Datenbank-Programmierung

Chapter 5: Einführung in den physischen Entwurf

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2018/19

http://www.informatik.uni-halle.de/~brass/dbp19/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Eingabe-Parameter für den physischen Entwurf aufzählen.
- Die Wirkungsweise des Puffers (Caches) für Plattenblöcke im Hauptspeicher erklären.
- Die grundlegende Struktur eines B^+-Baums erklären.
- Indexe zur Beschleunigung einer Anfrage vorschlagen.
- Erklären, warum Indexe nicht nur Vorteile haben.
- Das grundlegende `CREATE INDEX` Kommando in SQL schreiben.

Inhalt

1. Einleitung
2. B-Baum Indexe
3. Vor- und Nachteile von Indexen
4. Weitere Fragen des physischen Entwurfs
5. Anhang

Motivation (1)

- Es sei folgende Tabelle mit Kundendaten gegeben:

CUSTOMERS			
CUSTNO	FIRST_NAME	LAST_NAME	···
1000001	John	Smith	···
1000002	Ann	Miller	···
1000003	David	Meyer	···
⋮	⋮	⋮	⋮

- Weiter sei angenommen, dass die Tabelle recht groß ist.

  Z.B. gebe es 2 Millionen Datensätze. Wenn die Tabelle z.B. die Spalten FIRST_NAME, LAST_NAME, STREET, CITY, STATE, ZIP, PHONE, EMAIL hat, sind ca. 100 Byte Speicherplatz pro Zeile realistisch, und mit noch einigen Aufschlägen (Block-Header, Platzreserven) ca. 230 MB für die ganze Tabelle.

Motivation (2)

- Seien nun die Daten für einen bestimmten Kunden gefragt:

```
SELECT *
FROM   CUSTOMERS
WHERE  CUSTNO = 1000002
```

- Ohne spezielle Datenstrukturen geht das DBMS einfach alle Tabellenzeilen durch und prüft jeweils die Bedingung `CUSTNO = 1000002` ("Full Table Scan").

- Die Anfrage wird ungefähr 3 Sekunden laufen.

  Magnetplatten schaffen heute ungefähr 80 MB/s, wenn die Daten sequentiell auf der Platte gespeichert sind, und sie nichts anderes zu tun haben.

  3 Sekunden sind schon eine spürbare Verzögerung, man wünscht sich Antwortzeiten von unter einer Sekunde für flüssiges Arbeiten.

  Bei 100 Sachbearbeitern: Rechnerisch jeder nur eine Anfrage pro 5 Minuten!

Physischer Datenbank-Entwurf (1)

- Die Aufgabe des physischen Datenbank-Entwurfs ist es, sicherzustellen, dass das Datenbanksystem die Leistungs-Anforderungen erfüllt.
- Dies benötigt Wissen über (bzw. Schätzungen für):
  - Größe der Tabellen, Verteilung der Daten.
  - Welche Anwendungsprogramme gibt es, welche Anfragen und Updates enthalten sie, und wie häufig pro Stunde werden sie ausgeführt?
  - Was sind die Laufzeit-Anforderungen für welches Programm?

    Häufig benutze interaktive Programm sollten normalerweise Antwortzeiten von unter einer Sekunde haben, selten benutzte Programm können etwas langsamer sein, und manche Berichte/Statistiken können über Nacht berechnet werden.

Physischer Datenbank-Entwurf (2)

- Da die Größen und Ausführungs-Frequenzen schwer zu schätzen sind und sich im Laufe der Zeit ändern, muss man den physischen Entwurf im Laufe der Zeit anpassen.

  In relationalen Systemen ist es einfach, einen Index neu anzulegen oder einen zu löschen. Wenn man natürlich ganz neue Hardware kaufen muss, weil die Leistungsanforderungen anders nicht erfüllt werden können, hat man ein Problem. Es ist wichtig, beim Entwurf über eine realistische Systemlast nachzudenken. Es gibt Werkzeuge, um eine Systemlast zu simulieren [https://en.wikipedia.org/wiki/Load_testing]. Prüfen Sie vor Einführung einer neuen Software, dass sie den Anforderungen gewachsen ist.

- Physischer Entwurf hängt stark vom gewählten DBMS ab.

  Man muss die Funktionsweise des Systems hinreichend verstehen. Siehe Vorlesung "Datenbanken II B: DBMS-Implementierung" im Master Studium.

Platten

- Normalerwise ist die Zeit zum Lesen eines Plattenblocks viel länger als die Zeit für Berechnungen im Hauptspeicher.
- Daher war es früher üblich, für Datenbank-Anfragen nur zu berechen, wie viele Plattenblöcke gelesen werden müssen.

  Ein Block ist eine Einheit von ca. 2–8 KByte, die als Ganzes zwischen Platte und Hauptspeicher bewegt werden. Physisch kann man nur Sektoren von typisch 512 Byte lesen oder schreiben. Meist werden aber eine feste Anzahl hintereinander gespeicherter Sektoren zu einer größeren Einheit (Block) zusammengefasst, um den Overhead pro Block zu verkleinern.

- Ein wahlfreier Blockzugriff dauert ca. 2–10 ms, die CPU kann in dieser Zeit z.B. 100 Millionen Instruktionen ausführen.

  Der Schreib/Lesekopf muss sich zur richtigen Spur bewegen und dann warten, bis sich der gewünsche Block unter dem Kopf durchdreht (typische Geschwindigkeiten sind 5400 bis 15 000 Umdrehungen pro Minute).

Solid State Disks (SSDs)

- Solid-State Disks benutzen (wie USB-Sticks) Flash Speicher.

  Es gibt also keine beweglichen Teile.

- Während bei Magnetplatten sequentielles Lesen oder Schreiben wesentlich schneller ist als auf entfernte Blöcke zuzugreifen, sind Lesezugriffe bei SSDs überall gleich schnell.

  Bei Schreibzugriffen ist es günstig, wenn größere Einheiten geschrieben werden, da diese intern gelöscht und neu beschrieben werden müssen (z.B. 2 MB).

