Indexstrukturen

1. Motivation und Überblick
2. multidimensionale Indexe
3. Bulkloading von Indexen
4. (Hauptspeicherindexe)
5. (Indexe für historische Daten)
6. (Joinindexe, materialisierte Sichten)

Überblick über Zugriffspfade

Arten von Indexen

- sequentielle Speicherstrukturen
  1. Listen (physisch geclustert)
  2. Ketten (logisch verkettet)
- Baumstrukturen (Schlüsselvergleich)
  1. B-Baum, R-Baum, kdB-Baum, hB-Baum, ...
- gestreute Strukturen (Schlüsseltransformation)
  1. statisches Hashing
  2. dynamisches (erweiterbares) Hashing

Anfragearten

- exakt-match query
- partial-match query
- range query
- nearest neighbor search

Ziel eines Index

- unterstütze möglichst viele Anfragetypen
- erlaube schnellen Zugriff
- niedrige Kosten beim Update
- kleiner Speicherbedarf
- muß mit Schiefe in der Datenverteilung zurechtkommen

Bemerkungen

- alle Anfragen können auch ohne Index durch einen kompletten *scan* der Daten bearbeitet werden
- selbst wenn Index vorhanden, ist der *scan* manchmal die bessere Alternative; Entscheidung muß der Anfrageoptimierer für jede Anfrage einzeln treffen

B und B*-Baum – Wiederholung

Eigenschaften

- balanciert und mehrwegig
- kontrollierbarer Mindestfüllgrad von allen Knoten außer der Wurzel
- erlaubt Komprimierung von Schlüsseln und Wegweisern
- logarithmischer Zugriff und Update
- unterstützt alle Anfragetypen
- erlaubt sogar *sortiertes* Lesen der indizierten Sätze
- eindimensional
- super für die Festplatte; nicht gut für den Hauptspeicher
- realisiert in jedem vernünftigen DBMS

Unterschiede: B und B*-Baum

- B-Baum speichert Sätze in allen Knoten; B*-Baum speichert TID (Primärindex) oder TID-Listen (Sekundärindex) in Blättern
  (Generell ist Speicherung von Satz vs. TID ein wichtiger Freiheitsgrad, der bei allen Indexstrukturen besteht!)
- B*-Baum verkettet benachbarte Blätter für effizienten sortierten *scan*.
- B*-Baum hat höheren Verzweigungsgrad
- B-Baum nur als Primärindex sinnvoll, mehrere B-Bäume auf demselben Datenbestand führen zu Redundanz
- Abarbeitung von TID-Listen beim B*-Baum kann zu random I/O auf der Festplatte führen
- **N.B.** TID-Listen können auch als (komprimierte) Bitmaps realisiert werden. Unabhängig davon können TID-Listen auch separat gehalten werden; in diesem Fall enthalten die Blätter des B* Baumes Zeiger auf TID-Listen.

Erweiterbares Hashing – Wiederholung

Eigenschaften

- Speichern der Sätze (oder TIDs von Sätze) in Buckets
- Directory realisiert Abbildung von Schlüssel zu Bucket
- Zugriff auf Directory mittels (erweiterbarer) Hashfunktion
- max. zwei Seitenzugriffe bei exakt-match queries (Directory- und Bucketzugriff)
- Wieviel Seitenzugriffe beim Einfügen?
- nicht anwendbar bei range queries oder nearest neighbor search
- eindimensional
- in der Praxis ist es sehr wichtig beim Erzeugen bereits zu wissen, wie groß die Hashtabelle voraussichtlich wird, damit das Einfügen nicht zu einer Katastrophe wird
- es gibt viele, viele knifflige Varianten
- kaum ein RDBMS setzt Hashing ein (OODBMSe schon eher)

Mehrdimensionale Indexe

Art von Objekten

1. Punkte: Satz $t = (a_1, a_2, \ldots, a_k)$
2. räumlich ausgehnte Objekte:
   - werden angenähert durch kleinst-umfassende Rechtecke ($k = 2$) oder Schachteln (allgemein)
   - Schachtel (2^k Punkte)

     $(x_1^1, x_1^2, \ldots, x_1^k, x_2^1, x_2^2, \ldots, x_2^k, \ldots, x_{2^k}^1, x_{2^k}^2, \ldots, x_{2^k}^k)$
   - Annäherung führt zu *false drops*. Index dient als Vorfilter.

