Replikation

Zur Erinnerung:

- Replikation sehr gut bei lesenden Anfragen: Man kann sich die billigste Kopie aussuchen. Außerdem oft gut zur Erhöhung der Ausfallsicherheit.
- Replikation schlecht bei Updates: Man muß alle Kopien konsistent halten. Außerdem kann es im Falle von Updates auch die Verfügbarkeit des Systems verschlechtern.

Roadmap:

1. Was repliziert man?
   → Allokation – bereits vor einigen Wochen behandelt
2. Wie hält man Kopien konsistent
   → 1-Copy Serialisierbarkeit
   → Andere Formen
3. Sonstiges zum Thema Replikation
   → Caching
   → Replikation von Indexen

Literatur:

Dadam Kapitel 9

Was heißt Konsistenz?

Forscher haben schon viele Formen von Konsistenz erfunden und Anwendungen gefunden, für die sie jeweils sinnvoll sind. Beispiele sind:

Weak Consistency: Die Kopie einer Parition ist ein *Snapshot* einer zu einem (vergangenem Zeitpunkt) gültigen Version der Partition.

Convergence: Wenn eine neue Version einer Partition angelegt wird, dann wird diese Version auch irgendwann einmal an alle Kopien der Partitionen propagiert.

Strong Consistency: Weak Consistency + Convergence.

t-bound: Updates werden innerhalb von 10 Minuten an alle Kopien propagiert.

t-vintage: Alle Updates vor dem 1.1.1997 sind an alle Kopien propagiert. (Evtl. sogar auch schon ein paar spätere Updates.)

Heute wollen wir allerdings nur einen Konsistenzbegriff zulassen:

> *1-Copy-Serializability:* Transaktionen werden so auf einer replizierten Datenbasis ausgeführt, daß es eine serielle Ausführung der Transaktionen auf einer Datenbasis ohne Replikate gibt.

Das klassischste Verfahren: ROWA

Read One Write All, d.h.:

1. Eine Lesetransaktion kann sich **eine beliebige** Kopie aussuchen.
2. Eine Schreibetransaktion muß **alle** Kopien schreiben. (N.B.: Das bedeutet auch Schreibsperren auf alle Kopien anzufordern.)

ROWA ist der Extremfall aller möglichen Strategien:

1. extrem gut für Lesetransaktionen.
2. extrem schlecht für Schreibtransaktionen.

Alle anderen Verfahren versuchen einen Kompromiss zu finden – d.h. etwas teuereres Lesen, dafür nur eine Teilmenge (synchron) schreiben.

Grundlagen

Die anderen Verfahren kann man (nach Dadam) entlang folgender Dimensionen klassifizieren:

1. Kopien-Update Strategien: Welche Kopien werden durch eine Updatetransaktion beschrieben.
   Alternativen: Primary Copy, Abstimmungsverfahren
2. Strategien für den Fehlerfall.
   Alternativen: konservativ, progressiv
3. Synchronisation.
   Alternativen: optimistisch, pessimistisch, ...

Kopien-Update Strategien

Welche Kopien werden geupdated?

Primary Copy: Eine Updatetransaktion führt einen Update nur auf eine ausgezeichnete Copy durch. Nach dem Commit wird der (neue) Zustand der Primary Copy an alle anderen Kopien asynchron propagiert.

Abstimmungsverfahren: Lese-Schreibkonflikte werden behoben, indem jede Lese- und jede Schreibtransaktion, die Zustimmung von mehreren "Kopien" (i.e., Sites der Kopien) einholen.
Beispiel: Es gibt 10 Kopien einer Partition A. Zum Lesen von A benötigt man 3 Stimmen und zum Schreiben benötigt man 8 Stimmen. $(10 - 3 + 1 = 8)$

Bemerkungen:

1. ROWA ist ein Spezialfall von einem Abstimmungsverfahren, bei dem ein Stimme zum Lesen und n Stimmen zum Schreiben notwendig sind.
2. ROWA und Primary Copy sind sehr verschieden!

