Dateien:
- Blockgröße muss Vielfaches der Sektorgröße sein
- Zugriffszeit auf einen Block (Seite): seek time (Positionierung Lese/Schreib-Kopf) + latency time (warte auf Vorbeirotieren des gesuchten Blocks) + transfer time (Transport der Daten)
- Sequentielles Lesen = „Next Block“ Konzept
- Dateistrukturen:
  - Heap: für Scans oder wenn gesuchter Bereich nahezu die ganze Datei umfasst
  - Hash: für Lookups
  - Sortiert: für Range Queries
- Kosten für Dateioperationen: B…Datenseitenanzahl, R…Sätze pro Seite, D…Zeit zum Transfer einer Seite

	Heap Datei	Sortierte Datei	Hash Datei
Scan	BD	BD	1.25 BD
Lookup	BD	$D*\log_2 B$	D
Range Query	BD	$D*(\log_2 B$ + Anzahl matchender Seiten)	1.25 BD
Insert	2D	Search + BD	2D
Delete	Search + D	Search + BD	2D

Indexstrukturen:
- Alternativen für Einträge im Index:
  1. ganzer Datensatz mit Schlüsselwert k
  2. <k, ID des Datensatzes mit dem Wert des Suchschlüssels k>
  3. <k, Liste von Datensatz-IDs mit Suchschlüssel k>
- dünnbesetzte Indexe müssen geclustert sein, d.h. in Relation muss Sortierung nach dem Suchschlüssel existieren
- dünnbesetzt: ein Indexeintrag pro Seite, bei Alternative 3. Liste gleicher Suchschlüssel-Werte, wenn über mehrere Seiten erstreckend

Hash-Indexe:
- statisches Hashing: feste Anzahl von Primärseiten, Überlaufseiten wenn voll
- dynamisches Hashing:
  - für alle Buckets gilt: globale Tiefe >= lokale Tiefe
  - Verzeichnis von Zeigern auf Buckets, verdoppeln wenn nötig
  - Globale Tiefe: max. Bitanzahl um zu entscheiden wohin ein Eintrag gehört
  - Lokale Tiefe: tatsächliche Bitanzahl um zu entscheiden ob ein Eintrag in diesen Bucket gehört
  - Beim Einfügen: Bucket darf nicht voll sein, sonst:
    - Wenn lokale Tiefe < globale Tiefe, dann splitte nur Bucket, erhöhe lokale Tiefen der beiden Split-Buckets
    - Wenn lokale Tiefe = globale Tiefe, dann splitte Bucket, erhöhe lokale Tiefen der beiden Split-Buckets UND verdopple Zeigerverzeichnis, erhöhe globale Tiefe um 1
  - Beim Löschen: wenn Bucket dadurch leer wird:
    - Wenn Split-Bucket existiert, dann vereinige damit; wenn möglich verringere globale Tiefe (halbiere Zeigerverzeichnis hierbei)
    - Wenn kein Split-Bucket existiert, dann belasse Bucket leer

B^+ Baum Index:
- Wurzelknoten darf nicht leer sein, hat mind. 2 Söhne
- Ordnung d: jeder Zwischenknoten enthält d <= x <= 2d Einträge x und d+1 <= y <= 2d+1 Söhne y
- Einfügen: wenn Platz nicht reicht – splitte Knoten (rekursiv wenn nötig):
  - Blattseite: kopiere mittleren Schlüssel hoch
  - Indexseite: ziehe mittleren Schlüssel hoch
- Löschen: wenn Unterlauf entsteht:
  - Versuche von Bruder zu borgen, kopiere (Blattseite) bzw. ziehe (Indexseite) mittleren Schlüssel hoch
  - Wenn Borgen nicht möglich: mische mit Bruderknoten
- Ebenen des Baumes = Baumhöhe + 1 (Wurzel zählt nicht zu Höhe dazu)

