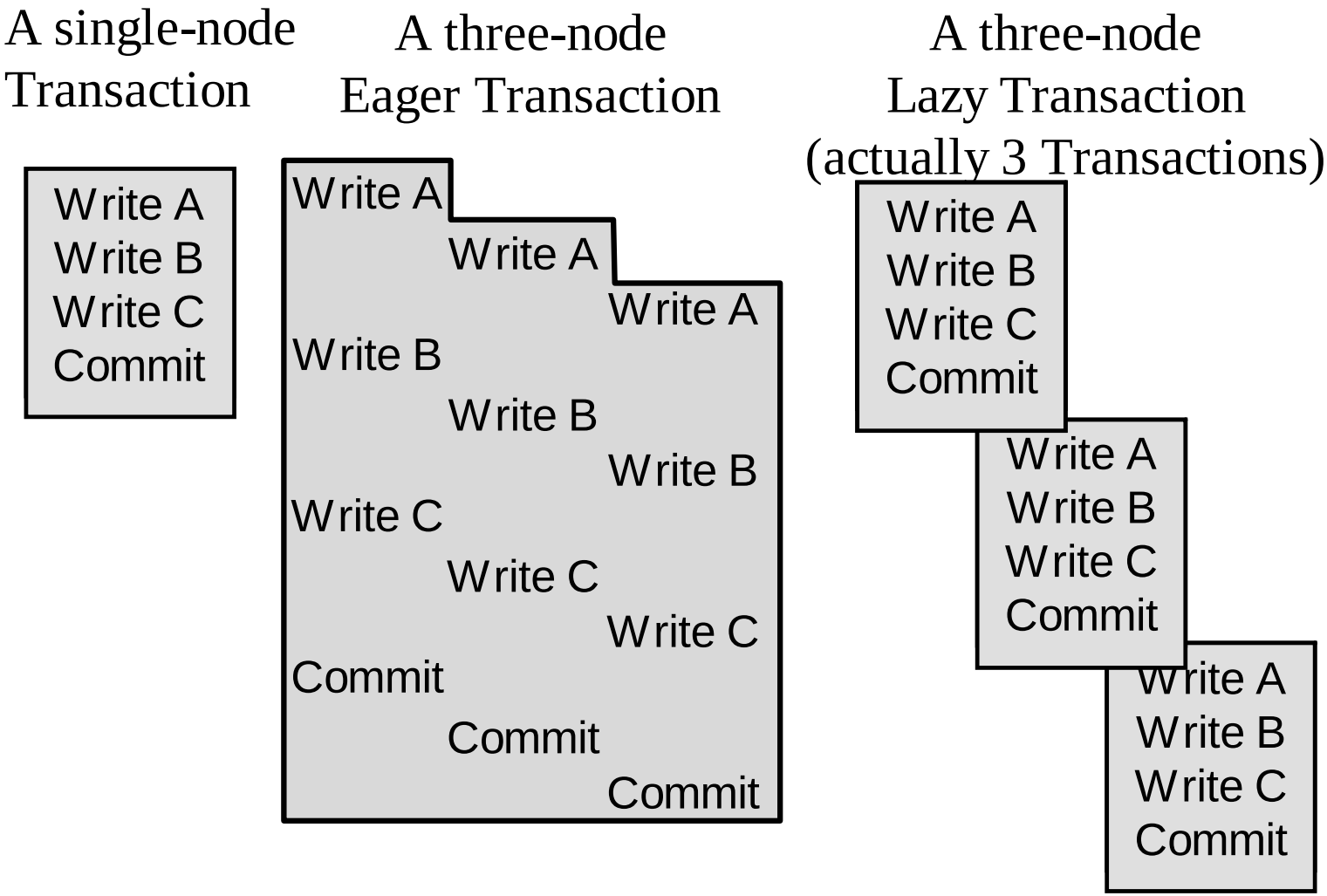
ROWA-Änderungstransaktionen (Keine Replikation)



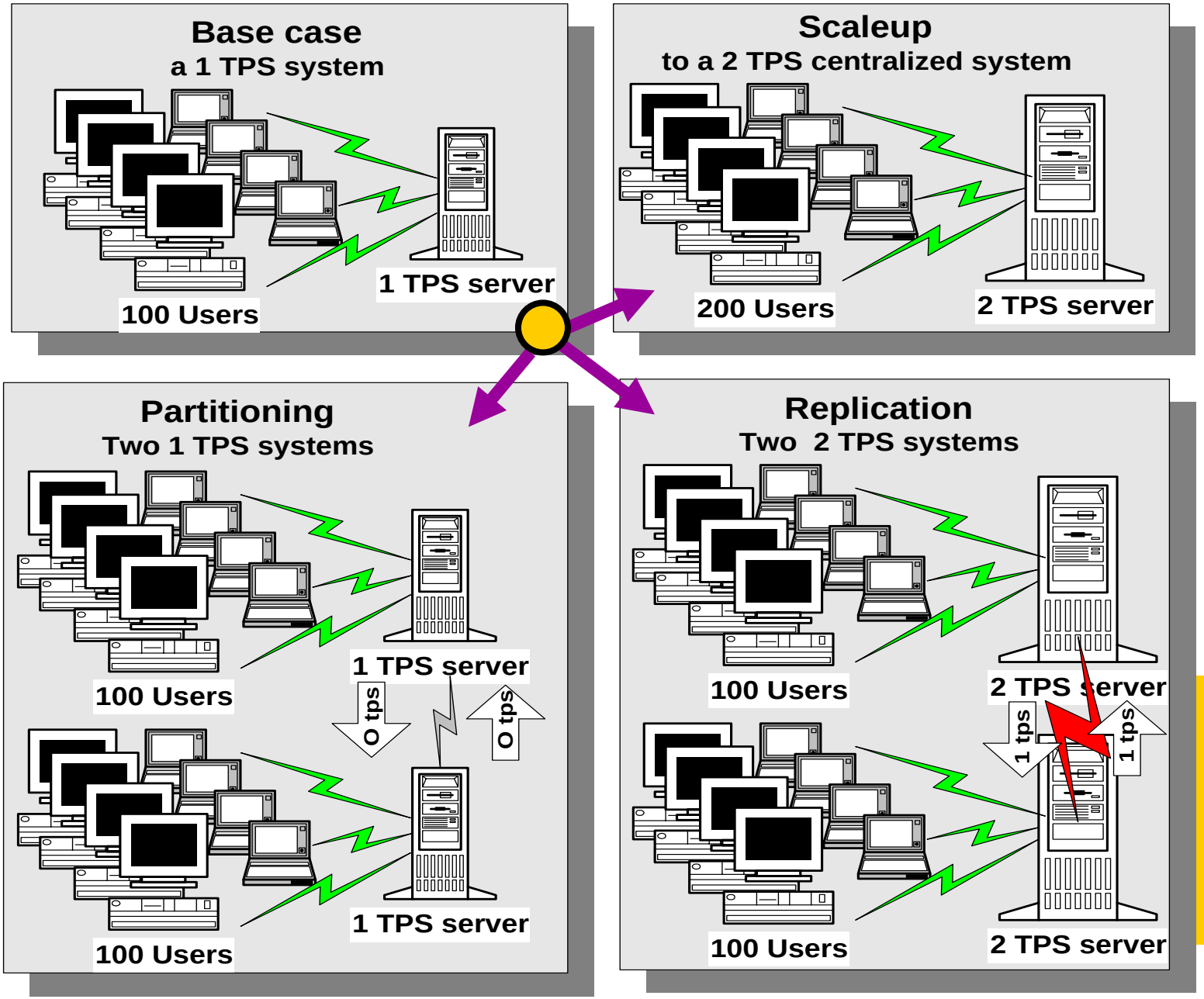

Kopien-Update-Strategien
Eager ~ synchron
Lazy ~ asynchron
Update anywhere, anytime, anyhow
Reconciliation ~ Konsistenz-Erhaltung „von Hand“ bei Konflikten
alle Kopien
ausgewählte Kopien
vorbestimmte Kopie
Abstimmungsverfahre
ungewichtete Stimmen
gewichtete Stimmen
statisches Quorum
dynamisches Quorum
statisches Quorum
dynamisches Quorum
ROWA
Primary Copy
Majority Consensus
Dynamic Voting
Tree Quorum
Reconfigurable Tree Quorum

Gegenüberstellung: Transaktion ohne Replikate, mit sofortiger synchroner (eager) Replikatänderung, mit asynchroner (lazy) Replikatänderung

[nach Gray et al., The Dangers of Replication ... , SIGMOD 1996]



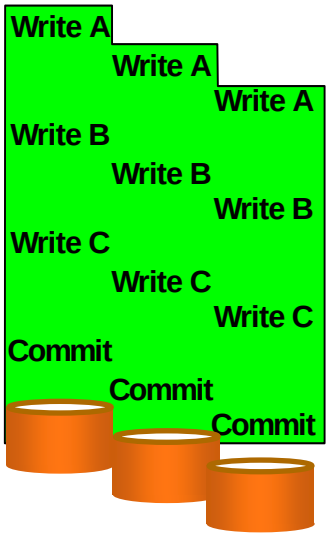
Scaleup, Replikation, Partitionierung

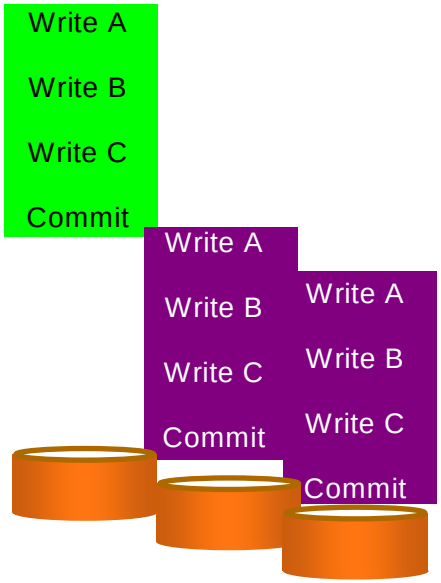


N Knoten
~
N-facher Durchsatz
➔
N^2 mehr Arbeit

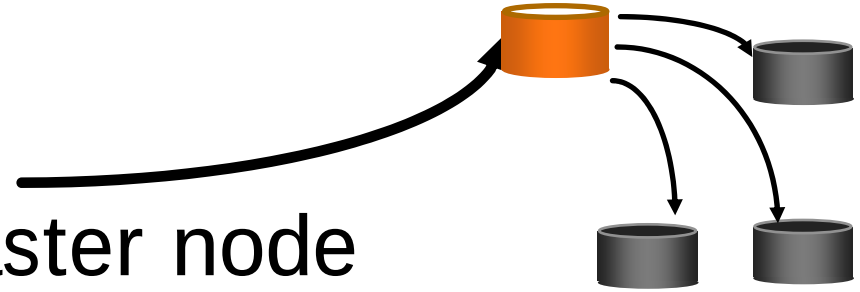
Propagation Strategies

- Eager: Send update right away
  - (part of same transaction)
  - *N* times larger transactions
- Lazy: Send update asynchronously
  - separate transaction
  - *N* times more transactions
- Either way
  - *N* times more updates per second per node
  - N^2 times more work overall

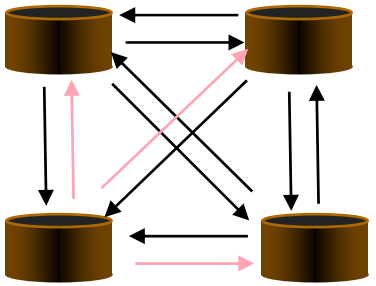




Update Control Strategies



- Master
  - Each object has a master node
  - All updates start with the master
  - Broadcast to the subscribers
- Group
  - Object can be updated by anyone
  - Update broadcast to all others



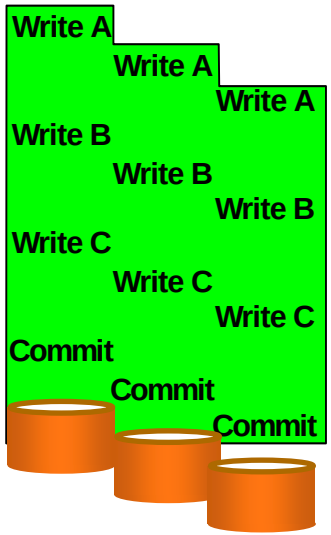
- Everyone *wants* Lazy Group:
  - update anywhere, anytime, anyway

Anecdotal Evidence

- Update Anywhere systems are attractive
- Products offer the feature
- It demos well
- But when it scales up
  - Reconciliations start to cascade
  - Database drifts “out of sync” *(System Delusion)*
- What’s going on?

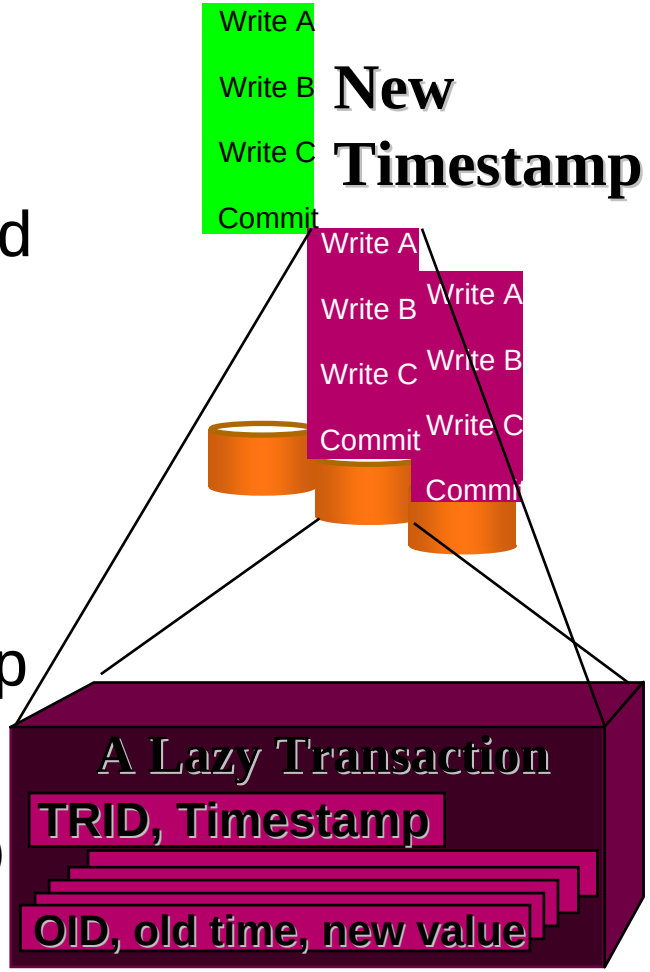
Eager Transactions are FAT

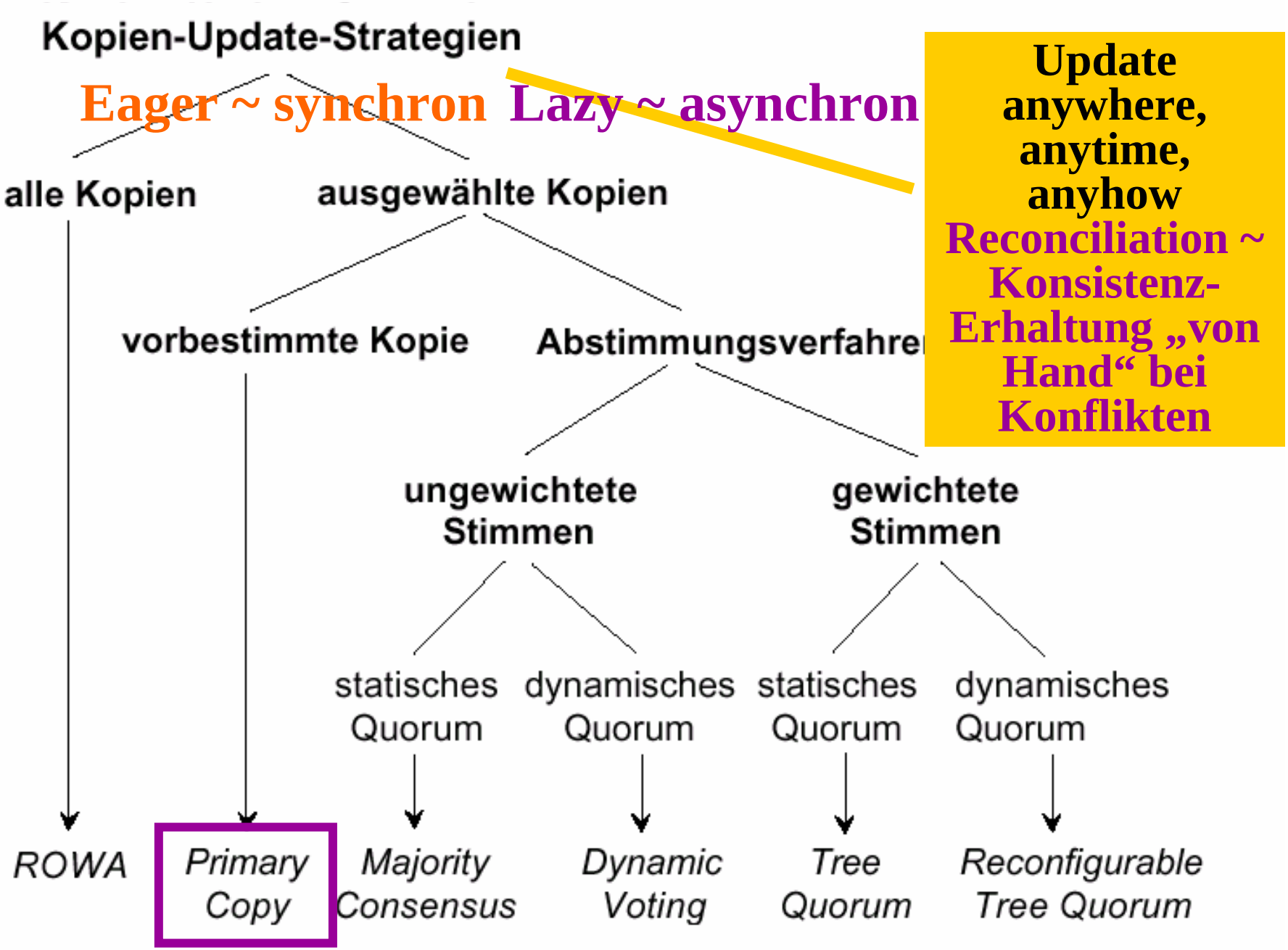
- If *N* nodes, eager transaction is *N*x bigger
  - Takes *N*x longer
  - 10x nodes, 1,000x deadlocks
  - (derivation in paper)
- Master slightly better than group
- Good news:
  - Eager transactions only deadlock
  - No need for reconciliation



Lazy Master & Group

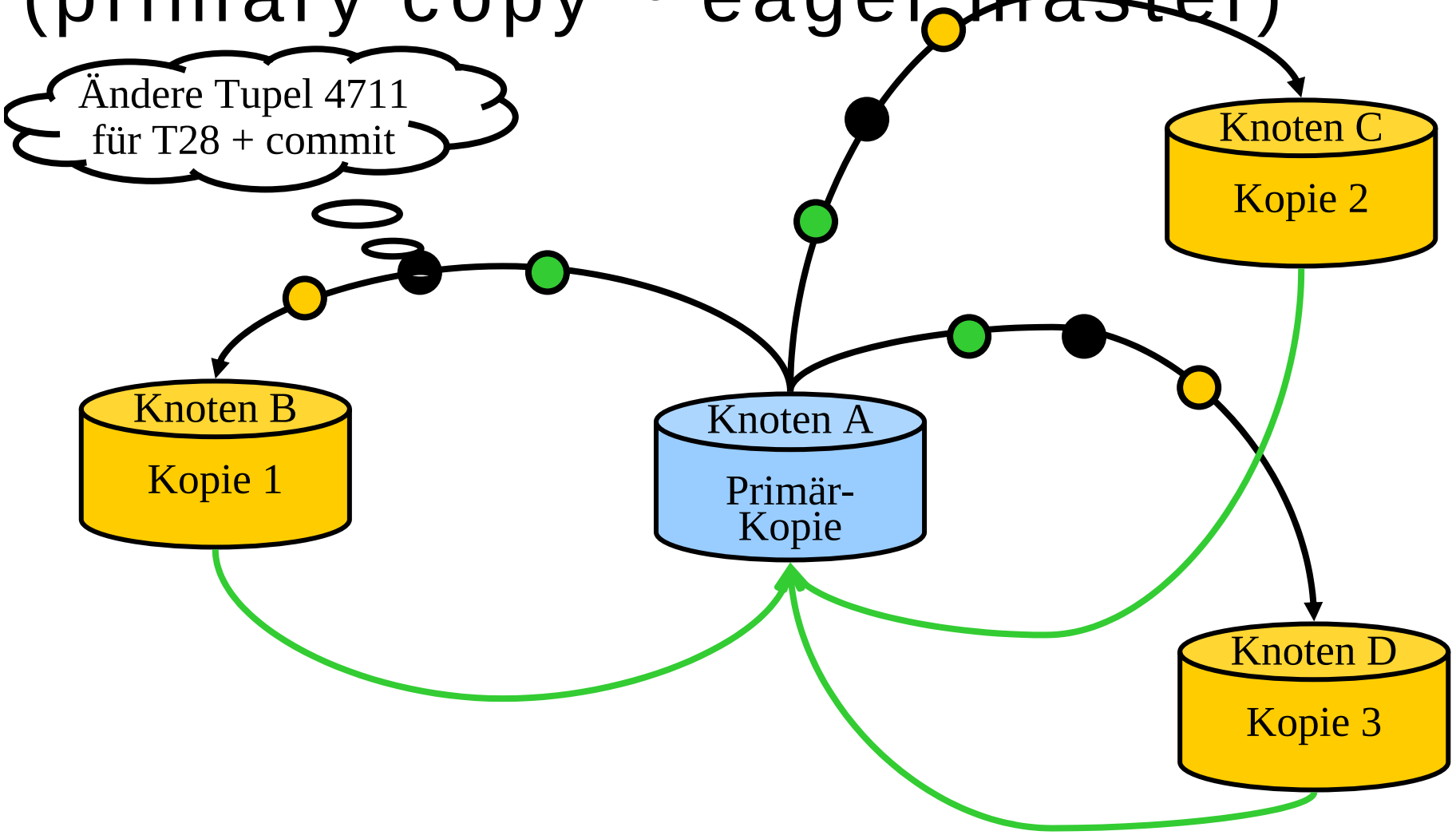
- Use optimistic concurrency control
  - Keep transaction timestamp with record
  - Updates carry old+new timestamp
  - If record has old timestamp
    - set value to new value
    - set timestamp to new timestamp
  - If record does not match old timestamp
    - reject lazy transaction
  - Not SNAPSHOT isolation (stale reads)
- Reconciliation:
  - Some nodes are updated
  - Some nodes are “being reconciled”



