Transaktionsverwaltung und Indexe

Wieso ist das einfach?

- Man braucht keine langen (2PL) Sperren, weil lange Sperren auf den Datensätzen die Synchronisation von Operationen, die in Konflikt stehen, implizieren.
- Anders ausgedrückt: Indexe sind eh redundant.

Wieso ist das schwierig?

- Trotz Redundanz, muß man etwas tun, da man den Index nicht jedes mal neu aufbauen möchte.
- Es gibt Operationen, die nur auf Indexen ausgeführt werden. Die müssen richtig bearbeitet werden. Z.B. *Gib mir das Durchschnittsgehalt der Top 10.000.*
- Es gibt eine Reihe von unangenehmen Effekten, die auftreten können.

Ziele

- **Synchronisation:** Höherer Parallelitätsgrad und niedrigerer Aufwand durch kurze Sperren (Latches).
- **Recovery:** Alles möglichst wie gehabt.

Unangenehme Effekte – Ziele

- Roll-Back einer Transaktion, die einen Split verursacht hat
  - REDO/UNDO einer anderen Transaktion findet plötzlich auf einer anderen Seite statt.
  - Split darf nicht UNDONE werden, weil evtl. gar nicht mehr möglich.
- Systemabsturz kann mitten während eines Splits erfolgen.
- Phantomproblem wollen wir auf dem Index lösen.
- Während eines Splits sollen soviel wie möglich andere Operationen auf dem Index parallel ausgeführt werden.
- Insgesamt sollen nur kurze Sperren gehalten werden und viel höhere Parallelität als auf den Daten erzielt werden.
- Es sollen keine Deadlocks durch Sperren im Index auftreten.
  (Siehe Übung.)

Begriff Latch

- Das ist die Art Sperre, die wir auf Indexseiten halten.
- (Kurze) Sperre auf eine Seite, die explizit angefordert und freigegeben werden kann.
- Shared (R) und exclusive (X) Latch mit üblicher Semantik.
- Latches werden zusammen mit der Seite vom Puffermanager und nicht vom Lockmanager verwaltet. (Geringerer Overhead.)

Synchronisation im B-Baum

1. Verfahren

- Leser und Schreiber hangeln sich durch das Halten von zwei R Latches zum richtigen Blatt.
- Leser halten R Latch auf Blatt, solange sie es brauchen. Sie müssen sich evtl. zu einem Nachbarblatt hangeln. (Begründung: Inkonsistenz durch Splits.)
- Schreiber halten X Latch auf Blatt, solange sie die Änderung noch nicht vorgenommen haben. Bei einem Split wird der X Latch nach Durchführung des Splits auf Blattebene freigegeben.
  (Durch Split kann es sein, daß ein Leser das falsche Blatt besucht und zum Nachbarn gehen muß.)
- Nach dem Split traversiert der Schreiber erneut den Baum und fordert auf dem Weg zum Blatt X Latches wie folgt an:
  - Auf dem aktuellen Knoten wird ein X Latch angefordert.
  - Wenn der aktuelle Knoten mehr als einen Eintrag frei hat, werden alle anderen X Latches (oberhalb) freigegeben.
- Resultat: Schreiber hält X Latches auf allen relevanten Knoten und kann Baum anpassen. Nach Abschluß werden alle X Latches wieder freigegeben.

Verfahren 1

Literatur

- Gray, Reuter: Kapitel 15.4
- Bayer, Schkolnick: Acta Informatica, 1977

Synchronisation im B-Baum

Problem: Was passiert, wenn ein anderer Schreiber einen weiteren Split in der Zwischenzeit durchgeführt hat?

- Gray, Reuter sagen zu diesem Fall leider nichts!
- Man kann diesen Fall allerdings auf zwei Arten ausschließen:
  - Der zweite Schreiber erkennt das Problem und wartet mit seinem zweiten Durchlauf, bis der erste Schreiber abgeschlossen hat.
  - Alle Schreiber, die Splits oder Merges durchführen, werden generell durch einen separaten Latch synchronisiert. Dann sollte man sich allerdings auch den zweiten Durchgang sparen und den Split traditionell von unten nach oben durchführen. Dabei muß man allerdings vor Deadlocks aufpasen. (Aries/IM, SIGMOD 1992).
- **Frage:** Was ist besser im Sinne von Aufwand und Parallelität?
- **Frage:** Kann das überhaupt passieren?