- Suchkosten: $\log_d N$ (d…Baumordnung, N…Anzahl an Dateneinträgen)
- Baumhöhe $h = \log_{2d+1} N$ (d…Baumordnung, N…Anzahl Indexseiten = (Anzahl Dateneinträge) / (Einträge pro Blattseite))
- Kapazität einer Indexseite >= 2d * Schlüsselgröße + (2d+1) * Zeigergröße
- Einbeziehung eines Füllfaktors:
  - Einträge pro Blattseite = max. Blattseiteneinträge * Füllfaktor
  - Anzahl Zeiger pro Indexseite = max. Anzahl an Indexseitenzeigern * Füllfaktor
  - $h = \log_{(2d+1)*\text{Füllfaktor}}$ [(Anzahl Dateneinträge) / (Einträge pro Blattseite * Füllfaktor)]

Kostenformeln für Indexe:
- Beispiel: Heap-File, unsortiert, dirchbesetzte Sekundär-Indexe:
  D…Anzahl Datensätze
  S…Anzahl Seiten
  h…B^+ Baum Höhe
  M…Anzahl Einträge auf einer Indexseite
  c…Belegungsfaktor für Hash-Index
  - Kosten beim sequentiellem Scan: S (für alle Anfragen)
  - Kosten bei Hash-Index:
    - Lookup: 1+1
    - Bereichsanfrage mit N Tupeln als Ergebnis: N+N
    - Lesen aller Tupel: D/(M*c) + D
  - B^+ Baum Index:
    - Lookup: h+1
    - Bereichsanfrage nach N Tupeln als Ergebnis: h+N/M+N
    - Lesen aller Tupel: h+D/M+D

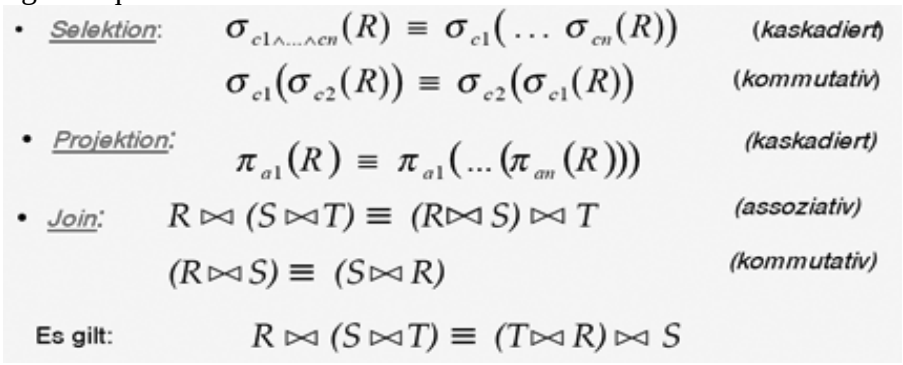
Joinkosten:
M…Seiten mit Tupeln in Relation R (äußere Relation) p_R…Tupel pro Seite in R
N…Seiten mit Tupeln in S (innere Relation) p_S…Tupel pro Seite in S
- Äußere Relation sollte die kleinere sein
- Einfacher Nested Loop Join:
  Kosten: $M + p_R*M*N$ Pufferbedarf: min. 3 Seiten
- Page-Oriented Nested Loop Join:
  Kosten: M + M*N Pufferbedarf: min. 3 Seiten
- Block Nested Loop Join:
  Blockgröße = Anzahl Pufferseiten – 2
  Blöcke äußere Relation = ceil(Anzahl Seiten äußere Relation / Blockgröße)
  Kosten: M + Anzahl Blöcke * N Pufferbedarf: für Block benötigte Seiten + 2
- Index Nested Loop Join:
  Relation mit Index auf Join-Spalte ist zur inneren Relation zu machen
  Kosten: $M+M*p_R*$(Suchkosten für matchende Tupel)
  Suchkosten (B^+ Baum) = Zugriff auf Blattseite + Lesekosten zugehöriger Datensätze
  Lesekosten abhängig von Clusterung: geclustert → 1 I/O, ungeclustert → bis 1 I/O pro matchendem Tupel
- Sort Merge Join:
  Kosten: M*log M + N*log N + M + N
- Minimal mögliche Join-Kosten:
  Kosten: M+N Pufferbedarf: M+1+1

Selektionskosten:
- Einfacher Scan: Größe der Relation * Reduktionsfaktor (RF)
- Mit Index auf Selektionsattribut:
  - Hash-Index:
    - Lookup: 1+1
    - Range Query: Anzahl der Tupel * RF

