Transaktionen und Synchronisation konkurrierender Zugriffe

Fragestellungen

Aufgaben des Transaktionsmanagers

- Aktivieren von Transaktionen entsprechend den Anforderungen von Anwendungsprogrammen. Dabei wird ein Ablauf (miteinander verschränkter) Aktionen der Transaktionen erzeugt. Deshalb wird der Transaktionsmanager oft auch *scheduler* genannt.
- Synchronisation konkurrierender Zugriffe entsprechend des für die jeweilige Transaktion geforderten Isolationslevels.
- Sicherstellung der Dauerhaftigkeit beim Commit bzw. der Möglichkeit zum Rollback – in Zusammenarbeit mit der Komponenten für das Recovery
- Maßnahmen zum Beenden einer Transaktion bei Commit bzw. beim Rollback.

Bestandteile/Komponenten eines Transaktionsmanagers:

- Ablaufplaner (*scheduler* im engeren Sinn)
  zuständig für die Serialisierung von Anweisungen
- Sperrmanager (*lock manager*)
  Sicherstellung der Isolation zwischen verschiedenen Transaktionen[1]
- Der Transaktionsmanager verwendet natürlich die Pufferverwaltung und das Zugriffssystem.

Grundlagen der Synchronisation konkurrierender Zugriffe

Erinnerung an die Veranstaltung „Datenbanken und Informationssystem“ – siehe Burkhardt Renz: *Serialisierbarkeit* Folien zur Vorlesung, http://homepages.thm.de/~hg11260/dis.html

[1] Die Implementierung von Isolation und Isolationsleveln durch Sperrmechanismen ist nur *eine* Variante der Implementierung, siehe Burkhardt Renz: *Isolationslevel in SQL* Vorlesungsskript, http://homepages.thm.de/~hg11260/dis.html.

Grundlagen

- Serielle Abläufe, Konflikte, Konfliktserialisierbarkeit
- Präzedenzgraph und Kriterium für Konfliktserialisierbarkeit
- Protokoll: 2-Phasen-Lock-Protokoll (2PL) garantiert Konfliktserialisierbarkeit
- Verklemmungen (*dead locks*): Wartegraph zur Erkennung und Auflösung von Verklemmungen

Isolationslevel in SQL

- Definition von Phänomenen, die beim konkurrierenden Zugriff auftreten können: *lost update*, *dirty read*, *non-repeatable read*, *phantom row*
- Definition der Isolationslevel in SQL durch Ausschluss des Auftreten solcher Phänomene (Garantie für die jeweilige Transaktion):
  - `READ UNCOMMITTED`,
  - `READ COMMITTED`,
  - `REPEATABLE READ`,
  - `SERIALIZABLE`
- Implementationsvarianten: Sperrmechanismen bzw. Multiversionierung kombiniert mit Sperrmechanismen.
  Unterschied IBM DB2 und Oracle in dieser Beziehung.

Sperrverfahren

Aufgabe des Sperrmanagers

- Setzen/Freigabe von Sperren gemäß 2PL. Entsprechend müssen Transaktionen in den Wartezustand versetzt werden, bzw. wieder angestoßen werden.
- Der Sperrmanager benötigt eine geeignete Datenstruktur für die Verwaltung von Sperren, die Sperrtabelle (*lock table*).

Sperreinheiten und hierarchische Sperren

Die Frage der Sperreinheiten bezieht sich auf die *Granularität* von Sperren, also welche Teile der Datenbank gesperrt werden. Dieses Thema betrifft nicht die Frage, *ob* die Versprechungen des Isolationslevels korrekt garantiert werden. Bei der Implementierung von Isolationsleveln durch Sperrmechanismen fallen Kosten der Synchronisation für *andere* Transaktionen an. Die Granularität der Sperren hat wesentliche Auswirkungen auf diese Kosten der Synchronisation.

Man unterscheidet zwischen der *logischen* Sicht auf die Granularität und der *physischen* Sicht:

Logische Sicht:

- Datenbank
- Tabelle
- Tupel
- Attribut

Physische Sicht:

- Sequenz von Seiten/Blöcken
- Seite/Block
- Datensatz
- Datenfeld

DBMS verwenden oft das Konzept der *hierarchischen* Sperren: Der Sperrmanager kann Sperren verschiedener Granularität setzen, je nachdem welche Teile der Daten durch die aktuelle Transaktion betroffen sind.