Synchronisation im B-Baum

2. Verfahren: Grundideen

- B-Link Tree: Knoten auf allen Ebenen werden verkettet. (Nicht nur Blätter.)
- Jeder Knoten enthält zusätzlichen *HIGH* Eintrag.
- Leser können auf allen Ebenen halbdurchgeführten Split tolerieren, so ähnlich wie sie den halbdurchgeführten Split auf der Blattebene im 1. Verfahren tolerieren können.
- Leser navigieren immer nach unten oder rechts.
- Schreiber machen Änderungen von unten nach oben und von links nach rechts.

Literatur

- Lehman, Yao: TODS 1981

Algo für Fetch ([LY81])

```
Input: v
current = root;
A = get(current);
while (current is not a leaf) {
      current = scannode(v, A);
      A = get(current);
}
while ((t=scannode(v, A)) == link pointer of A) {
      current = t;
      A = get(current);
}
if (v is in A) return success
else return failure
```

- Im *scannode* versteckt sich Entscheidung, ob nach rechts oder unten navigiert wird. Abhängig vom Wert von *HIGH*.
- Der Leser fordert keine Latches an!

Algo für Insert (1. Suche)

```
initialize stack;
current = root;
A = get(current);
while (current is not a leaf) {
      t = current;
      current = scannode(v, A);
      if (current not link pointer in A) push t;
      A = get(current);
}
```

Navigation in Blättern

```
lock(current);
while (t = scannode(v, A) is a link pointer) {
      lock(t);
      unlock(current);
      current = t;
      A = get(current);
}
```

Algo für Insert (2. Do It)

```
Input: ⟨v,w⟩ (v=Schlüssel, w=Zeiger)
if (A is safe) {
   insert ⟨v,w⟩ into A
   put(A, current);
   unlock(current);
}
else {
   u = allocate new page for B;
   redistribute A over A and B, adding ⟨v,w⟩;
   y = HIGH value stored in A now;
   // HIGH value stored in B is
   // previous HIGH value of A;
   put(B, u);
   put(A, current);
   oldnode = current;
   v = y; w = u;
   current = pop(stack);
   lock(current);
   move.right(); // wie Navigation in Blättern
   unlock(oldnode);
   wiederhole diese Routine;
   biege dabei v, w und oldnode richtig hin
}
```

Synchronisation im B-Baum

Bewertung

- 2. Ansatz ist wohl der Sieger was Parallelität betrifft
  - ein Schreiber hält max. 3 Latches gleichzeitig
  - Leser werden überhaupt nie blockiert
- Performance Experimente findet man in Carey: SIGMOD 1990
- laut Gray, Reuter ist das letzte Wort allerdings noch nicht gesprochen

Key Range Locking

Verwendung eines B-Baums zur Lösung des Phantom Problems

- Sperren auf Schlüssel in Blättern (= Datensätze) werden gemäß 2PL gehalten. (Keine Latches!)
- Idee: Sperre eines Schlüssels k_i in einem Blatt sperrt das Intervall:
$$(k_{i-1}; k_i]$$
- Ein Scan von k_l bis k_m sperrt gleichzeitig auch k_{m+1}. Hiermit wird implizit auch das Intervall $(k_m; k_{m+1}]$ gesperrt, um Phantome rechts von k_m zu verhindern.
- Ein Insert von k_n sperrt (exklusiv) auch k_{n+1}, den Schlüssel rechts vom neuen Schlüssel k_n. Hierdurch werden Phantomkonflikte zwischen Scans und Inserts entdeckt. Hierdurch werden allerdings auch zwei parallele und benachbarte Inserts, die sich nicht wehtun, ausgeschlossen.
- Beim Delete von k_d wird ebenfalls k_{d+1}, der Eintrag rechts neben k_d, mit gesperrt. Dies ermöglicht *Repeatable Read*, auch wenn die Löschtransaktion zurückgesetzt wird. (Das ist eine ganz besondere Form eines Phantoms.)

Bemerkungen

- Die nächsten Einträge können sich auf einer anderen Seite befinden.
- Man muß einen besonderen Dummy Eintrag für ∞ vorsehen. (Wird z.B. bei einem vollständigen Scan benötigt.)

Literatur:

- ARIES/KVL, VLDB 1990
- ARIES/IM, SIGMOD 1992

R-Baum und GIST

Wesentliche Beobachtungen

- Verfahren 1 funktioniert nicht mehr, weil man im zweiten Durchgang den Weg zum richtigen Blatt evtl. nicht mehr findet.
- Verfahren 2 funktioniert, muß aber verfeinert werden, da es das Konzept eines HIGH Eintrages auf einer Seite nicht mehr geben kann. Aus diesem Grund kann "scannode" nicht mehr wissen, ob es nach rechts oder nach unten navigieren muß.
- Key-range Locking funktioniert nicht mehr, da es keinen logisch gesehen nächsten Eintrag mehr gibt.

