Datenbanken

Indices und Transaktionen

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at

FB Computerwissenschaften
Universität Salzburg

Wintersemester 2013/14

Literatur und Quellen

Lektüre zu den Themen "Indices" und "Transaktionen":

- Kapitel 7.6-7.11, 7.15-7.16 (Indices) und Kapitel 9 (Transaktionen) aus Kemper und Eickler: Datenbanksysteme: Eine Einführung. 8. Auflage, Oldenbourg Verlag, 2011.
  7. Auflage: `http://www.oldenbourg-link.com/isbn/9783486592771`

Literaturquellen

- Elmasri and Navathe: Fundamentals of Database Systems. Fourth Edition, Pearson Addison Wesley, 2004.
- Silberschatz, Korth, and Sudarashan: Database System Concepts, McGraw Hill, 2006.

Danksagung Einige Folien nach einer Vorlage von:

- Sven Helmer, Freie Universität Bozen, Italien

Inhalt

1 Indices

2 Transaktionen

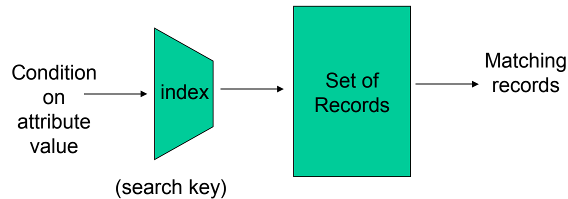
Inhalt

1 Indices

2 Transaktionen

Was ist ein Index?

- Index: Datenstruktur die einen effizienteren Datenzugriff erlaubt.



- Suchschlüsse (search key), kurz "Schlüssel":
  - einzelnes oder mehrere Attribute
  - nach den Suchschlüssel-Attributen werden die Tupel gesucht
  - Suchschlüssel müssen *nicht* eindeutig sein
- Indices sind wesentlich für die effiziente Ausführung von SQL.
- Falsche Index-Wahl kann führen zu:
  - Indices die ständig aktualisiert aber nie gebraucht werden
  - vollständiges Lesen einer Tabelle um ein einziges Tupel zu finden
  - Join-Anfragen die Stunden oder Tage dauern

Performance in Datenbanken

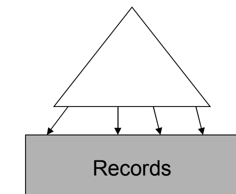
- Plattenzugriffe (5-10ms) sind bis zu 100.000 mal langsamer als RAM-Zugriffe (100ns)
- Vergleich: Sie heben ein Blatt Papier vom Schreibtisch auf (1s) gegenüber dasselbe Blatt Papier in Paris zu holen (24h).
- Geschwindigkeit in Datenbanken wird deshalb in Anzahl von Plattenzugriffen übersetzt.
- Sequentielles Lesen von der Platte ist bis zu 10 mal schneller als lesen einzelner Seiten an verschiedenen Stellen.
- Indexes versuchen Anzahl der Plattenzugriffe zu reduzieren.

Charakteristiken von Indices

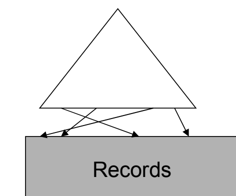
- Indices können oft als Bäume betrachtet werden (B^+-Baum, Hash Index)
  - einige Knoten sind im Hauptspeicher (z.B. Wurzelknoten)
  - je weiter unten im Baum ein Knoten ist, desto unwahrscheinlicher ist er im Hauptspeicher
- Tiefe/Ebenen des Baums: Anzahl der Knoten auf dem Weg von der Wurzel zu einem Blattknoten
  - ein Knoten ist typischerweise eine Seite auf der Platte
  - jede Ebene erfordert das Lesen einer Seite von der Platte
  - das kostet mehrere Millisekunden
- Fanout: Anzahl der Kinder eines Knotens
  - großer Fanout führt zu flachen Bäumen
  - flache Bäume brauchen weniger Ebenen und sind schneller

Primär- und Sekundärindices

- Primärindex auf Attribut X (*primary index, clustered index*)
  - Tupel werden physisch nach X gruppiert
  - B^+-Baum: Tupel werden nach X sortiert
  - nur *ein* Primärindex pro Tabelle
  - X ist typisch aber nicht notwendigerweise der Primärschlüssel



- Sekundärindex auf Attribut X (*secondary index*)
  - stellt keine Bedingung an physische Ordnung der Tupel in der Tabelle
  - mehrere Indices pro Tabelle möglich



- Index über mehrere Attribute:
  - X kann auch eine Sequenz von Attributen sein
  - Reihenfolge der Attribute spielt eine Rolle!

