Grundlagen Datenbanken

B.SC. TUM WS 2010/11
Dozent: Proffesor Neumann, Übungleitung Henrik Mühe

Janosch Maier

3. März 2011

Inhaltsverzeichnis

1 Datenbanksysteme

1.1 Vorteile von DBS

- Datenunabhängigkeit (Anwendung unabhängig von Datenstruktur)
- Deklarative Anfragesprache (Vermeidung von Fehlern)
- Mehrbenutzersynchronisation (DBS verwaltet gleichzeitige Zugriffe)
- Fehlerbehandlung (Logging)
- Datenintegrität (Schutz vor Fehlern)
- Effizienz & Skalierbarkeit (DBS für große Datenvolumen)

2 Datenbankentwurf

2.1 Entity-Relationship-Modell

- Entität (Objekt) ⟶ Rechteck
- Attribut (Werte einer Entity) ⟶ Ellipse
- Schlüssel (Definiert Entität) ⟶ Unterstrichenes Attribut
- Relationship (Beziehung) ⟶ Raute
- Rolle ⟶ Verdeutlichung der Beziehung

Funktionalitäten 1:1, 1:n, n:m – beziehen sich auf die Relation

Multiplizitäten beziehen sich auf die minimale / maximale Anzal des Objektes ⟶ $\forall e_i \in E$ gibt es mindestens x und maximal y Tupel, die e_i enthalten. (Ausprägung Werte in der Tabellen)

Schwache Entität Braucht eine starke Entität um eindeutig identifiziert zu werden. Bsp: Raum in Gebäude ⟶ Doppelrahmen

Generalisierung Vererbung von Attributen

Entwurfsentscheidungen Entität > Beziehung > Attribut

Ternäre Beziehungen Benötigen alle drei teilnehmenden Entitäten. Lassen sich nur mit Semantikverlust in zwei Binäre Beziehungen zerlegen.

Connection Trap Redundate / Zyklische Beziehung

2.2 Übersetzung ER in Relational

Entität zu Relation Jedes Attribut der Entität wird Attribut der Relation. Schlüssel der Entität wird Primärschlüssel

Beziehung zu Relation Schlüssel der teilnehmenden Relationen werden Attribute der Relation.
Schlüssel:

- N:M Beziehung ⟶ Beide Entitätsschlüssel,
- 1:N ⟶ Schlüssel der N-Seite,
- 1:1 ⟶ beliebiger Schlüssel

Schemavereinfachung Relationen mit gleichem Schlüssel werden zusammengefasst.

Schwache Entitäten Zusammengesetzter Schlüssel

Generalisierung Nur Obertyp / Nur Untertypen / Alle Typen

3 Das Relationale Modell

3.1 Aufbau

- Relationen R
  - Schema (Struktur der Relation) ⟶ Jedes Attribut hat eine Domäne (Datentyp D)
  - Instanz (Inhalt der Relation)
- $R \subseteq D_1 \times D_2 \times ... \times D_n$
- |R| = Kardinalität (Anzahl der Tupel)

3.2 Relationale Algebra

Mengenorientiert und abgeschlossen

Projektion $\pi_{A_1,...,A_n}(R)$ Wählt Spalten A_1 bis A_n aus Relation R aus. Duplikate werden eliminiert.

Umbenennung $\rho_{NeuerName \leftarrow A_1}(R_1)$

Selektion $\sigma_p(R)$ Wählt alle Tupel aus R aus, die p erfüllen.

Kreuzprodukt $R_1 \times R_2$ Verbindet Jedes Tupel aus R_1 mit jedem Tupel aus R_2.

Join $R_1 \bowtie_{R_1.A_i=R_2.A_j} R_2 = \sigma_{R_1.A_i=R_2.A_j}(R_1 \times R_2)$ Attributnamen müssen eindeutig sein.