Arten von Punktanfragen

- exact-match query

$$(A_1 = a_1) \wedge (A_2 = a_2) \wedge \ldots (A_k = a_k)$$

- partial-match query

$$(A_1 = a_{i_1}) \wedge (A_2 = ai_2) \wedge \ldots (A_s = a_{i_s})$$

  mit $1 \leq i_1 < i_2 < \ldots < i_s \leq k$

- exact range query

  $(L_1 \leq a_1 \leq H_1) \wedge (L_2 \leq a_2 \leq H_2) \wedge \ldots \wedge (L_k \leq a_k \leq H_k)$

- partial range query

  $(L_1 \leq a_{i_1} \leq H_1) \wedge (L_2 \leq a_{i_2} \leq H_2) \wedge \ldots (L_s \leq a_{i_s} \leq H_s)$

  mit $1 \leq i_1 < i_2 < \ldots < i_s \leq k$

- nearest neighbor search zu einem Punkt P mit Distanzfunktion d (z.B. d ist euklidischer Abstand); suche $\min(d(t, P))$.

Beispiele von Anfragen auf Schachteln

- Punktanfrage. Finde alle Schachteln, die einen gegebenen Punkt enthalten.
- Gebietsanfrage. Finde alle Schachteln, die mit einem gegebenen Gebiet überlappen.

Mehrdimensionale Indexe

1. Anwendung eindimensionaler Indexe
2. Einfache Ad-hoc Ansätze
   - Quadrantenbaum
   - k-d Baum
   - mehrschlüssel Hashing
3. Divide and Conquer Ansätze: Organisiere den Datenraum so, daß "nahe" Objekte möglichst auch im selben Bucket landen
   - eindimensionale Einbettung durch *space-filling curves*; UB-Baum
   - heterogener k-d Baum
   - k-d-B Baum
   - hB Baum
4. Dimensionsverfeinerung
   - Grid File und Varianten
   - mehrdimensionales dynamisches Hashing
5. Bäume für räumlich ausgehnte Objekte
   - R, R^+, R^* Bäume
   - X Baum
   - Pyramid Baum
6. GiST: "Suche Mutter aller Indexe"

Einsatz Eindimensionaler Indexe

1. Separate Indexe für jedes Attribut

- lege k eindimensionale Indexe an; für jede Dimension einen
- partial/exact-match query: probe auf jeden der s oder k Indexe und bilde Schnittmenge aller gefundenen TIDs
- partial/exact-range query: analog, falls B* Bäume (und kein Hashing) verwendet wurde
- nearest neighbor search: ein wenig tricky, aber möglich

Bewertung

- Aufwand beim Zugriff wächst linear mit Anzahl der Prädikate in der Anfrage; Schnittmengenbildung ist evtl. teuer – so teuer wie ein Join und erfordert Optimierung
- meist nur gut beim Zugriff, wenn Prädikate sehr selektiv sind
- Aufwand beim Update wächst linear mit k

2. Konkatieniere die k Attribute zu einem Schlüssel

- mehrdimensionaler Schlüssel ("10", "20") wird zum eindimensionalen Schlüssel "1020".
- exact-match query: trivial
- partial-match query: einfach, falls nach Präfix des konkatiernten Schlüssel gesucht wird; ansonsten wird partial-match nicht unterstützt oder es müssen mehrere Indexe angelegt werden
- range queries und nearest neighbor search: schwierig oder gar nicht unterstützt

Bewertung

- nur sehr eingeschränkt einsetzbar
- falls einsetzbar, sehr schneller Zugriff möglich
- Aufwand für Update so gering wie im eindimensionalen Fall (z.B. $\log n$ beim B*-Baum

Grid File

Grundidee

- Sätze (= Punkte) werden in Buckets abgelegt
- Wenn ein Bucket überläuft, wird dieser Bucket und alle anderen Buckets derselben Hyperebene gesplittet
- die Sätze von zwei benachbarten (logischen) Buckets können in einem physischen Bucket gespeichert werden
- Griddirectory übernimmt Abbildung von Punkten zu Bucketadressen
- $S_1, \ldots S_k$ eindimensionale Vektoren dienen als Einstiegspunkte in das Griddirectory; materialisieren die Splitgeschichte
- ähnelt in manchen Aspekten der Idee vom erweiterbaren Hashing

Grid File

Anfragebearbeitung

- exact-match query: finde Index im Griddirectory des relevanten Buckets mithilfe der S_i Vektoren
- partial-match query: finde Indizes im Griddirectory der relevanten Buckets mithilfe der S_i Vektoren; man muß praktisch eine Hyberebene von Bucketadressen aus dem Griddirectory lesen
- range queries und nearest neighbor search funktioniert analog

Einfügen und Split

- ein Split in Dimension d erweitert den relevanten S_d Vektor um einen Eintrag
- ein Split verursacht die Erweiterung des Gridfiles entlang einer kompletten $k-1$ großen Hyperebene
- die S_i bleiben verhältnismäßig klein, das Griddirectory wird sehr groß

