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

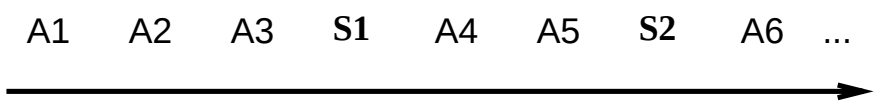
- Gebe allen Mitarbeitern eine 10% Gehaltserhöhung.
- Workflowmanagement; z.B. Bearbeitung eines Auftrages von der Annahme, Produktion, Lieferung und Rechnung bis zur Mahnung.
- CAD oder Software Entwicklungsprojekte.

Grundprobleme

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

Literatur: Kapitel 16: Härder, Rahm Buch

Transaktionen mit Rücksetzpunkten



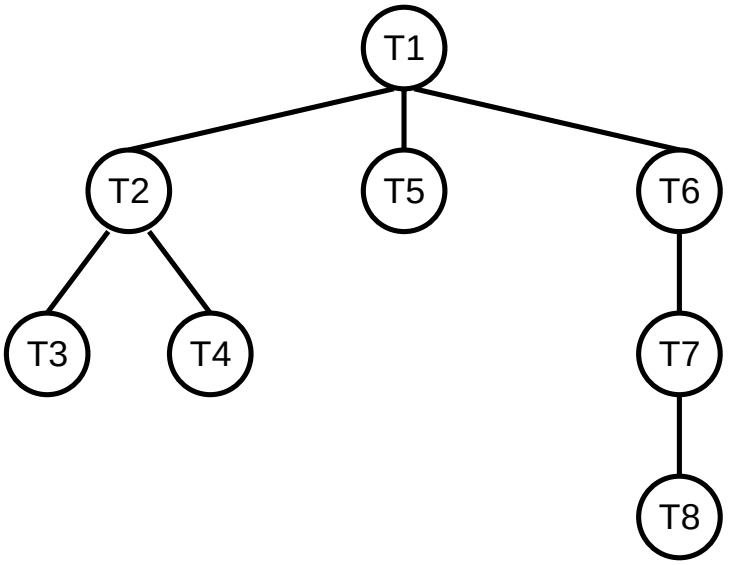
Prinzip

- Anwendung kann bei einer Transaktion Sicherungspunkte setzen.
- *SAVE* und *RESTORE* Befehle.
- Partielles UNDO möglich. (Auch beim Systemcrash.)
- Standardisiert in SQL 99.
- 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} $\ldots$

  (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

- **Stellen Sie sich 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.

Transaktionen in verteilten Systemen

- Lokale Transaktionen
  Greifen nur auf Daten eines Knotens zu.
- Verteilte (globale) Transaktionen
  Greifen auf Daten mehrerer Knoten zu.

  Werden auf einem Knoten gestartet und breiten sich, abhängig von den Daten die sie benötigen, auf weitere Knoten aus.

Verteilte Transaktionen

- Aus Sicht des Anwenders wie Transaktionen in einem zentralen DBMS. (Stichwort: Transaction Transparency)

- **Struktur**
  - Root-Agent
    Wird auf dem Startknoten erzeugt.
    Startet weitere Agenten auf anderen Knoten.
    Ist für die Koordination der verschiedenen Agenten zuständig.
  - Agent
    Führt Teiltransaktionen (Subtransaktion) auf seinem Knoten aus.

Ablauf einer verteilten Transaktion

1. Transaktion T_1 wird in NY gestartet.
2. T_1 wird an *query processor* geleitet, z.B. in D.C.
3. „Zerlegte“ T_1 wird an einen Transaktionsmanager geleitet, z.B. in PA.
4. Root-Agent $T_{1,0}$ wird in PA gestartet.
   Falls nötig, werden weitere Agenten $T_{1,1}, T_{1,2}, \ldots, T_{1,n}$ auf verschiedenen Knoten gestartet.

   Aufgaben des Root-Agenten:

   - Ablaufsteuerung
   - Abwicklung des Zwei-Phasen-Commit-Protokolls (2PC)
   - Fehlerbehandlung
   - Protokollierung
   - Zurückschicken des Ergebnisses nach NY

Aufrufstruktur

Hängt hauptsächlich vom Wissen des Query Processors über Partitionierung und Allokation ab.