R-Baum und GIST

Lösung

- Erweitere Verfahren 2.
  - Jede Seite bekommt eine Node Sequence Number (Zeitstempel).
  - Bei einem Split bekommt, rechter Zwilling alte NSN, während linker Zwilling eine neue NSN bekommt.
  - Nach dem Split erhält Vaterknoten beim Update die NSN des linken Zwillings.
  - Man erkennt veraltete Wegweiserinfo durch Vergleich der NSN von Vater und Sohnknoten.
  - Das Navigieren nach rechts kann man abbrechen, nachdem man auf die Seite mit der Vater NSN (oder $<$ Vater NSN) getroffen ist.

- Verwende anstelle von Key-range Locking Prädikatsperren. Trick ist, daß die Prädikatsperren an die einzelnen Wegweiser geheftet werden. Details sind ein wenig kompliziert.

Literatur: Kornacker et al.: VLDB 1995, SIGMOD 1997.

Recovery von Indexen

Zur Erinnerung

- Hauptproblem liegt darin, daß eine Transaktion, die einen Split (oder Merge) gemacht hat, zurückgesetzt werden kann.

Grundprinzipien

- Trennung von SMOs (Structure Modifying Operations) und Inserts und Deletes von Schlüsseln in den Blättern.
- Inserts und Deletes verändern die Semantik des Baumes, SMOs nicht.
- Beim UNDO einer Transaktion werden nur die Inserts und Deletes der Transaktion zurückgesetzt, die SMOs allerdings nicht.
- D.h. SMOs sind transaktionsneutral, obwohl sie natürlich von Operationen von Transaktionen verursacht wurden.
- Protokollierung:
  - SMOs: physisch
  - Inserts, Deletes: logisch

Literatur: Gray, Reuter: Kapitel 15.4.5

Erweiterte Transaktionskonzepte

Beobachtungen

- ACID sehr gut für kurze Transaktionen
  Banken (Überweisungen, Geldabheben)
- Atomicity und Isolaton zu restriktiv bei langen Transaktionen
  - Durch A geht zuviel Arbeit beim Zurücksetzen verloren.
  - Durch I entstehen zu viele Konflikte. Blockaden bei Sperrprotokollen und inakzeptable Rücksetzungen bei optimistischen Verfahren.

Anwendugnsbeispiele

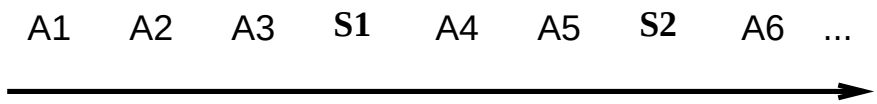
- Gebe allen Mitarbeitern eine 10% Gehaltserhöhung.
- Workflowmanagement; z.B. Bearbeitung eines Auftrages von der Annahme, Produktion, Lieferung und Rechnung bis zur Mahnung.
- CAD oder Software Entwicklungsprojekte.

Grundprobleme

- ACID Transaktionen sind flach.
- ACID Transaktionen haben keine Binnenstruktur.