Grid File

Erweiterungen

- das Griddirectory, welches ja eine große k dimensionale Datei ist, selber als Gridfile zu organisieren
- interpolationsbasierte Gridfiles; helfen bei schiefer Datenverteilung

Bewertung

- exact-match query kann idealerweise mit 2 Seitenzugriffen erledigt werden, wenn die S_i in den Hauptspeicher passen
- Update/Split ist teuer

R-Baum

Grundidee

- Organisation (Einfügen, Split und Balancierung) wie in einem B* Baum
- d.h. auch speichere Sätze (=Schachteln) oder TIDs von Sätzen nur in Blättern und Wegweiser Informationen in internen Knoten
- Wegweiser ist eine Schachtel, die alle Schachteln des Teilbaumes umschließt
- **beachte:** Wegweiser innerhalb eines internen Knotens können überlappen genauso wie die Sätze überlappen können

Anfragebearbeitung

- alle Anfragetypen möglich; man muß aber eventuell bei unglücklicher Überlappung der Wegweiser in mehrere Subbäume absteigen

Herausforderung

- Grad der Überlappung zwischen den Wegweisern minimieren
- klassische Einfügeheuristik: füge einen Satz in den Teilbaum ein, in dem er zu der geringsten Vergrößerung der Wegweiser führt
- klassische Splitheuristik: die Summe der Volumina der neu entstehenden Wegweiser sollte minimiert werden
- R* Baum: cleveres Splitkriterium, Kombination aus Summe des Umfangs, Summe der Volumina und Volumen der Überlappung der Wegweiser

Bewertung

- Suche kann zu Traversierung des gesamten Baumes entarten
- Einfügen hat logarithmischen Aufwand

Split im R Baum

Ziel: Minimiere Gesamtvolumen

Exhaustive Search

- betrachte alle möglichen Splits und wähle den besten aus
- exponentielle Laufzeit: $\mathcal{O}(2^M)$
- $M = 50$ bei $k = 2$ und 1024 Bytes Seiten

Heuristik

- wähle zwei "Extremsätze" aus; die Extremsätze bilden die Basis für die beiden neuen Knoten
- ordne alle weiteren Sätze Satz für Satz einem der beiden Knoten zu, so daß Flächenzuwachs minimiert wird
- quadratische Laufzeit: $\mathcal{O}(M^2)$

Literatur: Guttman, SIGMOD 1984

GiST

Ziel

- viele Indexe beruhen auf balancierten (B*) Bäumen
- stelle generischen balancierten Baum zur Verfügung, der dann zu einer bestimmten Indexstruktur instanziiert werden kann
- erleichtert Konstruktion neuer Indexstrukturen
- erleichtert fairen Vergleich bestehender Indexstrukturen, wenn Sie als Instanz eines GiST implementiert sind

Grundidee

- Abstrahiere von den Konzepten des Schlüsselvergleichs, der Datentypen und der Split- und Einfügeheuristiken
- zum besseren Verständnis kann man sich immer im Hinterkopf den R-Baum vorstellen

Literatur

- Hellerstein et al., VLDB 1995

Definition GiST vom Typ(f, M, h)

- jeder Weg von der Wurzel zum Blatt hat die Länge h
- jeder Knoten enthält zwischen $f \times M$ und M Einträge, sofern er nicht die Wurzel ist
- die Wurzel hat mindestens zwei Söhne, sofern sie kein Blatt ist
- jeder Indexeintrag (p, ptr) in einem Blatt liefert p=true, wenn er mit den Werten aus dem referenzierten Satz (ptr=D) instanziiert wird
- jeder Indexeintrag (p, ptr) in einem inneren Knoten liefert p=true, wenn er mit den Werten aus Sätzen, die über ptr erreicht werden können, instanziiert wird

Was ist GiST?

Eine Bibliothek von Funktionen

- *Suche*: bekommt Wurzel eines Index und Prädikat; liefert Zeiger auf alle Sätze, die das Prädikat erfüllen
  - benötigt Operation **Consistent(E, q)**, die überprüft, ob ein Element (E = Prädikat, Zeiger) des Index q erfüllt
  - navigiert durch den Suchbaum
- *sortierte Suche*: wie *Suche*, nur daß Zeiger in sortierter Ordnung geliefert werden
  - auslesen der Zeiger durch *FindMin* und *Next* Operationen
  - nur sinnvoll bei bestimmten Indexen (z.B. nicht bei R Bäumen)