Primärindex

- Gut für Mehr-Punkt Anfragen:
  - Gleichheitsanfrage auf nicht-eindeutigem Attribut
  - alle Ergebnistupel liegen physisch nebeneinander (=effizient)
  - Beispiel: einen Nachnamen im Telefonbuch suchen
- Gut für Bereichs-, Prefix-, Sortier- Anfragen:
  - funktioniert wenn Primärindex B^+-Baum ist
  - Prefix Beispiel: Suche nach allen Namen im Telefonbuch die mit 'St' beginnen
  - alle Ergebnistupel liegen physisch nebeneinander
- Gut für Gleichheits-Join (Prädikat mit "="):
  - auch für Joins auf nicht-eindeutige Attribute effizient
  - Index in nur einer Tabelle: Indexed Nested-Loop
  - Index in beiden Tabellen: Merge-Join

Covering Index

- Covering Index:
  - beantwortet eine Leseanfrage direkt auf dem Index
  - schnell, da Tabelle nicht gelesen werden muss
- Beispiel 1: Index auf Nachname:

```
SELECT COUNT(Nachname) WHERE Nachname='Smith'
```

- Beispiel 2: Index auf A, B, C (in dieser Reihenfolge)
  - Covering:

```
SELECT B, C
FROM R
WHERE A = 5
```

  - Covering, aber kein Präfix:

```
SELECT A, C
FROM R
WHERE B = 5
```

  - Kein Covering: D muss von Tabelle geholt werden

```
SELECT B, D
FROM R
WHERE A = 5
```

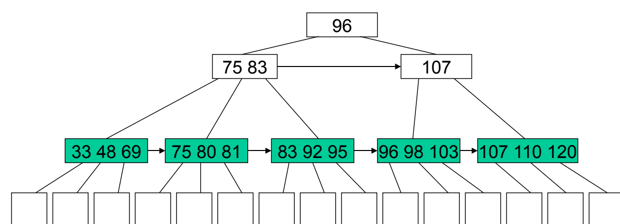

Sekundärindex

- Für Punkt-Anfragen (0 oder 1 Ergebnis-Tupel) immer gut.
- Besonders gut für Anfragen, die nur im Index beantwortet werden können (Covering Index).
- Mehr-Punkt Anfragen: nur gut für kleine Ergebnismenge
  - $\#T$: Anzahl der Ergebnistupel
  - $\#P$: Anzahl der Platten-Seiten auf denen Tabelle gespeichert
  - die $\#T$ Tupel sind gleichverteilt über alle Platten-Seiten
  - eine Anfrage wird $\min(\#T, \#P)$ Platten-Seiten lesen
- Sekundärindex verlangsamt u.U. Anfragen mit großer Ergebnismenge:
  - Sequenzielles Lesen ist pro Seite 10 mal schneller als Index-Zugriff
  - deshalb sollte $\#T$ deutlich kleiner als $\#P$ sein

Index Datenstrukturen

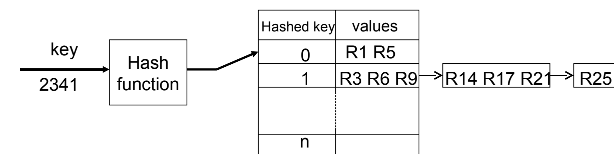
- Indices können mit verschiedenen Datenstrukturen implementiert werden.
- Wir besprechen:
  - B^+-Baum Index
  - Hash Index
- Weiter Index Typen:
  - Bitmap Index: in Data Warehouses
  - Dynamic Hash Index: Anzahl der Buckets wird dynamisch angepasst
  - R-tree: Index für 2D Objekte (Punkte, Linien, Formen)
  - Quadtree: teilt Ebene rekursiv in vier Quadranten
  - Octree: 3D Version von Quadtrees
  - Hauptspeicher Indices: T-Tree, 2-3 Tree, Binärbaum

B^+-Tree



- balancierter Baum von Schlüssel-Pointer Paaren
- Schlüssel nach Wert sortiert
- jeder Knoten ist mindestens halb voll
- Daten sind in Blättern gespeichert
- Zugriff: durchlaufe Baum von Wurzel bis Blatt