- Flach
  Vollständiges Verteilungswissen

- Geschachtelt
  Begrenztes Verteilungswissen

Frage: Kann immer vorab entschieden werden, wo Teiltransaktionen gestartet werden?

Korrektheit verteilter Transaktionsausführungen

Eine Menge verteilter Transaktionen T_v wird auf den Knoten $K_1, K_2, \ldots, K_n$ ausgeführt.

Die parallele Ausführung von T_v ist **korrekt** *gdw.*

1. die lokalen Schedules auf $K_1, K_2, \ldots, K_n$ jeweils *serialisierbar* sind, und
2. sich aus der Menge der äquivalenten seriellen Schedules an $K_1, K_2, \ldots, K_n$ eine widerspruchsfreie serielle Ausführungsreihenfolge der Transaktionen in T_v ableiten läßt.

Deadlocks (Verklemmungen)

In sperrbasierten Protokollen müssen Transaktionen warten, falls eine Sperre nicht gewährt werden kann.

$\longrightarrow$ *Deadlocks* können auftreten.

Deadlock:
Permanente, zyklische Wartebeziehung.

Eine Menge von Transaktionen T_d ist verklemmt (deadlocked) *gdw.* jeder Transaktion aus T_d auf die Freigabe von Sperren wartet, die andere Transaktionen aus T_d halten.

$\longrightarrow$ Alle Transaktionen sind blockiert.

Mehrere Deadlocks können in einer Menge verklemmter Transaktionen auftreten.

Deadlockbehandlung

- Verhindern (Prevention)
  Anforderungen so einschränken, daß keine Deadlocks auftreten können. (unrealistisch)
- Vermeiden (Avoidance)
  Bei jeder Anforderung überprüfen, ob sie zu einem Deadlock führen kann. (zu teuer)
- Erkennen und Auflösen
  Deadlocks zulassen und sie dann erkennen und auflösen.
- Timeout
  Vorgabe der maximalen Zeit, die eine Transaktion auf eine Sperre warten darf. (unnötige Aborts)

Wir machen Deadlockerkennung.
(Timeout vs. Erkennen ist Religionskrieg)

Korrektheit von Deadlockerkennungsalgorithmen

Korrektheitskriterien:

- *Progress Property*
  Jeder Deadlock wird erkannt.
- *Safety Property*
  Jeder erkannte Deadlock existiert tatsächlich, d.h. es werden keine *Phantomdeadlocks* erkannt.

- Spontane Aborts können bei jedem Algorithmus zur Erkennung von Phantomdeadlocks führen.

- Überflüssige Aborts sind auch durch falsche Opferauswahl möglich.

Verschiedene Deadlockmodelle

Welches Modell zutrifft hängt vom Zugriffsmodell ab.

- Single-Resource-Modell
- AND-Modell
- OR-Modell
- $\binom{n}{m}$-Modell
- Modell ohne Restriktionen

Single-Resource-Modell

Annahme:

> *Jeder Request einer Transaktion betrifft nur ein (Sperr-) Objekt.*

Wie Läuft eine Transaktion ab?

1. Transaktion macht einen Zugriff,
2. wartet auf Antwort,
3. falls weitere Zugriffe nötig, weiter bei 1),
   sonst commit/abort.

N.B.: Trotzdem mehrere Wartebeziehungen und Deadlocks möglich.

Transaktion muß auf die Freigabe *aller* Sperren, mit denen sie in Konflikt steht, warten.

$\longrightarrow$ Ein Deadlock existiert *gdw.* ein Zyklus im WFG existiert.

AND-Modell (Resource-Model)

- Eine Transaktion kann mehrere Sperren gleichzeitig anfordern.

- Zusammen mit dem einfachen Single-Resource-Modell das klassische Modell für VDBMSe.

- Eine Transaktion wartet ebenfalls auf die Freigabe *aller* Sperren, mit denen sie in Konflikt steht.
- Wieder existiert ein Deadlock *gdw.* ein Zyklus im WFG existiert.

OR-Modell (Communication Model)

- Häufig im Betriebssystembereich:
  Prozeß wartet auf *einen beliebigen* Drucker.

- In VDBMSe:
  Leseanforderung im read-one/write-all Protokoll (ROWA);
  Hier wartet Transaktion auf eine Sperre auf *einer beliebigen* Kopie des Objektes.