Verschiedene Abstimmungsverfahren

Statisch vs. dynamisch:

statisch: Man legt ein für allemal fest, daß z.B. 3 Stimmen zum Lesen und 8 Stimmen zum Schreiben erforderlich sind.

dynamisch: Man legt fest, daß z.B. $> \frac{1}{3}$ der Stimmen von verfügbaren Kopien zum Lesen und $> \frac{2}{3}$ der Stimmen von verfügbaren Kopien zum Schreiben erforderlich sind.

- Vorteil. Höhere Verfügbarkeit des Gesamtsystems, wenn z.B. 5 Sites abgestürzt sind, kann bei einem statischen Verfahren niemand mehr schreiben.
- Nachteil. Höhere Kosten zum Beobachten von Sites und zur Bestimmung des aktuellen Quorums für ein Objekt.

Ungewichtete vs. gewichtete Stimmen:

Bei 10 Kopien könnte man z.B. einer *Hauptkopie* 8-faches Stimmrecht und jeder der 9 *Nebenkopien* 1-faches Stimmrecht geben. Dann kann man (bei 50%-50% Lese-Schreib Quorum), ein Objekt Lesen oder Schreiben, wenn man die Stimme der Hauptkopie und einer Nebenkopie oder die Stimmen aller Nebenkopien auf sich vereinigt hat.

Frage: Wann hat eine Gewichtung von Stimmen Vorteile?

Strategien für den Fehlerfall

Es können zwei Arten von Fehlern auftreten: (1) Ausfall einzelner Knoten; (2) Netzwerkpartitionierung.

1. Ausfall einzelner Knoten:

Primary Copy: Wenn ein Knoten ($\neq$ Primary Copy Knoten) ausfällt können seine Kopien von Objekten beim Wiederanlauf des Knotens anhand der Primary Copys wieder up-to-date gebracht werden.

Falls der Knoten mit der Primary Copy ausfällt, ist das Objekt nicht mehr verfügbar oder es wird eine andere Kopie zur Primary Copy gekürt.
(Vorsicht: Beim Küren einer neuen Primary Copy ist es wirklich wichtig, daß die Site mit der ursprünglichen Primary Copy ausfällt und nicht z.B. durch Netzwerkpartitionierung nicht mehr erreichbar ist. Außerdem muß sichergestellt sein, daß alle (committeten) Updates bereits zur neuen Primary Copy propagiert wurden.)

Abstimmungsverfahren: Null problemo; ohnehin asynchrone Propagierung von Updates auf die Sites, die sich nicht gemeldet haben oder dagegen waren. Die Sites, die dafür waren und bei denen ein Update durchgeführt werden muß sind ja lebendig.

2. Netzwerkpartitionierung:

konservative Verfahren:

- Man kürt ein "Hauptnetz" für jedes Objekt. Alle Knoten innerhalb dieses Hauptnetzes können mit dem Objekt weiterarbeiten. Knoten außerhalb des Hauptnetzes (i.e., abgeschnittene Knoten) können nicht mit dem Objekt arbeiten, solange sie vom Hauptnetz abgeschnitten sind.
- Bei Primary Copy Verfahren ist das Hauptnetz eines Objektes bestimmt durch den Knoten, der die Primary Copy hält. (Logisch!)
- Bei Abstimmungsverfahren wird das Subnetz zum Hauptnetz gekürt, das die meisten Kopien (oder Stimmen) des Objektes vereinigt.

  Bei statischem Quorum könnte aber selbst das Hauptnetz blockiert sein. Das passiert bei dynamischem Quorum nicht.

progressive Verfahren:

- Es darf jeder Knoten auch bei Netzwerkpartitionierungen uneingeschränkt weitergearbeitet werden. Updates versucht man nachher zu Mergen. (Verfahren: Data Patches.)
- Problem: temporäre Inkonsistenzen.
  Übung: Ist das noch alles serialisierbar?

Synchronisation

Grundsätzlich kann man jede Art von Synchronisation betreiben. Hier beschränken wir uns auf klassisches Sperren mit 2PL zum Erreichen von 1-Copy Serialisierbarkeit:

ROWA: Lesesperre auf beliebige Kopie; Schreibsperren auf alle Kopien.