Natürlicher Join $R_1 \bowtie R_2$ Join, der Attribute mit gleichem Namen vergleicht

Outer Join $R_1 \supset\!\bowtie\!\subset R_2$ Tupel, die keinen Join-Partner fingen, bleiben erhalten. (Rechter/Linker Outer Join, nur für jeweils eine Seite)

Semi-Join $R_1 \ltimes R_2$ Prüft auf Join Bedingung, behält aber nur Tupel aus R_1 (Rechter Semi-Join analog)

Anti-Join $R_1 \setminus (R_1 \ltimes R_2)$ Bestimmt alle Tupel aus R_1, die keinen Join Partner haben

Mengenoperationen Bei gleichem Schema: $\cup \cap \setminus$

Relationale Divison $R_1 \div R_2$ ergibt die Attribute aus R_1, die in jeder Kombination mit den Attributen aus R_2 in R_1 vorkommen.

3.3 Relationales Tupelkalkül

$\{r | r \in R_1$
$\wedge \exists s \in R_2(r.i = s.i$
$\wedge \exists p \in R_3(s.j = p.j$
$\wedge p.k = \text{`}x\text{`}))\}$
Sicherheit: Ergebnis muss Teilmenge der Domäne der Formel sein

3.4 Relationales Domänenkalkül

$\{[n, m] | \exists s([m, n, s] \in R_1$
$\wedge \exists p, v([n, p, v] \in R_2$
$\wedge [p, v, \text{`}x\text{`}] \in R_3))\}$
Sicherheit: Ergebnistupel muss in der Domäne von P enthalten sein.

4 SQL

4.1 Einfache Anfragen

SELECT Attribute FROM Relationen WHERE Prädikat;
Duplicate werden mit DISTINCT eliminiert. SELECT DISTINCT Attribut FROM Relation;

Where Statement Verknüpfung: AND, OR, NOT, =, <, <=, >, >=
Vergleiche: BETWEEN, LIKE, IS NULL
Wildcards: _ (Ein beliebiges Zeichen), % (Eine beliebige Zeichenkette, auch)
Strings: mit ‘ eingeschlossen

4.2 Mehrere Relationen

Kreuzprodukt
SELECT * FROM R_1, R_2;

Join
SELECT * FROM R_1 [natural | left outer | right outer | full outer] join R_2 [on R_1.A = R_2.B];

Umbenennung
SELECT * FROM R_1 r, R_2 s;

4.3 Mengenoperationen

Gleiches Schema der Relationen benötigt!

Vereinigung
SELECT * FROM R_1 UNION SELECT * FROM R_2;
Duplikate werden eliminiert (da Mengenoperation). Falls duplikate erwünscht: UNION ALL

Schnitt
SELECT * FROM R_1 INTERSECT SELECT * FROM R_2;

4.4 Sortierung

SELECT * FROM R_1 ORDER BY a [ASC | DESC];
ASC: Aufsteigend (Kleinster Wert zuerst)
DESC: Absteigend (Größter Wert zuerst)

4.5 Geschachtelte Anfragen

unkorreliert (gut) / korreliert (Unteranfrage braucht Attribute der äußeren Anfrage; schlecht)

Select in Where Klausel
SELECT * FROM R_1 WHERE R_1.a IN (SELECT a from R_2 WHERE b = ‘x‘);
SELECT * FROM R_1 WHERE EXISTS (SELECT * FROM R_2);

Select in Select Klausel Inneres Select darf nur ein Tupel mit einem Attribut zurück liefern

Select in From Klausel Korrelierte Unteranfragen in From Klausel oft nicht erlaubt

4.6 Aggregatsfunktionen

- count()
- sum()
- avg()
- max()
- min()

Gruppieren Tupel in Gruppen aufteilen, und diese getrennt aggregieren.
SELECT a, COUNT(*) AS Anzahl FROM R_1 GROUP BY a;
Attribute, die nicht in der Group By Klausel stehen, dürfen nur aggregiert in der Select Klausel stehen.

Having Where Klausel wird vor Group By ausgewertet. Für Filter nach Group By wird having verwendet.
SELECT a, COUNT(*) AS Anzahl FROM R_1 GROUP BY a HAVING COUNT(*) > 5;

4.7 Sichten

Virtuelle Relationen; Einschränkung des Zugriffs (für Benutzergruppen), aber evtl. keine Änderungsoperationen möglich.
CREATE VIEW counter AS SELECT COUNT(*) AS c FROM R_1;
Nur Anweißung, wie Daten abgefragt werden sollen. Für Caching: CREATE MATERIALIZED VIEW. Nicht standadisiert.