- Bei exklusiven und Lese-/Schreibsperren gilt:
  Ein Deadlock existiert *gdw.* ein *Knot* im WFG existiert.

  v ist in einem Knot *gdw.*
  ($\forall w$: w ist von v aus erreichbar $\Rightarrow$ v ist von w aus erreichbar)

$\binom{n}{m}$-Modell

- Zugriffsverhalten:
  Transaktion benötigt *beliebige* m Sperren aus einer Menge von n Sperren.

- Beispiel: Majority-Consensus Synchronisationsprotokoll.

- Dieses Modell umfasst das AND- und OR-Modell
  AND-Modell entspricht $\binom{n}{n}$
  OR-Modell entspricht $\binom{n}{1}$

- Es gibt keine Graphstruktur, die einem Deadlock in diesem Modell entspricht.

  – Ein Zyklus ist eine *notwendige* aber *nicht hinreichende* Bedingung.
  – Ein Knot ist eine *hinreichende* aber *nicht notwendige* Bedingung.

- Deadlockerkennung durch Reduktion des WFG.

Algorithmus für zentrale DBMSe

- Der WFG wird aufgebaut und nach Deadlocks (Zyklen) durchsucht.
- Jeder Deadlock wird aufgelöst,
  durch den Abbruch von Transaktionen.
- Im WFG können mehrere Deadlocks gleichzeitig existieren.
  - Es wird versucht die Menge von Transaktionen abzubrechen, die am „günstigsten“ ist.
  - Aber: dies ist NP-hart.

Auflösungsstrategien

- Bei der Auflösung von Deadlocks sollten folgende Kriterien eingehalten werden:
  - Garantierter Fortschritt
    Es gibt immer eine Transaktion im System, die garantiert fertig wird.
  - Kein Verhungern
    Keine Transaktion darf unendlich oft abgebrochen werden.

- Natürlich wird versucht, möglichst billige Transaktionen abzubrechen.
- Mögliche Strategien:
  - die jüngste,
  - die mit den wenigsten Sperren,
  - die, die am wenigsten gearbeitet hat,
  - eine zufällig ausgewählte,
  - ⋮

Algorithmen für VDBMSe

- **Probleme:**
  - Alle, die in zentralen Systemen vorliegen.

    +
  - Jede Site hat nur lokale Sicht.
  - Information von anderen Sites kommt mit Verzögerung an, kann also veraltet sein.
  - (Allerdings nehmen wir an:
    Kommunikation "sicher" aufgrund von TCP.)

- Es gibt zwei Arten von Algorithmen zur Deadlockerkennung in VDBMSe:
  - zentralistische Algorithmen
  - verteilte Algorithmen

Zentraler Algorithmus

- Eine Site auswählen, die für die Deadlockerkennung zuständig ist.
- Es gibt zwei Varianten:
  1. Alle anderen Sites schicken der ausgezeichneten *periodisch* die Wartebeziehungen, die bei ihnen entstanden sind.

     Auf der ausgezeichneten Site wird der WFG zusammengebaut und nach Zyklen durchsucht.
  2. Jede Site baut WFG aus lokalen Wartebeziehungen auf und durchsucht ihn nach Zyklen.

     Den Rest des WFGs schickt sie, *periodisch*, an die ausgezeichnete Site.
- Häufig kombiniert mit zentraler Sperrverwaltung. (Wenn schon denn schon.)

Bewertung

- Nachteile:
  - Flaschenhals
  - Verfügbarkeit
  - Zu weiter Weg für Deadlocks auf Sites, die nahe beieinander liegen
  - Hohe Belastung für ausgezeichnete Site

- Vorteile?
  -

Klassifikation:

1. Path-pushing
2. Edge-chasing
3. Global state detection
   (a) Diffusing computation
   (b) Explizites Aufbauen des WFG

Path-Pushing Algorithmen

- Jede Site baut lokalen WFG auf, sucht nach Zyklen in diesem Teilgraphen und löst sie auf.
- Der Rest des lokalen WFGs wird an andere Sites geschickt.
  (Daher der Name *path-pushing*.)
- Empfängt eine Site Teile des WFGs von einer anderen Site, nimmt sie diese zu ihrem lokalen WFG hinzu, sucht nach Zyklen und löst sie auf.