Primary Copy: Lese- und Schreibsperren auf Primary Copy.

Abstimmungsverfahren: Lese- und Schreibsperren auf die Kopien, die ihre Zustimmung gegeben haben.

Zusätzlich muß man bei Abstimmungsverfahren noch folgendes beachten, da Updates asynchron auf andere Kopien propagiert werden:

1. Updates dürfen nur auf aktuelle Kopien durchgeführt werden.
2. "Lost Updates" beim Propagieren durch Race Conditions und Reihenfolgevertauschungen müssen vermieden werden.

(Arbeite mit Zeitstempeln oder Versionsnummern.)

Übung: Sind bestimmte Techniken zur Synchronisation für bestimmte Replikationsverfahren besonders geeignet?

Vorstellung konkreter Verfahren

Wir haben jetzt jeden Baum angepinkelt. Jetzt schlagen wir richtig zu. Es gibt folgende Verfahren:

1. ROWA. (Haben wir bereits versenkt.)
2. Primary Copy. (Da müssen wir noch einige Stückchen zusammenbatzen.)
3. Majority Consensus
4. Dynamic Voting
5. Tree Quorum
6. Reconfigurable Tree Quorum
7. Data Patches
8. Semantikbasierte Verfahren

Primary Copy

(Vorsicht das folgende ist leicht abgewandelt zum Original und der Beschreibung in Dadam, um 1-Copy Serialisierbarkeit zu erreichen!)

Das Verfahren:

1. Lesen eines Objektes (lokale Kopie verfügbar): Fordere Lesesperre von der Site der Primary Copy. Dabei melde auch aktuelle die Versionsnummer.
   - Falls die Sperre gewährt wird und die lokale Version up-to-date ist, lese lokale Kopie und sei glücklich.
   - Falls die Sperre gewährt wird aber die Versionsnummer veraltet ist, propagiere aktuelle Version und sei glücklich.
   - Falls die Sperre nicht gewährt werden kann, warte auf die Sperre und warte ggf. auch auf die Propagierung des neuen Zustandes des Objektes.
2. Schreiben eines Objektes (lokale Kopie verfügbar): Fordere Schreibsperre von der Site der Primary Copy.
   - Nach dem Gewähren, schreibe die lokale Kopie (achte auf Aktualität wie oben).
   - Am Ende, führe 2 Phasen Commit mit Primary Copies aller Objekte, die modifiziert wurden durch.
3. Propagierung des Updates zu anderen Kopien erfolgt asynchron durch die Site der Primary Copy.

Primary Copy (ctd.)

Noch einmal kurz die Vor- und Nachteile zusammengefaßt

- Nachteil. In dieser *strengen* Variante können Lokale Kopien nicht ohne Interaktion mit der Site der Primary Copy bearbeitet werden.
  → miese Verfügbarkeit
  → hohe Kosten, wenn die Site der Primary Copy hohe Last hat oder in Timbuktu liegt. → viele Vorteile von Replikation gehen verloren.
- Abhilfe. Küre andere Primary Copy. Fraglich, ob das möglich ist.
- Vorteil. Sehr einfach, asynchrone Propagierung von Updates.

Beobachtung Primary Copy Verfahren und ihre Varianten werden fast überall eingesetzt – allerdings nicht immer in unserer *strengen* Variante. Beispiel: Mirroring Caching im WWW.

Majority Consensus
(Dadam, Kapitel 9.3.2)

Vorbemerkungen

- Die Knoten sind untereinander durch einen logischen Ring verbunden, entlang dessen die Entscheidungen vorangetrieben werden. Jeder Knoten ist über die Entscheidung seiner Vorgängerknoten informiert.
- Es wird davon ausgegangen, daß es auch möglich ist, Broadcast Nachrichten zu verschicken (z.B. für Ablehnungen).
- Jede Transaktion wird mit einem eindeutigem Zeitstempel versehen.
- Jedes Objekte wird mit einem Zeitstempel versehen, welcher den Zeitpunkt der letzten erfolgreichen Änderung (Commit) angibt.
- In der ursprünglichen Definition des Verfahrens wird von einer voll replizierte Datenbasis. Das tun wir hier auch, man kann das Verfahren allerdings verallgemeinern.
- Das Verfahren verfährt *optimistisch*.

