Motivation

Als Konsequenz der Replikation. . .

. . . verwalten verteilte Datenbanksysteme mehrere Kopien des gleichen Datums.

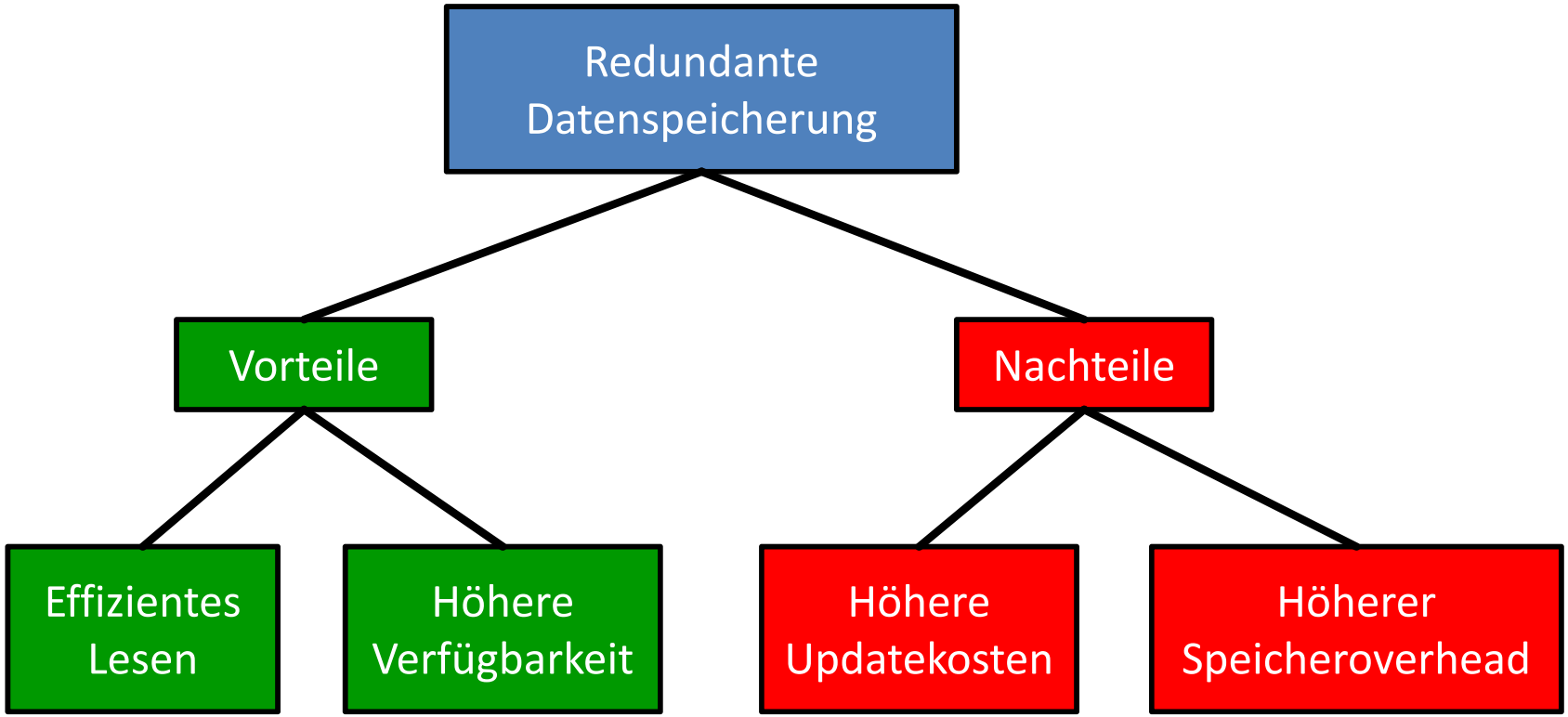
Problemszenario – Buchen eines Flugs nach Hawaii

- Alice möchte einen Flug nach Hawaii buchen
- Sie liest eine Kopie der Datenbank, findet den letzten noch freien Sitz, und bucht ihn sofort
- Bob möchte ebenfalls einen Flug nach Hawaii buchen
- Er liest eine andere Kopie der Datenbank (noch nicht geändert), findet ebenfalls den letzten freien Sitz, und versucht ihn zu buchen

Als Folgerung aus diesen Problemen. . .

. . . müssen Kopien synchron gehalten werden, auch wenn sie nicht am gleichen Ort gespeichert werden.

Motivation


Redundante Datenspeicherung
Vorteile
Nachteile
Effizientes Lesen
Höhere Verfügbarkeit
Höhere Updatekosten
Höherer Speicheroverhead

Motivation

Probleme und Aufgaben

- Propagieren von Änderungen
  Verteile Änderungen an alle Kopien
- Concurrency control
  Vermeide Probleme durch parallele Zugriffe verschiedener Anwendungen
- Lesender Zugriff
  Wähle eine Kopie, die gelesen wird (Aktualität vs. Nähe)
- Fehlererkennung und -behandlung
  kümmert sich um Netzfehler und -partitionierung

Ziele

- Replikationstransparenz
  Alle Kopien werden automatisch durch das verteilte Datenbanksystem geändert
- Verhalten äquivalent zu einer nicht-verteilten Datenbank (Ein-Kopien-Serialisierbarkeit)
  - Erfordert Konsistenz aller Kopien
  - Vor allem bei der Ausführung von Transaktionen:
    Ein-Kopien-Serialisierbarkeit
    Ausführung ist äquivalent zu serieller Ausführung in einer nicht-redundanten zentralen Datenbank

Motivation

Zwei grundlegende Fragen, mit denen sich Updatestrategien befassen müssen

- Wann sollen Änderungen propagiert werden?
  Synchron vs. asynchron
- Wo wird die Änderung ausgeführt?
  Überall vs. auf ausgewählten Kopien

Zwei grundlegende Familien von Änderungsstrategien

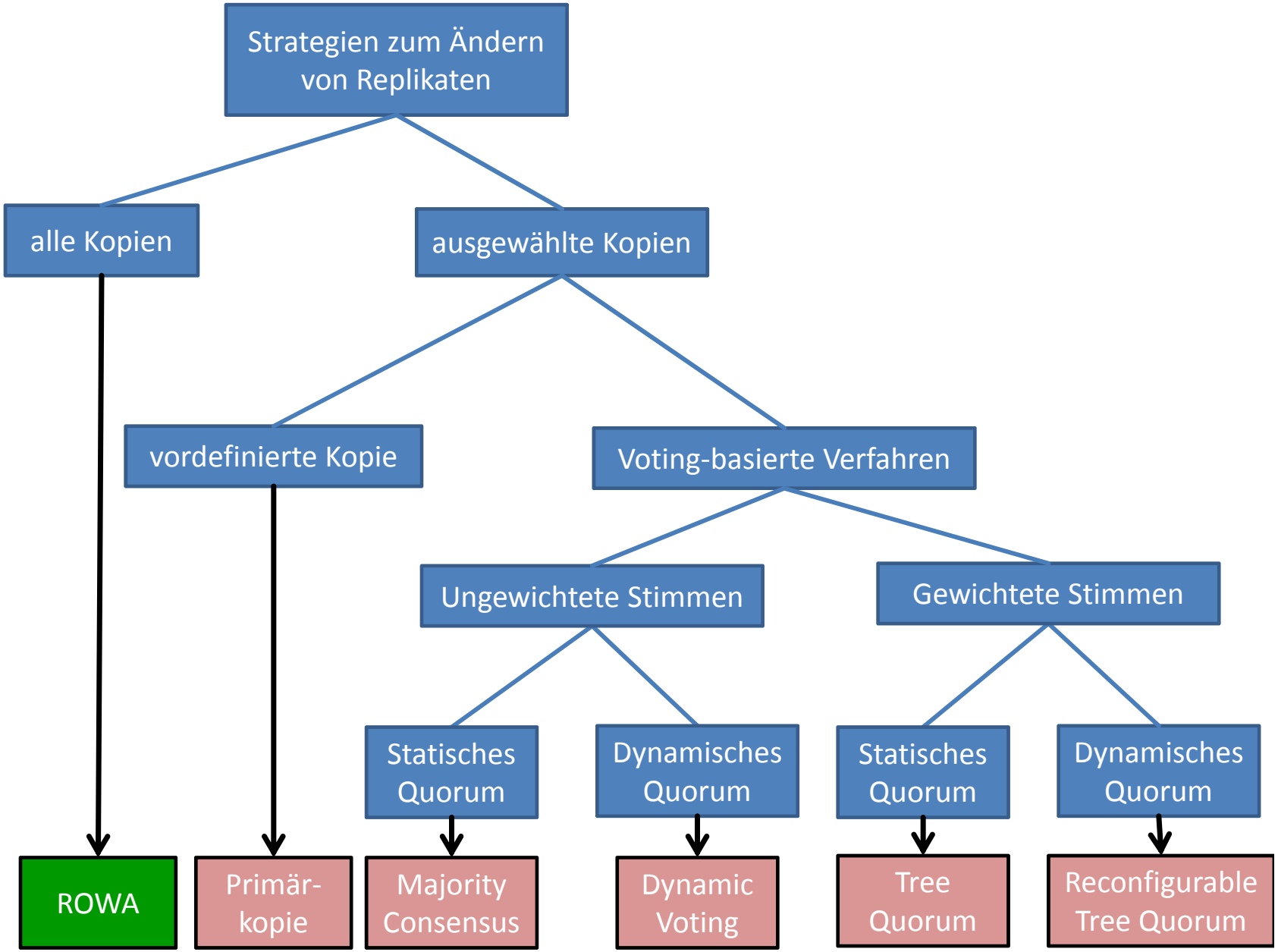
- Pessimistische Strategien: 1-copy-Konsistenz
  Das System verhält sich, als gäbe es nur eine Kopie der Daten
- Optimistische Strategien: Inkonsistenzen sind möglich
  Erlaubt, dass Kopien divergieren, und erfordert daher Konfliktauflösung

