Prof. Dr.-Ing. Dr. h. c. T. Härder
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Universität Kaiserslautern
www.haerder.de

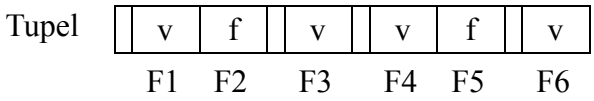
Übungsblatt 9 – Lösungsvorschläge

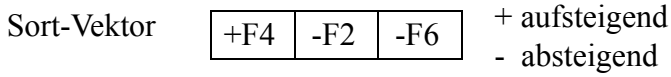
Unterlagen zur Vorlesung: „wwwdvs.informatik.uni-kl.de/courses/DBSREAL/“

Aufgabe 1: Sortieren variabel langer Schlüssel

Beim Sortieren von Tupeln kann ein Sortierschlüssel, der sich auf mehrere Felder in beliebiger Reihenfolge verteilt, spezifiziert werden. Zusätzlich ist die Angabe einer aufsteigenden oder absteigenden Sortierordnung pro Feld erlaubt. Es sind Felder konstanter Länge, für die die Längeninformation in den Beschreibungsdaten abgelegt ist, und Felder variabler Länge mit vorangestellter Längeninformation erlaubt. Die Feldinhalte seien vom Typ „binär“. Feldlänge Null ist im Fall variabler Felder möglich.

Beispiel:





Es gelten folgende Regeln für den Vergleich von Werten:

Für Felder fester Länge entscheidet über die Sortierreihenfolge der binäre Wert bei aufsteigender Sortierordnung. Bei absteigend definierter Sortierordnung dreht sich die Reihenfolge um. Bei Feldern variabler Länge gelten die Regeln des Falls fester Länge, wenn die zu vergleichenden Feldwerte gleiche Länge besitzen. Wenn Wert W1 kürzer als Wert W2 ist und die Länge von m1 Bytes hat, so wird er gegen den m1-Byte Präfix von W2 verglichen. Wenn keine Übereinstimmung auftritt, bestimmt dieser Vergleich die relative Ordnung. Bei Übereinstimmung wird W2 wegen seiner größeren Länge als größer als W1 bei aufsteigender Ordnung (kleiner als W1 bei absteigender Ordnung) eingestuft.

1.) Entwerfen Sie ein Algorithmus, der zwei Tupel mit verteilten Schlüsselfeldern entsprechend der oben gegebenen Vergleichsregeln sortiert.

2.) Geben Sie einen Algorithmus an, der die im Sort-Vektor spezifizierten Felder in einem variabel langen Feld so vercodet, dass jeweils zwei Sortierschlüssel direkt verglichen werden können. Durch die Kodierung muss natürlich die ursprüngliche Sortierfolge gewahrt bleiben. Die einfache Konkatenation der einzelnen Felder löst diese Aufgabe wegen des Auftretens variabel langer Werte nicht. Eine solche Kodierung ist auch wichtig für den Fall eines Mehrfeld-Index.

3.) Entwerfen Sie einen Algorithmus zum Vergleich von zwei kodierten Schlüsseln.

4.) Modifizieren Sie den unter 2.) entworfenen Algorithmus derart, dass das letzte Sort-Feld nicht in die Kodierung einbezogen, sondern nur angehängt wird. Dies ist besonders wichtig im häufig auftretenden Fall eines einzigen Sortierfeldes.

Implementierung der Kodierungsmethode

Die Kodierung einer Folge von einzelnen Feldwerten wird erhalten durch die Konkatenation der kodierten Strings der einzelnen Feldwerte. Jeder zu kodierende Wert der Länge L wird in m = ceil(L/n) Substrings der Länge n unterteilt. Der letzte Substring wird dabei gegebenenfalls durch ein spezielles Zeichen (binäre Nullen) auf die Länge n aufgefüllt, wobei r = m · n – L Auffüllzeichen notwendig werden. Zwischen die einzelnen Substrings wird ein spezielles Fortsetzungszeichen (high value = X'FF') als Kontrollzeichen eingefügt. Das Kontrollzeichen, mit dem der m-te Substring abgeschlossen wird, enthält die Anzahl der gültigen Zeichen dieses Substrings (n-r).

