TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einsatz und Realisierung von Datenbanksystemen* im SoSe19

Maximilian {Bandle, Schüle} (i3erdb@in.tum.de)
http://db.in.tum.de/teaching/ss19/impldb/

Blatt Nr. 09

Hausaufgabe 1

Schätzen sie die Anzahl der Cache Misses die entstehen, wenn man 1001 32-Bit Integer Werte (0-1000) in aufeinanderfolgender Reihenfolge in einen ART Baum einfügt. Wäre ein B+ Baum besser oder schlechter? Bei den Baumknoten müssen die Header nicht berücksichtigt werden, Pointer haben eine Größe von 64 Bit.

Größe der einzelnen ART Knoten (mit 64-Bit Pointern und ohne Header):

Node4 $4 + 4 * 8 = 36$ Byte

Node48 $256 + 48 * 8 = 640$ Byte

Node256 $256 * 8 = 2048$ Byte

Die Höhe eines ART-Baums ist durch die Schlüssellänge beschränkt (in unserem Fall maximal Höhe 4), da in jedem Knoten ein Byte des Schlüssels gespeichert wird. Da die Integerzahlen aufeinanderfolgend sind, unterscheiden sie sich maximal in den letzten zwei Bytes (die Werte zwischen 0 und 1000 haben immer 0x00 0x00 als Präfix). Für die ersten zwei Bytes reicht es einen Node4 zu nehmen, da hier alle Einträge den selben Wert besitzen. Auf dem letzten Level reichen 4 Node256 um die letzten Bytes der 1000 Integer Werte einzufügen. Da es nur vier Kindnoten gibt reicht auf Level drei auch ein Node4. Die Gesamtgröße des Baums ist somit $4 * 2048 + 3 * 36 = 8300$ Byte. Dies passt locker in den L1 Cache heutiger CPUs der typischerweiße 64 KB groß ist. Somit gibt es keine Cache Misses. Während der Baum gebaut wird sind auf der untersten Eben ursprünglich auch Node 4, die aber über Node 16, zu Node 48 zu Node 256 wachsen. Ein B+ Baum ist schlechter, da bei sequentiellen Einfügen die Knoten nur halb gefüllt sind. Außerdem werden in den Knoten jeweils pro Pointer auch noch ein ein kompletter 32-Bit Rangeschlüssel gespeichert was den Speicherbedarf zusätzlich erhöht.

Hausaufgabe 2

Sie sollen für die Alexander-Maximilians-Universität (AMU) ein Hauptspeicherdatenbanksystem optimieren. In dem System sind die Daten aller Studenten gespeichert. Schätzen Sie für jede der untenstehenden Anfragen einzeln, ob ein Row- oder Column-Store besser geeignet ist.

Relationen

Studenten: MatrNr (8 Byte), Name (48 Byte), Studiengang (4 Byte), Semester (4 Byte)

MatrNr ist der Primärschlüssel der indiziert ist.

Anfragen:

1. select * from Studenten;

Name	verfügbar	Versionsvektor
House	ja	-
Green	ja	-
Brinkmann	ja	-

Abbildung 1: Hauptspeicher Column-Store

2. select Semester, count(*) from Studenten group by Semester;
3. select Name, Studiengang, Semester from Studenten where MatrNr = 42;
4. select Studiengang from Studenten where MatrNr = 42;
5. select * from Studenten where Semester < 5;
6. select * from Studenten where Semester = 25;
7. insert into studenten values(4242, Max Meyer, Info, 7);

Lösung:

1. Beides optimal: Gesamte Tabelle wird gelesen
2. Column-Store: Nur Spalte Semester wird gelesen. Bei Row Store muss die gesamte Tabelle gelesen werden.
3. Row-Store: Zeile passt in eine Cache Line (CL). Bei Column Store müssen mindestens 3 CLs gelesen werden.
4. Theoretisch Column-Store: Nur Studiengang muss gelesen werden, da ein Index existiert. Row-Store aber in der Realität gleich schnell, da Daten immer CL granular gelesen werden.
5. Beides optimal: Es muss die gesamte Tabelle gelesen werden, da das Prädikat nicht sehr selektiv ist.
6. Column-Store: Prädikat ist hoch selektiv. Es wird wahrscheinlich kein Student gefunden. Demnach müssen die anderen Felder nicht nachgeladen werden.
7. Row-Store: Zeile passt in eine CL. Damit muss beim Einfügen nur eine CL in den Speicher geschrieben werden. Bei Column Store müsste jedes Element einzeln geschrieben werden.

Gruppenaufgabe 3

Beschäftigen wir uns mit *Multi-Version Concurrency Control* am Beispiel unserer verfügbaren Ärzte („Doctors on call/duty“), in dem wir sicherstellen wollen, dass immer mindestens ein Arzt verfügbar ist.

