Update Propagation Protokolle In Replizierten Datenbanken

Michael Kropfberger
michael.kropfberger@gmx.net

31. Januar 2000

Inhaltsverzeichnis

Kurzbeschreibung Dieser Artikel behandelt die verschieden Ansätze von Update Propagation Protokollen, welche zu möglichst effizienten und skalierbaren replizierten Datenbanken führen sollen, aber trotzdem die Serialisierbarkeit erhalten müssen. Ausgehend von der einfachsten, aber problematischsten Form, des Eager Updates, werden einige Lazy Replikationsmodelle vorgestellt. Beginnend mit dem Lazy-Master Protokoll werden Replikationsgraphenbasierte und Copygraph-basierte Protokolle erklärt und deren Vor- und Nachteile erläutert.

Schlüsselwörter Update Propagation Protokolle - Replizierte Datenbanken - Eager - Lazy - Replikationsgraph - Copygraph - Lazy-Master - DAG(WT) - DAG(T) - BackEdge

1 Einführung

Um der immer wichtigeren Anforderung nach höherer Verfügbarkeit und höherer Performanz nachkommen zu können, müssen Daten auf mehreren Rechnern gleichzeitig zur Verfügung gestellt werden. Für statische Daten (bzw. semistatische wie zB. Telefonbücher, etc.) bedeutet dies nur ein einmaliges Duplizieren mit darauffolgend garantiert konsistentem Zugriff. Bei dynamischen Datenbeständen wie Data Warehouses, Kundendateien, Bankkonten usw. müssen Transaktionen auf allen Replikationen konsistent und serialisierbar ausgeführt werden.

1.1 Ziele

Ideale Ziele eines Replikationsschemas wären: [3]

Verfügbarkeit und Skalierbarkeit durch Replikation bei Erhaltung von Stabilität

Mobilität erlaubt Mobilgeräten (Handys, Laptops) lesenden und schreibenden Zugriff auf den Datenbestand, während sie offline sind

Serialisierbarkeit bei beliebiger Ausführungsreihenfolge von globalen Transaktionen

Konvergenz an einen überall gültigen Datenbankzustand ohne kurzzeitige Diskrepanzen

Wir werden die später vorgestellten Replikationsmethoden auf Kompatibilität mit diesen Punkten noch eingehend prüfen.

1.2 "Eager" Replikationsprotokoll

Der naive Ansatz, um das Problem in den Griff zu bekommen, wäre die "Eager" Replikation. Hier wird jede Operation (read und write) einer Transakion noch während der Ausführung an die replizierenden Knoten gesandt und auf allen Knoten alle notwendigen Locks angefordert. Am Schluss wird mittels eines Zwei-Phasen Commits auf allen Knoten gleichzeitig bestätigt. Logischerweise führt dies bei steigender Knotenanzahl zu immer größer werdenden Transaktionen. Durch die ebenfalls erhöhte Anzahl von Locks steigt die Deadlockgefahr mit der Anzahl der Knoten kubisch an[3].

1.3 “Lazy” Replikationsprotokolle

Um diese großen Transaktionen aufzubrechen, wird bei “Lazy” Replikationsprotokollen zuerst auf dem Knoten, auf welchem die Transaktion gestartet worden ist, die gesamte Transaktion ausgeführt und bestätigt (commit). Erst dann wird diese Transaktion an alle Knoten weiter geschickt, die Replikationen von Daten besitzen. Zwar verringert sich so die Wahrscheinlichkeit eines Deadlocks, aber es kann nur mit ausgeklügelten Algorithmen die Serialisierbarkeit erhalten bleiben.

In Abb. 1 werden die verschiedenen Replikationsmethoden graphisch veranschaulicht.

