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

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. 10

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

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

```
struct Tuple {
  int MatrNr;
  RuntimeString Name;
  int Semester;
}
Tuple data[10000] = {};
```

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

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[10000] = {};
RuntimeString Names[10000] = {};
int Semesters[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 2

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 3

In Abbildung 1 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).

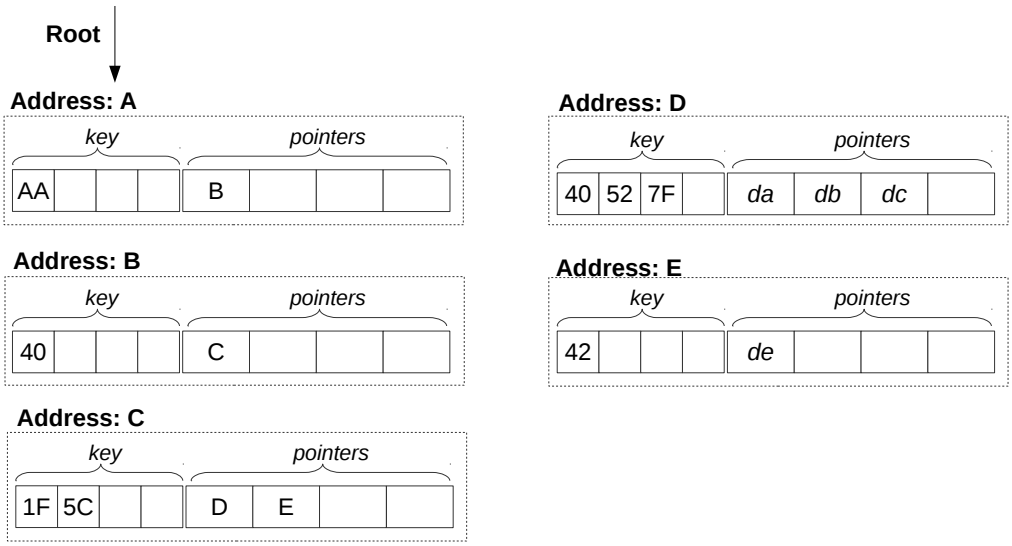

Abbildung 1: Knoten des ART (jeweils Node4)

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.

Gruppenaufgabe 4

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 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>
```