- Ein Blockzugriff dauert z.B. 0.1 ms,
  eine typische Lese/Schreibrate ist 200–500 MB/s.

- Die Anzahl der Schreibvorgänge pro Block ist begrenzt.

  Z.B. 3000 Mal. Der Controller versucht, alle Blöcke gleich häufig zu schreiben.

- Der Preis pro GB ist ca. das Zehnfache einer Magnetplatte.

Pufferung: Cache für DB-Blöcke

- Eine Kopie der zuletzt gelesenen Datenbank-Blöcke wird im Hauptspeicher (RAM) gehalten (Puffer, Cache).

  Das Lesen einer Cache-Line aus dem RAM (64 Byte) kostet z.B. 60 ns, ein Plattenblock (8 KB) wird in ca. 6 ms gelesen: Faktor von ca. 1:100.000.

- Man geht normalerweise davon aus, dass die Datenbank größer als der Hauptspeicher ist, deswegen können nicht alle Blöcke im Hauptspeicher gehalten werden.

  Der Hauptspeicher ist heute größer, aber die zu speichernden Daten auch.

- Die erste Ausführung einer Anfrage dauert meist deutlich länger als wenn man sie anschließend ein zweites Mal ausführt.

  Dann sind alle benötigten Datenbank-Blöcke ja schon im Hauptspeicher. Bei kleinen Datenbanken sind bald alle Blöcke im Speicher, die Platte wird dann nur noch benötigt, um Updates dauerhaft zu machen (nicht für Anfragen).

Indexe (1)

- Ein Index in einem Datenbank-Managementsystem ist eine Datenstruktur zum schnellen Zugriff auf alle Tabellenzeilen mit einem bestimmten Wert in einer bestimmten Spalte.

  Einige Indexstrukturen können auch andere einfache Bedingungen beschleunigen, nicht nur $A = c$, sondern z.B. auch $A > c$.

- Ein Index ist ja auch aus Büchern bekannt: Dort ist es eine sortierte Liste aller (wichtigen) Worte des Buches, jeweils zusammen mit einem Verweis auf die Vorkommen dieses Wortes (Seitennummern).

- Da SQL eine deklarative Sprache ist, müssen SQL-Anfragen nicht angepasst werden, nachdem man einen Index angelegt oder gelöscht hat (physische Datenunabhängigkeit).

  Der Anfrageoptimierer wählt automatisch einen günstigen Auswertungsplan.

Indexe (2)

Fußnote: Indexe vs. Indices

- Als Plural von "Index" ist in der DB-Literatur "Indexe" üblich, obwohl "Indices" auch vorkommt (selten).

  Z.B. verwenden Kemper/Eickler, Vossen, Härder/Rahm und die deutsche Übersetzung von Elmasri/Navathe den Plural "Indexe". In meinem deutschen Wörterbuch steht als Plural nur "Indices" bzw. "Indizes", aber das deckt sich nicht mit meinem persönlichen Sprachempfinden. In der Wikipedia stehen "Indexe", "Indices", "Indizes" als (offenbar gleichberechtigte) Alternativen (speziell für Datenbanken). Als Genitiv sind "des Indexes" und "des Index" beide möglich.

- Dagegen besteht Einigkeit, dass die kleinen tiefgestellten Buchstaben (wie i in x_i) "Indices" sind.

  Das gilt natürlich auch in der Datenbank-Literatur.

Inhalt

1. Einleitung
2. B-Baum Indexe
3. Vor- und Nachteile von Indexen
4. Weitere Fragen des physischen Entwurfs
5. Anhang

B^+-Bäume (1)

- Die häufigste Index-Datenstruktur in Datenbanken ist der B^+-Baum.

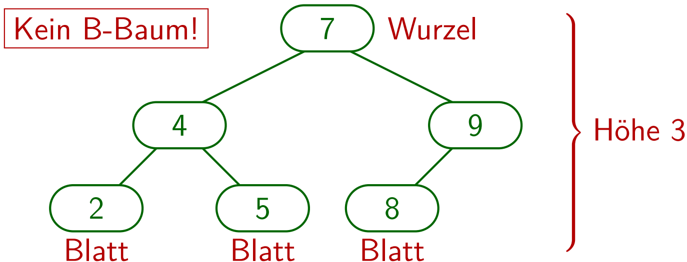
  Jedes moderne DBMS enthält eine Variante von B-Bäumen.
  Zusätzlich enthält es eventuell weitere spezialisierte Indexstrukturen
  (z.B. Hashverfahren, Bitmap-Indexe, R-Bäume).

- B-Bäume sind (vermutlich) nach ihrem Erfinder, Rudolf Bayer (TU München / Transact Software), benannt.

  Bayer/McCreight: Organization and Maintenance of Large Ordered Indices.
  Acta Informatica 1(3), 173–189, 1972.
  Es wäre aber auch möglich, dass das "B" für "balanced" steht (es ist ein balancierter Baum), oder auch für "Boeing" (die Autoren arbeiteten damals für Boeing). [https://en.wikipedia.org/wiki/B-tree]
  Bayer nannte den B^+-Baum ursprünglich B^*-Baum, aber das steht heute eher für eine von Knuth vorgeschlagene Variante mit Knoten-Füllungsgrad $> 2/3$.

B^+-Bäume (2)

- Im Prinzip funktionieren B-Bäume wie binäre Suchbäume, die aus Datenbstruktur-Vorlesungen bekannt sein sollen:



Die Suche startet im Wurzelknoten (ganz oben). Wenn der gesuchte Wert im aktuellen Knoten eingetragen ist, hat man ihn gefunden und ist fertig. Ist andernfalls der gesuchte Wert kleiner als der Wert im aktuellen Knoten, wird die Suche im linken Teilbaum fortgesetzt, sonst im rechten Teilbaum. Ist der jeweilige Teilbaum leer, so fehlt der gesuchte Wert im Baum.