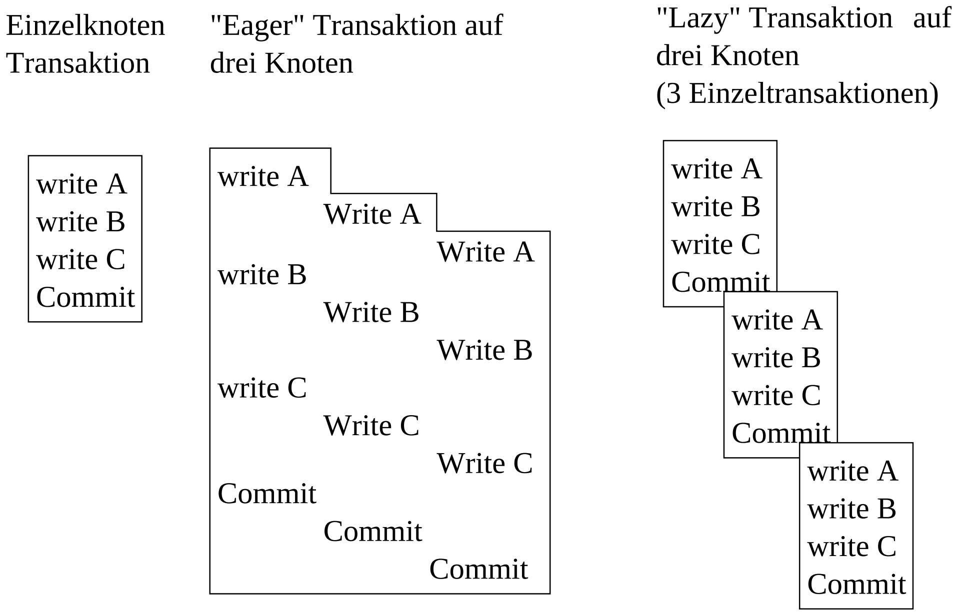

Abbildung 1: Arten von Replikation

Abb. 2 zeigt die beiden Möglichkeiten, um Updates an die Replikationsknoten weiterzuleiten. Entweder wird für jedes Datenobjekt ein Masterknoten auserwählt und dieser leitet Updates an alle weiter, oder jeder Knoten darf seine Kopie lokal ändern und diese Änderung an alle anderen weiterleiten.

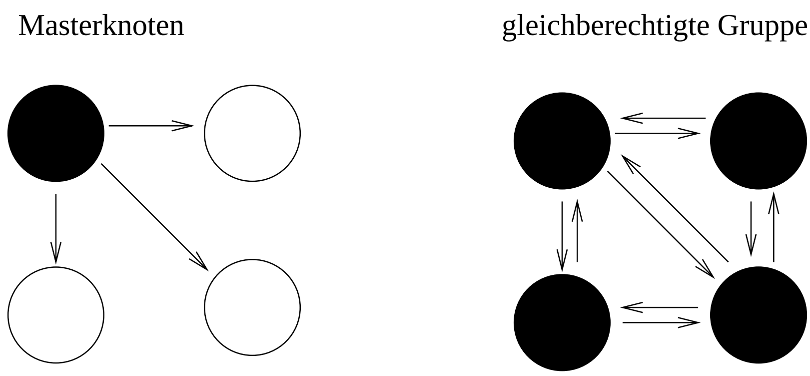

Abbildung 2: Methoden für die Weiterleitung von Updates

Bei der Aktivierung von Transaktionen sprechen wir immer vom lokalen *Hauptknoten*, wo diese Transaktion gestartet wurde, welcher aber nicht zwingend der Masterknoten eines Datenobjektes sein muß. Weiters bezeichnen wir diese erste Transaktion als *Haupttransaktion*, wobei alle folgenden global abgesetzen als *Subtransaktionen* in die Terminologie aufgenommen wird.

In den folgend vorgestellten Replikationsschematas kann sich jede Transaktion T_i in einem der folgenden Zustände befinden [5] (siehe Abb. 3):

Aktiv , wenn T_i auf dem Hauptknoten läuft, aber weder bestätigt noch abgebrochen ist. Von hier kann T_i in **bestätigt** oder **abgebrochen** übergehen.

Bestätigt , wenn T_i auf dem Hauptknoten bestätigt (Commit) ist, kann T_i nur noch in **beendet** übergehen. Nun werden alle Updates "lazy" auf die Replikationsknoten als Subtransaktion geschickt, wo sie ebenfalls auf ein **bestätigt** warten. Dort können sie zwar durch das lokale RDBMS beliebig oft abgebrochen werden, werden aber so oft wiederholt, bis T_i auch dort **bestätigt** ist.

Abgebrochen , wenn T_i auf dem Hauptknoten abgebrochen worden ist.

Beendet , wenn T_i auf allen Replikationsknoten bestätigt ist, und es dort keine lokalen aktiven Transaktionen gibt, die vor T_i in der Serialisierungsreihenfolge stehen.

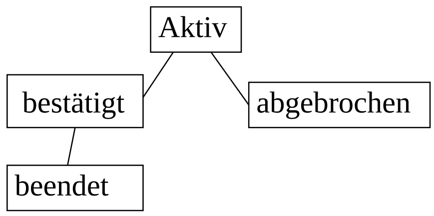

Abbildung 3: Zustandsübergangsdiagramm

1.4 Aktuelle Kommerzielle Produke

Aufgrund der enormen Performanz bieten die meisten kommerziellen Datenbanksysteme wie Sybase, Oracle, CA-OpenIngres Unterstützung für "lazy" Replikation an. Doch leider wurde bei der Realisierung nicht an Serialisierbarkeit gedacht: Die einzelnen Transaktionen werden lokal abgearbeitet und bestätigt, um dann asynchron ihre Updates an die replizierenden Knoten zu senden. Durch diese Entkopplung der Haupttransaktion[1] von den Subtransaktionen[2] wird die Gefahr von replikationsbedingten Deadlocks gebannt, doch können zwei Knoten gleichzeitig und unabhängig voneinander Daten ändern und dies führt zu einem undefinierten Zustand. Diesem Problem wird durch vordefinierte Abgleichungsregeln entgegengewirkt (zB wird immer das Update mit dem jüngsten Zeitstempel eingespielt). Dies setzt aber die Kommutativität der Updates voraus.