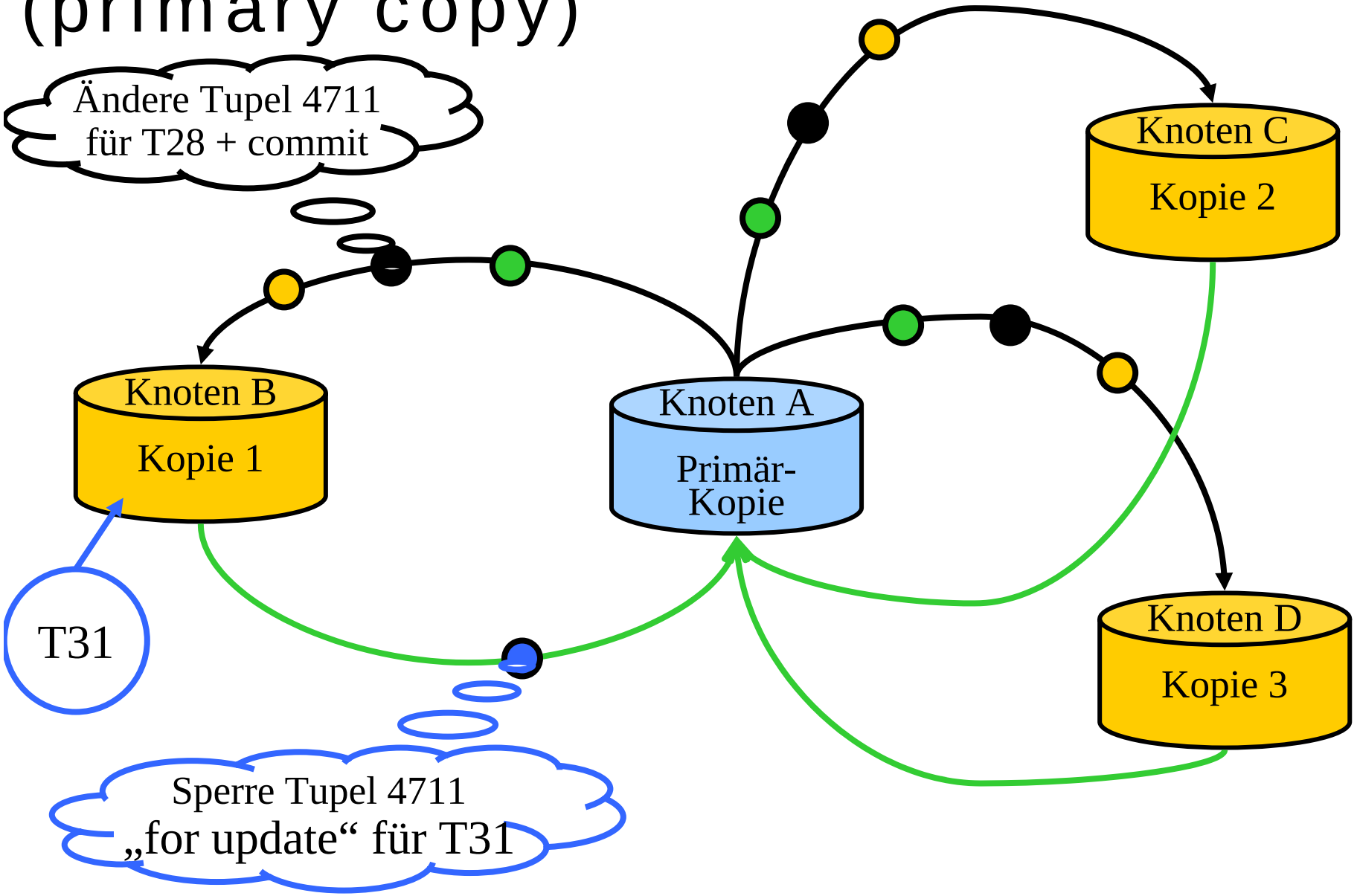


Update anywhere, anytime, anyhow
Reconciliation ~ Konsistenz-Erhaltung „von Hand“ bei Konflikten

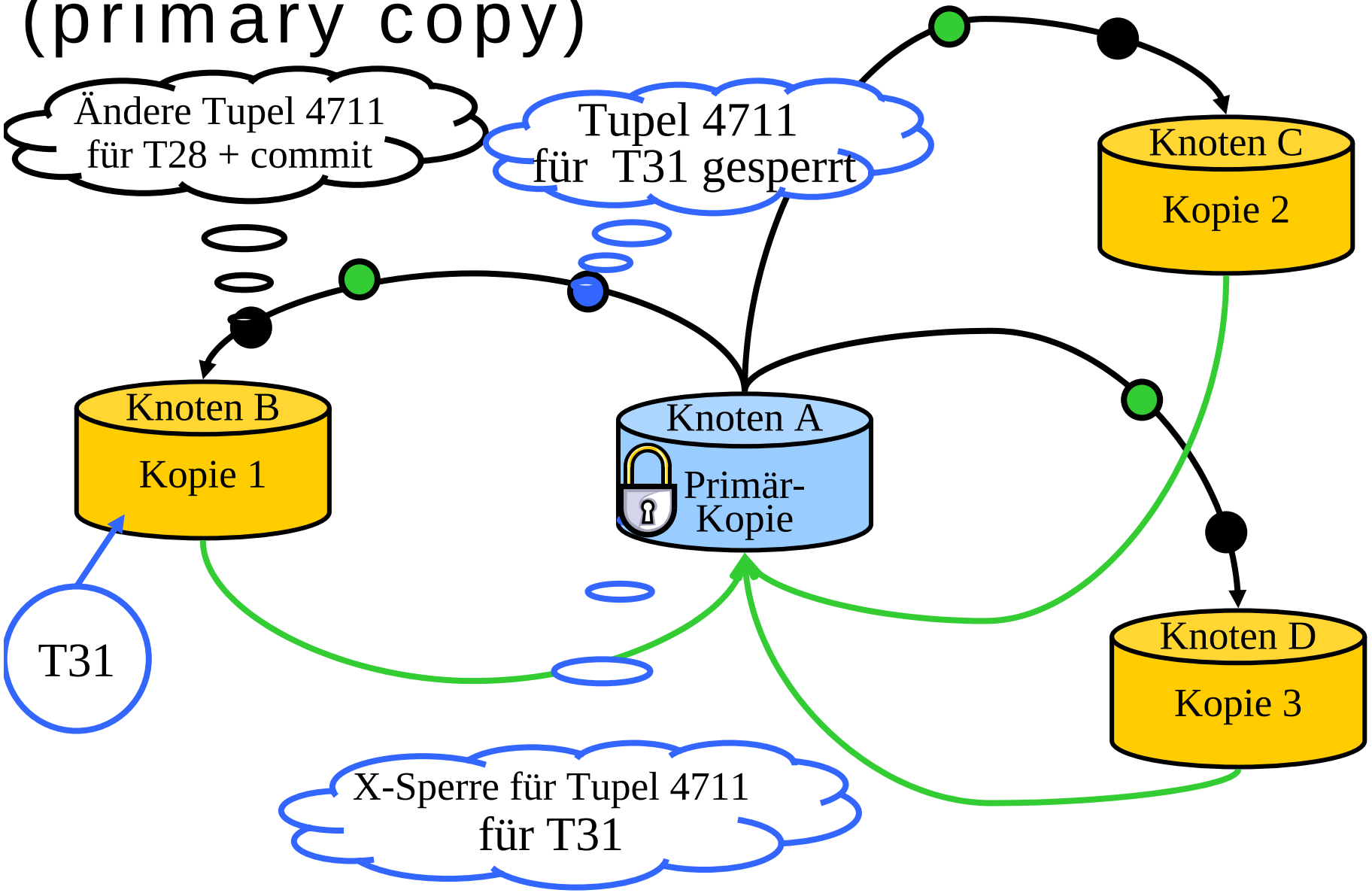
Primärkopien-Verfahren (primary copy ~ eager master)



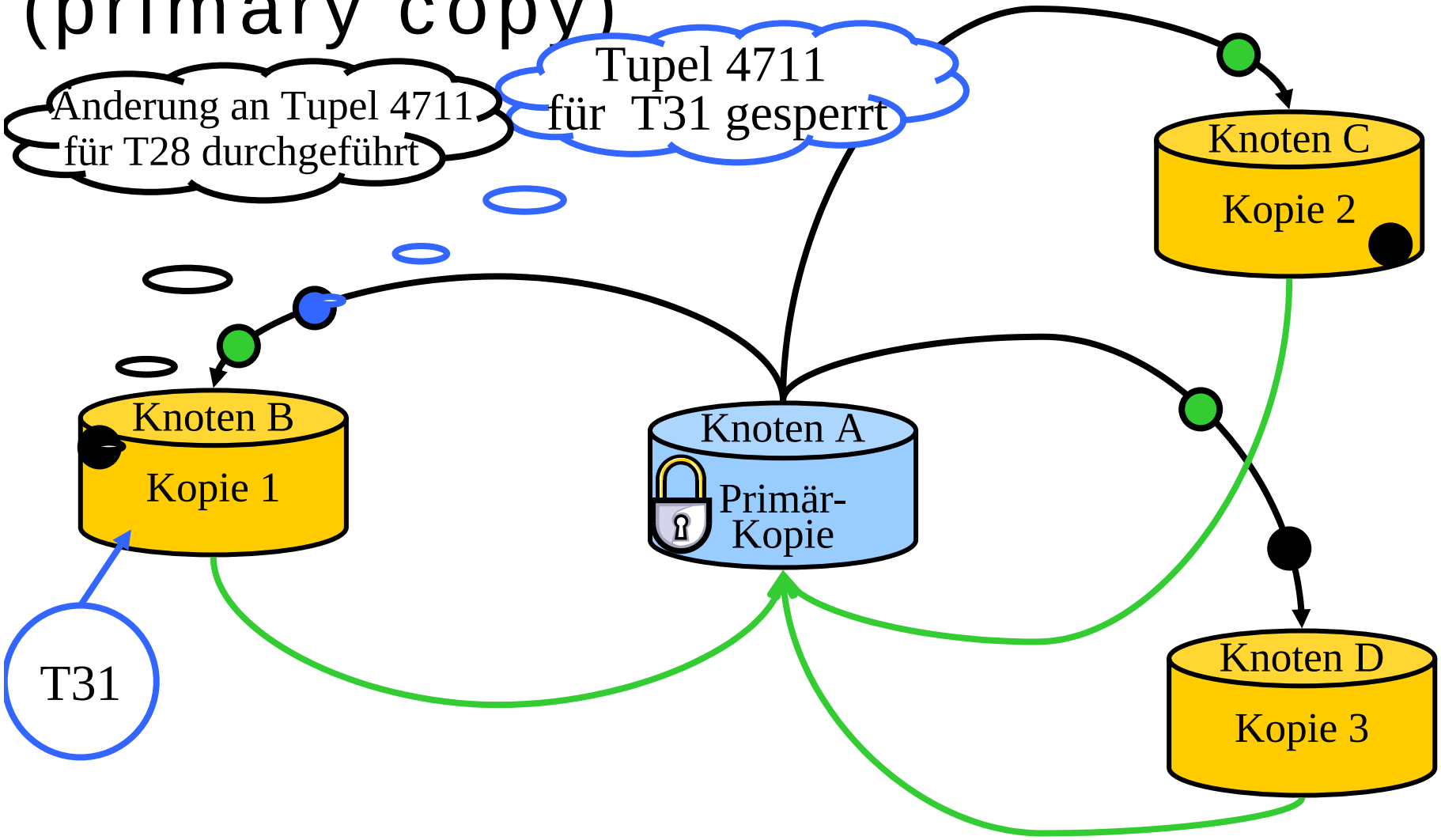
Primärkopien-Verfahren (primary copy)



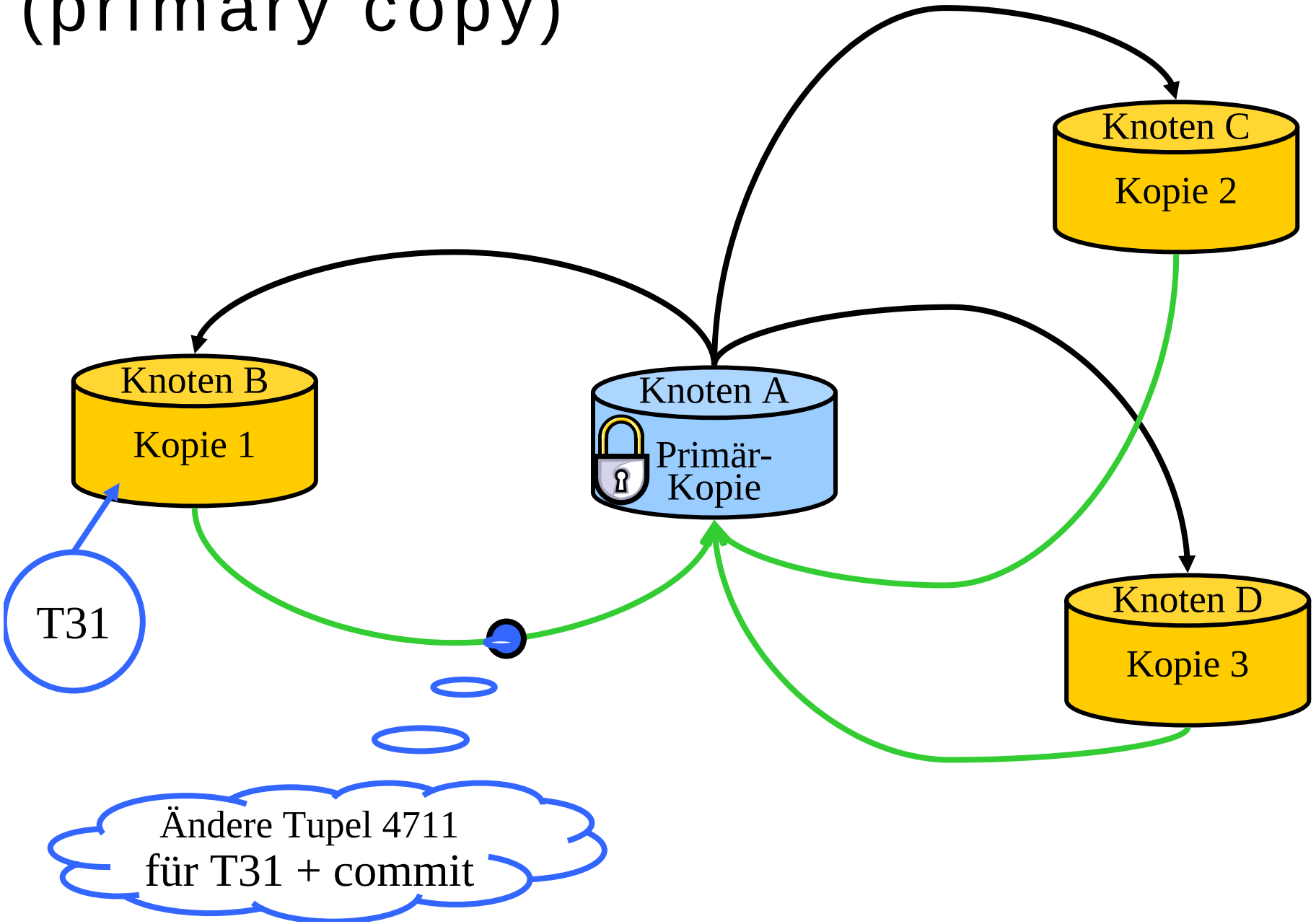
Primärkopien-Verfahren (primary copy)



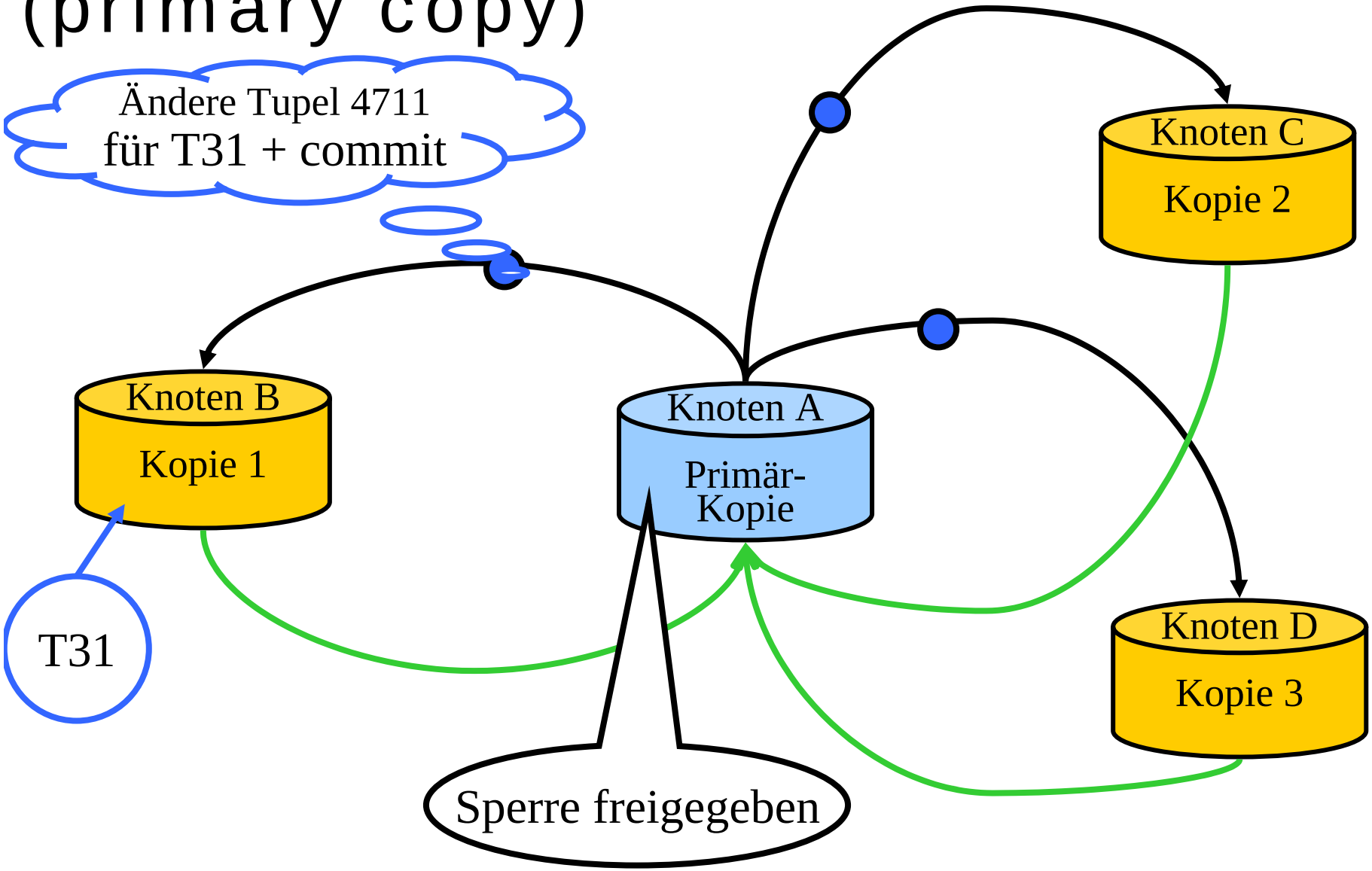
Primärkopien-Verfahren (primary copy)



Primärkopien-Verfahren (primary copy)