Das Verfahren (1. Teil)

- Jede (Update-) Transaktion verhält sich wie folgt:
  1. sie liest und führt alle Änderungen zunächst rein lokal durch; sie macht Änderungen zunächst aber nicht für andere Transaktionen sichtbar.
  2. erstellt eine Liste aller gelesenen und geschriebenen Objekte mit den jeweiligen Zeitstempeln
  3. schickt diese Liste zusammen mit ihrem Transaktionszeitstempel entlang des Rings an alle anderen Knoten
  4. darf die Änderungen permanent machen, wenn die Mehrheit der Knoten dem Update zustimmt.

Das Verfahren (2. Teil)

- Jeder Knoten stimmt über eingehende Änderungsaufträge wie folgt ab und reicht sein Votum zusammen mit den anderen Voten an den nächsten Knoten weiter:

  1. Er stimmt mit `ABGELEHNT`, wenn einer der übermittelten Objektzeitstempel veraltet ist.
     (D.h. Eine andere, committete Transaktion hat das Objekt inzwischen modifiziert.)
  2. Er stimmt mit `OK` und markiert den Auftrag als schwebend, wenn alle übermittelten Objektzeitstempel aktuell sind und der Auftrag nicht in Konflikt mit einem anderen, schwebenden Auftrag steht.
  3. Er stimmt mit `PASSIERE`, wenn alle Objektzeitstempel zwar aktuell sind, der Antrag aber mit einem anderen schwebenden Antrag mit höherem Zeitstempel in Konflikt steht. Falls durch **PASSIERE** keine Mehrheit mehr zustandekommen kann, so stimmt er mit `ABGELEHNT`.
  4. Er *verzögert* seine Abstimmung über den Antrag, wenn der Antrag in Konflikt mit einem Antrag mit niedrigerem Zeitstempel steht.

Das Verfahren (3. Teil)

- Annahme und Ablehnung:
  1. Der Knoten, dessen `OK` dem Antrag die *Mehrheit* verschafft hat, erzeugt eine *globale Commit Meldung* für die gesamte Transaktion.
     (Per Broadcast, aber natürlich sind Verzögerungen möglich!)
  2. Jeder Knoten, der mit `ABGELEHNT` gestimmt hat, löst ein *globales Abort* (wiederum per Broadcast aus).
  3. Wird eine Transaktion abgelehnt, so werden die *verzögerten Entscheidungen* an jedem Knoten erneut überprüft.
  4. Bei `ABGELEHNT` wird die Transaktion komplett wiederholt.

Majority Consensus – Klappe, die Letzte

Nachbermerkungen

- Der Majority Consensus Algorithmus basiert also auf *optimistischer* Synchronisation.
- Das Verfahren ist robust gegenüber Überholvorgängen.
- Ein Knoten darf ein einmal getroffenes Votum nicht mehr ändern.
- Die `PASSIERE`- und *Verzögerungs*-Regeln dienen dazu Deadlocks zu vermeiden.

Beobachtung

Ich kenne kein System, daß wirklich Majority Consensus einsetzt. Dieses Verfahren hat allerdings viele Forscher zur Entwicklung von neuen, weiteren Verfahren angeregt, die auch nicht eingesetzt werden.

Aber: Richtet Euch nicht danach was andere machen. Trefft Eure eigenen Entscheidungen, je nach Situation.

Dynamic Voting

Zur Erinnerung:

Es geht darum, Quorum an die Anzahl der *verfügbaren* Knoten anzupassen, damit eine Mehrheit erreicht werden kann, auch wenn einige Knoten ausfallen oder durch Netzwerkpartitionierung einige Knoten nicht mehr erreichbar sind.

Grundidee: Majority Consensus mit den folgender Besonderheit:

> Es haben nur die Knoten Stimmrecht auf ein Datum, die bei der letzten Abstimmung beteiligt waren.