[1] die "eigentliche" Transaktion am ausführenden Knoten

[2] welche nur noch die Updates auf den anderen Knoten einspielen

2 Lazy-Master Replikation (globales Locking)

Um trotz Lazy-Replikation die Serialisierbarkeit zu erreichen, kann man jedem Datenobjekt einen eindeutigen Masterknoten zuteilen. Falls nun eine Transaktion lokal nur ein Replikat dieses Datenobjekts zur Verfügung hat, muß zB per Remote Procedure Call (RPC) ein Write-Lock an den Masterknoten gestellt werden. Weiters muss bei jedem Read auch ein Read-Lock beim Masterknoten angefordert werden. Nach dem Commit der Haupttransaktion werden die neuen Updates von den Masterknoten zu den Replikationsknoten gesendet (siehe Abb. 2). Diese Lazy Updates beinhalten einen Zeitstempel, somit wird bei mehreren Transaktionen auf einen Knoten nur das neueste Update eingespielt (Thomas Write Regel bei write-write Konflikten [7]). Lazy-Master Replikation ist gegenüber der Eager-Replikation trotz der kleineren Transaktionsgröße wegen der globalen Locks nur etwas weniger deadlockgefährdet. Ausserdem eignet sich dieser Ansatz nicht für Mobilgeräte, da immer eine Verbindung zu den jeweiligen Masterknoten hergestellt werden muß.

2.1 Zwei-Schichten Replikation

Um auch mobile Knoten einbinden zu können, bietet sich eine Spaltung der Rechnergruppen in "mobile Knoten" und "Basisknoten" an. Basisknoten verhalten sich ganz nach dem Lazy-Master Motto. Mobile Knoten hingegen führen ihre Updates nur auf ihre lokale Datenbank aus und merken sich diese als "Versuch". Beim nächsten andocken werden diese "Versuchsupdates" mit dem Lazy-Master Algorithmus neu gestartet. Falls diese abgelehnt werden, wird dies dem Besitzer des Mobilgerätes angezeigt und dieser muß entsprechend reagieren. Weiters wird natürlich die lokale Datenbank des Mobilgeräts wieder an die gegebenenfalls neuen Datenbestände angeglichen.

Die Gegenüberstellung in Abb. 4 der bis jetzt vorgestellten Weiterleitungs- vs. Eigentumsstrategien verdeutlicht, daß Lazy+Gruppe Protokolle im Vergleich zu Eager+Master Protokollen viel anfälliger gegenüber Serialisierbarkeitsverletzungen sind.

<table>
<tr><th>Weiterleitung
vs. Eigentum</th><th>Lazy</th><th>Eager</th></tr>
<tr><td>Gruppe</td><td>N Transaktionen
N Objekteigentümer</td><td>eine Transaktion
N Objekteigentümer</td></tr>
<tr><td>Master</td><td>N Transaktionen
ein Objekteigentümer</td><td>eine Transaktion
ein Objekteigentümer</td></tr>
<tr><td>Zwei-Schichten</td><td colspan="2">N+1 Transaktionen, ein Objekteigentümer, lokale Versuchsupdates, Eager Updates auf die Basisknoten</td></tr>
</table>

Abbildung 4: Gegenüberstellung der Strategien bei N Knoten

3 Replikationsgraphen-basierte Protokolle mit RGTest

3.0.1 Virtuelle Knoten

Ein angepasstes RDBMS vorausgesetzt, kann man über jeden physischen Knoten k eine sich dynamisch ändernde Anzahl von virtuellen Knoten legen. Jede Transaktion i auf Knoten

k wird mit so einem virtuellen Knoten VK_i^k verbunden. Es ist wichtig, daß sich mehrere Transaktionen einen virtuellen Knoten teilen können. ($VK_i^k = VK_j^k$). Diese virtuellen Knoten werden nach folgenden Regeln gebildet:

Lokalitätsregel Jede lokale Transaktion wird immer genau auf einem virtuellen Knoten ausgeführt. Bei Updates auf andere Replikationsknoten wird dort wieder genau ein virtueller Knoten verwendet. VK_i^k beinhaltet immer genau die Datenobjekte, auf die T_i auf dem Knoten k bis dahin lesend oder schreibend zugegriffen hat.