Hash Index

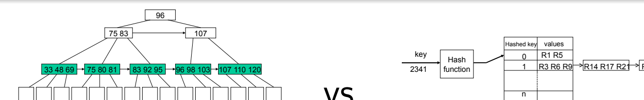


- Hash Funktion:
  - bildet Schlüssel auf Ganzahlen im Bereich $[0..n]$ ab (Hash Werte)
  - Pseudo-Randomisierung: die meisten Werte sind gleichverteilt in $[0..n]$
  - ähnliche Schlüssel haben normalerweise sehr verschiedene Hash-Werte!
  - Datenbank wählt geeignete Hash-Funktion aus
- Hash Index:
  - Hash Funktion fungiert als "Wurzel Knoten" des Index Baumes
  - Hash-Wert ist die Nummer eines Behälters (bucket)
  - im Behälter werden die Tupel zum Hash-Wert gespeichert

Überlauf-Seiten

- Hash Index ohne Überlauf: nur 1 Platten-Seite muss gelesen werden
- Wenn Behälter voll ist: Überlauf-Seiten
  - Pointer zu Überlauf-Seite wird im vollen Behälter abgespeichert
  - jede Überlauf-Seite erfordert zusätzlichen Plattenzugriff
  - in Behältern sollte deshalb genug freier Platz gelassen werden (Richtwert: 50%)
- Hash Index mit vielen Überlaufseiten: neu organisieren
  - spezielle Funktionen nutzen
  - oder Index einfach löschen und neu aufbauen

B^+-Baum vs. Hash Index



- Hash Index: nur (Mehr-)Punkt-Anfragen und Grouping
- Hash Index nicht nutzbar für:
  - Bereichs-, Prefix-, MIN/MAX-, Sortier-Anfragen
  - ähnliche Schlüssel haben verschiedene Hash-Werte
  - deshalb landen ähnliche Schlüssel in verschiedenen Buckets
- B^+-Baum: alle Anfrage-Typen
- Punkt-Anfragen:
  - Hash Index braucht nur 1 Seite von der Platte zu lesen
  - B^+-Baum muss zuerst Suchbaum durchlaufen
- In der Praxis werden B^+-Bäume häufig bevorzugt:
  - Hash Index nur selten schneller als B^+-Baum
  - B^+-Baum ist aber vielfältiger einsetzbar
  - einige Systeme bieten deshalb überhaupt keine Hash-Indices an

Indices in SQL

- SQL Standards definieren keine Indices.
  - Es hat sich eine gebräuchliche Syntax durchgesetzt.
  - Datenbanksysteme bieten spezifische Erweiterungen an (z.B. Auswahl von B^+-Baum oder Hash Index).
- Index erzeugen:

  create [unique] index *Indexname* **on** *Tabellenname*(Attributnamen)
- Index löschen:

  drop index *Indexname*
- Beispiel:

  create index *Name_idx* **on** *Kunden*(Nachname,Vorname)
- Indices werden in manchen Systemen automatisch angelegt, wenn Schlüssel oder Fremdschlüssel definiert werden.

Zusammenfassung

- Primär vs. Sekundär-Index:
  - nur ein Primärindex pro Tablelle
  - beliebig viele Secondär-Indices
- Sekundär-Index: Trade-off zwischen effizienten Anfragen und Updates
- Datenstrukturen: B^+-Tree ist flexibler und selten langsamer als Hash Index

Inhalt

1 Indices

2 Transaktionen

Was ist eine Transaktione?

- Eine Transaktion ist eine Programmeinheit, die auf Daten zugreift und diese möglicherweise verändert.
- Beispiel: überweise \$50 von Konto A nach Konto B
  1. $R(A)$
  2. $A \leftarrow A - 50$
  3. $W(A)$
  4. $R(B)$
  5. $B \leftarrow B + 50$
  6. $W(B)$
- Transaktionsverwaltung beschäftigt sich mit zwei Hauptproblemen:
  1. gleichzeitige Ausführung mehrere Transaktionen
  2. Fehler verschiedener Art (z.B. Hardware Fehler, Systemabsturz)

ACID Eigenschaften

- Datenbanksysteme müssen ACID für Transaktionen garantieren:
  - Atomicity: entweder alle Operationen einer Transaktion werden ausgeführt oder gar keine
  - Consistency: die Ausführung einer isolierten Transaktion erhält die Datenbank in konsistentem Zustand
  - Isolation: obwohl mehrere Transaktionen gleichzeitig ausgeführt werden ist es für jede einzelne Transaktion so, als wäre sie alleine
  - Durability: Nach erfolgreicher Beendigung einer Transaktion müssen deren Veränderungen in der Datenbank dauerhaft erhalten bleiben, auch bei Systemabsturz oder anderen Fehlern.