- Geclusterter B^+ Baum Index:
  - Range Query: Indexscan + Lesekosten = Seitenzahl * (Größe Dateneintrag im Index / Größe Datensatz) * RF + Seitenzahl * Tupel pro Seite * RF
  - Mehrere Selektionsbedingungen (x an der Zahl) und geclusterter Index darauf: Seitenzahl * x*(Größe Dateneintrag im Index / Größe Datensatz) * RF^x + Seitenzahl * Tupel pro Seite * RF^x
- Ungeclusterter B^+ Baum Index: wie geclustert, aber multipliziere mit Anzahl der Tupel pro Seite

Anfrageoptimierung:

- Clustering nur sinnvoll, wenn viele Tupel auf eine Seite passen
- Algebraäquivalenzen:



- Projektion ist kommutativ mit Selektion, die nur Projektions-Attribute benutzt
- Selektion zwischen Attributen der zwei Argumente eines Kreuzprodukts macht daraus einen Join

DB-Tuning:

- Schlüsselkandidaten: untersuche FDs
  - Attribute, die nur rechts stehen gehören zu keinem Sk
  - Attribute, die nur links stehen, gehören zu ejdem Sk
  - Ermittlung von Sk über transitive Hülle: Nimm Potenzmenge aller in den FDs vorkommenden Attribute → für jedes Element daraus: wenn durch Hüllenerzeugung alle Attribute abgeleitet werden können, ist Element Sk
  - Ein Sk darf nicht Element anderer Sk sein
- Normalformen:
  - 1NF: atomare Werte, keine zusammengesetzten Attribute
  - 2NF: jedes Nicht-Primärattribut (gehört zu keinem Sk) ist voll funktional abhängig von jedem Sk; d.h. Nicht-Primärattribute dürfen nicht aus Teilen eines Sk hergeleitet werden können
  - 3NF: jedes Nicht-Primärattribut ist von keinem Sk transitiv abhängig (die rechten Seiten aller FDs müssen unterschiedlich sein)
  - BCNF: jeder Determinant (linke Seite einer FD) ist Schlüsselkandidat
- Überführung in Normalformen:
  - In 1NF: für zusammengesetzte Attribute mit Primärschlüssel der Vaterrelation eigene Relation erzeugen
  - In 2NF: Partiell abhängige Attribute in eigener Relation mit einem Vater-Primärattribut zusammenfassen
  - In 3NF: Aufbrechen in 2 Relationen, abhängig von der Semantik der Daten
  - Aufbrechen der Relationen mit verletzten FDs
- Tuning von Queries:
  - Vermeide DISTINCT (wenn Datensätze eindeutig), GROUP BY/HAVING (wenn mit WHERE lösbar), ORDER BY
  - Vermeide: NULL-Werte, arithmetische Ausdrücke (i.a. weniger Datensätze zu betrachten), String-Ausdrücke, OR-Bedingungen (verhindert vielleicht Index-Nutzung, UNION besser)
  - Vermeide Zwischenrelationen
  - Versuche BETWEEN statt … AND … zu verwenden
  - Beispiel zur Vermeidung von Selektion: R.sid ist Fremdschlüssel auf S
    Statt: "SELECT S.sid FROM Sailors S, Reserves R WHERE S.sid=R.sid"
    Besser: "SELECT sid FROM Reserves WHERE sid IS NOT NULL"

Hierarchische Sperren:

- Steige Hierarchie hinab zu dem Objekt, auf das der Zugriff erfolgen soll
- Setze Intention Locks (ILs) auf jeder Ebene, auf der man vorbeikommt und in der man nicht auf Objekte direkt zugreifen will
- Bei Löschen oder Einfügen: setze X-Sperre auf nächst höherer Ebene (z.B. auf Seite bei Löschen eines Tupels)
- SIX-Sperre: gewährt Lesezugriff auf alle Knoten der Ebene mit Absicht darunter liegende Objekte zu ändern (X, IX, SIX Sperren auf diese möglich)
- Zu Vereinbarende Aktionen:
  - Mit keiner: alles außer SIX
  - Mit IS: keine, IS, IX, S, SIX
  - Mit IX: keine, IS, IX
  - Mit SIX: IS
  - Mit S: keine, IS, S
  - Mit X: keine
  - → wenn nicht vereinbar, muss TA warten