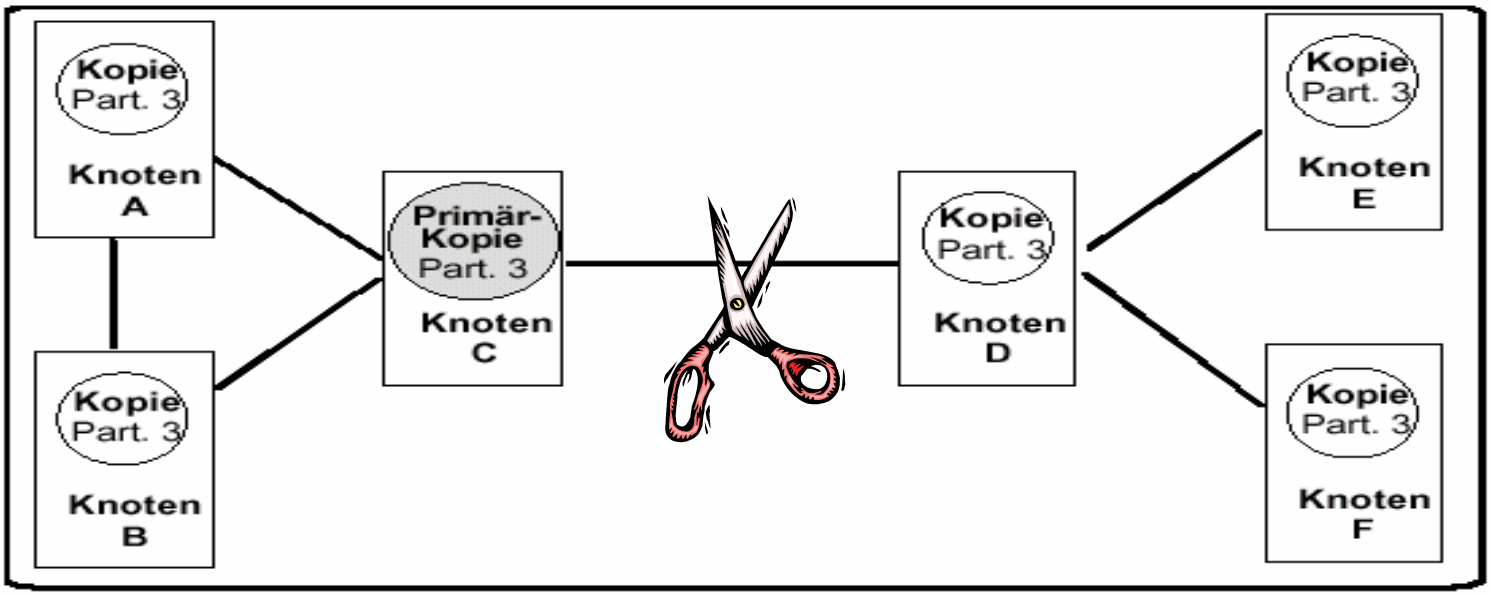


Primärkopien-Verfahren (primary copy)

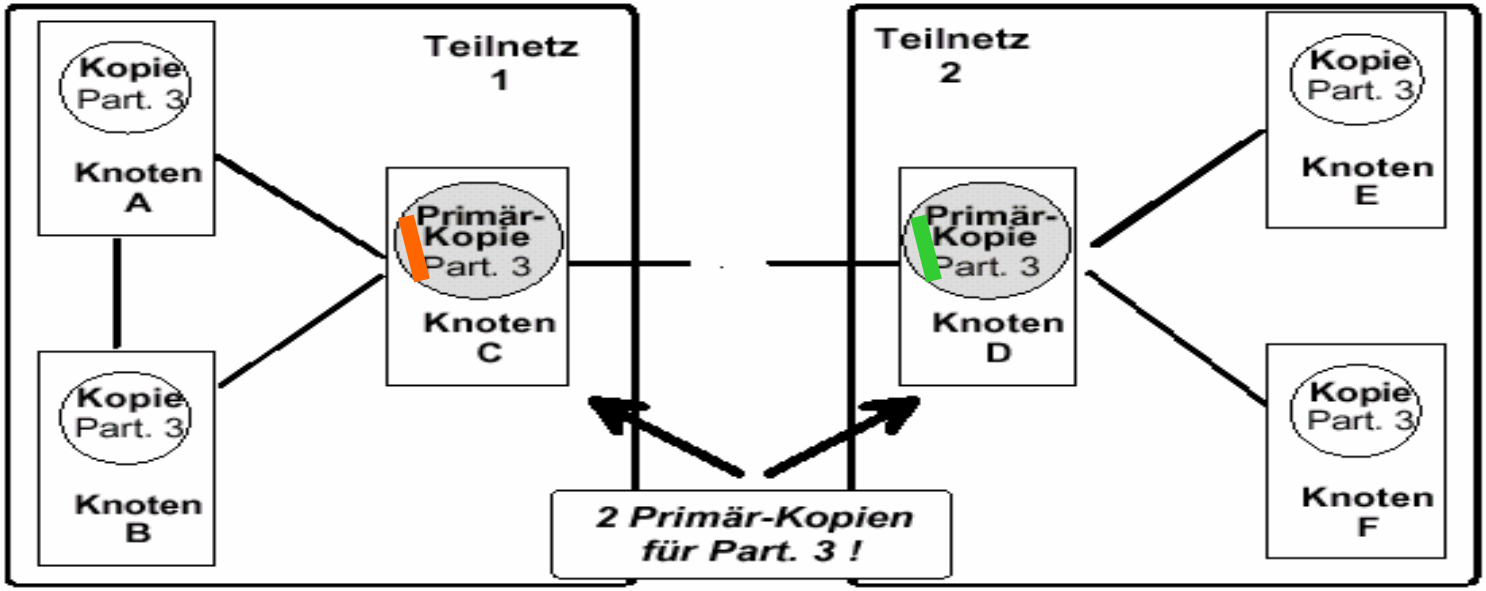


Primärkopien-Verfahren: Diskussion

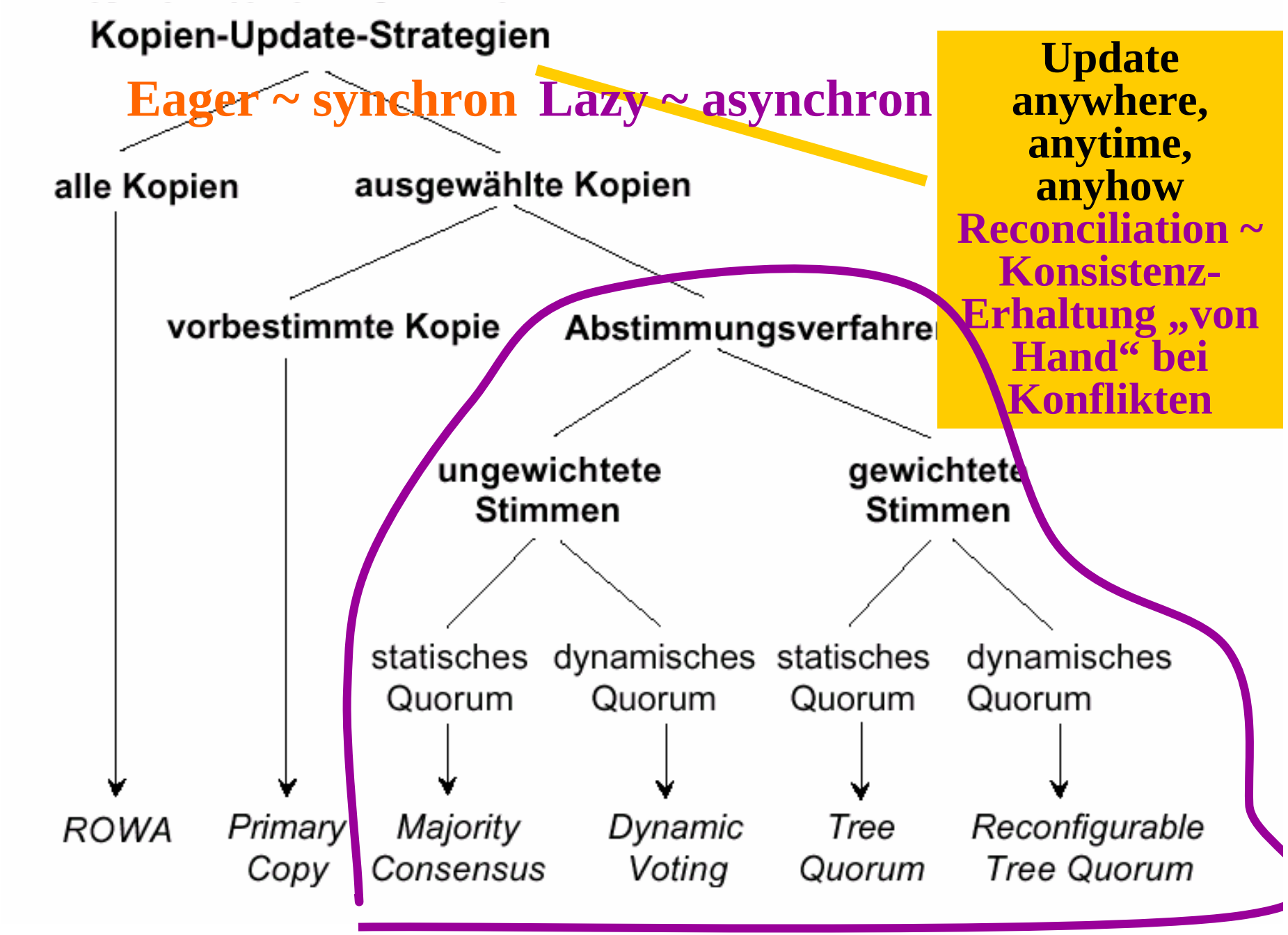
- Primärkopie-Server kann leicht zum Flaschenhals im System werden
- In der Praxis wird das Primärkopien-Verfahren durchaus angewendet
- In der Regel wird man nur X-Sperren bei der Primärkopie anfordern
  - also werden möglicherweise veraltete/inkonsistente Daten gelesen
  - Alternative: Anfordern einer S-Sperre bei der Primärkopie $\Rightarrow$ sehr aufwendig und macht Vorteil des lokalen Lesens zunichte
- Was passiert bei Ausfall des Primärkopie-Servers
  - Auswahl eines alternativen Primärkopie-Servers
    - z.B. derjenige mit der größten Kennung
    - Gefahr: mehrere Primärkopien in unterschiedlichen Netzpartitionen



a) Situation vor der Netz-Partitionierung



b) Situation nach der Netz-Partitionierung und
Bestimmung einer Primär-Kopie in Teilnetz 2

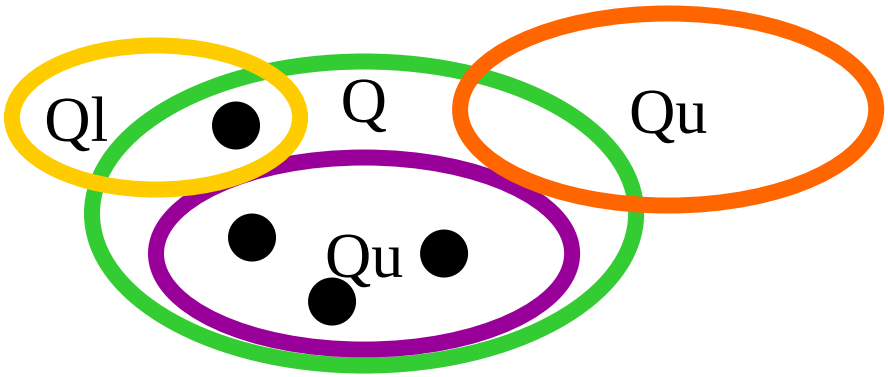

Kopien-Update-Strategien
Eager ~ synchron
Lazy ~ asynchron
alle Kopien
ausgewählte Kopien
vorbestimmte Kopie
Abstimmungsverfahre
ungewichtete Stimmen
gewichtete Stimmen
statisches Quorum
dynamisches Quorum
statisches Quorum
dynamisches Quorum
ROWA
Primary Copy
Majority Consensus
Dynamic Voting
Tree Quorum
Reconfigurable Tree Quorum
Update anywhere, anytime, anyhow
Reconciliation ~ Konsistenz-Erhaltung „von Hand“ bei Konflikten

Abstimmungsverfahren

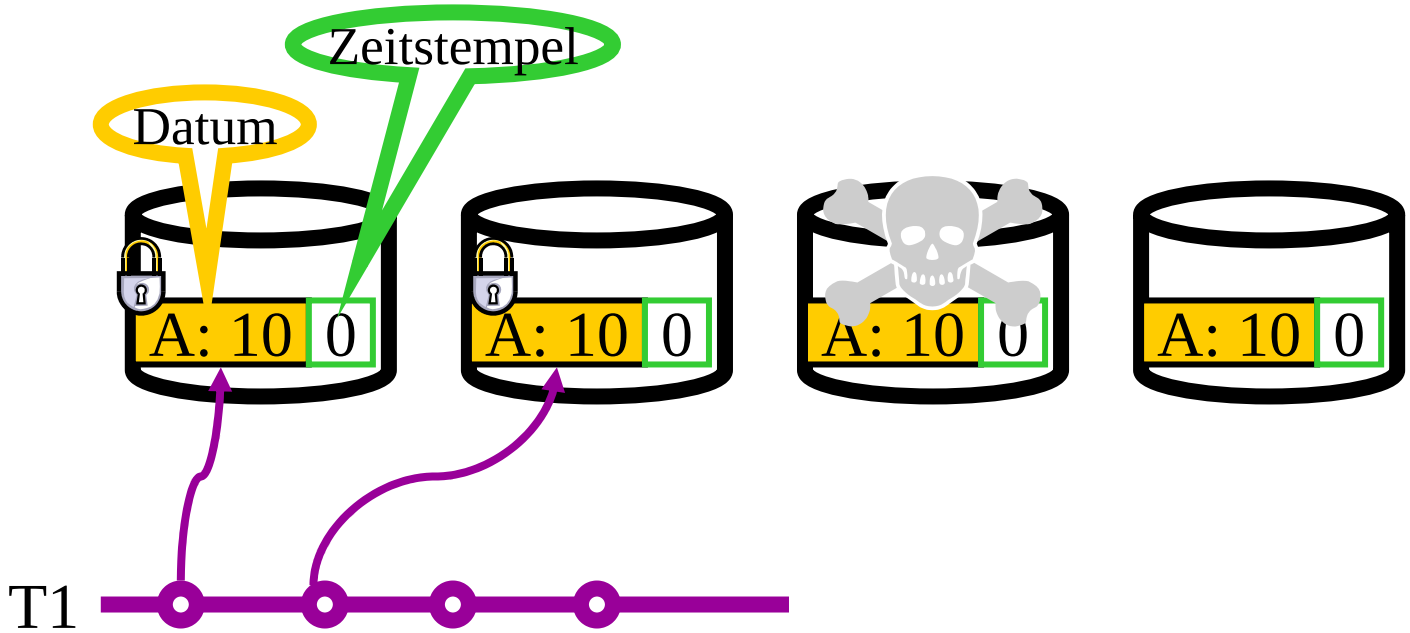
- Prinzip: ein Update auf einer Kopie wird nur dann ausgeführt wenn die TA in der Lage ist, die Mehrheit von Kopien dafür zu gewinnen (also zB geeignet zu sperren)
- Unterschiede in den Verfahren hinsichtlich
  - Gewichtung der individuellen Stimmen
    - one man one vote ~ ungewichtete Stimmen
    - gewichtete Stimmen (zB hoch verfügbare Stationen bekommen mehr Gewicht)
  - Anzahl der Stimmen, die für das Erreichen der Mehrheit erforderlich sind
    - fest vorgegeben (statische Quorum)
    - erst zur Laufzeit der TA bestimmt (dynamisches Quorum)
- Für Lesezugriffe (Lesequorum Ql) und für Schreibzugriffe (Schreibquorum Qu) können unterschiedliche Anzahl von erforderliche Stimmen festgelegt werden

Quoren-Überschneidungsregel

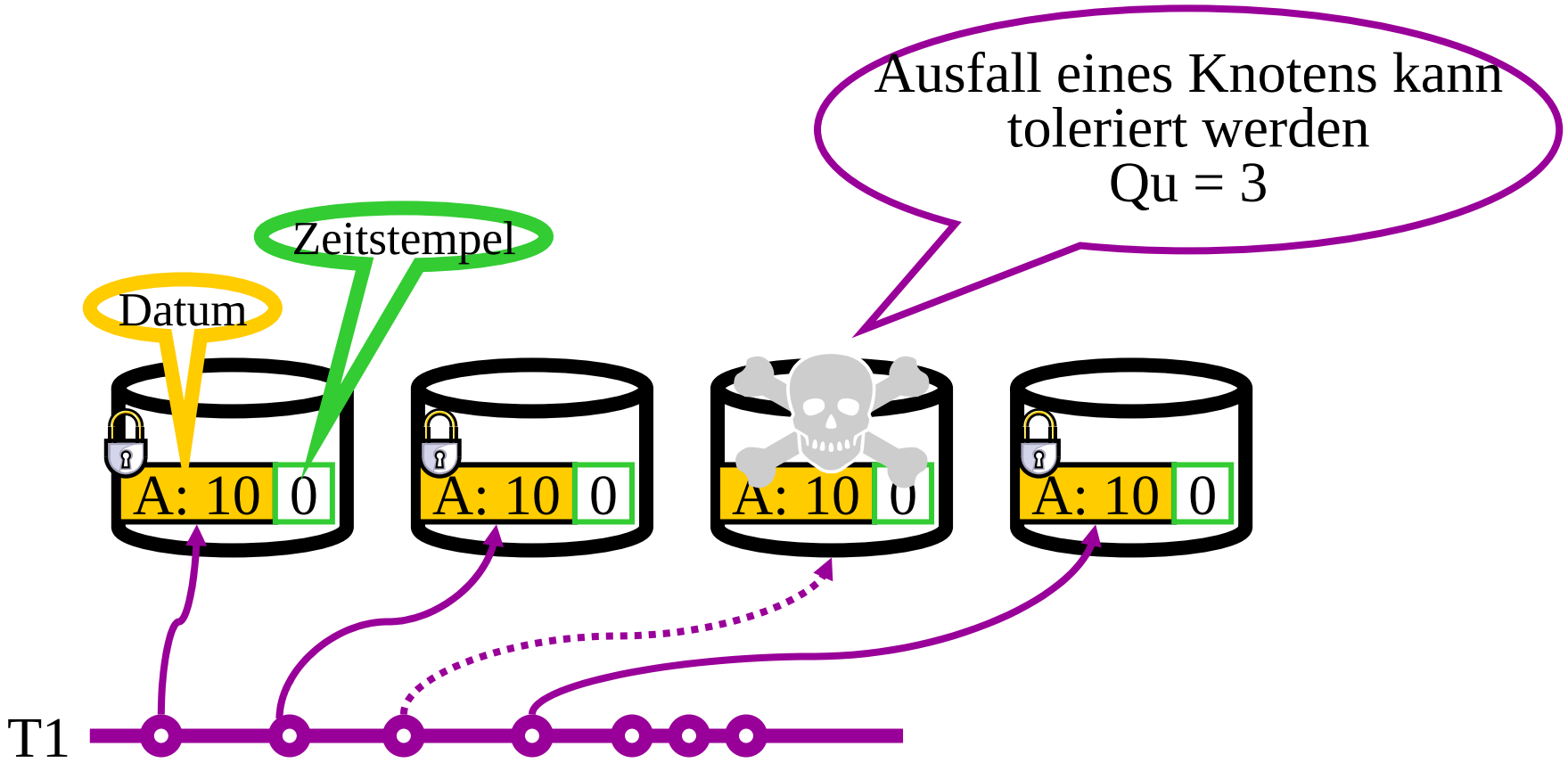
- Lese-TA verlangt S-Sperre
- Schreib-TA verlangt X-Sperre
- Es darf maximal eine Schreibtransaktion gleichzeitig „im System sein“
- Es darf keine Lesetransaktion gleichzeitig mit einer Schreibtransaktion „im System sein“
- Sei Q die Gesamtstimmenzahl
- Ein-Kopie-Serialisierbarkeit ist gewährleistet wenn gilt:
  - Qu + Qu > Q
  - Qu + Ql > Q
- Beispiel:
  - Q = 4
  - Qu = 3
  - Ql = 2



