Datenbank-Programmierung

Kapitel 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

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

B^+-Bäume (1)

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

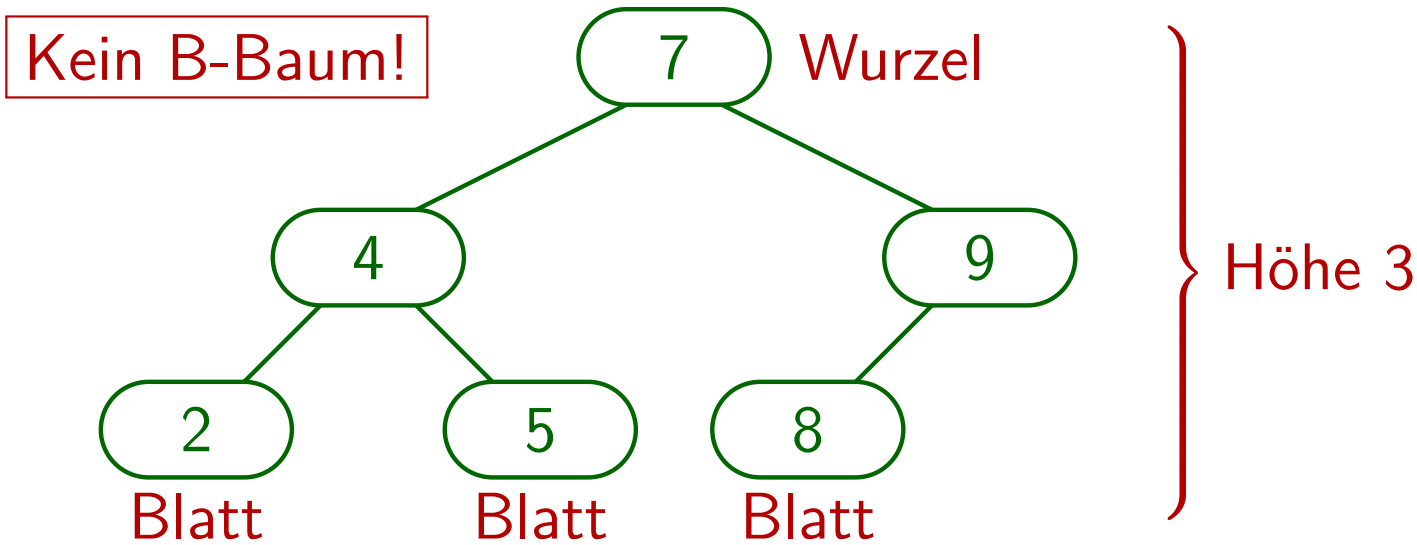
  Jedes moderne DBMS enthält eine Variante von B-Bäumen.
  Zusätzlich enthält es eventuell weitere spezialisierte Indexstrukturen (z.B. Hashverfahren, Bitmap-Indexe, R-Bäume).

- 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.

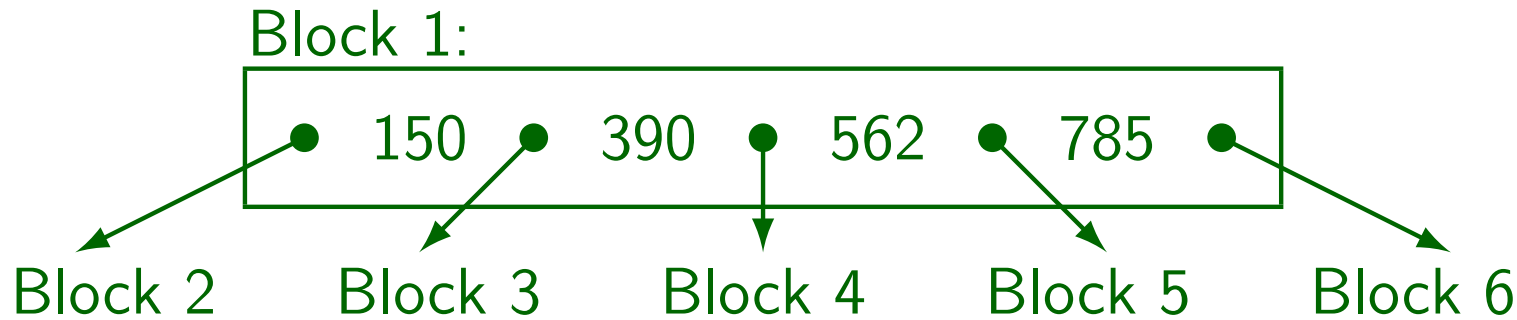
- Gewöhnliche binäre Suchbäume können zu einer linearen Liste degenerieren (bei Einfügung in sortierter Reihenfolge). Bei B-Bäumen kann das nicht geschehen, sie sind balanciert.