  Der resultierende Graph wird wieder weiter geschickt.
- Dies wird so lange wiederholt, bis eine Site genügend Informationen hat, um festzustellen, daß es keine Deadlocks mehr gibt.

Algorithmus von Obermarck

Literatur: R. Obermarck, ACM Transactions on Database Systems, 1982
(Auch bei Knapp beschrieben)

- Path-pushing Algorithmus für stark eingeschränktes AND-Modell:
  - Jede Transaktion arbeitet zu einem Zeitpunkt nur auf einer Site.

- Jede Transaktion wird der Site zugeordnet, auf der sie gestartet wurde, also auf der ihr Root-Agent läuft.
- Auf jeder Site gibt es im lokalen WFG einen ausgezeichneten Knoten, External (EX).

  Dieser repräsentiert den Teil des Graphen, der der Site nicht bekannt ist.

Algorithmus von Obermarck (2)

- Für jeden externen Agenten (Subtransaktion einer Transaktion T_i) wird

  $$EX \rightarrow T_i$$

  in den lokalen WFG aufgenommen.

  Grund: Auf einer anderen Site könnte jemand auf die Sperren von T_i warten.

- Für jeden Root-Agent, der einen Agenten auf einer anderen Site gestartet hat, wird

  $$T_i \rightarrow EX$$

  in den lokalen WFG aufgenommen. T_i ist dabei die Transaktion des Root-Agenten.

  Grund: T_i könnte auf einer anderen Site auf jemanden warten.

Algorithmus von Obermarck (3)

Verbesserungen gegenüber dem „normalen“ path-pushing:

- Es werden nur Pfade verschickt, die einen Zyklus bilden in dem EX vorkommt.
- Pfade werden an die Sites geschickt, auf die die letzte Transaktion vor EX wartet.
- Ein Pfad $EX \rightarrow T_1 \rightarrow \ldots \rightarrow T_n \rightarrow EX$ wird nur verschickt, falls $T_1 > T_n$.

  $\longrightarrow$ Reduziert die Anzahl der Nachrichten um die Hälfte.

Wie sieht es mit der Korrektheit aus?
(Übung)

Edge-Chasing Algorithmen

- WFG wird nicht explizit aufgebaut.
- Ausgezeichnete Nachrichten, *Probes*, werden entlang der Wartebeziehungen geschickt.
- Ein Deadlock existiert wenn die Probe, die die Initiator-Transaktion losgeschickt hat, wieder zurück kommt.

Algorithmus von Roesler und Burkhard

Literatur: Roesler und Burkhard, SIGMOD 1988

Wir wählen diesen Algorithmus, weil er:

- der schnellste in dieser Klasse ist,
- alle Deadlocks findet (trifft häufig nicht zu), und
- semantisches Locking „verträgt“.

Algorithmus von Roesler und Burkhard (2)

Algorithmus:

- Wenn eine Transaktion T_1 auf ein Objekt warten muß, schickt das *Objekt* eine initiale Probe an alle Transaktionen, auf die T_1 wartet.

  T_1 wird Initiator genannt.

- Empfängt eine Transaktion eine Probe, schickt sie sie an das Objekt weiter, auf das sie wartet.

  Wenn sie nicht wartet, schickt sie nichts weiter.

- Empfängt ein Objekt eine Probe von T_1, schickt sie die an die Transaktionen weiter, auf die T_1 wartet.

  Bemerkung: Der Algorithmus hält in den Objekten genügend Informationen, um zu wissen, an wen die Probe zu schicken ist. Daher funktioniert es auch mit semantischen Sperren.

- Wenn eine Transaktion die Probe zurück bekommt, deren Initiator sie ist
  $\longrightarrow$ Deadlock

Algorithmus von Roesler und Burkhard (3)

- Optimierung:
  Eine Probe wird an T_j nur weitergeschickt, falls $Initiator \geq T_j$.
- Objekte und Transaktionen merken sich Probes, die bei ihnen vorbeikommen.

  Frage: Warum?

- Wie werden Probes gelöscht?
  Wenn eine Abhängigkeit verschwindet, wird eine Antiprobe geschickt, die Probes, die durch diese Abhängigkeit entstanden sind, löscht.

  Sie durchläuft dafür den selben Weg wie die Probes.

Ein Deadlock wird immer von der jüngsten Transaktion im Zyklus entdeckt.
Warum?