Universität Trier

Konservative Strategien

Klassifikation

Klassifikation [Dadam, 1996]



Read One Write All

Read One Write All (ROWA)

Leseoperation

- Eine logische Leseoperation wird in eine physische Leseoperation einer beliebigen Kopie umgewandelt (**read one**)

Schreiboperation

- Eine logische Schreiboperation wird umgewandelt in physische Schreiboperationen auf allen Kopien (**write all**)

Dies führt zu synchronen Updates aller Kopien

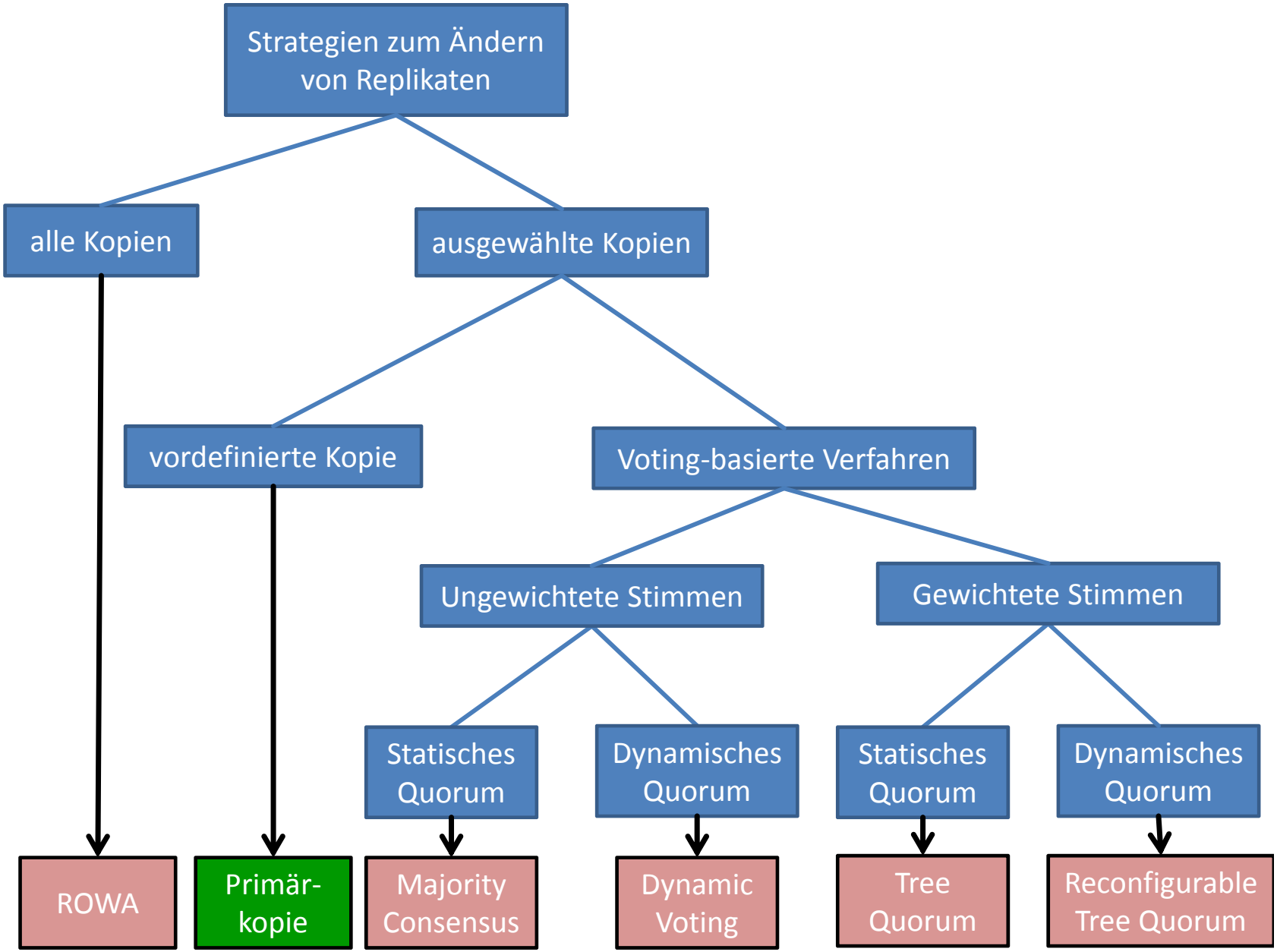
Read One Write All (ROWA)

Vorteile und Nachteile

- Sehr einfache Methode
- Leicht zu implementieren
- Jede Kopie ist zu jeder Zeit aktuell
- Effizienter lokaler Lesezugriff auf jedem Rechner
- Änderungsoperationen erfordern die Verfügbarkeit aller Rechner, die eine Kopie haben
- Updates benötigen recht viel Zeit

Primärkopie

Klassifikation [Dadam, 1996]



Primärkopie

Prinzip

- Bestimme eine Kopie als **Primärkopie** (Referenzversion)
- Alle anderen Kopien werden von der Primärkopie abgeleitet
- Die Ausführung einer Änderungsoperation erfordert es, die Primärkopie zu sperren
- Leseoperationen auf einem Rechner können effizient auf der lokalen Kopie ausgeführt werden