Vereinigungsregel Wenn zwei Transaktionen T_i und T_j auf das gleiche Datenobjekt zugreifen, müssen diese beiden virtuellen Knoten VK_i^k und VK_j^k zusammengeführt werden, sodaß alle Datenobjekte in beiden Knoten enthalten sind. Bei write-write Konflikten innerhalb eines vereinigten Knotens tritt die Thomas Write Regel in Kraft (Zeitstempelbasierte Reihenfolge).

Spaltungsregel Wenn der physische Knoten bemerkt, daß eine Transaktion T_i in den Zustand **abgebrochen** oder **beendet** überging, können alle exklusiv von T_i benutzen Datenobjekte aus VK_i^k und gegebenenfalls VK_j^k gelöscht werden. Falls es kein $VK_i^k = VK_j^k$ gibt, wird VK_i^k effektiv für das Replikationsprotokoll gelöscht.

3.0.2 Replikationsgraph

Für das gesamte verteilte System wird ein globaler Replikationsgraph angelegt, welcher die ablaufenden Transaktionen abbildet. Für jede Transaktion T_i existiert eine Kante (T_i, VK_i^k) zu allen virtuellen Knoten der involvierten physischen Knoten k. Jede Operation in einer beliebigen Transaktion könnte eine der drei oben erwähnten Regeln auslösen und somit den Replikationsgraphen verändern. In [4] wird gezeigt, daß ein zu jedem Zeitpunkt azyklischer Replikationsgraph immer global serialisierbar ist. Deshalb muß jede Operation einer beliebigen Transaktion T_i versuchsweise auf den Replikationsgraphen angewandt werden. Ein Test auf Azyklität (RGTest) muß erst zeigen, ob diese Operation nun ausgeführt werden darf. Wenn nicht, muß diese Transaktion warten oder sich beenden. Die Anzahl und Ausführungszeitpunkte des RGTests führen zu einem pessimistischen und optimistischen Ansatz.

3.1 Pessimistischer Ansatz [5]

1. Teile T_i bei seiner ersten Operation einen Zeitstempel zu.

2. führe den RGTest bei jedem Read oder Write von T_i auf dem lokalen Knoten durch.

   - Bei Erfolg führe die Operation durch[3] und aktualisiere den Replikationsgraphen.
   - Bei Mißerfolg und T_i war lokal, breche die Transaktion ab.
   - Bei Mißerfolg und T_i war global, teste, ob im Replikationsgraphen bereits eine Transaktion T_j im Zustand **beendet** ist. Falls dies zutrifft, muß T_i abgebrochen werden, sonst kann noch auf den Abbruch einer anderen Transaktion gehofft werden und es wird gewartet.

3. Bei einem Write/Update, daß nicht auf dem lokalen Knoten ausgeführt wird, wende die Thomas Write Regel an.

[3] gegebenenfalls unter Anwendung der Thomas Write Regel

4. Wenn T_i in den Zustand **bestätigt** wechselt, fahre mit der Ausführung fort. Beim Zustand **beendet** kann T_i aus dem Replikationsgraphen gelöscht werden und gegebenenfalls die Spaltungsregel aufgerufen werden. Teste, ob irgendwelche anderen wartenden Transaktionen aktiviert werden können.

5. Wenn T_i in den Zustand **abgebrochen** auf seinem lokalen Knoten wechselt, muß T_i aus dem Replikationsgraphen gelöscht werden und gegebenenfalls die Spaltungsregel aufgerufen werden. Es müssen gegebenenfalls auch alle Subtransaktionen aus diversen Warteschlangen gelöscht werden. Teste, ob irgendwelche anderen wartenden Transaktionen aktiviert werden können.

Man beachte, daß, obwohl lokale Transaktionen keine Knoten im Replikationsgraphen besitzen, diese trotzdem auf virtuellen Knoten arbeiten. Somit können auch lokale Transaktionen aufgrund der Vereinigungsregel die Entfernung von globalen Transaktionen aus dem Replikationsgraphen verzögern.

3.2 Optimistischer Ansatz [5]

Auch der optimistische Ansatz garantiert globale Serialisierbarkeit. Jedoch wird der RGTest nur ein mal durchgeführt, und zwar wenn eine Transaktion in den Zustand **bestätigt** übergeht. Es entstehen dadurch wesentlich weniger Aufrufe des RGTests und da es keine Wartezustände gibt, kommt es auch zu keinen globalen Deadlocks. Lokal sind natürlich weiterhin Deadlocks möglich, diese werden aber sowieso gesondert vom lokalen RDBMS behandelt.