- 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:



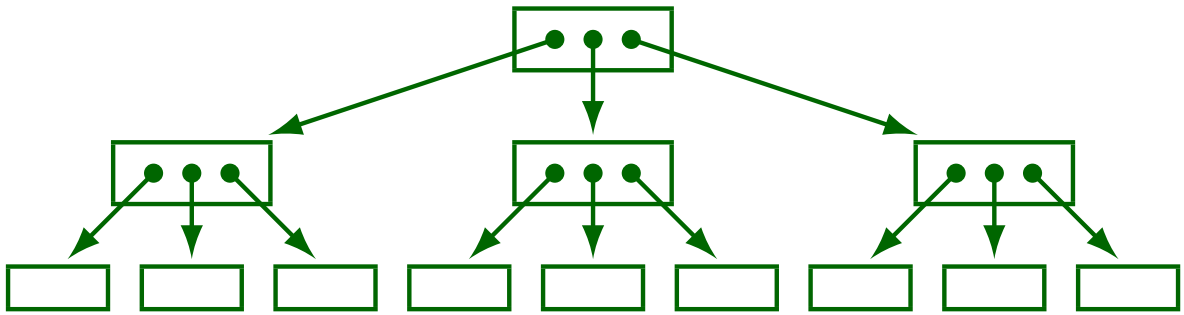
- Ist der gesuchte `CUSTNO`-Wert ≤ 150: Weiter bei Block 2.

  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)