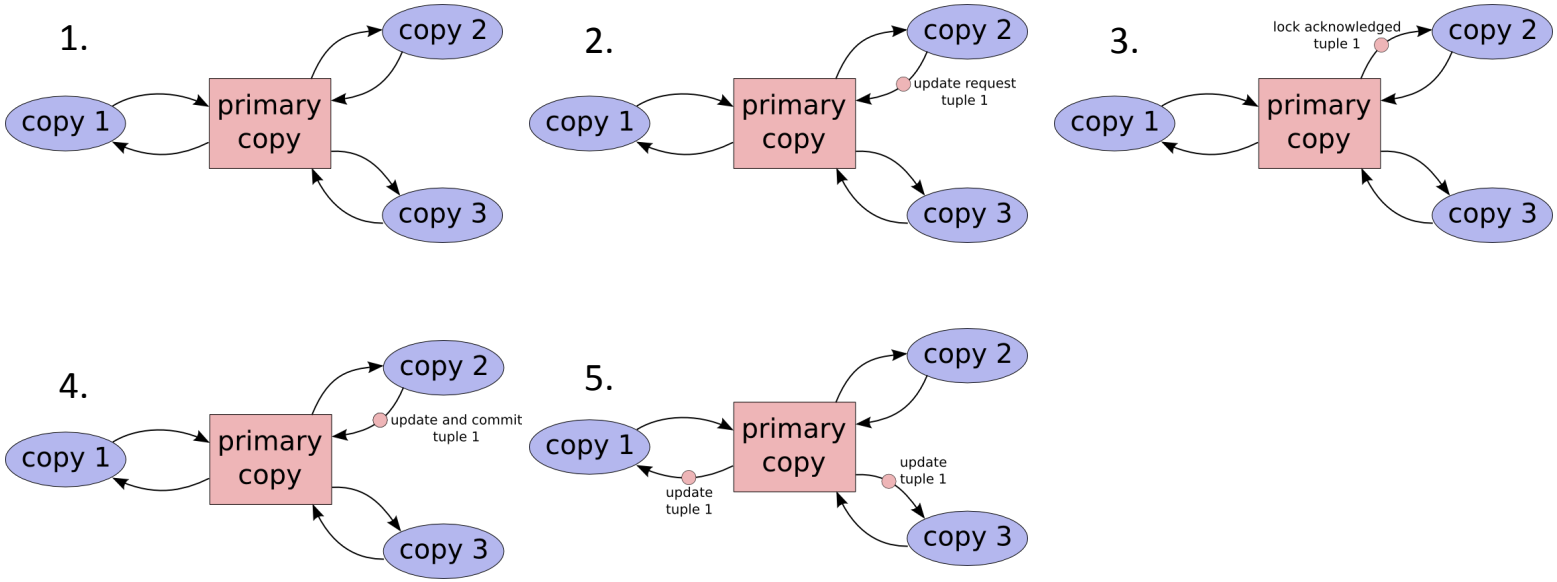
Workflow

- Die Änderungsoperation sperrt und ändert die Primärkopie
- Die Primärkopie propagiert die Änderung *asynchron* an alle Kopien

Konsistenz von Änderungen

Gewährleisten der Konsistenz bei parallen Änderungsoperationen auf dem gleichen Datenobjekt

- Logische Verbindung der Primärkopie mit den anderen Kopien durch einen FIFO-Kommunikationskanal



Dies garantiert, dass Änderungen immer in der richtigen Reihenfolge auf alle Kopien angewendet werden.

Lesekonsistenz

Gewährleisten der Konsistenz von Leseoperationen: Wie können wir gewährleisten, dass eine Leseoperation immer den aktuellen Wert liest?

- Unmöglich für den Basisalgorithmus!
  Weil wir nicht entscheiden können, ob alle relevanten Änderungen schon ausgeführt wurden

Mögliche Lösung

- Verwenden von Lesesperren mit besonderen Bestätigungsnachrichten
  - Erwerbe eine Lesesperre auf dem Datum, das gelesen werden soll
  - Warte auf die Bestätigungsnachricht
    Enthält Informationen, ob alle Änderungen bereits auf die Kopie, die gelesen werden soll, angewendet wurden
- Wesentlicher Nachteil: Der Vorteil, dass Leseoperationen lokal ausgeführt werden können ohne mit anderen Rechnern kommunizieren zu müssen, ist verloren
- Praktikable Lösung: **Schnappschuss-Semantik**
  Anwendungen wissen, dass sie nicht die aktuellsten Daten lesen könnten

Ausfall von Rechnern

Ausfall der Primärkopie

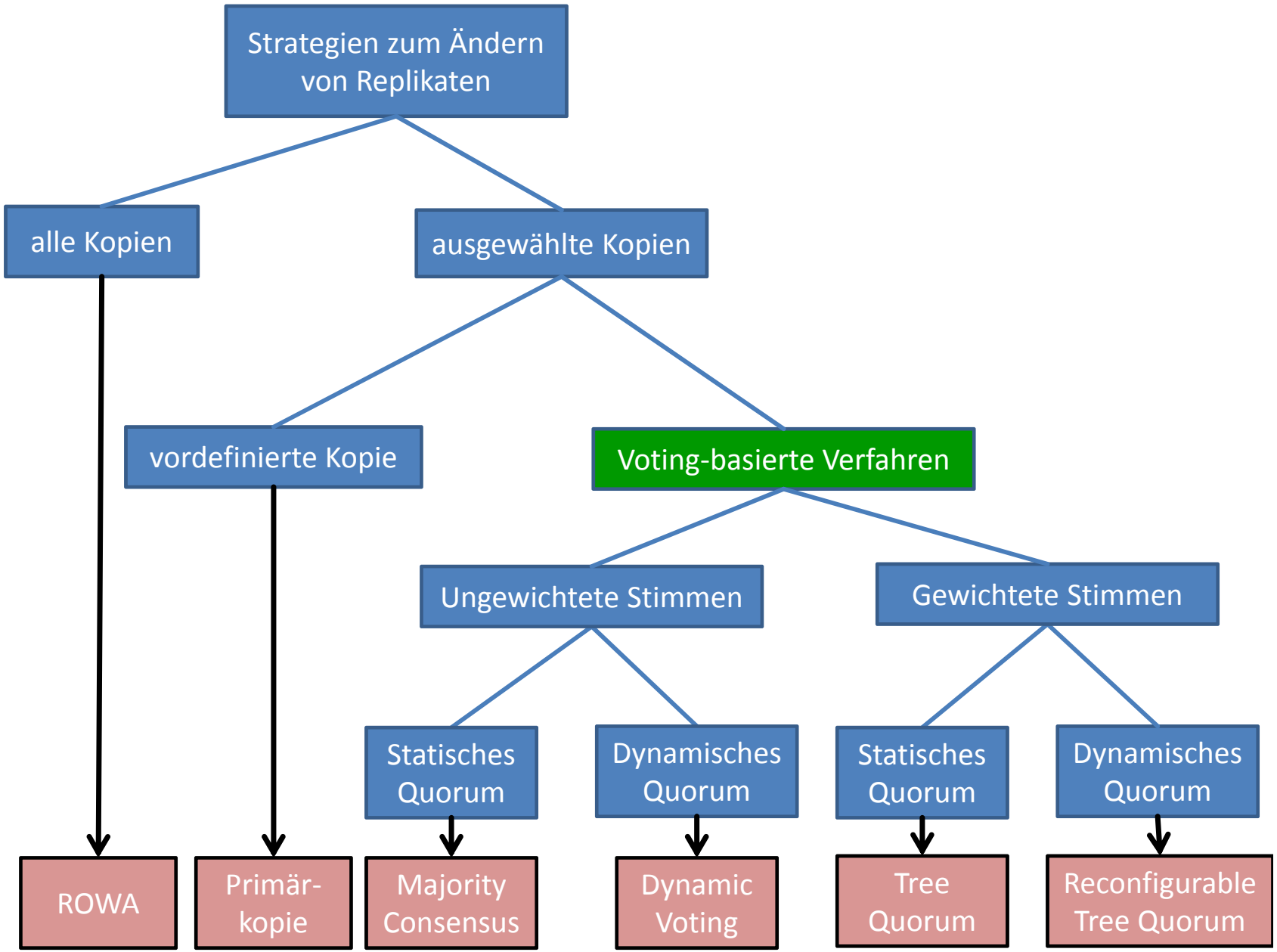
- Die Kopie auf dem Rechner mit der höchsten ID wird die neue Primärkopie

Partitionierung des Netzes

- Das Netz wird in zwei oder mehr Subnetze partitioniert, die nicht mehr miteinander kommunizieren können
- Durch die Anwendung der obigen Strategie können beide Subnetze weiter Daten verändern
  Aber: Wenn die beiden Subnetze wieder zusammengeführt werden, müssen die Daten konsolidiert werden

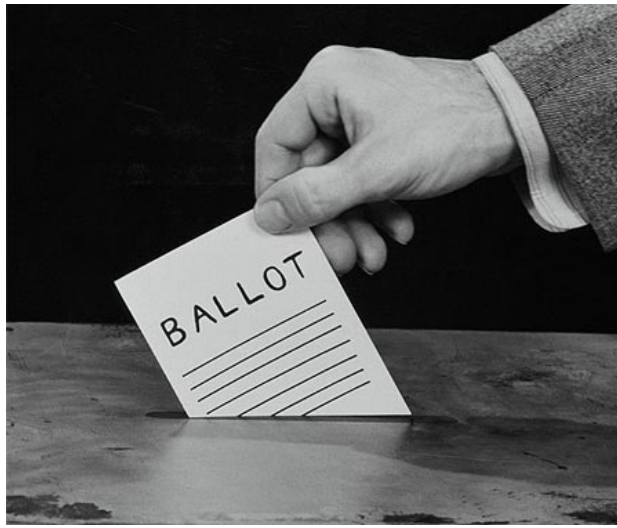
Voting-Strategien

Klassifikation [Dadam, 1996]