Substring 1 || 'FF'X || Substring 2 || 'FF'X . . . || Substring m || n-r

Beispiel: Mit n = 4 wird der Wert '123456789' eines variablen Feldtyps folgendermaßen zu einem String kodiert (in hexadezimaler Darstellung).

F1F2F3F4 || FF || F5F6F7F8 || FF || F9000000 || 01

Mit dieser Kodierung, bei der jeweils auf den Positionen k · (n + 1), (k = 1, 2, ...), ein Kontrollzeichen steht, wird erreicht, dass beim byteweisen Vergleich zweier kodierter Zeichenfolgen immer nur Kontrollzeichen mit Kontrollzeichen und Datenzeichen mit Datenzeichen verglichen werden. Da das Auffüllzeichen das niederstwertige Zeichen in der Kollationsfolge ist, fällt der Vergleich auf größer bei Strings unterschiedlicher Länge mit gleichem Präfix für den längeren String aus. Soll auf umgekehrte lexikographische Ordnung verglichen werden, so ist der kodierte String durch sein Komplement zu ersetzen (EXOR-Operation mit 'FF'X=high values).
Zur Darstellung dieser Kodiermethode entwickeln wird ein ausführliches Beispiel: Ein Sortierschlüssel oder ein Mehrattribut-Index sei für die Attribute A1, A2 und A3 spezifiziert. Dabei soll für die A1 absteigende und für A2 und A3 aufsteigende Sortierordnung erzielt werden. A1 sei in einem Feld festen Typs der Länge 3 (Byte), A2 und A3 in Feldern variablen Typs gespeichert. Die beiden zusammengesetzten Attributwerte seinen

W_1 =

ABC	123456	A12

W_2 =

ABC	1234	8906A
f	v	v

und ihre hexadezimale Darstellung (in EBCDIC-Code):

W_1 =

C1C2C3	F1F2F3F4F5F6	C1F1F2

W_2 =

C1C2C3	F1F2F3F4	F8F9F0F6C1

A1 hat feste Länge und braucht deshalb nicht kodiert zu werden. Wegen der Forderung nach absteigender Sortierordnung sind die Werte lediglich durch ihr Komplement zu ersetzen. Für A2 und A3 wird die beschriebene Methode zur Kodierung variabel langer Felder angewendet. Alle kodierten Strings werden konkateniert, so dass die resultierenden Strings direkt als binäre Folgen miteinander verglichen werden können. Mit der Substringlänge n = 4 erhalten wir:

W'_1 =

3E3D3C	F1F2F3F4	FF	F5F60000	02	C1F1F200	03

W'_2 =

3E3D3C	F1F2F3F4	04	F8F9F0F6	FF	C1000000	01

Die beiden WerteW'_1 und W'_2 können direkt miteinander verglichen werden, als ob es sich um einfache Attributwerte handeln würde. Da bei aufsteigender Sortierung von variabel langen Feldern bei Präfix-Gleichheit der längere Feldwert gewinnt, entscheidet in diesem Beispiel A2 für W'_1. Es wird durch diese Kodiermethode die gemäß der Vergleichsregeln richtige Reihenfolge $W'_2 < W'_1$ erhalten, obwohl im unkodierten Fall W2 die höherwertige Binärdarstellung (W1 < W2) besitzt.

Bestimmung einer optimalen Substringlänge
Die Länge eines zu kodierenden Feldes ist eine Zufallsveränderliche in einem gewissen Bereich. Zur Bestimmung einer optimalen Länge n der Substrings nehmen wir an, dass alle möglichen Feldlängen mit gleicher Wahrscheinlichkeit auftreten und die mittlere Feldlänge L ist. Wird n zu klein gewählt, entsteht ein hoher zusätzlicher Aufwand an Kontrollzeichen. Mit der Vergrößerung von n sinkt dieser Aufwand; es wächst dagegen mit Mittel der Overhead durch das Ergänzen des letzten Substrings mit Auffüllzeichen.
Der mittlere Gesamtaufwand für Kontrollzeichen und Auffüllzeichen ist deshalb für n zu minimieren. Wegen der Gleichverteilung der Feldlänge nehmen wir an, dass im Mittel die Hälfte des letzten Substrings durch Auffüllzeichen zu ergänzen ist (r=n/2). Als durchschnittlichen Kodierungsaufwand erhalten wir deshalb näherungsweise