Realisierung:

- Halte *Versionsnummer (VN)* für jedes Objekt an jedem Knoten
- Halte *Anzahl der beteiligten Knoten vom letzten Update (SK)* für jedes Objekt an jedem Knoten
- Um am Knoten i auf das Objekt zuzugreifen benötigt man SK/2+1 Stimmen.
- Versionsnummern sind nötig, um die "Gültigkeit" einer Stimme festzustellen.

Nachtrag zu Dynamic Voting

Wie bringt eine Site ihre Kopie up-to-date, nachdem eine Netzwerkpartitionierung aufgehoben wurde?

Ausgangssituation – Verbindung zwischen Sites 2 und 3 und zwischen Sites 5 und 6 wiederhergestellt. Bagger steht immer noch auf der Leitung zwischen 4 und 5:

Site	1	2	3	4	5	6	7
VN:	8	8	11	11	10	9	9
SK:	7	7	2	2	3	5	5

Kopie auf 1 und 2 sind up-to-date, Zähler sind angepaßt. Kopien 5, 6 und 7 können und dürfen nicht angepaßt werden.

Site	1	2	3	4	5	6	7
VN:	11	11	11	11	10	9	9
SK:	4	4	4	4	3	5	5

Allgemein Wenn eine Site merkt, daß ihre Kopie veraltet ist, dann führt sie folgende Aktionen aus:

1. Sie bestimmt die maximale Versionsnummer (aus Ihrer Sicht) und bestimmt wieviele Sites diese maximale Versionsnummer haben.
2. Sie schaut, ob sie sich in einer mehrheitsfähigen Partition befindet. (Das ist wichtig. Sonst würden 5-6-7 sich entschließen, selber weiterzumachen!!!)
3. Falls ja, refreshed sie ihre Kopie und paßt *SK* dieser Version an (i.e., erhöht überall den Wert um 1). Falls nicht, passiert gar nichts.

Tree Quorum

Ziel:

Anzahl der Nachrichten zur Quorumbildung reduzieren.

Grundidee:

- Alle Knoten sind in einem Baum angeordnet.
  Sei h die Höhe des Baumes und v der Verzweigungsgrad.
- Erziele die Mehrheit der *Ebenen*.
- Die Zustimmung einer Ebene wird erreicht, wenn die Mehrheit der Knoten auf der Ebene zustimmt.
- Wenn ein Knoten nicht antwortet, gilt auch die Zustimmung der Mehrheit seiner Kinder als Zustimmung des Knotens.

Bewertung:

- Man braucht tatsächlich weniger Nachrichten.
  Im Extremfall müssen nur 7 statt 61 Knoten bei einem Baum mit 121 Knoten zustimmen.
- Last auf Wurzel und Knoten "oben" im Baum sehr hoch. Die werden immer zuerst gefragt und können zum Flaschenhals werden.

Data Patches – Progressive Fehlerbehandlung

Hauptziel:

Jeder Knoten kann weiterarbeiten, auch wenn Knoten ausfallen oder Netzwerkpartitionierungen vorliegen.

Problem:

Hier ist keine Magie im Spiel. Man bezahlt die totale Verfügbarkeit mit potentiellen Inkonsistenzen.

Ansatz:

- Man versucht, Änderungen zu "Mergen"
- In einigen Fällen kann das System automatisch "Mergen." Der Systemadministrator gibt hierzu Regeln an.

  Einfügeregeln: `KEEP`, `REMOVE` oder `PROGRAM`

  Integrationsregeln: `LATEST`, `PRIMARY` k, `ARITHMETIK` oder `PROGRAM`

  Frage: Was ist mit Löschen?
- In einigen Fällen muß der Systemadministrator nachhelfen. ($\rightarrow$ `NOTIFY` Regeln).
- Schönes Analogon: CVS

Frage: Kriegt man serialisierbare Schedules?

Caching

Caching = Dynamic Replication + Second Class Ownership

1. Was heißt Dynamic Replication?
2. Was heißt Second Class Ownership?

Literatur: Buch von Mike Franklin. Client Data Caching. Kluwer Verlag. (Vielleicht schwierig zu besorgen! Aber das Zeug ist auch alles in SIGMOD und VLDB Papern erschienen.)