B^+-Bäume (3)

- In einem B-Baum ist der Verzweigungsgrad wesentlich größer als 2.

  B-Bäume sind für Externspeicher (Platten) gedacht. Dort muss man ohnehin einen ganzen Block (z.B. 8 KB) lesen. Die Größe eines Knotens im B-Baum ist gerade diese Blockgröße.

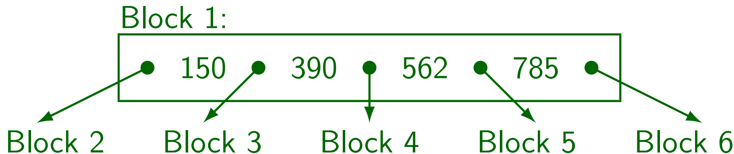
- Gewöhnliche binäre Suchbäume können zu einer linearen Liste degenerieren (bei Einfügung in sortierter Reihenfolge). Bei B-Bäumen kann das nicht geschehen, sie sind balanciert.

- In einem B^+-Baum (nicht im B-Baum) werden die Werte der inneren Knoten (Nicht-Blätter) in den Blättern wiederholt.

  Der Verzweigungsgrad wird größer, da die Zeiger auf die Tabellenzeilen dann in den inneren Knoten nicht nötig sind. Das kann die Baumhöhe verringern. Außerdem hat man auf Blatt-Ebene eine sortierte Liste.

B^+-Bäume (4)

- B^+-Bäume besteht aus Verzweigungsknoten ("branch blocks") und Blattknoten ("leaf blocks").
- Verzweigungsknoten steuern die Suche nach einem Datenwert:



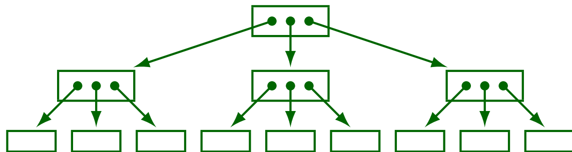
- Ist der gesuchte CUSTNO-Wert ≤ 150: Weiter bei Block 2.

  Man muss sich bei der Implementierung eines B^+-Baums entscheiden, ob man bei "$=$" nach links oder rechts geht.

- Bei CUSTNO > 150 und CUSTNO ≤ 390: gehe zu Block 3.
  ...
  Ist CUSTNO > 785: weiter in Block 6.

B^+-Bäume (5)

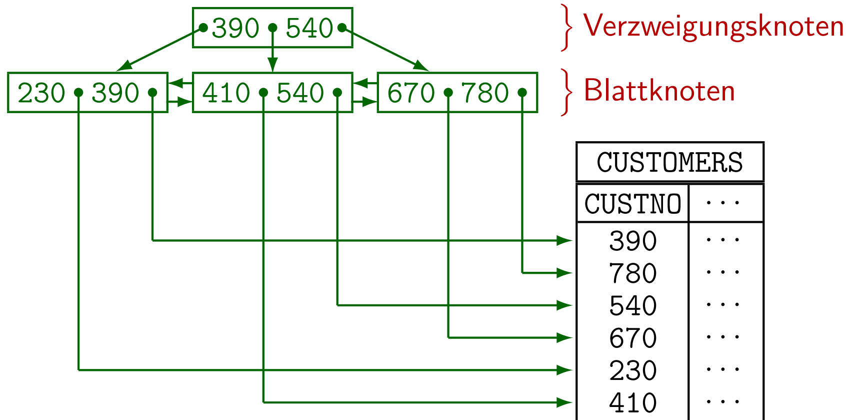
- Die referenzierten Blöcke (bis auf die unterste Ebene) haben die gleiche Struktur, so dass ein Baum entsteht:



- Die Blöcke auf unterster Ebene ("Blattknoten") enthalten alle vorkommenden Kundennummern in sortierter Reihenfolge zusammen mit der Speicheradresse ("ROWID") der zugehörigen Zeile der Tabelle "`CUSTOMERS`".

  Die Blattknoten sind auch untereinander verkettet, so dass man die sortierte Reihenfolge leicht durchlaufen kann.

B$^+$-Bäume (6)



B^+-Bäume (7)

- In einem B/B^+-Baum haben alle Blattknoten den gleichen Abstand (Anzahl Kanten) von der Wurzel. Daher sind B/B^+-Bäume balanciert.

  Dadurch ist sichergestellt, dass die Folge von Verweisen, die man von der Wurzel zu einem Blattknoten durchlaufen muss, nie sehr lang ist.
  Für B/B^+-Bäume ist die Komplexität der Suche $O\big(\log(n)\big)$, wobei n die Anzahl der Einträge ist (gute Komplexität für DBen mit oft sehr großen n).

- Knoten in einem B/B^+-Baum können unterschiedlich viele Werte enthalten, aber jeder Knoten (bis auf die Wurzel) muss mindestens halb voll sein.

  Wenn man verlangen würde, dass alle Blöcke (bis auf einen) komplett gefüllt sind, könnte die Einfügung eines Wertes eine völlige Umstrukturierung des Baums erfordern. Durch die abgeschwächte Anforderung sind auch Einfügung und Löschung in $O\big(\log(n)\big)$ möglich.

B^+-Bäume (8)

Einfügungs-Algorithmus:

- Suche den Blatt-Knoten, in den der neue Wert eingefügt werden muss.
- Wenn dieser Knoten noch ausreichend freien Platz hat: Fügen den neuen Wert (plus Verweis auf Tabellenzeile) ein. Fertig.
- Andernfalls spalte den Blattknoten (unter Einrechnung des neuen Eintrags) in der Mitte auf. Füge dabei ein.

  Aus einem 100% vollen Knoten werden zwei 50% volle Knoten, plus der neue Eintrag. Der vorhandene Zeiger im Verzweigungsknoten darüber muss auf den rechten Blatt-Knoten zeigen, und der größte Wert im linken Knoten muss mit einem Zeiger auf den linken Knoten in den Verzweigungsknoten neu eingefügt werden.