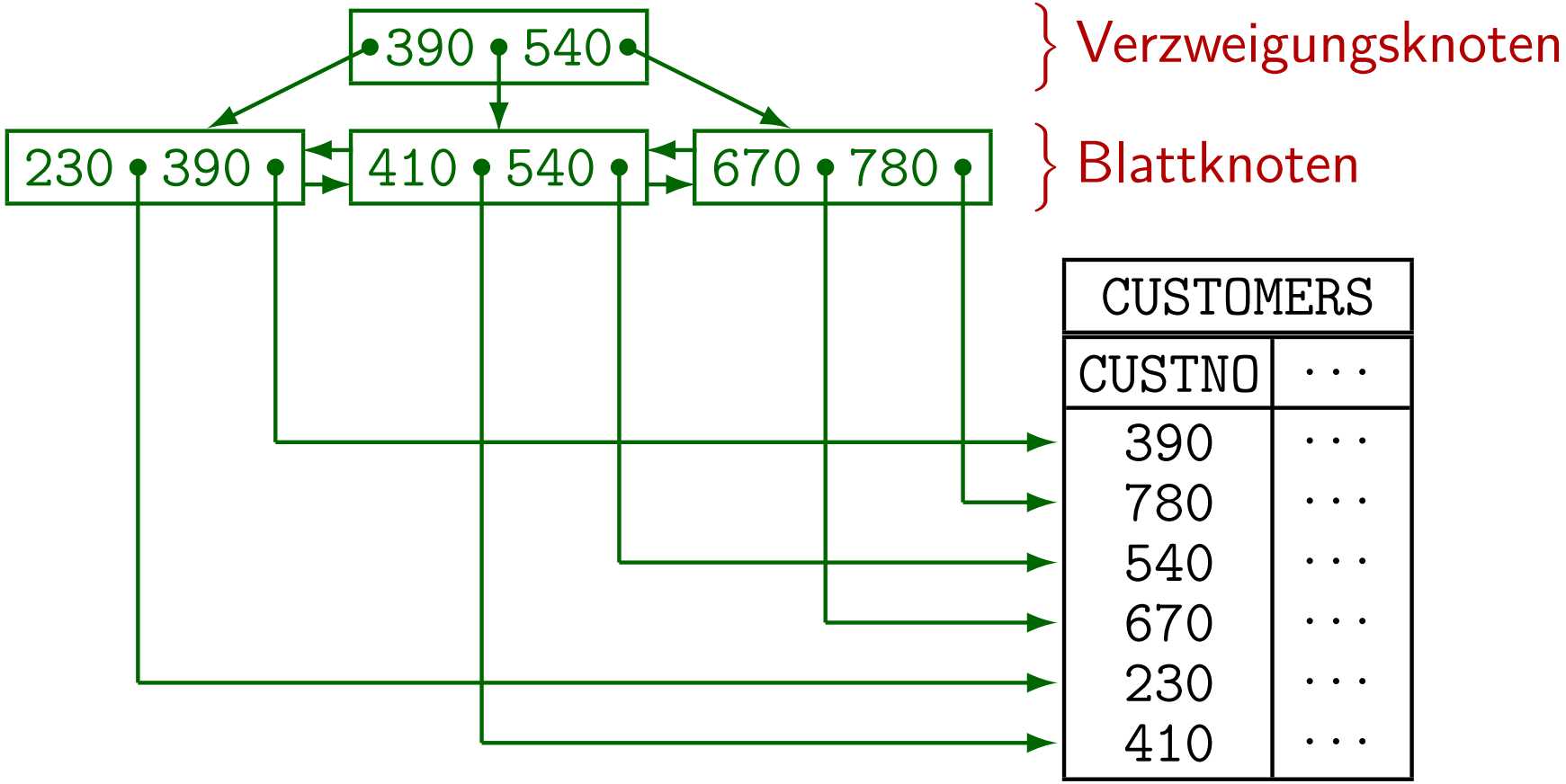
- Die referenzierten Blöcke (bis auf die unterste Ebene) haben die gleiche Struktur, so dass ein Baum entsteht:



- 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)


390 540
230 390
410 540
670 780
Verzweigungsknoten
Blattknoten
CUSTOMERS
CUSTNO
...
390
780
540
670
230
410

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.

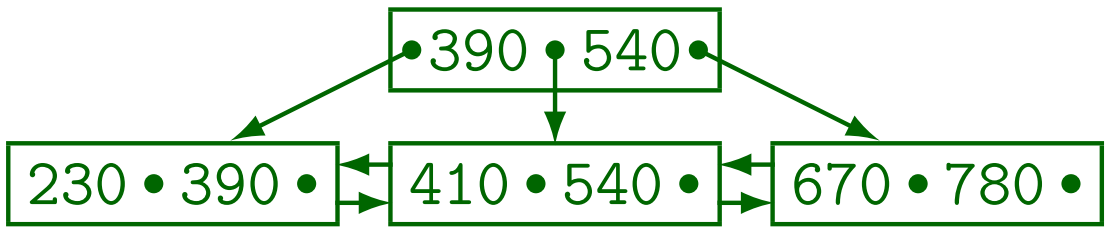
- Und so weiter. Wenn man am Ende die Wurzel aufspaltet, legt man darüber eine neue Wurzel an.

  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`.