Atomicity

- Beispiel: überweise \$50 von Konto A nach Konto B
  1. $R(A)$
  2. $A \leftarrow A - 50$
  3. $W(A)$
  4. $R(B)$
  5. $B \leftarrow B + 50$
  6. $W(B)$
- Was, wenn Fehler (Hardware od. Software) nach Schritt 3 auftritt?
  - Geld geht verloren
  - Datenbank ist inkonsistent
- Atomicity:
  - entweder alle Operationen oder gar keine
  - Änderungen von teilweise ausgeführten Transaktionen werden nicht in die Datenbank geschrieben

Consistency

- Beispiel: überweise \$50 von Konto A nach Konto B
  1. $R(A)$
  2. $A \leftarrow A - 50$
  3. $W(A)$
  4. $R(B)$
  5. $B \leftarrow B + 50$
  6. $W(B)$
- Konsistenzbedingung Beispiel: Summe $A + B$ muss unverändert bleiben
- Konsistenzbedingungen allgemein:
  - explizite Integritätsbedingungen (z.B. Fremdschlüsselbedingung)
  - implizite Integritätsbedingungen (z.B. Summe aller Kontostände einer Bank muss gleich der Summe der Kontostände aller Filialen sein)
- Transaktion:
  - muss eine konsistente Datenbank vorfinden
  - währen der Transaktion sind inkonsistente Zustände erlaubt
  - nach Ende der Transaktion muss Datenbank wieder konsistent sein

Isolation – Einführendes Beispiel

- Beispiel: überweise \$50 von Konto A nach Konto B
  1. $R(A)$
  2. $A \leftarrow A - 50$
  3. $W(A)$
  4. $R(B)$
  5. $B \leftarrow B + 50$
  6. $W(B)$
- Angenommen, es gibt eine zweite Transaktion T_2:
  - $T_2 : R(A), R(B), print(A + B)$
  - T_2 wird zwischen den Schritten 3 und 4 ausgeführt
  - T_2 sieht einen inkonsistenten Datenbankzustand und liefert das falsche Ergebnis

Isolation

- Triviale Isolation: alle Transaktionen laufen seriell (nacheinander)
- Isolation für nebenläufige (concurrent) Transaktionen: Für jedes Paar T_i und T_j von Transaktionen scheint es für T_i als wäre T_j bereits beendet bevor T_i gestartet ist oder hätte noch nicht begonnen, wenn T_i endet.
- Schedule: (Historie)
  - gibt die chronologische Ordnung einer Sequenz von Befehlen aus verschiedenen Transaktionen an
  - Äquivalente Schedules resultieren immer in identischen Datenbankinstanzen wenn sie mit identischen Instanzen starten
- Serialisierbarer Schedule:
  - äquivalent einem seriellen Schedule
  - serialisierbarer Schedule von $T1$ und $T2$ ist entweder zu $T1, T2$ oder $T2, T1$ äquivalent

Durability

- Wenn eine Transaktion endet, macht sie ein commit.
- Beispiel: Transaktion macht commit zu früh:
  - Transaktion schreibt A und macht ein commit
  - A wird in den Festplattenbuffer geschrieben
  - dann stürzt das System ab
  - der Wert von A geht verloren
- Durability: Nachdem eine Transaktion mit commit abgeschlossen hat, bleiben deren Änderungen auch im Falle eines Systemfehlers erhalten.
- Commit darf erst abgeschlossen werden, wenn alle Änderungen persistent gespeichert sind:
  - entweder in eine Log Datei oder direkt in die Datenbank
  - Datenbank muss im Falle eines Absturzes wiederhergestellt werden (recovery)

Unverwünschte Phänomene nebenläufiger Transaktionen

- Dirty read
  - Transaktion liest Daten, die von nebenläufiger uncommitted Transaktion geschrieben wurden
  - Problem: die Leseoperation gibt einen Wert der nie in der Datenbank war, da die schreibende Transaktion abgebrochen wurde
- Non-repeatable read
  - aufeinanderfolgende Leseoperationen auf denselben Dateneintrag ergeben verschiedene Werte innerhalb einer Transaktion (aufgrund von Änderung durch andere Transaktionen)
  - z.B. nebenläufige Transaktionen T_1: $x = R(A)$, $y = R(A)$, $z = y - x$ und T_2: $W(A = 2 * A)$, dann kann z entweder 0 oder den Anfangswert von A haben (sollte 0 sein!)
- Phantom read
  - dieselbe Anfrage innerhalb einer Transaktion gibt verschiedene Tupel zurück, wenn sie mehrmals ausgeführt wird
  - z.B. "Q: SELECT * FROM Konten WHERE $Guthaben > 1000$" ergibt 2 Tupel beim ersten Aufruf, dann wird ein neues Konto mit $Guthaben > 1000$ durch eine andere Transaktion eingefügt; beim zweite Aufruf gibt Q drei Tupel zurück.