Uns stehen drei Operationen zur Verfügung, $\sum$ zählt alle verfügbaren Ärzte, $(X)++$ ändert Xs Status in verfügbar, $(X)--$ zählt alle verfügbaren Ärzte und ändert Xs Status auf nicht verfügbar, wenn mindestens ein Arzt noch anwesend ist.

1. Welche Bedingungen gelten für die Zeitstempel?

TID	Startzeit	Commitzeit	Aktion
Ta	$T0$	-	$\sum$
Tb	$T2$	-	$(Green)--$
Tc	$T3$	-	$\sum$
Td	$T5$	-	$\sum$

Abbildung 2: Transaktionen (bereits committete gekennzeichnet durch eine Commitzeit)

$$\forall t \in TID, s \in Startzeit.s < t$$

Wir haben zwei Uhren, eine sowohl für die Startzeit wie die Commitzeit und eine für die TIDs, die immer größer ist. Bevor eine Änderungstransaktion committet, erhält der Eintrag im Undo-Puffer die TID als Zeitstempel.

2. Green möchte zum Zeitpunkt $T2$ seinen Feierabend antreten. Vervollständigen Sie Tabelle 2 und legen Sie einen geeigneten Undo-Puffer (Zeitstempel, Attribut, Undo-Image) an. Wann muss Tb committen, damit Td bereits die Änderung von Tb liest? Was lesen Ta und Tb?
   Tb ändert den Status von Green in-place und legt in einem Undo-Puffer das Before-Image ab. Im Versionsvektor speichert er eine Referenz auf den Eintrag im Undo-Puffer. Als Zeitstempel erhält der Eintrag die TID der aktuellen Transaktion Tb.

Name	verfügbar	Versionsvektor
House	ja	-
Green	nein	$*uTb0$
Brinkmann	ja	-

Undo-Puffer:

UID	TID	nextUID	Zeitstempel	Attribut	Undo
$uTb0$	Tb	-	Tb	verfügbar	ja

   Die Transaktion steht in keinem Konflikt zu anderen und kann committen. Der Zeitstempel im Undo-Puffer wird auf die aktuelle Commitzeit aktualisiert und somit ist die Änderung sichtbar für neustartende Transaktionen. Ta und Tb lesen den Wert, der zum Zeitpunkt $T0$ bzw. $T2$ gültig ist. Dazu navigieren sie durch den Undo-Puffer, bis $Zeitstempel \leq Startzeit$ oder kein Vorgängereintrag existiert, und lesen den für sie gültigen Eintrag. Somit kann Td den geschriebenen Wert nur lesen, wenn er mit $T4$ vor $T5$ committet worden ist. Sowohl $T4$ als auch Tb sind größer als $T0$ bzw. $T1$ und sie lesen den Wert aus dem Undo-Image. Da kein Schreibvorgang erfolgt ist, zählt bei Td die Startzeit und es ist irrelevant, dass zwischen $T3$ und $T6$ eine Änderung im Prädikatbereich erfolgt ist. Ta, Tc zählen drei verfügbare Ärzte, Td zwei, falls Tb vorher committet hat.

TID	Startzeit	Commitzeit	Aktion
Ta	$T0$	$T0$	$\sum$
Tb	$T2$	$T4$	$(Green)--$
Tc	$T3$	$T3$	$\sum$
Td	$T5$	$T5$	$\sum$

UID	TID	nextUID	Zeitstempel	Attribut	Undo
$uTb0$	Tb	-	$T4$	verfügbar	ja

3. Brinkmann und House wollen zeitgleich den Feierabend antreten. House startet bei

$T8$, Brinkmann bei $T9$. Wer darf gehen? Wie sorgt *Precision Locking* dafür, dass nur ein Arzt das Krankenhaus verlässt? Vervollständigen Sie die Einträge.

Zuerst starten beide Transaktionen.

TID	Startzeit	Commitzeit	Aktion
Ta	$T0$	$T0$	$\sum$
Tb	$T2$	$T4$	$(Green)--$
Tc	$T3$	$T3$	$\sum$
Td	$T5$	$T5$	$\sum$
Te	$T8$	-	$(House)--$
Tf	$T9$	-	$(Brinkmann)--$

UID	TID	nextUID	Zeitstempel	Attribut	Undo
$uTb0$	Tb	-	$T4$	verfügbar	ja
$uTe0$	Te	-	Te	verfügbar	ja
$uTf0$	Tf	-	Tf	verfügbar	ja

Anschließend werden alle Transaktionen berücksichtigt, deren Commitzeit zwischen der eigenen Startzeit und dem gezogenen Commitzeitstempel liegt. Dazu werden die in den zugehörigen Undo-Puffern vorgefundenden Änderungen auf Überlappungen mit dem eigenen Prädikatenraum überprüft. In unserem Beispiel entspricht der Prädikatenraum allen verfügbaren Ärzten.