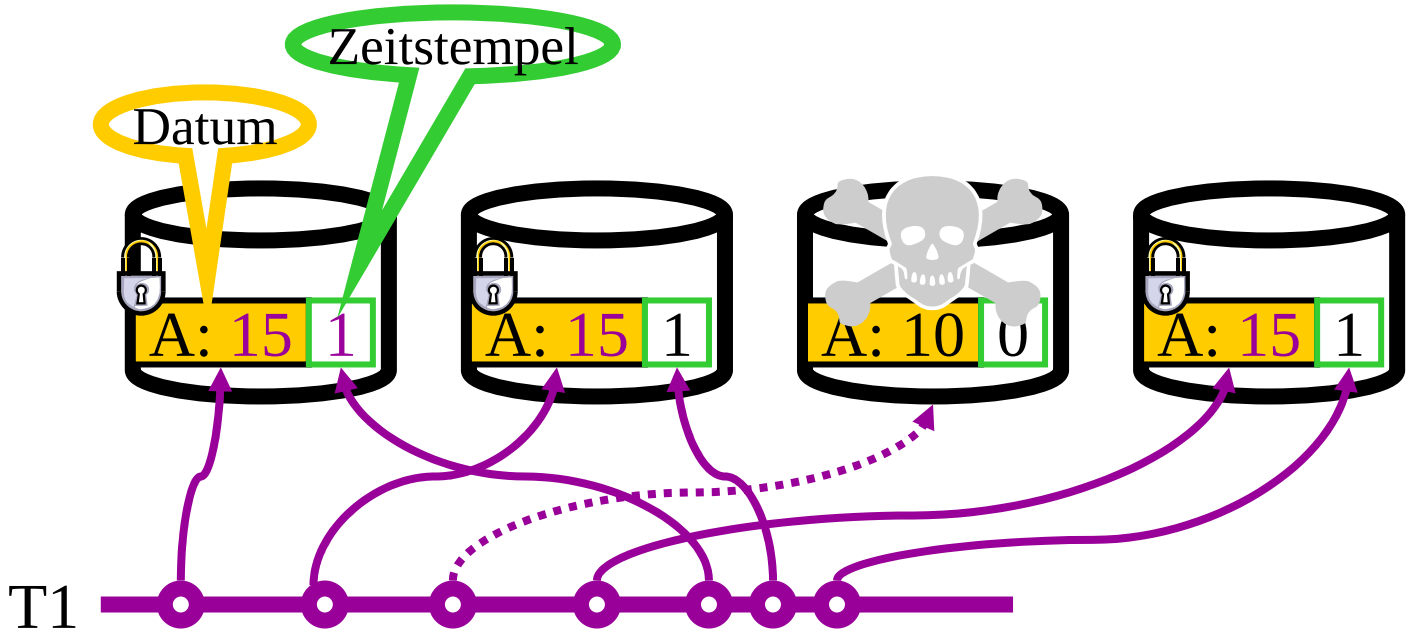
Grundidee des Majority-Consensus-Verfahrens



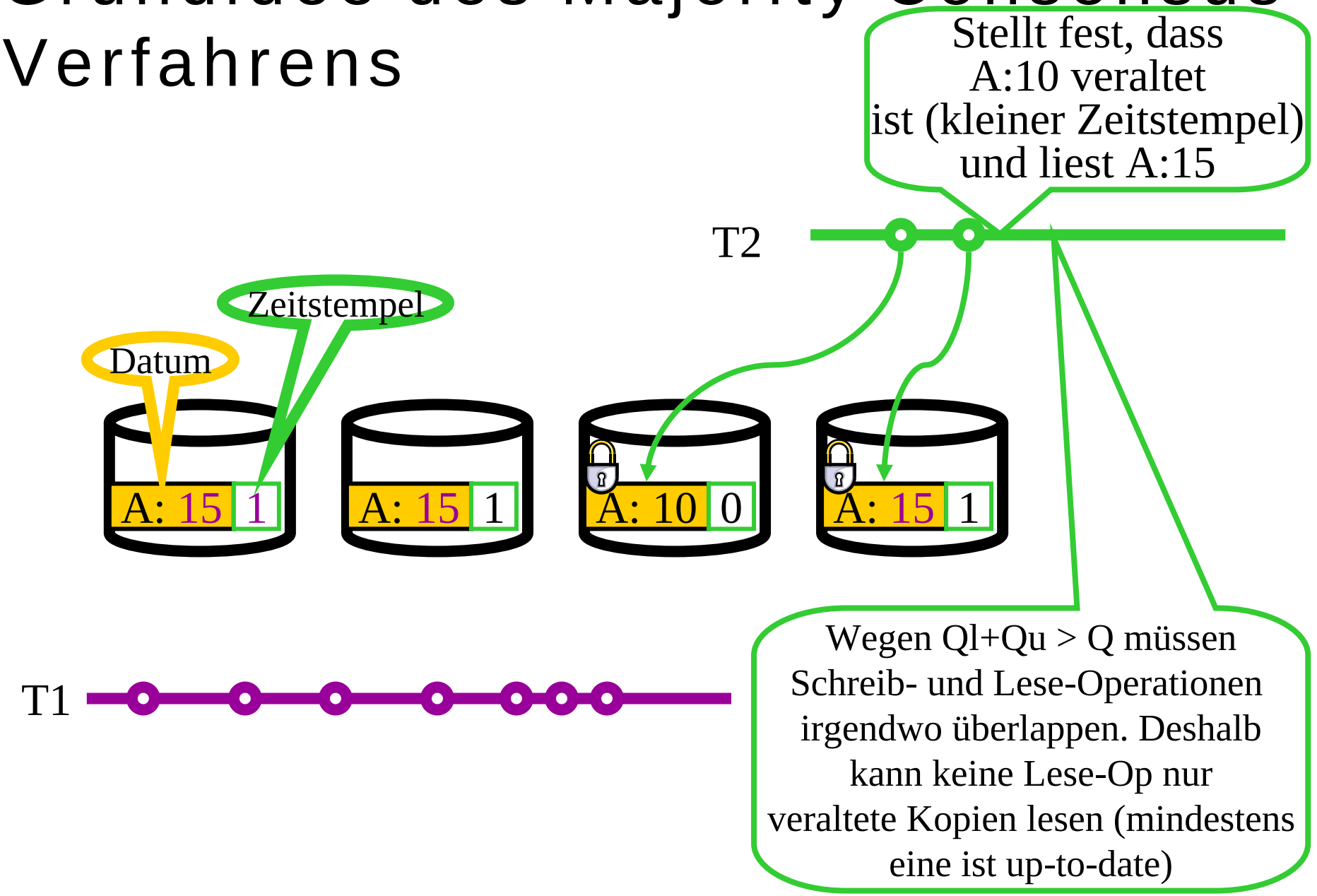
Grundidee des Majority-Consensus-Verfahrens



Grundidee des Majority-Consensus-Verfahrens



Grundidee des Majority-Consensus-Verfahrens



Implementierung des Majority-Consensus-Verfahrens ohne Sperren

- Jedes DB-Objekt ist mit einem Zeitstempel der letzten Änderung versehen
- Ähnlich einem optimistischen Synchronisationsverfahren
- Das Two-Phase-Commit (2PC) ist „irgendwie integriert“
- Jede Änderungs-Transaktion
  - führt alle Änderungen zunächst rein lokal aus (macht diese noch nicht sichtbar)
  - erstellt Liste aller Ein- und Ausgabe-Objekte mit deren Zeitstempeln
  - schickt diese Liste zusammen mit ihrem eigenen Zeitstempel an alle anderen Knoten (Annahme: alles ist überall repliziert)
  - Darf die Änderungen permanent machen (commit-ten) wenn die Mehrheit der Knoten (> N/2) zugestimmt hat

Implementierung des Majority-Consensus-Verfahrens ohne Sperren (2)

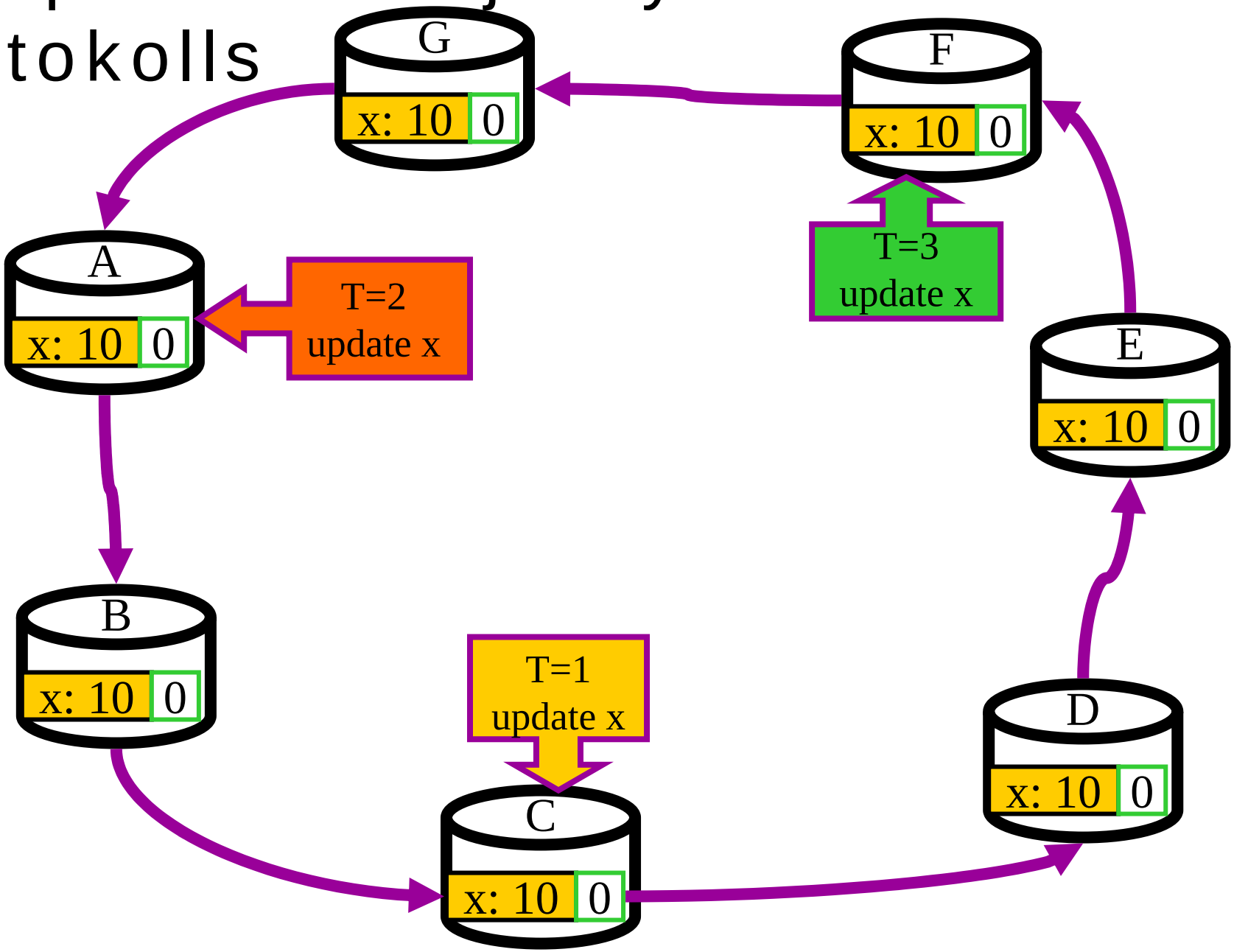
- Die Stationen sind ringförmig angeordnet
- Jeder Knoten (DB-Station) stimmt über eingehende Änderungs-Anträge wie folgt ab und reicht sein Votum zusammen mit den anderen Voten an den nächsten Knoten im Ring weiter:
  - ABGELEHNT wenn einer der übermittelten Zeitstempel veraltet ist
  - OK und markiert den Auftrag als schwebend, wenn alle übermittelten Objekt-Zeitstempel aktuell sind und der Antrag nicht in Konflikt mit einem anderen Antrag steht
  - PASSIERE wenn alle übermittelten Objekt-Zeitstempel aktuell sind aber der Antrag in Konflikt mit einem anderen Antrag höherer Priorität (aktuellerer=höherer TS) steht
    - Falls durch dieses Votum keine Mehrheit mehr erzielbar ist, stimmt er mit ABGELEHNT
  - er verzögert seine Abstimmung wenn der Antrag in Konflikt mit einem niedriger priorisierten Antrag in Konflik steht oder wenn einer der übermittelten Zeitstempel höher ist als der des korrespondierenden lokalen Objekts ist.

Majority/Consensus ohne Sperren (3)

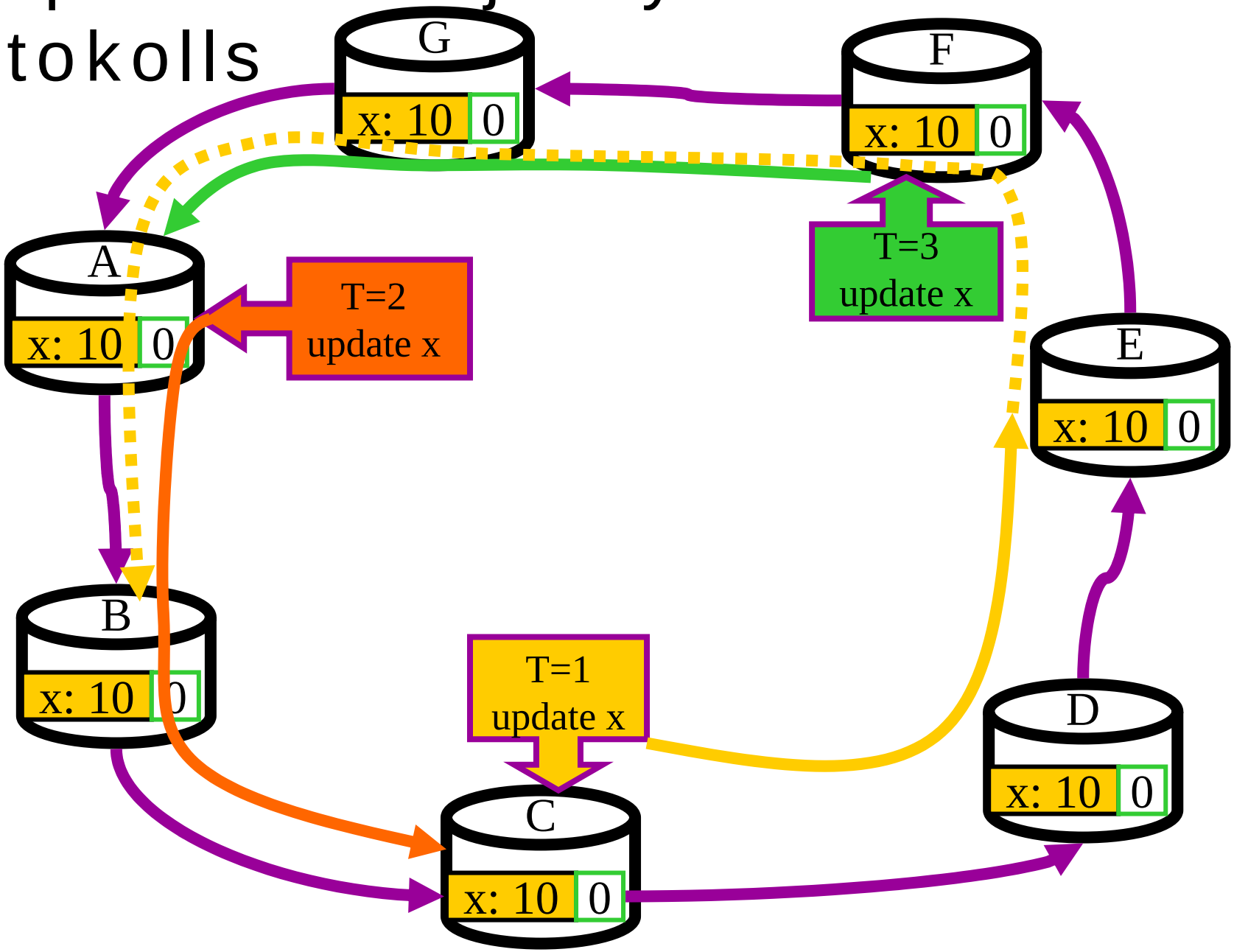
Annahme/Ablehnung

- Der Knoten, dessen Zustimmung den Ausschlag für die Mehrheit gibt, erzeugt die globale COMMIT-Meldung
- jeder Knoten, der mit ABGELEHNT stimmt, löst ein globales ABORT des Änderungsauftrags aus
- Wird eine Änderungs-Transaktion akzeptiert, werden die Änderungen bei Eintreffen eingebracht und die Objekt-Zeitstempel entsprechend aktualisiert.
- Wird eine Änderungstransaktion ABGELEHNT, so werden die verzögerten Abstimmungen je Knoten jeweils überprüft, ob jetzt eine Entscheidung möglich ist
- Bei ABLEHNUNG muss die Transaktion komplett wiederholt werden, einschließlich Objekt-Lesen (damit der jetzt aktuelle lokale Zustand der Eingabeobjekte gelesen wird)
- Lese-Transaktionen werden wie Pseudo-Schreib-Transaktionen behandelt. Nur so kann ermittelt werden, ob das gelesene Datum das aktuellste ist.

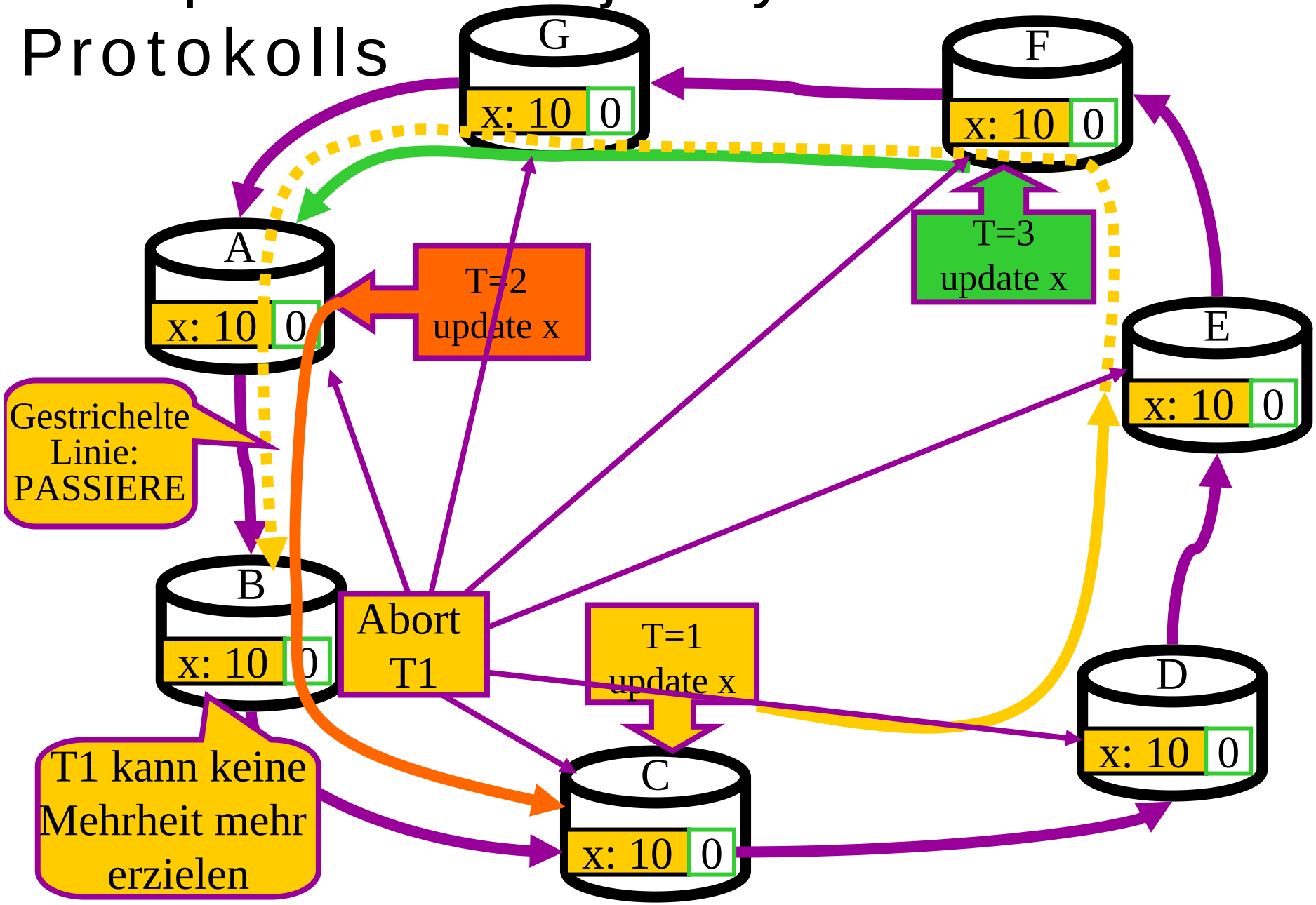
Beispiel des Majority/Consensus-Protokolls



