



Übung zur Vorlesung *Einsatz und Realisierung von Datenbanken im SoSe21*

Maximilian {Bandle, Schüle}, Josef Schmeißer (i3erdb@in.tum.de)

<http://db.in.tum.de/teaching/ss21/impldb/>

Blatt Nr. 09

Hausaufgabe 1

HyPer schafft 120.000 Transaktionen pro Sekunde. Pro Transaktion werden 120 Byte in die Log geschrieben. Berechnen Sie den benötigten Durchsatz zum Schreiben der Log.

Die Datenbank läuft für einen Monat und stürzt dann ab. Es wurde kein Snapshot erstellt. Berechnen Sie die Recoveryzeit. Gehen Sie davon aus, dass die Recovery durch die Festplatte limitiert ist (100 MiB / s). Wieviel Log Einträge werden pro Sekunde reconvert?

Durchsatz = $120.000 * 120 = 14400000 = 13,7 \text{ MiB/s}$.

LogEinträge = $120.000 * 60 * 60 * 24 * 30 = 31104000000$

LogGröße = $\text{LogEinträge} * 120 = 33,95 \text{ TiB}$

RecoveryZeit = 4,12 Tage

RecoveryDurchsatz = 873813 Tx / s .

Hausaufgabe 2

In (pseudo) Java kann eine 'Row-Store-artige' Datenstruktur wie folgt angelegt werden:

```
class Tuple {
    int MatrNr;
    String Name;
    int Semester;
}
Tuple data[] = new Tuple[10000];
```

Notieren Sie, wie die Daten in Form eines Column Stores gehalten werden können in (pseudo) Java.

Erklären Sie Ihrem Tutor, welche Vor- und Nachteile Row- und Column Stores jeweils haben. Was würden Sie für Amazons Webseite verwenden? Was verwenden Sie für die Controlling Datenbank?

```
int MatrNrs[]=new int[10000];
String Names[]=new String[10000];
int Semesters[]=new int[10000];
```

Row Store: Besser, wenn tendenziell viele Attribute des Tupels benutzt werden. Schlecht, wenn nur auf einen Bruchteil des Tupel zugegriffen wird, da viel mehr Daten geladen werden müssen und die Lokalität in der gesamten Speicherhierarchie dann schlechter ist.

Column Store defakto umgekehrt.

Im Schnitt verwendet man heute Row Stores für transaktionale Daten, Column Stores für analytische Daten. Hiervon kann abgewichen werden. Zu bedenken ist, welche Probleme entstehen können, wenn die Anwendungslogik nicht sinnvoll mit dem Datenbanksystem umgeht, beispielsweise weil immer alle Daten (`SELECT * FROM`) und nicht nur die benötigten ausgelesen werden.

Hausaufgabe 3

Gegeben eine Tabelle *Produkte* mit folgendem Schema und 10000 Einträgen:

Id (8 Byte) | Name (32 Byte) | Preis (8 Byte) | Anzahl (8 Byte)

Wieviele Daten werden für folgende Queries in die CPU-Caches geladen? Unterscheiden sie jeweils zwischen Row und Column Store.

1. *select * from Produkte*
2. *select Anzahl from Produkte*

Daten können maximal mit Granularität (64 Byte) in den Cache geladen werden. Das heißt, selbst wenn nur auf einen 64 Bit Integer Wert zugegriffen wird, muss ein kompletter 64-Byte Block geladen werden. Mit diesem Hintergrund ergeben sich folgende Ergebnisse:

1. *select * from Produkte*
 - a) Row: $10000 * 56 = 560000$ Byte
 - b) Column: $10000 * 8 + 10000 * 32 + 10000 * 8 + 10000 * 8 = 560000$ Byte
2. *select Anzahl from Produkte*
 - a) Row: $10000 * 56 = 560000$ Byte
 - b) Column: $10000 * 8 = 80000$ Byte

Hausaufgabe 4

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;*

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.

Hausaufgabe 5

Rekonstruieren Sie die ursprüngliche SQL-Anfrage aus dem folgenden (Pseudo-)Code eines codegenerierenden Datenbanksystems. Welche Art von physikalischem Join wurde benutzt? Handelt es sich um Column- oder Row-Store?

```

struct Student { int matrnr; std::string name; int semester; };
struct Hoeren { int matrnr; int vorlnr; };
struct Result { int vorlnr; int a; };

std::vector<Result> compute(std::vector<Student>&ses, std::vector<Hoeren>&hs){
    std::unordered_multimap<int, Hoeren*> h_map;
    std::unordered_map<int, Student*> s_map;
    for (auto &h : hs)
        h_map.insert(std::make_pair(h.matrnr, &h));
    for (auto &s : ses)
        s_map.insert(std::make_pair(s.matrnr, &s));

    // Group by h.vorlnr; avg(s.semester) = sum(s.semester)/count(*)

    std::unordered_map<int, int> count_map;
    std::unordered_map<int, int> sum_map;