Recovery:
- Analyse-Phase:
  - Suche letzten Checkpoint → hier beginnt die Analyse
  - Erstelle DPT (dirty page table – alle zum Crash schmutzigen Seiten) und TT (transaction table – alle zum Crash aktiven TAs) mit zugehörigen LSNs → dabei darf ein Objekt in DPT bzw. eine TA in TT jeweils höchstens einmal vorkommen
- REDO: startet bei kleinster LSN in DPT; wiederhole Aktionen aus DPT, wenn Seite noch nicht auf Disk geschrieben
- UNDO: gehe von Crash-Zeitpunkt zurück durch TT, setze dabei bei jeder in TT vorhandenen TA auf nächstniedrigere LSN dieser TA

Übung „Speicherverwaltung – Platten und Dateien“

5.) Berechnungen II

Sektorgröße 512 Bytes, 2000 Spuren pro Oberfläche, 10 Oberflächen, 50 Sektoren pro Spur, seek time durchschnittlich 10ms, Blockgröße 1024 Bytes; Speicherung einer Datei mit 100.000 Sätzen, 1 Satz = 100 Bytes

a) Sätze pro Block: 1024/100 = 10 Sätze, 24 Bytes verschenkt

b) Blöcke für komplette Datei: 10.000 Blöcke
benötigte Oberflächen: 25 Blöcke pro Spur → 400 Spuren für gesamte Datei → alle Oberflächen

c) Wieviele Sätze auf gesamter Platte speicherbar?: 10 Oberflächen * 2000 Spuren pro Oberfläche * 25 Blöcke pro Spur * 10 Sätze pro Block = 5.000.000 Sätze

d) Antwort: Seite 26, da eine Spur 25 Blöcke aufnehmen kann

e) Zeit zum Lesen der gesamten Datei: 100.000 Sätze = 10.000 Blöcke = 400 Spuren zu lesen
90 U/s → 1/90s = 0.011s für Transfer einer Spur → Gesamttransferzeit: 400 * 1/90s = 4,44s
addiere Gesamt-„seek time“: t_{sges} = 0.01s * 400 Spuren / 10 Oberflächen = 0,4s
→ Gesamtzeit = 0,4s + 4,44s = 4,84s
wenn parallel lesen möglich: Transferzeit wird durch 10 Oberflächen dividiert
→ $Gesamtzeit_{par}$ = 0,4s + 4,44s/10 = 0,844s

f) Zeit zum Lesen der gesamten Datei in beliebiger Reihenfolge (nicht sequentiell, kein Next Block Konzept):
10.000 Blöcke zu lesen, für jeden Block: seek time + latency time + transfer time
→ Gesamtzeit = 10.000 Blöcke * (10ms + 11ms/2 + 11ms/[25 Blöcke pro Spur]) = 159400ms = 159,4s

Übung „Dateiorganisation und Indexstrukturen“

4.) Dateneinträge in Indexen:

sortierte Datei, jede Seite kann 3 Tupel aufnehmen, Tupel identifiziert durch <pageID, slotID>

Gegeben:

…	age	gpa
	11	1.8
	12	2.0

a) dicht auf age mit (2): 11,<1,1>; 12,<1,2>; 18,<1,3>; 19,<2,1>; 19,<2,2>
b) dirch auf age mit (3): 11,<1,1>; 12,<1,2>; 18,<1,3>; 19,<2,1>,<2,2>
c) dünn auf age mit (2): 11,<1,1>; 19,<2,1>
d) dünn auf age mit (3): wie c)

	18	3.4	e) dicht auf gpa mit (2): 1.8,<1,1>; 2.0,<1,2>; 3.2,<2,1>; 3.4,<1,3>; 3.8,<2,2>
	19	3.2	f) dicht auf gpa mit (3): wie e)
	19	3.8	g) und h) nicht konstruierbar, da nicht nach gpa sortiert, aber Clustering für dünn nötig