Dynamic Replication

Zur Erinnerung:

Ziel ist weiterhin: Halte (Kopien von) Daten dort, wo sie gebraucht werden.

Bisher wurde "statisch" durch das Allokationsmodell festgelegt, welche Site Kopien von welchen Daten hält.

Caching macht es dynamisch:

Wenn eine Site Daten verarbeitet, dann behält sie eine Kopie der Daten auch nachdem sie die aktuelle Bearbeitung abgeschlossen hat. Somit ändert sich die Allokation potentiell mit jeder Ausführung einer Query.

Beispiel:

Partition A liegt in New York, und Partition B liegt in Washington. Nun wird $A \bowtie B$ in Passau ausgeführt (i.e., A und B werden nach Passau geschickt). Anschließend werden A und B in Passau gecached, so daß weitere Anfragen mit A und B in Passau ausgeführt werden können, ohne Daten von New York oder Washington anzufordern.

Hinweis:

Dieses Prinzip kennt man von Betriebssystemen, WWW Browsern (e.g., Netscape), ...

Was wird gecached?

- Prinzipiell: alles, was dem Cache durch die Finger kommt.
- Was ist wenn der Cache überläuft: Ersetzungsstrategie (z.B. LRU oder so).
  $\rightarrow$ Ersetzungsstrategie entscheidet was gecached wird.
- Man sieht "Caching" ist hier im Wesentlichen nichts anderes als bei Betriebssystemen oder anderen Systemen mit Caching.

Allerdings zusätzlicher Freiheitsgrad:

- Cache Ergebnisse von Anfragen (i.e., Views)
- Cache Rohdaten.

(Weiterer Freiheitsgrad – nicht DBMS spezifisch:)

- Caching nur im Hauptspeicher eines Rechners.
- Caching im Hauptspeicher und auf Festplatte.

Caching von Anfrageergebnissen

Beispiel I:

Query 1: Gib mir alle roten Sportautos.

Query 2: Gib mir alle roten Sportautos aus Italien.

Query 2 kann unter unter der Verwendung des Ergebnisses von Query 1 vollständig ausgewertet werden.

→ Caching des Ergebnisses von Query 1 bringt vermutlich Riesengewinn.

Beispiel II:

Query 1: Gib mir alle roten Sportautos.

Query 2: Gib mir alle roten und schwarzen Sportautos aus Italien.

Query 2 kann teilweise unter der Verwendung des Ergebnisses von Query 1 ausgewertet werden. (Es ist noch eine *Remainder* Query notwendig, um alle schwarzen Sportautos aus Italien zu finden.)

→ Immernoch Gewinn durch Caching des Ergebnisses von Query 1 vorhanden; man spart sich das wiederholte Schippern der "roten" Ferraris.

Caching von Rohdaten

Prinzip:

Cache eine 1:1 Abbild der Daten; d.h. nur ganze oder Teile von Basispartitionen.

Granularität:

Partition (File): Immer eine ganze Partition cachen.

Seite: Es wird immer eine Seite gecached; d.h., ein Block von Tupeln derselben Partition.

Tupel: Es werden einzelne Tupel gecached.

Tradeoffs:

- Je feiner die Granularität desto größer der Aufwand, den Cache zu verwalten. (Statistiken der Ersetzungsstrategie, Deskriptoren).
- Je feiner die Granularität desto größer ist der Aufwand des Initieren des Cachings. (Meistens Granularität des Cachings = Einheit des Transports im verteilten System.)
- Je gröber die Granularität desto mehr unnützes Zeug wird gecached. Clustering ist sehr wichtig bei Caching von Seiten und ganzen Partitionen.
- **Es hat sich herausgestellt, daß für die meisten Anwendungen eine Seite (ca. 8KB) die beste Einheit ist.**

Views oder Rohdaten?

1. Viewcaching schwieriger zu implementieren.
   z.B. schwierige Ersetzungsstrategie, schwierige Konsistenzhaltung der gecacheten Views, schwierige Entscheidung was und wie man gecachete Views einsetzen kann.
   $\rightarrow$ Perfektes Viewcaching ist nicht möglich.
2. Allerdings Viewcaching universeller einsetzbar und Gewinne sind in der Regel und im Idealfall beim Viewcaching größer.