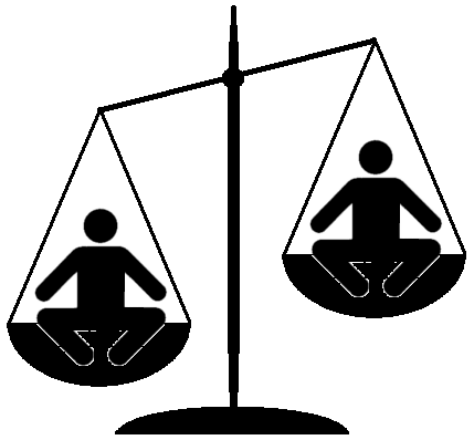
Grundlegende Idee

- Erlaube die Änderung einer Kopie, wenn die Mehrheit der Rechner mit Kopien zustimmt
- Stimmen (Votes)
  - Gesamtzahl der Stimmen: Q
  - Anzahl der Stimmen, die notwendig zum Lesen sind:
    Lesequorum Q_R
  - Anzahl der Stimmen, die notwendig zum Schreiben sind:
    Updatequorum Q_U
- Überlappungsregel
  - Stellt Ein-Kopien-Serialisierbarkeit sicher
  - $Q_R + Q_U > Q$
  - $Q_U + Q_U > Q$



Rechnergewichte für die Abstimmung

- Ungewichtete Stimmen
  - Alle Rechner/Kopien haben das gleiche Gewicht für die Abstimmung
  - Rechner werden in einer beliebigen Reihenfolge gefragt
- Gewichtete Stimmen
  - Rechner/Kopien haben verschiedene Gewichte für die Abstimmung
  - Rechner werden gemäß ihrer Gewichte gefragt (beginnend mit dem Rechner mit dem höchsten Gewicht)
  - Reduktion des Kommunikationsaufwandes
  - Kompensation von Stimmen mit hohem Gewicht durch mehrere Stimmen mit niedrigem Gewicht



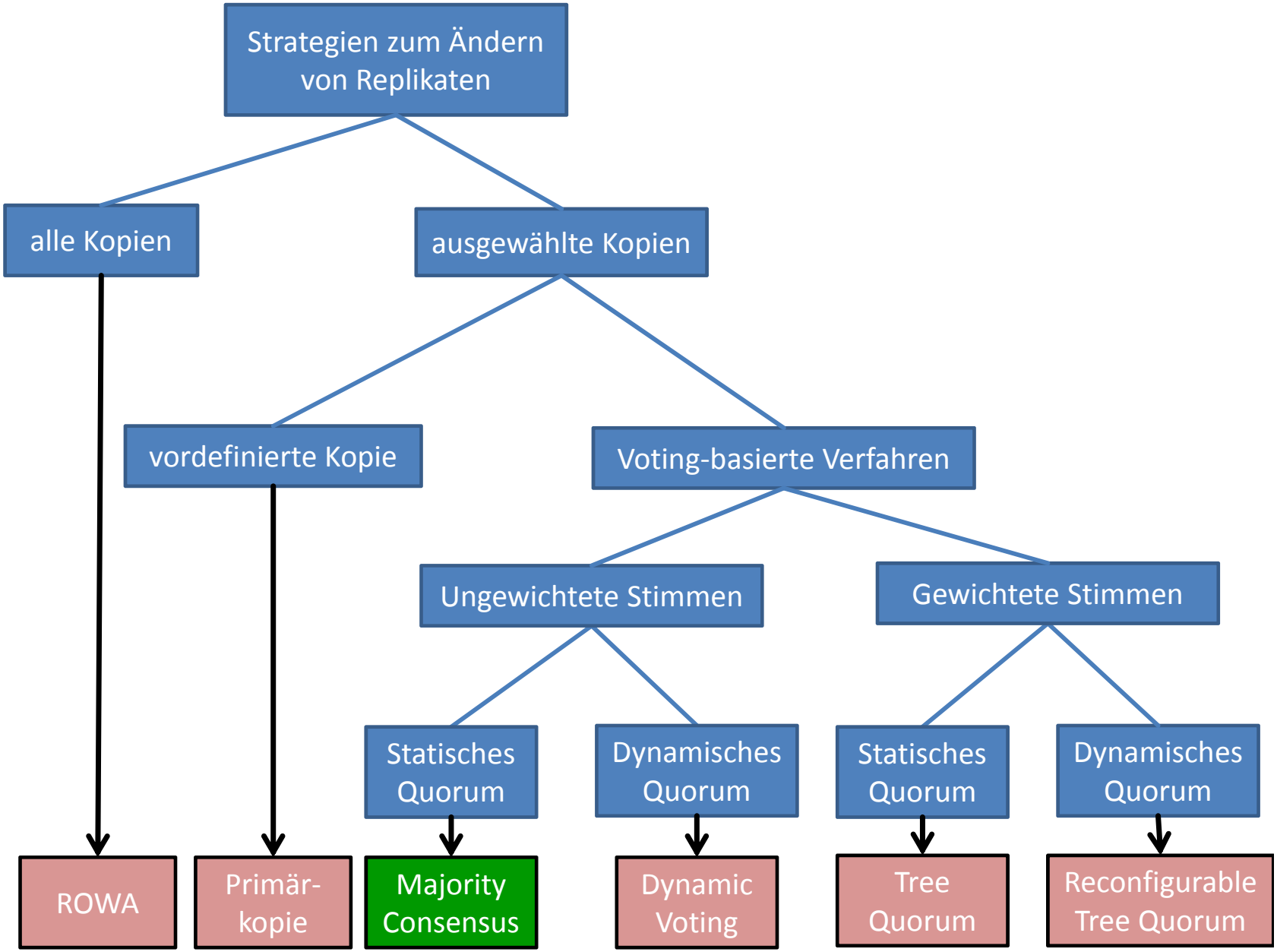
Voting-Strategien

Anzahl der notwendigen Stimmen

- Statisch
  - Definition der Quoren beim Systemstart
  - Wenn Rechner ausfallen, könnten die notwendigen Quoren nicht erreicht werden
- Dynamisch
  - Die Quoren werden an die Verfügbarkeit von Kopien/Rechnern angepasst

Majority consensus

Klassifikation



Majority consensus [Thomas, 1979]

Annahmen

- Ursprünglich entwickelt für Szenarien mit voller Replikation
  d.h. alle Relationen werden auf jedem Rechner gespeichert
- Jede Kopie hat das gleiche Gewicht
- Jedes lokale Datenbankobjekt x (z.B. Tupel) hat einen eigenen Zeitstempel $ts(x)$
  Er repräsentiert den Zeitpunkt des letzten erfolgreichen Updates
- Die Rechner kommunizieren über einen logischen Ring, der für jede Änderung neu aufgebaut wird
  Entscheidungen werden entlang dieses Rings übertragen

Entwurfsziele

- Das System soll mit Rechnerausfällen und Netzpartitionierung umgehen können
- Konsistenz von Kopien ohne die Anwendung globaler Sperrmechanismen
- Mehrheitsentscheidung im Fall von Zugriffskonflikten

Änderungen

Workflow

- Wende die Änderungen zunächst auf dem Initiatorrechner an (unsichtbar für die anderen)
- Erstelle eine Liste aller gelesenen Objekte I mit ihren Zeitstempeln ts_R und aller geschriebenen Objekte O, wobei $I \supseteq O$
- Schicke diese Liste zusammen mit dem aktuellen Zeitstempel t an alle anderen Rechner über den logischen Ring (*Update-Nachricht*)
- Mache die Änderungen permanent, wenn die Mehrheit der Rechner ($> n/2$) zustimmt

Abstimmen

Wenn ein Rechner eine Update-Nachricht $R = (I, O, ts_R, t)$ erhält, schickt er seine eigene Stimme zusammen mit den bisherigen Stimmen für diese Nachricht an einen Rechner, der noch nicht abgestimmt hat (so wird der logische Ring aufgebaut)

Zwei Update-Nachrichten R_1 und R_2 sind *in Konflikt*, wenn $I(R_1) \cap O(R_2) \neq \emptyset$ oder umgekehrt, d.h. wenn einer der Requests Informationen ändert, die der andere gelesen hat.