Übung „Hashbasierte Indexverfahren“

2.) I/O Kosten für Anfrageverarbeitung

Relation R: 1.000.000 Datensätze, 1 Seite = 10 Datensätze; Heap-File, dichbesetzte Sekundärindexe

R beliebig sortiert, a ist Primärschlüssel, a hat Werte im Bereich [0..999.999]

h…Baumhöhe, M…Anzahl an Einträgen auf einer Indexseite, c…Belegungsfaktor im Hash-Index

Aufgabe	Heap-File, seq.	B^+ Baum auf R.a	Hash-Index auf R.a
Finde alle Tupel in R	100.000	$h+10^6/M+10^6$	$10^6/(M*c)+10^6$
Finde alle Tupel in R mit a<50	100.000	h+50/M+50	50+50
Finde alle Tupel in R mit a=50	100.000	h+1	1+1
Finde alle Tupel in R mit a>50 und a<100	100.000	h+49/M+49	49+49

Übung „Baumbasierte Indexverfahren“

2.) dichter B^+ Baum, Alternative (2) auf Heap-Datei; 20.000 Datensätze, Schlüsselfeld = 40 Bytes – Schlüsselkandidat, 1 Zeiger auf Datensatz = 10 Bytes, 1 Seite = 1.000 Bytes

a) Ebenen des resultierenden Baumes?: Index umfasst 20.000 Einträge
   Kapazität einer Indexseite = 1000 >= 2d*Schlüsselgröße + (2d+1)*Zeigergröße → 80d+20d+10 <= 1000
   → d <= 9.9 → d=9 ➔ Platz für 18 Schlüssel und 19 Zeiger in einem Index-Knoten
   Kapazität einer Blattseite: 1 Eintrag = 10+40 Bytes → 1000 Bytes / 50 Bytes pro Eintrag = 20 Einträge
   ➔ Anzahl Ebenen = h+1 = ($\log_{2d+1}$ N) + 1 mit N…Anzahl Blattseiten = Anzahl Datensätze / Einträge pro Seite
   h = $\log_{2d+1}$ N = $\log_{19}$ (20000/20) <= 3 ➔ 4 Ebenen werden benötigt

b) Knotenanzahl auf jeder Ebene:
   Ebene 4: 1000 Blattknoten
   Ebene 3: 1000 Blattknoten / 19 Söhne = 53 Knoten
   Ebene 2: 53 Knoten / 19 Söhne = 3 Knoten
   Ebene 1: 1 Knoten (Wurzel)

c) Ebenenanzahl wenn Schlüssellänge = 10 Bytes:
   Kapazität einer Indexseite: 1000 >= 2d*10 + (2d+1)*10 → d <= 24.75 → d=24
   Kapazität einer Blattseite: 1 Eintrag = 10+10 Bytes → 1000 Bytes / 20 Bytes pro Eintrag = 50 Einträge
   ➔ Anzahl Ebenen = h+1 = $\log_{49}$ (20000/50) <= 3 ➔ 3 Ebenen werden benötigt

d) Ebenenanzahl, wenn Seiten zu 70% belegt: → Füllfaktor f=0.7
   Erst 19 Söhne pro Indexeintrag, jetzt 19*0.7 = 13.3 → 13 Söhne pro Indexeintrag
   Erst 1000 Blattseiten benötigt, jetzt 1000/0.7 = 10000/7 Blattseiten
   ➔ h+1 = log13 (10000/7) + 1 = 3.83 → 4 Ebenen werden benötigt (wie auch in a)

Übung „Anfrageverarbeitung“

1.) Join unter Bedingung R.a=S.b

R: 10.000 Tupel, 1 Seite = 10 Tupel

S: 2.000 Tupel, 1 Seite = 10 Tupel, b ist Primärschlüssel

Heap-Files, keine Indexe, 52 Pufferseiten

→ nimm kleinere Relation als äußere, als S; damit: M=2000, N=10000, p_S=10, p_R=10

a) Page-oriented Nested Loop Join:
   Kosten: M + M*N = 200 + 200*1000 = 200.200
   Pufferseiten: minimal 3 (1 für R, 1 für S, 1 für Ergebnis)

b) Block Nested Loop Join:
   B = 52 Pufferseiten verfügbar → 50 Seiten für Block
   Kosten: M + (M/(B-2) * N) = 200 + (4*1000) = 4200
   Pufferseiten: alle Pufferseiten belegt (50 für Block aus S, 1 für R, 1 für Ergebnis)

c) Niedrigstmögliche Join-Kosten:
   Kosten: M+N = 1200, da jeder Relation mind. einmal gelesen werden muss