B^+-Bäume (9)

Einfügungs-Algorithmus, Forts.:

- Hat der Verzweigungsknoten ausreichend Platz, ist man fertig. Sonst spaltet man den Verzweigungsknoten auf und fügt den mittleren Datenwert in die Ebene darüber ein.

  Das Aufspalten von Verzweigungsknoten funktioniert etwas anders als bei den Blattknoten: Beim Blattknoten wird der mittlere Datenwert in der Ebene darüber dupliziert, bei Verzweigungsknoten wird er verschoben.

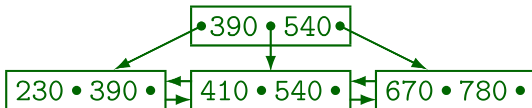
- Und so weiter. Wenn man am Ende die Wurzel aufspaltet, legt man darüber eine neue Wurzel an.

  B/B^+-Bäume wachsen also nach oben, während bei binären Suchbäumen typischerweise unten neue Blattknoten angefügt werden. Das würde bei B/B^+-Bäumen gar nicht gehen, weil die Distanz von der Wurzel zum Blatt für alle Blätter gleich sein muss. Die neue Wurzel enthält anfangs einen Wert und zwei Zeiger (auf die beiden Hälften des aufgespaltenen Knotens).

B^+-Bäume (10)

Aufgabe:

- Betrachten Sie nochmals den B^+-Baum von Folie 19:



- Angenommen, jeder Knoten kann höchstens zwei Werte enthalten (und muss daher mindestens einen enthalten).
- Fügen Sie den Wert 123 ein.
- Geben Sie ein Beispiel für einen Wert, der nun eingefügt werden kann, ohne weitere Knoten aufzuspalten.

B^+-Bäume (11)

- Der Verzweigungsgrad in der Praxis ist normalerweise weit größer als in den obigen Beispielen.
- Ein Block von 2 KB kann ca. 100 Kundennummern (vom Typ `NUMERIC(7)`) und die zugehörigen ROWIDs enthalten.

Höhe	Min. Anz. Zeilen	Max. Anz. Zeilen
1	1	100
2	$2 * 50 = 100$	$100^2 = 10\,000$
3	$2 * 50^2 = 5\,000$	$100^3 = 1\,000\,000$
4	$2 * 50^3 = 250\,000$	$100^4 = 100\,000\,000$

Höhe 1: Nur die Wurzel, die in diesem Fall ein Blattknoten ist.

Höhe 2: Die Wurzel als Verzweigungsknoten, darunter die Blattebene, wie im Beispiel auf Folie 19.

B^+-Bäume (12)

- Für die Tabelle `CUSTOMERS` mit 2 000 000 Zeilen hat der B^+-Baum unter den obigen Annahmen die Höhe 4.

  Die Höhe 5 würde mindestens $2 * 50^4 = 12.5$ mio Zeilen erfordern.

- Ein Baum der Höhe 4 benötigt also vier Blockzugriffe für den Index und einen für die Tabelle, um die gewünschte Zeile zu finden.

  Im Index ist die physische Speicheradresse der Zeile eingetragen: Diese ROWID besteht z.B. aus Dateinummer, Blocknummer innerhalb der Datei und Zeilennummer innerhalb des Blocks. In seltenen Fällen ("migrated row"), musste die Zeile aufgrund verlängender Updates in einen anderen Block verschoben werden, die würde noch einen sechsten Blockzugriff erfordern. Es gibt aber keine längeren Verweisketten (siehe Vorlesung "Datenbanken II B").

- Die Anfrage kann daher in 50 ms ausgeführt werden.

  10 ms pro wahlfreien Blockzugriff kann man heute von Platten erwarten.

B^+-Bäume (13)

- Tabellenzugriffe über einen Index profitieren auch besonders von der Pufferung der Plattenblöcke.

  Es ist sehr wahrscheinlich, dass die Wurzel der Baumstruktur im Hauptspeicher ist, wenn der Index mehrfach benutzt wird. Auch zumindest ein Teil der nächsten Ebene darunter wird sich bald im Puffer befinden.

- Da die Höhe von B/B^+-Bäumen nur logarithmisch in der Anzahl der Zeilen wächst, werden diese Bäume in der Praxis nie besonders hoch.

  Im Beispiel erforderte die Höhe 4 schon mindestens eine Viertelmillion Zeilen. Natürlich hängt die genaue Höhe auch von der Größe der Datenwerte ab: Wenn es längere Zeichenketten sind, passen weniger Datenwerte in einen Knoten. Aber selbst wenn die Minimalfüllung eines Knotens nur ein einziger Datenwert ist (maximal zwei), der schlechteste Fall also ein binärer Baum ist, hätte ein Index für eine Tabelle mit einer Million Zeilen nur die Höhe 20.

B^+-Bäume (14)

- Der Index für die Spalte CUSTNO der Tabelle CUSTOMERS ist ein UNIQUE INDEX: Es gibt nur eine Zeile für jeden Datenwert.

  CUSTNO ist Schlüssel der Tabelle CUSTOMERS.

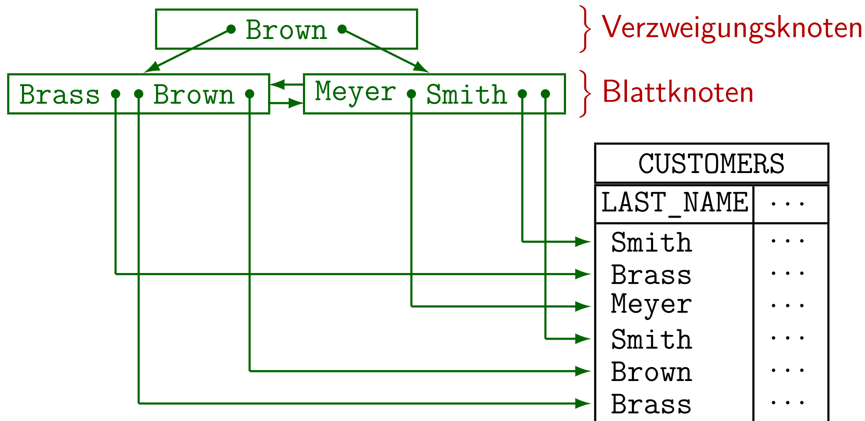
- In einem normalen Index (nicht "UNIQUE") können zu einem Datenwert mehrere Zeilen-Adressen gespeichert werden. Beispiel: Index über der Spalte LAST_NAME.