1. Teile T_i bei seiner ersten Operation einen Zeitstempel zu.

2. Führe jedes Read oder Write von T_i auf dem lokalen Knoten. aus (gegebenenfalls unter Einsatz der Thomas Write Regel). Alle Operationen werden mitprotokolliert, da dann in Punkt 4 daraus die entsprechenden virtuellen Knoten gebildet werden.

3. Bei einem Write/Update, daß nicht auf dem lokalen Knoten ausgeführt wird, wende die Thomas Write Regel an.

4. Wenn T_i lokal vom Zustand **aktiv** in **bestätigt** übergehen will, muß der RGTest auf alle Operationen von T_i ausgeführt werden. Bei Erfolg werden diese "Versuchsoperationen" übernommen. Ansonsten müssen alle "Versuchsoperationen" rückgängig gemacht werden und T_i geht in den Zustand **abgebrochen** über.

5. Falls T_i sich lokal schon im Zustand **bestätigt** befindet, und auf einem Replikationsknoten ebenfalls in **bestätigt** übergehen will, dann fahre fort. Falls dies zum Gesamtzustand **beendet** führt, kann T_i aus dem Replikationsgraphen gelöscht werden und gegebenenfalls die Spaltungsregel aufgerufen werden.

6. Bei einem lokalen **abgebrochen** von T_i, fahre fort und rufe gegebenenfalls die Spaltungsregel auf.

Der optimistische Ansatz erlaubt jeder Transaktion, auf ihrem lokalen Knoten komplett unabhängig zu laufen. Erst bei einem lokalen Commit (Wechsel in Zustand **bestätigt**) wird Koordination mit anderen Knoten und Transaktionen nötig.

4 Copygraph-basierte Protokolle

4.0.1 Copy Graph

Wie schon in der Lazy-Master Replikation besitzt jedes Datenobjekt einen eindeutigen Masterknoten und mehrere Replikationsknoten. Somit kann man einen Copygraph erstellen, welcher genau diese Abhängigkeiten darstellt. Falls ein Datenobjekt r auf dem Masterknoten K_k^r und eine Replikation auf dem Knoten l liegt, so bildet man eine gerichtete Kante von K_k^r nach K_l^r. Eine Transaktion schreiben wir als $T_t^k[w(r_x), r(s_y)]$, wenn Transaktion t auf dem Knoten k startet und auf Datenobjekt r in der version x schreibt und Datenobjekt s in der Version y liest.

Als BackEdges bezeichnet man genau jene gerichteten Kanten, welche durch ihre Entfernung dem Copygraphen zur Azyklität verhelfen. Nach der Entfernung der BackEdges entspricht der Copygraph einem "Directed Acyclic Graph (DAG)".

In [2] wird gezeigt, daß Lazy Replikationsprotokolle bei *ungerichteten* azyklischen Graphen serialisierbar sind. Bei gerichteten azyklischen Graphen (DAG) kann dies aber nicht garantiert werden, wie folgendes Beispiel zeigt:

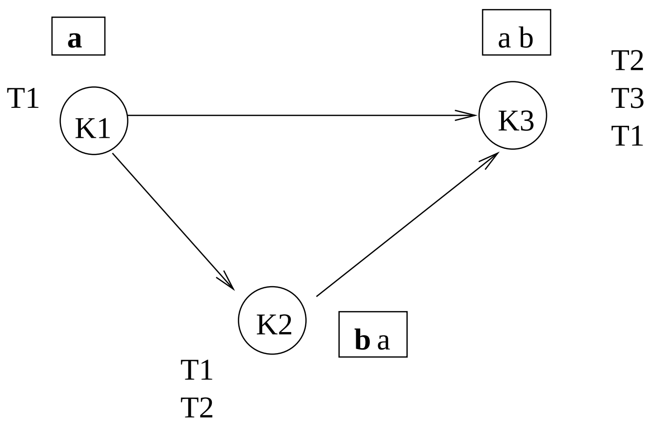

Abbildung 5: Copygraph-Beispiel einer nichtserialisierbaren Ausführung

Beispiel: Wie in Abb. 5 gezeigt wird, besteht unser verteiltes System aus drei Knoten, wobei das Datenobjekt a den Masterknoten K_1 und die Replikationsknoten K_2 und K_3 besitzt, b den Masterknoten K_2 und den Replikationsknoten K_2 besitzt. Transaktionen sind

$$T_1^1[w(a)],\ T_2^2[r(a), w(b)],\ T_3^3[r(a), r(b)]$$

. Also startet Transaktion T_1 auf K_1 und schreibt auf das Datenobjekt a und so weiter.