Literatur: Kapitel 16: Härder, Rahm Buch

Transaktionen mit Rücksetzpunkten



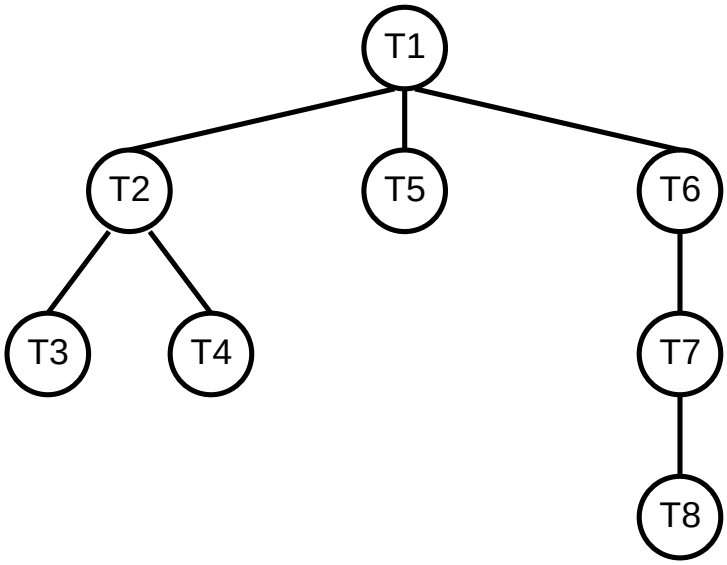
Prinzip

- Anwendung kann bei einer Transaktion Sicherungspunkte setzen.
- *SAVE* und *RESTORE* Befehle.
- Partielles UNDO möglich. (Auch beim Systemcrash.)
- Standardisiert in SQL 3.
- N.B.: Sicherungspunkte sichert auch Sperren. (Oder Read Set oder Write Set bei optimistischen Verfahren.)

Bewertung

- Löst *A* Problem aber nicht das *I* Problem.
- Sichern auf DB Ebene reicht in der Regel nicht aus. Nach einem Systemcrash ist auch das Sichern des Anwendungskontextes (lokale Programmvariablen, Ausführungsstack, etc.) notwendig.

Geschlossen geschachtelte Transaktionen



Eigenschaften

- Top-level Transaktion ist ACID.
- Sub-Transaktionen sind ACI (D hängt vom Vater ab).
- Befehl *INVOKE* zum Aufruf einer Subtransaktion.

Grundregeln

Es gelten folgende Regeln:

- **Rücksetzregel:** Wenn eine Transaktion zurückgesetzt wird werden alle Subtranskation (auch bereits abgeschlossene) rekursiv zurückgesetzt.
- **Sichtbarkeitsregel:** Commit einer Transaktion macht die Ergebnisse und Änderungen der Subtransaktion für die Vatertransaktion sichtbar. Eine Transaktion kann alle Objekte, die sie hält, ihren Subtransaktionen sichtbar machen.
- **Commitregel:** Commit einer Transaktion macht die Ergebnisse und Änderungen (zunächst) nur für die Vatertransaktion sichtbar. Für andere Transaktionen (außerhalb des Baumes) werden Änderungen erst mit dem Commit der Wurzeltransaktion sichtbar.

Varianten

Synchrone Aufrufe, serielle Ausführung:

Synchrone Aufrufe, parallele Ausführung:

Asynchrone Aufrufe, parallele Ausführung:

Vater-Sohn Kooperation

Single Call Schnittstelle

Konversationsschnittelle

- Abbruch der Subtransaktion erfordert (partielles) Rücksetzen und Wiederholen der Vatertransaktion.
- Das betrachten wir hier nicht.

Synchronisation von Geschachtelten Transaktionen

Annahmen

- Asynchrone Aufrufe, parallele Ausführung möglich.
- Single-Call Schnittstelle, keine Vater-Sohn Konversation.
- Wir betrachten nur eine Erweiterung des einfachen R-X Sperrverfahrens.

Verfahren

- *Retained* Sperren für Vatertransaktionen; i.e., $r - R$ und $r - X$ Sperren.
- Transaktionen fordern ganz normal R und X Sperren zum Lesen und Schreiben von Objekten an. Transaktionen innerhalb desselben Baumes müssen diese R und X Sperren beachten und werden somit synchronisiert.
- Wenn eine Subtransaktion committed, erhält die Vatertransaktion alle Sperren der Subtransaktion als retained Sperren. Die Subtransaktion gibt alle ihre Sperren frei.
- Vatertransaktionen erben auch alle retained Sperren ihrer Subtransaktionen. (In diesem Fall bleiben die Sperren retained Sperren.)

- Beim Abbruch einer Transaktion werden alle Sperren der Transaktion freigegeben. Die Sperren der Vorfahren der abgebrochenen Transaktion bleiben unberührt.
- Eine Transaktion kann eine X Sperre erwerben, falls keine andere Transaktion eine X oder R Sperre hält sowie alle Transaktionen, welche eine $r-X$ oder $r-R$ Sperre halten, Vorfahren sind.
- Eine Transaktion kann eine R Sperre erwerben, falls keine andere Transaktion eine X Sperre hält sowie alle Transaktionen, welche eine $r - X$ oder $r - R$ Sperre halten, Vorfahren sind.
- Eine Transaktion kann retained Sperren in ordentliche Sperren umwandeln und umgekehrt(!!!).
  - Das Umwandeln einer retained Sperre in eine ordentliche Sperre ermöglicht einer Transaktion den Zugriff auf ein Objekt, nachdem eine Subtransaktion auf das Objekt zugegriffen hat.
  - Das Umwandeln einer ordentlichen Sperre in eine retained Sperre ermöglicht einer Subtransaktion den Zugriff auf ein Objekt, nachdem ein Vorfahr auf das Objekt zugegriffen hat.
  - **Vorsicht allerdings:** Zugriffe von Vater und Kinder und Enkel auf ein Objekt werden allerdings synchronisiert! Nach Rückwandlung in eine retained Sperre verliert eine Transaktion die direkten Zugriffsrechte.