Pufferseiten: 200+1+1 = 202 (200 für S, 1 für R, 1 für Ergebnis)

d) Von Join produzierte Tupel:
10.000 Tupel unter der Annahme, dass jedes Tupel in R einen Join-Partner in S hat
→ ergibt 1000 Seite jeweils für R und S, Join verbindet beide, somit werden für das Ergebnis 1000+1000 = 2000 Tupel benötigt

2.) gleiche Bedingungen wie in 1.)
Erreichen eines Blattes in R-Index: 3 I/O; Erreichen eines Blattes in S-Index: 2 I/O
Annahme: zu jedem Tupel in S werden 5 Tupel in R gefunden, welche die Join-Bedingung erfüllen

a) 52 Pufferseiten, ungeclusterte Indexe – Index Nested Loop Join vs. Block Nested Loop Join:
Index Nested Loop Join:
Kosten = M + M*p_S*Suchkosten für matchende Tupel
Suchkosten = Zugriff auf Blattseite + Lesekosten des zugehörigen Datensatzes
➔ Kosten = 200 + 200*10*(3+5) = 16.200
Block Nested Loop Join:
B= 52 Pufferseiten verfügbar
➔ Kosten = M+N*(M/(B-2)) = 200+1000*(200/50) = 4.200
→ Block Nested Loop Join auf jeden Fall günstiger

b) 5 Pufferseiten – Änderung im Ergebnis von a):
Index Nested Loop Join: keine Änderung, Kosten=16.200
Block Nested Loop Join: B=5
Kosten = 200+1000*200/(5-2) = 67.200
→ Index Nested Loop Join auf einmal sehr viel besser

c) S: 10 Tupel – Änderung im Ergebnis von a):
Alle 10 Tupel passen auf 1 Seite
Index Nested Loop Join:
Kosten = 1+1*10*(3+1000) = 10.031 (1000, da bei 2.000 Tupeln 5 Join-Partner, somit bei 10 Tupeln 1000 Join-Partner)
Block Nested Loop Join:
B = 3 (es werden 3 Seiten benötigt: 1 für S, 1 für R, 1 für Ergebnis)
Kosten = 1 + 1000*(1/1) = 1001
→ Block Nested Loop Join besser

3.) geclusterte Indexe auf R.a und S.b, 52 Pufferseiten; Rest siehe Aufgabe 1 bzw. 2
Index Nested Loop Join vs. Block Nested Loop Join:
Index Nested Loop Join:
Kosten = 200 + 200*10*(3+1) = 8.200 (+1, da geclustert und somit nur 1 Seite gelesen werden muss)
Block Nested Loop Join:
Kosten bleiben wie bei 2a) bei 4.200, damit besser als Index Nested Loop

Übung „Anfrageoptimierung“

1.) Relation: Employees(*eid*:integer, *ename*:string, *sal*:integer, *title*:string, *age*:integer)
verfügbare Indexe: Hash-Index auf *eid*, B^+ Baum auf *sal*, Hash-Index auf *age*, geclusterter B^+ Baum auf <*age,sal*>
1 Datensatz = 100 Bytes, 1 Indexeintrag = 20 Bytes, Relation umfasst 10.000 Seiten, 1 Seite = 20 Datensätze, RF=0.1

a) sal > 100:
File Scan: Kosten = 10.000
B^+ Baum auf sal:
Kosten = h + 10.000*(20 Bytes / 100 Bytes)*0.1 + 10.000*20 Tupel pro Seite*0.1 = 20200
Gilt für ungeclusterten B^+ Baum
→ File Scan am günstigsten

b) age = 25:
File Scan: Kosten = 10.000
Hash-Index auf age:
Kosten > 2 * (10.000 Seiten * 20 Tupel pro Seite * 0.1) = 40.000
(2*, da für jedes Tupel ein IndexScan und ein Scan der jeweiligen Seite erforderlich ist)
B^+ Baum auf <age,sal>, geclustert:

Kosten = h + 10.000*(2*20/100)*0.1 + 10.000*0.1 = 1.402 (bei h=2)
(2*20/100, da 2 Attribute im Index vorhanden sind – 1 Eintrag somit doppelt so lang)
→ B^+ Baum auf <age,sal> am günstigsten

c) eid = 1000:
eid ist Schlüssel, somit bei Hash-Index auf eid nur ca. 2 I/Os nötig

d) sal > 200 und age = 20:
geclusterter B^+ Baum auf <age,sal>:
Kosten = h + 10.000*(2*20/100)*0.1*0.1 + 10.000*0.1*0.1 = 142 (bei h=2)

3.) Relation Manager(*ename, title, dname, address*), ename ist Schlüsselkandidat, Relation hat 10.000 Seiten
Anfrage: SELECT title, ename, FROM Manager WHERE title='CFO' → RF=0.1

a) geclusterter B^+ Baum auf *title*:
Indexgröße = 2.500 Seiten, da nur ein Attribut indexiert
Kosten = h + 10.000*0.1 + 2.500*0.1 = 1.252 (bei h=2)

b) ungeclusterter B^+ Baum auf *title*:
genauer Wert ist abhängig von der Anzahl Tupel pro Seite T
Kosten = h + 10.000*T*0.1 + 2.500*0.1

c) geclusterter B^+ Baum auf *ename*:
Sortierung nach ename bringt nichts, Filescan günstiger

d) geclusterter B^+ Baum auf *address*:
Sortierung nach address bringt nichts, Filescan günstiger

e) geclusterter B^+ Baum auf <*ename,title*>:
Kosten = 2.500 * 2 = 5000 Seiten für Indexscan, da 2 Attribute indexiert → IndexOnly!
Reduktionsfaktor hier nicht anwendbar, da primär nach ename sortiert ist

Übung „Physischer Datenbankentwurf und Tuning"

1.) Welcher Index ist am besten?
Relationen: Emp(eid, ename, sal, age, did), Dept(did, budget, floor, mgr_eid)
sal = [10.000…100.000], age = [20…80], floor = [1…10], budget = [10.000…1.000.000]
im Durchschnitt 5 Angestellte pro Abteilung
Anfrage 1: Gebe Namen, Alter und Gehalt für alle Angestellten aus.
In Frage kommen e) ein File Scan und a) geclusterter Hash-Index auf <ename,age,sal>, b) ungeclusterter Hash-Index auf <ename,age,sal> oder c) geclusterter B^+ Baum auf <ename,age,sal> für IndexOnly-Prinzip, davon aber nur b), da Clustering hier nicht sinnvoll ist. Da Hash-Index und gesamte Relation gescannt werden muss, ist File Scan vorzuziehen.
File Scan Kosten = N (N…Anzahl der Seiten)
Ungeclusterter Hash Index auf <ename,age,sal>: 1.2 * N * 3/5 (da 3 von 5 Attributen im Index stehen) < N
Anfrage 2: Finde die did der Dept mit floor=10 und budget<15.000
Reduktionsfaktoren: für floor - RF_{floor} = 0.1, für budget – RF_{budget} = (15.000-10.000)/(1.000.000-10.000) = 5/990
c) Geclusterter dichter B^+ Baum auf <floor,budget>: Kosten = h + N*2/4*0.1*5/990 + N*0.1*5/990
d) Geclusterter dünner B^+ Baum auf budget: Kosten = h + N*1/4*5/990 + N*5/990