Isolation Levels (SQL Standard)

- Read uncommitted: dirty, non-repeatable, phantom
  - Schreiboperationen überschreiben keine "uncommitted" Daten
  - Leseoperationen können Daten lesen, die nicht "committed" sind
- Read committed: non-repeatable, phantom
  - Leseoperationen können nur "committed" Daten lesen
  - cursor stability: innerhalb einer `SELECT` Anfrage sind Leseoperationen "repeatable"
- Repeatable read: phantom
  - phantom reads sind möglich
- Serializable:
  - keine der Phänomene sind möglich

Isolation Levels (SQL Standard)

- "Serializable" in SQL ist nicht identisch mit Serialisierbarkeit nach unserer Definition.
  - Oracle und ältere PostgreSQL Versionen erlauben nicht-serialisierbare Schedules, die jedoch der Definition von SQL Serializable genügen.
  - Jeder serialierbare Schedule ist jedoch "Serializable" nach SQL.
- Viele Systeme implementieren nur zwei Levels:
  - Read committed (meist sogar "cursor stability")
  - Serializable
- Es muss ein Kompromiss eingegangen werden:
  - read committed ist schneller, aber Ergebnisse könnten falsch sein
  - serializable garantiert exakte Ergebnisse, ist aber langsamer

Wann sollte schwächerer Level genommen werden?

- Anfrage braucht keine exakten Antworten (z.B. statistische Anfragen)
  - Beispiel: Zähle alle Konten mit Guthaben> $1000.
  - read committed ist genug
- Transaktionen mit menschlicher Interaktion
  - Beispiel: Flug Reservierung
  - Kosten für Serialisierbarkeit zu hoch, da Transaktionen zu lange dauern

Beispiel: Flug Reservierung

- Reservierung umfasst drei Schritte:
  1. rufe verfügbare Sitzplätze ab
  2. Kunde entscheidet sich für Sitzplatz
  3. reserviere Sitzplatz
- Einzelne Transaktion:
  - Sitzplätze sind gesperrt (können weder gelesen noch geschrieben werden) während Kunde sich entscheidet
  - alle anderen Benutzer müssen warten
- Zwei Transaktionen: (1) Liste holen, (2) Reservieren
  - Sitz ist möglicherweise inzwischen schon weg, wenn Kunde versucht zu reservieren
  - ist leichter zu tolerieren als das System zu blockieren

Transaktionen in SQL/1

- Parameter für Transaktion in SQL setzten:

  set transaction *level, access mode*
- level ist einer der Isolation Level:
  - **read uncommitted**
  - **read committed**
  - **repeatable read**
  - **serializable**
- access mode kann sein:
  - **read only**: nur Leseoperation in Transaktion
  - **read write**
- Read only vs. read write:
  - read only Performance deutlich erhöhen, das es zwischen read-only Transaktionen keine Konflikte gibt
  - sobald eine Transaktion schreibt, kann es (auch mit read-only Transaktionen) zu Konflikten kommen

Transaktionen in SQL/2

- Eine Transaktion beginnen:

  start transaction

- Eine Transaktion abbrechen:

  rollback [work]

  - Beispiel: Überweisung muss abgebrochen werden da zu wenig Geld auf Konto.
- Eine (erfolgreiche) Transaktion beenden:

  commit [work]

  - **commit** Befehl kann von Datenbank auch mit *rollback* beantwortet werden, falls ein *commit* nicht möglich ist
  - Beispiel: Konflikt mit anderen Transaktionen, die vorher *commit* gesandt haben
- Auch die Datenbank kann Transaktionen abbrechen, z.B., wenn sich zwei Transaktionen gegenseitig blockieren (Deadlock)

Zusammenfassung

- Transaktion: Programmeinheit, die als Ganzes ausgeführt werden soll.
- ACID soll für Transaktionen garantiert werden:
  - Atomicity: alles oder nichts
  - Consistency: konsitente Zustände sehen und hinterlassen
  - Isolation: nebenläufige Transaktionen stören sich nicht
  - Durability: auch im Fehlerfall Konsistenz und keine verlorenen Transaktionen
- SQL bietet Transaktionen mit unterschiedlichen Garantien.
- Trade-Off zwischen Korrektheitsgarantie und Effizienz