$$C(L) = m + \frac{n}{2} \approx \frac{L}{n} + \frac{n}{2} \quad .$$

Wegen

$$\frac{dC(L)}{dn} = -\frac{L}{n^2} + \frac{1}{2} = 0$$

ergibt sich als optimale Länge eines Substrings

$$n = \sqrt{2 \cdot L} \quad .$$

Für eine mittlere Feldlänge von L = 8 Bytes ist folglich n = 4 als Länge des Substrings zu wählen.

Aufgabe 2: Übersetzung von SQL auf eine satzorientierte Schnittstelle 1 69-171

a) Schreiben Sie für folgende SQL-Anfrage ein Programm mit den Operationen auf der satzorientierten Schnittstelle:

```
SELECT    ENO, ENAME, SAL
```

```
FROM    EMP
WHERE   AGE > 45
        AND SAL > 50 000
        AND DNO >= K50 AND DNO <= K60
```

Folgende Grundannahme wird hier für die Verarbeitung über einen Index (Index-Scan) gemacht:

Für jeden Schlüsselwert eines Anfragebereichs werden aus der Indexseite die TIDs der zugehörigen Sätze gelesen, so daß dann unmittelbar auf die Sätze in einer Datenseite zugegriffen werden kann. Somit wird eine Seite für jeden Wert maximal einmal gelesen. Aufgrund der Pufferersetzung ist es jedoch möglich, daß eine Seite verdrängt wird, bevor sie für den Zugriff auf den nächsten Wert erneut gebraucht wird.

Eine solche Verarbeitung hat den Vorteil, daß die Sätze nach dem Indexattribut sortiert ausgegeben werden. Sie hat den Nachteil, daß eine Seite ggf. mehrfach gelesen werden muß. Wenn die Sätze nicht sortiert ausgegeben werden müssen, wäre es günstiger, zunächst alle TIDs zu allen Werten des Anfragebereichs zu lesen und nach ihrer Seitennummer zu sortieren (Merge-Sort ist möglich). Beim darauffolgenden Zugriff wird jede Datenseite maximal einmal gelesen und die Zugriffsbewegungen auf der Magnetplatte werden (bei einer ungestörten Verarbeitung) minimal.

Es soll versucht werden, ein optimales Zugriffsprogramm mit Hilfe der Scan-Technik zu entwerfen, wenn

1. je ein Index für AGE, SAL und DNO vorhanden ist. Welchen Einfluß auf die Optimierung hat die Existenz eines Index mit Cluster-Bildung für eines der drei Attribute?

OPEN SCAN auf EMP
(Index-Scan auf Index mit der größten Selektivität und Nutzung der Startbedingung)

FETCH TUPLE (SCB, Next, ...)
(mit den beiden beiden Prädikatstermen, für die kein Index-Scan ausgewählt wurde, als Suchbedingung (SSA))

CLOSE SCAN
(wenn Stoppbedingung für Index-Scan erreicht; also wenn DNO > K60, falls Index-Scan auf DNO gewählt wurde)

Bei Cluster-Bildung hängt die Wahl der Indexstruktur nicht nur von ihrer Selektivität ab, vielmehr muß jetzt berücksichtigt werden, wie viele Seitenzugriffe notwendig werden. Dies wurde vorher schon implizit gemacht, da in allen drei Fällen keine Cluster-Bildung vorausgesetzt wurde und somit bei niedriger Selektivität auch mehr Seitenzugriffe angenommen wurden.

2. je ein Index für AGE und SAL und zusätzlich ein Index mit Cluster-Bildung für ENO vorhanden ist.