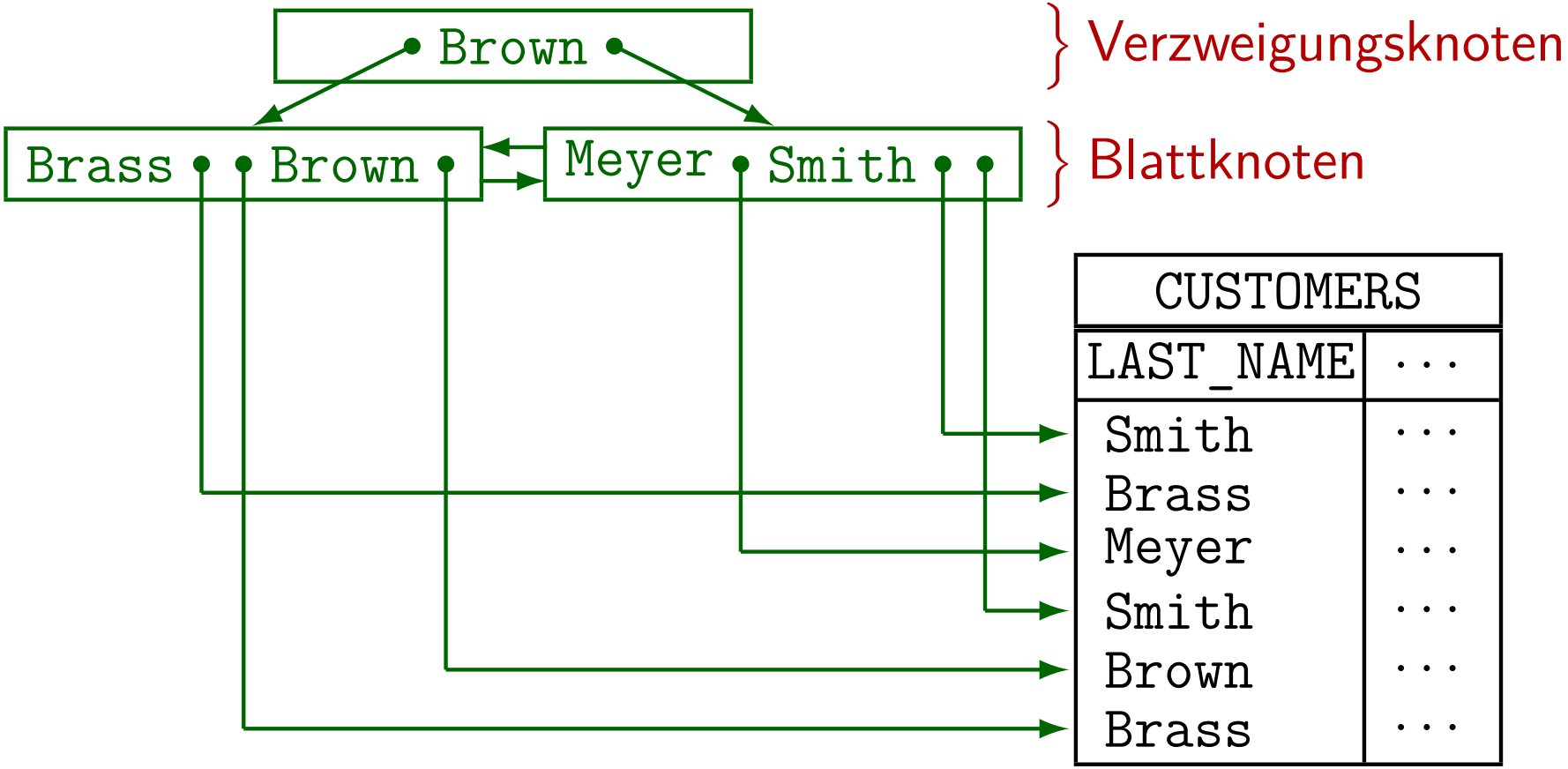
- In einem normalen Index (nicht "`UNIQUE`") können zu einem Datenwert mehrere Zeilen-Adressen gespeichert werden. Beispiel: Index über der Spalte `LAST_NAME`.

  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)


Brown
Brass
Brown
Meyer
Smith
Verzweigungsknoten
Blattknoten
CUSTOMERS
LAST_NAME
...
Smith
Brass
Meyer
Smith
Brown
Brass

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

Applications of Indexes (1)

- An index on the column A of the relation R is especially useful for equality conditions $A = c$ on tuple variables over the relation R.

  In relational algebra, this corresponds to the selection $\sigma_{A=c}(R)$.

- A B^+-tree index can also be used for `<`, `<=`, `>`, `>=` conditions (range queries, `LIKE` with known prefix).

- However, indexes are only useful when only a small percentage of the rows are retrieved via the index.