Bewertung: Geschlossen geschachtelte Transaktionen

Vorteile

- Flexible Rücksetzpunkte.
- Parallelität innerhalb einer Transaktion.
- Besonders gut für verteilte, parallele System.
- Modularisierung von Anwendungen.
- Erste Implementierungsansätze: Encina.

Nachteile

- Bei Abbruch der Top-level Transaktion wird immer noch komplette Arbeit zerstört.
- Immernoch rigorose Isolierung zwischen Top-level Transaktionen (i.e., verschiedenen Bäumen).

Offen geschachtelte Transaktionen

Prinzip

- Eine Subtransaktion gibt alle ihre Sperren (ohne Vererbung und Retained Sperren) nach Commit frei.
- Die Änderungen einer Subtransaktion sind für alle nach dem Commit der Subtransaktion sichtbar.
- **Vorsicht: Jetzt wird es fuzzy!**

Synchronisation

- Subtransaktionen werden wie normale Transaktionen mit allen anderen Transaktionen synchronisiert.
- Serialisierbarkeit der Vatertransaktionen ist nicht mehr gewährleistet.

Recovery

- Nur kompensationsbasierte Recovery möglich. (Rücksetzen des Vaters impliziert nach wie vor Rücksetzen der Kinder.)
- Übliches Problem: kompensationsbasierte Recovery nicht immer möglich (z.B. Loch gebohrt, Bargeld ausbezahlt, etc.)

SAGAs

- Spezielle, eingeschränkte Form von offenen geschachtelten Transaktionen.
- Nur zweistufig.
- ACID Eigenschaften für Subtransaktionen.
- ACD für komplette SAGA (i.e., Wurzeltransaktion).
- Rücksetzung durch Compensation Transaktions:

  BS $T_1\ T_2\ \ldots T_j$ (Abort während T_{j+1}) $C_j\ C_{j-1}$ ...

  (T_{j+1} muß durch normale UNDO Recovery zurückgesetzt werden.)
- Unterstützung von Savepoints.

Literatur: Garcia-Molina, Salem: SIGMOD 1987

ConTracts

- Erweiterung vom SAGA Konzept.
- Unterstützung von Workflowanwendungen.
- Teiltransaktionen heißen Steps – sind aber dasselbe (i.e., ACID Transaktionen).
- Workflow ist beliebiger Graph von Steps: Verzweigung von Schleifen von Steps sind modellierbar.
- Es können Invarianten zwischen Steps definiert werden. Beispiel: Freie Parkplätze. Implementierung ähnlich wie VERIFY/MODIFY in IMS Fast Path.
- Anwendungskontext ist Teil eines Savepoints. (Behebt eines der wesentlichen Probleme von Transaktionen mit Rücksetzpunkten.)
- Prototypische Implementierung an der Uni Stuttgart.

Literatur: Wächter, Reuter: Tech Report, 1991.

Entwurfsumgebungen

- **Stellt Euch RCS vor.**
- Client/Server Umgebung.
- Check In (ci)/Check Out (co) Paradigma mit Versionsverwaltung.
- Check Out zum Sperren legt neue Version eines Objektes an. Alte Versionen des Objektes können weiter von jedem gelesen werden.
- Check In checkt neue Version eines Objektes ein und macht sie für alle anderen sichtbar.
  - Neue Version muß konsistent sein.
  - Bei Inkonsistenz allerdings kein Zurücksetzen sondern Eingriff des Benutzers.
- Kooperation zwischen Gruppen von Benutzern wird durch eine dreistufige "Client/Group-Server/Server" Architektur realisiert.
  - Striktes ci-co Protokoll auf der Group-Server / Server Ebene.
  - Email zur Synchronisation der Aktivität der Clients einer Gruppe.
- **Wichtig:** In solchen Umgebungen spielt die Musik außerhalb des DBMS (i.e., des Servers). Viel schwierigere Aufgaben an den Clients.