Der Index auf ENO, auf dem keine Selektion erfolgt, könnte in dieser Anfrage nur dann etwas bringen, wenn als Ausgabe ORDER BY ENO verlangt wird. Deshalb bleibt alles wie oben, nur daß der Index auf DNO nicht zur Verfügung steht. (Die Besonderheit der Cluster-Bildung auf den Datenseiten bedeutet, daß die Sätze gemäß ihrer ENO sortiert abgespeichert werden.)

3. für alle Attribute der Qualifikationsbedingung Indexstrukturen ohne Cluster-Bildung vorhanden sind, die Tabelle EMP jedoch alleine in einem Segment mit hoher Lokalität (Belegungsfaktor b > 0.9) gespeichert ist.

Ggf. ist ein Tabellen-Scan (d. h. sequentielles Lesen der Tabelle) günstiger, da keine Indexinformation gelesen werden muß.

OPEN SCAN auf EMP
(Tabellen-Scan mit der Startbedingung BOS)

FETCH TUPLE (SCB, Next, ...)
(mit allen Prädikatstermen als Suchbedingung (SSA))

CLOSE SCAN
(wenn Stoppbedingung EOS erreicht)

b) In welcher Weise kann folgende SQL-Anfrage auf der satzorientierten Schnittstelle ausgewertet werden:

```
SELECT   X.ENAME, Y.ENAME
FROM     EMP X, EMP Y
WHERE    X.MNO = Y.PNO
         AND X.SAL > Y.SAL
```

Diskutieren Sie mögliche Lösungswege, wenn

1. Indexstrukturen für MNO, PNO und SAL vorhanden sind.

2. keine Zugriffspfade zur Unterstützung der Auswertung herangezogen werden können.

Geben Sie die prinzipiellen Lösungen mit Hilfe der auf der satzorientierten Schnittstelle zur Verfügung stehenden Operationen an.

zu 1.

Schritthaltende Index-Scans auf I_{EMP}(PNO) und I_{EMP}(MNO): Wenn Y.PNO=X.MNO erfüllt ist, muß der TID-Zugriff auf jeweils (1+n) Sätze erfolgen, um die Bedingung X.SAL>Y.SAL zu testen. Der Index auf SAL läßt sich dafür nicht einsetzen.

zu 2.

Nested Loops Join auf einer physischen Tabelle mit Y.PNO als Test der inneren Schleife; dann kann der Test im Mittel nach einem halben Schleifendurchlauf abgebrochen werden!

Hier wäre es auch noch interessant, den Fall zu betrachten, daß eine verallgemeinerte Zugriffspfad-

struktur mit X.MNO und Y.PNO existiert oder für die Auswertung dynamisch angelegt wird. Die Auswertung der SAL-Bedingung erzwingt jedoch den Zugriff auf alle X-Sätze (bis auf die, welche die k höchsten Chefs repäsentieren (k >= 1). Bei den Y-Sätzen wird nur auf die zugegriffen, die Manager sind, also typischerweise wesentlich weniger als die Hälfte der EMP-Sätze.

c) In welcher Weise würden Sie folgende SQL-Anfrage, die eine Partitionierung der (Zwischen-)Ergebnismenge nach den Attributwerten eines Attributs erfordert, auswerten:

```
SELECT   DNO
FROM     EMP
WHERE    JOB = 'PROGRAMMER'
GROUP BY DNO
HAVING COUNT (x) > 10
```

Geben Sie ein Programm für die satzorientierte Schnittstelle an, wenn

1. eine Indexstruktur für das Attribut JOB vorhanden ist

2. eine Indexstruktur für das Attribut DNO vorhanden ist

Welches ist die beste Auswertungsstrategie?

zu 1.

- Index-Scan über EMP auf dem Index JOB
- zu jeder DNO zählen, wie oft sie auftritt
- am Ende Ergebnisse ausgeben, welche die HAVING-Bedingung erfüllen

zu 2.