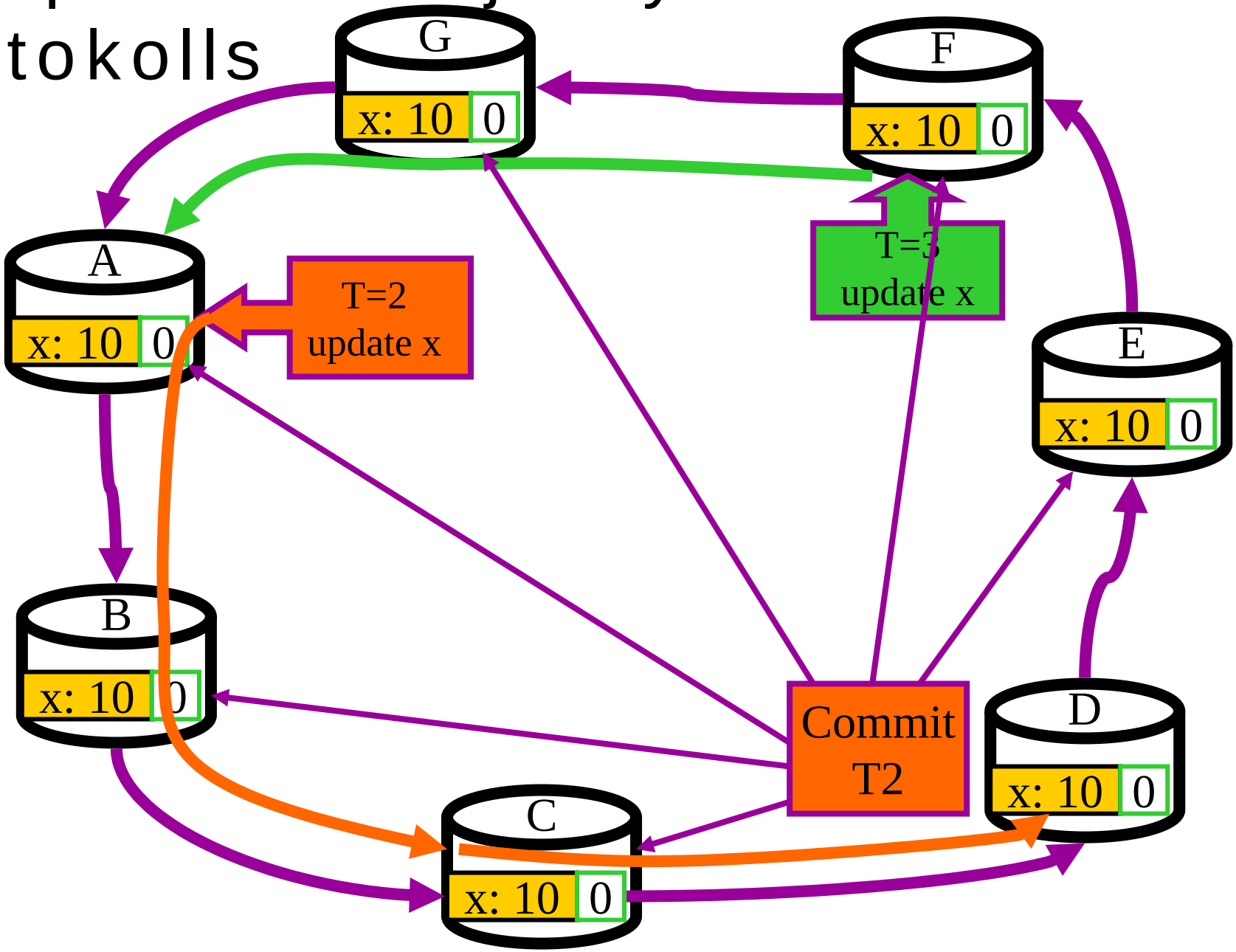
Beispiel des Majority/Consensus-Protokolls



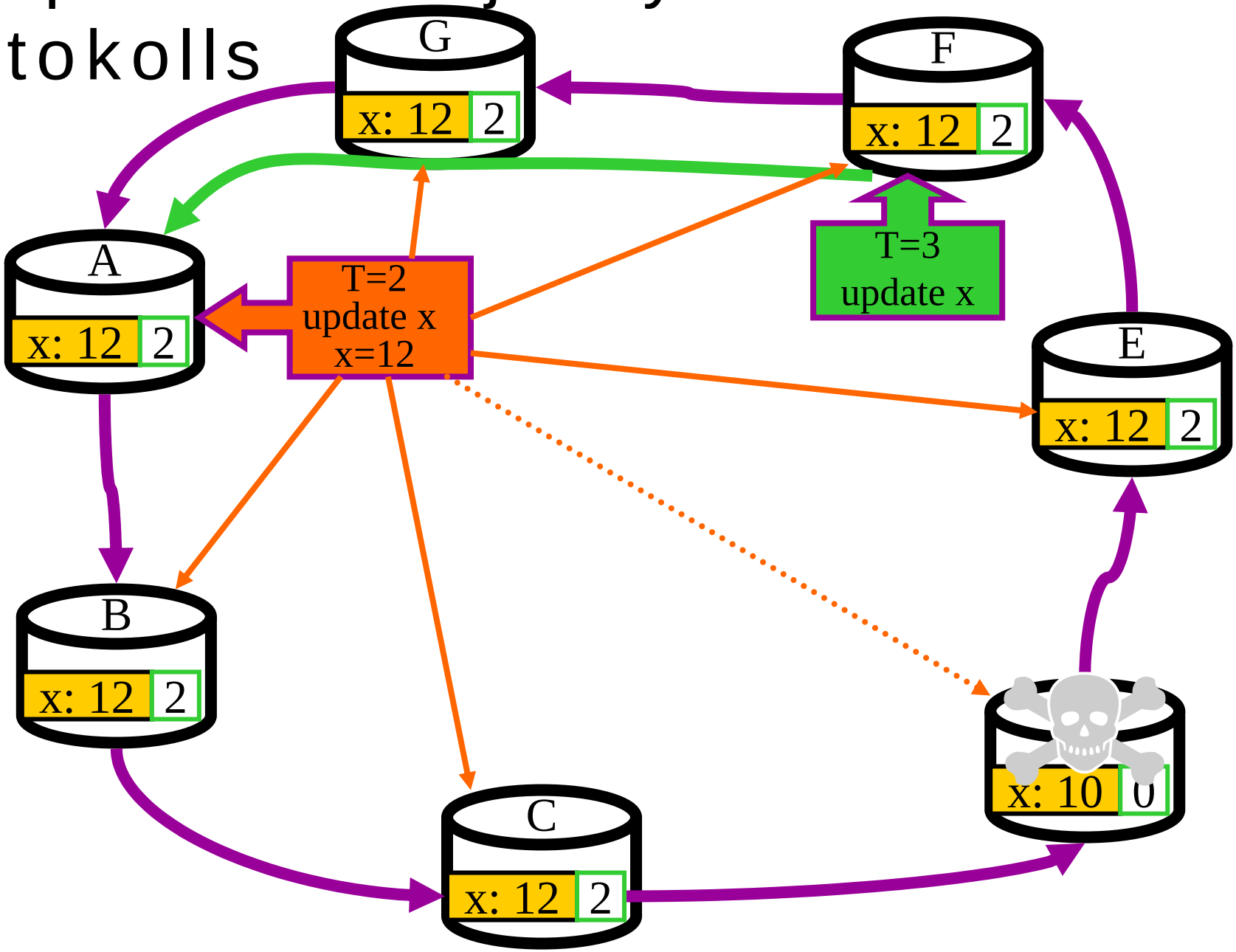
Beispiel des Majority/Consensus-Protokolls



Beispiel des Majority/Consensus-Protokolls

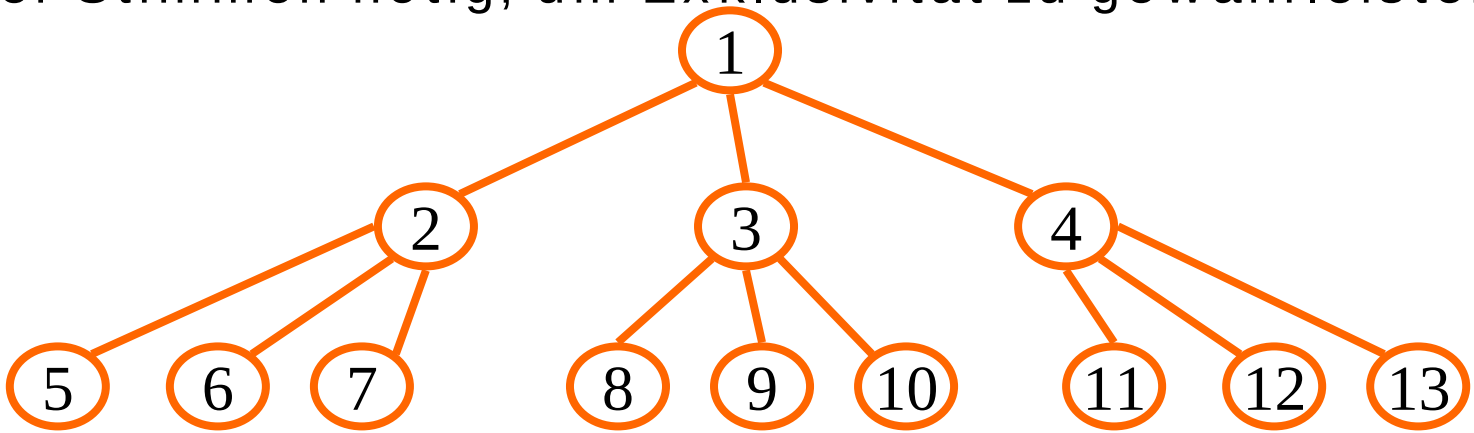


Beispiel des Majority/Consensus-Protokolls



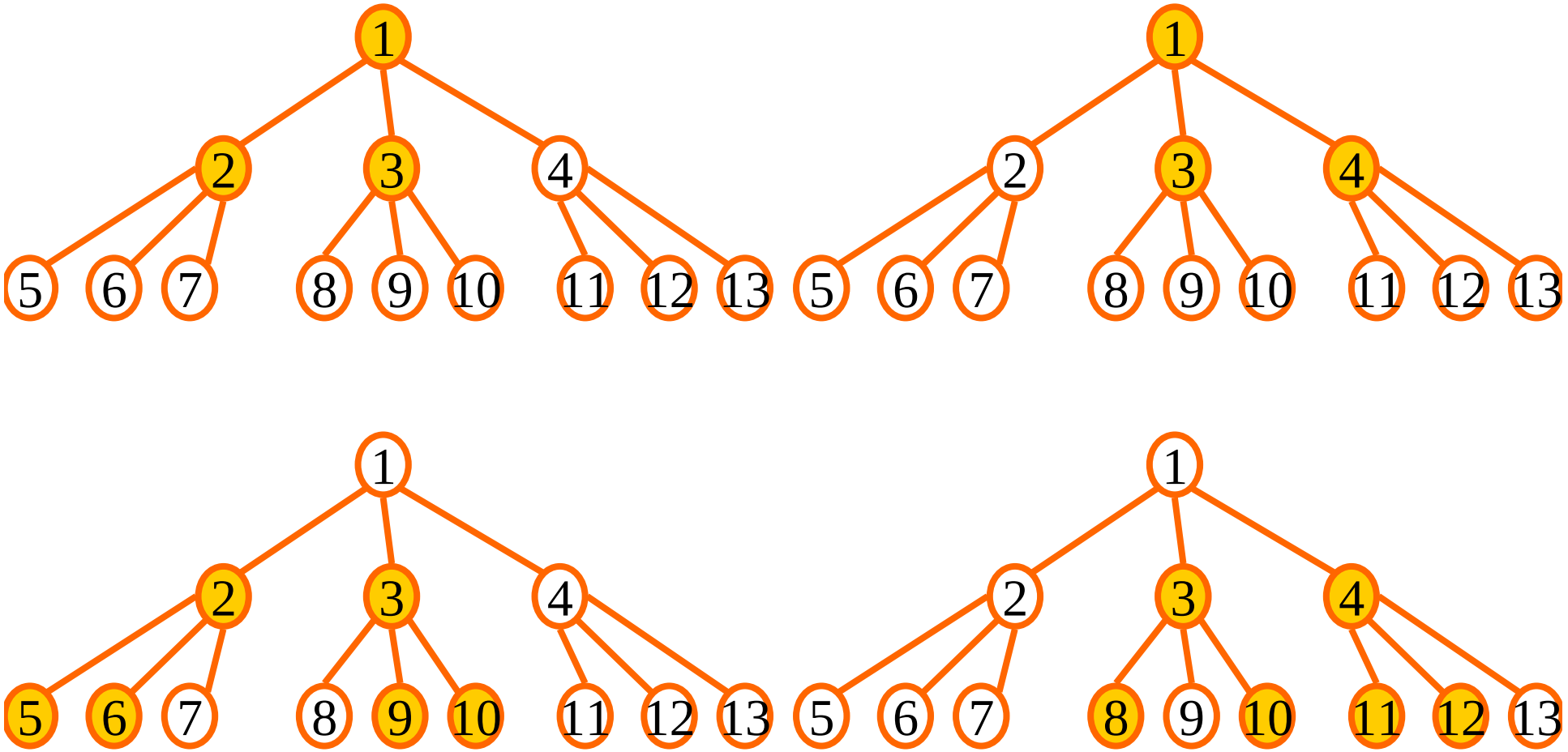
Tree Quorum

weniger Stimmen nötig, um Exklusivität zu gewährleisten



- Quorum q=(l,w)
  - sammle in l Ebenen jeweils mindestens w Stimmen je Teilbaumebene
  - wenn eine Ebene (Wurzel zB) weniger als w Stimmen hat, sammle alle
  - muss so gewählt werden, dass keine 2 Quoren gelichzeitig erzielbar sind
- Beispiel q=(2,2) wäre erfüllt mit
  - {1,2,3} oder {1,2,4} oder {2,3,5,7,8,9} oder {3,4,8,10,11,12}

Beispiel-Quoren



Größeres Beispiel

[Agrawal und Abadi: The generalized tree quorum protocol ..., ACM TODS, Dez. 1992]

q = (3,2)

Wenn man die Mehrheit der Kinder hat, hat man auch implizit den Vater

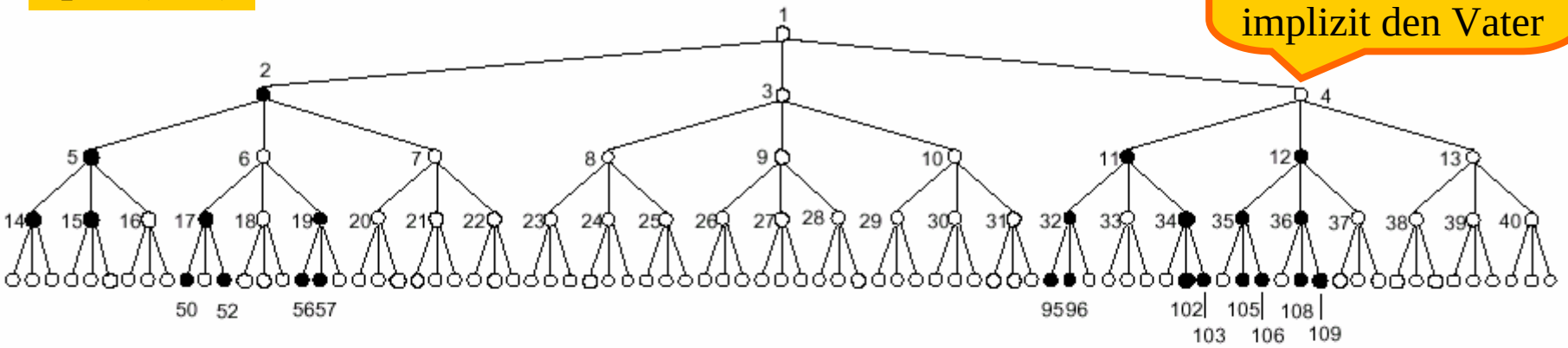


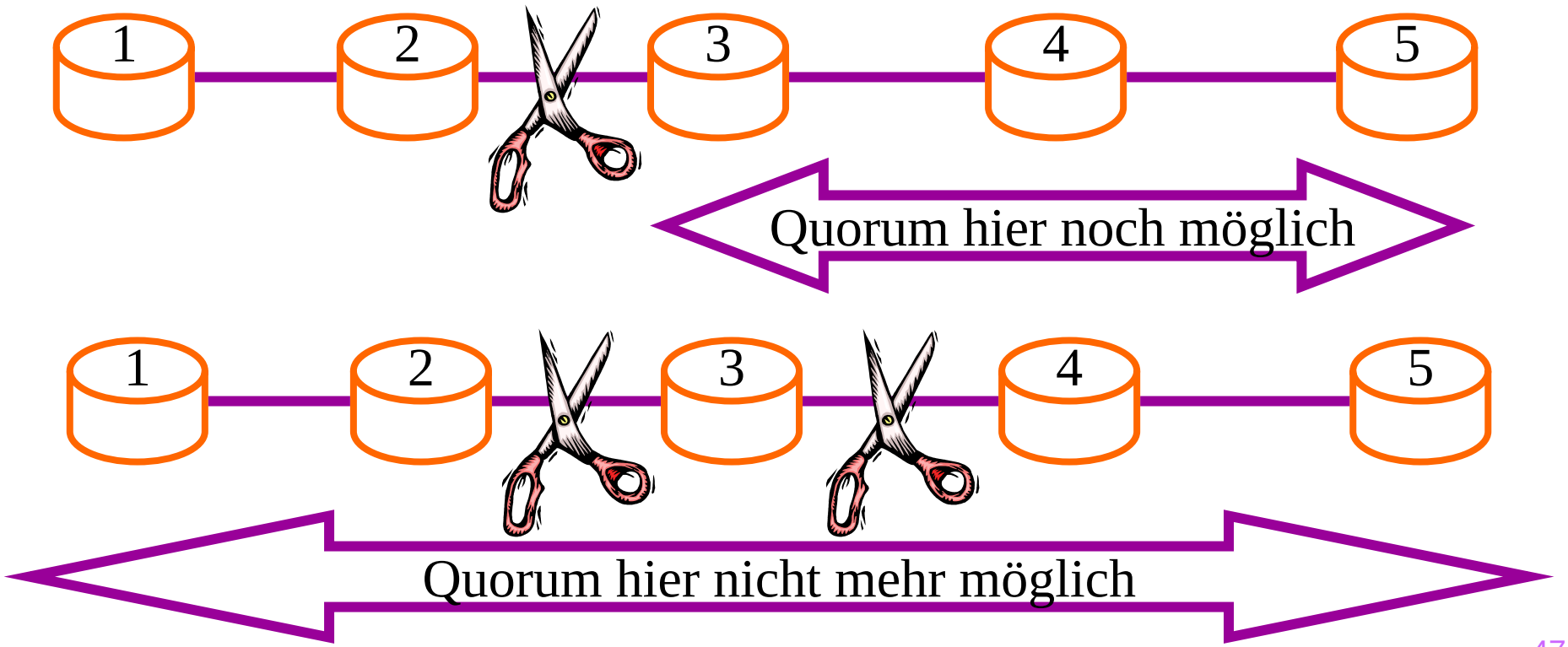
Abb. 9-8: Tree Quorum (Beispiel 9-7)

Anmerkungen:

- Majority Consensus würde die Zustimmung von mind. 61 Knoten benötigen, im dargestellten Fall genügen 24 Stimmen.
- Wäre die Wurzel mit dabei, so hätten sogar 7 Stimmen genügt.

Dynamische Quorums-Bildung

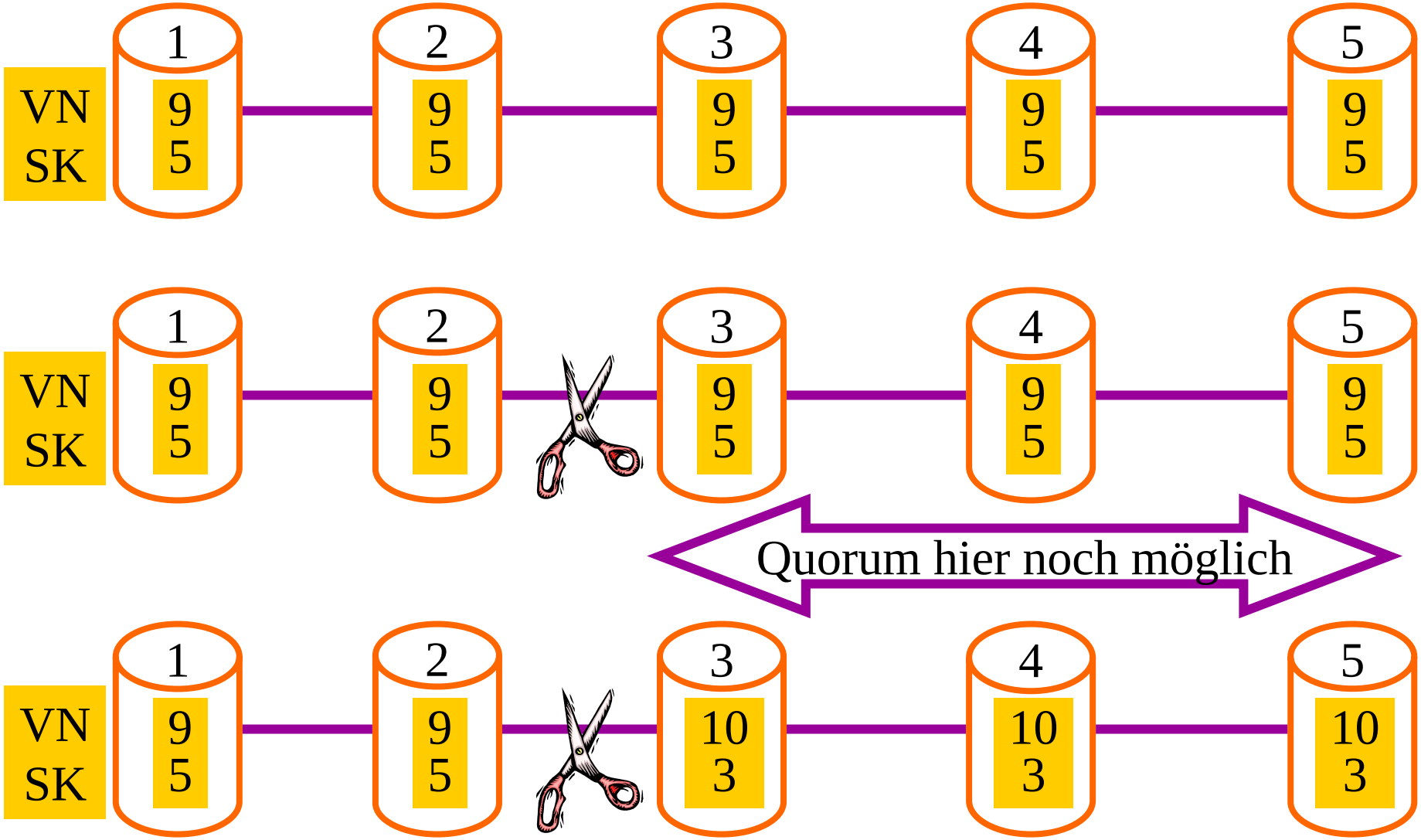
- Problem des statischen Quorums:
  - durch Netzpartitionierung oder mehrfachem Knotenausfall ist u.U. kein Quorum mehr erzielbar
- Beispiel: Q = 5, Qu = 3, Ql = 3