Wenn eine Transaktion nur auf wenige Seiten/Blöcke einer Datei zugreift, sollte sie möglichst nur diese Blöcke sperren, nicht aber die gesamte Datei; und entsprechend für die anderen Stufen der Granularität.

Bei hierarchischen Sperren bedeutet eine *explizite* Sperre auf einer höheren Ebene der Hierarchie automatisch *implizite* Sperren auf allen darunter liegenden Objekten in der Hierarchie. Typischerweise wird dabei folgende Hierarchie verwendet:

- Datenbank

- Datei
- Block
- Datensatz

Arten von Sperren

Sperrmanager verwenden für ihr Sperrprotokoll oft folgende Arten von Sperren:

Lesesperre *rl* **(*read lock*, auch *shared lock*)** Eine Sperre, die nicht exklusiv ist, d.h. andere Transaktionen können die gesperrten Objekte lesen, nicht aber schreiben.

Schreibsperre *wl* **(*write lock*, auch *exclusive lock*)** Eine exklusive Sperre, d.h. andere Transaktionen haben weder lesenden noch schreibenden Zugriff auf die gesperrten Objekte.

Intentions-Lesesperre *irl* **(*intention read lock*)** Eine Sperre, die besagt, dass die Transaktion beabsichtigt, Lesesperren auf *feinere* Einheiten der Sperrhierarchie zu erhalten.

Intentions-Schreibsperre *iwl* **(*intention write lock*)** Eine Sperre, die besagt, dass die Transaktion beabsichtigt, Schreibsperren auf *feinere* Einheiten der Sperrhierarchie zu erhalten.

Lese-Intentions-Schreibsperre *riwl* **(*read intention write lock*)** Eine Lesesperre zusammen mit einer Intentions-Schreibsperre.

Der Sinn der Intentionssperren: Um ein Objekt x z.B. lesend zu sperren, muss der Sperrmanager zunächst alle Oberobjekte von x mit einer *ir*-Sperre versehen, um sicherzustellen, dass nicht wegen einer bestehenden Sperre auf einem Oberobjekt die gewünschte Sperre verweigert werden muss. D.h. Intentionssperren werden verwendet, um die Hierarchie der Objekte samt ihren Sperren zu verwalten.

Ein spezielles Verfahren gilt für die Lese-Intentions-Schreibsperren *riwl*: Häufig hat man die Situation, dass eine Transaktion alle Datensätze einer Datei lesen und einige dieser Datensätze verändern möchte. Solch eine Transaktion benötigt eine Lesesperre auf der Datei und außerdem für die Schreibsperren auf den Datensätzen eine *iw*-Sperre auf der Datei. Für diese Situation gibt es die kombinierte *riwl*-Sperre.

Kompatibilitätsmatrix für Sperren

Die Matrix in Tabelle 1 beschreibt, wann eine Transaktion T_j eine Sperre erhält (Eintrag in der Zelle 1) oder nicht erhält (Eintrag in der Zelle 0), wenn eine andere Transaktion T_i bereits eine bestimmte Sperre hält.

Tabelle 1: Kompatibilitätsmatrix für Sperren

T_i hält	Transaktion T_j fordert an				
	rl	wl	irl	iwl	$riwl$
rl	1	0	1	0	0
wl	0	0	0	0	0
irl	1	0	1	1	1
iwl	0	0	1	1	0
$riwl$	0	0	1	0	0

Regeln des hierarchischen Sperrprotokolls

1. Wenn x nicht die höchste Ebene der Speicherhierarchie ist und T_j möchte eine Lesesperre $rl_j(x)$ oder eine Intentions-Lesesperre $irl_j(x)$ setzen, muss sie eine Intentions-Sperre (zum Lesen oder Schreiben) auf dem Vater von x besitzen.

2. Wenn x nicht die höchste Ebene der Speicherhierarchie ist und T_j will eine Schreibsperre $wl_j(x)$ oder eine Intentions-Schreibsperre erreichen, muss sie eine Intentions-Schreibsperre auf dem Vater von x haben.