Durch Lazy-Propagation Strategie könnte es passieren, daß T_1 Objekt a vor dem Start von T_2 auf K_2 updatet, aber K_3 erst nach dem Update von b von T_2 und der Beendigung von T_3 erreicht. Diese Reihenfolge würde formal aufgeschrieben so lauten:

$$T_1^1[w(a_2)],\ T_1^2[w(a_2)],\ T_2^2[r(a_2), w(b_2)],\ T_2^3[w(b_2)],\ T_3^3[r(a_1), r(b_2)],\ T_1^3[w(a_2)]$$

Zum Vergleich wäre hier eine korrekte Variante, die sich nur dadurch unterscheidet, daß das Update von T_1 auf K_3 noch vor T_3 stattfindet:

$$T_1^1[w(a_2)],\ T_1^2[w(a_2)],\ T_2^2[r(a_2), w(b_2)],\ T_2^3[w(b_2)],\ T_1^3[w(a_2)],\ T_3^3[r(a_2), r(b_2)]$$

Da die Ausführungsreihenfolge aber vom Zufall[4] bestimmt ist, führt dieser Ansatz zu nicht serialisierbaren Ergebnissen.

4.1 DAG(WT) Protokoll (DAG ohne Zeitstempel) [1]

Um das Problem der zufälligen Update-Reihenfolge zu lösen, wird im DAG(WT) Protokoll ein Baum eingeführt. Dieser zeigt jeden Knoten K_l als Unterknoten von K_k, wenn K_k eine Masterkopie eines Datenobjekts besitzt und K_l nur eine Replikation (vgl. Abb. 6).

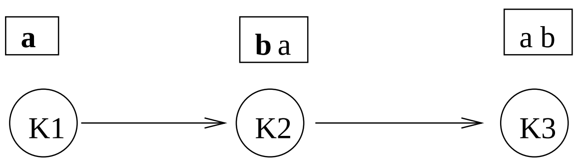

Abbildung 6: DAG(WT) Baum aus dem Copygraphen von Abb. 5

Wenn nun eine Transaktion ein Update startet, werden diese Subtransaktionen entlang des Baums an alle relevanten Kinder weitergegeben. Diese führen - bei Besitz eines Replikates - das Update aus und geben es an deren Kinder weiter. Auf jedem Knoten werden diese Subtransaktionen in einer First-Come-First-Serve (FCFS) ausgeführt, mit Commit bestätigt und weitergeleitet.

Somit würde die Ausführung unseres Beispiels wie folgt aussehen:

$$T_1^1[w(a_2)],\ T_1^2[w(a_2)],\ T_1^3[w(a_2)],\ T_2^2[r(a_2), w(b_2)],\ T_2^3[w(b_2)],\ T_3^3[r(a_2), r(b_2)]$$

Wichtig ist hier, daß das Update von $T_1[w(a_2)]$ noch vor T_2 zum Knoten K_3 weitergeleitet wird und somit serialisierbar bleibt.

Größter Nachteil von DAG(WT) ist, daß alle Subtransaktionen alle Kindesknoten des Baumes durchlaufen müssen, obwohl möglicherweise dort gar keine Replikationen liegen. Dies führt zu einem starken Nachrichtenoverhead und zu hohen Propagation delays.

4.2 DAG(T) Protokoll (DAG mit Zeitstempel) [1]

Deshalb versucht man mit dem DAG(T) Protokoll, die Updates nur entlang des Copygraphen und daher nur zu relevanten Knoten zu propagieren. Um trotzdem eine eindeutige Reihenfolge für die Ausführung der Subtransaktionen zu gewährleisten, werden systemweite Zeitstempel eingeführt.

4.2.1 Zeitstempel

Durch die Azyklität des Copygraphen ist eine totale Ordnung über die Knoten gegeben ($K_1 < K_2 < K_3$). Somit besteht ein Zeitstempelvektor eines Knotens aus Tupeln von Zeitstempeln der vorhergehenden Knoten des Copygraphen und seines eigenen. Um das "Verhungern" von Transaktionen zu verhindern, wird zusätzlich noch ein Epochenzähler eingeführt. Ein Beispiel für so einen Zeitstempelvektor der Epoche 1 wäre $E_1(K_1, 2)(K_2, 2)(K_3, 4)$ mit den einzelnen Tupeln für jeden vorhergehenden Knoten von K_3.