WITH Sicht wird nur für einen Query erstellt.
WITH tmp_1 AS (SELECT * FROM R_1), tmp_2 AS (SELECT COUNT(*) AS c FROM R_1
SELECT * FROM tmp_1, tmp_2;

4.8 Datenmanipulation

Einfügen
INSERT INTO R_1 VALUES (V_1, V_2, V_3);
INSERT INTO R_1 (a, c) VALUES (V_1, V_3);
INSERT INTO R_1 (a, c) SELECT V_1, V_3 FROM R_2 WHERE id = 1;

Ändern
UPDATE R_1 VALUES SET a = V_1, b = V_2 WHERE id = 1;

Löschen
DELETE FROM R_1 WHERE id = ‘1‘;

Sichten Nur änderbar, wenn Sicht nur eine Basisrelation, mit Schlüssel, ohne Aggregate, Gruppierungen, Duplikateliminierungen

4.9 Datendefinition

Definition des Schemas; Kontrolle des Zugriffs auf Daten

Relationen anlegen

CREATE TABLE R_1 (id integer, a varchar(80) DEFAULT 'abc' , b boolean, PRIMARY KEY (a));

Einschränkungen: PRIMARY KEY, NOT NULL, UNIQUE, CHECK

Check Klausel

CHECK (b > 0 AND b < 1000),
CHECK (b NOT IN (SELECT b FROM R_2)),

Referentielle Integrität FOREIGN KEY (a) REFERENCES R_2(a) [ON DELETE | ON UPDATE SET NULL | CASCADE]

Referenzen können i.A. nicht gelöscht werden. Set null setzt Werte beim Löschen auf Null, Cascade löscht Werte, wenn Referenz gelöscht.

Indizies

CREATE [UNIQUE] INDEX i ON TABLE R_1 (a [ASC | DECS], b [ASC | DESC]);

Objekte entfernen

DROP TABLE R_1;
DROP VIEW v;
DROP INDEX i;

4.10 Varianten von SQL

SQL kann in andere Programmiersprachen eingebettet werden, aber SQL ist Mengenorientiert.

Embedded SQL Befehle werden in Hostsprache eingebettet und mit EXEC SQL markiert. Präprozessor ersetzt Befehle durch richtige Befehle der Hochsprache

Dynamic SQL Anfragen zur Übersetzungszeit noch nicht bekannt; Schnittschtellen wie ODBC/JDBC (Open/Java Database Connectivity); Flexibler aber i.A. langsamer als Embedded SQL.

5 Relationale Entwurfstheorie

5.1 Funktionale Abhängigkeiten (functional dependency)

FD: a $\longrightarrow$ b gdw. für alle Instanzen von R gilt: für alle Paare von Tupeln $r, t \in R$ gilt $r.a = t.a \Rightarrow r.b = t.b$

Schlüssel

k ist (Super)Schlüssel von R, gdw. $k \subseteq R$, $k \longrightarrow R$ (Vollständig)
k ist Kandidatschlüssel gdw. $\nexists k' \subset k; k \longrightarrow R$ (Minimal)
Wähle einen Primärschlüssel als Kandidatschlüssel
F^+ ist die Hülle von, mit algebraischen Mitteln, aus F herleitbaren, FDs

5.2 Normalenformen

1NF, 2NF, 3NF, BCNF, 4NF

1NF Attribute haben nur atomare Werte (Keine Listen als Wert)
Bsp: R(a, b, c, d); a $\longrightarrow$ c, b $\longrightarrow$ d

2NF Jedes Nichtschlüsselattribut (NSA) hängt voll funktional von jedem Schlüssel ab; Ein NSA ist nicht funktional von einem Teilschlüssel (Keine Mischung von Beziehungen)
Bsp: R(a, b, c, d); ab $\longrightarrow$ cd, c $\longrightarrow$ d

3NF Für jede FD $a \longrightarrow b$ muss eine der folgenden Bedingungen gelten:

- $a \longrightarrow b$ ist trivial ($b \subseteq a$)
- a ist Superschlüssel
- Jedes Attribut in b ist in einem Schlüssel enthalten