- *Insert*: bekommt Wurzel eines Index, Element (= Prädikat und Zeiger) und Level in den das Element eingefügt werden soll
  - normales Einfügen: Level = 0 (d.h. Einfügen am Blatt)
  - fügt Element ein
  - ruft *ChooseSubtree* auf, um den günstigsten Teilbaum auszuwählen
  - ruft *AdjustKeys* auf, um Element (Wegweiser) anzupassen
  - bei sortierten Indexen wird Operation **Compare($\mathbf{E_1}$, $\mathbf{E_2}$)** anstatt *ChooseSubtree* aufgerufen
- *ChooseSubtree*: bekommt Wurzel eines Index, Element und Level; liefert günstigsten Teilbaum
  - benötigt Operation **Penalty(F, E)**, um auf jeder Ebene den günstigsten Teilbaum auszuwählen
- *Split*: bekommt Knoten eines Baumes und ein neues Element; liefert zwei Knoten mit neuem Element eingefügt
  - benötigt Operation **PickSplit(P)**, die eine Menge von Elementen in zwei Teilmengen aufspaltet

- *AdjustKeys*: bekommt Wurzel eines Baumes und einen Knoten; paßt die Teilbaum vom Knoten
  - benötigt Operation **UNION(P)**, die ein umhüllendes Prädikat aus einer Menge von Prädikaten berechnet
- *Delete*: löschen eines Elementes

Instanziierung eines GiST, also Implementierung eines neuen Index, bedeutet die Definition der **fettgedruckten Operationen.**

Zusätzliches Gimmick

- Komprimierung von Elementen (i.e., Prädikaten)

Beispiel für GiST

Realisierung eines R Baumes für k=2 als GiST

- **Schlüssel, Prädikate**: (x_1, y_1, x_2, y_2)
- **Consistent(E, q)**:

  $E.x_1 \leq q.x_1 \wedge E.y_1 \leq q.y_1 \wedge E.x_2 \geq q.x_2 \wedge E.y_2 \geq q.y_2$
- **UNION($\mathbf{E_1}$, $\mathbf{E_2}$)**:

  $(\min(E_1.x_1, E_2.x_1), \min(E_1.y_1, E_2.y_1), \max(E_1.x_2, E_2.x_2), \max(E_1.y_2, E_2.y_2))$

- **Penalty($\mathbf{E_1}$, $\mathbf{E_2}$)**:

  1. $q =$ **UNION($\mathbf{E_1}$, $\mathbf{E_2}$)**
  2. return *area(q) - area(E_1.p)*
- **PickSplit(P)**: implementiere Split Heuristik nach Wahl

Bulkloading von Indexen

Ziel

- gegeben eine Menge von N Sätzen
- Ziel 1: baue Index für alle N Sätze möglichst schnell auf
- Ziel 2: der aufgebaute Index sollte möglichst kompakt sein

Naiver Ansatz

1. erzeuge leeren Index
2. füge alle Sätze einzeln ein

Bulkloading von B* Bäumen

Bewertung naiver Ansatz

- I/O Kosten: $N * \log_B N$
- Blätter sind nicht 100% voll; bei sortierter Eingabe entsteht evtl. der Worst Case
- einfach zu implementieren und funktioniert immer

Alternativer Ansatz

1. sortiere die Sätze
2. baue B* Baum von unten nach oben auf

Bewertung alternativer Ansatz

1. I/O Kosten: $N * \log_M N$
   (M ist die Größe des Hauptspeichers; $M \gg B$)
2. Blätter sind zu 100% gefüllt

Frage: Wie macht man Bulkloading von (erweiterbaren) Hashtabellen?

Bulkloading von R Bäumen

Bewertung naiver Ansatz

- funktioniert auch hier
- I/O Kosten: wiederum $N * \log_B N$
- Blätter sind nicht 100% voll

Alternativer Ansatz

- sortiere Sätze nach einer Dimension oder gemäß einer *space filling curve*
- baue R Baum von unten nach oben auf (Nachbarn kommen in denselben Knoten)

Bewertung Alternativer Ansatz

- I/O Kosten: $N * \log_M N$ (gut)
- Resultat ist ein häßlicher R Baum; durch Linearisierung entsteht sehr hoher *Overlap*.
- anders ausgedruckt: Effekt einer guten Split- und Einfügeheuristik ist verloren

Bulkloading von R Bäumen

Grundideen

- baue einen initialen "R Baum" mit M als Knotengröße auf
- Aufbau dieses initialen "R Baumes" geschieht als Pufferbaum
- übersetze den initialen "R Baum" Schicht für Schicht in einen (richtigen) R Baum mit B als Knotengröße

Bewertung

- I/O Kosten: $N * \log_M N$
  (asymptotisch optimal)
- nicht 100% kompakt, aber Ausnutzung der Split- und Einfügeheuristik
- funktioniert nicht nur für R Bäume sondern auch für GiST und andere Bäume, die sich nicht als GiST implementieren lassen

Literatur

- van den Bercken, Seeger, Widmayer, VLDB 1997