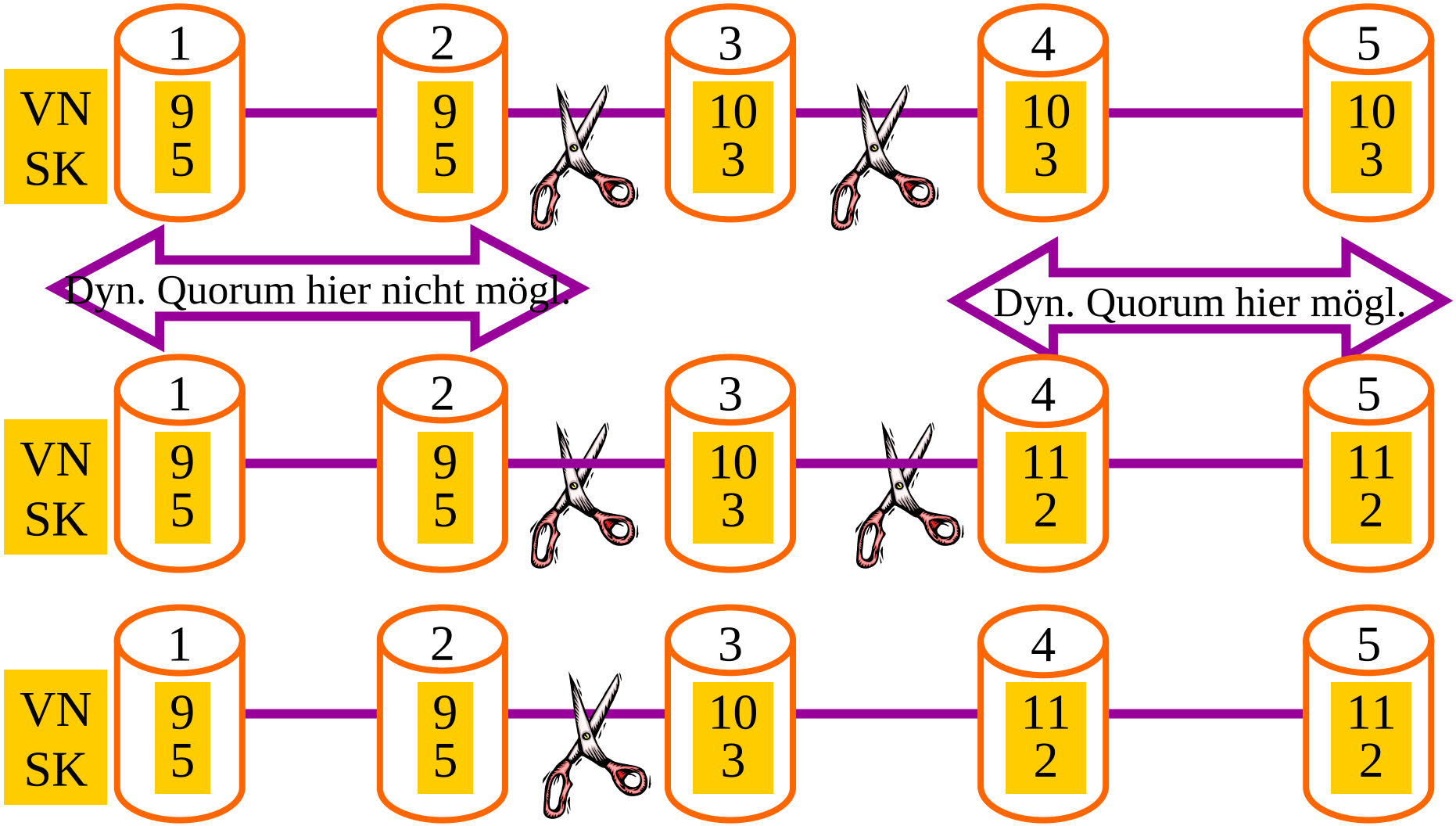
Dynamische Quorums-Bildung

- Nur diejenigen Knoten, die bei dem letzten Update „dabei waren“, besitzen Stimmrecht
  - Eine Mehrheit braucht man dann nur unter diesen abstimmungsberechtigten Knoten zu erzielen
- Dazu ist zusätzliche Verwaltungsinformation je Kopie notwendig
  - VN: Versionsnummer
  - SK: Anzahl der Knoten, die beim letzten Update dabei waren
    - ~ gegenwärtige Gesamtstimmenzahl

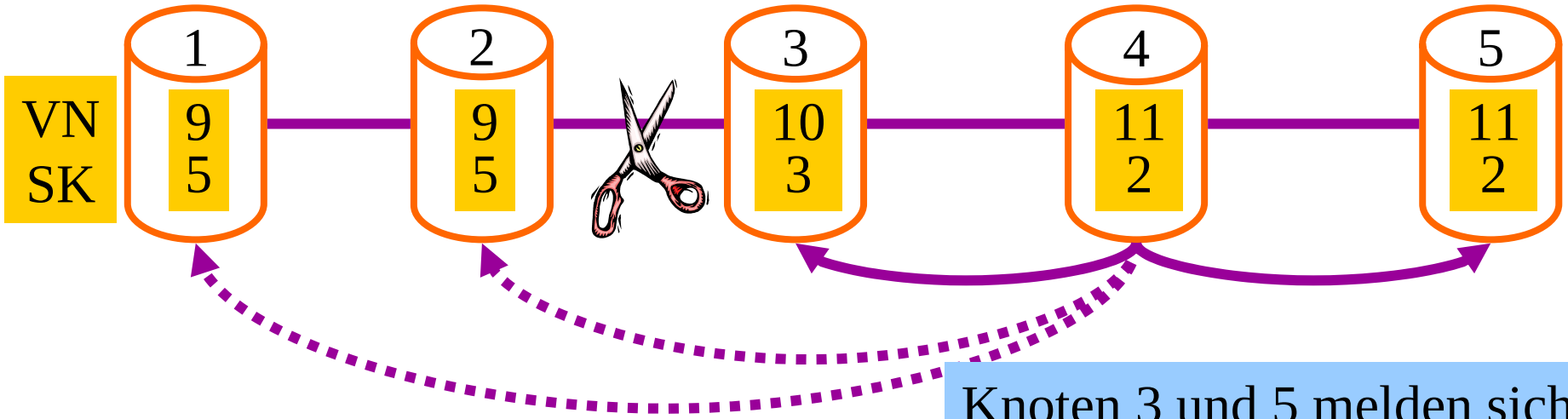
Dynamisches Quorum: Beispiel



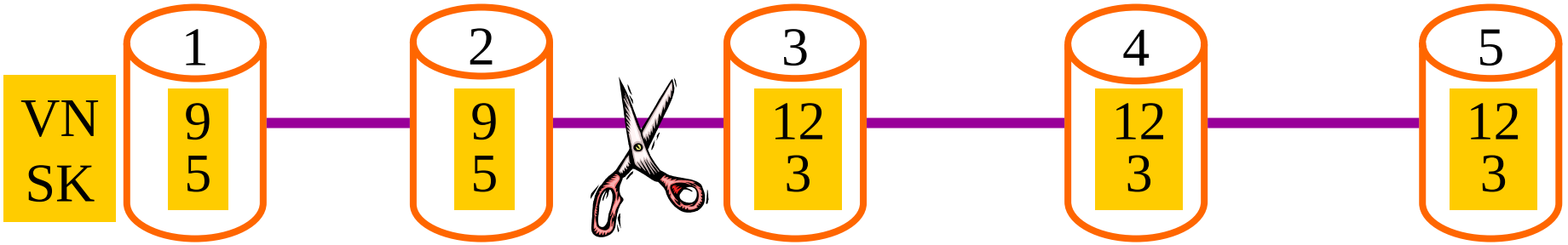
Dynamisches Quorum: Beispiel



Dynamisches Quorum: Wiedereingliederung eines deaktivierten Knoten (hier K3)



Knoten 3 und 5 melden sich mit ihren jeweiligen VN und SK Werten. K4 sendet K3 den neuesten Stand -- damit ist K3 wieder aktuell.

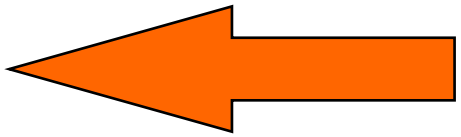


Replikation: Lazy (asynchron) versus Eager (synchron)

- Eager skaliert nicht
  - sehr hohe Deadlock-Rate
  - sehr „fette“ update Transaktionen
  - nur einsetzbar bei eher statischen Datenbanken
    - Bibel
    - Telefonbuch
- Lazy ist notwendig bei mobilen Benutzern, die mit Ihrem Notebook „durch die Gegend ziehen“
  - Versicherungsvertreter
  - Verkäufer
  - etc.
  - Transaktionen werden einfach mal ausgeführt und später wird versucht, die Replikate wieder „auf Vordermann“ zu bringen
    - Reconciliation bzw Konsolidierung

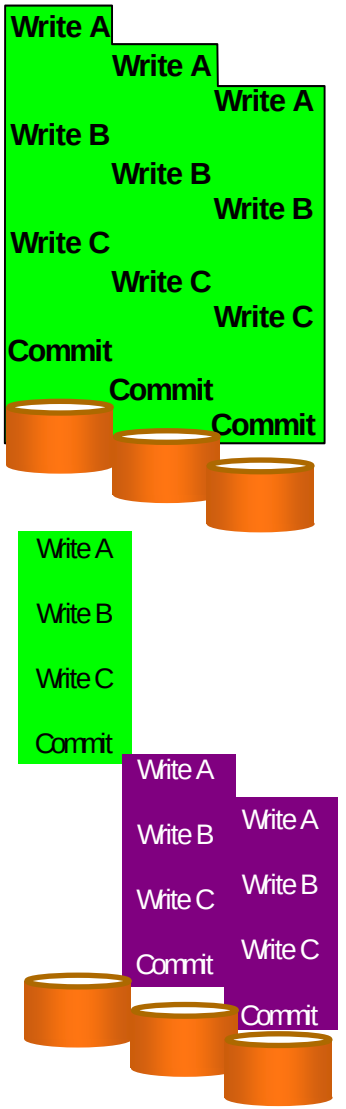
Outline

- Based on: Jim Gray, Pat Helland, Patrick E. O'Neil, Dennis Shasha: The Dangers of Replication and a Solution. SIGMOD Conf. 1996: 173-182
- Replication strategies
  - Lazy and Eager
  - Master (Primärkopie) and Group
- How centralized databases scale
  - deadlocks rise non-linearly with
    - transaction size
    - concurrency
- Replication systems are unstable on scaleup
- A possible solution



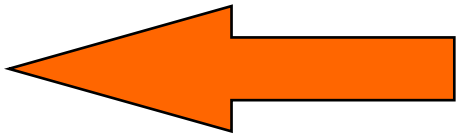
Propagation Strategies

- Eager: Send update right away
  - (part of same transaction)
  - *N* times larger transactions
- Lazy: Send update asynchronously
  - separate transaction
  - *N* times more transactions
- Either way
  - *N* times more updates per second per node
  - N^2 times more work overall



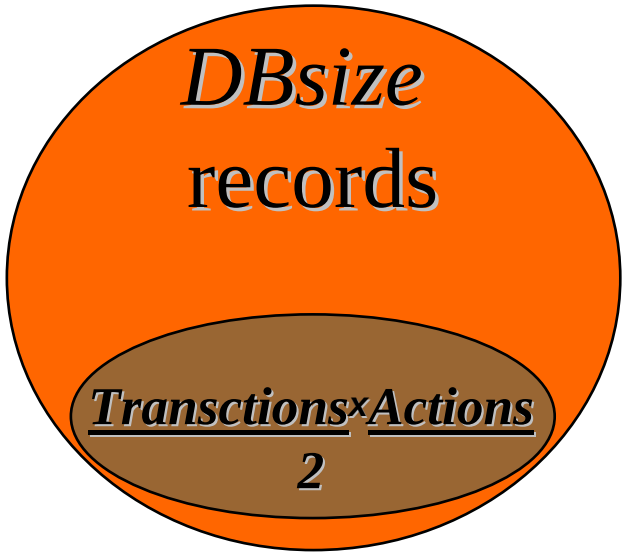
Outline

- Replication strategies
  - Lazy and Eager
  - Master and Group
- How centralized databases scale
  - deadlocks rise non-linearly
- Replication is unstable on scaleup
- A possible solution



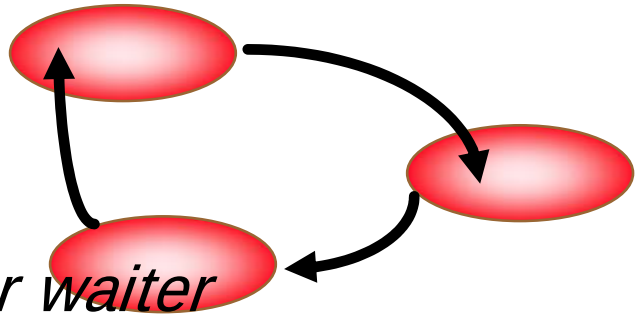
Simple Model of Waits

- *TPS* transactions per second
- Each
  - Picks *Actions* records uniformly from set of *DBsize* records
  - Then commits



- About *Transactions x Actions/2* resources locked
- Chance a request waits is $\frac{Transactions \times Actions}{2 \times DB\_size}$
- Action rate is *TPS x Actions*
- Active Transactions *TPS x Actions x Action_Time*
- Wait Rate = *Action rate x Chance a request waits*
- $= \frac{TPS^2 \times Actions^3 \times Action\_Time}{2 \times DB\_size}$
- 10x more transactions, 100x more waits

Simple Model of Deadlocks



- A *deadlock* is a wait cycle
- Cycle of length 2:
  - *Wait rate x Chance Waitee waits for waiter*
  - *Wait rate x (P(wait) / Transactions)*

$$\frac{TPS^2 \times Actions^3 \times Action\_Time}{2 \times DB\_size}$$

$$\frac{\frac{TPS \times Actions^3 \times Action\_Time}{2 \times DB\_size}}{TPS \times Actions \times Action\_Time}$$

$$\frac{TPS^2 \times Actions^5 \times Action\_Time}{4 \times DB\_size^2}$$

- Cycles of length 3 are PW^3, so ignored.

- 10x bigger trans = 100,000x more deadlocks

Eager Transactions are FAT

- If *N* nodes, eager transaction is *N*x bigger
  - Takes *N*x longer
  - 10x nodes, 1,000x deadlocks
  - N nodes, N**3 more deadlocks
  - (derivation in Gray´s paper)
- Master slightly better than group
- Good news:
  - Eager transactions only deadlock
  - No need for reconciliation

Outline

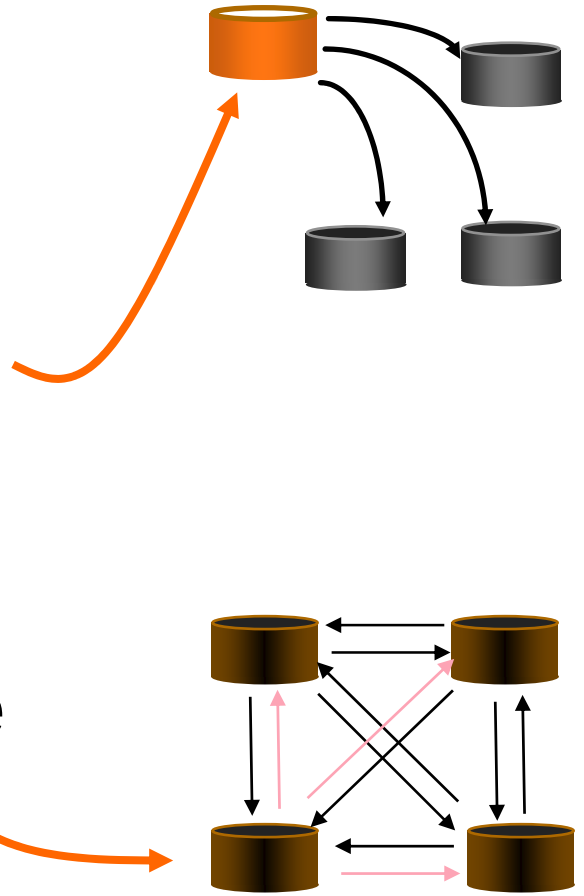
- Replication strategies **(lazy & eager, master & group)**
- How centralized databases scale
- Replication is unstable on scaleup
- A possible solution
  - Two-tier architecture: Mobile & Base nodes
  - Base nodes master objects
  - Tentative transactions at mobile nodes
    - Transactions must be commutative
  - Re-apply transactions on reconnect
  - Transactions may be rejected

Safe Approach

- Each object mastered at a node
- Update Transactions only
  read and write master items
- Lazy replication to other nodes
- Allow reads of stale data (on user request)
- PROBLEMS:
  - doesn’t support mobile users
  - deadlocks explode with scaleup
- ?? How do banks work???

Update Control Strategies

- Master
  - Each object has a master node
  - All updates start with the master
  - Broadcast to the subscribers
- Group
  - Object can be updated by anyone
  - Update broadcast to all others
- Everyone *wants* Lazy Group:
  - update anywhere, anytime, anyway



Quiz Questions: Name One

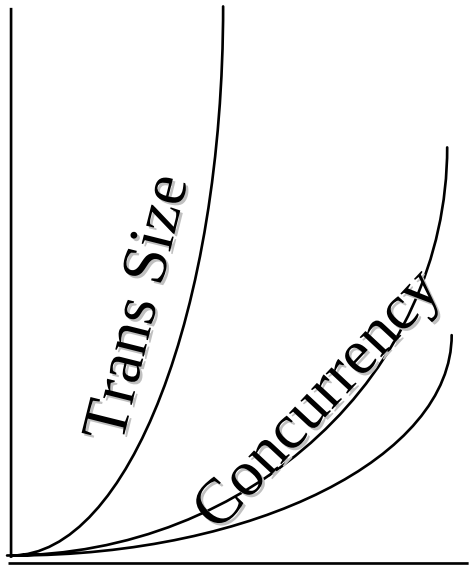
- Eager
  - Master: N-Plexed disks
  - Group: ?
- Lazy
  - Master: Bibles, Bank accounts, SQLserver
  - Group: Name servers, Oracle, Access…
- Note: Lazy contradicts Serializable
  - If two lazy updates collide, then … *reconcile*
    - discard one transaction (or use some other rule)
    - Ask for human advice
- Meanwhile, nodes disagree =>
  - Network DB state diverges: *System Delusion*

Anecdotal Evidence

- Update Anywhere systems are attractive
- Products offer the feature
- It demos well
- But when it scales up
  - Reconciliations start to cascade
  - Database drifts “out of sync” *(System Delusion)*
- What’s going on?
- In diesen Systemen wird nicht mehr unbedingt Serialisierbarkeit angestrebt
- Konvergenz
  - Alle Knoten (~Replikate) konvergieren „langfristig“ zu demselben Wert
  - zB führe nur Änderungen mit größerem Zeitstempel durch
    - Lost update - Problem

Summary So Far

- Even centralized systems unstable
- Waits:
  - Square of concurrency
  - 3rd power of transaction size
- Deadlock rate
  - Square of concurrency
  - 5th power of transaction size