O.B.d.A. zieht Te den Commitzeitstempel $T10$, Tf zieht $T11$. Zwischen $T8$ und $T10$ gab es keinen Änderungen in diesem Prädikatenraum, Te kann committen. Zwischen $T9$ und $T11$ liegt der Commitzeitstempel $T10$. Dazu müssen wir im Undo-Puffer von Te nachsehen und stellen fest, dass das Attribut *verfügbar* mit vorherigem Wert *ja* im Prädikatenraum liegt. Somit haben wir einen Konflikt aufgedeckt und setzen Tf zurück. In diesem Beispiel ist also die Transaktion erfolgreich, die den kleineren Commitzeitstempel erhält. Ein möglicher Ausgang sieht wie folgt aus:

Name	verfügbar	Versionsvektor
House	nein	$*uTe0$
Green	nein	$*uTb0$
Brinkmann	ja	-

TID	Startzeit	Commitzeit	Aktion
Ta	$T0$	$T0$	$\sum$
Tb	$T2$	$T4$	$(Green)--$
Tc	$T3$	$T3$	$\sum$
Td	$T5$	$T5$	$\sum$
Te	$T8$	$T10$	$(House)--$

UID	TID	nextUID	Zeitstempel	Attribut	Undo
$uTb0$	Tb	-	$T4$	verfügbar	ja
$uTe0$	Te	-	$T10$	verfügbar	ja

Hausaufgabe 4

Gegeben seien die folgenden Anfragen:

T1: insert into foo (select Note from Noten where MatrNr=12345)

T2: insert into bar (select count(*) from Noten where Note<1.5)

T3: insert into Noten(MatrNr,Note) values (54321, 3.0)

T4: update Noten set Note=1.4 where MatrNr=32154

T5: insert into Noten(MatrNr,Note) values (54321, 1.3)

T6: update Noten set Note=1.6 where MatrNr=12345

Analysieren Sie, ob die folgenden Historien unter dem MVCC Model, wie in der Vorlesung vorgestellt, auftreten können. Jede Historie steht für sich selbst und startet jeweils von einem ursprünglichen Datenzustand. Die Buchstaben innerhalb der Klammer entsprechen dabei jeweils den Tupeln auf die zugegriffen wird. Wenn in T2 z.B. drei Werte das 'Prädikat Note<1.5' erfüllen, gäbe es entsprechend drei $r(...)$ Einträge auf die jeweiligen Tupel.

H1 (T1 und T3): $bot_1, r_1(A), bot_3, w_3(B), w_1(C), commit_1, commit_3$
Kann so auftreten. Die Prädikatenräume von Anfrage 1 und 3 überschneiden sich nicht. Anfrage 1 greift nur auf das Tupel A (mit MatrNr. 12345) zu und Anfrage 3 fügt ein neues Tupel B mit MatrNr. 54321 in die Relation ein.

H2 (T2 und T3): $bot_2, r_2(A), bot_3, w_3(B), r_2(C), w_2(D), commit_2, commit_3$
Kann so auftreten. Anfrage 2 zählt wie oft es eine Note < 1.5 gab. Anfrage 3 fügt einen Studenten mit der Note 3.0 in die Relation hinzu. Zu Beginn der Historie befinden sich nur zwei Tupel mit Note < 1.5 in der Relation Noten: A, C. Bei der Ausführung der Historie liest Anfrage 2 zuerst das Tupel A, anschließend fügt Anfrage 3 das Tupel B mit der Note 3.0 hinzu und Anfrage 2 ließt das Tupel C. Da sich der Prädikatenraum von Anfrage 2 und Anfrage 3 nicht überschneiden kann auch Anfrage 3 committen.

H8 (T2 und T3): $bot_2, r_2(A), bot_3, w_3(B), r_2(C), commit_3, w_2(D), commit_2$
Kann so auftreten. Der eingefügte Wert ist außerhalb des Prädikatsbereichs der Anfrage 2, daher gibt es keinen Konflikt. Der durch Anfrage 3 eingefügte Wert ist außerhalb des Prädikatenraums der Anfrage 2, daher gibt es keinen Konflikt.

H3 (T2 und T4): $bot_2, r_2(A), r_2(B), bot_4, r_4(B), w_4(B), r_2(C), w_2(D), commit_2, commit_4$
Kann so auftreten. Bei 2PL wäre diese Historie nicht möglich, da B gesperrt wäre. Zu Beginn der Anfrage 2 haben nur die Tupel A und B eine Note < 1.5 und sind somit für diese Anfrage qualifiziert. Die Anfrage 2 liest die Tupel A und B. Dann liest Anfrage 4 Tupel B und updated die Note auf 1.4, anschließend liest Anfrage 2 das letzte Tupel C und committet. Anfrage 4 kann ebenfalls committen, da Anfrage 2 vor Anfrage 4 committet und Tupel B liest, bevor Anfrage 4 die Note auf 1.4 ändert. Somit entspricht das Ergebnis der Historie der seriellen Ausführung von Anfrage 2 gefolgt von Anfrage 4.