  Obwohl LAST_NAME nicht Schlüssel der Tabelle ist, wird er in diesem Zusammenhang gelegentlich "(Such-)Schlüssel" der Baumstruktur genannt.

- Datenbanksysteme legen üblicherweise für deklarierte Schlüssel (PRIMARY KEY oder UNIQUE) automatisch einen UNIQUE INDEX an (um den Schlüssel zu überwachen).

  Explizit anlegen wird man also eher normale (nicht UNIQUE) Indexe.

B$^+$-Bäume (15)



B^+-Bäume (16)

- Es ist auch möglich, einen Index über einer Kombination von zwei oder mehr Spalten anzulegen.

  Man kann sich das so vorstellen, dass die Konkatenation der beiden Spaltenwerte (mit eindeutigen Trennzeichen) in den Index eingetragen wird.

- Weil B^+-Baum Indexe auf einer Sortierung basieren, kann es für die Verwendung des Indexes einen Unterschied machen, ob man den Index über der Kombination (A, B) oder (B, A) anlegt (Erstes Attribut: Haupt-Sortierkriterium).

- Z.B. kann man einen Index über `(LAST_NAME, FIRST_NAME)` auch verwenden, um Zeilen für einen gegebenen `LAST_NAME` zu finden (mit beliebigem `FIRST_NAME`).

  Bei dieser Sortierung stehen ja alle Einträge mit dem gegebenen `LAST_NAME` hintereinander in den Blattknoten.

Inhalt

1. Einleitung
2. B-Baum Indexe
3. Vor- und Nachteile von Indexen
4. Weitere Fragen des physischen Entwurfs
5. Anhang

Anwendungen von Indexen (1)

- Ein Index über der Spalte A der Relation R ist besonders nützlich für Gleichheits-Bedingungen der Form $A = c$ für Tupelvariablen über R (mit einer Konstanten c).
- In der relationalen Algebra: Selektion $\sigma_{A=c}(R)$.
- Ein B$^+$-Baum Index kann auch für Bedingungen mit den Vergleichsoperatoren <, <=, >, >= genutzt werden (Bereichsanfragen, auch `LIKE` mit gegebenem Präfix).
- Indexe sind aber nur nützlich, wenn nur auf einen sehr kleiner Teil der Zeilen zugegriffen wird.

  Der genaue Prozentsatz, bei dem es sich lohnt, hängt von vielen Faktoren ab, und liegt oft in der Größenordnung von 1%. Wenn viele Zeilen die Bedingung erfüllen, ist ein "Full Table Scan" schneller: Sequentielles Lesen von Magnetplatten ist ungefähr 100 Mal schneller als wahlfreie Blockzugriffe.

Anwendungen von Indexen (2)

- Ein Index kann auch dann genutzt werden, wenn es noch weitere Bedingungen gibt außer denen, die vom Index unterstützt werden:

```
SELECT *
FROM   CUSTOMERS
WHERE  LAST_NAME = 'Smith'
AND    CITY = 'Pittsburgh'
```

- Wenn es einen Index über dem Attribut `LAST_NAME` gibt, kann das DBMS ihn nutzen, um alle Zeilen zu finden, die die erste Bedingung erfüllen, und dann die zweite Bedingung für jede solche Zeile prüfen:

$$\sigma_{CITY=\text{'Pittsburgh'}}\Big(\sigma_{LAST_NAME=\text{'Smith'}}(CUSTOMERS)\Big)$$

Dabei benutzt die innere Selektion den Index.

Anwendungen von Indexen (3)

- Ein Verbund (Join) kann mit einem Index auf einer der verknüpften Spalten ausgewertet werden.
- Beispiel:

```
SELECT C.LAST_NAME
FROM   INVOICES I, CUSTOMERS C
WHERE  I.AMOUNT > 20000
AND    C.CUSTNO = I.CUSTNO
```

- Das DBMS könnte zuerst große Rechnungen `I` finden (vielleicht gibt es dazu auch einen Index), und dann mit einem Index über `CUSTOMERS(CUSTNO)` die Kunden-Daten zu jeder solchen Rechnung holden.

  Jede einzelne Zeile `I` enthält eine Kundennummer. Mit dieser Kundennummer kann man dann die zugehörige Zeile `C` finden. Im Prinzip funktioniert das wie eine iterierte Selektion mit einer Konstanten.

Anwendungen von Indexen (4)

- Manche Anfragen können schon aus dem Index beantwortet werden, ohne auf die Tabelle selbst zuzugreifen, z.B. "Gibt es einen Kunden mit einer gegebenen Kundennummer?"

  Dies ist wichtig, um Schlüssel und Fremdschlüssel zu überwachen, deswegen legt das DBMS üblicherweise automatisch einen Index für jeden Schlüssel an.

- Sortieraufgaben können eventuell von einem Index profitieren.

  Dies kann z.B. `ORDER BY`, `GROUP BY` und `DISTINCT` beschleunigen.
  Die Blattknoten eines B^+-Baums enthalten alle in der Spalte vorkommenden Werte in Sortierreihenfolge. Allerdings bringt das nur etwas, wenn ein relativ kleines Intervall abgefragt wird (Bereichsanfrage), oder wenn die Anfrage ganz aus dem Index beantwortet werden kann. Sonst kann der Zugriff auf die Tabellenzeilen (über den Zeiger "ROWID" im Index) länger dauern als z.B. ein Lauf des "Mergesort" Algorithmus, der auf die Daten in sequentieller Reihenfolge zugreift, aber mehrere Durchläufe benötigt.