Viewcaching und seine Vor- und Nachteile sind noch Forschungsthema.

Wie hält man gecachete Kopien konsistent?

Hierzu gibt es die folgenden vier Ansätze:

1. Propagierung (genau wie statische Replikation)
2. Invalidierung (second-class ownership)
3. Optimistisch (ziemlich so wie gehabt)
4. TTL – Protokoll (second-class ownership, weak consistency)

Im folgenden behandeln wir die vier Ansätze nur für das Caching von Seiten (i.e., Rohdaten).

Übung: Kann man die vier Ansätze auch zum Caching von Anfrageergebnissen verwenden? Wenn ja, wie?

Propagierung von Updates

Prinzip:

Wie beim Konsistenthalten von (statischen) Replikaten propagiert man alle Updates zu den gecacheten Kopien.

Mehrere Spielarten möglich:

Physical: Man schickt neue Version einer Seite.

Logical: Man schickt Update Log (i.e., Delta wie "addiere 3 zum 5. Byte von Seite 17 in Deinem Cache")

Direct: Man propagiert direkt bei einem Update.

Deferred: Man propagiert auf Anfragen; d.h. sobald der Cache benutzt wird werden alle relevanten Updates propagiert.

Fazit:

Selbe Vorgehensweise wie bei statischen Replikaten. Unterschiede ergeben sich höchstens durch Nichtteilnahme bei Abstimmungen etc. (Das könnte Second-class Ownership in diesem Zusammenhang heißen. Aber wie erreicht man dann strong consistency und Serialisierbarkeit?)

Invalidierung (Callback Locking)

Motivation für etwas anderes als Propagierung:

In Passau wird $A \bowtie B$ ausgeführt. Danach passiert in Passau nichts mehr. A und B sind aber gecached und New York und Washington müssen ständig Updates propagieren.

Prinzip – Invalidierung:

Bei einem Update wird einer Site ihre gecachte Kopie entzogen. (Das ist nun wirklich Second-class Ownership.) Neben Caching von Daten findet hier auch ein Caching von (Lese-) Sperren statt.

Protokoll – Callback Locking (Primary Copy):

- Wenn Passau eine Kopie einer Seite von A hat, dann darf Passau diese Kopie lesen und benutzen. (Caching von Lesesperren!)
- Wenn München oder irgendwer ein Update auf diese Seite durchführen möchte, dann muß München eine Schreibsperre auf die Seite von dem Eigentümer der Primary Copy (i.e., New York) anfordern.
- New York schickt "callback message" an alle Sites (z.B. Passau), die eine gecachete Kopie der Seite haben.

Callback Locking (ctd.)

- Passau reagiert auf die "callback message" wie folgt. Falls eine laufende Transaktion die Seite gelesen hat, "wait" bis diese Transaktion fertig ist und dann "okay." Falls keine laufende Transaktion die Seite gelesen hat, schmeißt sie die Seite aus dem Cache heraus und signalisiert "okay."
- New York gewährt die Schreibsperre sobald alle betroffenen Site (wie Passau) mit "okay" auf ihre "callback message" reagiert haben.

Übung: Wie sähe Callback Locking aus, wenn man auch Schreibsperren cachen würde?

Fine-Grained (Callback) Locking

Problem:

Client I möchte Tupel 5 auf Seite 3 lesen. Client II möchte gleichzeitig Tupel 6 auf Seite 3 schreiben. Bei seitembasiertem Caching liegt hier ein Konflikt vor.

Idee:

Fine-grained locking aber nicht auf das seitenweise Verschicken von Daten verzichten.

Prinzip: Eskalieren und Deeskalieren von Sperren.

