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

Übung zur Vorlesung *Einsatz und Realisierung von Datenbanken* im SoSe25

Alice Rey, Maximilian Reif, Tobias Goetz (i3erdb@in.tum.de)
http://db.in.tum.de/teaching/ss25/impldb/

Blatt Nr. 09

Hinweise Die Aufgaben können auf `http://xquery.db.in.tum.de/` getestet werden. Die Daten für das Unischema können mit doc('uni2') geladen werden. Zur Lösung der Aufgaben können Sie die folgenden XQuery-Funktionen verwenden:

max(NUM), count(X), tokenize(STR,SEP), sum(NUM), contains(HAY,NEEDLE)

1. max(NUMBERS) - Returns largest number from list
2. count(LIST) - Return the number of elements in the list
3. tokenize(STR,SEP) - Splits up the string at the seperator
4. sum(NUMBERS) - Returns sum of all numbers in list
5. contains(HAY,NEEDLE) - Checks if the search string (NEEDLE) is contained in the string (HAY)
6. distinct-values(LIST) - Returns the distinct values from the list

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(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 B 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.

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?

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

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 3

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 und zu Node 256 wachsen.
Ein B+ Baum ist schlechter, da bei sequentiellem Einfügen die Knoten nur halb gefüllt sind. Außerdem werden in den Knoten jeweils pro Pointer auch noch ein kompletter 32-Bit Rangeschlüssel gespeichert, was den Speicherbedarf zusätzlich erhöht.

Hausaufgabe 4

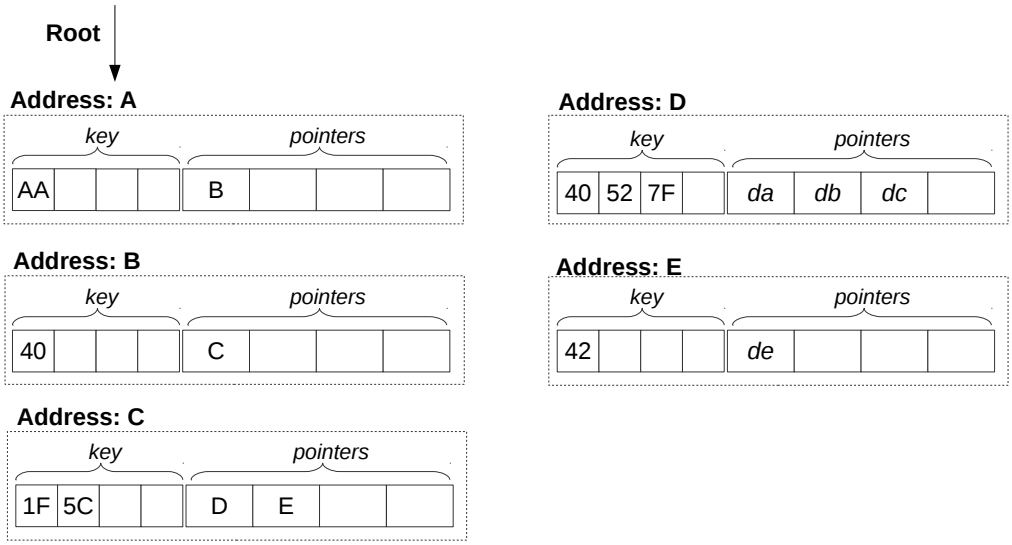

Abbildung 3: Knoten des ART (jeweils Node4)

In Abbildung 3 sehen Sie die Knoten eines ART Baums. Der Wurzelknoten liegt an Adresse `A`. Zeiger die mit `d` anfangen (z.B. da, db, ...) zeigen auf Daten. Suchschlüssel sind in den Aufgaben jeweils sowohl als Zahl z.B. 99, als auch hexadezimal codiert angegeben, z.B. der Wert 99 als 32 Bit Integer (0x00 0x00 0x00 0x63).

1) Beschreiben Sie kurz den Pfad durch den Baum für den 32-bit Suchschlüssel 2856344642 (0xAA 0x40 0x5C 0x42).
   - Suche 1. Byte in Wurzel A. Gefunden, gehe zu Knoten bei B
   - Suche 2. Byte in Knoten B. Gefunden, gehe zu Knoten bei C
   - Suche 3. Byte in Knoten C. Gefunden, gehe zu Knoten bei E
   - Suche 4. Byte in Knoten E. Pointer zu Daten de

   Ergebnis: Schlüssel ist in Daten enthalten
2) Welche dieser Suchschlüssel sind im Baum enthalten? 291 (0x00 0x00 0x01 0x23), 2856329024 (0xAA 0x40 0x1F 0x40), 2856329026 (0xAA 0x40 0x1F 0x42)
   - 291: Nicht enthalten
   - 2856329024: Enthalten
   - 2856329026: Nicht enthalten
3) Beschreiben Sie kurz wie sich der Baum beim Einfügen des Schlüssels 2856352578 (0xAA 0x40 0x7B 0x42) verändert. Der Schlüssel soll auf den Wert an der Adresse `df` zeigen.
   - Die ersten beiden Bytes sind schon im Baum enthalten. Dafür keine Änderung notwendig.
   - Im Knoten C wird im Schlüsselfeld an der dritten Stelle der Wert 0x7B eingetragen und an der dritten Pointer Stelle dann X.
   - Das letzte Schlüsselbyte muss ein neuer Node4 Knoten an der Stelle F erstellt werden. Dieser Knoten enthält das Suchbyte 0x42 und den Pointer df jeweils an der ersten Stelle.