Mögliche Stimmen

- **reject**:
  - falls einer der erhaltenen Zeitstempel der gelesenen Objekte veraltet ist, d.h. wenn $\exists i \in I : ts(i) > ts_R(i)$
- **okay** und markiere den Request als **offen**:
  - Wenn alle erhaltenen Zeitstempel der gelesenen Objekte aktuell sind, d.h. $\forall i \in I : ts(i) = ts_R(i)$

  und
  - Die Änderungsnachricht nicht mit einer anderen offenen Änderungsnachricht in Konflikt ist
- **pass**:
  - Wenn alle erhaltenen Zeitstempel von gelesenen Objekten aktuell sind, aber die Änderungsnachricht in Konflikt ist mit einer anderen offenen Nachricht $R' = (I', O', t', ts'_R)$ mit einem höheren Zeitstempel, d.h. $t' > t$
    Wenn die Nachricht R' akzeptiert wird, wird die Nachricht R keinen Erfolg haben können
  - Wenn das Quorum nicht mehr erreicht werden kann, wenn mit pass gestimmt wird, dann stimme mit reject

Die Nachricht wird **verzögert**, also ohne weitere Aktionen auf dem Rechner zwischengespeichert,

- falls die Nachricht in Konflikt ist mit einer anderen offenen Nachricht mit kleinerem Zeitstempel, oder
- falls einer der erhaltenen Zeitstempel der gelesenen Objekte neuer ist als der lokale Zeitstempel, d.h. $\exists i \in I : ts(i) < ts_R(i)$
  In diesem Fall hat die lokale Datenbank eine bereits akzeptierte Änderung noch nicht erhalten

Annehmen oder Verwerfen einer Änderung

Treffen der finalen Entscheidung

- Der Rechner, dessen **okay** zum Erreichen des Quorums führt, erzeugt eine **globale Annahme-Nachricht** für die Änderung
- Ein Rechner, der mit **reject** abstimmt, ruft eine **globale Ablehnung** hervor und erzeugt eine entsprechende Ablehnungsnachricht
- Wenn die Änderung **akzeptiert** wird, aktualisieren Rechner ihre lokale Kopie, wenn sie die globale Annahme-Nachricht erhalten
  Für alle verzögerten Nachrichten, die in Konflikt mit dieser Nachricht sind, wird mit **reject** gestimmt
- Wenn die Änderung **abgelehnt** wird, prüfen alle Rechner, ob sie Stimmen jetzt abgeben können, die zuvor wegen eines Konflikts mit dieser Änderung verzögert wurden

Wenn die Änderung abgelehnt wird, wird die vollständige Transaktion einschließlich aller Leseoperationen wiederholt

Bemerkungen

Bemerkungen

- Wenn ein Rechner einmal gestimmt hat, darf er seine Stimme nicht mehr verändern
- Rechner, die ihre Stimmabgabe verzögern, geben die Änderungsnachricht nicht weiter
- Akzeptierte Änderungen können sich “überholen”
  Lösung: Prüfe Zeitstempel und ignoriere veraltete Änderungen

Lesekonsistenz

- Kann nur garantiert werden, wenn Leseoperationen wie Pseudo-Änderungen behandelt werden

Ausfall von Rechnern

Ausfall von Rechnern

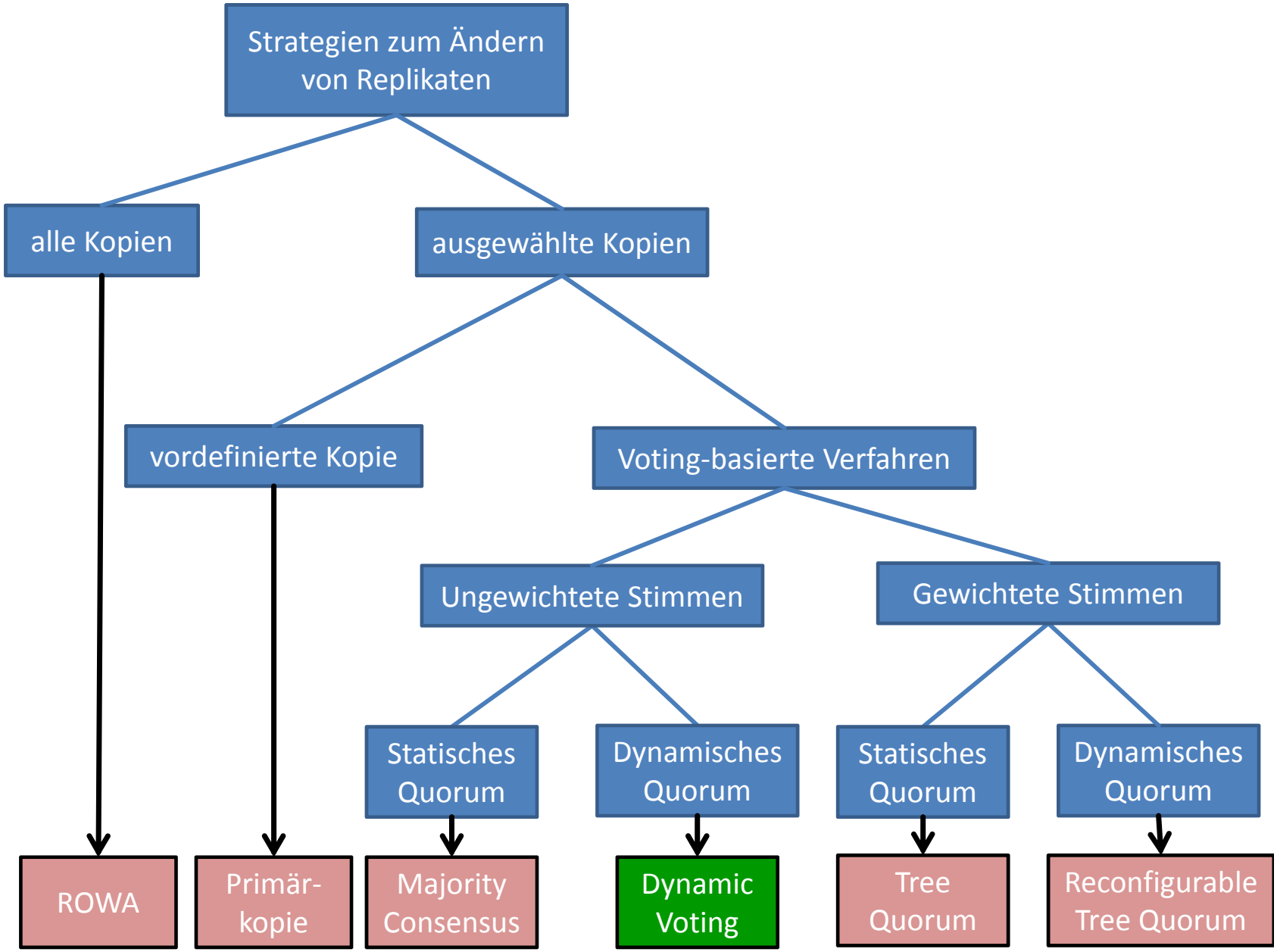
- Wenn nur einige Rechner ausfallen, können die übrigen immer noch über Annahme oder Ablehnung entscheiden
- Wenn so viele Rechner ausfallen, dass das Quorum nicht mehr erreicht werden kann, kann keine Entscheidung getroffen werden

Partitionierung des Netzes

- So lange eine Partition genügend Stimmen hat (Mehrheit des ursprünglichen Netzes), kann sie eine Entscheidung treffen
- Im Fall von Rechnerausfällen oder Netzpartitionierung kann eine noch nicht endgültig entschiedene Änderung nach einem Timeout durch den Initiator erneut verschickt werden
  In diesem Fall führt eine **reject**-Stimme nicht sofort zur Ablehnung der Änderung, damit zwei Partitionen nicht zu verschiedenen Entscheidungen kommen können