(Es darf keine FD bestehen $a \longrightarrow b$ mit a, b sind NSA)
Bsp: R(a, b, c, d); ab $\longrightarrow$ cd, c $\longrightarrow$ a

BCNF Für jede FD $a \longrightarrow b$ muss eine der folgenden Bedingungen gelten:

- $a \longrightarrow b$ ist trivial ($b \subseteq a$)
- a ist Superschlüssel

(Alle Attribute hängen nur noch von einem Superschlüssel ab)
Bsp: R(a, b, c, d); a $\twoheadrightarrow$ c

4NF Für jede mehwertige Abhängigkeit (multivalued dependency) MVD $a \longrightarrow b$ muss eine der folgenden Bedingungen gelten:

- $a \twoheadrightarrow b$ ist trivial ($b \subseteq a$) oder $a \cup b = R$
- a ist Superschlüssel

(Wie BCNF, nur für MVDs)
Bsp: R(a, b, c, d)
MVD: Für alle Tupel mit gleichem Wert a kommen alle Kombinationen mit b, d vor.

5.3 Zerlegungsalgorithmen

Schema soll in Teilschemata $R_1, ..., R_n$ zerlegt werden, die verlustlos ($R = R_i \bowtie R_j$) und abhängigkeitswahrend ($F_R \equiv F_{R_1} \cup ... \cup F_{R_2}$) sind

3NF-Synthesealgorithmus

- Kanonische Überdeckung
- Erstelle Schema für jede FD
- Erstelle Schema mit Kandidatschlüssel
- Entfern redundante Schemata

Kanonische Überdeckung F_c ist kanonische Überdeckung von F, wenn $F_c \equiv F$, keine FDs in F_c mit überflüssigen Attributen, jede linke Seite einer FD ist einzigartig.
Erstellen der kanonischen Überdeckung:

- Linksreduktion (Prüfe, ob sich ein linkes Element einer FD entfernen lässt, so dass die rechte Seite gleich bleibt)
- Rechtsreduktion (Prüfe, ob sich ein rechtes Element einer FD entfernen lässt, so dass es (in der Attributhülle der rechten Seite) wieder auftaucht
- Entfernen leerer FDs $a \longrightarrow \emptyset$
- Vereinigen von FDs mit gleicher linker Seite

Zerlegung in BCNF (Dekompositionsalgorithmus)

- Starte mit Z = R
- Solange $R_v \in Z$, das nicht in BCNF ist:
  - Finde eine für R_i geltende FD mit
    - $a \cup b \subseteq R_i$

    * $a \cap b = \emptyset$
    * $a \nrightarrow R_i$
  - Zerlege R_i in $R_{i_1} := a \cup b$ und $R_{i_2} := R_i - b$
  - Entferne R_i aus Z und füge R_{i_1} und R_{i_2} ein

Manche Schemate können nicht abhängigkeitswahrend in BCNF / 4NF zerlegt werden, Höhere Normalformen bevorzugen Update vor Select Operationen; Meist reicht 3NF.

6 Physische Datenorganisation

6.1 Speicherhierachie

vgl. Speicherhierachien in Computern. Hauptspeicher und Festplatten wichtigste Speichermedien für DBMS

6.2 Speicherung von Relationen

Tupel einer Relation werden auf mehrere Seiten verteilt im Hintergrundspeicher gespeichert. Jedes Seite hat eine Datensatztabelle mit Verweisen auf die enthaltenen Tupel.

Referenz der Tupel erfolgt über Tupel-Identifikatoren (TIDs)

6.3 Indizes

Indizes erlauben assoziativen Zufriff auf Daten; Nur Daten, die für eine Anfrage gebraucht werden, werden in den Haupspeicher geladen, da Laden aller Tupel sehr teuer

Hierachisch (Bäume)

Index-Sequential Access Method (ISAM)

Idee: Anlegen von Index-Seiten, Wenn Datenseite gesucht wird, zuerst nach passendem Index suchen (Indexseiten « Datenseiten), Referenz auf Datenseite.