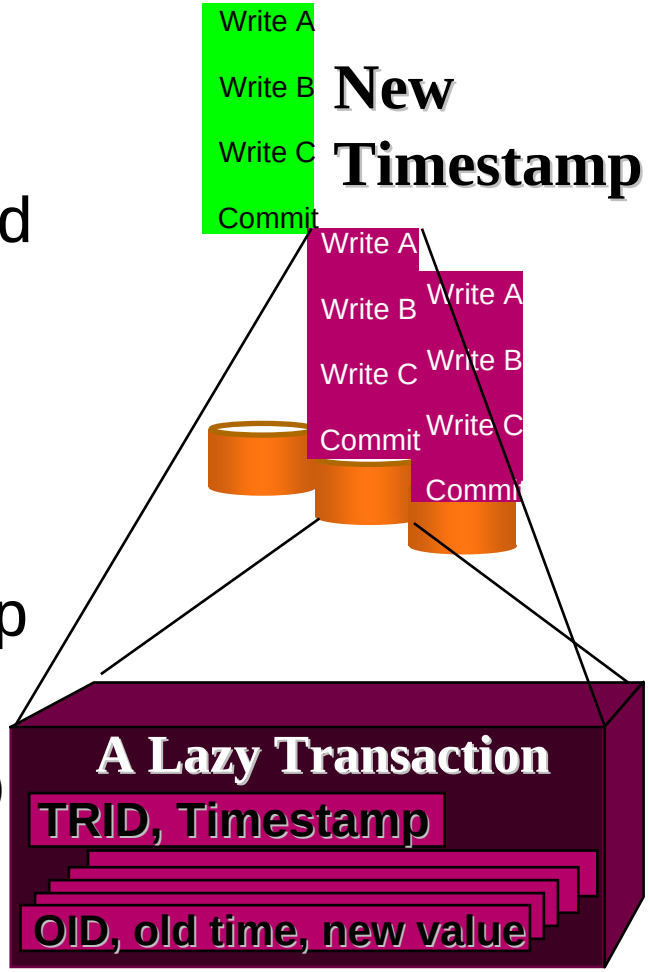


Eager Transactions are FAT

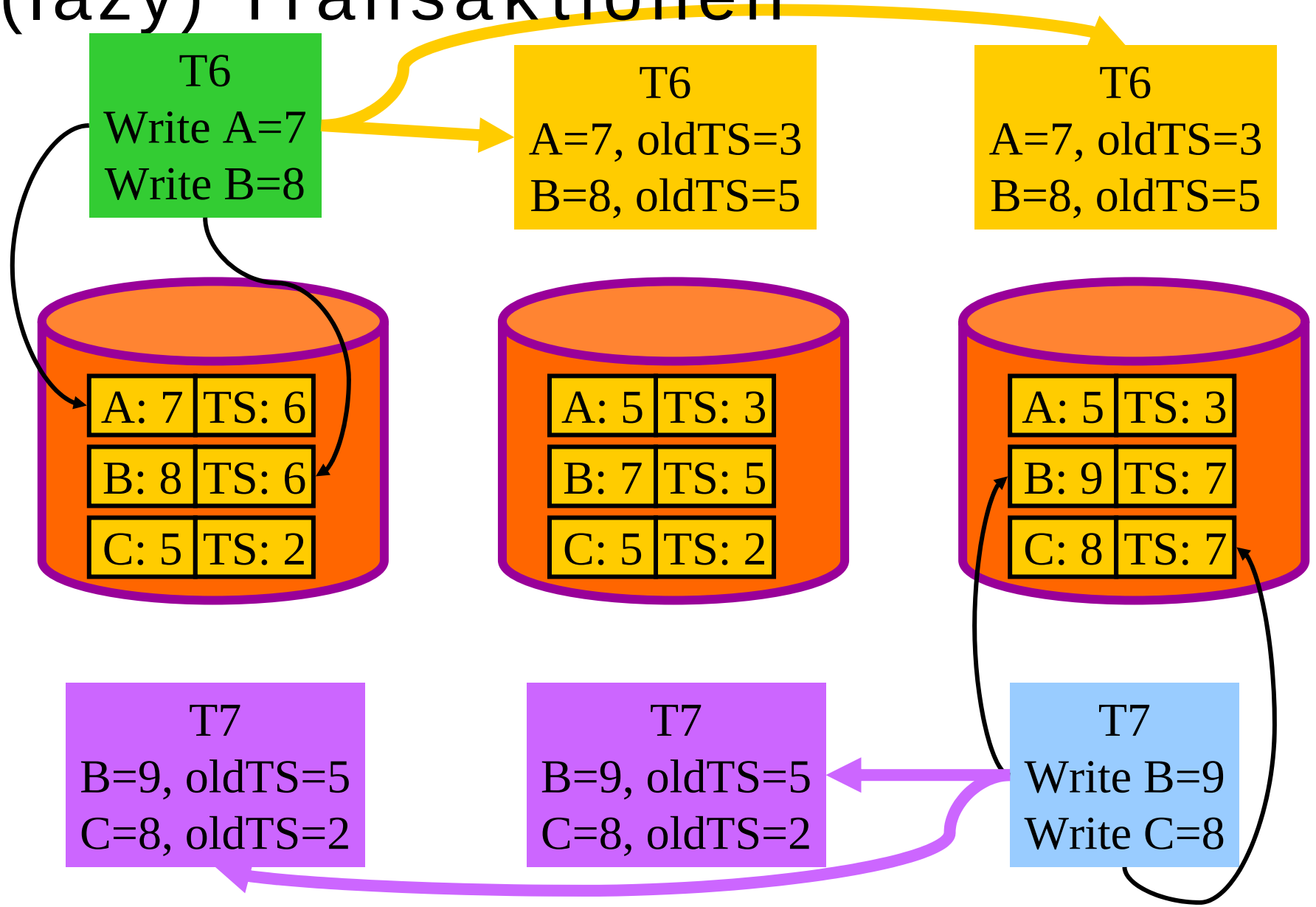
- If *N* nodes, eager transaction is *N*x bigger
  - Takes *N*x longer
  - 10x nodes, 1000x deadlocks
  - (derivation in Gray et al´s paper)
- Master slightly better than group
- Good news:
  - Eager transactions only deadlock
  - No need for reconciliation

Lazy Master & Group

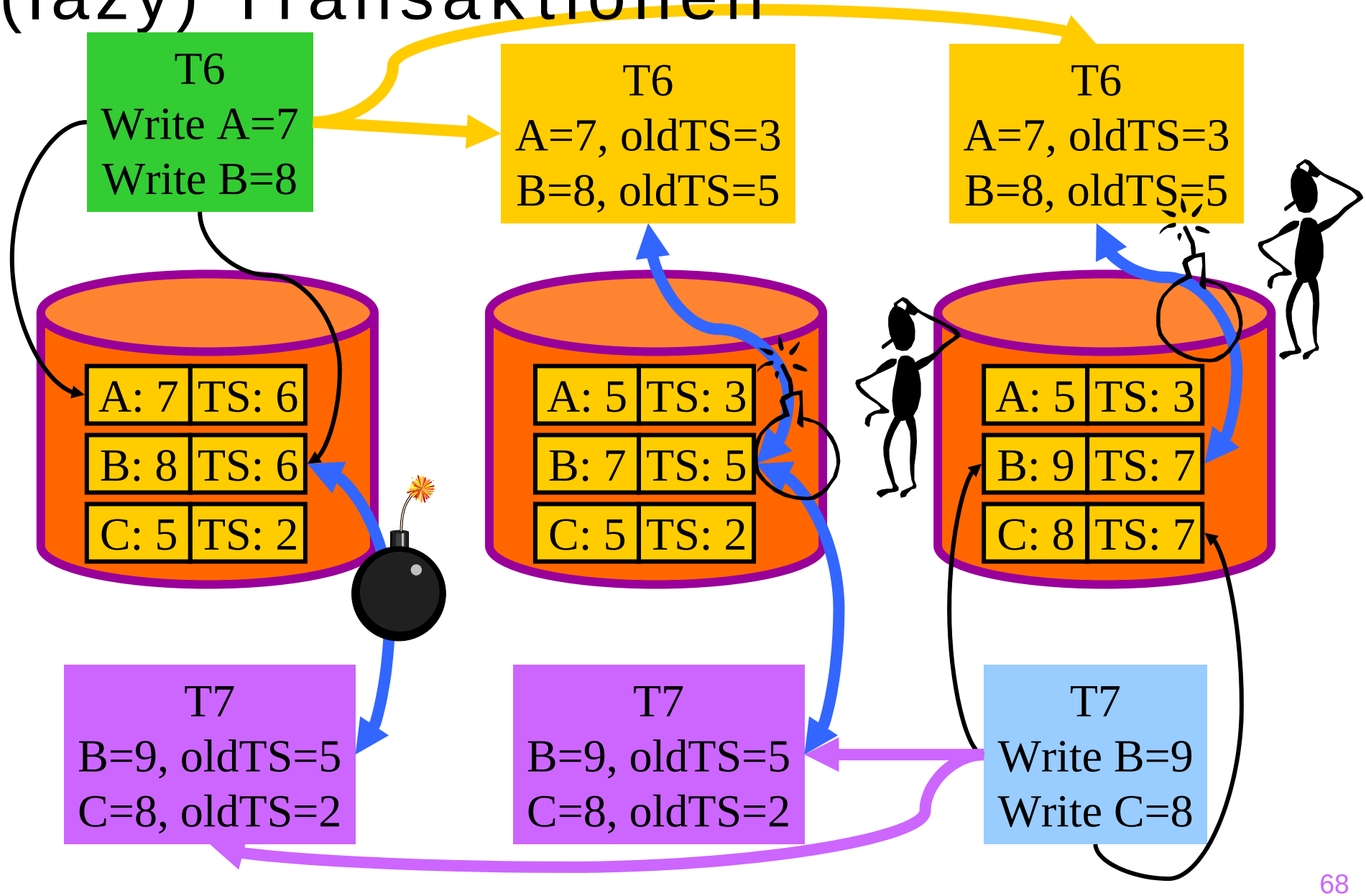
- Use optimistic concurrency control
  - Keep transaction timestamp with record
  - Updates carry old+new timestamp
  - If record has old timestamp
    - set value to new value
    - set timestamp to new timestamp
  - If record does not match old timestamp
    - reject lazy transaction
  - Not SNAPSHOT isolation (stale reads)
- Reconciliation:
  - Some nodes are updated
  - Some nodes are “being reconciled”



Zwei konkurrierende asynchrone (lazy) Transaktionen



Zwei konkurrierende asynchrone (lazy) Transaktionen



Reconciliation

- Reconciliation means System Delusion
  - Data inconsistent with itself and reality
- How frequent is it?
- Lazy transactions are not fat
  - but N times as many
  - Eager waits become Lazy reconciliations
  - Rate is:

$$\frac{TPS^2 \times (Actions \times Nodes)^3 \times Action\_Time}{2 \times DB\_size}$$

  - Assuming everyone is connected

Konsolidierung (Reconciliation) zweier Transaktionen

- LATEST: zuletzt eingebrachte Änderung gilt
  - (-) „Lost update“ der älteren Transaktion
  - (+) Konvergenz: alle Kopien werden irgendwann denselben Wert annehmen
- PRIMARY k: Tupel an Knoten k gilt
- ARITHMETIK: neuer Wert = Wert1 + Wert2 – alter Wert
  - Alter Wert muss aus dem Log geholt werden
- PROGRAM
  - Benutzer-definierte Prozedur zur Konsolidierung
- NOTIFY
  - Benutzer wird aufgefordert, die Konsolidierung „von Hand“ durchzuführen
  - Kann man nicht zu oft machen (skaliert nicht)

Konto#	Name	Kontostand	Einfügungsregel	Integrationsregel
			KEEP	ARITHMETIC

Ausgangssituation:

Knoten 1		
Konto#	Name	Kontostand
1723	Maier	1.000 DM

Knoten 2		
Konto#	Name	Kontostand
1723	Maier	1.000 DM

Abhebung: 200 DM

Abhebung: 300 DM

Knoten 1		
Konto#	Name	Kontostand
1723	Maier	800 DM

Knoten 2		
Konto#	Name	Kontostand
1723	Maier	700 DM

<u>Wiedervereinigung</u>: Integrationsregel = ARITHMETIC

$\Rightarrow$ $Kontostand_{neu}$:= $Kontostand_1$ + $Kontostand_2$ - $Kontostand_{alt}$

= 800 DM + 700 DM - 1.000 DM

= 500 DM

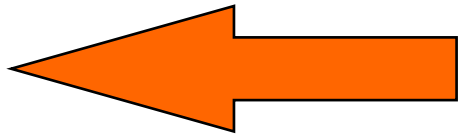
Eager & Lazy: Disconnected

- Suppose mobile nodes disconnected for a day
- When reconnect:
  - get all incoming updates
  - send all delayed updates
- Incoming is $\frac{Nodes \times TPS \times Actions \times disconnect\_time}{Action\_Time}$
- Outgoing is: $\frac{TPS \times Actions \times Disconnect\_Time}{Action\_Time}$
- Conflicts are intersection of these two sets

$$\frac{Disconnect\_Time \times (TPS \times Actions \times Nodes)^2}{DB\_size}$$

Outline

- Replication strategies **(lazy & eager, master & group)**
- How centralized databases scale
- Replication is unstable on scaleup
- A possible solution



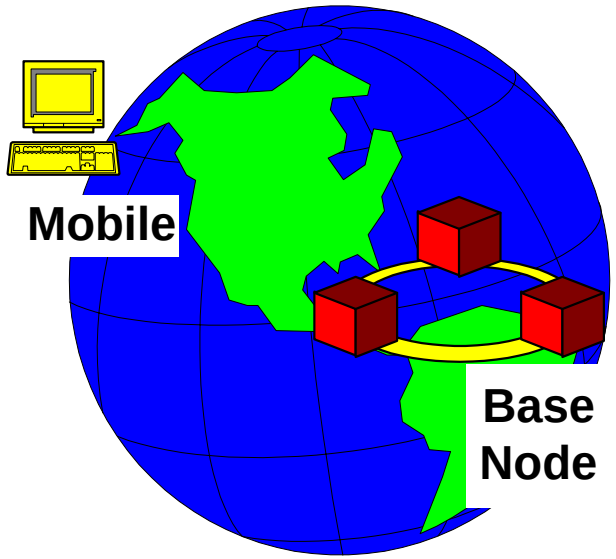
  - Two-tier architecture: Mobile & Base nodes
  - Base nodes master objects
  - Tentative transactions at mobile nodes
    - Transactions must be commutative
  - Re-apply transactions on reconnect
  - Transactions may be rejected

Safe Approach

- Each object mastered at a node
- Update Transactions only
  read and write master items
- Lazy replication to other nodes
- Allow reads of stale data (on user request)
- PROBLEMS:
  - doesn’t support mobile users
  - deadlocks explode with scaleup
- ?? How do banks work???

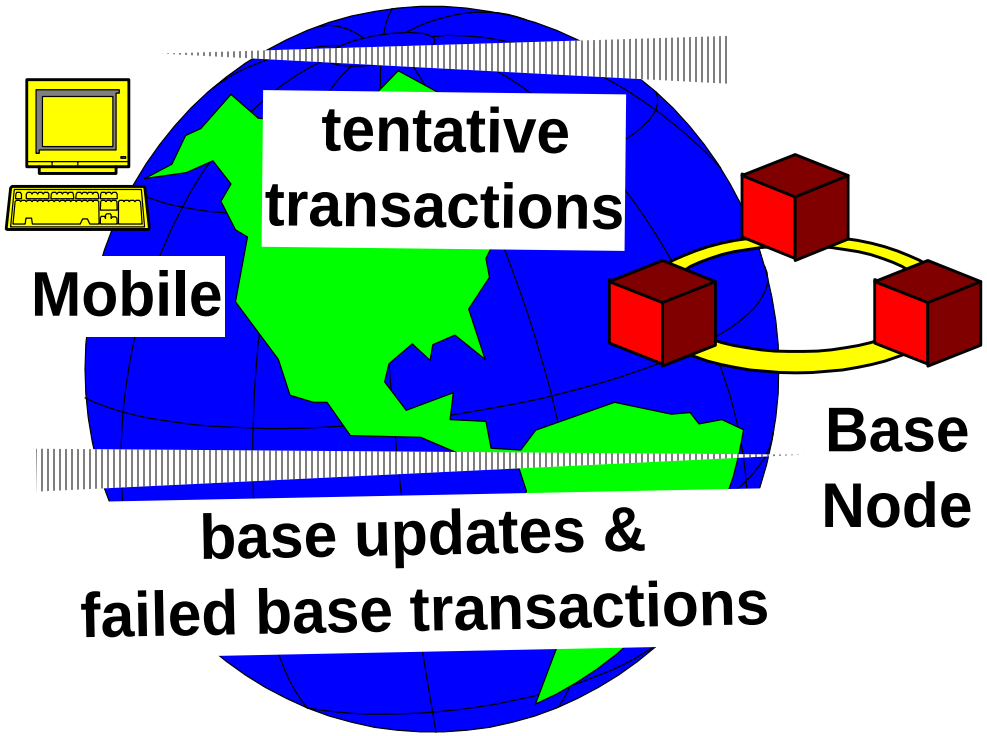
Two Tier Replication

- Two kinds of nodes:
  - Base nodes always connected, always up
  - Mobile nodes occasionally connected
- Data mastered at base nodes
- Mobile nodes
  - have stale copies
  - make tentative updates



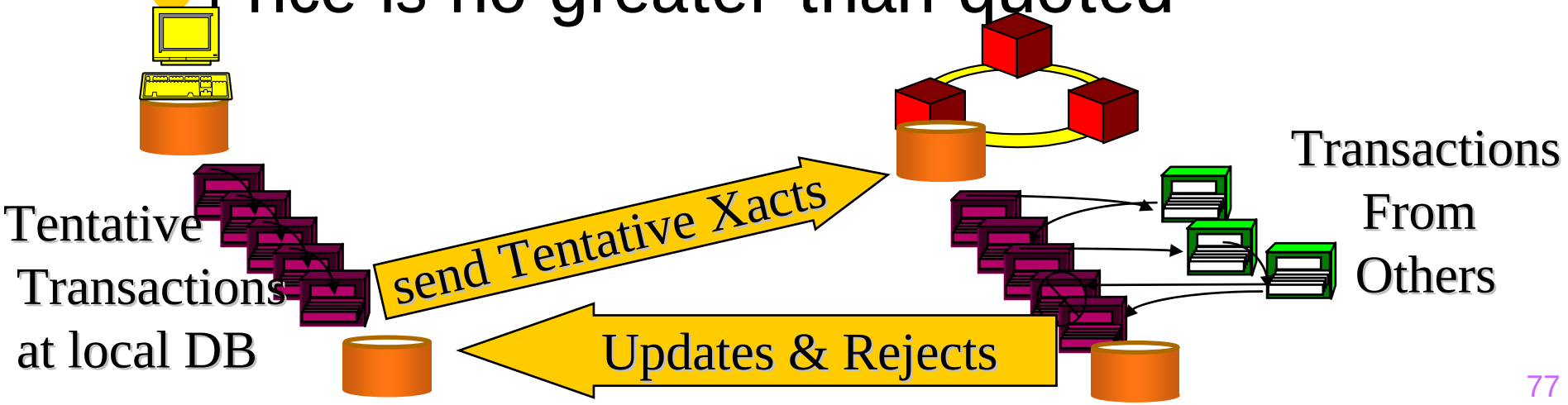
Mobile Node Makes Tentative Updates

- Updates local database while disconnected
- Saves transactions
- When Mobile node reconnects:
  Tentative transactions re-done
  as Eager-Master
  (at original time??)
- Some may be rejected
  - (replaces reconciliation)
- No System Delusion.

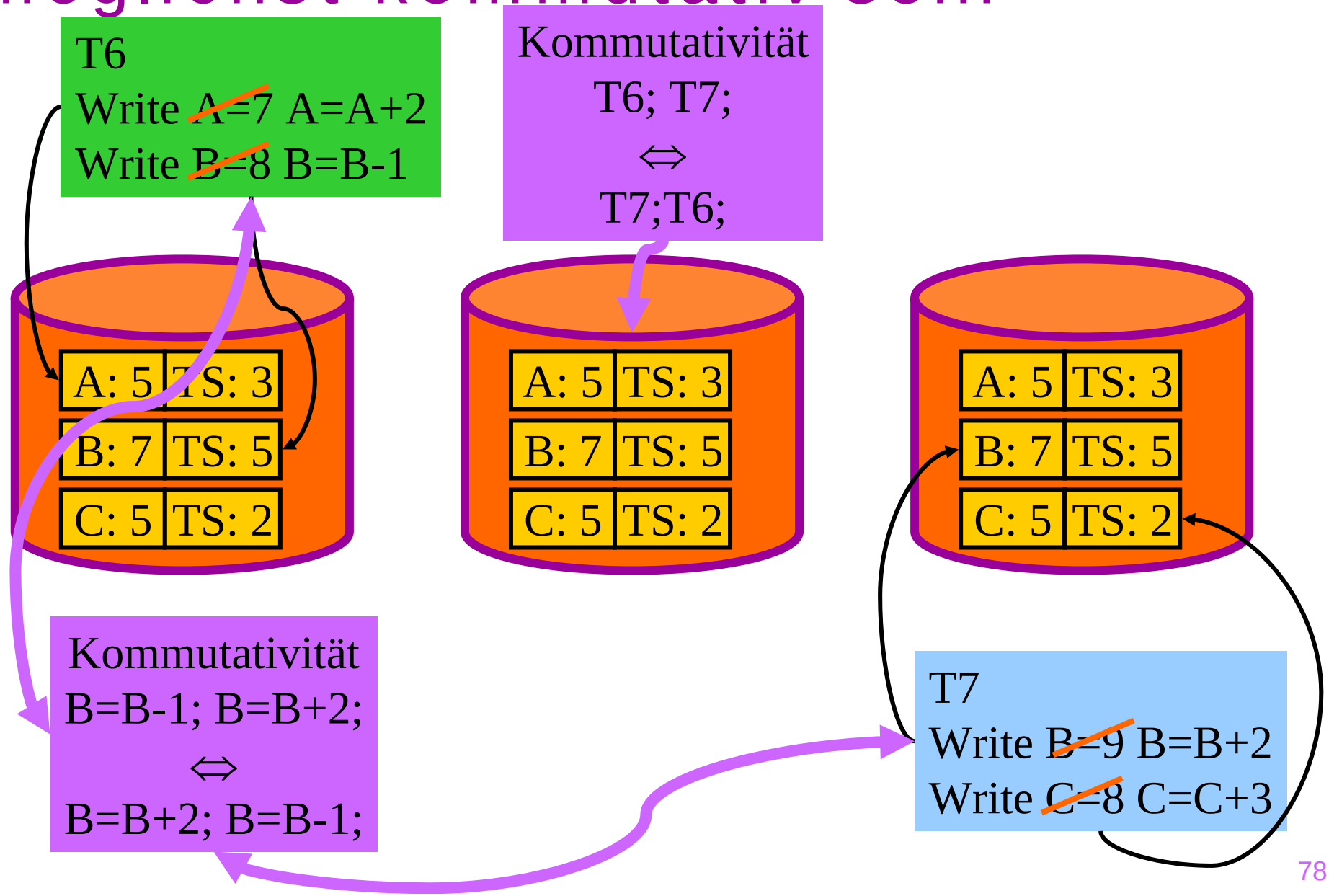


Tentative Transactions

- Must be commutative with others
  - Debit 50$ rather than Change 150$ to 100$.
- Must have acceptance criteria
  - Account balance is positive
  - Ship date no later than quoted
  - Price is no greater than quoted



Tentative Transaktionen sollten möglichst kommutativ sein



Kommutative (C-TA) vs Nicht-Kommutative TAs (NC-TA)

- C-Tas können in beliebiger Reihenfolge eingebracht werden
  - Es ist irrelevant, in welcher Reihenfolge zwei Einzahlungen auf ein Konto durchgeführt werden
    - (E1;E2) = (E2,E2)
  - Zwei Auszahlungen bzw eine Auszahlung und eine Einzahlung sind dann kommutativ, wenn „genug Geld auf dem Konto ist“
    - Anwendungsspezifische Kriterien für Kommutativität
  - Es gilt (A1;A2)=(A2;A1) wenn (Kontostand - A1 - A2) > 0
- Zinsausschüttung und Einzahlung/Auszahlung sind nicht kommutativ
  - Konvertiere NC-TA in eine C-TA
  - Konvertiere Zinsausschüttung in eine Einzahlung
  - Siehe Beispiel

Knoten 1

Knoten 2

Einzahlungs-TA: $T_E(e,x) = x + e$: Zahlt e DM auf Konto x ein

Auszahlungs-TA: $T_A(e,x) = x - e$: Hebt e DM von Konto x ab

Zinszahlungs-TA: $T_Z(x) = x + 10\%$: Erhöht Konto x um 10%

C-TA: T_E, T_A

NC-TA: T_Z

Ausführung der NC-TA T_Z:

1. Berechnung der Zinsen (= 10 DM) auf Kopie x_1 und Aktualisierung von Kopie x_1 (= COMMIT)

2. Übermittlung der Zins-Gutschrift als Einzahlungs-TA
   $T_{E_Z}(s,x_2) = T_E(10\ DM,x_2)$ an Kopie x_2

Refinement: Mobile Node Can Master Some Data

- Mobile node can master *"private"* data
  - Only mobile node updates this data
  - Others only read that data
- Examples:
  - Orders generated by salesman
  - Mail generated by user
  - Documents generated by Notes user.