Dynamic voting

Klassifikation



Grundlegende Idee

Problem mit Majority Consensus

- Wenn es zu Rechnerausfällen und Netzpartitionierung kommt, kann es unmöglich sein, Stimmen von der Mehrheit der Rechner zu sammeln

Grundlegende Idee

- Prinzip: Majority Consensus mit dynamischem Quorum
- Es dürfen nur die Rechner abstimmen, die aktuelle Information haben, d.h. die letzte Änderung verarbeitet haben

Zusätzliche Information

Zusätzliche Information muss gesammelt werden

- **Versionsnummer (VN)**: Version des Datenbankobjektes
- **Anzahl der Rechner (NC)** die an der letzten erfolgreichen Abstimmung teilgenommen haben
  Entspricht der Gesamtzahl der aktuellen Stimmen

Beispiel

- Initiale Situation:
  - 6 Änderungen wurden auf allen Kopien ausgeführt (VN = 6)
  - 5 Kopien, d.h. 5 Rechner (NC = 5)

Kopie:	1	2	3	4	5
VN:	6	6	6	6	6
NC:	5	5	5	5	5

Beispiel

Eine erste Partitionierung des Netzes resultiert in $\{1, 2\}$ und $\{3, 4, 5\}$

- Änderungsnachricht geht von Rechner 3 aus
  Rechner 3 gehört zu Partition $\{3, 4, 5\}$
  Quorum erreichbar, da $3 > NC/2 = 5/2$
  Quorum nicht erreichbar in Partition $\{1, 2\}$
- Änderung akzeptiert (Rechner 4 und 5 stimmen mit 'okay' ab)
  Rechner 3, 4 und 5 führen die Änderung aus
  Neue Versionsnummer VN = 7 und NC = 3

Kopie:	1	2	3	4	5
VN:	6	6	6	6	6
NC:	5	5	5	5	5

Kopie:	1	2	3	4	5
VN:	6	6	7	7	7
NC:	5	5	3	3	3

Beispiel

Zweite Netzpartitionierung resultiert in $\{1,2\}$, $\{3\}$ und $\{4,5\}$

- Änderungsnachricht geht von Rechner 4 aus
  Rechner 4 gehört zu Partition $\{4,5\}$
  Quorum erreichbar, da $2 > NC/2 = 3/2$
  Quorum nicht erreichbar in den Partitionen $\{1,2\}$ *und* $\{3\}$
- Änderung akzeptiert (Rechner 5 stimmt mit 'okay')
  Rechner 4 und 5 führen die Änderung aus
  Neue Versionsnummer VN = 8 und NC = 2

Kopie:	1	2	3	4	5
VN:	6	6	7	7	7
NC:	5	5	3	3	3

Kopie:	1	2	3	4	5
VN:	6	6	7	8	8
NC:	5	5	3	2	2

Majority Consensus würde diese Änderung nicht akzeptieren, da es immer noch mindestens drei positive Stimmen erfordern würde (ohne Anpassung von NC).

Beispiel

Ende der zweiten Netzpartitionerung, resultierend in $\{1, 2\}$, $\{3, 4, 5\}$

- Änderungsnachricht, die von Rechner 5 ausgeht
  Rechner 5 gehört zu Partition $\{3, 4, 5\}$
  Rechner 5 fragt alle verfügbaren Rechner
  Rechner 3 und 4 antworten
  Rechner 4 hat das Recht zur Abstimmung, da VN und NC aktuell sind
  Quorum erreichbar, da $2 > NC/2 = 2/2$
- Änderung wird akzeptiert (Rechner 4 stimmt mit 'okay')
  Rechner 3, 4 und 5 führen die Änderung aus
  Neue Versionsnummer VN = 9 und NC = 3

Kopie:	1	2	3	4	5
VN:	6	6	7	8	8
NC:	5	5	3	2	2

Kopie:	1	2	3	4	5
VN:	6	6	9	9	9
NC:	5	5	3	3	3

Beispiel

Ende der ersten Netzpartitionerung resultiert in $\{1, 2, 3, 4, 5\}$

- Änderungsnachricht, die von Rechner 3 ausgeht
  Rechner 3 fragt alle erreichbaren Rechner
  Rechner 1, 2, 4 und 5 antworten
  Rechner 4 und 5 haben das Recht abzustimmen, weil VN und NC aktuell sind
- Änderung wird akzeptiert (Rechner 4 und 5 stimmen mit 'okay')
  Rechner 1, 2, 3, 4 und 5 führen die Änderung aus
  Neue Versionsnummer VN = 10 und NC = 5

Kopie:	1	2	3	4	5
VN:	6	6	9	9	9
NC:	5	5	3	3	3

Kopie:	1	2	3	4	5
VN:	10	10	10	10	10
NC:	5	5	5	5	5

Bemerkungen

Bemerkungen

- Ein Rechner, der eine Partition/ein Netz wieder betritt, kann das Stimmrecht zurückerhalten, wenn er eine leere Änderungsnachricht an die anderen Rechner in der Partition bzw. im Netz schickt

Lesekonsistenz

- Kann nur garantiert werden, wenn Leseoperationen als Pseudo-Änderungsoperationen behandelt werden

Progessive Strategien

Konservative vs. progressive Strategien

Konservativ

- Priorität:
  Sicherstellen von Konsistenz
- Nach einer Netzpartitionierung darf nur eine Partition Änderungen ausführen
- Nachteile: Leistungsverlust, zeitweise Nichtverfügbarkeit

Progressiv

- Priorität:
  Sicherstellen von Verfügbarkeit
- Nach einer Netzpartitionierung dürfen alle Partitionen Änderungen vornehmen
- Nachteil: zeitweise Inkonsistenzen, die aufgelöst werden müssen, wenn das Netz nicht mehr partitioniert ist

Alle Verfahren, die wir bisher betrachtet haben, sind konservative Strategien.

Data patches

Data patches [Garcia-Molina, 1983]

Grundlegende Idee

- Auch bei Partitionierung des Netzes dürfen alle Partitionen Änderungen ausführen (progressive Strategie)
- Die Strategie zum Auflösen von Konflikten wird bereits zur Entwurfszeit festgelegt

Zwei Klassen von Regeln

- Tupel-Einfüge-Regeln
  Für Tupel, die während der Partitionierung in eine Partition eingefügt wurden
- Tupel-Integrations-Regeln
  Für Tupel, die während der Partitionierung geändert wurden

Methode

- Für jede Relation wird eine Regel aus jeder Klasse bestimmt

Tupel-Einfüge-Regeln

Nehmen wir an, dass es zwei Partitionen P_1 und P_2 gibt und wir ein Tupel nur in P_1 eingefügt haben.

- **Keep** Regel
  Füge das Tupel in P_2 ein
- **Remove** Regel
  Entferne das Tupel aus P_1
- **Program** Regel
  Starte ein bestimmes Programm mit dem Tupel als Eingabe
- **Notify** Regel
  Informiere den Administrator (manuelle Konfliktauflösung)

Tupel-Integrations-Regeln

Nehmen wir an, dass es zwei Partitionen P_1 und P_2 gibt und Tupel mit den selben Primärschlüsselwerten in P_1 und/oder P_2 geändert wurden.