Instandhaltung kann sehr teuer werden; Einfügen wenn Datenseite (+ Indexseite) voll ist führt zu verschiebungen; Indexseiten können wieder viel werden

B-Baum

Idee: Indexseiten für Indexseiten

Eigenschaften:

- Jeder Pfad Wurzel zu Blatt hat gleiche Länge
- Jeder Knoten (außer Wurzel) hat mindestens i und höchstens 2i Einträge
- Einträge sind sortiert
- Jeder Knoten (außer Blätter) mit n Einträgen hat n+1 Kinder

- Unterbaum links eines Eintrages enthält nur Einträge, die kleiner als dieser Eintrag sind (rechts: größer)

Einfügealgorithmus:

1. Schlüssel in

   Blattknoten einfügen

2. Wenn kein Platz ⟶ Knoten teilen, Median in Elternknoten, Zeiger anpassen

3. Kein Platz in Elternknoten. Wenn Wurzel ⟶ neuen Wurzelknoten erstellen und Median einfügen, sonst 2. Widerholen

Löschalgorithmus:

1. Eintrag löschen (in Blatt unproblematisch), sonst nächstgrößere/nächstkleinere Schlüssel aus Kindknoten in Knoten ziehen

2. Wenn Knoten unterbelegt (weniger als i Einträge), verschmelzung mit Nachbarknoten; ⟶ evtl. Unterbelegung in Elternknoten ⟶ Verschmelzung

Verschmelzung aufwendig und häufig nicht realisiert (DB wachsen eher, als schrumpfen)

B^+-Baum

Daten werden nur in Blattknoten gespeichert. Innere Knoten liefern nur Referenzen; Blattknoten sind meist verkettet

$\sim \log_k(n)$ Zugriffe um ein Element zu lesen (k = Verzweigungsgrad, n = Anzahl indexierter Datensätze)

Partitioniert (Hashing)

Hashtabelle $\sim$ 2 Seitenzugriffe um ein Element zu lesen

Erweiterbares Hasching

- Umgedrechte binärdarstellung
- Einfügen in Bucket (Lokale Tiefe gibt an, wie viele Bitstellen am Anfang gleich sind)
- Bei Overflow: Wenn lokale tiefe < globale tiefe (= Mehr als ein Pfeil auf Bucket) –> Lokale Tiefe erhöhen; Sonst Indextabelle verdoppeln (Globale Tiefe verdoppeln

7 Anfragebearbeitung

Da SQL deklarativ, Umsetzung in prozedurale Sprache; i.A. relationale Algebra

- SELECT ⟶ Projektion

- FROM $\longrightarrow$ Kreuzprodukt der Relationen
- WHERE $\longrightarrow$ Selektion
- $\Rightarrow$ Kanonische Übersetzung
- SELECT a, sum(d) AS s FROM ... GROUP BY a, b, c $\longrightarrow \pi_{a,s}(\rho_{a,b,c;s:sum(d)})(C)$, C ist kanonische Übersetzung des inneren Teils
- SELECT ... HAVING p $\longrightarrow \sigma_p(C)$, C ist kanonische Übersetzung inklusive Group by
- SELECT ... ORDER BY a, b, c $\longrightarrow sort_{a,b,c}(C)$, C ist kanonische Übersetzung inklusive having

Kanonischer Plan nicht sehr effizient, DBMS besitzt optimierer

7.1 Optimierung

Kosten werden nur abgeschätzt, Benutzer kann eventuell manuell optimieren

Logische Optimierung

Transformationen des relationalen Algebraausdruck der kanonischen Übersetzung

- Selektionen Aufbrechen und nach „unten“ schieben
- Selektionen und Kreuzprodukte zu Joins zusammenfassen
- Joinreihenfolge bestimmen
- Projektionen einfügen und nach „unten“ schieben

Physische optimierung

Umsetzunge der logischen Ausdrücke: Nutzen von Indizes, Zwischenergebnisse, etc.

Iteraturkonzept

Iterator Scan sucht linear alle Tupel ab, die Bedingung erfüllen

Iterator IndexScan sucht erstes Tupel, das die Bedingung erfüllt und iteriert so lange, bis ein Tupel die Bedingung nicht mehr erfüllt.

Join Reihenfolge

Joins sind häufig und teuer; Veränderung der Tupelzahl; Starke Laufzeitbeeinflussung $\longrightarrow$ Optimierung der Joinreihenfolge (meist nur Heuristiken)

Greedy-Heuristik: Beginne mit Relation, joine „günstigste„ Relation dazu, bis alle gejoint sind.