3.) Relation G(S,G,O,P) mit S…Sektion, G…Gehalt, O…Standort, P…Präsident
Anfrage: „Wer war Präsident von Sektion X, als sich diese am Ort Y befand?" → ohne Join zu berechnen

a) FDs: S → G, SO → P, P → S

b) X^+ = U, U…Menge aller Attribute, X^+…transitive Hülle
Attribute, die nur rechts stehen gehören zu keinem Sk: G
Attribute, die nur links stehen, gehören zu jedem Sk: O
1-elementige Mengen:
nur O kommt in Frage: $(O)^0 = O = D^+ != U$ → kein Sk
2-elementige Mengen:
$(SO)^0$ = SO, $(SO)^1$ = SOPG = $(SO)^+$ = U → SO ist Sk
$(PO)^0$ = PO, $(PO)^1$ = SOP, $(PO)^2$ = SOPG = $(PO)^+$ = U → PO ist Sk
GO kommt nicht in Frage, da G zu keinem Sk gehört
3-elementige Mengen:
SOP als einziger Kandidat nicht möglich → kein Sk darf in einem anderen Sk enthalten sein

c) In 1NF?: ja, da alle Attribute atomar
In 2NF?: nein durch S→G, da Nicht-Schlüssel-Attribut G partiell abhängig ist von Sk SO

d) „gutes“ Schema → Normalisierung
Überführung in 2NF:
Lösen der partiellen Abhängigkeit durch Aufspalten von G in G1 und G2:
G1(S,O,P), G2(S,G); FDs: SO→P, P→S, PO→SO (hinzu kommt nur die dritte wegen Sks)
→ Schema ist in 3NF, da keine transitiven Abhängigkeiten
→ Schema ist nicht in BCNF wegen P→S: P ist kein Sk
➔ weitere Dekomposition in BCNF nicht möglich, da P und S in separate Relationen ausgelagert werden würden und sich somit die Anfrage nicht ohne Join stellen ließe

Übung „Concurrency Control und Recovery“

2.) Hierarchische Sperrverfahren
Datenbank = D, 2 Files = F1 und F2, 1000 Pages pro File = P1…P1000 und P1001…P2000, 100 Records pro Page
Hierarchie: DB – File – Page – Record

a) Lese Record P1200:5:
IS(D), IS(F2), IS(P1200), S(P1200:5)

b) Lese die Seiten P500 bis P520:
IS(D), IS(F1), S(P500), …, S(P520)

c) Lese alle Seiten in F1 und modifiziere 10 Seiten, die erst nach dem Lesen von F1 identifiziert werden können:
IX(D), SIX(F1), IX auf den 10 zu modifizierenden Seiten

d) Lösche Record P1200:98:
IX(D), IX(F2), X(P1200)

e) Lösche den ersten Record aus jeder Seite:
X(D), da wir sowieso auf jede Seite zugreifen müssen

4.) Recovery II:

b) Ausführung bis zum Crash:

LSN	LOG
00	begin_checkpoint
10	end_checkpoint
20	update: T1 writes P5
30	update: T2 writes P3
40	T2 commit
50	T2 end
60	update: T3 writes P3
70	T1 abort
	CRASH, RESTART

Analyse-Phase:
- Beginnt bei letztem Checkpoint, also LSN10
- Schreibe DPT (dirty page table) und TT (transaction table)
- LSN20: (T1,20) in TT, (P5,20) in DPT
- LSN30: (T2,30) in TT, (P3,30) in DPT
- LSN40: Status von T2 von „U“ auf „C“
- LSN50: lösche T2 aus TT
- LSN60: (T3,60) in TT
- LSN70: (T1,20) → (T1,70) in TT

REDO-Phase:
- Startet bei LSN20 (kleinste recLSN in DPT), lese DPT aus
- LSN20: wiederhole Update, da T1 nicht commited
- LSN30: P3 lesen und pageLSN überprüfen: wenn Seite noch vor Crash auf Disk geschrieben wurde, dann keine Aktion, anderenfalls Wiederholung des Updates
- LSN40/50: keine Wiederholung
- LSN 60: wiederhole Update, da T3 nicht commited
- LSN70: keine Aktion

UNDO-Phase:
- Gehe von hinten nach vorn in TT
- LSN70: füge LSN20 zur Verlierer-Menge {60,70} hinzu → neue Menge = {20,70}
- LSN60: Rückgängig-Machen des Updates auf P3, Hinzufügen CLR im Log, neue Verliermenge: {20}
- LSN20: Rückgängig-Machen des Updates auf P5, auf CLR schreiben