Anwendungen von Indexen (5)

- Indexe über Kombinationen von zwei oder mehr Spalten sind besonders günstig für Anfragen, in denen Werte für alle diese Spalten gegeben sind:

```
SELECT *
FROM   CUSTOMERS
WHERE  FIRST_NAME='John'
AND    LAST_NAME='Smith'
```

- Hier z.B. Index über `CUSTOMERS(LAST_NAME, FIRST_NAME)`.

  Für diese Anfrage spielt die Reihenfolge der beiden Attribute im Index keine große Rolle. Wenn der Index ein B^+-Baum ist, kann er aber auch verwendet werden, wenn nur ein Präfix der Attribute bekannt ist (hier nur `LAST_NAME`). Da der Nachname eher bekannt sein wird als der Vorname, bietet sich diese Reihenfolge an. Ein eigener Index nur über `LAST_NAME` wäre noch etwas günstiger, aber wegen der Kosten (s.u.) würde man den meist nicht zusätzlich anlegen.

Anwendungen von Indexen (6)

Aufgabe:

- Es sei die folgende Anfrage gegeben:

```
SELECT C.FIRST_NAME, C.LAST_NAME, C.PHONE
FROM   CUSTOMERS C, ORDERS O, ORDER_DETAILS D
WHERE  D.PRODNO = 123
AND    C.CITY = 'Pittsburgh'
AND    O.ORD_DATE >= '01-JAN-19'
AND    C.CUSTNO = O.CUSTNO AND O.ORDNO = D.ORDNO
```

- Welche Indexe könnten für diese Anfrage günstig sein? Skizzieren Sie zwei verschiedene Auswertungs-Möglichkeiten.

  Welches Wissen über die Daten würde das DBMS benötigen, um zu entscheiden, welche Auswertungs-Möglichkeit günstiger ist?

Nachteile von Indexen

- Indexe kosten Plattenplatz.

  Der Beispiel-Index über `CUSTOMERS(CUSTNO)` benötigt ca. 30–50% des Plattenplatzes der Tabelle. Zu einer Tabelle kann es viele Indexe geben.

- Anfragen werden zwar schneller, aber Updates werden langsamer, da die Indexe auch aktualisiert werden müssen.
- Die Anfrage-Optimierung muss mehr Alternativen zur Anfrageauswertung bewerten.

  Wenn die Anfrage aus einem Programm ist, wird sie aber meist häufig ausgeführt, wobei sich die Kosten der Anfrage-Optimierung amortisieren.

- Falls auf die meisten Blöcke der Tabelle sowieso zugegriffen werden muss, läuft ein "Full Table Scan" viel schneller.

  Weil sequentielles Lesen von der Platte deutlich schneller geht als wahlfreie Zugriffe mit vielen Sprüngen.

Tipps zur Auswahl von Indexen

- Prüfen Sie, ob der Anfrage-Optimierer den Index überhaupt verwendet.

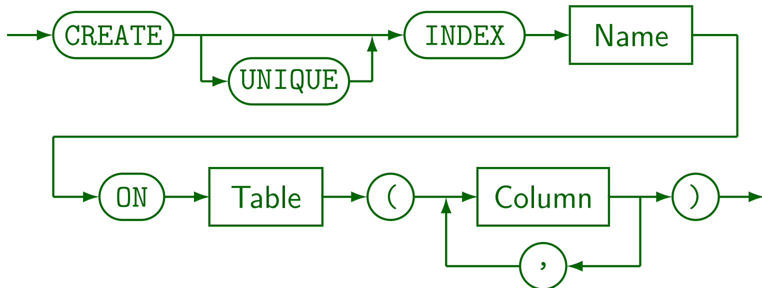
  Sonst haben Sie die Kosten ohne irgendeinen Nutzen. Bei den meisten DBMS kann man sich die Anfrage-Auswertungspläne anzeigen lassen.

- Indexe für kleine Tabellen sind überflüssig (außer für Schlüssel).
- Deklarieren Sie nicht zu viele Indexe auf Tabellen, auf denen es viele Updates gibt.
- Ein Index ist normalerweise nur nützlich, wenn weniger als 1% der Zeilen die Bedingung erfüllt.
- Indexe, die schon alle notwendigen Daten für die Anfrage enthalten, sind besonders praktisch.

  Ggf. auch wenn auf alle Einträge zugegriffen wird, liefert Sortierreihenfolge.

Indexe in SQL (1)

- Typisches SQL-Kommando, um einen Index anzulegen:



- Beispiele:
  - `CREATE INDEX CUSTIND1 ON CUSTOMERS(CITY)`
  - ```
    CREATE UNIQUE INDEX CUSTIND2 ON
           CUSTOMERS(LAST_NAME, FIRST_NAME, CUSTNO)
    ```
    ```

Indexe in SQL (2)

- Das `CREATE INDEX` Kommando ist nicht in den SQL Standards enthalten, sollte aber von den meisten DBMS akzeptiert werden (mit systemspezifischen Erweiterungen).

  Die SQL Standards behandeln keine physischen Speicherstrukturen. Diese sind grundsätzlich systemabhängig.

- `CREATE UNIQUE INDEX` bewirkt, dass es für jeden Datenwert im Index nur eine Zeile geben kann.

  D.h. die Spalte oder Spaltenkombination, über der der Index aufgebaut wird, ist ein Schlüssel (nicht notwendig minimal). Ganz alte SQL-Skripte enthalten eventuell statt Schlüssel-Deklarationen "`CREATE UNIQUE INDEX`" Befehle.

- Die meisten DBMS legen für Schlüssel (`PRIMARY KEY` und `UNIQUE`) automatisch einen Index an.

  Damit werden die Schlüssel dann effizient mit einem Index überwacht. Es wäre falsch, einen weiteren Index für die gleiche Attributkombination anzulegen.

Indexe in SQL (3)

- Wenn man einen Index auf einer großen Tabelle anlegt, kann das
  - eine ganze Zeit dauern,
  - viel temporären Plattenplatz verbrauchen,
  - und die Tabelle die ganze Zeit für Updates sperren.

    Man sollte also mit Indexen nicht zu den Haupt-Geschäftszeiten experimentieren. Moderne Systeme sollten zumindest die Möglichkeit haben, die Tabelle nicht lange zu sperren (lesen Sie das Handbuch). In DB2 müssen Sie nach Änderungen relevanter Indexe eventuell ein `REBIND` für Pakete zu Anwendungen mit eingebettetem SQL durchführen.

- Kommando zum Löschen eines Index:

Beispiel: `DROP INDEX CUSTIND1`

Hinweise zu PostgreSQL (1)

- `EXPLAIN SELECT ...` gibt einen Auswertungsplan für die Anfrage aus.

  Die Variante `EXPLAIN ANALYZE SELECT ...` führt die Anfrage wirklich aus und gibt Laufzeiten für die Knoten aus. Es gibt auch `EXPLAIN VERBOSE` bzw. `EXPLAIN ANALYZE VERBOSE` für mehr Informationen.
  Sie müssen den Auswertungsplan nicht verstehen, Auswertungspläne (momentan für Oracle) werden in "Datenbanken IIB" ausführlich besprochen. Sie können so aber kontrollieren, ob der Index überhaupt verwendet wird. Es wäre ja ungünstig, für einen Index zu "bezahlen" (Speicherplatz, langsamere Updates) und überhaupt keinen Vorteil davon zu haben.
  [https://thoughtbot.com/blog/reading-an-explain-analyze-query-plan]
  [https://www.postgresql.org/docs/10/using-explain.html]

Hinweise zu PostgreSQL (2)

- Statistiken für die Optimierer werden mit folgendem Befehl erstellt: `ANALYZE Tabellename`.

  Dadurch kann sich der gewählte Auswertungsplan verändern/verbessern (z.B. bei kleinen Tabellen eher keine Index-Nutzung).

- Wenn man in `psql` mit dem Befehl "`\d Tabellenname`" Informationen zu einer Tabelle anzeigen lässt, werden dabei auch die existierenden Indexe für diese Tabelle aufgelistet.

  Der Systemkatalog enthält auch eine Tabelle `pg_indexes` mit Spalten `schemaname`, `tablename`, `indexname`, `tablespace`, `indexdef`.
  Dabei enthält `indexdef` einen `CREATE INDEX` Befehl. Es gibt auch eine Tabelle `pg_index` mit interneren Informationen. Sie enthält nur die `oid` der Tabelle, den Tabellennamen erhält man durch Join mit `pg_class` wobei `pg_class.oid = pg_index.indrelid`, (Index-Name ebenso mit `indexrelid`).
  Siehe [https://www.postgresql.org/docs/9.4/ddl-system-columns.html] und [https://www.postgresql.org/docs/9.4/catalog-pg-index.html].

Hinweise zu PostgreSQL (3)

- In `psql` kann man mit dem Befehl "`\timing on`" verlangen, dass für zukünftige Anfragen und Updates die Dauer (abgelaufene Realzeit) ausgegeben wird.

  Diese Einstellung gilt bis zum Ende der `psql`-Sitzung.

- Die Wirkung eines Indexes wird sich aber nur für große Tabellen nachweisen lassen, bei denen sehr wenige Zeilen gesucht sind.

  Beachten Sie auch die Pufferung, aufgrund derer der erste Durchlauf immer deutlich länger dauert als wenn die Datenbank-Blöcke schon im Hauptspeicher sind.

- Der `CREATE INDEX` Befehl in PostgreSQL hat noch wesentlich mehr Möglichkeiten, siehe [https://www.postgresql.org/docs/9.4/sql-createindex.html]

Inhalt

1. Einleitung
2. B-Baum Indexe
3. Vor- und Nachteile von Indexen
4. Weitere Fragen des physischen Entwurfs
5. Anhang

Mehr zum physischen Entwurf (1)

- Die Index-Auswahl ist das klassische Problem des physischen Entwurfs.
- Aber es gibt noch viele andere Möglichkeiten, um die Geschwindigkeit der Anfragebearbeitung zu steigern.

  So viel, dass es eine eigene Vorlesung verdient: Hören Sie im Master-Studium "Datenbanken IIB: DBMS-Implementierung", wenn Sie diese Fragen interessieren.

- Normalerweise wird die Tabelle selbst als "Heap File" gespeichert, also ohne bestimmte Ordnung.

  Dies hat nichts mit "Heap Sort" zu tun. Zeilen werden gespeichert, wo Platz ist.

- Selbst dafür gibt es verschiedene Speicherparameter.

  Z.B. wie viel Platz im Block für Updates reserviert wird (aufgrund der Pointer aus dem Index ist es ungünstig, wenn eine Zeile in einen anderen Block "migriert" werden muss). PostgreSQL: `CREATE TABLE ...(...) WITH (fillfactor=70)`

Mehr zum physischen Entwurf (2)

- Es gibt oft auch weitere Indexstrukturen (neben den B^+-Bäumen, die jedes DBMS unterstützen sollte), z.B.
  - Hash-Verfahren

    Hier wird die Position der Zeile (Plattenblock) aus einem Datenwert berechnet. Wenn es gut funktioniert, hat man z.B. zu gegebener Kundennummer mit einem Blockzugriff die Zeile. Aber weniger flexibel.
  - Bitmap-Indexe

    Index über $R(A)$ enthält für jedes Tupel $t \in R$ und jeden Datenwert c ein Bit, das zeigt, ob $t.A = c$. Nützlich bei wenigen verschiedenen Werten.
  - Cluster (Oracle)

    Speicherung von Zeilen mit gleichem Datenwert im gleichen Block.
  - Index-organized Tables

    Statt vom Index Zeigern auf die Zeilen folgen zu müssen, kann man auch die Zeilen selbst in einem B^+-Baum speichern. (Weitere Indexe schwierig.)

Inhalt

1. Einleitung
2. B-Baum Indexe
3. Vor- und Nachteile von Indexen
4. Weitere Fragen des physischen Entwurfs
5. Anhang

Lösung der Aufgabe (1)

Aufgabe (von Folie 36):

- Es sei die folgende Anfrage gegeben:

```
SELECT C.FIRST_NAME, C.LAST_NAME, C.PHONE
FROM   CUSTOMERS C, ORDERS O, ORDER_DETAILS D
WHERE  D.PRODNO = 123
AND    C.CITY = 'Pittsburgh'
AND    O.ORD_DATE >= '01-JAN-19'
AND    C.CUSTNO = O.CUSTNO AND O.ORDNO = D.ORDNO
```

- Welche Indexe könnten für diese Anfrage günstig sein? Skizzieren Sie zwei verschiedene Auswertungs-Möglichkeiten.

  Welches Wissen über die Daten würde das DBMS benötigen, um zu entscheiden, welche Auswertungs-Möglichkeit günstiger ist?

Lösung der Aufgabe (2)

Auswertungsplan 1:

- Wenn es nicht viele Bestellungen für das Produkt 123 gibt, könnte man damit beginnen. Ideal wäre ein Index über ORDER_DETAILS(PRODNO, ORDNO).

  Dann würde man D.ORDNO schon aus dem Index bekommen, und müsste nicht mehr auf die Tabelle selbst zugreifen. Weniger extrem wäre ein Index über ORDER_DETAILS(PRODNO), den man natürlich auch nutzen könnte.

- Nachdem man zunächst also die benötigten Daten über ORDER_DETAILS D gefunden hat, muss man nun über den Join O.ORDNO = D.ORDNO auf ORDERS O kommen.

  Man muss also eine Reihenfolge für die Belegung der Tupelvariablen finden.

- Für den Schlüssel hat das DBMS schon einen Index über ORDERS(ORDNO) angelegt. Siehe aber nächste Folie.

Lösung der Aufgabe (3)

Auswertungsplan 1, Forts.:

- Optimal (aber schon sehr extrem) wäre ein Index über `ORDER(ORDNO, ORD_DATE, CUSTNO)`.

  `ORDNO` alleine ist schon ein Schlüssel, aber es ist nicht verboten, in einen Index noch weitere Attribute hineinzupacken. Man würde dadurch wieder den Zugriff auf die Tabelle sparen. Da man für `ORDNO` einen Wert hat und für `DATE` eine >=-Bedingung, müsste die Reihenfolge der Attribute im Index genau so sein (wenn man wirklich den zusätzlichen, sehr speziellen Index will).

- Zuletzt würde man den Index über `CUSTOMERS(CUSTNO)` verwenden, den das DBMS für den Schlüssel angelegt hat.

  Natürlich wäre auch hier ein Index günstiger, der schon alle benötigten Attribute enthält, und `CUSTNO` als erstes Attribut. Wenn man nicht ganz so weit gehen will, wäre auch ein Index über `CUSTOMERS(CUSTNO, CITY)` interessant. Man könnte so Kunden aus Pittsburgh filtern, bevor man auf die Tabelle zugreift.

Lösung der Aufgabe (4)

Auswertungsplan 2:

- Wenn es wenig Kunden in Pittsburgh gibt, aber viele Bestellungen für das Produkt 123, wäre es günstiger, mit den Kunden aus Pittsburg zu beginnen.
- Hierfür wäre ein Index über CUSTOMERS(CITY) nötig.

  Alternativ auch CUSTOMERS(CITY,CUSTNO), dann könnte man zuerst prüfen, ob der Kunde das Produkt bestellt hat, bevor man die Zeile aus CUSTOMERS läd.
- Anschließend braucht man einen der folgenden Indexe:
  - ORDERS(CUSTNO,ORD_DATE,ORDNO) (alle nötigen Attribute)
  - ORDERS(CUSTNO,ORD_DATE)

    Dies erlaubt die Datums-Prüfung vor dem Tabellenzugriff.
  - ORDERS(CUSTNO).

Lösung der Aufgabe (5)

Auswertungsplan 2, Forts.:

- Als letztes muss man auf `ORDER_DETAILS` zugreifen, hier wäre ein Index über `ORDER_DETAILS(ORDNO,PRODNO)` günstig (dann spart man sich den Tabellenzugriff).

Allgemein:

- Zur Entwicklung eines Auswertungsplans muss man also eine Reihenfolge finden, in der man auf die verschiedenen Tabellen zugreift.

  Genauer: Die Tupelvariablen mit Werten belegt.

- Die meisten DBMS betrachten nur lineare Join-Anordnungen, fangen also mit einer Tupelvariable an, und joinen dann jeweils eine dazu.

Literatur/Quellen

- Elmasri/Navathe:Fundamentals of Database Systems, 3rd Edition, 1999.
  Chapter 6, "Index Structures for Files"
  Section 16.3, "Physical Database Design in Relational Databases"
- Kemper/Eickler: Datenbanksysteme (in German), 4th Ed., Ch. 7, Oldenbourg, 2001.
- Ramakrishnan: Database Management Systems, Mc-Graw Hill, 1998, Chap. 4: "File Organizations and Indexes", Chap. 5: "Tree-Structured Indexing", Chap. 16: "Physical Database Design and Tuning".
- Oracle 8i Concepts, Release 2 (8.1.6), Oracle Corporation, 1999. 10-23 ff: "Indexes"
- Oracle 8i Administrator's Guide, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76956-01. Chapter 14: "Managing Indexes".
- Oracle 8i Designing and Tuning for Performance, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76992-01. Chapter 12: "Data Access Methods".
- Gray/Reuter: Transaction Processing, Morgan Kaufmann, 1993, Chapter 15.
- Wikipedia: Solid State Drive [https://en.wikipedia.org/wiki/Solid-state_drive]
- StorageReview: SSD vs. HDD. [https://www.storagereview.com/ssd_vs_hdd]
- [https://stackoverflow.com/questions/4087280/approximate-cost-to-access-various-caches-and-main-memory]