Oft: Versuche Selektion nach Join zu minimieren

Dynamisches Programmieren
Einfache Teilprobleme optimal Lösen, um damit kompliziertere Probleme zu lösen → Joine alle Tabellen, um optimale Lösung zu finden. Optimale Lösung, aber u.U. exponentielle Laufzeit

8 Transaktionsverwaltung

Transaktionsverwaltung zu Recovery, Synchronisation

8.1 Operationen

begin of transactiton (BOT)

commit Erfolgreiche Beendigung, Festschreiben der Änderungen

abort Selbstabbruch, Rücksetzen der Änderungen

define savepoint Sicherungspunkt, auf den Transaktionen Rücksetzbar sind

backup transaktion Transaktion auf letzten Sicherungspunkt zurücksetzten

8.2 Transaktionen

commit (work) Beende Transaktion, Festschreiben; Nur, wenn keine Fehler aufgetreten sind

rollback (work) Beende Transaktion und setze Änderungen zurück; DBMS muss rollback immer erfolgreich ausführen können

8.3 ACID

Atomicity „alles oder nichts“ wird ausgeführt

Consistency Wenn Datenbank vor Transaktion konsistent ist, ist sie auch danach konsistent

Isolation Nebenläufige Transaktionen haben keine Seiteneffekte

Durability Alle commiteten Änderungen sind dauerhauft (selbst bei Absturz)

9 Mehrbenutzersynchronisation

Serielle Ausführung sicher, aber langsam, TAs blockierend → Nebenläufigkeit. Aber Probleme Möglich.

9.1 Probleme

Lost Update $b_1, r_1(x), b_2, r_2(x), w_1(x), w_2(x), c_1, c_2 \rightarrow$ Ergebnis von T_1 geht verloren

Dirty Read $b_1, b_2, r_2(x), w_2(x), r_1(x), c_1, w_2 \rightarrow T_1$ liest einen ungültigen Wert für x

Non-Repeatable Read $b_1, r_1(x), b_2, w_2(x), c_2, r_1(x), ... \rightarrow T_1$ liest x mit verschiedenen Ergebnissen

Phantom Problem b_1, T_1 : select count(*) from R , b_2, T_2 : insert into R ... , c_2, T_1 : select count(*) from R, ... $\rightarrow T_1$ findet bei zweiter Anfrage weiteres Tupel

9.2 Isolation Leves

Vermeidung der Probleme

Idealerweise sollen alle Probleme vermieden werden; Aber Performance vs. Genauigkeit; Je höher die Sicherheit, desto langsamer; Isolation Levels legen Sicherheisstufe Fest

Isolation Levels einer Transaktion

set transaction Stufe, Zugriffsmodus
Stufen:

- read uncommitted (Vermeidet: lost update)
- read committed (Vermeidet: lost update + dirty read)
- repeatable read (Vermeidet: lost update + dirty read + non-repeatable read)
- serializable (Vermeidet: lost update + dirty read + non-repeatable update + phantom problem)

Zugriffsmodi

- read only
- read write

Nebebläufigkeit von TAs, die nur lesen ist unkritisch; Erst wenn read write, müssen Vorkehrungen getroffen werden

SQL

start transaction;

commit [work];

rollback [work];

9.3 Historien

Gibt an, wie Operationen aus verschiedenen TAs relativ zu einander ausgeführt werden können

Operationen einer Transaktion (TA)

- $r_i(A)$: Lesen von A
- $w_i(A)$: Schreiben von A
- a_i: Abbruch
- c_i: Commit
- bot: Begin of Transaction (implizit)

Darstellung von Transaktionen Oft als gerichteter azyklischer Graph

Konfliktoperationen Zwei Operationen auf dem gleichen Datenobjekt, von denen mindestens eine schreibend ist, stehen in Konflikt und dürfen nicht parallel ausgeführt werden.