- grundsätzlich hält (cached) Passau eine Lesesperre auf die ganze Seite 3.
- Passau erhält eine Callback Message auf Seite 3/Tupel 6.
  1. **Fall:** Keine laufende Transaktion benutzt irgendein Tupel auf Seite 3. Passau gibt komplette Seite 3 frei und "okay" (wie oben).
  2. **Fall:** Eine laufende Transaktion benutzt Tupel 6 von Seite 3. Konflikt und "wait" (wie oben).
  3. **Fall:** Eine laufende Transaktion benutzt Tupel 5 aber keine laufende Transaktion benutzt Tupel 6. Deeskalation des Locks von Seite auf Tupelebene. D.h. Passau merkt sich, daß Tupel 6 tabu ist, und gewährt "okay" auch nur für Tupel 6.

Fine-grained Callback Locking (ctd.)

- Abhängig von der Art der "okays" (Seite oder Tupel) gewährt New York München eine Schreibsperre auf die ganze Seite oder nur auf Tupel 6.

Literatur: Carey, Franklin, Zaharioudakis SIGMOD 1994

Optimistisch

Prinzip:

Das funktioniert eigentlich wie ganz gewöhnliche optimistische Synchronisation unter der Annahme das New York die Primary Copy der gesamten Datenbasis hält:

- Passau schickt die Read und Write Sets seiner Transaktionen nach New York.
- Genauso machen es München und alle anderen Sites.
- New York schaut in Validierungsphase, ob alles okay ist, und führt auch die atomare Schreibphase durch.
- Bei Abbruch (Lese-Schreibkonflikt) kommt ein refresh von New York, damit der Restart der Transaktion eine Chance auf Gelingen bekommt.

Hinweise:

- Fine-grained Synchronisation kommt bei optimistischen Protokollen kostenlos. (Großer Vorteil!!!)
- Das Verfahren funktioniert auch bei Transaktionen die Daten verwenden, deren Primary Copy auf verschiedenen Sites gehalten wird.

Literatur: Adya et al. SIGMOD 1995

Time-to-live (TTL) Protokoll

Vorgehensweise:

- Wenn New York, die Seite 3 nach Passau schickt, teilt New York Passau gleichzeitig mit, daß sich diese Seite innerhalb der nächsten Woche (oder so) nicht ändern wird.
- Passau benutzt die gecachte Kopie von Seite 3 zum Lesen völlig frei innerhalb der nächsten Woche (oder so).
- Nach Ablauf der Woche fragt Passau vor jeder Benutzung New York, ob die gecachte Kopie noch aktuell ist. Falls ja, weitere Benutzung, ansonsten erhält Passau eine neue Kopie mit neuer time-to-live.

TTL (ctd.)

Hinweise:

- TTL garantiert keine Serialisierbarkeit.
  (Wenn sich Seite innerhalb der Woche ändert, hat Passau Pech gehabt.)
- TTL evtl. sogar teurer als z.B. Callback Locking.
  (Betrachte eine Welt ohne Updates.)
- TTL wird im WWW z.B. von Netscape unterstützt. Im WWW können TTLs auch vom Administrator vergeben werden. (Z.B. wird die TTL von Werbeobjekten üblicherweise auf 0 gesetzt, damit bei jedem neuen Laden einer Seite eine neue Werbebotschaft kommt.)

Abschließende Bemerkungen

1. Caching findet zusätzlich zur statischen Allokation/Replikation statt.
   D.h. man muß natürlich nachwievor eine Grundallokation finden und diese ist auch sehr wichtig.

2. Caching findet automatisch statt. Statische Allokation und Replikation findet auf Initiative eines Systemadministrators statt.

3. Caching ist "schwächer" als Replikation; z.B. gibt Caching keine Garantien für das Vorhandensein (und häufig auch für die Gültigkeit) von Kopien.

4. Replikation ist etwas für Servermachinen (administrierte Rechner). Caching ist etwas für Clientmachinen (nicht administrierte Rechner).

5. Hier haben wir Caching immer in Zusammenhang mit *Primary Copies* betrachtet. Theoretisch könnte man aber wohl auch Caching mit Abstimmungsverfahren kombinieren, wenn statische Allokation mehrere Kopien für ein Objekt vorsieht. Das hat nur noch niemand gemacht. (Wieso?)

6. Anfrageoptimierung sollte Caching beachten. ($\rightarrow$ Übung!)