



Übung zur Vorlesung *Einsatz und Realisierung von Datenbanken im SoSe24*

Alice Rey, Maximilian Bandle, Michael Jungmair (i3erdb@in.tum.de)

<http://db.in.tum.de/teaching/ss24/impldb/>

Blatt Nr. 09

Hinweise Für die aktive Teilnahme an der Vorlesung am 4. Juli benötigen Sie Spark auf Ihrem Rechner installiert. Eine Anleitung finden Sie unter https://db.in.tum.de/teaching/ss24/impldb/Spark_preparation.pdf. Falls Sie Schwierigkeiten haben, wenden Sie sich bitte in dieser Übungswoche an Ihren Tutor.

Hausaufgabe 1

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): $bot_1, r_1(A), bot_3, w_3(B), w_1(C), commit_1, commit_3$

H2 (T2 und T3): $bot_2, r_2(A), bot_3, w_3(B), r_2(C), w_2(D), commit_2, commit_3$

H8 (T2 und T3): $bot_2, r_2(A), bot_3, w_3(B), r_2(C), commit_3, w_2(D), commit_2$

H3 (T2 und T4): $bot_2, r_2(A), r_2(B), bot_4, r_4(B), w_4(B), r_2(C), w_2(D), commit_2, commit_4$

H5 (T2 und T4): $bot_2, r_2(A), bot_4, r_4(B), w_4(B), r_2(C), commit_4, w_2(D), commit_2$

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

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

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

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

Gruppenaufgabe 2

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 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?
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 ?
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.

Hausaufgabe 3

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?

Hausaufgabe 4

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*

Hausaufgabe 5

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);`

Hausaufgabe 6

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));

    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;  
}
```