```

```

for (auto &h : hs) {
    count_map.insert(std::make_pair(h.vorlnr, 0));
    sum_map.insert(std::make_pair(h.vorlnr, 0));
}
for (auto &h : h_map) {
    sum_map[h.second->vorlnr] += s_map[h.first]->semester;
    count_map[h.second->vorlnr]++;
}
std::vector<Result> res;
for (auto &r : sum_map)
    res.push_back({ r.first, r.second / count_map[r.first] });
return res;
}

```

Lösung: Anfrage gibt Durchschnittssemester pro Vorlesung aus, Hash-Join (Verbesserung wäre Singleton-Join, da `matrnr` unique), Row-Store.

```

select vorlnr, avg(s.semester) from studenten s, hoeren h
where s.matrnr=h.matrnr group by h.vorlnr

```

Gruppenaufgabe 6

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.

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

Abbildung 1: Hauptspeicher Column-Store

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)

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

1. Welche Bedingungen gelten für die Zeitstempel?

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

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 T_2 seinen Feierabend antreten. Vervollständigen Sie Tabelle 2 und legen Sie einen geeigneten Undo-Puffer (Zeitstempel, Attribut, Undo-Image) an. Wann muss T_b committen, damit T_d bereits die Änderung von T_b liest? Was lesen T_a und T_b ?

T_b 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 T_b .

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

Undo-Puffer:

UID	TID	nextUID	Zeitstempel	Attribut	Undo
$uTb0$	T_b	-	T_b	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. T_a und T_b lesen den Wert, der zum Zeitpunkt T_0 bzw. T_2 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 T_d den geschriebenen Wert nur lesen, wenn er mit T_4 vor T_5 committet worden ist. Sowohl T_4 als auch T_b sind größer als T_0 bzw. T_1 und sie lesen den Wert aus dem Undo-Image. Da kein Schreibvorgang erfolgt ist, zählt bei T_d die Startzeit und es ist irrelevant, dass zwischen T_3 und T_6 eine Änderung im Prädikatbereich erfolgt ist. T_a, T_c zählen drei verfügbare Ärzte, T_d zwei, falls T_b vorher committet hat.

TID	Startzeit	Commitzeit	Aktion
T_a	T_0	T_0	Σ
T_b	T_2	T_4	(<i>Green</i>) --
T_c	T_3	T_3	Σ
T_d	T_5	T_5	Σ

UID	TID	nextUID	Zeitstempel	Attribut	Undo
$uTb0$	T_b	-	T_4	verfügbar	ja

3. Brinkmann und House wollen zeitgleich den Feierabend antreten. House startet bei T_8 , Brinkmann bei T_9 . 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
<i>Ta</i>	<i>T0</i>	<i>T0</i>	Σ
<i>Tb</i>	<i>T2</i>	<i>T4</i>	(<i>Green</i>) --
<i>Tc</i>	<i>T3</i>	<i>T3</i>	Σ
<i>Td</i>	<i>T5</i>	<i>T5</i>	Σ
<i>Te</i>	<i>T8</i>	-	(<i>House</i>) --
<i>Tf</i>	<i>T9</i>	-	(<i>Brinkmann</i>) --

UID	TID	nextUID	Zeitstempel	Attribut	Undo
<i>uTb0</i>	<i>Tb</i>	-	<i>T4</i>	verfügbar	ja
<i>uTe0</i>	<i>Te</i>	-	<i>Te</i>	verfügbar	ja
<i>uTf0</i>	<i>Tf</i>	-	<i>Tf</i>	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 vorgefundenen 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 keine 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	* <i>uTe0</i>
Green	nein	* <i>uTb0</i>
Brinkmann	ja	-

TID	Startzeit	Commitzeit	Aktion
<i>Ta</i>	<i>T0</i>	<i>T0</i>	Σ
<i>Tb</i>	<i>T2</i>	<i>T4</i>	(<i>Green</i>) --
<i>Tc</i>	<i>T3</i>	<i>T3</i>	Σ
<i>Td</i>	<i>T5</i>	<i>T5</i>	Σ
<i>Te</i>	<i>T8</i>	<i>T10</i>	(<i>House</i>) --

UID	TID	nextUID	Zeitstempel	Attribut	Undo
<i>uTb0</i>	<i>Tb</i>	-	<i>T4</i>	verfügbar	ja
<i>uTe0</i>	<i>Te</i>	-	<i>T10</i>	verfügbar	ja

Hausaufgabe 7

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(MatrnNr,Note) values (54321, 3.0)

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

T5: insert into Noten(MatrnNr,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 $r(\dots)$ 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 liest 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 5 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.