- **Latest** Regel
  Wähle das zuletzt geänderte Tupel
- **Primary** Regel
  Wähle das Tupel auf Rechner K (Primärkopie)
- **Arithmetic** Regel
  Berechne den neuen Wert als: new value = $value_1$ + $value_2$ - old value
- **Program** Regel
  Starte ein bestimmes Programm mit dem Tupel als Eingabe
- **Notify** Regel
  Informiere den Administrator (manuelle Konfliktauflösung)

Data patches

Workflow zur Vereinigung von Partitionen

- Bestimme alle Tupel, die nur in einer Partition verändert wurden, und wenden die Tupel-Einfüge-Regel an
- Bestimme alle Tupel, die in beiden Partitionen während der Partitionierung verändert wurden, und wende die Tupel-Integrations-Regel an

Lockern der Isolationsanforderungen

Snapshot replication

Snapshot replication [Abida et al., 1980]

Problem der konservativen Strategien

- Um alle Kopien aktuell zu halten (Synchronisation), ist ein hoher Kommunikationsaufwand notwendig

Grundlegende Idee

- Lockern der Isolationsanforderungen
- **Reine Lesezugriffe** werden niemals blockiert
- Ein Schnappschuss (Snapshot) entspricht einem **materialisierten View** auf einer Relation und kann als Anfrage spezifiziert werden
  Diese Anfragen können auch Filter oder Aggregationen enthalten
- Schnappschüsse werden regelmäßig aktualisiert (Refresh-Operation) – **asynchrone Synchronisation**
  Aktualisierungsfrequenz wird bestimmt, wenn der Snapshot angelegt wird

Nur-Lese- vs. änderbare Schnappschüsse

Zwei Arten von Schnappschüssen

- Nur-Lese-Schnappschuss
  - Änderungen sind nur auf dem Masterrechner möglich, d.h., der Rechner, der die Originalrelation hält (ähnlich wie bei Primary Copy)
- Änderbarer Schnappschuss
  - Änderungen können auch auf Schnappschüssen vorgenommen werden
  - Erfordert Konfliktauflösung auf dem Masterrechner

kann zusammen mit Primary Copy angewendet werden.

Abschließende Bemerkungen zur Konsistenz

Lesekonsistenz

Im Prinzip gibt es drei Fälle

- Gelesene Daten müssen immer aktuell und konsistent sein
  - Die gleichen Konsistenzanforderungen wie für Änderungen *erfordert die gleichen Methoden*
  - geringer Durchsatz
- Gelesene Daten müssen immer konsistent sein, aber dürfen leicht veraltet sein
  - erfordert Versionierung/Zeitstempel
  - Leseoperationen können alte Versionen von Daten lesen (z.B. aus einem Schnappschuss
  - Reduktion des Kommunikationsaufwandes
- Konsistenz muss nicht gewährleistet werden
  - Keine Versionierung notwendig
  - Definition eines ε, das die maximale Differenz zwischen den gelesenen Daten und den aktuellen Daten angibt (ε-Serialisierbarkeit); Beispiele für ε:
    - Anzahl der verpassten Änderungen
    - maximale zeitliche Differenz zur letzten Änderung
  - wegen möglicher Inkonsistenzen nur für wenige Anwendungen geeignet

Lesekonsistenz

Konsistenzstufen [Vogels, 2008]:

- Starke Konsistenz
  Wenn die Änderung abgeschlossen ist, wird jeder folgende Zugriff den geänderten Wert liefern.
- Schwache Konsistenz
  Inkonsistenz-Fenster: Zeit zwischen der Änderung und dem Zeitpunkt, wenn folgende Zugriffe garantiert den geänderten Wert liefern werden.
- Schließliche (eventual) Konsistenz
  Besondere Form der schwachen Konsistenz. Wenn keine weiteren Änderungen an dem Objekt vorgenommen werden, werden irgendwann alle Zugriffe den zuletzt geschriebenen Wert liefern.

Ausfälle von Rechnern

Vorhandensein einer Primärkopie

- Wenn die Primärkopie verfügbar ist, sind Änderungen möglich

Zusammen mit voting-basierten Verfahren

- keine besonderen Maßnahmen notwendig
- Solange die Lese- und Schreibquoren erreichbar sind, sind Lese- und Schreibzugriffe möglich

Netzpartitionierung

Konservative Ansätze versuchen, Konsistenz zu jeder Zeit zu gewährleisten

- Im Fall einer Partitionierung kann nur eine Partition Änderungen ausführen
- Die Hauptpartition wird durch die Primärkopie oder durch die Quoren bei voting-basierten Verfahren bestimmt

Progressive Ansätze versuchen, die Systemverfügbarkeit zu maximieren

- Änderungen sind in allen Partitionen möglich
  kann zu zeitweiligen Inkonsistenzen führen
- erfordert Konsolidierungsstrategien, wenn die Partitionen wieder vereint werden (z.B. Data Patches)

Proaktive Strategien

Epidemic replication

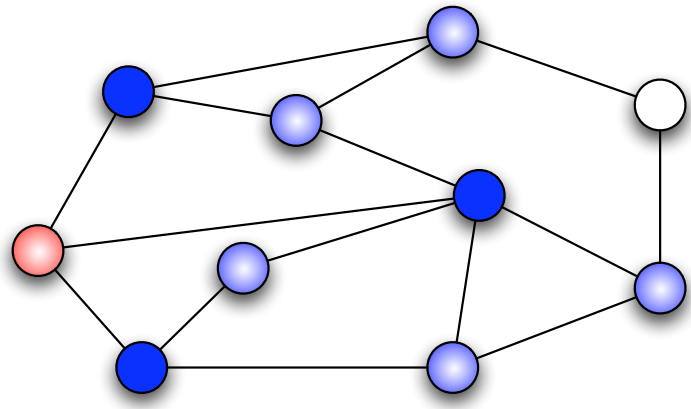
Epidemic replication

Nachteile der klassischen Strategien

- Eingeschränkte Skalierbarkeit und Verlässlichkeit im Fall einer großen Zahl von Rechnern, instabilen Verbindungen und Rechnern

Grundlegende Idee

- Wende das Peer-to-Peer-Modell der epidemischen Verteilung an
  - Änderungsoperationen sind auf allen Rechnern möglich
  - Alle Rechner verarbeiten Änderungen auf die gleiche Art
  - Änderungen werden an eine zufällige Menge von Nachbarn übertragen
- Proaktives Verhalten: Keine Fehlerbehandlung notwendig



Universität Trier

Epidemic replication

Workflow

- Führe die lokalen Änderungen aus
- Synchronisiere die Änderungen von Zeit zu Zeit mit anderen Kopien
- Paarweise Interaktion:
  - Vergleich von Versionen (z.B. auf Basis von Ereignislogs)
  - Auflösung von Konflikten
  - Propagieren der Änderungen

Ereignislogs

- Loggen der **Änderungshistorie einer Kopie**
  $h = \{op_1, \ldots, op_n\}$
- **Inkonsistenz** zwischen Kopien:
  Die Kopien x_A und x_B eines Objekts x sind in Konflikt, wenn x_A Operation op_i enthält, aber nicht op_k, und x_B Operation op_k enthält, aber nicht op_i
- Die Kopie x_A ist **älter** als Kopie x_B, wenn das Ereignislog von x_A ein echter Präfix des Ereignislogs von x_B ist.

Versionsvektoren

Gegeben eine Menge von Rechnern $(1, \ldots, n)$, die Kopien des Datenobjekts x haben

- x_i ist die Kopie von x auf Rechner i
- Jeder Rechner hat einen **Versionsvektor** $v_i(x)$ für seine lokale Kopie von x mit einem Eintrag $v_{ik}(x)$ für jeden Rechner k, der eine Kopie von x hat

Änderungsregeln

- Am Anfang sind alle Versionszähler 0
- Wenn eine Kopie **geändert** wird, **erhöht sie ihren eigenen Zähler** im Vektor um eins,
  d.h., wenn Rechner i Objekt x_i ändert: erhöhe den lokalen Eintrag $v_{ii}(x)$ des lokalen Versionsvektors
- Wenn zwei Kopien **synchronisiert werden**, wird ein neuer Versionsvektor für beide Kopien erstellt - er enthält das **paarweise Maximum** alle Zähler,
  d.h., beim Synchronisieren der Änderungen von Rechner k mit Rechner i:

$$v_{im}^{new}(x) = max(v_{im}^{old}, v_{km}) \quad (1 \leq m \leq n)$$

Vergleich von Versionsvektoren

Identität

- Zwei Versionsvektoren sind gleich, wenn **alle Einträge gleich sind**
  Das heißt, dass die zugehörigen Kopien identisch sind

Dominanz

- Beispiel: Gegeben zwei Kopien x_i, x_k, Rechner m, $v_{im} < v_{km}$, und $v_{km} - v_{im} = u$, dann hat x_i u **Änderungen weniger von Rechner** m gesehen als x_k
- x_i ist **älter** als x_k, falls
  - $v_i(x)$ komponentenweise $\leq v_k(x)$
  - wenigstens eine Komponente von $v_i(x)$ ist kleiner als die entsprechende Komponente von $v_k(x)$, in diesem Fall **dominiert** $v_k(x)$ über $v_i(x)$