3. Um x lesen zu können, benötigt T_j eine Lesesperre $rl_j(x)$ (explizite Sperre) oder eine Lesesperre auf einem Vorgänger von x (implizite Sperre).

4. Um x schreiben zu können, benötigt T_j eine Schreibsperre $wl_j(x)$ (explizite Sperre) oder eine Schreibsperre auf einem Vorgänger von x (implizite Sperre).

5. Eine Transaktion darf eine Intentionssperre auf einem Objekt x nicht freigeben, solange sie noch eine Sperre auf einem Nachfolger hält.

Natürlich muss der Sperrmanager außerdem 2PL befolgen:

- 2PL gibt an, *wann* Objekte zu sperren, bzw., freizugeben sind.

- Das Sperrprotokoll für hierarchische Sperren, wie eben beschrieben, gibt an, *wie* in der Hierarchie eine Sperre zu setzen bzw. freizugeben ist.

Bemerkungen

1. Umwandlung von Sperrarten:
   In vielen Fällen ist es gar nicht nötig, Sperren erst freizugeben und dann neue Sperren anzufordern, sondern eine vorhandene Sperre kann *umgewandelt* werden. Dabei muss der Sperrmanager natürlich das Sperrprotokoll auch einhalten.

   Beispiel: Man kann $irl_j(x)$ in $rl_j(x)$ umwandeln, d.h. die Sperre wird durch die Umwandlung *verstärkt*.

2. Sperranpassung (*lock escalation*):
   Der Sperrmanager kann die Granularität verändern. Stellt er z.B. fest, dass eine Transaktion sehr viele Blöcke einer Datei zu sperren sucht, dann kann er sich dazu entschließen, die Sperre gleich für die ganze Datei zum machen, d.h. auf einer höheren Ebene der Sperrhierarchie. Er müsste ja eh die Intentions-Sperren setzen und erspart sich mit der Sperranpassung Verwaltungsaufwand.

Sperrverfahren für B^+-Bäume

Das gängige 2PL-Protokoll eignet sich nicht für B^+-Bäume. Denn Zugriffe auf den B^+-Baum erfolgen ja stets über die Wurzel, man braucht also immer wieder Sperren auf der Wurzel – ein Flaschenhals.

Deshalb verwendet man für B^+-Bäume oft das sogenannte *Write-only tree locking protocol* (WTL-Protokoll):

1. Für alle Knoten außer der Wurzel erhält eine Transaktion nur dann eine exklusive Sperre, wenn sie bereits eine solche auf dem Vorgänger hat.
2. Wird eine Sperre auf einem Knoten freigegeben, darf eine Transaktion niemals wieder eine Sperre auf diesem Knoten verlangen bzw. bekommen.

Wie setzt man dieses Protokoll ein?

- WTL bei der Suche (ohne Änderung des Baums):
  Die Lesesperre auf dem Vorgänger von x wird freigegeben, sobald man die Lesesperre auf x hat. Dadurch wandern die Lesesperren entlang des Suchpfades im B^+-Baum.

- WTL bei Änderungen im Baum:
  Wenn ein Knoten nicht voll ist, erfolgt durch eine Änderung keine Teilung des Knotens. Ebenso wird keine Verschmelzung von Knoten stattfinden, wenn die Einträge im Knoten durch die Änderung nicht auf weniger als die Hälfte der möglichen Einträge verringert wird. In diesen Fällen benötigt man also keine Schreibsperren auf dem Vorgänger des Knotens – die Transaktion kann sie freigeben.

  Komplizierter wird die Sache, wenn Teilung oder Verschmelzen von Knoten notwendig wird. Dann darf die Transaktion nur die Knoten freigeben, die nicht geteilt oder verschmolzen werden.

Was erreicht man durch das WTL-Protokoll?

- Alle Transaktionen müssen von der Wurzel her arbeiten.
- Keine Transaktion kann eine andere auf demselben Pfad im Baum „überholen“, wodurch man Serialisierbarkeit erreicht.
- Es kann keine Verklemmung geben.

Burkhardt Renz
TH Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.2 – 15. Mai 2014