Bringt nichts, wenn keine Clusterung nach DNO vorliegt, weil zu jeder DNO noch alle EMP-Sätze gelesen werden müssen, um die WHERE-Bedingung zu evaluieren, wenn die Anzahl der Werte zu einer DNO größer als 10 ist. Bei diesem Vorgehen werden i. d. R. mehr Seitenzugriffe notwendig, als wenn EMP einmal sequentiell gelesen wird (Tabellen-Scan mit JOB='PROGRAMMER' als Suchbedingung) und intern die Bedingungsauswertung und eine Statistik für die verschiedenen DNOs durchgeführt wird.

Bei Cluster-Bildung über DNO und vielen kleinen Abteilungen können evtl. Seitenzugriffe gespart werden.

Aufgabe 3: Join-Programmierung auf der satzorientierten DBS-Schnittstelle

Skizzieren Sie in Pseudo-Code den Ablauf eines Sort Merge Join sowie den Ablauf eines Hash Join, wenn zwischen den Verbundattributen eine n:m-Beziehung besteht und keine Indexstrukturen zur Verfügung stehen. Ein SORT-Operator ist vorhanden. Die zur Partitionierung herangezogene Hash-Funktion bilde die Werte von ORT und WOHNORT direkt auf die Nummer der Partition ab.

```
SELECT  P.PNR, P.WOHNORT, P.ANR
FROM    ABT A, PERS P
WHERE   P.WOHNORT = A.ORT AND
        P.ALTER > 30 AND
        A.ANZ_MITARBEITER < 10
```

Lösungsvorschlag:

Sort-Merge-Join

```
Open Scan (ABT, BOS, EOS);                                    /*
SCB1 */
Sort ABT On ORT asc Into T1 Using Scan(SCB1, Next,
ANZ_MITARBEITER < 10);
Close Scan (SCB1);
Open Scan (PERS, BOS, EOS);                                   /*
SCB2 */
Sort PERS On WOHNORT asc Into T2 Using Scan (SCB2, Next, ALTER >
30);
Close Scan (SCB2);

Open Scan (T1, BOS, EOS);                                      /*
SCB3 */
Open Scan (T2, BOS, EOS);                                      /*
SCB4 */
Fetch Tuple (SCB3, Next);
Fetch Tuple (SCB4, Next);
Repeat
    While T2.WOHNORT < T1.ORT
        Fetch Tuple (SCB4, Next);
    Endwhile;
    While T2.WOHNORT > T1.ORT
        Fetch Tuple (SCB3, Next);
    Endwhile;
    Temp = T1;
    Array1 = Array2 = EMPTY;
    While T1.ORT = Temp.ORT
        Store T1 in Array1;
        Fetch Tuple (SCB3, Next);
    Endwhile;
    While T2.WOHNORT = Temp.ORT
        Store T2 in Array2;
```

```
        Fetch Tuple (SCB4, Next);
    Endwhile;
    Array1 X Array2     /*Transfer nach Ausgabebereich erforder-
lich*/
Until End of T1 or T2;
```

Hash Join

max : # Partionen
h : ORT --> {1, ..., max};
h : WOHNORT --> {1, ..., max};
Ai[], Pi[]: Pointer-Arrays für die i-ten Partitionen (liegen im Hauptspeicher), i = 1 ... max
pa, pp : Hash-Partitionen für ABT, PERS
ta, tp : Tupel aus ABT, PERS

initialisiere Ai und Pi
/* Partitionierung der beiden Tabellen mit Hash-Funktion h */

```
While (t = next_tuple in ABT mit t.ANZ_MITARBEITER < 10)
    i := h(t);
    Ai.add(address of t);
Endwhile;
While (t = next_tuple in PERS mit t.ALTER > 30)
    i := h(t);
    Pi.add(address of t);
Endwhile;
For i := 1 to max do
    pa := NULL;
    pp := NULL;
    for each pointer pt in Ai do
        pp.add(tuple, auf das pt zeigt);
    endfor;
    for each pointer pt in Pi do
        pp.add(tuple, auf das pt zeigt);
    endfor;
```

/* Hier ist vereinfachend ein Nested Loops Join pro Partition skizziert *

```
    for each tuple tp in pp
        for each tuple ta in pa
            if pp.wohnort = pa.ort
            join (pa, pp);
            endif;
        endfor;
```

endfor;
endfor;

Aufgabe 4: Kostenmodelle für die Selektionsoperation

Gegeben sei Tabelle R(A1, A2, A3, ..., An) mit zusammenhängender Speicherung der Sätze in Seiten des Segmentes S.