H5 (T2 und T4): $bot_2, r_2(A), bot_4, r_4(B), w_4(B), r_2(C), commit_4, w_2(D), commit_2$
Kann so nicht auftreten. Die Transaktion 2 würde abbrechen. Die Relation Noten enthält die Tupel A,B,C. Bei bot2 erfüllen nur die Tupel A, C das Prädikat von Anfrage 2, sodass das Tupel D von Anfrage 2 nicht gelesen wird. Während Anfrage 2 ausgeführt wird, wird Tupel B von Anfrage 4 auf Note = 1.4 geupdated und Anfrage 4 committet vor Anfrage 2. In der Validierungsphase von Anfrage 2 tritt nun ein Konflikt auf, da es zwischen startZeit und commitZeit von Anfrage 2 einen neuen commit im Prädikatenraum gab und Tupel B nun auch gelesen werden müsste. Das aktuelle Ergebnis ist Count: 2, es müsste zur commitZeit aber Count: 3 sein. Somit überschneiden sich der Prädikatenraum von Anfrage 2 und Anfrage 4, da B durch das Update von Anfrage 4 in den Prädikatenraum von Anfrage 2 fällt. Deshalb wird Anfrage 2 abgebrochen.

H4 (T1 und T6): $bot_1, r_1(B), bot_6, r_6(B), w_6(B), w_1(C), commit_1, commit_6$

Kann so auftreten. Bei 2PL wäre diese Historie nicht möglich, da B gesperrt wäre. Anfrage 6 ändert die Note von Tupel B erst nachdem Anfrage 1 das Tupel gelesen hat und committet auch erst nach Anfrage 1.

H6 (T1 und T6): $bot_1, r_1(B), bot_6, r_6(B), w_6(B), commit_6, w_1(C), commit_1$

Kann so nicht auftreten. Die Transaktion 1 würde abbrechen. Der Prädikatenraum der beiden Anfragen überschneiden sich. Anfrage 1 liest das Tupel B mit MatrNr. = 12345, anschließend wird das Tupel von Anfrage 6 geupdated und Anfrage 6 committed die Änderungen. In der Validierungsphase von Anfrage 1 tritt nun ein Konflikt auf, da das Tupel B zwischen der startZeit und commitZeit von Anfrage 6 geändert wurde, da sich die Prädikatenräume überschneiden. Deshalb wird Anfrage 1 abgebrochen.

H7 (T2 und T5): $bot_2, r_2(A), bot_5, w_5(D), commit_5, r_2(D), w_2(E), commit_2$

Kann so nicht auftreten. Die Transaktion 2 kann nur Werte basierend auf ihrem eigenen Startzeitstempel lesen, daher wäre $r_2(D)$ nicht möglich. Zu Beginn der Anfrage 2 existiert D nicht und ist somit nicht für die Anfrage qualifiziert. Durch Anfrage 5 wird der Wert von D nun eingefügt und fällt in den Prädikatenraum. Anschließend ließt Anfrage 2 D laut der Historie, was allerdings nicht möglich ist, da jede TA nur die Werte oder die Version eines Wertes mit zeitStempel<startZeit lesen darf. Somit kann die Historie so nicht auftreten.

H9 (T2 und T5): $bot_2, r_2(A), bot_5, w_5(B), r_2(C), commit_5, w_2(D), commit_2$

Kann so nicht auftreten. Die Schnittmenge zwischen dem Prädikat Note<1.5 von Anfrage 2 und dem abgeleiteten Prädikat Note=1.3 von Anfrage 5 ist nicht leer, daher Konflikt. Während Anfrage 2 aktiv ist, wird ein neues Tupel durch Anfrage eingefügt und die Anfrage 5 committed. In der Validierungsphase von Anfrage 2 kommt es nun zum Konflikt, die Schnittmenge zwischen dem Prädikat Note<1.5 von Anfrage 2 und dem abgeleiteten Prädikat Note=1.3 von Anfrage 5 ist nicht leer und der Prädikatenraum der beiden Anfragen überschneidet sich. Somit wurde zwischen der startZeit und commitZeit von Anfrage 2 ein qualifizierendes Tupel hinzugefügt, da die Anfrage aber nur auf die Datenversion zur eigenen Startzeit zugreifen kann, wird dieses neue Tupel nicht berücksichtigt. Deshalb wird Anfrage 2 abgebrochen.