Inkonsistenz

- x_i und x_k sind inkonsistent, wenn es m, l gibt, so dass $v_{im}(x) < v_{km}(x)$ und $v_{il}(x) > v_{kl}(x)$

Versionsvektoren

Problem

- Die Anzahl der Vergleiche wächst linear mit der Anzahl der Datenobjekte – Skalierbarkeitsproblem

Mögliche Lösung [Rabinovich et al., 1996]

- Datenbank-Versionsvektoren (DBVV) zusätzlich zu dem Item-Versionsvektoren (IVV)

DBVV und IVV

Datenbank-Versionsvektor (DBVV)

- Jeder Rechner i hat einen DBVV V_i für die Datenbank und IVVs für jedes lokal gespeicherte Datenobjekt
  $|V_i|$ gibt die Anzahl der Rechner mit Kopien der Datenbank an
- Initialisierung aller Komponenten von V_i mit 0

Änderungen

- Lokale Änderung auf Rechner i: $V_{ii}^{new} = V_{ii}^{old} + 1$
- Rechner k aktualisiert Rechner i in Bezug auf Datenobjekt x:

$$V_{il}^{new} = V_{il}^{old} + (v_{kl}(x) - v_{il}(x)), (1 \leq l \leq n)$$

  - $v_{il}(x)$ gibt die l-te Komponente des IVV von Objekt x auf Rechner i an
  - x_i hat $v_{il}(x)$ Änderungen von Rechner l gesehen, x_k hat $v_{kl}(x)$ Änderungen von Rechner l gesehen
  - Wenn x_k aktueller ist, d.h. $v_{il}(x) \leq v_{kl}(x)$, dann wird x von Rechner k an Rechner i propagiert
    Das führt zu $v_{kl}(x) - v_{il}(x)$ zusätzlichen Änderungen, die zum DBVV von Rechner i hinzugefügt werden müssen

Logging

- Logvektor L_i von Änderungen, verwaltet auf Rechner i
  - Komponente L_{ik}: Änderungen erhalten von Rechner k
  - Logeintrag (x, m): x ist ein Datenobjekt, m ist der Wert von V_{kk} von Rechner k zur Zeit der Änderung $\rightarrow$ "Sequenznummer" des Updates auf Rechner k
  - Nur die letzte Änderung jedes Objektes wird geloggt
- Gesamtzahl der Einträge: $\leq n \cdot N$ mit n Rechnern und N Datenobjekten

Propagieren von Änderungen (Rechner i wird durch Rechner k geändert)

1. Rechner i sendet V_i an Rechner k
2. Rechner k vergleicht V_i und V_k und schickt fehlende Änderungen an Rechner i
   - Wenn V_i gleich V_k ist oder es dominiert:
     Keine Änderungen auf Rechner i notwendig
   - Andernfalls: erzeuge "Tailvektor" D
     m-te Komponente enthält (i) Änderungen von Rechner m, die noch nicht auf Rechner i existieren, und (ii) eine Liste S mit allen betroffenen Datenbankobjekten und ihren IVVs, d.h.

     `for` $m = 1$ `to` n
       `if` $V_{km} > V_{im}$ `then`
         $D_m =$ Suffix des Logvektors L_{km} mit Einträgen (x, p)
           so dass $p > V_{im}$
   - Sende S und D an Rechner i

Propagieren von Änderungen (Rechner i wird durch Rechner k geändert)

3. Rechner i vergleicht alle Objekte x in S mit seiner lokalen Kopie, d.h. er vergleicht $v_k(x)$ und $v_i(x)$
   - Wenn $v_k(x)$ $v_i(x)$ dominiert: Ändere Objekt x und V_i
   - Andernfalls: Inkonsistenz, d.h. löse einen Alarm aus und enferne betroffene Logeinträge aus D
   - Füge die Logeinträge aus D zum lokalen Logvektor hinzu

CAP Theorem

Ziele für ein gutes System

Wir erwarten, dass ein verteiltes Datenbanksystem die folgenden drei Eigenschaften gewährleistet:

- *Konsistenz*: Die Daten sind in einem konsistenten Zustand nach Abschluss einer Transaktion
  - mit Replikation haben alle Kopien den gleichen Wert
- *Verfügbarkeit*: Rechner, die nicht ausgefallen sind, antworten in vernünftiger Zeit
- *Partitionstoleranz*: Das System toleriert den Verlust von Nachrichten
  - Dies schließt die Partitionierung des Netzes als Spezialfall ein.

Theorem 5.1 (CAP-Theorem)

Es ist unmöglich, in einem verteilten System gleichzeitig Konsistenz, Verfügbarkeit, und Partitionstoleranz zu gewährleisten [Brewer 2000, Gilbert & Lynch 2002]

CAP und die Replikationsmethoden

Wir können die Replikationsmethoden, die wir bisher betrachtet haben, in den Kontext des CAP-Theorems setzen:

- **ROWA, Primary Copy**: Fokus auf C; muss Änderungen verzögern oder abbrechen (also nicht A), wenn Netzpartitionierung auftritt
- **Voting**: Fokus auf C und A, toleriert Partitionierung nur für Partitionen, in denen das Quorum noch erreicht werden kann
- **Progressive**: Fokus auf A und P; Konsistenz kann zeitweilig verletzt sein

PACELC: Weiterentwicklung von CAP [Abadi, 2012]

CAP ist missverständlich, weil es eine Auswahlmöglichkeit zwischen A und P suggeriert, die nicht exisiert – CA und CP sind im wesentlichen gleich

Erweiterung zu PACELC:

- Wenn das System partitioniert ist (P), soll es den Fokus auf Availability (A) oder Konsistenz (C) legen?
- andernfalls (E für else), soll es den Schwerpunkt auf geringer Latenz (L) oder Konsistenz (C) legen?

In diesem Framework ist die Primary-Copy-Methode PC/EC, während die progressive Methode PA/EL ist

Interessante Blogposts über CAP

- http://pl.atyp.us/wordpress/index.php/2009/11/availability-and-partition-tolerance/
- http://mysqlha.blogspot.com/2010/04/cap-theorem.html
- http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
- Eric Brewer's keynote slides: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Zusammenfassung

Summary

- Conservative strategies
  - ROWA, primary copy
  - Voting-based strategies
- Progressive strategies
  - Data patches
- Snapshots to relax isolation
- Proactive strategies
  - Epidemic replication
- CAP Theorem

[Abadi, 2012] D. Abadi
Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story.
IEEE Computer 45(2), 2012
http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf

[Dadam, 1996] P. Dadam.
Verteilte Datenbanken und Client/Server-Systeme.
Springer-Verlag, Berlin, Heidelberg 1996.

[Rahm, 1994] Erhard Rahm
Mehrrechner-Datenbanksysteme Addison-Wesley, Bonn, 1994

[Thomas, 1979] R. H. Thomas
A Majority Consensus Approach to Concurrency Control for Multiple Copies Data Bases
ACM Transactions on Database Systems 4(2), 180–209, 1979

[Agrawal et al., 1992] D. Agrawal, A. El Abbadi
A Generealized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data
ACM Transactions on Database Systems 17(4), 689–717, 1992

[Agrawal et al., 1992a] D. Agrawal, A. El Abbadi

Resilient Logical Structures for Efficient Management of Replicated Data
VLDB, 151–162, 1992

[Garcia-Molina, 1983] H. Garcia-Molina
Data-patch: Integrating Inconsistent Copies of a Database after a Partition
IEEE Symposium on Reliable Distributed Systems, 38–48, 1983

[Abida et al., 1980] M. Abida, B. Lindsay
Database Snapshots
VLDB, 86–91, 1980

[Rabinovich et al., 1996] M. Rabinovich, N. H. Gehani, A. Kononov
Scalable Update Propagation in Epidemic Replicated Databases
EDBT, 207–222, 1996

[Vogels, 2008] W. Vogels
Eventually Consistent
ACM Queue, (6) 6, 14–19, 2008

[Özsu Valduriez, 2011] M. Tamer Özsu, P. Valduriez.
Principles of Distributed Database Systems.
Third Edition, Springer, 2011.