Wir definieren $ZS_i < ZS_j$, wenn gilt:

[4] und von anderen Faktoren, auf die wir hier nicht näher eingehen wollen :)

- Epoche von ZS_i war vor der Epoche von ZS_j, oder
- ZS_i ist ein Präfix von ZS_j, oder
- Zwei Zeitstempel $ZS_i = X(K_k, lokalerZS_k)Y_i$ und $ZS_j = X(K_l, lokalerZS_l)Y_j$ teilen sich das gleiche Präfix X und unterscheiden sich erst im Tupel $(K_k, lokalerZS_k)$ und $(K_l, lokalerZS_l)$, dann gilt $ZS_i < ZS_j$, wenn
  - $K_i > K_j$, oder
  - $K_i = K_j$ und $lokalerZS_k < lokalerZS_l$

Somit gilt

- $E_1(K_1, 1) < E_1(K_1, 1)(K_2, 1)$
- $E_1(K_1, 1)(K_3, 1) < E_1(K_1, 1)(K_2, 1)$
- aber auch $E_1(K_1, 1)(K_2, 1) < E_2(K_1, 1)(K_3, 1)$
- $E_1(K_1, 1)(K_2, 1) < E_1(K_1, 1)(K_2, 2)$

4.2.2 Details des DAG(T) Protokolls

Auf jedem Knoten wird ein Zeitstempelvektor gespeichert. Zu Beginn werden alle Tupel auf $(K_i, 0)$ gesetzt. Nach einem Commit einer lokalen Transaktion wird der lokale Zeitstempel erhöht und alle Updates inklusive des Zeitstempelvektors an die Replikationsknoten geschickt. Somit weiß jeder Knoten bei Erhalt einer Transaktion inklusive Zeitstempel, welche Transaktionen vorher noch zu erledigen sind, um die Serialisierbarkeit zu erhalten.

In unserem Beispiel würde also T_1 den Zeitstempel $E_1(K_1, 1)$ bekommen, und T_2 hätte $E_1(K_1, 1)(K_2, 1)$. Somit würde Knoten K_3 auf jeden Fall zuerst T_1 ausführen, da T_2 ja den Zeitstempel von T_1 als Präfix führt.

5 Hybride Replikationsprotokolle

5.1 BackEdge Protokoll [1]

Bisher hatten wir die große Einschränkung, keine zyklischen Copygraphen gestatten zu können. In Abb. 7 sehen wir eine Situation, die mit den bisher bekannten Lazy Protokollen (bis auf das Lazy-Master Protokoll) nicht serialisierbar ist. Betrachten wir dazu folgende Transaktionen $T_1^1[w(a), r(b)]$ und $T_2^2[w(b), r(a)]$, die gleichzeitig auf K_1 und K_2 gestartet werden und dort auch mit Commit bestätigen.

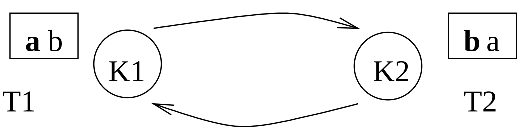

Abbildung 7: Zyklischer Copygraph

Wie bereits weiter oben besprochen, beschreiben BackEdges genau jene Kanten, welche einen Copygraphen zyklisch machen. Das BackEdge Protokoll verwendet nun ein Eager Propagation Protokoll entlang der BackEdges und für den übrigbleibenden azyklischen Graphen ein Lazy Protokoll wie zum Beispiel das DAG(WT) Protokoll.

Falls vom Knoten K_i eine BackEdge zum Knoten K_j besteht, muß logischerweise im Copygraphen ohne BackEdges eine Kante von K_j zu K_i existieren. Dadurch kann es passieren, daß Updates nicht nur an Kindesknoten propagiert werden müssen, sondern auch nach oben zu den Vorfahren. Wir nennen nun die notwendigen Startknoten für die Subtransaktionen $S_1 \dots S_f$ in den Vorgängerknoten $K_{i_1}, \dots, K_{i_f}$, wobei K_{i_1} der am weitesten entfernteste zu K_i ist.

Die notwendigen Erweiterungen des DAG(WT) für das BackEdge Protokoll sind:

1. Nach der Abarbeitung von T_i wird die erste Subtransaktion S_1 an S_{i_1} geschickt, wobei T_i weiterhin alle Locks hält und noch nicht mit Commit bestätigt. Die weiteren Subtransaktionen $S_2 \dots S_f$ werden auch gestartet, aber behalten ebenfalls ihre Locks und bestätigen noch nicht mit Commit.

2. Nach der Abarbeitung von S_1 (S_1 hält weiterhin alle Locks und hat noch nicht mit Commit bestätigt) werden alle Updates an alle relevanten Kinder zwischen K_{i_1} und K_i gesendet.

3. Nachdem dieses Update entlang der ersten BackEdge wieder bei K_i angelangt ist, werden T_i und alle seine Subtransaktionen $S_1 \dots S_f$ atomar mit Commit bestätigt (mit einem verteilten Commit Protokoll wie zum Beispiel dem Zwei-Phasen Commit) und die Locks freigegeben.