  If many rows satisfy the condition, a full table scan is faster: Sequential reading of blocks is faster than random accesses, and all rows in a block are together processed (in an index lookup only single rows).

Applications of Indexes (2)

- An index can be used even if there are other conditions besides the one supported by the index:

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

- If there is an index on the attribute `LAST_NAME`, the DBMS can use it to retrieve all rows that satisfy the first condition, and then simply check the second condition for each such row.

Applications of Indexes (3)

- A join can be evaluated using an index on one of the joined columns, e.g.:

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

- E.g. the DBMS can first find large invoices `I` and then use an index on `CUSTOMERS(CUSTNO)` to retrieve the customer data for each such invoice.

  Each single row `I` contains a specific customer number, for which one can use the index to find the matching row `C`.

Applications of Indexes (4)

- Some queries can even be answered entirely out of the index, without accessing the table itself: E.g. is there a customer with a given number?

  This is important for enforcing key/foreign key constraints.

- Sorting might sometimes profit from an index.

  This may speed up `ORDER BY`, `GROUP BY`, `DISTINCT`. The leaves of the B^+-tree contain all values in sorted sequence. However, this is really effective only in combination with a range query or if the query can be answered entirely out of the index. Otherwise accessing all rows of the table in random order might be slower than using e.g. the Mergesort algorithm (which does several *sequential* passes over the data).

Applications of Indexes (5)

- Indexes on combinations of two or more columns of a table will especially speed up queries that provide values for all these columns:

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

- If the index is a B-tree, it can also be used as an index for the first column (or in general a prefix).

  The first column of the combination is the main sorting criterion for the index. Here an index on `LAST_NAME, FIRST_NAME` would probably be more useful than the other way round. It could be used also as an index on `LAST_NAME` (it is slightly less efficient than a pure index).

Applications of Indexes (6)

Exercise:

- Consider the following query:

```
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-02'
AND    C.CUSTNO = O.CUSTNO AND O.ORDNO = D.ORDNO
```

- Which indexes could be useful for evaluating the query? Sketch two different evaluation possibilities.

  What information would the DBMS need for deciding which evaluation method is better?

Disadvantages of Indexes

- Indexes use disk space.

  The example index needs 30-50% of the space of the table.

- Queries become faster, but updates become slower because the indexes must be updated, too.

- Query optimization becomes slower: More alternatives for evaluating the query must be considered.

  But it might be possible to amortize the cost of query optimization over many executions of the same query.

- A full table scan can be much faster if nearly all blocks must be accessed anyway.

Hints for Selecting Indexes

- Check whether the query optimizer really uses the indexes.

  DBMS usually offer the possibility to see the result of query optimization (query evaluation plans, internal query programs).

- It makes no sense to declare indexes on small tables.