Virtue of 2-Tier Approach

- Allows mobile operation
- No system delusion
- Rejects detected at reconnect (know right away)
- If commutativity works,
  - No reconciliations
  - Even though work rises as $(\text{Mobile} + \text{Base})^2$

Replikationskontrolle: Beispielsysteme

- Domain Name Server (DNS)
  - Primärkopien liefern autoritative Antworten
  - Update immer erst auf der Primärkopie
  - Asynchrone Propagierung der Updates an andere Replikate
  - Zusätzlich Caching/Pufferung
    - TTL: Time To Live
    - Ungültiger Cache-Eintrag wird erkannt und ersetzt
- CVS
  - Front-end für RCS
  - Update anywhere
  - Konsolidierung beim check-in
    - Zeitstempel-basierte Kontrolle
    - Wenn Zeitstempel zu alt, wird automatische Konsolidierung (~zeilenweiser merge zweier Dateien) versucht
    - Evtl. wird der Benutzer um Hilfe „ersucht“

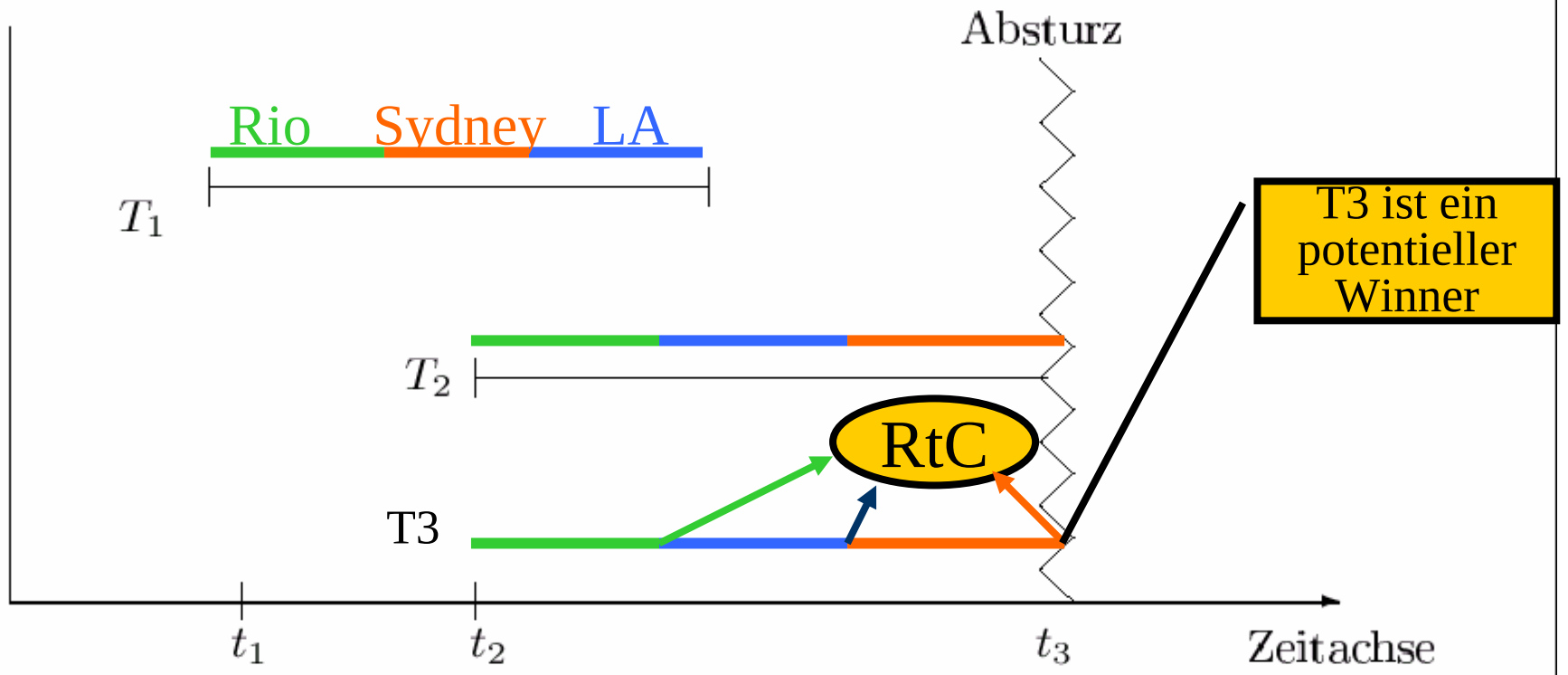
Replikationskontrolle: Beispielsysteme (cont´d)

- MS Access
  - Primärkopie für Schema-Daten
    - Können nur an der Primärkopie geändert werden
  - Update anywhere auf replizierten Datensätzen
  - Periodische Propagierung der Änderungen an andere Replikate
    - Der mit dem jüngsten Zeitstempel „gewinnt“
      - Konvergenz ist oberstes Gebot
      - Lost update des Verlierers
    - Der Verlierer wird informiert und muß versuchen, seine Änderungen nochmals einzubringen
- Oracle
  - Erlaubt auch „Group Update“
  - Konfigurierbare und eigen-programmierbare Konsolidierungsregeln
  - Replikatkonsolidierung besonders wichtig für den Abgleich zwischen Oracle Lite und Oracle Server
    - Eigentlich zwei unterschiedliche Systeme

Recovery verteilter Datenbanken

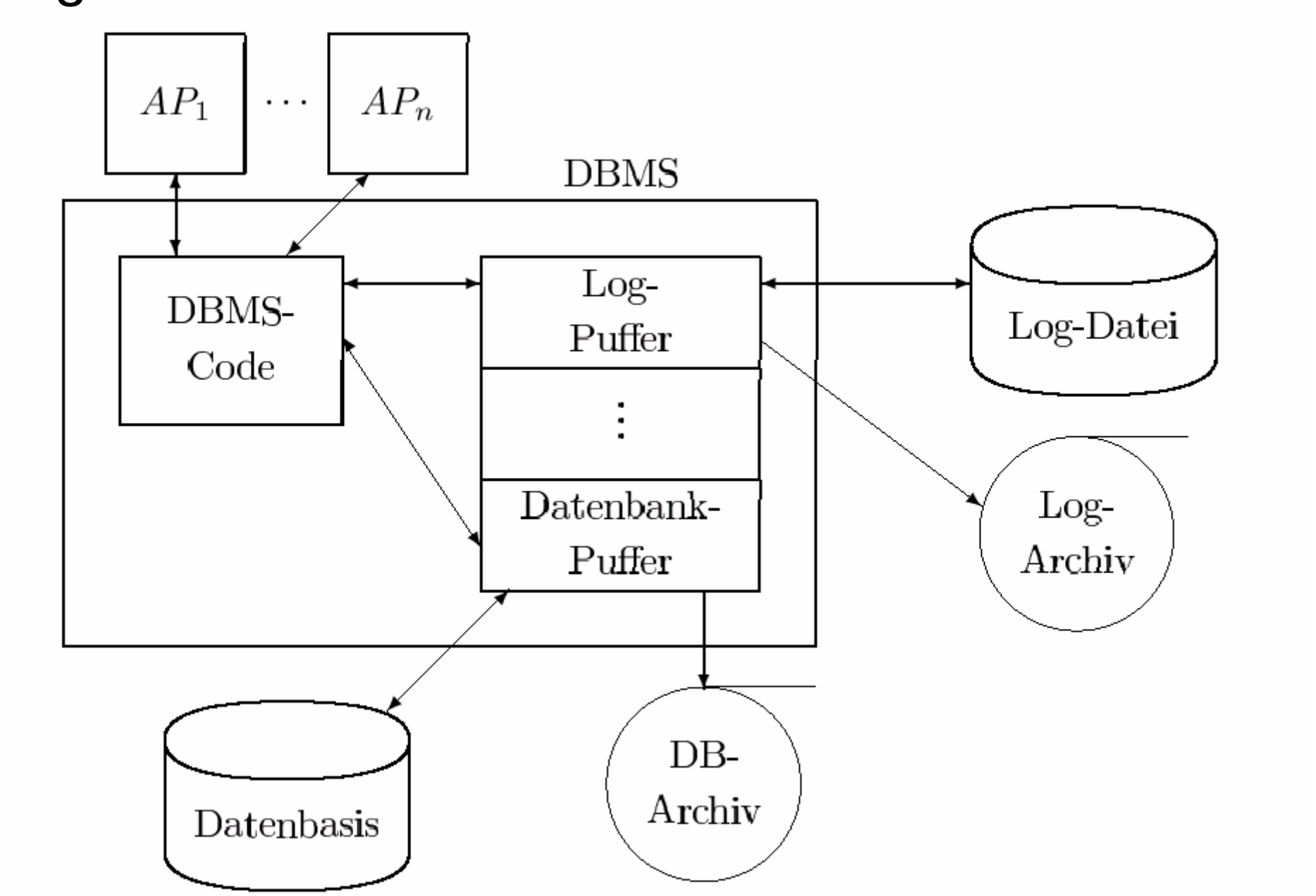
Protokollierung
Sicherungspunkte
Restart/Wiederanlauf

Transaktionsbeginn und -ende relativ zu einem Systemabsturz



- Transaktionen der Art T_1 müssen hinsichlich ihrer Wirkung vollständig nachvollzogen werden. Transaktionen dieser Art nennt man *Winner*.
- Transaktionen, die wie T_2 zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden. Diese Transaktionen bezeichnen wir als *Loser*.

Log-Architektur einer lokalen Station



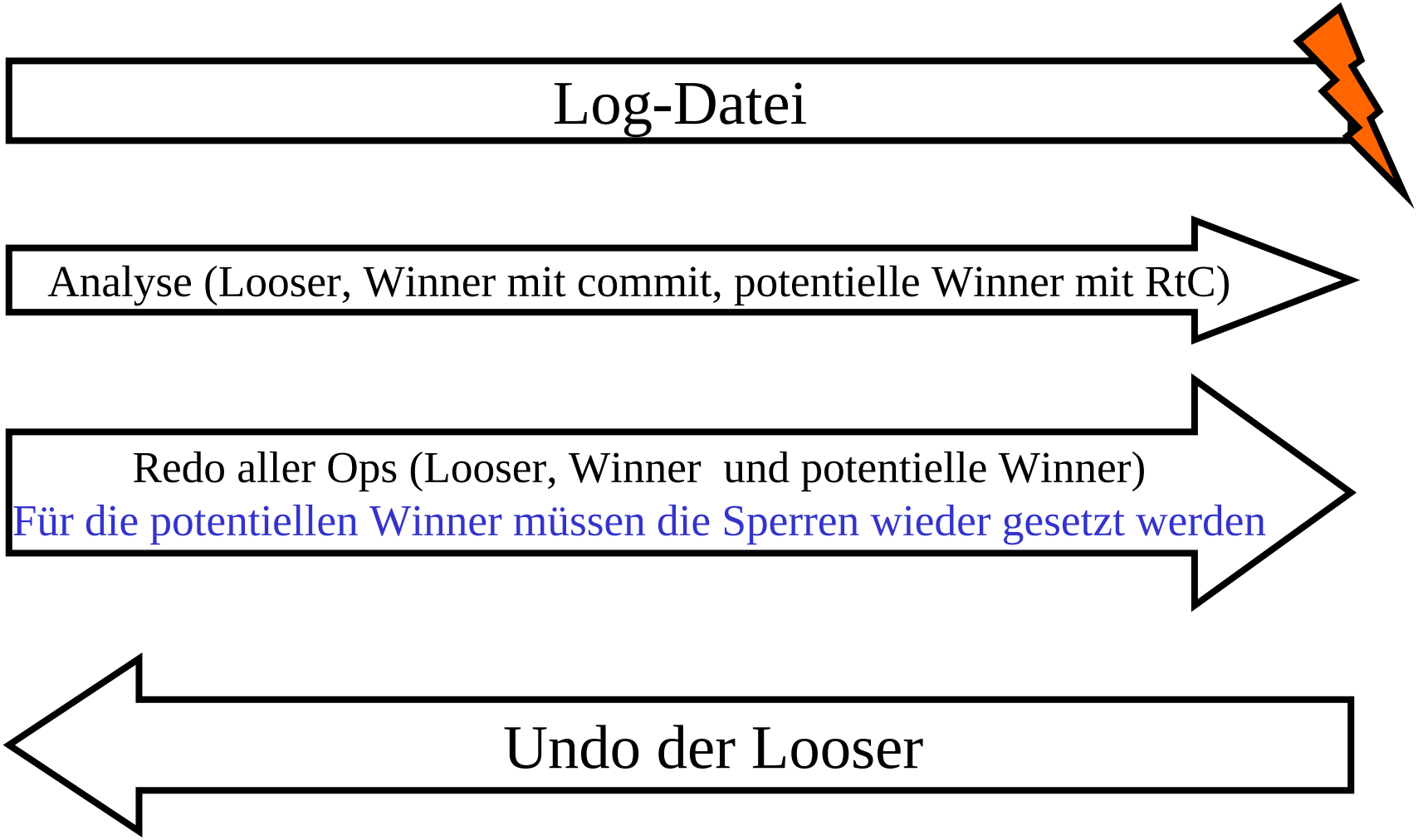
Write Ahead Log-Prinzip

1. Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle „zu ihr gehörenden“ Log-Einträge ausgeschrieben werden.

2. Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in das temporäre und das Log-Archiv ausgeschrieben werden.

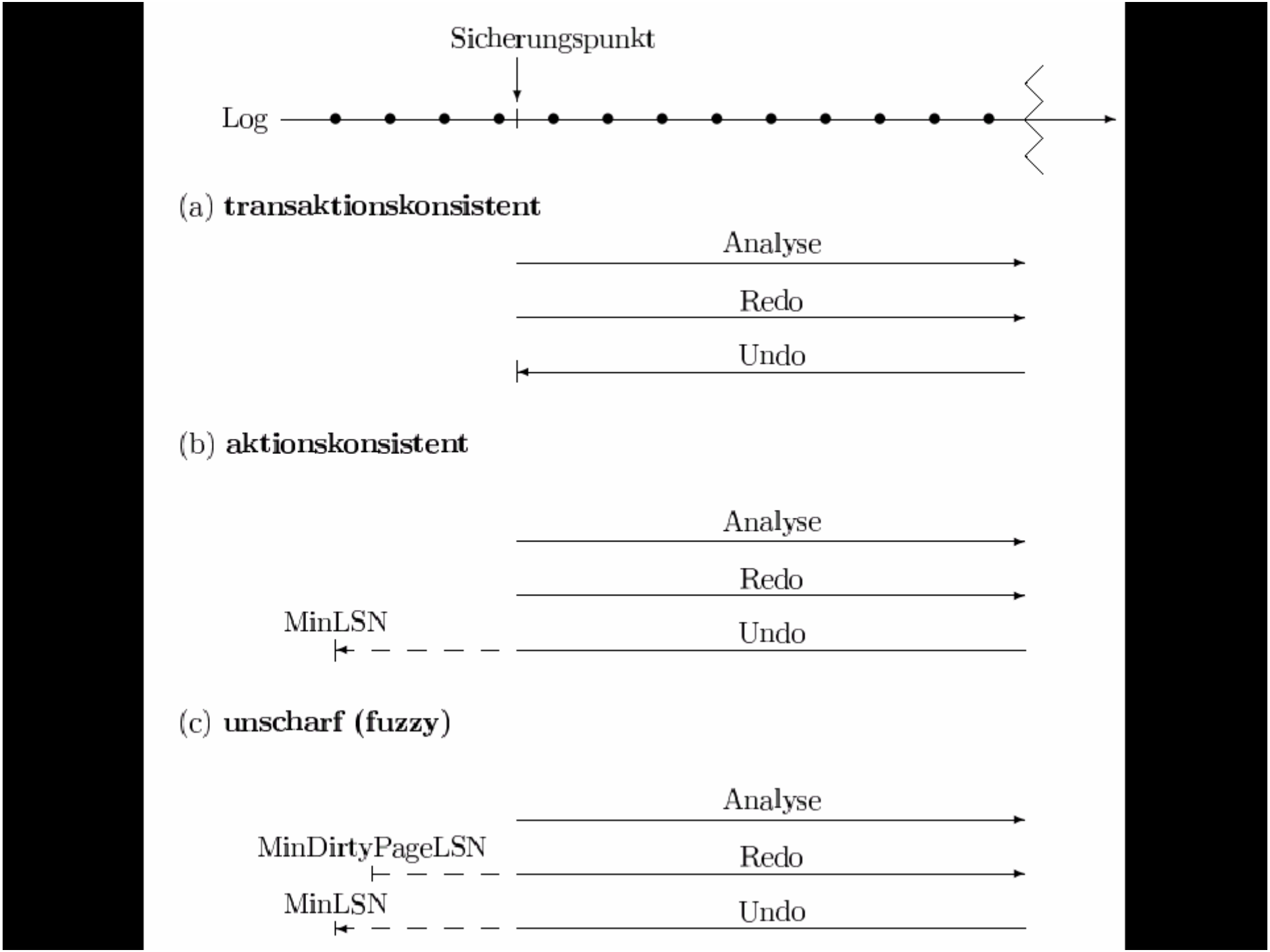
- Log-Record wird für jede Änderungsoperation geschrieben
  - before-image für das „Undo“
  - after-image für das Redo
- Eine verteilte Transaktion schreibt vor dem Verschicken der READY (RtC) Nachricht
  - alle Log-Records ihrer Änderungen ins Archiv (Platte)
  - Es muß aus dem Log hervorgehen, welche Daten gesperrt wurden
  - Die RtC- Entscheidung wird ins Log geschrieben
- Der Koordinator schreibt seine Entscheidung (commit oder abort) ins Log

Wiederanlauf in drei Phasen



Sicherungspunkte

- Dienen dazu, den Wiederanlauf zu beschleunigen
- Global konsistente Sicherungspunkte würden verlangen
  - Gesamtes System muß ruhig gestellt werden
  - Alle Stationen schreiben dann synchron ihren transaktionskonsistenten Sicherungspunkt
  - Nicht praktikabel
- Lokal transaktionskonsistente Sicherungspunkte
  - Auch nicht praktikabel
  - 7 mal 24 Stunden Betrieb wird heute angestrebt
- Nicht-transaktions-konsistente Sicherungspunkte
  - Lokal aktionskonsistente Sicherungspunkte
  - Lokal unscharfe Sicherungspunkte
  - I.w. wie in zentralen DBMS
  - Siehe Einf. Informationssysteme


Sicherungspunkt
Log
(a) transaktionskonsistent
Analyse
Redo
Undo
(b) aktionskonsistent
Analyse
Redo
MinLSN
Undo
(c) unscharf (fuzzy)
Analyse
MinDirtyPageLSN
Redo
MinLSN
Undo