Annahmen:

- Segment S habe $M_S = 10^4$ Seiten
- Tabelle R habe
  - $N_R = 10^5$ Sätze und ggf.
  - Cluster-Faktor $c_R = 50$
- Indexe
  - $I_R(A1)$ mit $j_{A1} = 100$
  - $I_R(A2)$ mit Cluster-Bildung und $j_{A2} = 10$
  - als B*-Bäume mit Höhe $h_B = 2$ und $N_B = 100$ Blattseiten realisiert

a) Wie teuer ist die Auswertung der SQL-Anfrage

```
Select   *
From     R
Where    A3 = 'x'
```

1. bei einem Tabellen-Scan
   $C_1 = M_s = 10^4$ (Seiten)

2. bei Nutzung von $I_R(A1)$
   $C_2 = h_B + N_B + N_R = 2 + 100 + 10^5$ (Seiten)

3. bei Nutzung von $I_R(A2)$
   $C_3 = h_B + N_B + N_R / c_R = 2 + 100 + 2 \cdot 10^3$ (Seiten)

4. Wie teuer ist die Auswertung der obigen Anfrage, wenn die Tabelle als Hash-Struktur mit A3 als Primärschlüssel angelegt ist?
   $C_4 = 1$

b) A1 habe 100 Werte, die von 1 bis 100 gleichverteilt vorkommen ($j_{A1} = 100$)
Wie teuer ist die Auswertung der SQL-Anfrage

```
Select   *
From     R
Where    A1 > 50
```

5. bei Index-Nutzung
   $C_5 = h_B + N_B/2 + (N_R/j_{A1}) \cdot j_{A1}/2 = 2 + 50 + 10^5/2$

6. bei Annahme einer Cluster-Bildung bei $I_R(A1)$
   $C_6 = h_B + N_B/2 + (N_R/j_{A1}) \cdot j_{A1}/(2 \cdot c_R)$

   $= 2 + 50 + 10^3$

7. ohne Indexnutzung
   $C_7 = M_s = 10^4$ (aber sequentieller Zugriff!)

c) Welche Kosten verursacht die SQL-Anfrage?

```
Select   *
From     R
Where    A1 = 50   AND   A2 = 10
```

8. bei Nutzung von $I_R(A1)$ und $I_R(A2)$ (ohne Cluster-Bildung)
   $C_8 = h_B + N_B/j_{A1} + h_B + N_B/j_{A2} + N_R/(j_{A1} \cdot j_{A2}) = 115$

9. bei zusätzlicher Annahme einer Cluster-Bildung bei $I_R(A2)$ und Zugriff über beide Indexe
   $C_9 = h_B + N_B/j_{A1} + h_B + N_B/j_{A2} + N_R/(j_{A1} \cdot j_{A2} \cdot c_R)$
   $= 2 + 1 + 2 + 10 + 10^5/(5 \cdot 10^4) = 17$

10. bei Zugriff nur über $I_R(A2)$ mit Cluster-Bildung
    $C_{10} = h_B + N_B/j_{A2} + N_R/(j_{A2} \cdot c_R) = 12 + 10^5\ (5 \cdot 10^2) = 212$

11. bei Zugriff nur über $I_R(A1)$ mit Cluster-Bildung
    $C_{11} = h_B + N_B/j_{A1} + N_R/(j_{A1} \cdot c_R) = 3 + 20 = 23$

12. bei k-d-Scan auf Grid-File für A1, A2
    Es sind $q = N_R/(j_{A1} \cdot j_{A2}) = 100$ Treffer zu erwarten, wobei mindestens 2 Buckets gefüllt werden. Annahme: Die Treffer verteilen sich auf 4 benachbarte Buckets:
    $C_{12} = 1 + 4 = 5$