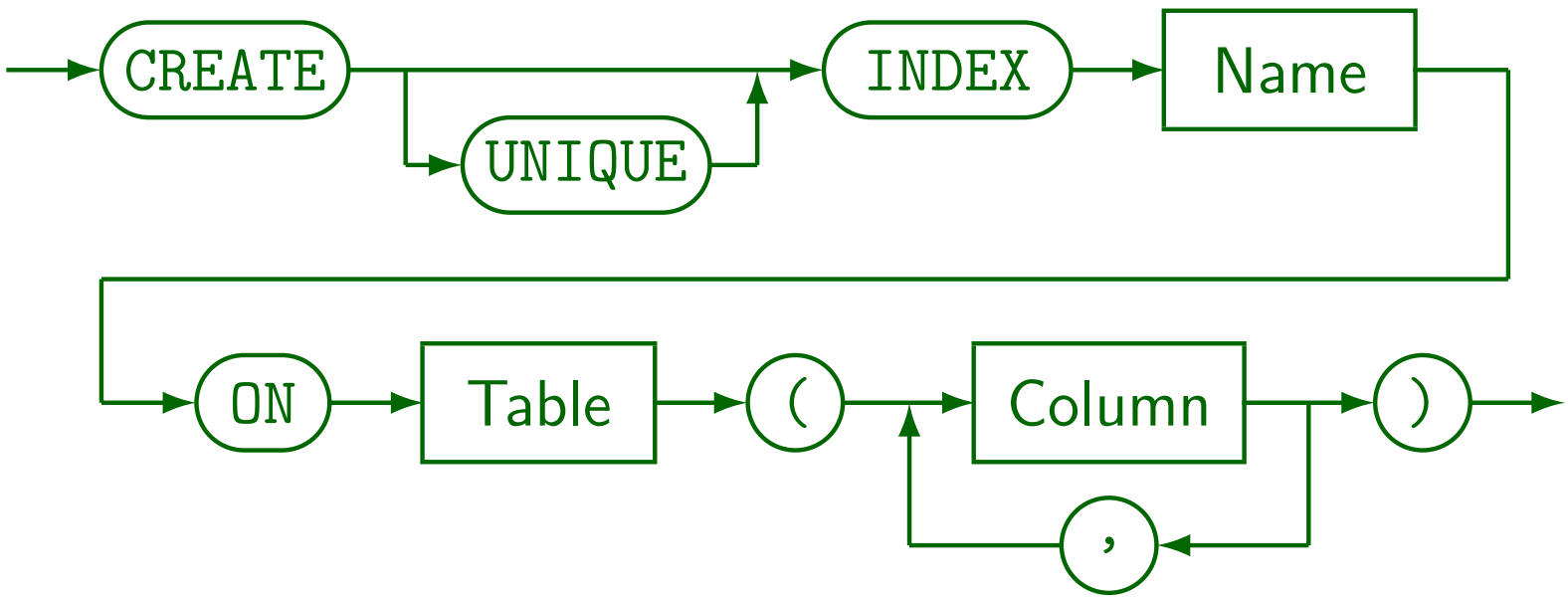
  Except the indexes needed to enforce keys, see below.

- Do not declare too many indexes on tables that are often updated.

- Normally, an index is useful only if the condition is satisfied only by a small percentage of the rows.

Indexes in SQL (1)

- SQL command for creating an index (not SQL-92):



- E.g.: `CREATE INDEX CUSTIND1 ON CUSTOMERS(CITY)`

Indexes in SQL (2)

- The `CREATE INDEX` command is not contained in the SQL standards, but it is supported by most DBMS.

  The SQL standards do not treat physical storage concepts.

- `UNIQUE` means that for every value for the index column there is only one tuple.

  I.e. the index column is a key. Older SQL versions had no key declarations, so a unique indexes were used.

- Most DBMS automatically create a unique index for key constraints (`PRIMARY KEY` and `UNIQUE`).

  Thus the keys are enforced as before by means of unique indexes.
  It would be wrong to explicitly create another index for a key.

Indexes in SQL (3)

- Creating an index on a large table can take quite some time, need a lot of temporary disk space, and lock the entire table.

  One should not experiment with indexes during the main production hours. On some systems (e.g. DB2, but not Oracle), it might be necessary to rebind (re-optimize) application programs after relevant indexes were changed.

- Command for deleting indexes:

- E.g.: `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 II B" 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

Physical Design Issues (1)

- The index selection, i.e. deciding which indexes to create, is the classical physical design issue. But there is much more to do.
- Normally the table itself is stored as a "heap file" (without any specific order), but even for that there are various storage parameters.

  E.g. how much space in a block should be kept free for updates that make a row longer (because of physical pointers from indexes, it is not good to "migrate" a row to a different block). One should also ensure that the table is stored sequentially in one chunk of disk space (or only a few big pieces), not scattered over the entire disk (in many small pieces). Thus, one must define a storage size.

Physical Design Issues (2)

- Database management systems offer other access structures besides B-trees, e.g. hash methods.

  Oracle allows clustering tables together (good for joins). Instead of having separate indexes and table, it might also be possible to store the table data directly in the index (index-organized table).

- The distribution of tables etc. among different disks is also important for performance.

  Tables can be split into several parts (horizontal/vertical partitioning).

- In exceptional cases, tables could be denormalized.

  One pays for the performance gain with programmer time and less flexibility for changes.

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]