Definiton von Historien

Eine Historie H ist eine partielle Ordung über einer Menge von Transaktionen T mit einer Ordnungsrelation <, bei der für zwei in Konflikt stehende Operation p, q gilt p < q oder q < p

(Konflikt-)Äquivalenz

Zwei Historien sind äquivalend, wenn sie die gleichen TAs enthalten und die Konfliktopertionen der nicht abgebrochenen TAs in der gleichen Reihenfolge anordnen (Das Ergebnis ist das gleiche)

9.4 Serialisirbarkeit

Historie, die äquivalent zu einer seriellen Historie ist; Abgeschlossene Projektion C(H) enthält nur abgeschlossenen TAs. H ist serialisierbar, wenn C(H) äquivalent zu einer seriellen Historie.

Überprüfung der Serialisierbarkeit Serialisierbar, wenn Seriealisierbarkeitsgraph SG(H) azyklisch. Knoten sind abgeschlossene TAs aus H, Kante, wenn Konfliktoperationen zwischen TAs

9.5 Rücksetzbarkeit

Wenn T_i von T_j liest (T_j schreibt), dann muss $c_j < c_i$ sein.

T_j muss vor T_i commiten. Wert von T_j ist gültig (wenn T_j abbricht $\longrightarrow$ T_i liest einen Wert, der nie existiert hat)

9.6 Vermeidung kaskadierenden Rücksetzens

Wenn T_i von T_j liest, $c_j < r_i(x)$

Bei Schlange von Lesezugriffen. Wenn erste Transkaktion abgebrochen wird, müssen alle nachfolgenden Transaktionen abgebrochen werden.

9.7 Striktheit

Bei $w_j(x) < o_i(x)$ (o = r oder w), dann: $a_j < o_j(x)$ oder $c_j < o_i(x)$

Wenn ein Wert geändert wird, muss die Transaktion fest geschrieben oder abgebrochen werden, bevor andere Transaktionen auf den Wert zugreifen dürfen.

9.8 Scheduler

Programm, das Operationen ordnet, und für serialisierbare und rücksetzbare Historie sorgt. Operationen können ausgeführt, zurückgewiesen oder verzögert werden.

Pessimistisch Operationen werden verzögert, später geschickte Reihenfolge wird festgelegt (Sperrbasierte Scheduler)

Sperren 2 Phasen Sperrprotokoll: shared (read lock), exclusive (write lock), TA braucht Sperre, um mit Objekt zu arbeiten. Sobald eine Sperre zurück gegeben würde, darf keine neue mehr angefordert werden. Bei Transaktionsende werden alle Sperren zurück gegeben.

Bei strengem Sperrprotokoll werden Sperren bis zum Ende der Transaktion behalten ⟶ strikte Schedules

Deadlocks Verklemmung von Sperren (2 Transaktionen können nicht zu Ende geführt werden, da sie auf eine Sperre warten, die die andere TA hält); Wartegraph (Wartet-Auf Kanten); Deadlock, wenn zyklisch ⟶ zurücksetzen

Vermeidung durch Preclaiming (Alle Sperren bei BOT anfordern), Setzen von Prioritäten (Möglichkeit von Livelocks, Vordrängeln ⟶ Verhinderung durch Zeitstempel beachtung)

Phantom Problem Nur Lösbar durch hierachische Sperrgranulate ⟶ Vererbung der Sperren auf ganze Datensegmente

Optimistisch Operationen werden möglichst ausgeführt, evtl. später zurücksetzen (Zeitstempelbasierter Scheduler)

Zeitstempelbasierte Verfahren TA bekommt Zeitstempel, den jede Operation der TA erhält; Zeitstempel werden genutzt, um Konflikte aufzulösen.

u.U. nicht Rücksetzbare Schdules, Vermeidung durch commiten der TAs in Zeitstempelreihenfolge

Probleme: Phantom-Problem nicht gelöst, Jede Operation wird quasi zur Schreiboperation

10 NoSQL

Not Only SQL bezeichnet Datenbanken, die mit dem relationalen Schema brechen. Sie brauchen keine Tabellenschemas und versuchen Joins zu vermeiden. Horizontale Skalierung. Auch: Strukturierte Datenspeicher

11 Mögliche Klausurfragen

- Ist eine Anfrage in SQL / Relationaler Algebra / Tupelkalkül / Domänenkalkül gültig? Wie ist die Kardinalität der Anfrage?
- Was ist kaskadierendes Zurücksetzen, und wie vermeide ich es?