4. Nun können alle weiteren Updates zu Kindern von K_i "lazy" (ohne Locks) nach dem DAG(WT) Protokoll propagiert werden.

Diese Schritte bergen natürlich eine erhöhte Deadlockgefahr, sind aber aus Gründen der Serialisierbarkeit und Atomarität notwendig. Man beachte aber, falls der Copygraph ein normaler gerichteter azyklischer Graph (DAG) ist[5], degeneriert somit das gesamte BackEdge Protokoll zu dem weitaus effizienteren DAG(WT), da keine BackEdge Transaktionen mehr notwendig sind.

6 Zusammenfassung

Wir haben in dieser Arbeit die grundsätzliche Problematik der Updates in replizierten Datenbanken kennengelernt. Wir kennen nun zwei große Gruppen von Replikationsprotokollen, nämlich die "Eager" und "Lazy" Protokolle.

Das "Eager" Protokoll eignet sich wegen der vielen globalen Locks nicht besonders für große replizierte Knotenanzahlen[6] und führt zu schlechter Skalierbarkeit und hoher Deadlockgefahr. Die "Lazy" Protokolle hingegen bieten je nach Komplexität der verwendeten Algorithmen und Datenstrukturen weitaus höhere Skalierbarkeit und schalten die Möglichkeit von globalen Locks weitestgehend aus.

[5]und daher keine BackEdges hat

[6]und gar nicht für Mobilgeräte, da diese ja sonst immer angedockt sein müßten

- Es existiert der Lazy-Master Ansatz, welcher jedem Datenobjekt einen Masterknoten zuteilt. Updates werden nur dort durchgeführt und dann "lazy" an alle Replikationsknoten weitergeleitet.

- Durch Replikationsgraphen und virtuelle Knoten wird diese Abhängigkeit an einen Hauptknoten eliminiert, es muß aber jede Transaktion einen RGTest bestehen, um propagiert zu werden. Auch der Aufwand für die Verwaltung von virtuellen Knoten ist nicht zu unterschätzen.

- Bei Copygraph-basierten Protokollen wird wieder ein Masterknoten pro Datenobjekt verwendet, jedoch bieten erweiterte Datenstrukturen und bessere Algorithmen messbar bessere Performance [1]. Im Speziellen wurde hier das DAG(WT) und DAG(T) Protokoll vorgestellt.

- Durch hybride Protokolle wie dem BackEdge Protokoll können auch zyklische Copygraphen durch teilweise "Eager" Propagation gehandhabt werden.

Festzustellen bleibt noch, daß dieses Forschungsgebiet durch Mobilgeräte und deren Applikationen mit replizierten Datenbeständen immer wichtiger wird. Auch Teleworking fordert den unabhängigen Wechsel des Arbeitsplatzes und verlangt immer mehr nach Replikation. Nicht zu vergessen ist hier auch die notwendige Replikation (Mirrors) zur Performancesteigerung für die Bereithaltung von Daten über das Internet. Somit wird noch einiges an Arbeit notwendig sein, um das neu gewonnene Wissen von den Testlabors auch in die heutigen kommerziellen Produkte einfliessen zu lassen.

Literatur

[1] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, Avi Silberschatz: "Update Propagation Protocols For Replicated Databases", SIGMOD Philadelphia, 99: 97-108

[2] Parvathi Chundi, Daniel J. Rosenkrantz, S. S. Ravi: "Deferred Updates and Data Placement in Distributed Databases", Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, 1996: 469-476

[3] Jim Gray, Pat Helland, Patrick E. O'Neil, Dennis Shasha: "The Dangers of Replication and a Solution", Proceedings of the ACM SIGMOD Conference, Montreal, Quebec, Canada 1996: 173-182

[4] Yuri Breitbart, Henry F. Korth: "Replication and Consistency: Being Lazy Helps Sometimes", Proceedings of the ACM SIGACT-SIGMOD-SIGART Symphsium on Principles of Database Systems, Tucson, Arizona, 1997: 173-184

[5] Todd A. Anderson, Yuri Breitbart, Henry F. Korth, Avishai Wool: "Replication, Consistency, and Practicality: Are These Mutually Exclusive?", Proceedings of ACM SIGMOD International Conference on Management of Data, Seattle, WA, 1998: 484-495

[6] J. Gray and A. Reuter: "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, San Mateo, CA, 1993

[7] P. A. Bernstein, V. Hadzilacos, N. Goodman: "Concurrency Control and Recovery in Database Systems", Addison-Wesley, Reading, MA, 1987