Hausaufgabe 5

Lösen Sie in **reinem XPath** folgende Aufgaben und testen Sie diese auf `xquery.db.in.tum.de`.

1. Lassen Sie sich das gesamte Schema anzeigen.

   ```
   doc('uni')
   ```

2. Finden Sie die Namen aller Fakultäten.

   ```
   doc('uni')//FakName
   ```

3. Finden Sie die Namen aller Studenten, die Vorlesungen hören.

   ```
   doc('uni')//Student[./hoert]/Name
   ```

Hausaufgabe 6

Lösen Sie mit XQuery folgende Anfragen und testen Sie diese auf `xquery.db.in.tum.de`.

1. Geben Sie eine nach Rang sortierte Liste der Professoren aus (C4 oben).

```
<Professoren>
{
    for $p in doc('uni')//ProfessorIn
        order by $p/Rang descending
        return $p
}
</Professoren>
```

2. Finden Sie die Namen der Professoren, die die meisten Assistenten haben.

```
<ProfMitAssistenten>
{
     let $maxAssi := max(
       for $p in doc('uni')//ProfessorIn
          return count($p//Assistent)
       )
     return doc('uni')//ProfessorIn[count(.//Assistent)=$maxAssi]/Name
}
</ProfMitAssistenten>
```

3. Finden Sie für jede von einem Studenten gehörte Prüfung den Namen des Prüfers und den Titel der Vorlesung.

```
<Studenten> {
   let $pr := doc('uni')//Assistent union doc('uni')//ProfessorIn
   for $s in doc('uni')//Student
      return <Student>
      {$s/Name}
      <Pruefungen>
      {
      for $p in $s//Pruefung
         let $prName := $pr[./@PersNr=$p/@Pruefer]/Name
         let $vlTitel := doc('uni')
            //Vorlesung[./@VorlNr=$p/@Vorlesung]/Titel
         return <Pruefung Pruefer="{$prName}">{$vlTitel}</Pruefung>
      }
      </Pruefungen>
      </Student>
} </Studenten>
```

Hausaufgabe (wird nicht in der Übung besprochen)

In traditionellen Datenbanksystemen sind die Festplatte und der Buffermanager oft der Hauptgrund für Performanceengpässe. Wie ändert sich dies in Hauptspeicherdatenbanken, wo sind die neuen Flaschenhälse? Unterscheiden Sie auch zwischen Analytischen und Transaktionalen Workloads.

Der Unterschied zwischen traditionellen Datenbanksystemen und Hauptspeicherdatenbanke ist, dass wir in der Speicherhierarchie ein paar Stufen nach oben gehen. Hauptspeicher ist teurer, aber gleichzeitig auch schneller und hat eine geringere Latenz. Genauso wichtig ist aber auch, dass die Daten nun alle in einem Adressraum liegen. Bei der Nutzung von Festplatten muss das Datenbanksystem explizit die Daten von der Festplatte in den Speicher laden. Beim Hauptspeicher ist der Wechsel zwischen RAM, L3 und L1 Cache transparent für das System. Das heißt ein Buffermanager wird nicht mehr benötigt. Auch wenn der Wechsel zwischen den verschiedenen Hauptspeicherhierarchien für das System nicht explizit sichtbar ist, so ist es doch in der Performance bemerkbar. Der Latenzunterschied zwischen RAM und L3 ist ähnlich groß wie zwischen Festplatten und RAM. Die Datenbank muss nun so strukturiert werden, dass möglichst viele Operationen auf Daten in den schnelleren Speicherschichten ausgeführt werden. Ein weiterer neuerer Flaschenhals sind die Lockingverfahren. Im Vergleich zu einfachen Operationen wie Lesen, Schreiben, Addition, etc. ist ein Lock zu erstellen viel teurer. Viele Hauptspeichersystem versuchen daher Locks zu vermeiden. Ein Problem, dass hauptsächlich nur Transaktionale Workloads betrifft ist, dass beim Ändern von Daten die Änderung immer noch persistiert werden muss (Logging). Es reicht nicht aus, die Daten nur im Hauptspeicher zu halten, da diese dann nach einem Systemabsturz verloren wären. Das Schreiben auf Festplatte ist selbst mit SSDs noch wesentlich teurer als Änderungen im Hauptspeicher vorzunehmen. Auch ein Problem in Transaktionale Workloads ist die Annahme der Anfragen. Transaktionale Anfragen sind typischerweiße sehr schnell. Ein einzelner Rechner kann sehr einfach mehrere Hunderttausend Anfragen verarbeiten. Hier ist die Netzwerklatenz auch wesentlich größer als die Verarbeitungszeit der Anfragen. Daher kann ein einzelner Client die Datenbank garnicht auslasten wenn er jede Anfrage einzeln abschickt.

Hausaufgabe (wird nicht in der Übung besprochen)

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 recovert?

Durchsatz = 120.000 * 120 = 14400000 = 13,7 MiB/s.

LogEinträge =120.000*60*60*24*30=311040000000

LogGröße = LogEinträge*120 = 33,95 TiB

RecoveryZeit = 33,95 TiB / (100 MiB/s) = 4,12 Tage

RecoveryDurchsatz = 873813 Tx / s.